# Branching Alignment based Synthesis of Regular Expressions

Andreas Scherbakov `andreas@softwareengineer.pro`

The University of Melbourne

**Abstract.** We propose a novel Multiple Sequence Alignment algorithm which is able to build an optimized branching graph given a set of positive matching sample strings. The algorithm is principally based on Minimum Edit Distance approach being applied incrementally. However, we essentially extended the set of edit operations. The newly added operations allow implementing an acyclic graph drawing feature. The feature yields a better visual and appropriate representation of partial alignment gaps between sample strings. For instance, it may produce an `/abc|de[fg]/` regular expression from a set of `"abc", "def", "deg"` strings. For further optimization, we adjust edit penalties based on character class hierarchy. The algorithm may be used in Bioinformatics as well as for extracting Regular Expressions (RE, patterns) from sets of observed or prospective documents (that may be useful in Web data mining, Spam filtering, RE design advisory tools and so on). This paper also considers integration of the proposed algorithm into a full stack Regular Expression extraction flow, particularly, in conjunction to Repetition detection.

**Keywords:** Regular expression, Sequence, Alignment, Multiple Sequence Alignment, Alignment Gap, Regular expression synthesis, Regular expression extraction, Pattern extraction, Hammock graph, Branching Regular Expression, Edit Distance, Character class

## 1 Introduction

Regular expressions (RE) are used widely in various applications for the text data processing. While in many cases the pattern a developer or analyst intends to establish is clear and formally well defined, there are many areas where regular expressions used to recognize and mine a piece of text that is similar to some sample [6]. Examples are Web data mining, Spam filtering, Bioinformatics, State machine analysis and plenty of others. In these cases building and testing a proper RE becomes technically challenging. Algorithms that generate a RE that matches large number of samples are of high demand in these areas. Also, such algorithms may provide priceless help to those application developers who want to obtain necessary RE patterns just by supplying several matching samples rather than by deepening into RE building details [7]. There exist a lot of software tools aiding the regular expression building of blocks and testing but few applications that generate a RE from samples. Examples of the latter presented in [11],[2],[4],[3] employ genetic algorithms or Machine Learning approaches directly to the block composition. In order to take an advantage of such an universal approach, (1) one needs a big bunch of samples, as every possible usage of building block adds a feature while the number of test examples should be sufficient to learn efficiently against that number of features [1]; (2) a set of negative examples should be provided as well as a set of positive ones which often does mean extra human work in designing negative samples. Also, character abstraction rules are mostly beyond user's control.

There are approaches that attempt to build an optimal automaton to satisfy a given input sequence [5]. They do produce regular expressions solely of positive samples. However, they are focused on repeated subsequence extraction while don't mind possible alignments and splitting branches for more neat representation. In this paper we propose an algorithm that performs alignment and partial join of input samples. Please note that the repetition extraction itself is generally beyond our discussion scope here, although being a part of the proposed technique. In a short word, the new algorithm performs fuzzy alignment of samples and produces branches while leaving the repetition extraction to an external method.

Our major motivation was producing regular expressions for advising programmers and users and for assisting human's work in areas where a content schema isn't much strict and clear. Following this, we pay attention to make a RE concise while adjustable both in terms of look and feel and of degrees of freedom (perplexity).

## 2 Representation of Regular Expressions

### 2.1 Hammock Graph

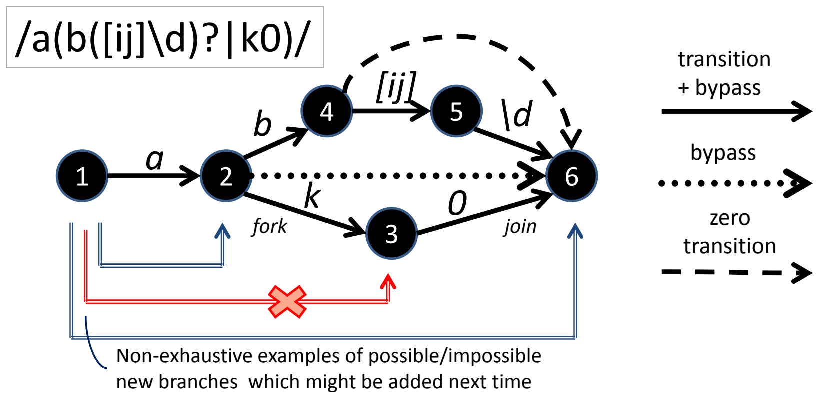

/a(b([ij]\d)?|k0)/
1
2
3
4
5
6
a
b
[ij]
\d
k
0
fork
join
transition + bypass
bypass
zero transition
Non-exhaustive examples of possible/impossible new branches which might be added next time

**Fig. 1.** An example of regular expression graph

For the purposes of our algorithm, a regular expression is represented as a directed acyclic graph (Fig. 1). More precisely, as a "hammock graph" [14], allowing exactly one entrance and one exit to each area. A node of such a graph is referred to as a RE ***position***, while an edge as a RE ***transition*** (there may be normal transitions that correspond to a character place as well as *zero transitions* shunting constructs like **/(abc)?/**). For the simplicity, in the current paper a graph is assumed to be sorted and having all positions enumerated with a continuous $[0..N]$ integer range.

