A closer look at TDFA

Angelo Borsotti
angelo.borsotti@mail.polimi.it

Ulya Trafimovich
skvadrik@gmail.com

2022

Abstract

We present an algorithm for regular expression parsing and submatch extraction based on tagged deterministic finite automata. The algorithm works with different disambiguation policies. We give detailed pseudocode for the algorithm, covering important practical optimizations. All transformations from a regular expression to an optimized automaton are explained on a step-by-step example. We consider both ahead-of-time and just-in-time determinization and describe variants of the algorithm suited to each setting. We provide benchmarks showing that the algorithm 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 describes tagged deterministic finite automata (TDFA). To the best of our knowledge, it is the first *practical* submatch extraction and parsing algorithm based on deterministic finite automata that is capable of both POSIX and leftmost greedy disambiguation. Most of the theory behind TDFA is not new, but the previous papers are incomplete and lack important details. This paper consolidates our previous research and provides a comprehensive description of the algorithm. We hope that it will make TDFA easier to implement in practice.

Here is a brief history of TDFA development. In 2000 Laurikari published the original paper [1]. In 2007 Kuklewicz implemented TDFA in a Haskell library with POSIX longest-match disambiguation; he gave only an informal description [2]. In 2016 Trafimovich presented TDFA with lookahead [3], implemented them in the open-source lexer generator RE2C [4] and formalized Kuklewicz disambiguation algorithm. In 2017 Borsotti implemented TDFA in an experimental Java library [5]. In 2019 Borsotti and Trafimovich adapted Okui-Suzuki disambiguation algorithm to TDFA and showed it to be faster than Kuklewicz algorithm [6]. In 2020 Trafimovich published an article about TDFA implementation in RE2C [7]. Finally, the present paper incorporates our past research and adds novel findings on multi-pass TDFA that are better suited to just-in-time determinization.

Before diving into details, we recall the key concepts discussed in the paper.

Regular expressions (REs) are a notation for describing sets of strings known as regular languages, or Type-3 languages in the Chomsky hierarchy. They were first defined by Kleene [9] as sets of strings constructed from the alphabet symbols and the empty word via the application of three basic operations: concatenation, alternative and iteration. Later REs were formalized via the notion of Kleene algebra [10]. In practice REs have many extensions that vary in complexity and expressive power.

Submatch extraction is an extension of REs that allows one to identify input positions matching specific positions in a RE. Recall the difference between *recognition* and *parsing*: to recognize a string means to determine its membership in a language, but to parse a string means to also find its derivation in a language grammar. Submatch extraction is in between: on one extreme it approaches recognition (if there are no submatch markers in a RE), but on the other extreme it is identical to parsing (if every position in a RE is marked). In general it requires constructing a partial derivation, which can be implemented more efficiently than full parsing.

Finite state automata are a formalism equivalent to REs in the sense that every RE can be converted to a deterministic finite automaton (DFA) or a nondeterministic finite automaton (NFA), and vice versa. There are many different kinds of NFA, but there is a unique minimal DFA for a given RE. Both NFA and DFA solve the recognition problem for REs in linear time in the length of input. In practice DFA are faster because they follow a single path, while NFA have to track multiple paths simultaneously. NFA can be converted to DFA using a *determinization* procedure, but the resulting DFA may be exponentially larger than the NFA.

Tags are submatch markers: they mark positions in a RE that should be mapped to offsets in the input string. When a RE is converted to an NFA, tags are placed on the NFA transitions. This effectively turns NFA into a nondeterministic finite-state *transducer* that rewrites symbolic strings into tagged strings (where tags are placed in-between symbols, marking submatch boundaries). Conversion from a RE to a tagged NFA is natural if NFA mirrors the structure of RE, as in the case of Thompson's construction.

Determinization of a tagged NFA is problematic, because in a DFA multiple NFA paths are collapsed into one, causing conflicts when the same tag has different values on different NFA paths. To keep track of all possible tag values, a DFA is augmented with *registers* and *operations* on transitions that update register values. The number of registers and operations depends only on the RE structure and tag density, but not on the input string, therefore it adds only a constant overhead to the DFA execution. We describe techniques that reduce redundant operations and minimize the overhead in practice.

Ambiguity is yet another problem for submatch extraction; it means the existence of multiple different parse trees for the same input. Ambiguity should not be confused with non-determinism, which means the existence of multiple possibilities during matching that get canceled as more input is consumed; ambiguity is a property of a RE. One way to resolve it is a *disambiguation policy*, the most notable examples being the leftmost-greedy and the longest-match (POSIX) policies. TDFA can work with both policies, and there is no runtime overhead on disambiguation — it is built into TDFA structure. 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.

RE engines based on DFA can be divided in two groups: those using *ahead-of-time* (AOT) determinization (e.g. lexer generators) and those using *just-in-time* (JIT) determinization (e.g. runtime libraries). The former can spend considerable amount of time on preprocessing, but the latter face a tradeoff between the time spent on preprocessing and the time spent on matching. Therefore it makes sense to use different variants of the algorithm in each case. We describe *single-pass* TDFA that are a natural fit for ahead-of-time determinization, and *multi-pass* TDFA that are better suited to just-in-time determinization.

In practice performance of a matching algorithm depends on the representation of results. The most generic representation is a parse tree; it precisely reflects a derivation. A more lightweight representation is a list of offsets or a single offset per submatch position in a RE (the latter is used in the POSIX `regexec` function). Another representation, more suitable for transducers, is a *tagged string* — a sequence of input symbols interspersed with tags. If a RE contains tags for every subexpression, then it is possible to reconstruct a parse tree from offset lists or a tagged string (a procedure is given in [6], section 6). TDFA can be used with all the above representations, but it is more natural to use offsets with single-pass TDFA and tagged strings with multi-pass TDFA.

The rest of the paper is structured as follows. Section 1 defines REs and their conversion to nondeterministic automata. Section 2 defines TDFA and determinization. Section 3 describes optimizations and practical implementation details. Section 4 describes multi-pass TDFA and their application to just-in-time determinization. Section 5 provides benchmarks and comparison with other algorithms. Section 6 contains conclusions, and section 7 contains ideas for future work.

Conventions

In this paper we use *pseudocode* rather than formal mathematical notation to describe algorithms. We focus on the practical side, because we want to encourage TDFA implementation in real-world programs. The most theoretically challenging part of the algorithm (POSIX disambiguation) is formalized in our previous paper [6], and the core of the algorithm is based on the well-known idea of determinization via powerset construction that does not need a formal introduction.

In the pseudocode, we try to balance between formality and clarity. We omit the definitions of basic operations on data structures, such as "append to a list" or "push on stack". We sometimes use set notation with predicates, and sometimes prefer explicit loops that iterate over the elements of a set. To reduce verbosity, we assume that function arguments are passed by reference and modifications to them are visible in the calling function (although some functions have explicit return values).

All algorithms presented here are implemented in the open-source lexer generator RE2C and are known to work in practice.

1 TNFA

In this section we define regular expressions, their conversion to nondeterministic automata and matching.

Definition 1. *Regular expressions (REs) 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 REs over Σ.*

Tags mark submatch positions in REs. Unlike capturing parentheses, tags are not necessarily paired. Capturing groups can be represented with tags, but the correspondence may be more complex than a pair of tags per group, e.g. POSIX capturing groups require additional hierarchical tags [6].

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 usually desugared via concatenation, but we avoid desugaring as it may duplicate tags and change submatch semantics in a 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 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 $\Sigma \cup T \cup \overline{T}$. $\overline{T} = \{-t \mid t \in T\}$ 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 a RE, such as alternative or repetition with zero lower bound. Negative tags serve a few purposes: they prevent propagation of stale submatch values from one iteration to another, they spares the need to initialize tags, and they are needed for POSIX disambiguation [6]. 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 a 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((Σ, T, Q, q0, qf, Δ), a1 ... an)
1  m0 : vector of offsets of size |T|
2  C = {(q0, m0)}
3  for k = 1, n do
4      C = epsilon_closure(C, Δ, qf, k)
5      C = step_on_symbol(C, Δ, ak)
6      if C = ∅ then return ∅
7  C = epsilon_closure(C, Δ, qf, n)
8  if ∃(q, m) in C | q = qf then return m
9  else return ∅

