fAST: regular expression inference from positive examples using Abstract Syntax Trees

Maxime Raynal MAXIME.RAYNAL@NOKIA.COM
Nokia Bell Labs France
Univ. Grenoble Alpes, CNRS, Grenoble INP, LIG, F-38000 Grenoble France

Marc-Olivier Buob MARC-OLIVIER.BUOB@NOKIA-BELL-LABS.COM
Nokia Bell Labs France

Georges Quénot GEORGES.QUENOT@IMAG.FR
Univ. Grenoble Alpes, CNRS, Grenoble INP, LIG, F-38000 Grenoble France

Editors: François Coste, Faissal Ouardi and Guillaume Rabusseau

Abstract

Our paper presents a new algorithm that infers a regular expression matching a given set of strings known as *positive examples*[1]. This algorithm has practical applications in automating file parsing for files with an unknown template. In practice, prior works hardly apply because they require negative examples and hardly scale. By restricting to positive examples, the problem becomes especially challenging: many regular expressions can match the set of positive examples, but only a few are useful in practice. To assess the quality of a regular expression, we introduce two performance metrics, called accuracy and conciseness. The contributions of the paper are threefold. First, we introduce an algorithm that infers a regular expression from positive examples only while optimizing accuracy and conciseness. Second, we adapt this algorithm to generate a regular expression based on a set of predefined patterns. Third, we demonstrate the tractability and the usefulness of our solution by performing experiments on synthesized and real-world datasets.

Keywords: Regular expressions, grammar synthesis, abstract syntax trees (ASTs)

1. Introduction

Unstructured data are ubiquitous in computer networks. However, these data are purely textual, which allow the automation of some tasks, *e.g.*, monitoring the status of a service Barth (2008), taking counter measures in case of network attacks Jaquier (2004), orchestrate devices Arundel (2013); Zadka (2022), or pinpointing the cause of a failure Lahmadi and Beck (2015). To this end, the unstructured data must be transformed into structured data by using a parser. Most of programming languages provide on-the-shelf parsers to load data conforming to a standard formats, like JSON or XML. If the file format is not standardized, the parser must manually be developed, which is time-consuming, error-prone and cumbersome, or rely on tools trying to perform automatic information retrieval Fisher et al. (2008); Gao et al. (2018); Petrova-Antonova and Tancheva (2020).

Designing a parser is not straightforward. It requires to understand the structure of the file (how it is organized, what are the variables, what are their respective domains) to deduce

1. Positive (resp. negative) examples are strings that do (resp. do not) belong to the target language.

the appropriate set of instructions required to extract the data. Such instructions typically involve *regular expressions* that depend on the nature of the data, *i.e.*, the variable types. Finding the parser from a given file (or from a set of files conforming to a same unknown template) falls into the grammar induction problem De la Higuera (2010). Indeed, grammar induction consists in inferring a *target language* from a set of observations.

The nature of the set of observations determines which algorithm(s) can be applied. Unfortunately, only few approaches are designed to infer a language from a (small) set of positive examples. Gold (1978); Oncina and Garcia (1992); Lee et al. (2016); Bartoli et al. (2016); Kim et al. (2021) infer correct but inaccurate (and often trivial) languages without negative example, and are thus inadequate for parsing. Miclet (1980); Angluin (1982); Garcia et al. (1990); García and Ruiz (1996); Avellaneda and Petrenko (2019) do not require negative examples, but infer optimized automata, while we are rather interested in short and accurate regular expressions. Ideally, we aim at reaching the following goals:

- *Correctness:* the inferred regular expression matches each positive example;
- *Accuracy:* the inferred regular expression is strict enough to derive the accurate parsing rules that lead to homogeneous and structured data;
- *Conciseness:* the inferred regular expression should be as concise as possible to lead to readable parsing rules;
- *Scalability:* the algorithm should be able to infer a regular expression from a large set of examples in reasonable time;
- *Prior knowledge:* as regular expression are usually crafted to extract values conforming to standardized data types (strings, numeric values, network addresses, dates, times, paths, etc.), the algorithm should infer a regular expression involving the patterns matching those data types.

Intuitively, the conciseness and the accuracy are two contradictory objectives, and our goal here is to find a good trade-off. By doing so, the expression is both generic enough to recognize all the input positive examples, and strict enough to still capture the structure of the file to be parsed while remaining short and readable.

To this end, we propose a new algorithm, called fAST (find Abstract Syntax Tree), designed to infer an accurate regular expression from a set of positive examples W (called *sample*) without requiring negative examples. fAST maintains a population of *ASTs (Abstract Syntax Trees)* representing candidate regular expressions by processing each positive example character by character.

The contributions of this paper are threefold. First, we propose an algorithm that finds a solution optimizing the accuracy and the conciseness. Second, we extend our algorithm so that the end-user may specify patterns (typically, common data types) which allows to find languages adapted for practical use cases. Third, we demonstrate the tractability of our algorithm by running experiments on real-world datasets.

The rest of the paper is organized as follows. Section 2 presents the works related to grammar induction and automatic parsing. Section 3 introduces the definitions and notations used in this paper. Section 4 formally defines the problem we aim to solve.

Section 5 presents the big picture of the fAST algorithm. Section 6 shows how to extend the fAST algorithm to take into account predefined patterns and build regular expressions using those. Section 7 evaluates our proposal on several real datasets. Finally, Section 8 concludes the paper.

2. Related works

The grammar induction problem consists in finding a target language from a collection of observations that usually comprises positive and negative examples, the nature of which varies depending on the proposal. Gold (1967) proved that there is no learner that can identify in the limit regular languages by restricting to positive data. Nonetheless, the grammar induction problem falls within the Probably Approximately Correct (PAC) learning framework as per Valiant (1984). The basic idea of PAC learning is that we can minimize the chance of learning something that is wrong without being completely sure that we are right Adriaans et al. (2004).

The Gold algorithm Gold (1978) is a state merging algorithm that constructs a deterministic finite automaton by using a set of positive and negative examples. The merging of states in the algorithm is a direct consequence of the Myhill-Nerode Nerode (1958) relation, which enables the merging of states that lead to the same suffixes. The RPNI algorithm Oncina and Garcia (1992) and the blue-fringe algorithm Lang et al. (1998) are modified versions of the Gold algorithm, with some adjustments. In the RPNI algorithm, unlike the Gold algorithm, the merging of states is performed opportunistically, as long as it does not contradict the negative examples provided. On the other hand, the blue-fringe algorithm uses a heuristic to determine which states should be merged. The RPNI algorithm can be enhanced by parallelizing it Balcázar et al. (1994), or extending it to handle stochastic automata Higuera and Thollard (2000), or noisy data Sebban et al. (2004). Miclet (1980); Avellaneda and Petrenko (2019) infer deterministic automata by relying only on positive examples, so do Angluin (1982); Garcia et al. (1990); García and Ruiz (1996) with additional assumption on the nature of the target language. Nonetheless, all these approaches optimize the output automaton, which once converted, does not always result into a simple and accurate regular expression.

RegexGenerator++ Bartoli et al. (2016) infers regular expression using a genetic algorithm that matches selected substrings of a text, but assumes that the non-selected part are implicit negative examples.

AlphaRegex Lee et al. (2016) maintains a population of regular expression-like elements that contain placeholders, and updates this population using a search algorithm that considers the most promising elements. AlphaRegex suffers from two limitations. First, it does not take into account the accuracy of the output regular expression, which can lead to inaccurate solutions. Second, it is designed for a binary alphabet, which makes it unsuitable for most of practical use cases.