We refer to a sub-graph between some entrance position and some exit position as a ***hammock***. Thus, for example, all edges in a plain chain are hammocks. Hammocks of course may be nested. The way from a hammock's entrance position to it's exit is referred as a ***bypass***. An entrance or exit position shared by multiple hammocks is referred as a ***fork*** or ***join*** position, respectively.

## 2.2 Character Selection

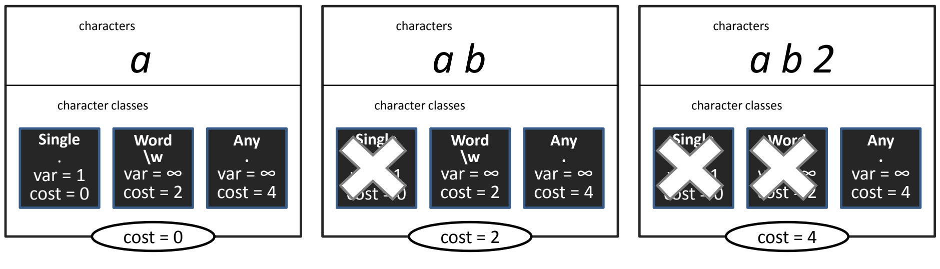


**Fig. 2.** A character selection cost evolution while new characters are merging.

We represent every character place in a RE by a ***character selection***. A character selection is both considered as a concrete character set and as a set of character classes that fit it. A character set in our scope means all characters really falling to this place in some sample string(s). A ***character class*** is en entity that corresponds to some predefined group of characters while also (optionally) restricts the maximum variety of participating characters. For example, a character class with $[a-z]$ character group and maximum allowed variety of 2 matches possible character selection of $[ax]$ but doesn't match the $[axz]$ due to the variety exceeding. The choice of character classes and their parameters essentially affects the resulting regular expression look and actually represents the user preferences that cannot be deducted from a limited number of samples (like whether should we display [123] or $\backslash d$ for a concrete selection of digits at some RE element). Thus, character classes are considered to be user-defined parameters in our algorithm. The only mandatory requirement is that every character should belong to at least one class. Every character class is attributed with a class *cost*. The class cost may be treated as an inverse metric to the prettiness of a class occurrence in a resulting expression record. The cost of a character selection is defined as the minimum cost of a character class containing all characters selected.

$$\Theta_{sel}(chars) = \min_{\substack{cls \in charClasses \\ \forall char \in chars \rightarrow char \in cls.chars \\ cls.maxVarety \geq ||chars||}} \Theta_{class}(cls) \tag{1}$$

When we consider an option to add (merge) a new character to a selection (for example, **/ab/** + **/ac/** → **/a[bc]/** – adding the second character ‘c’ into a position previously occupied by just one character ‘b’), we must calculate the selection cost both before and after the planned merge. That (non-negative) difference will be the *merge cost* used in a dynamic programming procedure described below.

$$\Psi_{merge}(chars, c) = \Theta_{sel}(chars \cup \{c\}) - \Theta_{sel}(chars) \tag{2}$$

As a variant, a character class match may be additionally rewarded by substituting a zero resulting value with some negative one:

$$\text{If} \quad \Psi_{merge} = 0 \quad \text{then} \quad \Psi_{merge} := \Psi_{match} < 0 \tag{3}$$

## 3 Branching Alignment algorithm

In order to produce a regular expression from a set of samples, we first merge the samples into a hammock graph. The core part of our alignment algorithm generally looks like an extension of well known Minimum Edit Distance method [13]. The procedure incrementally transforms the current RE $r$, attempting to align it to the next sample $s$ with the minimum penalty score. We use a dynamic programming algorithm that optimizes the total cost of regular expression modification choosing the best alignment between $r$ and $s$ positions. In order to get a final RE, we start with some input sample as an initial RE and then merge all other samples one by one into it.

*Note.* The resulting expression look may depend on the order of sample string merging. Although the effect of ordering yet needs exploration, our experiments showed that following a longer first, shorter next order ensures an optimal alignment or a good approximation to it in vast majority of cases.