step_on_symbol(C, Δ, a)
10 return {(p, m) | (q, m) in C and (q, a, p) ∈ Δ}

epsilon_closure(C, Δ, qf, k)
11 C' : empty sequence of configurations
12 for (q, m) in C in reverse order do
13     push (q, m) on stack
14 while stack is not empty do
15     pop (q, m) from stack
16     append (q, m) to C'
17     for each (q, i, t, p) ∈ Δ ordered by priority i do
18         if t > 0 then m[t] = k
19         else m[−t] = n
20         if configuration with state p is not in C' then
21             push (p, m) on stack
22 return {(q, m) in C' | q = qf or
23                        ∃(q, a, _) ∈ Δ where a ∈ Σ}
```

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 [6]. Figure 1 in section 2 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 $(\Sigma, T_1 \cup T_2, Q_1 \cup Q_2, q_1, q_f, \Delta_1 \cup \Delta_2)$

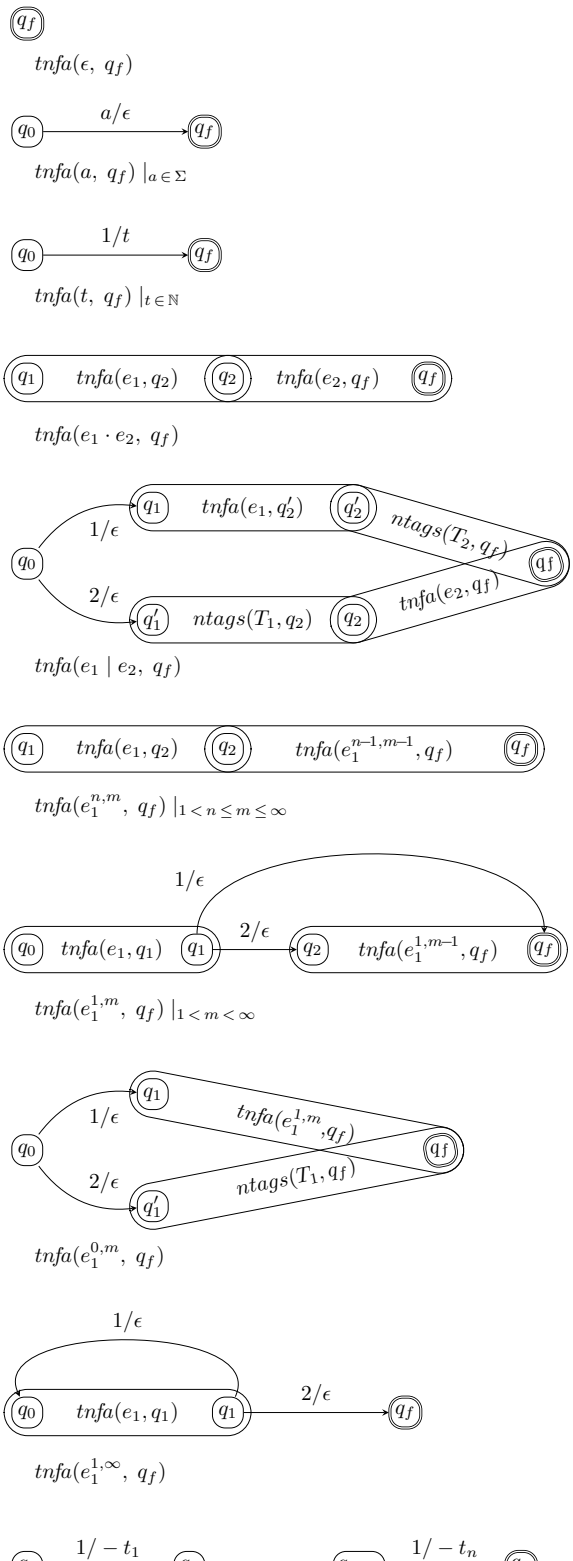
else if $e = e_1^{1,m} \mid_{1<m<\infty}$ **then**
  if $m = 1$ **then return** $tnfa(e_1, q_f)$
  $(\Sigma, T_1, Q_1, q_1, q_f, \Delta_1) = tnfa(e_1^{1,m-1}, q_f)$
  $(\Sigma, T_2, Q_2, q_0, q_2, \Delta_2) = tnfa(e_1, q_1)$
  $\Delta = \Delta_1 \cup \Delta_2 \cup \{(q_1, 1, \epsilon, q_f), (q_1, 2, \epsilon, q_2)\}$
  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 *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 *is a finite set of registers with a subset 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, but does not have to be based on whether the tag is under repetition). Register values are denoted by special symbols $\mathbf{n}$ and $\mathbf{p}$, 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 [8], 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, a 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 them to the outgoing transitions. This allows filtering tags by the lookahead symbol: configurations 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 distinguishes TDFA(1) from TDFA(0) [3]; 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 scoped to functions *precedence*, *step_on_symbol* and *epsilon_closure*, 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 an order on configurations, represented by *precedence* as a vector of TNFA states: *step_on_symbol* uses it to construct the initial closure, and

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 $R_f[t] \leftarrow regop\_rhs(r, l_t, t)$ 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 $(\Sigma, T, Q, q_0, q_f, \Delta)$.

epsilon_closure performs depth-first search following transitions from left to right. POSIX policy is more complex; we do not include pseudocode for it here, but it is extensively covered in [6].

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 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). 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* which does not consume input characters and gets executed only once 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 made identical (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 12); 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 and inhibits 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 with no internal structure. Disambiguation decisions are embedded in TDFA; there is no disambiguation at runtime. The only runtime overhead compared to an ordinary DFA is the execution of register operations on transitions. A TDFA may have more states than a DFA for the same RE with all tags removed, 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.

Figure 1 shows an example of TDFA construction:

- ➔ The RE is $(1a2)^*3(a|4b)5b^*$. It defines language $\{a^n b^m \mid n+m>0\}$ and has five tags t_1, t_2, t_3, t_4, t_5.
- ➔ TNFA has three kinds of transitions: bold transitions on alphabet symbols (four of them for each symbol in the RE), thin ϵ-transitions with priority and dashed ϵ-transitions with priority and tag. Tags t_1, t_2 are under repetition, so the zero-repetition path $0 \to 5 \to 6 \to 7$ contains transitions with negative tags $-t_1$, $-t_2$. Likewise tag t_4 is in alternative, so path $8 \to 9 \to 10 \to 13$ contains transition with negative tag $-t_4$. Tags t_3, t_5 are in top-level concatenation and do not need negative tags.
- ➔ TNFA simulation on string *aab* consists of four steps. The first step starts with state 0. Every other step starts with states from the previous step and follows transitions labeled with the current symbol. Each step constructs ϵ-closure by following ϵ-paths and collecting tag sequence along the way. The value of positive tags in the corresponding row of the closure matrix is set to the number of characters consumed so far. The value of negative tags is set to nil **n**. The value of tags not in the sequence is inherited from the previous closure. Simulation ends when all characters have been consumed. Since the last closure contains a row with the final state 17, it is a match and the final tag values are $1, 2, 2, 2, 3$.

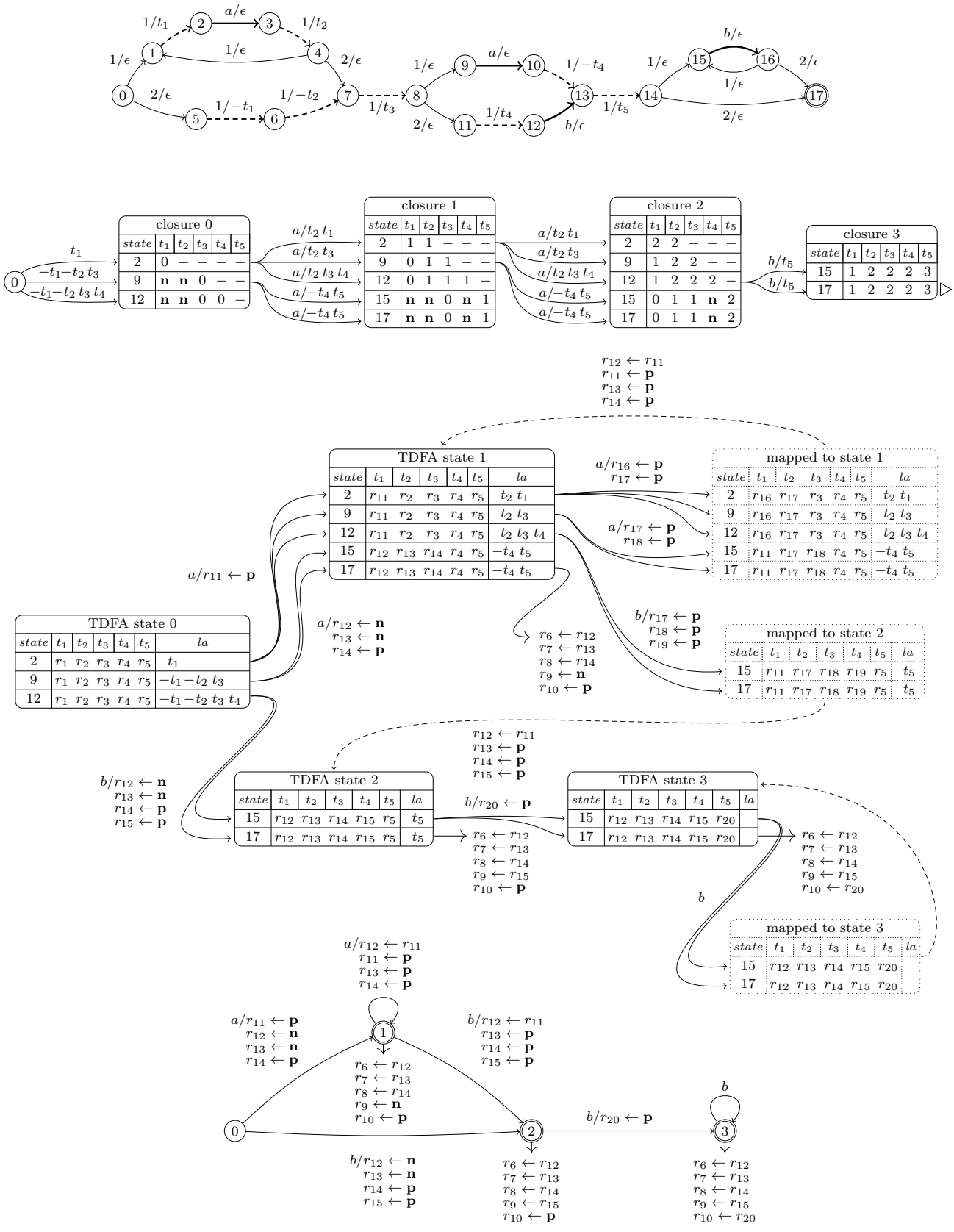


Figure 1: Example for a RE $(1a2)^*3(a|4b)5b^*$: TNFA, simulation on string aab, determinization, TDFA.

- ➔ The match is ambiguous: it is possible to match *aab* by following path $0 \rightsquigarrow 2 \rightsquigarrow 2 \rightsquigarrow 12 \rightsquigarrow 17$ (let the greedy repetition consume *aa*) but it is also possible to follow path $0 \rightsquigarrow 2 \rightsquigarrow 9 \rightsquigarrow 15 \rightsquigarrow 17$ (let the greedy repetition consume only the first *a*). The first match is preferable by both POSIX and leftmost-greedy policies.
- ➔ Determinization is similar to simulation, but TDFA states store registers instead of offsets. This solves the problem of mapping states that differ only in tag values: for example, closures 1 and 2 cannot be mapped, although they have identical states and tag sequences, but TDFA state corresponding to closure 2 is mapped to state 1. This is possible due to the register operations on the dashed backward transition. Note that there is one copy operation $r_{12} \leftarrow r_{11}$, but other copy operations for r_{11}, r_{13}, r_{14} are combined with set operations, e.g. $r_{11} \leftarrow \mathbf{p}$ is the combination of $r_{16} \leftarrow \mathbf{p}$ and $r_{11} \leftarrow r_{16}$ (see lines 43 – 45 of algorithm 3).
- ➔ Unlike simulation, determinization does not immediately apply tag sequences to registers. Instead, it stores them as part of TDFA state (in the *lookahead* column, shortened as *la*). Compare tag sequences on transitions to closures 0, 1, 2 to that in states 0, 1, 2 respectively — these are the same tags. Lookahead tags form register operations on the outgoing transitions: e.g. lookahead tag t_1 in the first row of TDFA state 0 (corresponding to tagged TNFA transition $1 \rightarrow 2$) forms operation $r_{11} \leftarrow \mathbf{p}$.
- ➔ For every distinct set or append operation *transition_regops* allocates a new register and stores the updated tag value in it. Note that it would be impossible to reuse the same register (e.g. to have $r_1 \leftarrow \mathbf{p}$ instead of $r_{11} \leftarrow \mathbf{p}$ on transition from state 0 to 1) because there may be conflicting operations (e.g. $r_{12} \leftarrow \mathbf{n}$ for lookahead tag $-t_1$). Therefore tag t_1 in TDFA state 1 is represented with two registers r_{11} and r_{12}, reflecting the fact that state 1 may be reached by different TNFA paths with conflicting submatch values.
- ➔ Final TDFA states are all states containing TNFA state 17 (i.e. states 1, 2 and 3). In addition to normal transitions final TDFA states have quasi-transitions that set final registers $r_6 - r_{10}$. These quasi-transitions do not consume any symbol, and the operations on them are executed once at the end of match.
- ➔ In the resulting TDFA all internal structure in the states is erased, leaving atomic states with transitions and register operations. Registers can be renamed to occupy consecutive numbers, and the number of registers and operations can be reduced (see section 3).

3 Implementation

In this section we describe optimizations and practical details that should be taken into account when implementing TDFA. None of the optimizations is particularly complex or vital for TDFA operation, but applied together and in the correct order they can make TDFA considerably faster and smaller.

3.1 Multi-valued tags

The most straightforward representation of multi-valued tags is a vector of offsets. It is very inefficient because copy and append operations need to copy entire vectors (which could grow arbitrarily long). A more efficient representation is a *prefix tree*. It is possible because tag sequences in the operations map on the path tree constructed by the ϵ-closure. The tree can be stored as an array of nodes $(pred, offs)$ where *pred* is the index of a predecessor node, and *offs* is a positive or negative tag value. Individual sequences in the tree are addressed by integer indices of tree nodes (zero index corresponds to the empty sequence). This representation is space efficient (common prefixes are shared), but most importantly it makes copy operations as simple as copying scalar values (tree indices). Append operations are more difficult, as they require a new slot (or a couple of slots) in the prefix tree. However, if the backing array is allocated in large chunks of memory, then the amortized complexity of each operation is constant. This representation was used by multiple researches, e.g. Karper describes it as the *flyweight pattern* [12].

3.2 Fallback operations

In practice it is often necessary to match the longest possible prefix of a string rather than the whole string. After matching a short prefix, TDFA may attempt to match a longer prefix. If that fails, it must fallback to the previous final state and restore the input position accordingly. A final state is also a *fallback state* if there are non-accepting paths out of it, and a path is non-accepting if does not go through another final state (which may happen either because the input characters do not match or due to a premature end of input).

For an ordinary DFA the only information that should be saved in a fallback state is the input position. For TDFA it is also necessary to backup registers that may be clobbered on the non-accepting paths from the fallback state. Backup operations should be added on transitions out of the fallback state, and restore operations should be added on the fallback quasi-transition, which replaces the final quasi-transition for fallback paths. Final registers can be reused for backups, as by TDFA construction they are used only on the final quasi-transitions. Backup registers are only needed for copy and append operations (set operations do not depend on registers).