All the aforementioned related works suffer from the same limitations: they all require negative examples to output relevant results. Moreover, these algorithms are not designed to take into account patterns supplied by the end-user. In that sense, no prior work seems to address the problem we aim to solve.

Finally, many works try to directly infer parsing rules given some input file examples. In particular, Zhu et al. (2019) presents several works designed for log parsing, such as Logmine Hamooni et al. (2016), Spell Du and Li (2016) and Drain He et al. (2017). The core idea of these approaches consist in forming clusters of homogeneous log lines and then infer for each cluster a skeleton involving fixed strings and wildcards using a greedy heuristic. Unfortunately, by design, such skeletons are unable to detect disjunctions, repetitions, or optional fields.

3. Notations and definitions

Unless otherwise specified, all words, regular expressions and automata in this paper are defined over a finite alphabet Σ. We denote the Kleene star by *, so the set of all words over Σ is Σ^*, and any language defined over Σ is a subset of Σ^*.

The *sample* of positive examples is denoted by $W = \{w_1, w_2, ..., w_n\}$. The i^{th} positive example of W is denoted by w_i and its j^{th} character is denoted by $w_i[j]$.

A *regular expression* is a string involving symbols from Σ and meta-characters which correspond to the parentheses and the usual regular expressions operators $\{\cdot, |, +, *, ?\}$. Equation 1 recalls their definition, where $\mathcal{L}(r)$ (resp. $\mathcal{L}(r')$) refers to the regular language induced by a regular expression r (resp. r') and ε denotes the empty word:

$$\begin{array}{lll} \mathcal{L}(r|r') = \mathcal{L}(r) \cup \mathcal{L}(r') & \mathcal{L}(r \cdot r') = \mathcal{L}(r)\mathcal{L}(r') & \\ \mathcal{L}(r?) = \mathcal{L}(r) \cup \{\varepsilon\} & \mathcal{L}(r*) = \mathcal{L}(r)^* & \mathcal{L}(r+) = \mathcal{L}(r) \cup \mathcal{L}(r*) \end{array} \tag{1}$$

An *Abtrast Syntax Tree (AST)* is a (binary) tree that allows to represent an expression (in our case, a regular expression). Each leaf is labeled by a symbol of Σ, and each internal node is labeled with a regular expression operator. The internal nodes labeled by a unary operator (i.e., $+, *$, and ?) have exactly one child. The internal nodes labeled with a binary operator (i.e., $\cdot$ and $|$) have exactly two children. Our ASTs comprise an additional root node $\perp$, introduced for convenience. Figure 1 depicts a small AST.

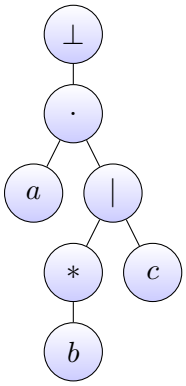


Figure 1: AST of $a \cdot (b* \mid c)$, where $\perp$ corresponds to the root of the AST.

We denote by $|A|$ the number of nodes in an AST A. Given a node u of A, we denote by $\mathrm{p}(u)$ the parent node of u, and by $\mathrm{c}(u)$ (resp $\ell(u)$, resp. $\mathrm{r}(u)$) the only child (resp. left child, resp. right child) of u. We denote by $\mathrm{d}(e)$ the *direction* of an arc $e = (u, v)$, defined by $\uparrow$

(resp. $\downarrow$) if u is the child (resp. parent) of v; e is then said to be *upward* (resp. *downward*). $\mathcal{L}(A)$ denotes the regular language represented by A.

Given a node u of an AST A, $\lambda(u)$ denotes its label. By convention and for sake of simplicity, we denote by $\perp$ the label of the root. If $u \neq \perp$ and is a leaf, then $\lambda(u) \in \Sigma$. If $u \neq \perp$ is an internal node, then $\lambda(u) \in \{\cdot, |, ?, *, +\}$.

4. Problem formulation

Our goal is to find a useful regular expression from a set of positive examples. For instance, if $\Sigma = \{a, b, c\}$ and $W = \{abc, abcabc, abcabcabc\}$, two trivial solutions can be found in $O(|W|)$, namely $(a \mid b \mid c)^*$ and $(a{\cdot}b{\cdot}c \mid a{\cdot}b{\cdot}c{\cdot}a{\cdot}b{\cdot}c \mid a{\cdot}b{\cdot}c{\cdot}a{\cdot}b{\cdot}c{\cdot}a{\cdot}b{\cdot}c)$. The first one recognizes any word in Σ^*. The second is neither concise nor capable of generalizing the sample W. As a consequence, the practical utility of these two solutions is limited. A solution like $(a \cdot b \cdot c)^+$ is much more relevant, as it is concise and generalizes W while remaining quite accurate. To this end, we introduce an objective function f that balances between *accuracy* and *concision* defined by:

$$\begin{gathered} f(A) = \alpha{\cdot}\frac{1}{N}{\cdot}|A| + (1-\alpha){\cdot}\sum_{k\in\mathbb{N}} \beta_k \frac{|L_k|}{|\Sigma|^k} \\ \text{where } N = 1 + n + \sum_{k=1}^{n} |w_k|,\ L_k = \{w, w \in \mathcal{L}(A) \wedge |w| = k\} \end{gathered} \tag{2}$$

In Equation 2, α parametrizes the relative importance of conciseness (1^{st} term of the sum) versus precision (2^{nd} term of the sum). The series β is positive, sums to 1, and is introduced to mitigate the bias inherent to the high values of k. The *conciseness* is a value in $[0, 1]$; the smaller the accuracy, the smaller the AST. The factor $1/N$ guarantees that the conciseness is in practice always in $[0, 1]$. Indeed, N corresponds to the size of the second trivial solution, and fAST never considers a solution that is simultaneously less precise and less concise. The *precision* is also a value in $[0, 1]$; the lower the precision, the stricter the language.

Given α and W, we aim at finding an AST A that minimizes f such that $W \subseteq \mathcal{L}(A)$.

5. fAST

5.1. Overview of the algorithm

fAST is a search algorithm which maintains a population of solutions under construction, called *partial solutions*. fAST processes W character by character and updates the population of solutions consequently. When a word of W is processed, fAST moves to the next word of W. Each partial solution is characterized by a tuple (A, e, i, j), where:

- A denotes the AST related to the partial solution;
- i and j indicate the *progression* of the solution: it means the last character of W processed by this partial solution is the j^{th} character of the i^{th} word of W.

- e is an arc of A, called *active arc*, which constrains where the next modification of A can be done (intuitively, e maps the progression (i, j) in A.);

The partial solutions are stored in a priority queue based on CBFS Morrison et al. (2017) and ordered according to f . We initialize the priority queue by inserting the AST consisting of its root node and a leaf labeled $w_1[1]$. Its active arc e is the arc connecting the leaf to the root. Its progression is set to $(i, j) = (1, 1)$. At each iteration, fAST processes the partial solution (A, e, i, j) having the highest priority and updates it locally according to e and $w_i[j]$. To obtain new resulting partial solutions, fAST determines where (see Section 5.2) and how (see Section 5.3) to update A. This transformation, called *complete mutation*, updates A and e.

If a partial solution has fully processed W (i.e., if $(i, j) = (n, |w_n|)$), then this is a *candidate solution* to our problem. This candidate solution is not inserted in the priority queue. If moreover, it improves the best candidate solution found so far (according to f), we memorize it. If the priority queue is empty (or if the algorithm is interrupted by the user[2]), we return the best candidate solution.