Our algorithm adds a new operation to the original operation set consisting of deletion, insertion and substitution. This new operation enables drawing a new graph branch instead of mere aligning characters between two given positions whenever it seems to yield better score. Table 1 displays the list of possible atomic ***editing events*** available at a given $(i, j)$ combination where $i$ is the target position in $r$ while $j$ is the number of processed characters of $s$.

We associate two cumulative costs, $J_L(i, j)$ and $J_S(i, j)$, to every possible $(i, j)$ pair. Initially, $J_L(0, 0) = 0$ while all other $J_L(i, j)$ and $J_L(i, j)$ are set to $+\infty$. (Subscripts $_L$ ans $_S$ near $J$ stand for the ‘light’ and ‘shadow’ words, respectively). The difference between $J_L$ and $J_S$ lays in the fact that the former immediately correspond to some position in $r$ while the latter points to a bypass

**Table 1.** Editing events

| Operation | Description | Sample string position advance | RE position advance | Penalty |
|---|---|---|---|---|
| Merge | Merging a character from the sample string into a RE character selection | yes | yes | $\Psi_{merge}$ |
| Insert | Adding a new single character element to the RE. As the RE should keep matching all previous samples, the element will be optional | yes | no | $\Psi_{insert}$ |
| Delete | Letting a RE element to be skipped. It means the element becomes optional | no | yes | 0, if the element is already marked optional; $\Psi_{insert}$, otherwise |
| Insert into a branch | Adding a character to a newly created branch | yes | no | $\Psi_{disp}$ |
| Bypass | Drawing a newly created branch over a RE hammock bypass. | no | yes | $\Psi_{branch}$, once per branch |

connection position inside a potential new branch coming "over" $i^{th}$ position in current RE. For example, if $r$ was **/ab/**, $s$ is "cd" and $r$ will be **/(ab)|(cd)/** then $J_S(1,1)$ may refer to a connection point between a bypass "over" 'a' and a bypass "over" 'b'.

We fill matrices of $J_L$ and $J_S$ for all possible positions $[0..M]\times[0..N]$ iterating positions in any convenient non-decreasing order. We use the following equations in order to take all possible edit events at $(i,j)$ point into account.

$$J_L(i,j) = \min \begin{cases} J_L(t.src, j-1) + \Psi_{insert}, \quad \text{if } j > 1 \\ \forall t \in trans(i) \quad J_L(t.src, j) + \Psi_{delete} \\ \forall t \in trans(i) \quad J_L(t.src, j-1) + \Psi_{merge}(t.sel, s[j]), \quad \text{if } j > 1 \\ \forall b \in bypass(r,i) : \neg b.fork \quad J_S(b.src, j) \\ \forall b \in bypass(r,i) \quad J_L(b.src, j) + \Psi_{branch} \\ \forall t \in zerotrans(i) \quad J_L(t.src, j) \end{cases} \tag{4}$$

$$J_S(i,j) = \min \begin{cases} \forall b \in bypass(r,i) : \neg b.join \land \neg b.fork, \quad J_S(b.src, j) \\ \forall b \in bypass(r,i) : \neg b.join \quad J_L(b.src, j) + \Psi_{branch} \\ J_S(i, j-1) + \Psi_{disp}, \quad \text{if } j > 1 \\ J_L(i, j-1) + \Psi_{disp} + \psi_{branch} \quad \text{if } j > 1 \end{cases} \tag{5}$$

As seen from the equation, $J_S(i,j)$ cannot be used or assigned "through" a bypass that shares a fork or join, respectively. This restriction prevents projecting a hammock breaking branch, as discussed in the previous section. Once we consider a new branch creation, we apply a $\Psi_{branch}$ penalty. This penalty (acting like a simplified Graph Edit Distance [8]) enforces the willingness to compose a laconic and uniform regular expression rather than to proliferate branches.

Having the matrix fully constructed, we traverse the lowest total penalty path from $J_L(N, length(s))$ back to $J_L(0,0)$ in order to extract the least costly sequence of editing events. To do this, we reuse the same equations (4), (5) but in place of calculating minimums (that are already known at this stage) we notice the choice that yields minimum value to the left hand side. Then we apply these editing events over elements of $r$ producing a modified RE version $r\prime$ (See Fig. 3 for an example). The new branch insertion is a special case. To add each new projected branch, if any, into the RE, we do the following: (1) outline a branch-related subsequence of editing events (consisting of "Insert into branch" and "Bypass" events); (2) compose a new branch and (3) insert the new branch into $r$ at proper start and end positions.

