



HAL
open science

fAST: comment inférer une expression régulière exploitable à partir d'un petit échantillon d'exemples

Maxime Raynal, Marc-Olivier Buob, Georges Quénot

► To cite this version:

Maxime Raynal, Marc-Olivier Buob, Georges Quénot. fAST: comment inférer une expression régulière exploitable à partir d'un petit échantillon d'exemples. AlgoTel 2023 - 25èmes Rencontres Francophones sur les Aspects Algorithmiques des Télécommunications, May 2023, Cargèse (Corse), France. hal-04087120

HAL Id: hal-04087120

<https://hal.science/hal-04087120v1>

Submitted on 2 May 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

fAST: comment inférer une expression régulière exploitable à partir d'un petit échantillon d'exemples

Maxime Raynal^{1,2†}, Marc-Olivier Buob¹ et Georges Quénot²

¹Nokia Bell Labs, France

²Univ. Grenoble Alpes, CNRS, Grenoble INP, LIG, F-38000 Grenoble France

Les expressions régulières sont omniprésentes dans le monde des réseaux, mais sont rébarbatives à écrire. Cet article présente un nouvel algorithme, fAST (find Abstract Syntax Tree), qui infère une expression régulière à partir d'un petit échantillon d'exemples. Sa force réside dans sa capacité à réaliser cette inférence en l'absence de contre-exemples.

Mots-clefs : Expressions régulières, synthèse de grammaire, arbres de syntaxe abstraite

1 Introduction et état de l'art

Les expressions régulières sont communément utilisées en informatique pour représenter un langage régulier[‡] par une chaîne de caractères. Elles sont omniprésentes dans le monde des réseaux. Elles permettent notamment de définir des règles de filtrage (proxy HTTP, ACL, BGP, etc.). Elles servent aussi à extraire l'information contenue dans des fichiers de données non structurées (par exemple des fichiers de logs).

Gold a montré qu'un algorithme ne peut pas parfaitement identifier un langage régulier avec un nombre fini d'exemples et de contre-exemples [Gol67]. Toutefois, ce problème est dit PAC-learnable (Probably Approximately Correct) [Val84] : plus l'échantillon est important, meilleure est la qualité du langage inféré.

À ce jour, aucun algorithme ne semble permettre d'inférer une expression régulière intéressante à partir d'un petit échantillon ne comprenant pas de contre-exemples. L'algorithme de Gold [Gol78] infère un langage régulier à partir d'exemples et de contre-exemples en s'appuyant sur une table d'observation découlant du théorème de Myhill-Nerode. RPNI [OG92] est une variante de Gold qui agrège de manière opportuniste les états de l'automate caractérisant le langage inféré. Alpharegex [LSO16] construit progressivement une population d'expressions régulières définies sur un alphabet binaire et munies d'espaces réservés. Tous ces algorithmes, de même que leurs variantes, souffrent de la même limitation : en l'absence de contre-exemples, l'expression régulière est beaucoup trop laxiste pour être exploitable en pratique.

Pour répondre à ces limitations, nous proposons un nouvel algorithme appelé fAST (find Abstract Syntax Tree). fAST construit incrémentalement une population d'arbres de syntaxe abstraite (AST) en traitant chaque exemple de l'échantillon caractère par caractère.

2 Notations et définitions

Nous appelons *échantillon* un ensemble fini de mots $W = \{w_1, \dots, w_n\}$, tel que chaque mot est fini, non vide[§], et défini sur un même *alphabet* fini Σ . Nous notons Σ^* l'ensemble des mots finis pouvant être formés à partir de Σ et $|w|$ la longueur d'un mot $w \in \Sigma^*$.

Une *expression régulière* est une chaîne de caractères impliquant des symboles de Σ et des méta-caractères qui correspondent aux parenthèses et aux opérateurs des expressions régulières ($\cdot, |, +, *, ?$).

[†]Une partie de ces travaux a été effectuée au LINCOS (Laboratory of Information, Networking and Communication Sciences), et à MIAI@Grenoble Alpes, (ANR-19-P3IA-0003).

[‡] Un langage régulier est un langage reconnu par un automate fini déterministe.

[§] Cette hypothèse est mineure : on peut toujours filtrer les mots vides de W et compléter la solution inférée pour accepter le mot vide.