Otherwise (i.e. if some characters of W are not yet processed), the partial solution must be inserted in the priority queue. If $j \leq |w_i|$, its progression becomes $(i, j+1)$ and its active arc is returned by the mutation. If $j = |w_i| + 1$, its progression becomes $(i + 1, j)$ and its active arc is reset to the out-arc of $\perp$.

5.2. Navigating in an AST

This section explains how to walk in an AST A when matching it against an arbitrary word.

We call *valid path* of A any path π that conforms with regular expression semantics, i.e., with the following properties: for all node u of the path:
if $u = \perp$, π traverses exactly once its sub-AST;
if $\lambda(u) =?$, π traverses 0 or 1 time the sub-AST rooted in u;
if $\lambda(u) = *$, π it traverses 0 or more times the sub-AST rooted in u;
if $\lambda(u) = +$, π it traverses 1 or more times the sub-AST rooted in u;
if $\lambda(u) = \cdot$, π it visits each child sub-AST of u from left to right;
if $\lambda(u) = |$, π it visits each exactly one sub-AST of u;
if $\lambda(u) \in \Sigma$, the next node traversed by π is the parent node of u.

More formally, these navigation rules are gathered in Table 1, where $\mathrm{s}(u, v)$ denotes the set of possible successors just after π traverses an arbitrary (u, v) arc.

We call *induced word* the word obtained by concatenating the symbols labeling the sequence of leaves traversed by π. We denote this word by $\mathcal{W}(\pi)$.

For example, in Figure 1, the path that consecutively traverses the nodes labeled $\perp$, $\cdot$, a, $\cdot$, $|$, $*$, b, $*$, b, $*$, $|$, $\cdot$, $\perp$ is a valid path and it induces the word *abb*.

We are especially interested in valid paths that induce the empty word. Indeed, such paths reveal where the next mutation can apply.

Definition 1 (ε-reachable) *Let e be an arc in an AST A. An arc e' of A is* ε-reachable *from e if and only if there exists a valid path starting with e and ending with e' inducing the empty word (note that possibly, $e = e'$).*

2. In that sense, fAST is an anytime algorithm.

$\mathrm{d}(e)$	$\lambda(v)$	$\mathrm{s}(u,v)$
$\downarrow$	$+$	$\{\mathrm{c}(v)\}$
$\downarrow$	$*$ or $?$	$\{\mathrm{c}(v), \mathrm{p}(v)\}$
$\downarrow$	$\cdot$	$\{\ell(v)\}$
$\downarrow$	$\mid$	$\{\ell(v), \mathrm{r}(v)\}$
$\downarrow$	$a \in \Sigma$	$\{\mathrm{p}(v)\}$
$\uparrow$	$?$	$\{\mathrm{p}(v)\}$
$\uparrow$	$*$ or $+$	$\{\mathrm{c}(v), \mathrm{p}(v)\}$
$\uparrow$	$\mid$	$\{\mathrm{p}(v)\}$
$\uparrow$	$\cdot$	$\{\mathrm{r}(v)\}$ if $u = \ell(v)$, $\{\mathrm{p}(v)\}$ if $u = \mathrm{r}(v)$

Table 1: Valid successors of v when it is reached from the (u, v) arc.

We denote the set of the ε-reachable arcs from e by $E_\varepsilon(e, A)$.

5.3. Mutations

Given a partial solution (A, e, i, j), this section explains how fAST derives the resulting new partial solutions (called *mutants*) when processing next character of W.

Each mutant results from a composition of mutations, called *complete mutation*. A *mutation* is a function that takes in parameter a pair (A, e) where A is the AST and e the active arc of a partial solution. It returns a pair (A', e') where A' refers to the updated AST and e' to the updated active arc. The progression (i, j) is updated as explained in Section 5.1.

A *mutation* always operates on a modifiable arc $\tilde{e}$, where $\tilde{e} \in E_\varepsilon(e, A)$. We denote by A' the AST obtained once a mutation is applied and by e' its active arc.

A *complete mutation* consists in zero or more *preliminary mutations* and ends with one *simple mutation*[3]. Intuitively, a simple mutation injects in the AST A several nodes to take into account $w_i[j]$ according to e (see Section 5.3.1).

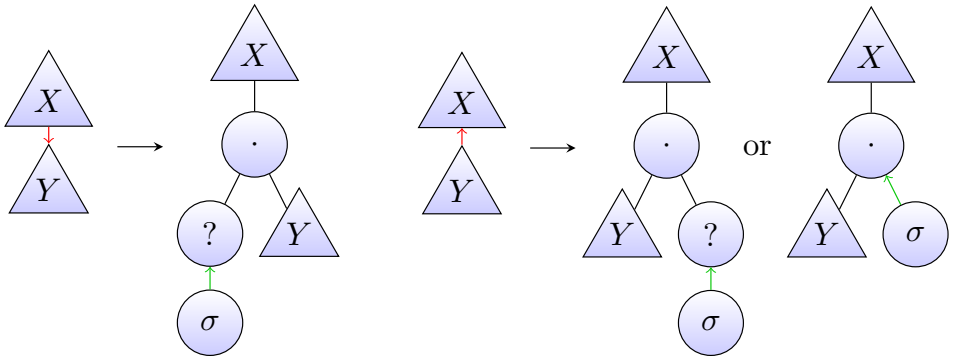
A *preliminary mutation* modifies the set of reachable arcs by inserting a node labeled by ? or + (see Section 5.3.2). Intuitively, preliminary mutations are required to alter $E_\varepsilon(e, A)$ so that fAST can apply each simple mutation at every possible places.

5.3.1. Simple mutations

A simple mutation inserts zero or more nodes in an AST A so that it takes into account the last processed character $\sigma = w_i[j]$. If it inserts nodes, this includes a leaf labeled by σ. The active arc resulting from a simple mutation always starts from the added leaf. As these mutations involve a new leaf, they necessarily involve a binary operator, and hence comprise $\cdot$-mutations and $\mid$-mutations.

Definition 2 ($\cdot$-mutation) *If $\tilde{e} \in E_\varepsilon(e, A)$, then a $\cdot$-mutation can be applied (see Figures 2). We distinguish two cases, depending on the direction of $\tilde{e}$.*

3. Note if the complete mutation involve no preliminary mutation, once A' is simplified (see Section 5.4), we may have $A = A'$.



(*a*) Left ·-mutation. (*b*) Right ·-mutation.

Figure 2: Diagram summarizing the possible ·-mutations.

If $d(\tilde{e}) = \downarrow$*, then the mutant solution* (A', e') *is obtained from* (A, e) *as follows: (i) a node* u' *labeled by* $\cdot$ *is added and* (u, v) *is replaced by* (u, u') *and* (u', v)*; (ii) a new node labeled by* ? *and a new leaf* ℓ *labeled by* σ *are appended as a left child of* u'*.*

If $d(\tilde{e}) = \uparrow$*, then the mutant solution* (A', e') *is obtained from* (A, e) *as follows: (i) a node* v' *labeled by* $\cdot$ *is added and* (v, u) *is replaced by* (v, v') *and* (v', u)*; (ii) a new leaf* ℓ *labeled by* σ *is appended as a left child of* u'*; (iii) a node labeled* ? *is added if and only if* Y *has already been traversed twice or more when processing the previous characters of* W*.*

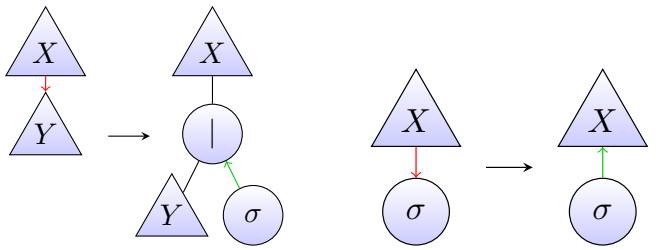
Figure 2 depicts the ·-mutations considered in fAST. On each diagram: the left (resp. right) AST corresponds to the AST before (resp. after) applying the mutation; each circle represents a node inserted by the mutation; each triangle corresponds to an unmodified sub-AST part of the original AST; the red arc is the modifiable arc $\tilde{e}$ where the mutation operates; the green arc corresponds to the new active arc e' resulting from the mutation.