The idea of algorithm described above may remind the Affine Gap Penalty concept known in Bioinformatics [9] ; the major difference is the fact that we consider the gap not merely as a modifier to an element-by-element edit distance metric, but also as a reason to draw a new graph branch.

See Algorithm 1 illustrating the approach discussed.

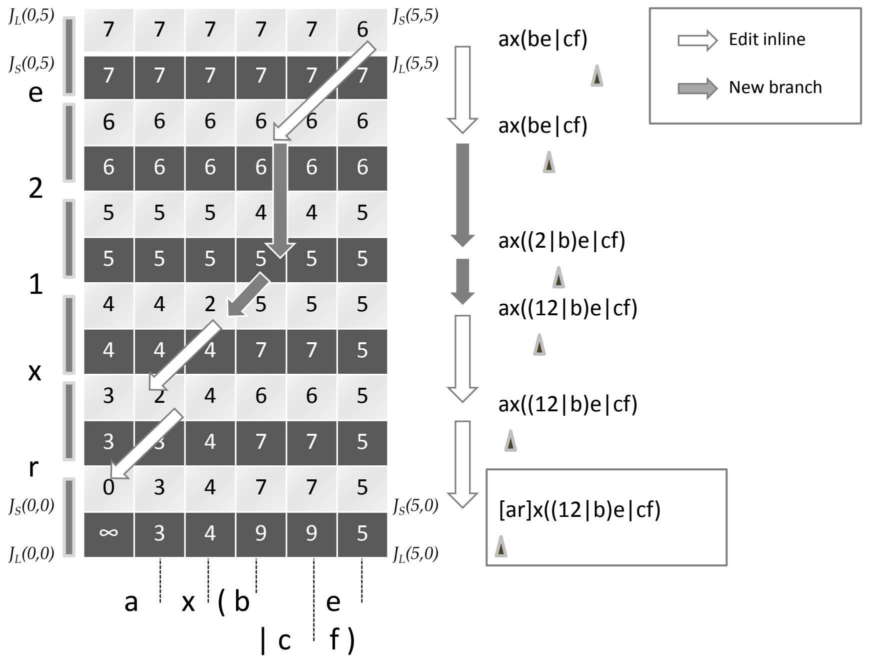


**Fig. 3.** An example: merging "rx12e" string to /ax(be|cf)/ expression

**A note on the complexity.** The algorithm shows the same complexity as Minimum Edit Distance (per one sample), of $O(M*N)$, which is meanly proportional to $N^2$, where $M$ is the regular expression length, $N$ is the sample string length. Still there are many known methods aimed to bound the complexity of Edit Distance minimization, and most of them are directly applicable to the proposed algorithm. Particularly, it might be bounding of maximum edit distance to some constant number [12]

## 4 Repetition Capture

A regular expression produced with the algorithm described above yet needs a repetition capture in order to get its final form. We may employ any known method capable of finding and removing repetitions of adjacent substring in a plain symbol string . Let $M(s) \rightarrow s'$ denote such a method, where $s$ and $s'$ are input and output strings, respectively. Now, we may process the RE graph recursively applying $M$ to all edge sequences of the graph, recursively down up, representing each nested hammock by a single canonical symbol. In more detail, we use a $T : (\alpha, R_{min}, R_{max})$ tuple in order to represent a hammock, where $\alpha$ is a symbol of some alphabet, $R_{min}$ and $R_{max}$ are the minimum and maximum number of $\alpha$'s repetition, respectively. So, we are to slightly modify the $M$ procedure in order to fit the following form.

$$M_T(q) \rightarrow q', \quad q, q' : S, \tag{6}$$

where $S$ denotes the sequence of $T$ tuples. We expect that $M_T$ compares two tuples in a sequence solely relating to the $\alpha$ value (ignoring $R_{min}, R_{max}$). Once has found a repetition, the procedure merges it into single tuple with proper sums for the total $R_{min}$ and $R_{max}$:

$$(\alpha, R_{min1}, R_{max1}), (\alpha, R_{min2}, R_{max2}) \rightarrow (\alpha, R_{min1} + R_{min2}, R_{max1} + R_{max2}) \tag{7}$$

Then, we need a hammock canonical representation function $C(s_i) \rightarrow s_j$. The aim of such a function is to represent a hammock using a minimal variety of $\alpha$. It's expected to satisfy, for instance, the following properties.

- $C(\{x, []\}) = (C(x).\alpha, 0, C(x).R_{max})$
- $C(\{[x]\}) = C(x)$

$C$ function may expect that its arguments already represented in a canonical form.