Un *arbre de syntaxe abstraite* est un arbre qui permet de représenter une expression (en l'occurrence, une expression régulière). Dans cet article, chaque feuille est étiquetée par un symbole de Σ , et chaque nœud interne est étiqueté par un opérateur. Les nœuds internes étiquetés par un opérateur unaire (+, * et ?) ont exactement un enfant, et les nœuds étiquetés par un opérateur binaire (· et |) ont exactement deux enfants. Nos ASTs ont un nœud racine supplémentaire noté \perp , introduit pour des raisons pratiques. Nous notons $|A|$ le nombre de nœuds impliqués dans un AST A et $\mathcal{L}(A)$ le langage régulier induit par A . La direction d'un arc $e = (u, v)$ est définie par $d(e) = \uparrow$ si u est l'enfant de v et par $d(e) = \downarrow$ si u est parent enfant de v .

3 Formulation du problème

Notre objectif est de trouver une expression régulière exploitable. Si par exemple on pose $\Sigma = \{a, b, c\}$ et $W = \{abc, abcabc, abcabcabc\}$, deux solutions triviales peuvent être trouvées en $O(|W|)$: $(a|b|c)^*$ et $(abc|abcabc|abcabcabc)$. Ces deux solutions ont toutefois un intérêt pratique limité. La première reconnaît n'importe quel mot de Σ^* . La seconde n'est ni concise, ni capable de généraliser l'échantillon. Une solution comme $(a \cdot b \cdot c)^+$ est bien plus pertinente, car elle est concise et généralise W tout en restant assez précise.

Dans ce but, nous introduisons une fonction objectif f qui permet de trouver un juste équilibre entre *précision* et *concision*. f est définie par :

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

Dans cette formule, α paramètre l'importance relative de la concision (1^{er} terme) par rapport à la précision (2nd terme). La *précision* est une valeur dans $[0, 1]$ d'autant plus proche de 0 que le langage est restreint. La *concision* est d'autant plus proche de 0 que l'AST A est petit. Le facteur $1/N$ garantit que la concision est en pratique dans $[0, 1]$, car N correspond à la taille de la seconde solution triviale, et fAST n'est jamais amené à considérer une solution simultanément moins précise et moins concise. Étant donnés α et W , notre problème consiste à trouver un AST A qui minimise f tel que $W \subseteq \mathcal{L}(A)$.

4 fAST

fAST maintient une population de solutions en cours de construction qui mutent et se multiplient à chaque fois que l'on traite le prochain caractère σ de l'échantillon. Ces solutions sont organisées dans une file à priorité. Cela permet à fAST de se focaliser en priorité sur les solutions les plus prometteuses, mais oblige en contrepartie à mémoriser pour chaque solution sa progression dans le traitement de W . Ainsi, chaque solution est caractérisée par un tuple (A, e, i, j) , où A désigne l'AST associé à la solution ; e est un arc de A , appelé *arc actif*, qui contraint où peut avoir lieu la prochaine modification de A ; i et j indiquent la *progression* de la solution (le dernier caractère traité est le $j^{\text{ème}}$ caractère du $i^{\text{ème}}$ mot de W , noté $w_i[j]$).

On initialise la file à priorité en y insérant l'AST constitué d'une racine à laquelle est raccrochée une feuille étiquetée par $w_1[1]$. Son arc actif e est l'arc qui connecte la feuille à la racine et sa progression vaut $(i, j) = (1, 1)$. À chaque itération, fAST traite la solution la plus prioritaire et engendre de nouvelles solutions, obtenues en faisant muter A en fonction de e . Si à l'issue de la mutation, il reste des caractères de W à traiter, ces solutions mutantes sont à leur tour insérées dans la file à priorité. Leur progression est obtenue soit en incrémentant i si $i < |w_j|$, soit en remettant i à un et en incrémentant j si $i = |w_j|$. Si une solution atteint la fin de l'échantillon, elle constitue une solution candidate à notre problème. On vérifie si elle améliore la meilleure solution trouvée jusqu'ici (conformément à f) et si c'est le cas, on la mémorise, ce qui fait de fAST un algorithme *anytime*. Quand la file à priorité est vide, ou si l'algorithme est interrompu par l'utilisateur, on retourne la meilleure solution trouvée.

5 Mutations

Nous appelons *chemin valide* dans un AST tout chemin qui se conforme à la sémantique des expressions régulières : un chemin π doit traverser zéro ou une fois (resp. zéro ou plusieurs fois, resp. une ou plusieurs fois) un sous-AST dont la racine est étiquetée par ? (resp. *, resp. +) ; lorsqu'un nœud étiqueté par · est

atteint, π doit traverser chacun de ses sous-ASTs (de gauche à droite) ; lorsqu'un nœud étiqueté par $|$ est atteint, π doit traverser exactement un de ses sous-ASTs.

Nous appelons *mot induit* le mot formé par les étiquettes des feuilles traversées par un chemin valide π et le notons $w(\pi)$. Si π commence et finit par le nœud racine, alors par construction $w(\pi) \in \mathcal{L}(A)$.