As $\cdot$ is not commutative, it is important to distinguish if the leaf is appended to the left or to the right of Y. These two cases correspond to Figures 2(*a*) and 2(*b*). Remember that if a valid path traverses a sub-AST rooted in a node labeled by $\cdot$, it must traverse each of its sub-AST (see Section 5.2). That is why it is important to ensure that the inserted sub-AST does not contradict the walk induced by previous characters of w_i and the previous positive examples. Hence, a node labeled by ? may be inserted in best effort.

The additional node labeled by ? is inserted if and only if Y has been traversed twice or more[4], either when processing a previous positive example of W, or when processing the previous characters of w_i. By doing so, we guarantee that A' accepts the prefix of $w_i[1] \dots w_i[j]$ and $\forall i' < i, w_{i'} \in \mathcal{L}(A')$. For example, if $W = \{abc\}$, we expect to infer $a \cdot b \cdot c$ (where Y corresponds to ab and is traversed once when processing W) and the node labeled by ? is not required. On the contrary, if $W = \{abab, abc\}$, we expect to infer $(a \cdot b \cdot c?)+$ and the node labeled by ? is required.

Definition 3 (|-mutation) *If* $d(\tilde{e}) = \downarrow$ *and* $\tilde{e} \in E_\varepsilon(e, A)$*, then the* |*-mutation can be applied. The mutant solution* (A', e') *is obtained from* (A, e) *as follows: (i) a node* u' *labeled*

4. Another option consists in creating the mutant without the node labeled by ?. If it matches $\{w_1 \dots w_i\}$, we keep this mutant, otherwise, we create the other one instead.



(a) |-mutation. (b) Identity mutation.

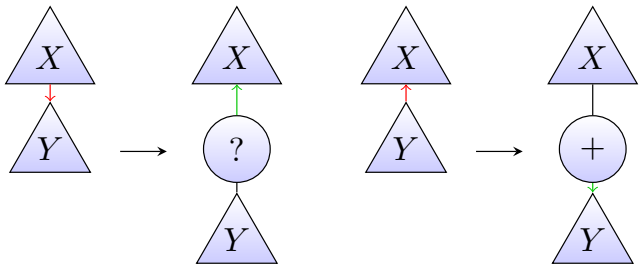
Figure 3: The | and the identity mutations.

by | is added and (u, v) is replaced by (u, u') and (u', v); (ii) a new leaf ℓ labeled by σ is appended so that it is a child of u'. The returned value is $((\ell, p(\ell)), A')$.

Definition 4 (identity mutation) *If $d(\tilde{e}) =\downarrow$ and $\tilde{e} \in E_\varepsilon(e, A)$ and $\tilde{e}$ reaches a leaf ℓ labeled by σ, then the identity-mutation can be applied. The returned value is $((\ell, p(\ell)), A')$.*

Figure 3 illustrates the |-mutation by following the same conventions as in Figure 2. This mutation is simpler than the ·-mutation for two reasons. First, contrary to ·, a node labeled by | does not impose to traverse the newly appended sub-tree, and thus, introducing a node labeled by ? is not required. Second, | is commutative, so there is no need to distinguish whether the leaf is appended to the left of to the right of Y.

5.3.2. Preliminary mutations



(a) ?-mutation. (b) +-mutation.

Figure 4: Preliminary mutations involved in fAST.

In order to get an exhaustive exploration, fAST must apply zero or more *preliminary mutations* on A before applying a simple mutation. Preliminary mutations allows to consider every possible places where the new leaf may be appended.

Like simple mutations, preliminary mutations always operate on a modifiable arc $\tilde{e} = (\tilde{u}, \tilde{v})$. We additionally impose that $(\tilde{v}, \tilde{u})$ is not modifiable, because it prevents generating sub-optimal solutions and thus limits the combinatorial explosion. Each preliminary mutation inserts exactly one node either labeled by + or by ? and such that: it extends the set of modifiable arcs ($E_\varepsilon(e, A) \subseteq E_\varepsilon(e', A')$) ; $\mathcal{L}(A) \subseteq \mathcal{L}(A')$; and no leaf is inserted.

Definition 5 (+-mutation) *If* $d(\tilde{e}) = \uparrow$, $(\tilde{u}, \tilde{v}) \in E_\varepsilon(e, A)$ *and* $(\tilde{v}, \tilde{u}) \notin E_\varepsilon(e, A)$ *then the* $+$*-mutation can apply. The mutant solution* (A', e') *is obtained from* (A, e) *as follows: a node* u' *labeled by* $+$ *is added and* (u, v) *is replaced by* (u, u') *and* (u', v).

Figure 4(*b*) illustrates the +-mutation. Intuitively, this mutation allows to bounce on the inserted node and traverse once more the sub-tree Y

Definition 6 (?-mutation) *If* $d(\tilde{e}) = \downarrow$ *and* $(\tilde{u}, \tilde{v}) \in E_\varepsilon(e, A)$ *and* $(\tilde{v}, \tilde{u}) \notin E_\varepsilon(e, A)$, *then the* ?*-mutation can be applied. The mutant solution* (A', e') *is obtained from* (A, e) *as follows: a node* u' *labeled by* ? *is added and* (u, v) *is replaced by* (u, u') *and* (u', v).

Figure 4(*a*) illustrates the ?-mutation. Intuitively, this mutation allows to bounce on the inserted node to reach its downward arcs.

It worth to note there is no need to consider a $*$-mutation: nodes labeled by $*$ naturally appear when simplifying the AST (see Section 5.4).

5.4. Mitigating the combinatorial explosion

An AST A generates as many mutants as one can construct complete mutations. As the number of possible complete mutations is huge, this combinatorial explosion must be carefully mitigated to make fAST scalable.

Two mutants (A, e, i, j) and (A', e', i', j') are said to be equivalent if and only if $(A = A') \wedge (e = e') \wedge (i = i') \wedge (j = j')$.

It is worth to understand that two equivalent mutants may be obtained from the same or distinct partial solution(s) and by applying the same or a distinct complete mutation. Thus, detecting equivalents mutants helps to mitigate the combinatorial explosion inherent to the fAST algorithm, as there is no need for fAST to process equivalent mutants multiple times. To this end, fAST rely on the following implementation tricks. First, each mutant is simplified according to the notable identities listed in Equation 3. Second, a mutant is inserted in the priority queue if and only if it has not yet been already inserted since fAST has started. Third, to ease AST comparisons, we take advantage of the associativity of $\cdot$ and $|$ to manipulate and compare n-ary ASTs instead of binary ASTs. For example, the ASTs corresponding to $(a \cdot b) \cdot c$ and $a \cdot (b \cdot c)$ both corresponds to $(a \cdot b \cdot c)$, and it is enough to only consider $(a \cdot b \cdot c)$. Fourth, to prevent redundant mutants, we exploit the commutativity of $|$ as follows. For each node labeled by $|$, we sort its children by a same arbitrary order [5].

$$r{+}? = r?{+} = r*, \quad r** = r*, \quad r{+}{+} = r+, \quad r?? = r?, \quad r \mid r = r, \tag{3}$$

6. Regular expression inference using custom patterns

This section describes how to extend the fAST algorithm so that it builds regular expression based on patterns $\mathcal{P}$ defined by the end-user. Such patterns are regular languages defined over Σ, and typically corresponds to usual data types (e.g;, integers (P_{int}), floating numbers

5. A possible order consists in building the string made of the labels involved in a sub-AST using a pre-order tree traversal and order them according to the lexicographic order.