We initially annotate each character selection $c_j$ found in our graph with a tuple

$$(cHash(c_j), \left\{ \begin{matrix} 0, & \text{if } c_j \text{ marked as optional} \\ 1, & \text{otherwise} \end{matrix} \right\}, 1) : T, \tag{8}$$

where $cHash(c_j)$ is a canonical symbol for the best character class that fits $c_j$. Now we may use a depth-first search procedure like the following one to capture repetitions in our RE graph.

```
procedure CAPTUREREPEAT($q : S$)
    for $e \in q$ do
        if $e \notin$ single char selections then
            for $b \in e.branches$ do
                captureRepeat($b$);
            $b \leftarrow C(b)$;
    $q \leftarrow M_T(q)$
```

Now we have all elements annotated with their repetition bounds $[R_{min} \rightarrow R_{max}]$ and thus we may deliver a RE. However, an user may want to see unbounded repetitions in it (like in /a+/ or /ab*/ ). We cannot rigorously extract an unbounded repetition from a finite number of samples, so we have to use empiric generalization rules. The simplest of them (while working well in practice) may be assigning $R_{max}$ to $+\infty$ if $R_{max} - R_{min} >= R_{bound}$, where $R_{bound}$ is some constant threshold (usually equal to 3).

## 5 Experiments

We applied the proposed algorithm to the Spam classification task [10]. We used 195 series of Spam messages (i.e. 195 mimes). We extracted six sample strings from each message (four headers, a plain text and an HTML). Every series was split into the train and test sets (60% and 40% of messages, respectively). We randomly chose the sets three times per mime. As we don't know in advance which combination of sample strings (like "Subject+body" or "From+Html") better classifies a Spam mime, we have tried all $2^6 - 1 = 32$ combinations and selected one per mime that yields the best F1 metric value once applied for classifying the union of the test set for this mime (Spam) and a standalone collection of non-Spam messages. We have repeated these experiments at various values of the branch cost ratio (that tells one how many avoided highest-cost character merges "worth" a new branch creation).

Fig. 4 displays F1 metric values distribution over mimes. It seems that branch cost should be at least about 3 for the best result. Further increase does not affects the quality significantly.

Fig. 5 shows how the precision and the perplexity correlate to the recall. The precision in most cases equals one and behaves quite irregularly in the remaining cases that demonstrates a quite low probability of matching against some random document. A surprisingly inverse overall correlation in the (perplexity, recall) pair is due to the fact that an increase in perplexity of a generated RE is a reaction to a high content variety of training samples. A congestion near (50, 0.7) coordinate seems to be caused by the generator's attempts to satisfy the sets of altering string constants frequently occurring in Spam messages with the aim to challenge filters. The perplexity of generated regular expression still seems rather low to cover the expected variety of training set samples. Of course being a natural consequence of a synthetic approach to RE building, this fact