Une *mutation simple* consiste à insérer dans un AST plusieurs nœuds, dont une feuille étiquetée par σ . Un arc \bar{e} est dit *modifiable* si et seulement s'il existe un chemin valide de e à \bar{e} induisant le mot vide. Une mutation simple est toujours réalisée au niveau d'un arc modifiable \bar{e} . L'arc actif qui résulte d'une mutation simple part toujours d'une feuille étiquetée par σ . Les figures 1a, 1b et 1c rassemblent les mutations simples que *f*AST peut provoquer sur un AST, étant donné son arc modifié \bar{e} et sa direction $d(\bar{e})$.

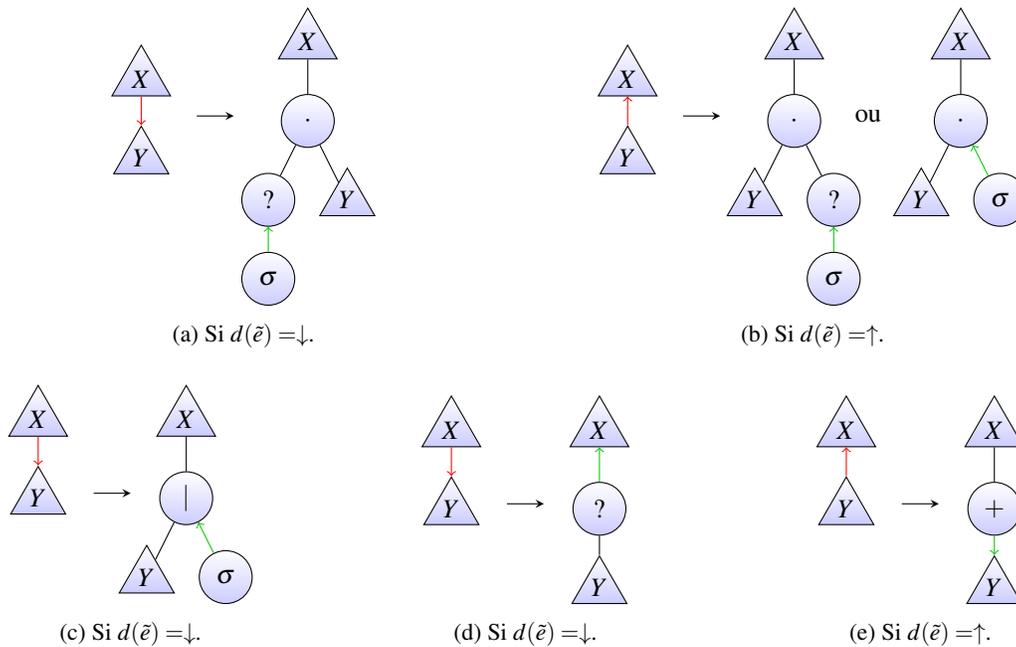


Fig. 1: Sur chacun de ces schémas, l'AST de gauche correspond à l'AST avant, et celui de droite après mutation. L'arc rouge correspond à l'arc modifié \bar{e} impliqué dans la mutation. L'arc vert correspond au nouvel arc actif e résultant de la mutation. Les cercles représentent un nœud inséré au cours de la mutation. Les triangles correspondent à un sous-AST non modifié de taille arbitraire.

Toutefois, afin que l'exploration soit complète, *f*AST peut appliquer zéro ou plusieurs *mutations préalables* sur A . À l'image des mutations simples, les mutations préalables ont toujours lieu sur un arc modifiable $\bar{e} = (\bar{u}, \bar{v})$. Nous imposons de plus que (\bar{v}, \bar{u}) ne soit pas modifiable [¶]. Chaque mutation préalable insère exactement un nœud étiqueté par $+$ ou $?$, de sorte à ce que l'ensemble des arcs modifiables depuis e varie. Les mutations préalables sont représentées sur les figures 1d et 1e.

Une *mutation complète* est une composition mettant en jeu zéro ou plusieurs mutations préalables et se finit par exactement une mutation simple. Un AST A engendre donc autant d'ASTs mutants que l'on peut construire de mutations complètes. Cette explosion combinatoire explique pourquoi il est nécessaire que *f*AST se focalise sur les solutions les plus prometteuses. Les mutations complètes sont construites de sorte à garantir l'existence d'un chemin valide π allant de \perp à la feuille insérée tel que $w(\pi) = w_i[0] \dots w_i[j]$. De plus, elles préservent la propriété $\{w_k, k < i\} \subseteq \mathcal{L}(A)$.