```
fallback_regops()
1  ψ : undefined fallback function
2  for each fallback state s ∈ S do
3      O : empty list of register operations
4      for each operation on quasi-transition φ(s) do
5          if append i ← j · h and j is clobbered then
6              backup_regops(s, i, j)
7              append operation i ← i · h to O
8          else if copy i ← j and j is clobbered then
9              backup_regops(s, i, j)
10         else
11             append a copy of this operation to O
12     ψ(s) = O
13 return ψ
```

```
backup_regops(s, i, j)
14 for each alphabet symbol a ∈ Σ do
15     (s′, O) = δ(s, a)
16     if exist non-accepting paths from s′ then
17         append copy operation i ← j to O
```

Algorithm 4: Adding fallback operations to TDFA $(\Sigma, T, S, S_f, s_0, R, R_f, \delta, \varphi)$.

Algorithm 4 shows how to add such operations. It assumes that fallback states and clobbered registers for each fallback state have already been identified. This can be done as follows. First, augment TDFA with a *default state* that makes transition function δ total (if a premature end of input is possible, add a quasi-transition from non-final states to the default state). Then compute reachability of the default state by doing backward propagation from states that have transitions to it. If the default state is reachable from a final state, then it is a fallback state. Clobbered registers can be found by doing depth-first search from a fallback state, visiting states from which the default state is reachable, and accumulating left-hand-sides of register operations.

3.3 Register optimizations

TDFA induces a *control flow graph* (CFG) with three kinds of nodes:

- *basic blocks* for register operations on symbolic transitions
- *final blocks* for final register operations
- *fallback blocks* for fallback register operations

There is an arc between two blocks in CFG if one is reachable from another in TDFA without passing through other register operations. Additionally, fallback blocks have arcs to all blocks reachable by TDFA paths that may fall through to these blocks. Figure 2 shows CFG for the example from section 2.

CFG represents a program on registers, so the usual compiler optimizations can be applied to it, resulting in significant reduction of registers and operations. RE2C uses the following optimization passes for the number of repetitions $N = 2$ (pseudocode is given by the algorithms 5 and 6):

1. Compaction
2. Repeat N times:
   a. Liveness analysis
   b. Dead code elimination
   c. Interference analysis
   d. Register allocation with copy coalescing
   e. Local normalization

Compaction pass is applied only once immediately after determinization. It renames registers so that they occupy contiguous range of numbers with no "holes". This is needed primarily to allow other optimization passes use registers as indices in liveness and interference matrices.

optimizations(G)

```
1  V = compaction(G)
2  G = renaming(G, V)
3  for i = 1, 2 do
4      L = liveness_analysis(G)
5      dead_code_elimination(G, L)
6      I = interference_analysis(G, L)
7      V = register_allocation(G, I)
8      renaming(G, V)
9      normalization(G)
```

renaming(G, V)

```
10 for each block b in G do
11     for each operation in b do
12         if set operation i ← v then
13             rename i to V[i]
14         if copy or append operation i ← j... then
15             rename i to V[i] and j to V[j]
```

liveness_analysis(G)

```
16 L : boolean matrix indexed by blocks and registers
17 for each block b in G do
18     for each register i in G do
19         L[b][i] = false
20 for each final block b in G do
21     for each final register i in G do
22         L[b][i] = true
23 while true do
24     fix = true
25     for each basic block b in G in post-order do
26         L_b = copy of row L[b]
27         for each successor s of block b do
28             L_s = copy of row L[s]
29             for each operation in s in post-order do
30                 if set operation i ← v then
31                     L_s[i] = false
32                 if copy operation i ← j then
33                     if L_s[i] then
34                         L_s[i] = false
35                         L_s[j] = true
36             for each register i in G do
37                 L_b[i] = L_b[i] ∨ L_s[i]
38         if L[b] ≠ L_b then
39             L[b] = L_b
40             fix = false
41     if fix then break
42 for each fallback block b in G do
43     for each final register i in G do
44         L[b][i] = true
45     L_b = copy of row L[b]
46     for each operation i ← _ in b do
47         L_b[i] = false
48     for each copy or append operation _ ← j... in b do
49         L_b[j] = true
50     for each block s in G that may fall through to b do
51         for each register i in G do
52             L[s][i] = L[s][i] or L_b[i]
53 return L
```

compaction(G)

```
54 U : boolean vector indexed by registers
55 V : integer vector indexed by registers
56 for each register i in G do
57     U[i] = false
58 for each block b in G do
59     for each operation in b do
60         if set operation i ← v then
61             U[i] = true
62         if copy or append operation i ← j... then
63             U[i] = U[j] = true
64 n = 0
65 for registers i in G such that U[i] do
66     n = n + 1, V[i] = n
67 return V
```

dead_code_elimination(G, L)

```
68 for each basic block b in G do
69     L_b = copy of row L[b]
70     for each operation i ← _ in b in post-order do
71         if L_b[i] then
72             if set operation i ← v then
73                 L_b[i] = false
74             if copy operation i ← j then
75                 L_b[i] = false
76                 L_b[j] = true
77         else remove dead operation
```

interference_analysis(G, L)

```
78 I : boolean matrix indexed by registers
79 V : vector of histories indexed by registers
80 for each register i in G do
81     for each register j in G do
82         I[i][j] = I[j][i] = false
83 for each block b in G do
84     for each copy or append operation i ← j... in b do
85         V[j] = j
86     for each operation in b do
87         I_b = copy of row L[b]
88         if set operation i ← v then
89             V[i] = v
90             I_b[i] = false
91         else if copy operation i ← j then
92             V[i] = V[j]
93             I_b[i] = I_b[j] = false
94         else if append operation i ← j · h then
95             V[i] = V[j] · h
96         for operations k ← _ in b with V[k] = V[i] do
97             I_b[k] = false
98         for registers k in G such that I_b[k] do
99             I[i][k] = I[k][i] = true
100 for registers i in G not used in append operations do
101     for registers j in G used in append operations do
102         I[i][j] = I[j][i] = true
103 return I
```

Algorithm 5: Register optimizations (part 1).

register_allocation(G, I)

```
1  V : vector of registers indexed by registers
2  B : vector of registers indexed by registers
3  S : vector of register sets indexed by registers
4  for each register i in G do
5      B[i] = −1
6      S[i] = ∅
7  for each block b in G do
8      for each operation in b do
9          if copy or append i ← j... and i ≠ j then
10             x = B[i], y = B[j]
11             if x = −1 and y = −1 then
12                 B[i] = B[j] = i
13                 S[i] = {i, j}
14             else if x ≠ −1 and y = −1 then
15                 if ∀k ∈ S[x] : ¬I[k][j] then
16                     B[j] = x
17                     S[x] = S[x] ∪ {j}
18             else if x = −1 and y ≠ −1 then
19                 if ∀k ∈ S[y] : ¬I[k][i] then
20                     B[i] = y
21                     S[y] = S[y] ∪ {i}
22 for registers i in G such that B[i] = i do
23     for registers j in G such that B[j] = j and j > i do
24         if ∀i ∈ S[x], j ∈ S[y] : ¬I[i][j] then
25             B[y] = x
26             S[x] = S[x] ∪ S[y]
27             S[y] = ∅
28 for registers i in G such that B[i] = −1 do
29     if ∃j in G : B[j] = j and ∀k ∈ S[j] : ¬I[i][k] then
30         B[i] = j
31         S[j] = S[j] ∪ {i}
32 n = 0
33 for registers i in G such that B[i] = i do
34     n = n + 1
35     for registers j ∈ S[i] do
36         V[j] = n
37 return V
```

normalization(G)

```
38 for each block b in G do
39     for each contiguous set operation range O do
40         remove_duplicates(O)
41         sort(O)
42     for each contiguous copy operation range O do
43         remove_duplicates(O)
44         topological_sort(O)
45     for each contiguous append operation range O do
46         remove_duplicates(O)
```

topological_sort(O)

```
47 I : vector of in-degree indexed by registers
48 for each copy or append operation i ← j... in O do
49     I[i] = I[j] = 0
50 for each copy or append operation _ ← j... in O do
51     I[j] = I[j] + 1
52 O′ : empty list of operations
53 nontrivial_cycle = false
54 while O is not empty do
55     for each operation i ← _ in O do
56         if I[i] = 0 then
57             remove operation from O, append to O′
58             if this is a copy/append operation i ← j... then
59                 I[j] = I[j] − 1
60     if nothing added to O′ but O is not empty then
61         if ∃ operation i ← j... in O with i ≠ j then
62             nontrivial_cycle = true
63         append O to O′
64         break (only cycles left)
65 O = O′
66 return ⌀nontrivial_cycle
```

remove_duplicates(O)