(P_{float}), timestamps, system paths, network addresses, etc). By doing so, fAST becomes a tool able to infer practical regular expression.

In general, there are several ways to decompose a string according to the patterns in $\mathcal{P}$, and deciding which composition is the right one is not straightforward or even possible. To cope with this problem, Raynal et al. (2022b) introduces a data structure, called *pattern automaton (PA)*. A PA is a finite automaton using $\mathcal{P}$ as an alphabet; each path from its root to its sink represents a possible decomposition of a word according to $\mathcal{P}$. In the PA of a word w built according to $\mathcal{P}$, there are $|w|+1$ states where state 0 corresponds to the beginning of the word and state j to the j^{th} character. A transition from the state j to the state j', labeled by $P \in \mathcal{P}$ exists if and only if $w[j+1]...w[j'+1] \in P$. Finally, one can discard from this automaton some irrelevant transition, e.g. to make the PA deterministic or to consider the largest possible pattern matches. By doing so, one can represent a text at the pattern level rather than at the character level, as shown in Figure 5. Note that contrary to Raynal et al. (2022b), fAST does not require the PAs to be deterministic. To support PAs in fAST, several modifications are required.

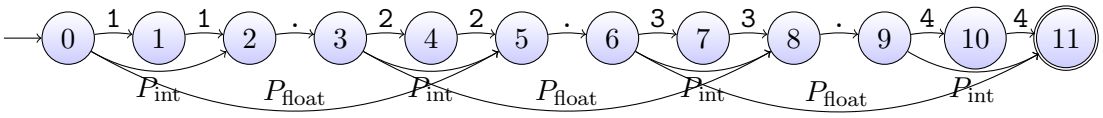


Figure 5: Toy example of pattern automaton representing the string `11.22.33.44`. Here, pattern-based arcs are filtered as follows: an arc (u, v) labeled by $P \in \mathcal{P}$ is discarded if there exists an arc (u', v) labeled by $P \in \mathcal{P}$ such that $(u' < u \wedge v \leq v') \vee (u' \leq u \wedge v < v')$.

First, we extend the alphabet Σ so that it now comprises a new set of symbols plus one extra meta-character per pattern in $\mathcal{P}$. As a result, the new alphabet is $\mathcal{P} \cup \{\{a\}, a \in \Sigma\}$.

Second, for each positive example involved in W, we build the corresponding PA, resulting in n PAs. When processing a partial solution (A, e, i, j), fAST processes each out-transition of the node j of the i^{th} PA (if any). For each (j, k) out-transition, fAST computes the corresponding mutants (just like fAST was processing a single character) and the corresponding progression becomes (j, k). If j has no out-transitions (i.e., if $j = |w_i|$), the progression becomes $(i+1, 0)$ if there is a next positive example (else, the algorithm stops).

Third, we also need revisit how f is evaluated, as leaves related to a pattern are related to a set of words. To do so, when computing $f(A)$ we replace the leaves representing a pattern by their corresponding AST. Nonetheless, it is important to keep A as it is in the priority queue so that fAST does not alter the pattern modeled by each of these leaves in the future mutations.

7. Experimental validation

To evaluate the relevance of our approach, we proceed to an experimental evaluation of fAST. Our experiments have been realized on a 24-core AMD Ryzen @3.9 GHz CPU and 32GB of RAM. Our implementation is publicly available on GitHub Maxime Raynal (2023).

We generate a sample of 300 regular expressions involving between 4 and 12 characters (symbols or meta-characters) on an alphabet of size $|\Sigma| = 6$. For each generated regular expression r_0, we randomly draw a sample of up to 500 distinct words.

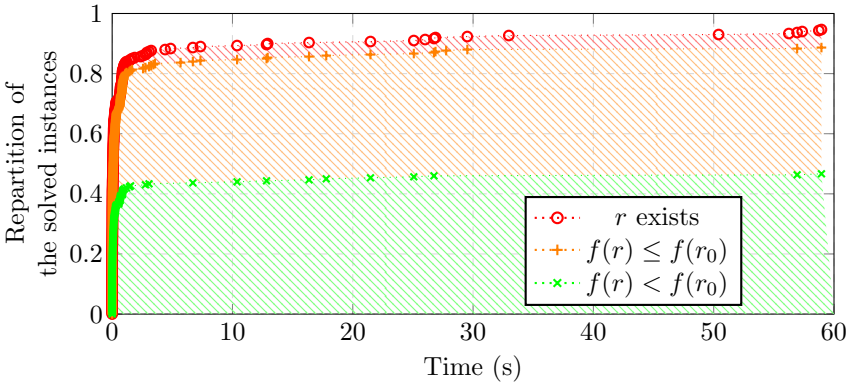


Figure 6: Fraction of solved instances versus time

Figure 6 summarizes the results obtained by running fAST with $\alpha = 0.5$. This experiment shows that in less than one second, fAST found in 80% of the instances a result at least as good (including 40% strictly better) than r_0 and 89% after 60s. In less than a second, fAST finds a regular expression recognizing all the examples of the sample for 83% of the instances (r exists). 80% of the instances are solved with a result at least as good as r_0 ($f(r) \leq f(r_0)$) and even strictly better in 40% of cases ($f(r) < f(r_0)$). After one minute, fAST finds a solution for 95% of the instances, as good as r_0 in 89% of the cases, and even strictly better in 46% of the cases.

8. Conclusion

In this paper, we presented fAST, an algorithm capable of inferring a regular expression that generalizes a sample of examples. Unlike its predecessors, fAST can find a regular expression that is both relevant, accurate, concise and does not require counterexamples. As fAST can incorporate custom patterns (e.g. characterizing usual data types), it can build regular expression built on top of them and return relevant regular expressions for parsing tasks. Our experiments conducted on synthetic data demonstrates that fAST performs well in reasonable time.

Acknowledgments

This work has been partially supported by MIAI@Grenoble Alpes, (ANR-19-P3IA-0003). A part of this work has been carried out at LINCS (http://www.lincs.fr). The authors also thank the reviewers for their feedback and suggestions.

References

Pieter W Adriaans, Menno van Zaanen, et al. Computational grammar induction for linguists. *Grammars*, 7:57–68, 2004.

Dana Angluin. Inference of reversible languages. *Journal for the Association of Computing Machinery*, 29(3):741–765, 1982.

John Arundel. *Puppet 3 Cookbook*. Packt Publishing Ltd, 2013.

Florent Avellaneda and Alexandre Petrenko. Inferring dfa without negative examples. In *International Conference on Grammatical Inference*, pages 17–29. PMLR, 2019.

José L Balcázar, Josep Diaz, Ricard Gavalda, and Osamu Watanabe. An optimal parallel algorithm for learning dfa. In *Proceedings of the seventh annual conference on Computational learning theory*, pages 208–217, 1994.