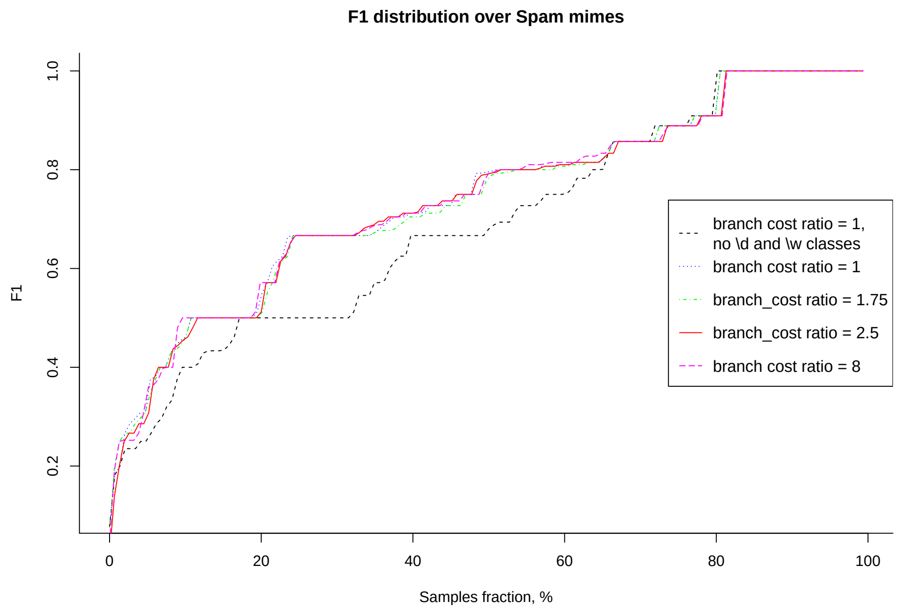


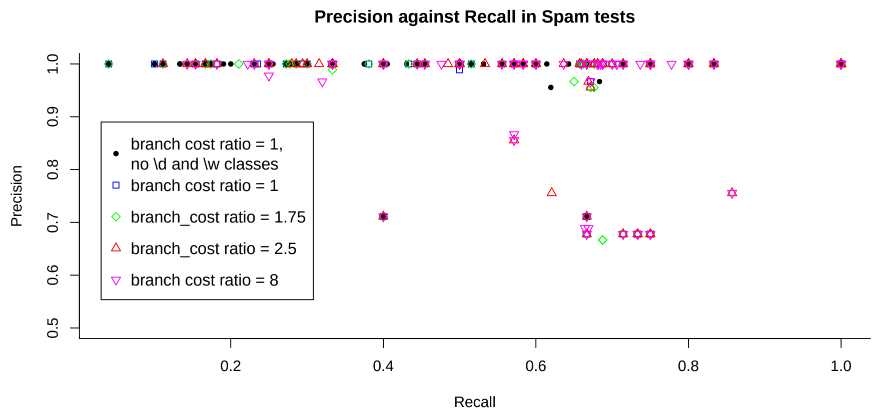
**Fig. 4.** A distribution of F1 metrics observed in a Spam classification test based on RE extraction

indicates that further adjustments toward the perplexity boosting at ”volatile” sample text regions might improve the recall values.

Spam messages provide an extremely challenging test bench because they are specially designed to foolish automated filters as possible. Thus, we didn’t expect the recall to demonstrate as high values as it may be seen in ”normal” content processing. A bunch of expression generated were analyzed and it was found that it’s usually easy to do a manual correction (mostly meaning the removal of some parts) which results in a recall of 95..100%. Thus, the algorithm at it’s today’s stage may be considered as an effective advisory solution for draft RE extraction.

## 6 Manual Assessment

We have found that each piece of a generated RE looks reasonable at a proper selections of major cost and limit parameters. However, the best choice of these parameters may significantly vary through the entire expression. At the same time, the overall alignment of samples works fine at a wide range of parameters. Thus we conclude that for the best advisory value the technique should allow re-compilation of any selected part of generated RE at a different choice of options. Meanly, an optimal cost of branch containing 3 characters is expected to be of



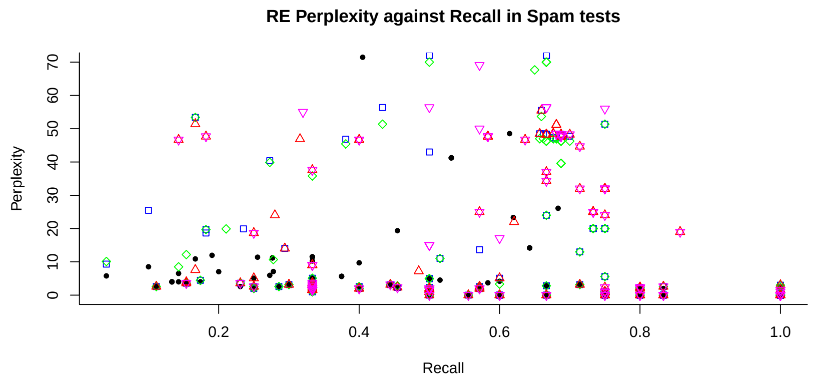


**Fig. 5.** Precision and Perplexity against Recall for Spam tests at various parameters.

about 1.5..2.5 times the highest cost of single character merge (also referred to as the Wild card cost). Table 2 displays examples of regular expressions generated with a set of E-Mail addresses taken from a Spam distribution *"From:"* field. The table also includes manual assessment of expression quality in terms of readability, recall, and precision. Expressions were taken under various settings of three major parameters such as (1) Wild card character class cost (max $\Psi_{merge}$), (2) Branch cost ($\Psi_{branch}$) and (3) Branch Character cost ($\Psi_{disp}$). Also, Match bonus ($-\Psi_{match}$) was varied in some cases.

## 7 Discussion

The algorithm has demonstrated its potential as an advisory solution for an analyst or an engineer willing to compose regular expressions in a semi-automated manner. Tuning of parameters may be really needed to get the desired look of