```
67 for each operation o in O do
68     for each subsequent operation o′ = o in O do
69         remove duplicate operation o′
```

Algorithm 6: Register optimizations (part 2).

Liveness analysis builds a boolean 2-dimensional matrix indexed by CFG blocks and registers. A cell $L[b][i]$ is true iff register i is alive in block b (meaning that its value is used). Function *liveness_analysis* uses iterative data-flow approach. Initially only the final registers in the final blocks are alive. The algorithm iterates over CFG blocks in post-order, expanding the live set, until it reaches a fix point. Lastly it marks backup registers as alive in all blocks reachable from fallback blocks by non-accepting paths.

Dead code elimination removes operations whose left-hand-side register is not alive.

Interference analysis builds a boolean square matrix indexed by registers. A cell $I[i][j]$ is true iff registers i and j interfere with each other (their lifetimes overlap, so they cannot be represented with one register). Initially none of the registers interfere. Function *interference_analysis* considers each CFG block b and inspects each register j used on the right-hand-side of an operation: all registers alive in block b interfere with j, except for registers that have the same value (tracked by the vector V). Finally registers for multi-valued and single-valued tags are marked as interfering with each other.

Register allocation partitions registers into equivalence classes. Registers inside of one class do not interfere with each other, so they can all be replaced with a single *representative*. Initially none of the registers belongs to any class. Function *register_allocation* loops over copy operations and tries to put source and destination into one class (so that the copy can be removed). Vector B maps registers to their representative, and vector S

maps representatives to their class. The algorithm tries to merge non-interfering equivalence classes, and then puts the remaining registers into an non-interfering class (allocating a new class if necessary). Finally it maps representatives to consecutive numbers and stores them in the V vector. The constructed partitioning is not minimal, but it's a good approximation, since finding the minimal clique cover of a graph is NP-complete.

Local normalization reconciles operations after previous passes. The *normalization* function removes duplicate operations that might appear after different registers are collapsed into one. It also sorts operations, so that operation sequences could be compared easily (which is used in further optimizations like in minimization). Each continuous range of set, copy or append operations is handled separately, because operations of different kinds should not be reordered (that could change the end result).

Figure 2 is a continuation of example on figure 1:

- ➔ The CFG contains 9 blocks: basic blocks 0 – 5 (one for each tagged transition on symbol in TDFA, plus the start block 0) and final blocks 6 – 8 (one per each final quasi-transition). There are no fallback blocks in this example, because there are no fallback states in TDFA: every transition out of a final state goes to another final state, so the attempt to match a longer string will either fail immediately (before leaving the final state), or it will succeed immediately.
- ➔ The second CFG is after compaction and the first pass of liveness and interference analysis. Compaction renames registers $r_6 - r_{15}$ and r_{20} to $r_1 - r_{11}$, reducing the size of the register range from 20 to 11 and the size of the liveness and interference matrices almost 2x and 4x respectively. Liveness information is shown at the top of each block. Interference matrix uses asterisk for interfering register pairs and dot for non-interfering ones. It can be seen that there are many dots in the table, which means optimization opportunities. The interference matrix is symmetrical, as the interference relation is commutative.
- ➔ The third CFG is after the second pass of liveness and interference analysis. The number of registers is reduced from 11 to 5. Many operations have been eliminated, for example the copy operation $r_1 \leftarrow r_7$ in the final block 8 of the second CFG was removed by copy coalescing, because registers 1 and 7 did not interfere and register allocation put them in one equivalence class (and likewise for the other copy operations in block 8). In basic blocks 1 and 3 set operations $r_6 \leftarrow \mathbf{p}$ and $r_9 \leftarrow \mathbf{p}$ were collapsed into $r_3 \leftarrow \mathbf{p}$ after non-interfering registers r_6 and r_9 had been renamed to r_3. Interference matrix has dots only on the main diagonal (a register does not interfere with itself), which leaves no room for further optimization.
- ➔ The resulting optimized TDFA is at the bottom of figure 2. The final registers are now $r_1 - r_5$.

3.4 Minimization

Minimization can be applied to TDFA in the same way as to an ordinary DFA (e.g. the Moore's algorithm), except that transitions on the same alphabet symbol but with different register operations should be treated as different transitions, so their destination states cannot be merged. To get optimal performance minimization algorithm should be able to compare operations on transitions in constant time. This is possible if operation sequences are inserted into a hash map and represented with unique numeric identifiers. Such a comparison may have false negatives, as non-identical operations lists may be equivalent (e.g. $r_1 \leftarrow \mathbf{p}, r_2 \leftarrow \mathbf{n}$ is not identical, but equivalent to $r_2 \leftarrow \mathbf{n}, r_1 \leftarrow \mathbf{p}$). False negatives do not affect minimization correctness, but the end result may be suboptimal. To avoid that, minimization should be applied after register optimizations (which may remove some register operations) and most importantly after normalization (defined on page 12).

3.5 Fixed tags

Fixed tags is a very important optimization that happens at the RE level. In cases with high tag density (such as POSIX REs with nested submatch groups) this optimization alone may be more effective than all register optimizations combined. The key observation is, if a pair of tags is within fixed distance from each other, there is no need to track both of them: the value of one tag can be computed from the value of the other tag one by adding a fixed offset. This optimization is fast (linear in the size of a RE) and has the potential to reduce both TDFA construction time and matching time.

Algorithm 7 finds fixed tags by performing top-down structural recursion on a RE. It has four parameters: e is the current sub-RE, t is the current base tag, d is the distance to base tag, and k is the distance to the start of the current level. Levels are parts of a RE where any two points either both match or both do not match. A

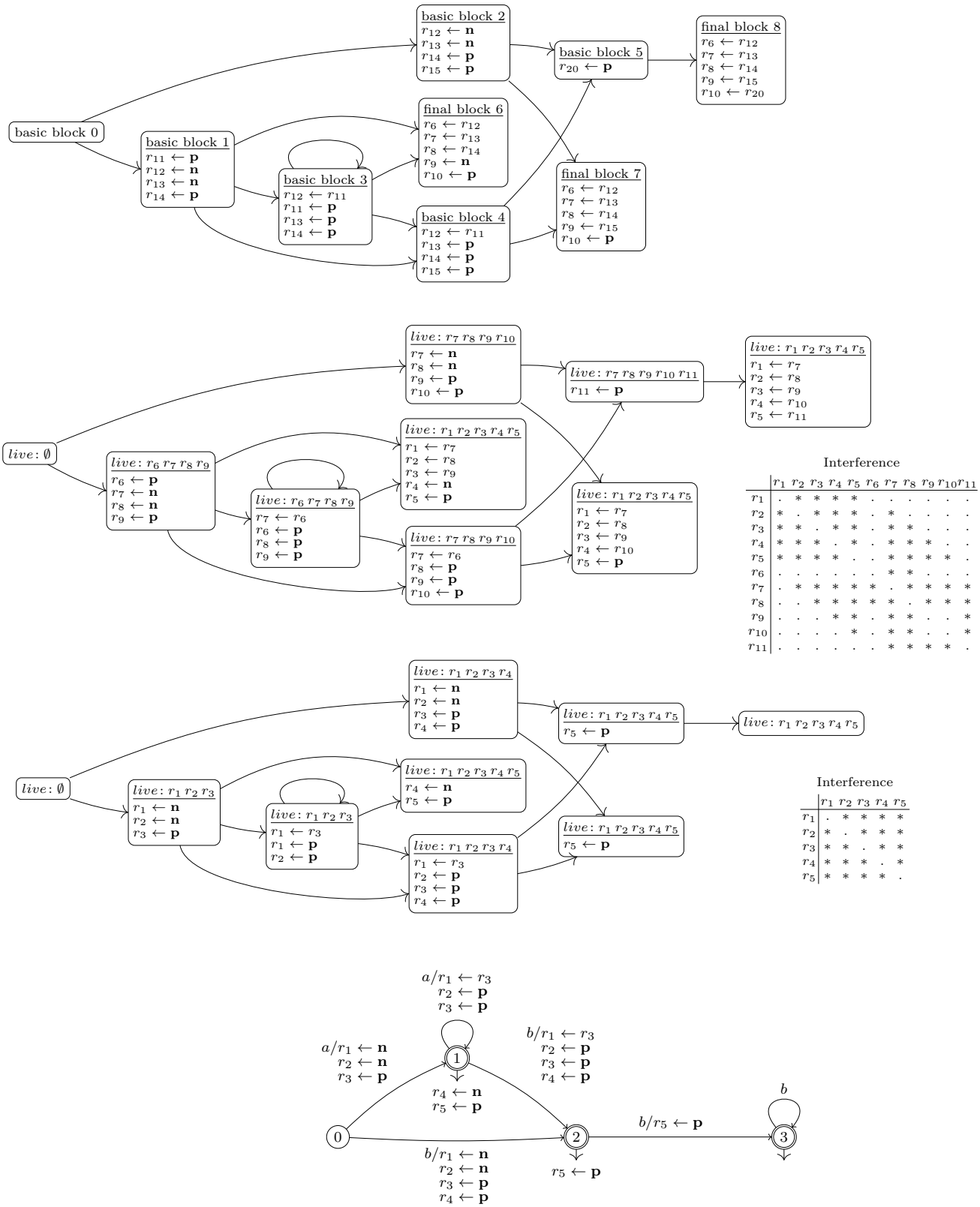


Figure 2: Register optimizations for TDFA on figure 1.
Top to bottom: initial CFG, CFG after compaction with per-block liveness information and interference table, CFG on the second round of optimizations, optimized TDFA with final registers r_1 to r_5.

level increases on recursive descent into alternative or repetition subexpressions, but not concatenation. Tags on different levels should not be fixed on each other, even if they are within fixed distance on any path that goes through both of them, because there are paths that go through only one tag (so the other one is nil). Tag value -1 denotes the absence of base tag: when descending to the next level initially there is no base tag, and the first tag on the current level becomes the base. One exception is the top level, where the initial base tag should be a special value denoting the rightmost position (which is always known at the end of the match). The algorithm recursively returns the new base tag, the updated distance to base tag, and the updated level distance. Special distance value *NaN* (not-a-number) is understood to be a fixed point in arithmetic expressions: any expression involving *NaN* amounts to *NaN*.

$fixed\_tags(e, t, d, k)$

```
1  if e = ε then
2      return t, d, k
3  else if e = a ∈ Σ then
4      return t, d + 1, k + 1
5  else if e = e1|e2 then
6      _, _, k1 = fixed_tags(e1, −1, NaN, 0)
7      _, _, k2 = fixed_tags(e2, −1, NaN, 0)
8      if k1 = k2 then
9          return t, d + k1, k + k1
10     return t, NaN, NaN
11 else if e = e1e2 then
12     t2, d2, k2 = fixed_tags(e2, t, d, k)
13     t1, d1, k1 = fixed_tags(e1, t2, d2, k2)
14     return t1, d1, k1
15 else if e = e1^{n,m} then
16     _, _, k1 = fixed_tags(e1, −1, NaN, 0)
17     if n = m then
18         return t, d + n ∗ k1, k + n ∗ k1
19     return t, NaN, NaN
20 else if e = t1 ∈ T then
21     if t ≠ −1 and d ≠ NaN then
22         mark t1 as fixed on t with distance d
23         return t, d, k
24     return t1, 0, k
```

Algorithm 7: Fixed tags optimization.

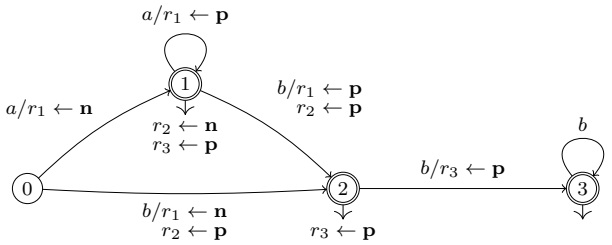


Figure 3: Optimized TDFA with fixed tags $t_1 \leftarrow (\mathbf{n}$ if $t_2 = \mathbf{n}$ else $t_2 - 1)$ and $t_3 \leftarrow (t_5 - 1)$. Tags t_2, t_4, t_5 correspond to final registers r_1, r_2, r_3.

Figure 3 shows the effect of fixed tags in addition to other optimizations on figure 2:

- ➔ In the example RE $(1a2)^*3(a|4b)5b^*$ tags t_1 and t_2 are within one symbol from each other, so the value of t_1 can be computed as nil if t_2 is nil, or $t_2 - 1$ otherwise. Likewise t_3 can be computed as $t_5 - 1$ (although there are multiple different paths through $(a|4b)$, they all have the same length).
- ➔ Fixed tags are identified at the RE level and excluded from TNFA construction and determinization. There are no registers and register operations associated with t_1 and t_3, except for computing their values from the base tags t_2 and t_5 at the end of match.

4 Multi-pass TDFA

TDFA with registers described in section 2 are well suited for ahead-of-time determinization (e.g. in lexer generators) when one can spend considerable time on optimizations in order to emit a more efficient TDFA. However, in RE libraries the overhead on determinization and optimizations is included in the run time, therefore it is desirable to reduce TDFA construction time (although the overhead may be amortized if a RE is compiled once and used to match many strings).

Another concern is the density of submatch information in a RE. TDFA with registers are perfect for *sparse* submatch extraction, when the number of tags is small compared to the size of RE and the runtime performance is

expected be close to an ordinary DFA. However, if a RE contains many tags (in the extreme, if every subexpression is tagged) then transitions get cluttered with operations, making TDFA execution very slow. Moreover, the optimizations described in section 3 become problematic due to the size of liveness and interference tables.

Multi-pass TDFA address these issues: they reduce TDFA construction time and they are better suited to dense submatch extraction. The main difference with canonical TDFA is that multi-pass TDFA have no register operations. Instead, as the name suggests, they have multiple passes: a *forward* pass that matches the input string and records a sequence of TDFA states, and one or more *backward* passes that iterate through the recorded states and collect submatch information (one backward pass is sufficient, but an extra pass may be used e.g. to estimate and preallocate memory for the results). The representation of submatch results may vary and affects only the backward pass(es); forward pass is the same for all representations.

Multi-pass TDFA construction differs from algorithm 3 in section 2 in a few ways. Recall that a closure C corresponds to a TDFA transition between states s and s'. For multi-pass TDFA closure configurations are extended to five components (q, o, r, h, l) where the new component o is the *origin* TNFA state in s that leads to state q in s', and the remaining components are as in algorithm 3. Origins are needed to trace back the matching TNFA path from a sequence of TDFA states. Path fragments corresponding to closure configurations are represented with *backlinks*, and every TDFA transition is associated with a backlink array. A backlink is a pair (i, h) where i is an index in backlink arrays on preceding transitions, and h is a tag sequence corresponding to the h-component of a configuration. Backlinks on transitions do not map one-to-one to configurations, because in TDFA(1) contrary to TDFA(0) configurations with identical origins have identical h-components (inherited

unique_origins(C)