Wolfgang Barth. *Nagios: System and network monitoring*. No Starch Press, 2008.

Alberto Bartoli, Andrea De Lorenzo, Eric Medvet, and Fabiano Tarlao. Inference of regular expressions for text extraction from examples. *IEEE Transactions on Knowledge and Data Engineering*, 28(5):1217–1230, 2016.

Colin De la Higuera. *Grammatical inference: learning automata and grammars*. Cambridge University Press, 2010.

Min Du and Feifei Li. Spell: Streaming parsing of system event logs. In *2016 IEEE 16th International Conference on Data Mining (ICDM)*, pages 859–864. IEEE, 2016.

Kathleen Fisher, David Walker, Kenny Q Zhu, and Peter White. From dirt to shovels: fully automatic tool generation from ad hoc data. *ACM SIGPLAN Notices*, 43(1):421–434, 2008.

Yihan Gao, Silu Huang, and Aditya Parameswaran. Navigating the data lake with datamaran: Automatically extracting structure from log datasets. In *Proceedings of the 2018 International Conference on Management of Data*, pages 943–958, 2018.

Pedro García and José Ruiz. Learning k-piecewise testable languages from positive data. In Laurent Miclet and Colin de la Higuera, editors, *Grammatical Interference: Learning Syntax from Sentences*, volume 1147 of *Lecture Notes in Computer Science*, pages 203–210. Springer, 1996.

Pedro Garcia, Enrique Vidal, and José Oncina. Learning locally testable languages in the strict sense. In *Proceedings of the Workshop on Algorithmic Learning Theory*, pages 325–338, 1990.

E Mark Gold. Language identification in the limit. *Information and control*, 10(5):447–474, 1967.

E Mark Gold. Complexity of automaton identification from given data. *Information and control*, 37(3):302–320, 1978.

Hossein Hamooni, Biplob Debnath, Jianwu Xu, Hui Zhang, Guofei Jiang, and Abdullah Mueen. Logmine: Fast pattern recognition for log analytics. In *Proceedings of the 25th ACM International Conference on Information and Knowledge Management*, pages 1573–1582, 2016.

Pinjia He, Jieming Zhu, Zibin Zheng, and Michael R Lyu. Drain: An online log parsing approach with fixed depth tree. In *2017 IEEE International Conference on Web Services (ICWS)*, pages 33–40. IEEE, 2017.

Colin de la Higuera and Franck Thollard. Identification in the limit with probability one of stochastic deterministic finite automata. In *International Colloquium on Grammatical Inference*, pages 141–156. Springer, 2000.

Cyril Jaquier. Fail2ban home page, 2004. URL https://www.fail2ban.org/wiki/index.php/Main_Page.

Su-Hyeon Kim, Hyunjoon Cheon, Yo-Sub Han, and Sang-Ki Ko. Splitregex: Faster regex synthesis via neural example splitting. 2021.

Abdelkader Lahmadi and Frédéric Beck. Powering monitoring analytics with elk stack. In *9th international conference on autonomous infrastructure, management and security (aims 2015)*, 2015.

Kevin J Lang, Barak A Pearlmutter, and Rodney A Price. Results of the abbadingo one dfa learning competition and a new evidence-driven state merging algorithm. In *International Colloquium on Grammatical Inference*, pages 1–12. Springer, 1998.

Mina Lee, Sunbeom So, and Hakjoo Oh. Synthesizing regular expressions from examples for introductory automata assignments. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, pages 70–80, 2016.

Marc-Olivier Buob Maxime Raynal. fAST github repository, 2023. URL https://github.com/nokia/find-abstract-syntax-tree.

Laurent Miclet. Regular inference with a tail-clustering method. *IEEE Transactions on Systems, Man, and Cybernetics*, 10(11):737–743, 1980.

David R Morrison, Jason J Sauppe, Wenda Zhang, Sheldon H Jacobson, and Edward C Sewell. Cyclic best first search: Using contours to guide branch-and-bound algorithms. *Naval Research Logistics (NRL)*, 64(1):64–82, 2017.

Anil Nerode. Linear automaton transformations. *Proceedings of the American Mathematical Society*, 9(4):541–544, 1958.

José Oncina and Pedro Garcia. Inferring regular languages in polynomial updated time. In *Pattern recognition and image analysis: selected papers from the IVth Spanish Symposium*, pages 49–61. World Scientific, 1992.

Dessislava Petrova-Antonova and Rumyana Tancheva. Data cleaning: a case study with openrefine and trifacta wrangler. In *International Conference on the Quality of Information and Communications Technology*, pages 32–40. Springer, 2020.

Maxime Raynal, Marc-Olivier Buob, and Georges Quénot. A novel pattern-based edit distance for automatic log parsing. In *2022 26th International Conference on Pattern Recognition (ICPR)*, pages 1236–1242. IEEE, 2022a.

Maxime Raynal, Marc-Olivier Buob, and Georges Quénot. A novel pattern-based edit distance for automatic log parsing. In *International Conference on Pattern Recognition*, 2022b.

Marc Sebban, Jean-Christophe Janodet, and Frédéric Tantini. Blue*: a blue-fringe procedure for learning dfa with noisy data. In *Proceedings of the Int Conf on Genetic and Evolutionary Computation*, 2004.

Leslie G Valiant. A theory of the learnable. *Communications of the ACM*, 27(11):1134–1142, 1984.

Moshe Zadka. Ansible. In *DevOps in Python*, pages 167–174. Springer, 2022.

Jieming Zhu, Shilin He, Jinyang Liu, Pinjia He, Qi Xie, Zibin Zheng, and Michael R Lyu. Tools and benchmarks for automated log parsing. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 121–130. IEEE, 2019.

The following appendices try to answer the questions asked by the ICGI reviewers.

Appendix A. The fAST algorithm

Appendix A.1 details the main algorithm (see Section 5.1). Its auxiliary algorithms are detailed in the other appendices. Appendix A.2 details how to list the *ϵ-reachable successors* (see Section 5.2). Appendix A.3 explains how to build the set of *mutants* when processing a symbol of W. Appendix A.3 explains how to *simplify* an AST using Equation 3.

A.1. Main algorithm

Assume that $\varepsilon \notin W$ and $W \neq \varnothing$ (this hypothesis is relaxed in the end of this section). Algorithm 1 shows the pseudo-code of the fAST algorithm presented in Section 5. A_0 denotes the AST made of a root and a leaf labeled $w_1[1]$.

Input: $W = \{w_1 \dots w_n\}$, the positive examples of non empty words.
Input: f, the objective function (cf Section 4).
Data: M, the function that enumerates mutants (cf. A.3).
Data: simplify, the function that simplies ASTs (cf. A.4).
Output: R, a set of ASTs modeling regular expression that recognize the positive examples in W.

```
1  Let Q be a queue always ordered according to f  Let A_0 be the AST made of a root ⊥ and
     a leaf ℓ labeled by w_1[1]  (i, j) ← (1, 1)  e ← (ℓ, p(ℓ))  Q.push(A_0, e, i, j)  while Q ≠ ∅ do
2    (A, e, i, j) = Q.pop()  σ ← w_i[j]  for (A', e') ∈ M(A, e) do
3      A' ← simplify(A')  if A' ∉ Q then
4        if j ≤ |w_i| then
5          j' ← j + 1
6        else
7          i' ← i + 1  j' ← 1  if (c_⊥, ⊥) ∉ E_ε(e', A') then
8            (u, v) ← e'  while v ≠ ⊥ do
9              if (λ(u) = ·) ∧ (u = ℓ(v)) then
10               Insert a node labeled by ? in A' between v and r(v).
11             (u, v) ← (p(u), p(v))
12           simplify(A')
13         e' ← (⊥, c(⊥))
14       if i ≤ |W| then
15         Q.push(A', e', i', j')
16       else
17         R ← R ∪ {A'}
18 return R
```

Algorithm 1: The fAST algorithm

Termination Whenever a mutant is poped from the queue, the progression (i', j') of the subsequent mutants A' is strictly increasing (either $i' > i$ or $j' > j$). As $j \leq |w_i|$ and $i \leq |W|$ and as these upper bounds are finite, Algorithm 1 terminates.