Pour chaque mutant, nous procédons à un nettoyage au niveau des nœuds insérés. En effet, les expressions régulières $r++$ et $r+$ sont équivalentes, donc fusionner deux nœuds voisins étiquetés par $+$ n'altère pas le langage induit par leur AST. Sur le même principe, nous exploitons les identités remarquables suiv-

[¶] Ce faisant, nous évitons d'engendrer des solutions sous-optimales et nous limitons donc l'explosion combinatoire.

antes : pour tout $a \in \Sigma, a | a = a$, et pour tout expression régulière $r : r** = r*, r?? = r?, r+? = r?+ = r*$. C'est cette dernière identité remarquable qui engendre des nœuds étiquetés par $*$ dans l'AST.

Pour éviter de stocker un nombre exponentiel d'ASTs équivalents, nous exploitons l'associativité de \cdot et de $|$ en les représentant par des nœuds n -aires $((a \cdot b) \cdot c = a \cdot (b \cdot c) = (a \cdot b \cdot c))$. Enfin, afin d'améliorer la performance de fAST, nous n'insérons jamais plusieurs fois la même solution dans la file à priorité.

6 Évaluation expérimentale

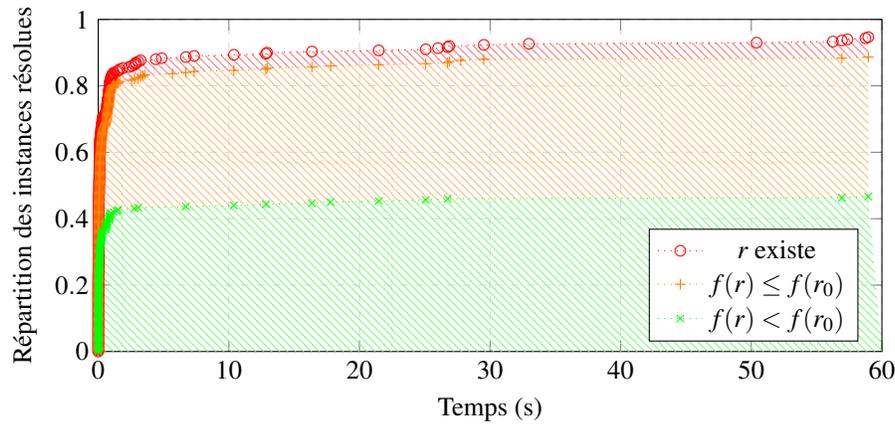


Fig. 2: Fraction des instances résolues en fonction du temps

Afin d'évaluer la pertinence de notre approche, nous procédons à une évaluation expérimentale de fAST. Nous avons généré pour cela un échantillon de 300 expressions régulières impliquant entre 4 et 12 caractères (symboles ou méta-caractères) sur un alphabet de taille $|\Sigma| = 6$. Pour chaque expression régulière générée r_0 , nous tirons aléatoirement un échantillon pouvant aller jusqu'à 500 mots. La figure 2 synthétise les résultats obtenus en lançant fAST avec $\alpha = 0.5$. En moins d'une seconde, fAST trouve une expression régulière reconnaissant tous les exemples de l'échantillon pour 83% des instances (r existe). 80% des instances sont résolues avec un résultat au moins aussi bon que r_0 ($f(r) \leq f(r_0)$) et même strictement meilleur dans 40% des cas ($f(r) < f(r_0)$). Au bout d'une minute, fAST trouve une solution pour 95% des instances, d'aussi bonne qualité que r_0 dans 89% des cas, et même strictement meilleure dans 46% des cas.

7 Conclusion

Dans cet article, nous avons présenté fAST, un algorithme capable d'inférer une expression régulière qui généralise un échantillon d'exemples. Contrairement à ses prédécesseurs, fAST parvient à trouver une expression régulière à la fois pertinente, précise et concise en l'absence de contre-exemple. Notre validation expérimentale montre qu'en pratique, fAST donne de bons résultats en temps raisonnable.

References

- [Gol67] E Mark Gold. Language identification in the limit. *Information and control*, 1967.
- [Gol78] E Mark Gold. Complexity of automaton identification from given data. *Information and control*, 1978.
- [LSO16] M. Lee, S. So, and H. Oh. Synthesizing regular expressions from examples for introductory automata assignments. In *Proceedings of the 2016 ACM SIGPLAN*, 2016.
- [OG92] J. Oncina and P. Garcia. Inferring regular languages in polynomial updated time. In *Pattern recognition and image analysis, IVth Spanish Symposium*, 1992.
- [Val84] Leslie G Valiant. A theory of the learnable. *Communications of the ACM*, 1984.