```
U : mapping from TNFA states in C to integers
i = 0
for each unique origin o in C do
    for each (q, o', _, _, _) in C such that o' = o do
        U[q] = i
    i = i + 1
return U
```

match$(\mathcal{F},\ a_1 \dots a_n)$

```
V = {s_0}
for k = 1, n do
    if s = δ(s, a_k) is defined then
        append s to V
    else return ∅
return V
```

extract_tstring$(\mathcal{F},\ a_1 \dots a_n,\ s_0 \dots s_n)$

```
(i, h) = φ(s_n)
x = h
for k = n, 1 do
    (_, B) = δ(s_{k-1}, a_k)
    (i, h) = B[i]
    x = h · a_k · x
return x
```

extract_offsets$(\mathcal{F},\ a_1 \dots a_n,\ s_0 \dots s_n)$

```
E = {∅}_{i=1}^{|T|} (no value for each tag)
(i, h) = φ(s_n)
k = n
while true do
    for tag t in h in reverse order do
        if t > 0 and E[t] = ∅ then
            E[t] = k
        else if E[−t] = ∅ then
            E[−t] = −1
    if k = 0 then break
    (_, B) = δ(s_{k-1}, a_k)
    (i, h) = B[i]
    k = k − 1
return E
```

construct_backlinks(C, U, U')

```
B : backlink array of size |range(U')|
for each (q, o, _, h, _) in C do
    i = U'[q]
    if B[i] is undefined then
        B[i] = (U[o], h)
return B
```

extract_offset_lists$(\mathcal{F},\ a_1 \dots a_n,\ s_0 \dots s_n)$