**Table 2.** Examples of RE extracted at different parameter settings from a set of header field values found in a Spam distribution

| cost | | | | | |
|---|---|---|---|---|---|
| Wild | Branch | | Match | Regular Expression produced | Assessment |
| | Start | Char | | | |
| 4 | 1..6 | 1 | 0 | `(?:Ponto Frio\|(?:Sepho\|Ext)ra\| "?InfoJobs"?)<κ@mail2m\d{1,2}\.info>` | Ok |
| 4 | 7 | 1 | 0 | `(?:Ponto Frio\|InfoJobs\|"InfoJobs" \| (?:Sepho\|Ext)ra) <κ@mail2m\d{1,2}\.info>` | Ok but less readable |
| 4 | 2 | 2 | 0 | `.?\w{3}o?(?:ra\|Jobs"?\| Frio) <κ@mail2m\d?\d\.info>` | too general, less readable |
| 2 | 2 | 2 | 0 | `.?(?:[Io]n)?(?:.{2})?.{5} <κ@mail2m\d?\d\.info>` | too general, less readable |
| 4 | 1 | 3 | -2..0 | `["P]?\wn?\wo?(?:.\w{3})?. <κ@mail2m\d?\d\.info>` | irrelevant |
| 7 | 1 | 4 | -2..0 | `(?:P\|")?\wn?(?:\w{2})?(?: \|\w)\w{3} (?:o\|")? <κ@mail2m\d?\d\.info>` | too general |
| 2 | 10 | 1 | 0 | `(?:.{2})?(?:n[ft]o\|f)?.{5} <κ@mail2m\d?\d\.info>` | low relevancy |
| 2 | 1 | 4 | 0 | `["P]?.n?.o?(?:.{2})?.{3} <κ@mail2m\d?\d\.info>` | too general |
| 3 | 1 | 2 | -2..0 | `.?\w{3}o?(?:ra\|Jobs"?\| Frio) <κ@mail2m\d?\d\.info>` | acceptable still unreadable |
| 4 | 1 | 1 | -2 | `(?:Sep\|E\|(?:Po\|"?I)n)\w\w(?:ra\|Jobs"?\| Frio) <κ@mail2m\d{1,2}\.info>` | acceptable still unreadable |

| Samples | |
|---|---|
| `"InfoJobs" <κ@mail2m37.info>` | `Sephora <κ@mail2m38.info>` |
| `Ponto Frio <κ@mail2m10.info>` | `Sephora <κ@mail2m16.info>` |
| `"InfoJobs" <κ@mail2m37.info>` | `Sephora <κ@mail2m39.info>` |
| `Extra <κ@mail2m18.info>` | `"InfoJobs" <κ@mail2m37.info>` |
| `"InfoJobs" <κ@mail2m32.info>` | `Ponto Frio <κ@mail2m10.info>` |
| `Extra <κ@mail2m18.info>` | `InfoJobs <κ@mail2m30.info>` |
| `Sephora <κ@mail2m35.info>` | `Sephora <κ@mail2m1.info>` |

*κ stands for a constant name.*

an expression produced. (And some extra logic/arithmetic in order to have an easy-to-use parameter control may be desirable in implementations).

Due to a quadratic complexity of the proposed algorithm, a preliminary alignment step should be done first (using less consuming algorithms, for example, suffix tree based ones) in case of long samples. This fact limits the maximum size of a branched RE region rather than the overall RE size and in such a way it doesn't principally prevent the usage of the proposed technique with text samples of arbitrary length.

Indeed, the proposed algorithm is expected to fill the gap between concrete substring alignment and the regularity (repetition) extraction. We mean a cascaded pipeline as follows.

1. A plain string based alignment of input samples splits a RE extraction task into a set of tasks with shorter text samples (complexity $= O(K \times N \times \log N)$ or lower, where $K$ is the sample count, $N$ is a representative sample length).
2. The proposed extraction algorithm aligns characters in a fuzzy (class based) way and creates branches (complexity $= O((N/N_1) \times K \times N_1^2) = O(K \times N \times N_1))$, where $N_1$ is a representative length of sample substring produced by the previous phase.
3. A repetition extraction step over an aligned RE (complexity $= O(N_2^2)$ or lower, where $N_2$ is a representative length of hammock chain in a graph resulting from phase 2, $N_2 <= N_1$).

Phase 2 complexity is critical in such a schema. The overall complexity depends on $N_1$ behavior w.r.t. $N$. Fortunately, in real documents $N_1$ usually seems to be bounded as thy contain some non-volatile structure that may be captured at phase 1. At such an assumption, overall complexity of the pipeline may be kept 'almost' equal to the complexity of phase 1.

An internet service demo based on the proposed algorithm is available at http://regexus.com. All interested people are welcome to test and use it.

## 8 Next steps

We expect the following major directions of further related research.

- to make parameters adjustable with Machine Learning techniques;
- to explore an automated ordering of sample string as well as non-sequential merging methods;
- to apply a preliminary alignment and partitioning of samples in order to reduce the overall complexity;
- to explore an option to extract more 'popular' constructs of RE (say, references to previously matched substrings).