Correctness Each mutation processes an (A, e) pair, such that e is an upward arc starting from a leaf of A or e is the downward out-arc of $\bot$. A mutation always returns an upward active arc from a leaf labeled by σ. Besides, when a positive example is fully processed, the active arc becomes $(\bot, c(\bot))$. Therefore, e is always well-formed after applying a mutation.

We must also guarantee that any mutant A' verifies the following properties. In one hand, $\{w_1, \dots, w_{i-1}\} \in \mathcal{L}(A')$. In the other hand, there exists a valid path from $(\bot, c(\bot))$ to e' in A' which induces the prefix $w_i[1] \dots w_i[j]$.

Intuitively, we must guarantee that once w_i is processed, there is no "trailing characters" expected by A'. For instance, if $W = \{abcd, ab\}$ and A models `a.(b.(c.d))`, when w_2 is fully processed, the suffix `c.d` must become optional, leading to `a.(b.(c.d)?)`.

Line $17 - 23$ serves this goal. As at this step, at least symbol of w_i has been processed, $d(e') = \uparrow$. Among all the nodes between e and $\bot$, only the nodes labeled by $\cdot$ reached from the left may induce trailing characters due to their right child. That is why this block systematically inserts a node labeled by ? on top of its right child: it guarantees the right sub-AST accepts ε. Once this is done, by construction, $(c_\bot, \bot) \in E_\epsilon(e', A')$, and $\{w_1, \dots, w_{i-1}\} \in \mathcal{L}(A')$.

Finally, by construction of the mutations, $\mathcal{L}(A) \subseteq \mathcal{L}(A')$. Thus, the positive examples previously processed remain accepted.

What if $\varepsilon \in W$ or $W = \varnothing$? Algorithm 1 must be slightly adapted to handle this corner case. Let $W' = W - \{\varepsilon\}$. If $W' = \varnothing$, then the only solution is the empty regular expression. Otherwise, if $i = |W| \wedge j = |w_i| \wedge W \neq W'$, a node labeled by ? must be inserted between the root of A and its child (so that $\varepsilon \in \mathcal{L}(A)$), and then A is simplified.

A.2. Navigation in the ASTs

Algorithm 2 lists the ε-successors from an arbitrary active arc e in an AST A.

Input: A, an AST modeling a well-formed regular expression.
Input: $e = (u, v)$, the active arc.
Data: $S_\varepsilon(e', A)$ lists the valid arcs that can follow e' according to Table 1.
Output: $R = E_\varepsilon(e_0, A)$ the set of ε-reachable arcs from e_0 in A.

```
1 Let Q be a stack; Q.push(e_0)  R ← {e}  S ← {u}  while Q ≠ ∅ do
2 |   e ← Q.pop()  for e' ∈ S_ε(e, A) do
3 |   |   (u', v') ← e'  if v ∉ S then
4 |   |   |   R ← R ∪ {e'}  S ← S ∪ {v}  Q.push(e')
5 return R
```

Algorithm 2: Find the ε-reachable arcs.

Termination and complexity Algorithm 2 is a tree traversal restricted to valid paths. Thanks to S, it processes each node at most once, and thus runs in $O(|A|)$.

A.3. Mutants

Algorithm 3 presents a recursive function M that lists the mutants resulting from an AST A and its active arc e when processing the next character σ of W.

As explained in Section 5.3, a mutation starts with zero or more preliminary mutations (corresponding to the first recursive calls of M) and ends with exactly one simple mutation (corresponding to the last recursive call of M). Each preliminary (resp. simple) mutation affects a modified arc $\tilde{e} = (\tilde{u}, \tilde{v})$ which is ε-reachable from the current active arc e.

Frozen arcs To guarantee that Algorithm 3 terminates, we must first explain why it is enough to only consider at most one preliminary mutation (i.e., a +-mutation or a ?-mutation) for each arc of A. As $|A|$ is finite, it implies that (A, e) induces a finite set of relevant mutations.

Intuitively, consider $W = \{ab, ac\}$. There is no need to consider `ab+?+?c` as `ab?c` is enough). From the AST perspective, this observation is reflected as follows. The +-mutator (resp. the ?-mutators) is needed to reach a new set of ε-reachable arcs "bouncing" downward (resp. upward) on the newly inserted node.

Consider an arbitrary arc $\tilde{e}$ that we want to alter using preliminary mutations. Inserting in $\tilde{e}$ two nodes labeled by ? (resp. by +) is useless (as $r?? = r?$ and $r++ = r+$). If e is downward, inserting in $\tilde{e}$ a node labeled by ?, itself parent of a node labeled by +, does not change the set of ε-reachable arcs before modifying A. The same consideration holds if e is upward when inserting a node labeled by + itself parent of a node labeled by ?.

As a result, for each arc in A, there is no need to insert more than one unary node to build A'. To guarantee this, a +-mutator (resp. a ?-mutator) must not modify arcs introduced by a previous preliminary mutation (involved in the current mutation). That is why the +-mutator (resp. a ?-mutator) must *freeze* the arcs it creates. Assuming this information is embedded in A, this does not impact the signature of these mutations. Once A' is built, its arcs A' are unfrozen.

Input: A, an AST modeling a well-formed regular expression.
Input: e, the active arc of A.
Data: σ, the symbol of W being processed.
Output: The corresponding mutants $R = M(A, e)$

$R \leftarrow \emptyset$ **for** $\tilde{e} \in E_\varepsilon(e, A)$ **do**
  if $d(\tilde{e}) = \downarrow$ **then**
    $R \leftarrow R \cup$ |-mutation(copy$(A), \tilde{e}$) $\cup$ left-.-mutation(copy$(A), \tilde{e}$) $\cup$ M(+-mutation(copy$(A), \tilde{e}$)) **if** ($\tilde{v}$ *is a leaf*) $\wedge$ ($\lambda(\ell) = \sigma$) **then**
      $R \leftarrow R \cup$ id-mutation(copy$(A), \tilde{e}$)
  else if $d(\tilde{e}) = \uparrow$ **then**
    $R \leftarrow R \cup M($?-mutation(copy$(A), \tilde{e}$)$) \cup$ right-.-mutation(copy$(A), \tilde{e}$)
return R

Algorithm 3: Find the mutants

Termination Each preliminary mutation alters exactly one arc in A. As each arc of A is altered by at most one preliminary mutation, the set of relevant preliminary mutation compositions is finite. The set of possible simple mutations is finite. Therefore, the set of relevant mutations is also finite and thus Algorithm 3 terminates.

Complexity Each arc in A is impacted either altered by a +-mutation, or ? mutation, or no mutation, which leads to 3 possibilities. Then, a simple mutation is chosen. Therefore,

a mutant A leads to $O(3^{|A|})$ possible mutations. Each simple and preliminary mutation is done in $O(1)$, but requires copying A to not impact the rest of the recursion. Each copy is done in $O(|A|)$. Therefore, Algorithm 3 runs in $O(|A|.3^{|A|})$.