```
E = {{}}_{i=1}^{|T|} (empty list for each tag)
(i, h) = φ(s_n)
k = n
while true do
    for tag t in h in reverse order do
        if t > 0 then
            prepend k to E[t]
        else
            prepend −1 to E[−t]
    if k = 0 then break
    (_, B) = δ(s_{k-1}, a_k)
    (i, h) = B[i]
    k = k − 1
return E
```

Algorithm 8: Backlink construction and matching with multi-pass TDFA $\mathcal{F} = (\Sigma, T, S, S_f, s_0, \delta, \varphi)$.

from the lookahead tags), resulting in identical backlinks. To deduplicate such backlinks, *unique_origins* in algorithm 8 creates a per-state mapping from origin state to a unique origin index. Transition function is defined as $\delta(s, a) = (s', B)$ where $B = construct\_backlinks(C, U, U')$ and U, U' are the unique origin mappings for s and s' respectively. Final TDFA states are associated with a single backlink, and the final function is defined as $\varphi(s) = (i, l)$ where i is the unique origin index of the final state q_f in C and l is the lookahead tag sequence (l-component of the final configuration). The resulting TDFA has no registers or register operations; the R and R_f components are removed from TDFA, and functions *transition_regops*, *final_regops* and their dependencies in algorithm 3 are not needed.

Algorithm 8 shows matching with a multi-pass TDFA. The forward pass is defined by the function *match*, which executes TDFA on a string $a_1 \dots a_n$ and returns the matching sequence of TDFA states $s_0 \dots s_n$ (or $\varnothing$ on failure). Backward pass depends on the representation of submatch results; we provide three variants for offsets, offset lists and tagged strings. In each case the backward pass follows a sequence of backlinks from the final state to the initial state. Function *extract_offsets* extracts the last offset for each tag and avoids overwriting it by initializing all offsets to $\varnothing$ and checking each offset before writing. Function *extract_offset_lists* is similar, but it collects offsets into lists. Function *extract_tstring* concatenates the h-components of backlinks interleaved with input symbols (this representation can be used to reconstruct a full parse tree, see [6] section 6).

In practice we found that the following details affect performance of algorithm 8. The h-components of backlinks should be stored as arrays which allow fast access to individual tags, rather than linked lists packed in a prefix tree (the latter representation was used for multi-valued tags in section 3). The forward pass should record references to backlink arrays instead of TDFA states in order to reduce the number of indirections and lookups. For tagged strings a separate backward pass may be used to estimate the amount of space for the resulting string and preallocate it. If tags in a RE have *nested* structure (e.g. in the case of POSIX capturing groups) then negative transition should be added only for the topmost tag of a subexpression (as described in [6] section 9) rather than for all nested tags (as described in algorithm 2). The mapping from a tag to its nested tags should be stored separately and used during matching (as in [6] section 6).

Figure 4 shows multi-pass TDFA for the running example (compare it with figure 1):

- ➔ TDFA transition $0 \to 1$ has two backlinks because the five configurations in state 1 originate in two TNFA states 2 and 9. Likewise transition $1 \to 1$ has two backlinks corresponding to origins 2 and 9, transitions $0 \to 2$ and $1 \to 2$ have one backlink corresponding to origin 12, and transitions $2 \to 3$ and $3 \to 3$ have one backlink corresponding to origin 15.
- ➔ Final states 1, 2 and 3 have a backlink corresponding to the final configuration with TNFA state 17.
- ➔ The i-component of each backlink equals to the unique origin index of the configuration o-component. For example, both backlinks on transition $1 \to 1$ have $i = 0$ because their configurations (in the shadow TDFA state mapped to state 1 on figure 1) have origins 2 and 9 in TDFA state 1, which have the same unique origin index 0 (because they both have origin 2 in TDFA state 0). Consequently, both backlinks on transition $1 \to 1$ connect to the first backlink on transitions $0 \to 1$ and $1 \to 1$. On the other hand, the final backlink in state 1 connects to the second one.

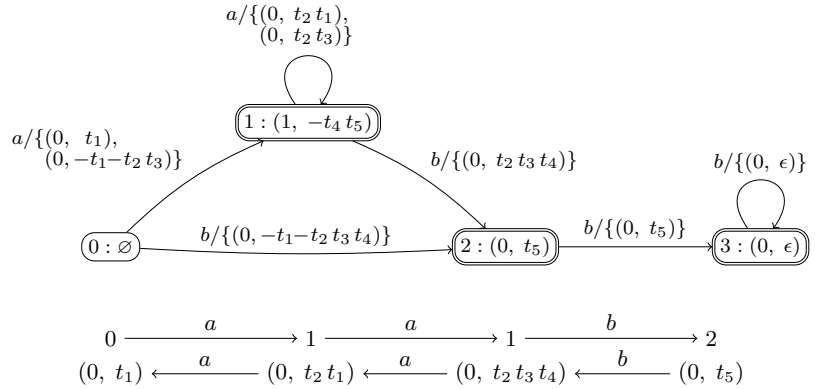


Figure 4: Multi-pass TDFA for RE $(1a2)^*3(a|4b)5b^*$ matching string aab.

➔ The sequence of TDFA states matching *aab* is $0 \to 1 \to 1 \to 2$, and backlinks can be traced back from the final backlink in state 2 using i-component as index in backlink arrays.

➔ Submatch results for string *aab* are as follows. Single offsets: $t_1 = 1$, $t_2 = 2$, $t_3 = 2$, $t_4 = 2$, $t_5 = 3$. Offset lists: $t_1 = \{0, 1\}$, $t_2 = \{1, 2\}$, $t_3 = \{2\}$, $t_4 = \{2\}$, $t_5 = \{3\}$. Tagged string: $1\,a\,2\,1\,a\,2\,3\,4\,b\,5$.

5 Evaluation

In this section we evaluate TDFA performance in practice and compare it to other algorithms. We present three groups of benchmarks that cover different settings and show different aspects of the algorithm:

1. **AOT determinization** (figures 5, 6, 7, 8). We compare three lexer generators: RE2C [4], Ragel [14] and Kleenex [13] that are based on different types of deterministic automata. All of them generate optimized C code, which is further compiled to binary by GCC and Clang. The generated programs do string rewriting: they read 100MB of input text and insert markers at submatch extraction points. These benchmarks use leftmost-greedy disambiguation. The following automata are compared:

   - **TDFA(1)**, the algorithm described by Trafimovich in [3] and presented in this paper. It is implemented in RE2C with the optimizations described in section 3.
   - **TDFA(0)**, the original algorithm described by Laurikari in [1]. Contrary to TDFA(1) that use one-symbol lookahead, TDFA(0) do not use lookahead: the two types of automata are called so by analogy with LR(1) and LR(0). TDFA(0) apply register operations to the incoming transition, while TDFA(1) split them on the lookahead symbol and apply them to the outgoing transitions, which reduces the number of tag conflicts. As a consequence, TDFA(0) typically require more registers and copy operations, which makes them slower than TDFA(1); see [3] for a detailed comparison. TDFA(0) algorithm is also implemented in RE2C and benefits from the same optimizations as TDFA(1).
