Automatic Synthesis of Regular Expressions from Examples

Alberto Bartoli, Giorgio Davanzo, Andrea De Lorenzo,
Eric Medvet, and Enrico Sorio *

July 5, 2013

Abstract

We propose a system for the automatic generation of regular expressions for text-extraction tasks. The user describes the desired task only by means of a set of labeled examples. The generated regexes may be used with common engines such as those that are part of Java, PHP, Perl and so on. Usage of the system does not require any familiarity with regular expressions syntax. We performed an extensive experimental evaluation on 12 different extraction tasks applied to real-world datasets. We obtained very good results in terms of precision and recall, even in comparison to earlier state-of-the-art proposals. Our results are highly promising toward the achievement of a practical surrogate for the specific skills required for generating regular expressions, and significant as a demonstration of what can be achieved with GP-based approaches on modern IT technology.

1 Introduction

A regular expression is a means for specifying string patterns concisely. Such a specification may be used by a specialized engine for extracting the strings matching the specification from a data stream. Regular expressions are a long-established technique for a large variety of text processing applications and continue to be a routinely used tool due to their expressiveness and flexibility. Indeed, regular expressions have become an essential device in broadly different application domains, including construction of XML

*A. Bartoli, G. Davanzo, A. De Lorenzo, E. Medvet and E. Sorio are with the Department of Engineering and Architecture (DIA), University of Trieste, Via Valerio 10, 34127 Trieste, Italy

schemas, extraction of bibliographic citations, network packets rewriting, network traffic classification, signal processing hardware design, malware and phishing detection and so on.

Constructing a regular expression suitable for a specific task is a tedious and error-prone process, which requires specialized skills including familiarity with the formalism used by practical engines. For this reason, several approaches for generating regular expressions automatically have been proposed in the literature, with varying degrees of practical applicability (see Section 2 for a detailed discussion). In this work we focus on text extraction tasks and describe the design, implementation and experimental evaluation of a system for the *automatic* generation of regular expressions from *examples*. The user is required to describe the desired task by providing a set of examples, in the form of strings in which each string is accompanied by the (possibly empty) substring to be extracted. Based on these examples, the system generates a regular expression suitable for use with widespread and popular engines such as libraries of Java, PHP, Perl and so on. The system is internally based on multi-objective *Genetic Programming (GP)*: GP is a computational paradigm inspired by biological evolution [1]. We remark that all the user has to provide is a set of examples. In particular, the user need not provide any initial regular expression or hints about structure or symbols of the target expression. Usage of the system, thus, requires neither familiarity with GP nor with regular expressions syntax.

We performed an extensive experimental evaluation of our proposal on 12 different extraction tasks: email addresses, IP addresses, MAC (Ethernet card-level) addresses, web URLs, HTML headings, Italian Social Security Numbers, phone numbers, HREF attributes, Twitter hashtags and citations. All these datasets were not generated synthetically, except for one: the Italian Social Security Numbers dataset. We obtained very good results for precision and recall in all the experiments. Some of these datasets were used by earlier state-of-the-art proposals and our results compare very favorably even to all these baseline results.

We believe these results may be practically relevant also because we obtained very good figures for precision and recall even with just a few tens of examples and the time required for generating a regular expression is in the order of minutes.

It seems reasonable to claim, thus, that the system may be a practical surrogate for the specific skills required for generating regular expressions, at least in extraction problems similar to those analyzed in our evaluation.

A prototype of our system is publicly available at `http://regex.inginf.units.it`.

2 Related work

The problem of synthesizing regular expressions [2] or deterministic finite automata (DFAs) [3] from examples is long-established. DFAs and regular expressions may solve similar problems but differ in expressiveness and compactness: a DFA may be exponentially larger than the corresponding regular expression. When using an evolutionary approach to learn a DFA from examples, this difference may imply a very large search space—unpractical on non trivial alphabets. Indeed, most of the research about DFA learning considers strings constructed from an alphabet including very few symbols (usually 2, as in [3]), which clearly does not match text extraction problems.

An evolutionary approach is proposed in [4] and assessed on the extraction of hyperlinks from HTML files. This work considers a *flagging* problem: an example is handled correctly when the generated regex does extract some text from the example, irrespective of the string that is actually extracted. We consider instead an *extraction* problem: an example consists of a string paired with a substring in that line; the example is handled correctly only when exactly that (possibly empty) substring is extracted. We included in our experimental evaluation the dataset of [4]. Interestingly, our results improve those of the cited work even in terms of flagging precision and recall.

Problem and fitness definition in [5] are more similar to ours. The authors applies a Genetic Algorithm for evolving regular expressions in several populations, followed by a composition module that composes two given regular expressions in several predefined ways and selects the composition which scores better on a validation set. The criteria for choosing from the final populations the two specific expressions to be input to the composition module are not given. The proposal is assessed in the context of web data extraction, in particular URLs and phone numbers. According to the authors, when applied to real web documents, the generated expressions are often not able to extract essential URLs components. A later work by the same authors proposes a Genetic Algorithm that restricts the search space by using simple regular expressions constructed with a dedicated algorithm as building blocks for candidate solutions [6]. Performance in URL extraction from real web documents is still quite low, the reported value for F-measure being 0.27 (on datasets that are not public).