## Appendix

**Algorithm 1** Merging a string into a RE

```
$r \leftarrow$ empty regex
for s $\in$ samples do
    N $\leftarrow$ max position number(r)
    $J_L(*,*) \leftarrow +\infty$
    $J_S(*,*) \leftarrow +\infty$
    $J_L(0,0) \leftarrow 0$
    for j = 0 $\rightarrow$ length(s) do
        for i = 0 $\rightarrow$ N do
            $J_L(i,j) \leftarrow$ use Equation (4)
            $J_S(i,j) \leftarrow$ use Equation (5)
    $current$ $\leftarrow$pointer to$J_L(N, length(s))$
    $newBranch \leftarrow$ empty list
    while $current \neq 0$ or $j \neq 0$ do
        $bestChoice \leftarrow findBestChoice(current)$ ▷ finds a case in Equation(4) or (5)
yielded the minimum value to the matrix element pointed by $current$
        if not empty($newBranch$) and $current \in J_L(*,*)$ then
            addBranch($r, current, branchEnd$)
            $newBranch \leftarrow$ empty branch
        if $isBypass(bestEvent)$ or $E \in L_S(*,*)$ then          ▷ dealing with a new RE
branch
            if $empty(newBranch)$ then then
                $newBranchStart \leftarrow currentElement$
            if $bestChoice.ancestor.j < current.j$ then
                $newBranch$ $\leftarrow$concat(single char selection($s[j]$), $newBranch$
        else                                            ▷ dealing with an existing RE branch
            if $bestChoice.ancestor.i < current.i$ then
                if $bestChoice.ancestor.j < current.j$ then
                    character merge ($s[current.j], charSelection(bestChoice.transition)$))
                else
                    $newSingle \leftarrow$ single char selection ($s[current.j]$)
                    markAsOptional($newSingle$)
                    insert($r, current.i, newSingle$)
            else
                if $bestChoice.ancestor.j < current.j$ then
                    markAsOptional($charSelection(bestChoice.transition)$))
        $current \leftarrow bestChoice.ancestor$
    if not empty($newBranch$) then
        $addBranch(r, current, newBranch)$
```

## References

1. Hussein Almuallim and Thomas G Dietterich. Learning boolean concepts in the presence of many irrelevant features. *Artificial Intelligence*, 69(1-2):279–305, 1994.
2. Alberto Bartoli, Giorgio Davanzo, Andrea De Lorenzo, Marco Mauri, Eric Medvet, and Enrico Sorio. Automatic generation of regular expressions from examples with genetic programming. In *Proceedings of the 14th annual conference companion on Genetic and evolutionary computation*, pages 1477–1478. ACM, 2012.
3. Alberto Bartoli, Giorgio Davanzo, Andrea De Lorenzo, Eric Medvet, and Enrico Sorio. Automatic synthesis of regular expressions from examples. 2013.
4. Alberto Bartoli, Andrea De Lorenzo, Eric Medvet, and Fabiano Tarlao. Playing regex golf with genetic programming. In *Proceedings of the 2014 conference on Genetic and evolutionary computation*, pages 1063–1070. ACM, 2014.
5. Geert Jan Bex, Wouter Gelade, Frank Neven, and Stijn Vansummeren. Learning deterministic regular expressions for the inference of schemas from xml data. *ACM Transactions on the Web (TWEB)*, 4(4):14, 2010.
6. Falk Brauer, Robert Rieger, Adrian Mocan, and Wojciech M Barczynski. Enabling information extraction by inference of regular expressions from sample entities. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, pages 1285–1294. ACM, 2011.
7. Jeffrey Friedl. *Mastering regular expressions.* " O'Reilly Media, Inc.", 2006.
8. Xinbo Gao, Bing Xiao, Dacheng Tao, and Xuelong Li. A survey of graph edit distance. *Pattern Analysis and applications*, 13(1):113–129, 2010.
9. Osamu Gotoh. An improved algorithm for matching biological sequences. *Journal of molecular biology*, 162(3):705–708, 1982.
10. Jean-Michel Jolion. Some experiments on clustering a set of strings. In *Graph Based Representations in Pattern Recognition*, pages 214–224. Springer, 2003.
11. Efim Kinber. Learning regular expressions from representative examples and membership queries. In *Grammatical Inference: Theoretical Results and Applications*, pages 94–108. Springer, 2010.
12. Esko Ukkonen. Finding approximate patterns in strings. *Journal of algorithms*, 6(1):132–137, 1985.
13. Robert A Wagner and Michael J Fischer. The string-to-string correction problem. *Journal of the ACM (JACM)*, 21(1):168–173, 1974.
14. Fubo Zhang and Erik H. DHollander. Using hammock graphs to structure programs. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 30(4):231–245, 2004.