   - **StaDFA**, the algorithm described by Chowdhury in [11], with a few modifications of our own that were necessary for correctness. It is very similar to TDFA, but the automata have register operations in states rather than on transitions (which implies that staDFA do not use lookahead). The algorithm is implemented in RE2C and uses the same optimizations as TDFA(1) and TDFA(0).
   - **DSSTs**, the algorithm described by Grathwohl in [8]. DSSTs stands for Deterministic Streaming String Transducers; these are more distant relatives to TDFA, better suited to string rewriting and full parsing. DSST states contain path trees constructed by the ϵ-closure, while TDFA states contain similar information decomposed into register tables and lookahead tags. DSST registers contain fragments of strings over the output alphabet (the analogue of our tagged strings). Register operations on transitions concatenate and move string fragments. DSSTs are implemented in Kleenex.
   - Ordinary **DFA** with ad-hoc user-defined actions and manual conflict resolution via precedence operators, implemented in Ragel. This approach is fast, but it has correctness issues: in some cases it is impossible to resolve the conflicts between actions by preferring one action over the other; instead, it is necessary to keep both actions until more input is consumed and non-determinism is resolved. But this is also impossible, as the actions modify the same shared state (e.g. set the same local variables). An action may conflict with itself on different transitions due to non-determinism.

2. **JIT determinization, C++** (figure 9). These benchmarks compare **TDFA(1)** and **multi-pass TDFA(1)** presented in section 4 in the case of **single offsets** and **offset lists**. Both algorithms are implemented in a C++ library based on RE2C. These benchmarks use POSIX disambiguation.

3. **JIT determinization, Java** (figures 10, 11). We compare two independent implementations: one in pure Java (by Borsotti), and one in C++ via JNI, based on RE2C (by Trafimovich), both published as part of the RE2C repository [5]. These benchmarks compare **TDFA(1)** and **multi-pass TDFA(1)** in the case of **single offsets**, **offset lists** and **tagged strings**, and with different submatch density: **sparse tags** and **full parsing** (where every subexpression in a RE is tagged). They use POSIX disambiguation.

Hardware specifications: Intel Core i7-8750H CPU with 32 KiB L1 data cache, 32 KiB L1 instruction cache, 256 KiB L2 cache, 9216 KiB L3 cache, 32 GiB RAM. Software versions: RE2C 3.0, Ragel 7.0.4, Kleenex built from Git at commit d474c60, GCC 11.2.0, Clang 13.0.0, OpenJDK 17.0.1.

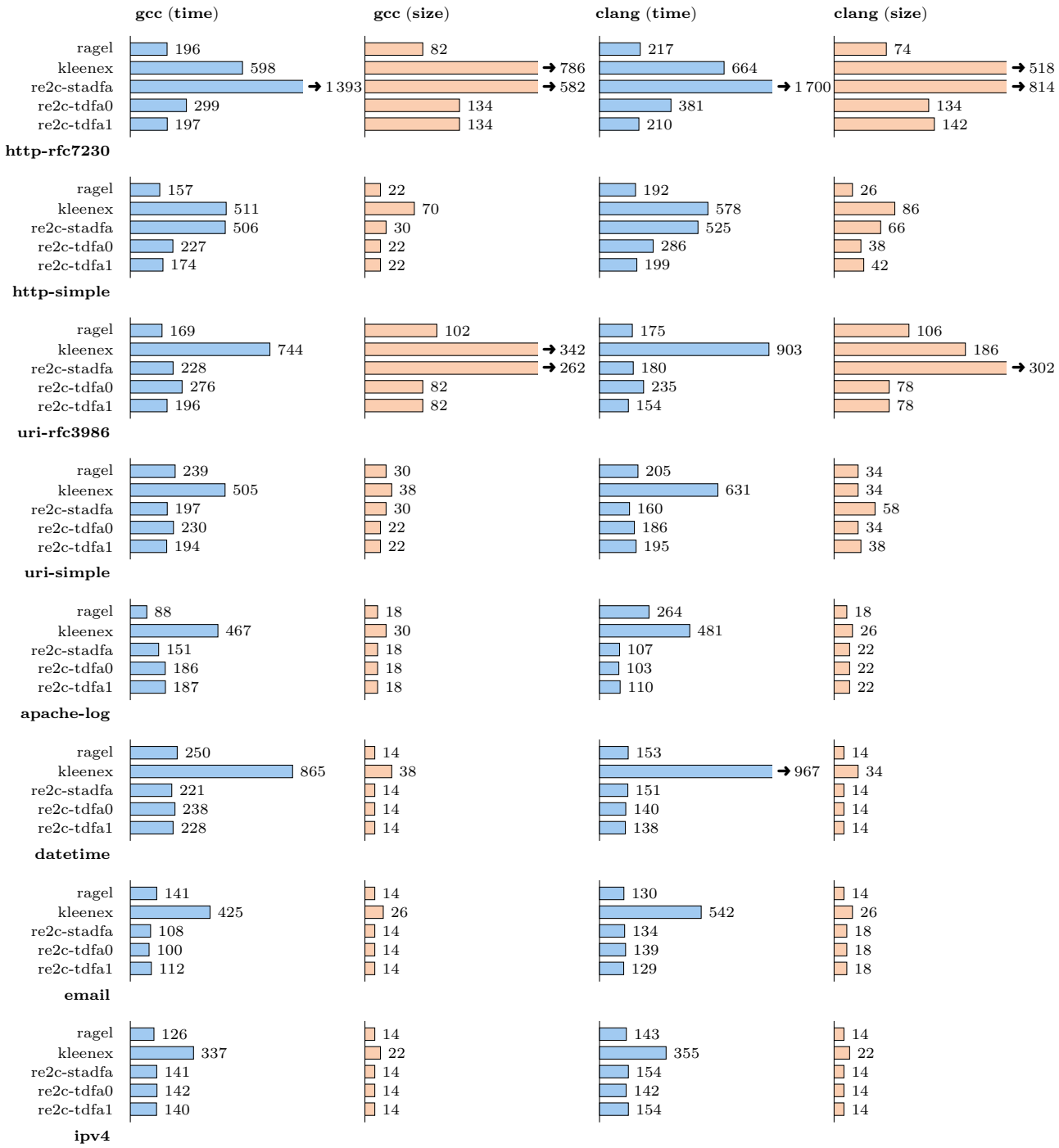


Figure 5: Benchmarks for AOT determinization, real-world REs.

Figure 5 shows benchmark results for AOT determinization in the case of real-world REs that are likely to be used in practice: HTTP message headers, URI, Apache logs, date, email addresses and IP addresses. REs vary from very large and complex to small and simple; the number of tags in REs varies accordingly.

The main conclusions are:

- TDFA(1) and ordinary DFA are close in size and speed (the result often depends on GCC/Clang).
- In simple cases staDFA and TDFA(0) are on par with TDFA(1), but in complex cases they are considerably slower and larger, and staDFA can get extremely large.
- DSSTs are generally slower and larger in most of the cases.

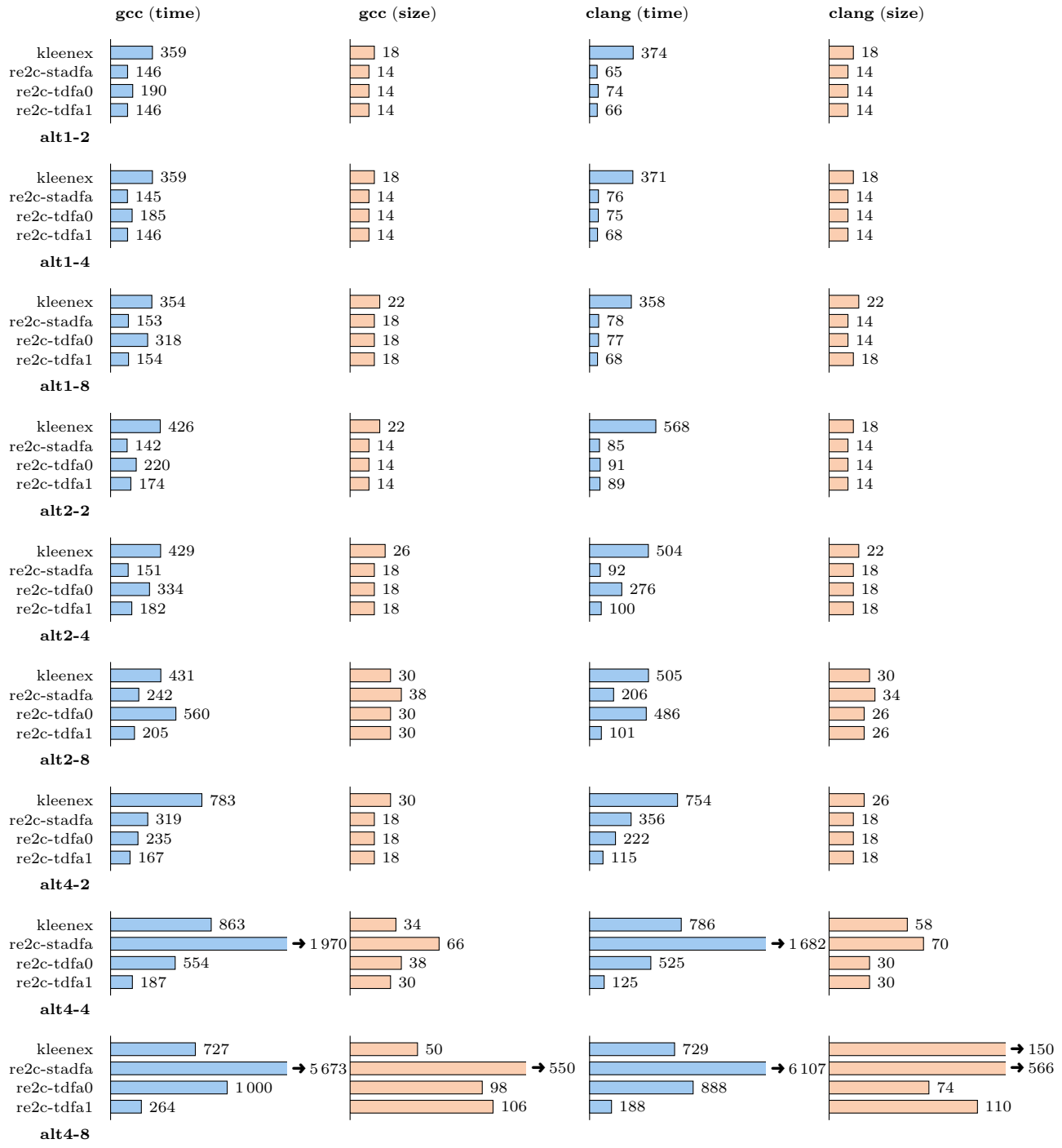


Figure 6: Benchmarks for AOT determinization, artificial REs with alternative.

Figure 6 shows benchmark results for AOT determinization in the case of artificial REs with emphasis on alternative, in series of increasing size, complexity and the number of tags. Ordinary DFA are excluded because Ragel's ad-hoc disambiguation operators do not allow to implement all cases correctly. Conclusions:

- TDFA(1) perform better than other algorithms.
- TDFA(0) are generally slower than TDFA(1); the difference grows with RE size.
- StaDFA are close to TDFA(1) on small REs, but they degrade on large REs in both size and speed.
- DSSTs are generally slower and almost always larger than TDFA(1).

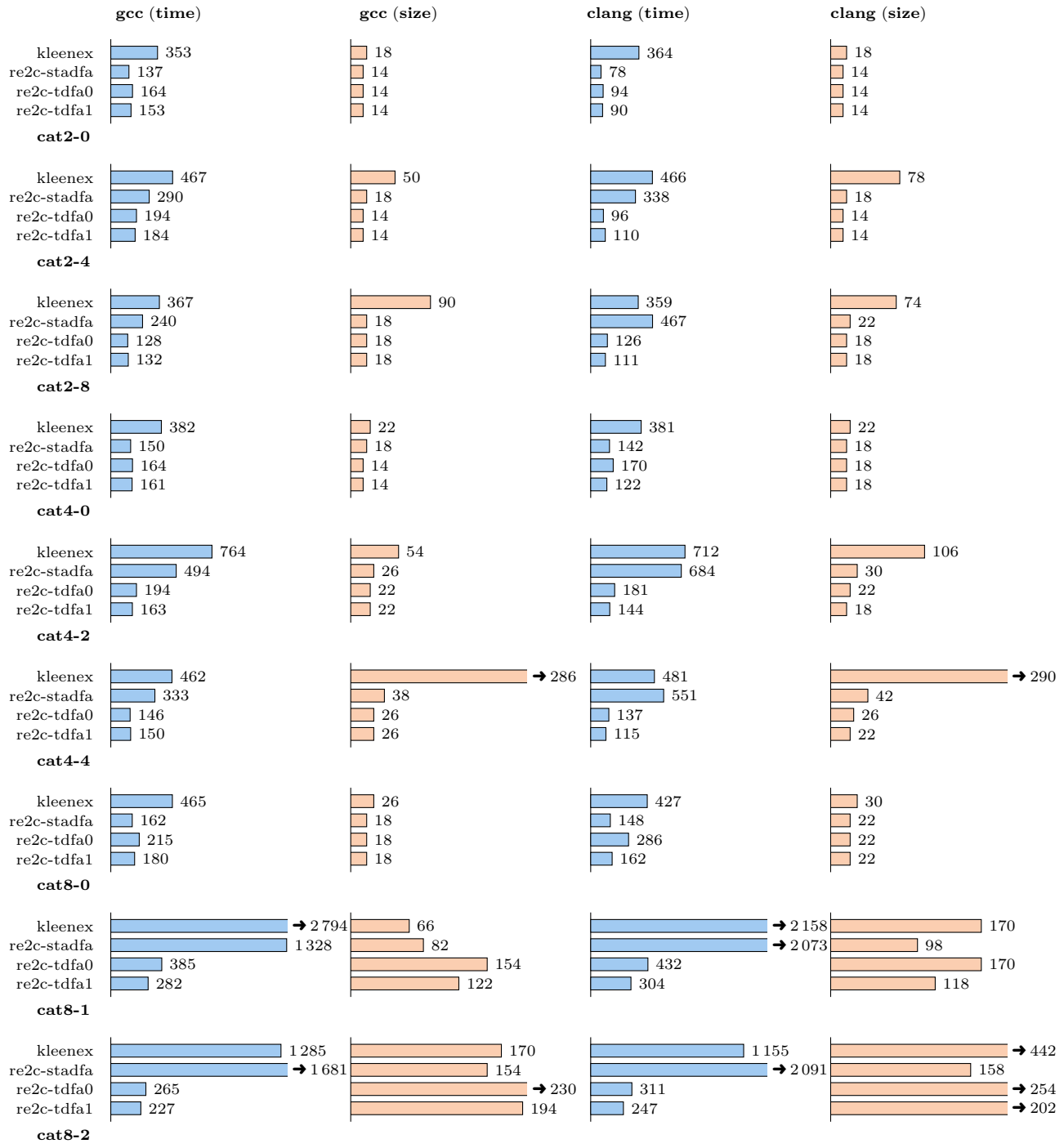


Figure 7: Benchmarks for AOT determinization, artificial REs with concatenation.

Figure 7 shows benchmark results for AOT determinization in the case of artificial REs with emphasis on concatenation, in series of increasing size, complexity and the number of tags. Ordinary DFA are excluded because Ragel's ad-hoc disambiguation operators do not allow to implement all cases correctly. Conclusions:

- TDFA(1) perform better than other algorithms.
- TDFA(0) are slower than TDFA(1), but the difference is not radical.
- StaDFA are slower than TDFA(1) on small REs, and the difference gets radical with RE size.
- DSSTs are generally slower and almost always larger than TDFA(1).

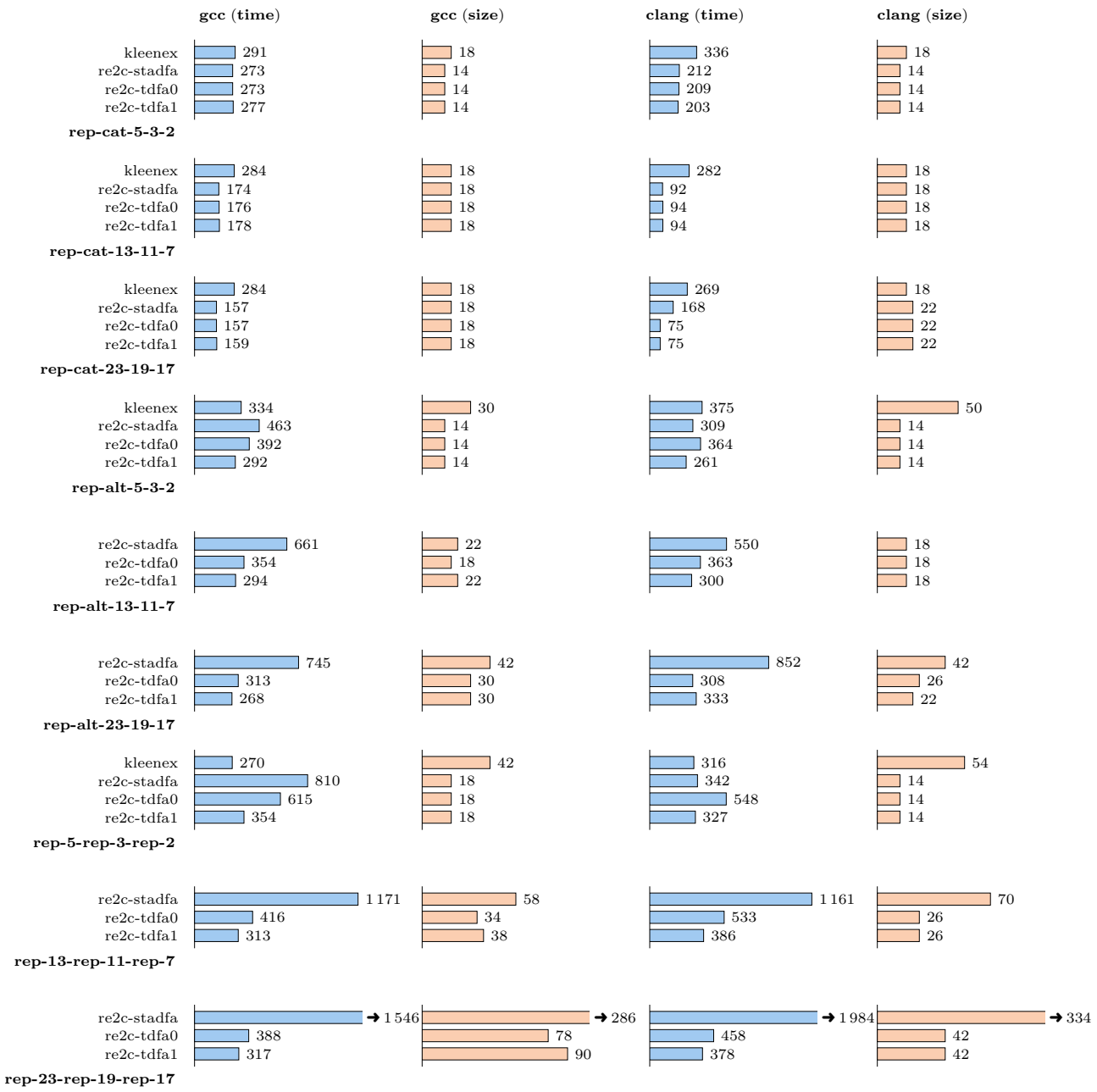


Figure 8: Benchmarks for AOT determinization, artificial REs with repetition.

Figure 8 shows benchmark results for AOT determinization in the case of artificial REs with emphasis on repetition, in series of increasing size, complexity and the number of tags. Ordinary DFA are excluded because Ragel's ad-hoc disambiguation operators do not allow to implement all cases correctly, and DSSTs are excluded in cases where they get too large to be compiled. Conclusions:

- TDFA(1) perform better than other algorithms.
- TDFA(0) are slower than TDFA(1), but the difference is not radical.
- StaDFA are slower and larger than TDFA(1) on small REs, and the difference gets radical with RE size.
- DSSTs are generally larger than TDFA(1), and the difference gets extreme with RE size.

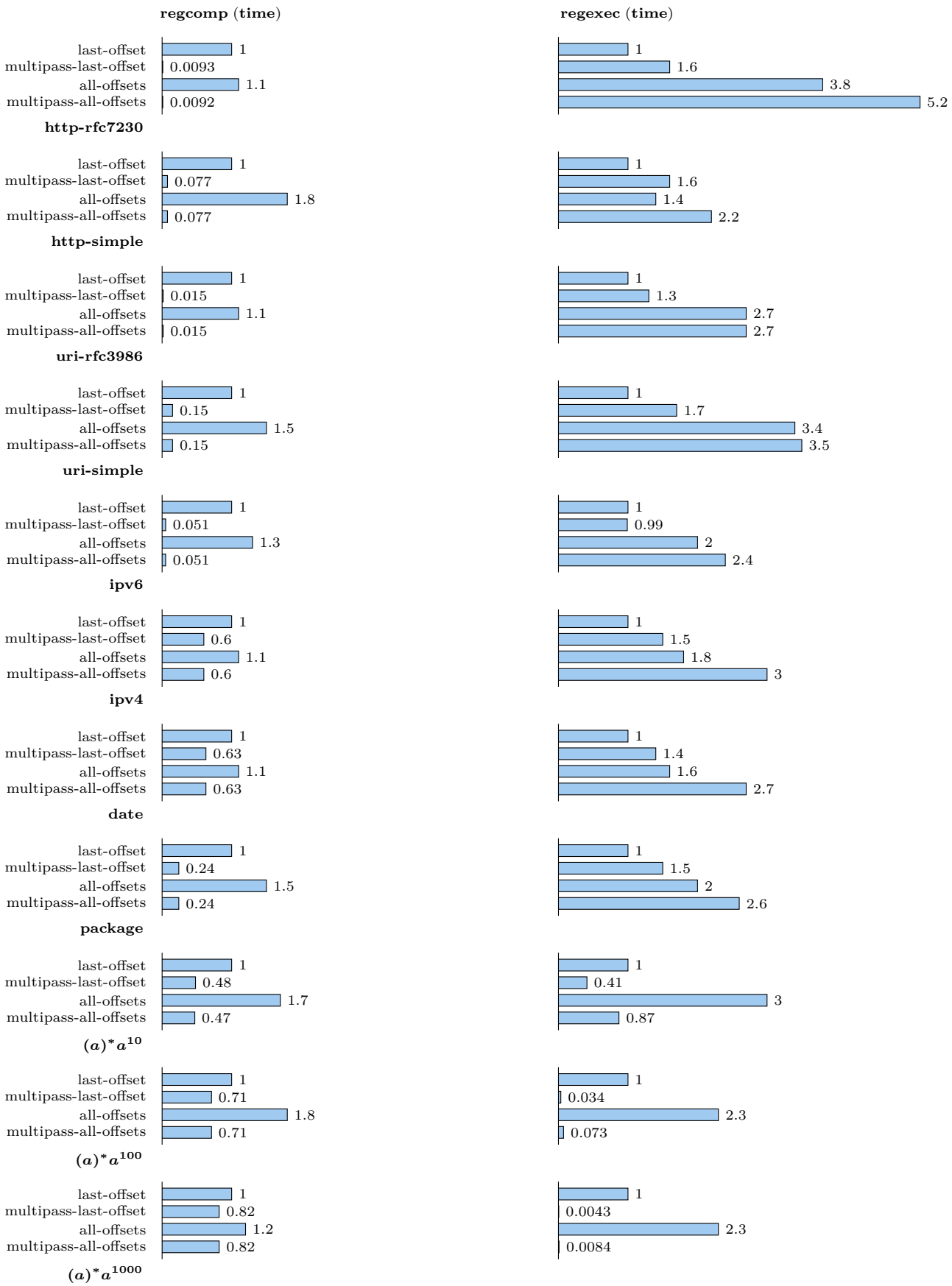


Figure 9: Benchmarks for JIT determinization, C++ (`regcomp`/`regexec` time, relative).

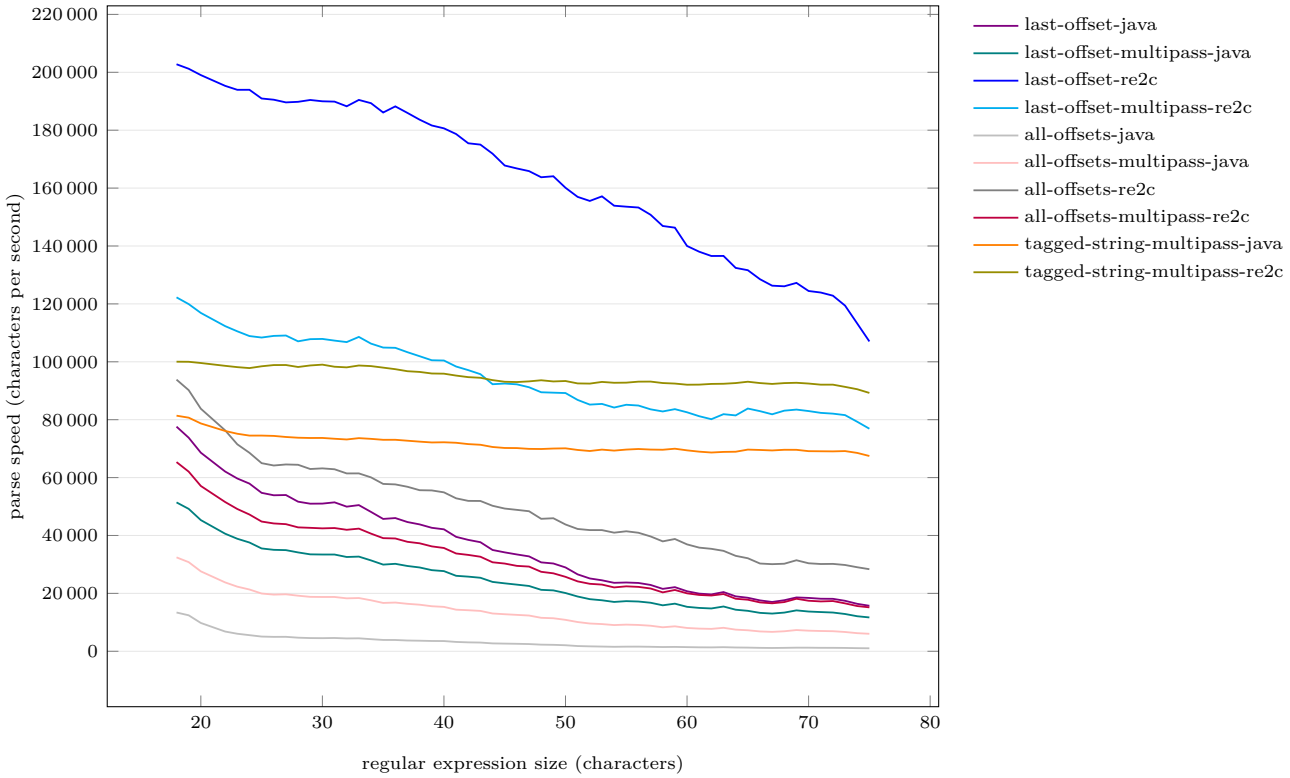


Figure 10: Benchmarks for JIT determinization, Java, sparse tags (`regexec` speed).

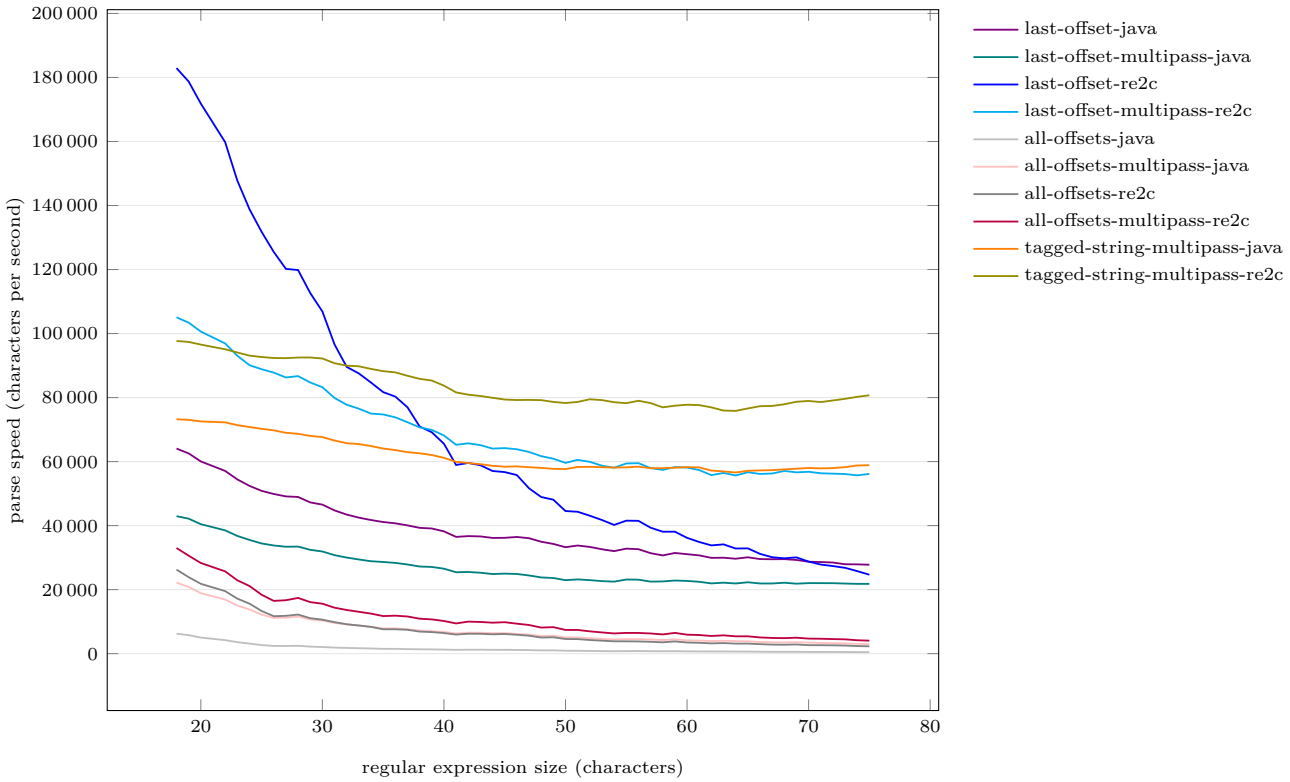


Figure 11: Benchmarks for JIT determinization, Java, full parsing (`regexec` speed).

Figure 9 shows benchmark results for JIT determinization, C++. Time is shown relative to the first row. There are two groups of benchmarks: real-world REs and artificial REs $(a)^*a^{10^k}$ for $k \in \{1, 2, 3\}$. Conclusions:

- Compilation is predictably slower for TDFA than for multi-pass TDFA for both groups, as multi-pass TDFA do not need register actions and subsequent register optimizations.
- Execution time differs for the two groups: for real-world benchmarks TDFA are generally faster than multi-pass TDFA, while for artificial benchmarks TDFA are much slower than multi-pass TDFA, and the difference grows with the size of RE. In fact artificial REs demonstrate a pathological case for TDFA with register actions: increasing k results in increased degree of nondeterminism, which requires more registers and copy operations in order to track all nondeterministic values. High degree of nondeterminism is specific to some REs with counted repetition, as demonstrated in [3] (page 21).
- The results are similar for single-offset and offset-list cases, although the latter is predictably slower.

Figures 10 and 11 show benchmark results for JIT determinization, Java, in the case of sparse tags and full parsing respectively. The plots show the dependence of matching speed on RE size. Conclusions:

- Remarkably, the case of tagged strings with multi-pass TDFA is the only one that shows almost no degradation with RE size (the lines are almost horizontal). This holds for both implementations.
- TDFA with register actions (the RE2C implementation) is clearly the fastest algorithm in the case of sparse tags. However, in the case of full parsing it either degrades faster than multi-pass TDFA (in the last-offset case), or it is generally slower (in the offset-list case). For pure-Java implementation multi-pass TDFA is almost always faster than TDFA with register actions.

6 Conclusions

TDFA(1) are generally faster and smaller than other automata capable of submatch extraction.

Optimizations play a very important part in any performance-sensitive TDFA implementation (compare the unoptimized TDFA on figure 1 with the final optimized TDFA on figure 3).

The overhead on submatch extraction depends on tag density and degree of nondeterminism in a RE. In the case of sparse tags with low nondeterminism TDFA with register actions are by far the fastest and have negligible difference compared to ordinary DFA. In the case of high tag density (in the extreme, full parsing) or in the case of highly nondeterministic REs multi-pass TDFA are more efficient.

The overhead on submatch extraction depends on the representation of submatch results. Tagged string extraction with multi-pass TDFA is the only algorithm that shows almost no degradation with RE size. Extracting only the last offset is predictably faster than extracting all offsets (fortunately, the choice is individual for each tag, so all offsets can be extracted only for a selected subset of tags).

Multi-pass TDFA are better suited to JIT determinization than TDFA with register actions.

7 Future work

One very useful direction of future work is to find *deterministic points* in a RE. Often shifting a tag by a fixed number of characters in a concatenation subexpression can reduce its degree of nondeterminism (the maximum number of registers in a single TDFA state needed to track all parallel versions of the same tag). As a consequence, this means fewer registers and register operations. For example, tag t_1 in $a^*1a^ka^*$ has nondeterminism degree k and requires $2 * k$ register operations, while tag t_2 in $a^*a^k2a^*$ has degree is 1 and only 1 operation. But tags t_1 and t_2 are within fixed distance of k characters, so t_1 can be the computed as $t_2 - k$. In other words, t_2 is a deterministic point for t_1. Identifying such points in a RE would be a useful optimization.

Acknowledgments

I want to thank my parents Vladimir and Elina, my dearest friend and open source programmer Sergei, my teachers Tatyana Leonidovna and Demian Vladimirovich and the whole open source community. And, of course, my coauthor Angelo who was the greatest inspiration and help in this work! *Ulya Trafimovich*

References

[1] Ville Laurikari, *NFAs with tagged transitions, their conversion to deterministic automata and application to regular expressions*, Proceedings Seventh International Symposium on String Processing and Information Retrieval. SPIRE 2000, pp. 181-187, `http://laurikari.net/ville/spire2000-tnfa.pdf`, 2000.

[2] Chris Kuklewicz, *Regular expressions/bounded space proposal*, `http://wiki.haskell.org/index.php?title=Regular_expressions/Bounded_space_proposal&oldid=11475`, 2007.

[3] Ulya Trafimovich, *Tagged Deterministic Finite Automata with Lookahead*, arXiv:1907.08837 [cs.FL], 2017.

[4] RE2C, a lexer generator for C, C++, Go and Rust. Source code: `https://github.com/skvadrik/re2c`. Official website: `https://re2c.org`.

[5] Experimental Java library for TDFA. Source code: `https://github.com/skvadrik/re2c/tree/master/benchmarks/submatch_java`.

[6] Angelo Borsotti, Ulya Trafimovich, *Efficient POSIX submatch extraction on nondeterministic finite automata*, Software: Practice and Experience 51, no. 2, pp. 159–192, DOI: `https://doi.org/10.1002/spe.2881`, preprint: `https://www.researchgate.net/publication/344781678_Efficient_POSIX_submatch_extraction_on_nondeterministic_finite_automata`, 2019.

[7] Ulya Trafimovich, *RE2C: A lexer generator based on lookahead-TDFA*, Software Impacts, 6, 100027, DOI: `https://doi.org/10.1016/j.simpa.2020.100027`, 2020.

[8] Niels Bjørn Bugge Grathwohl, *Parsing with Regular Expressions & Extensions to Kleene Algebra*, DIKU, University of Copenhagen, 2015.

[9] Stephen Cole Kleene, *Representation of events in nerve nets and finite automata*, RAND Project US Air Force, 1951.

[10] Dexter Kozen, *A completeness theorem for Kleene algebras and the algebra of regular events*, Elsevier, Information and computation, vol. 110 (2) pp. 366–390, 1994.

[11] Mohammad Imran Chowdhury, *staDFA: An Efficient Subexpression Matching Method*, Master thesis, Florida State University, 2018.

[12] Aaron Karper, *Efficient regular expressions that produce parse trees* (thesis), University of Bern, 2014.

[13] Kleenex language, DIKU, University of Copenhagen. Official website: `https://kleenexlang.org`. Source code: `https://github.com/diku-kmc/kleenexlang`,

[14] Ragel State Machine Compiler. Official website: `https://www.colm.net/open-source/ragel`.