A.4. AST simplification

For each arc $e = (u, v)$ of an AST A, Algorithm 4 checks whether the label of u and v (denoted by $\lambda(u)$ and $\lambda(v)$) match Equation 3. If so, u and v are merged, and the resulting label directly depends on $\lambda(u)$ and $\lambda(v)$. Note that Algorithm 4 updates A in place. This imposes to re-indexing the merged nodes and then fixing their corresponding arc iterators. For sake of simplicity, Algorithm 4 deliberately omits these implementation details.

Input: A an AST modeling a well-formed regular expression.
Input: $E(A)$ denotes the arcs of A.
for $e \in E(A)$ **do**
  $(u, v) \leftarrow e$ $\lambda' \leftarrow \bot$ **if** $(\lambda(u) = *) \vee (\lambda(v) = *) \vee (\{\lambda(u), \lambda(v)\} = \{?, +\})$ **then**
    $\lambda' \leftarrow *$
  else if $\lambda(u) = \lambda(v) = +$ **then**
    $\lambda' \leftarrow +$
  else if $\lambda(u) = \lambda(v) =?$ **then**
    $\lambda' \leftarrow ?$
  if $\lambda' \neq \bot$ **then**
    merge(u, v, A)
return A

Algorithm 4: Find the ε-reachable successors

Termination and complexity Algorithm 4 processes at most once each arc of A, thus it terminates and runs in $O(|A|)$.

Appendix B. Toy example

This section illustrates a small toy example where $W = \{abc, abcabc\}$.

The combinatorial explosion of fAST hardly allows us to depict the whole population of ASTs, so we only focus on the mutants leading to `(a.(b.c))+`.

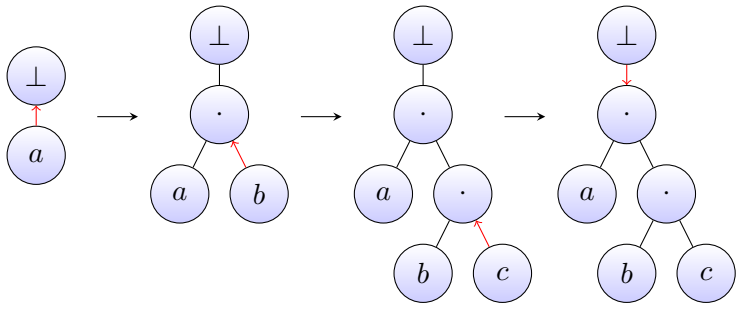


Figure 7: Mutations when processing $w_1 = abc$. .

Figure 7 depicts the mutations applied when processing $w_1 = abc$. The active arc of each AST is colored red. The progression of the three leftmost mutants is respectively $(1,1)$, $(1,2)$, $(1,3)$. The first AST corresponds to A_0. Its leaf is labeled by the $w_1[1] = a$. The second (resp. third) AST is obtained by applying the $\cdot$-mutator on the active arc of the previous AST ($\tilde{e} = e$). These three mutations involve no preliminary mutation: each of them only involves the left-$\cdot$-mutation. Once w_1 is fully processed, the active arc is reset to $(\perp, c(\perp))$ and the progression becomes $(2,1)$, which results to the fourth AST.

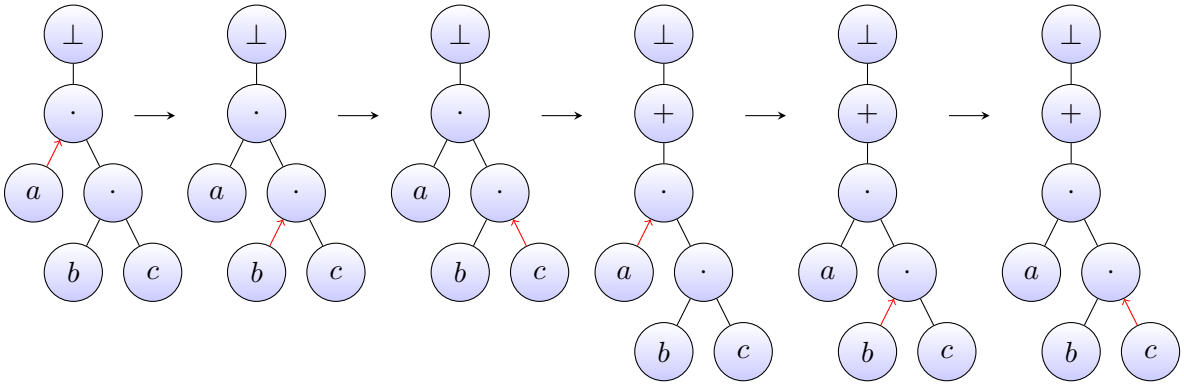


Figure 8: Mutations when processing $w_2 = abcabc$.

Figure 8 depicts the consecutive mutants obtained when processing each character of $w_2 = abcabc$. The progression (i,j) is of these three ASTs is respectively $(2,1), \ldots, (2,6)$.

The three first mutations only involve the identity-mutation, and thus only alter the active arc.

The fourth mutation involves a preliminary mutation (+-mutation) and a simple mutation (identity mutation). The +-mutation applies to the upward arc e' connecting $\perp$ and its child. This arc is indeed ε-reachable from the current active arc e (which is the out-arc of the leaf labeled by c). Thanks to this preliminary mutation, the new active arc becomes the downward out-arc of the node labeled by +. Any (unfrozen) downward arcs reachable from e' become ε-reachable. The simple mutation is the identity mutation and is applied to the downward arc that reaches the leaf labeled by a, and its upward out-arc becomes the new active arc.

The fifth (resp. sixth) mutant is obtained using the identity mutation, which only updates the active arc. Once w_2 is fully processed, the active arc is reset to the out-arc of $\perp$ (not represented in Figure 8).

Appendix C. Supporting custom patterns

Given an arbitrary collection of custom *patterns*, we transform each positive example to its corresponding pattern automaton, as explained in Raynal et al. (2022a). Each pattern is define according to a same alphabet whose symbols are called *characters*. A pattern may possibly only recognize a word made of a single given character.

Consider the word w_i and its corresponding pattern automaton G_i. Each transition of G_i is labeled by a pattern (as shown in Figure 5). The state indices of G_i correspond to the character index (indexed from 0) of the underlying word. Each path from 0 to $|w_i|$ correspond to a possible pattern-based decomposition of w_i. We now explain how to adapt Algorithm 1 to ingest pattern automata instead of words.

To do so, Algorithm 1 requires some modifications:

- *Progression* (i, j). fAST now considers any path from 0 to w_i. j now corresponds to the current node of G_i. When starting to process a pattern automaton, j is set to 0. Then, fAST processes each out-arc (j, j') involved in G_i. If $j' = |w_i|$, then a decomposition induced by G_i has been fully processed, and fAST moves to the next pattern automaton G_{i+1}, if any. If $j' < |w_i|$, the progression is set to (i, j').

- *Objective function* f. As each transition of G_i is labeled by a pattern, so does each AST leaf. But each pattern corresponds to a regular expression (defined on top of the character alphabet). Hence, any pattern-based AST can be translated to its corresponding character-based AST, which gives the opportunity to compute its density and its size.

- *Active arc* e. Similarly, the active arc is only well-defined in the character-based AST. That is why the sub-AST modeling an expended pattern must remain frozen (and its root may be mapped to the represented pattern). Using this representation, we unify in a single data-structure the pattern-based and the character-based ASTs.