An active learning approach for text extraction is explored in [7]. The application domain is criminal justice information systems and the main focus is minimizing the manual effort required by operators. Starting from a single positive example, human operators are introduced in the active learning loop in order to manually prune irrelevant candidate examples generated

by the learning procedure. The approach is assessed on datasets with training set larger than the corresponding testing set—in our experiments the training set is a small fraction of the testing set. The cited work proposes an algorithm that may generate only *reduced* regular expressions, i.e., a restricted form of regular expressions not including, for example, the Kleen operator used for specifying zero or more occurrences of the previous string (e.g., "`a*`" means zero or more occurrences of the "`a`" character). This limitation is not present in the active learning algorithm proposed in [8], which requires a single positive example and an external oracle able to respond to membership queries about candidate expressions—the role played by human operators in the previous works. This algorithm is provably able to generate arbitrarily complex regular expressions—not including the union operator "`|`"—in polynomial time, but no experimental evaluation is provided.

An approach that may be applied to a wide range of practical text extraction cases is proposed in [9]. This proposal requires a labeled set of examples and an initial regular expression that has to be prepared with some domain knowledge—which of course implies the presence of a skilled user. The algorithm applies successive transformations to the starting expression, for example by adding terms that should not be matched, until reaching a local optimum in terms of precision and recall. The proposal is assessed on regular expressions for extracting phone numbers, university course names, software names, URLs. These datasets were publicly available and we included some of them in our experimental evaluation. The requirement of an initial regular expression is not present in [10], which is based on the identification in the training corpus of relevant patterns at different granularity, i.e., either tokens or characters. The most suitable of these patterns are then selected and combined into a single regular expression. This proposal is assessed on several business-related text extraction tasks, i.e., phone numbers, invoice numbers, SWIFT codes and some of the datasets in [9] (we included these datasets in our evaluation).

Automatic generation of regular expressions from examples is an active research area also in application domains very different from text extraction, in particular, gene classification in biological research [11]. This algorithm extracts patterns (mRNA sequences) that have biological significance but cannot be annotated in advance. Our approach focuses on a radically different scenario, because we require that each positive example is annotated with the exact substring which is to be identified.

3 Our approach

3.1 User experience

The user provides a set of examples. Each example is composed by a string t and the substring s of t which has be extracted by the desired regular expression; without loss of generality, we assume that each example contains at most one substring to be extracted. We call negative example an example where s is empty.

The system generates a regular expression fully compatible with all major regular expression engines, including those of Java, Perl and PHP.

3.2 Implementation

Our system is internally based on GP. We represent each candidate solution—an *individual*, in GP parlance—as an abstract syntax tree. A leaf node is an element from a predefined *terminal set* (strings between parentheses indicate the *label* of the corresponding node):

- a large alphabet of *constants* including common characters and punctuation symbols (“`a`”,...,“`z`”,“`A`”,...,“`Z`”,“`0`”,...,“`9`”,“`@`”,“`#`”,...),
- the numerical and alphabetical *ranges* (“`a-z`”,“`A-Z`”,“`0-9`”),
- two common *predefined character classes* (“`\w`” and “`\d`”),
- the *wildcard character* (“`.`”).

A branch node of the tree is instead an element from a predefined *functions set* consisting of the following regular expressions operators:

- the *possessive quantifiers* (“c_1`*+`”, “c_1`++`” and “c_1`?+`”),
- the *non-capturing group* (“`(`c_1`)`”),
- the *character class* and *negated character class* (“`[`c_1`]`” and “`[^`c_1`]`”),
- the *concatenator* (“c_1c_2”),
- and the ternary *possessive quantifiers* (“c_1`{`c_2`,`c_3`}+`”).

Labels c_i of function set elements are *templates* used for transforming the corresponding node and its children into (part of) a regular expression. For example, a node of type “possessive question mark” will be transformed into a string composed of the string associated with the child node followed by

the characters "`?+`". The string associated with each child node will be constructed in the same way, leaf nodes being associated with their respective labels. We transform a tree τ into a string R_τ which represents a regular expression by means of a depth-first post order visit (see [12] for an example and further details).

The ability of an individual to solve the problem of interest is quantified by its *fitness*. The fitness is usually defined in terms of some predefined performance indexes of the solution represented by the individual and computed on a set of solved instances of the problem (the *learning corpus*). We will describe our fitness definition later in this section.

A GP execution starts from an initial population of individuals generated at random and consists in an evolutionary search as follows. 1. Generate an intermediate population with this composition: 10% of the individuals are generated at random; 10% of the individuals are generated by applying the genetic operator "mutation" to an individual selected from the current population; selection is performed with a *tournament* of size 7, i.e., 7 individuals are selected at random and then the individual with highest fitness in this set is selected; finally, 80% of the individuals are generated by applying the genetic operator "crossover" to a pair of individuals selected from the current population, each with a tournament as above. 2. Construct a new population composed of individuals with highest fitness among those in the current population (a strategy called *elitism*) and those in the intermediate population. These steps constitute a *generation*. We iterate this process until either a solution with perfect fitness is found or a predefined maximum number of generations have evolved. We keep the population size constant across all generations. Upon generation of a new individual, we check the syntactic correctness of the corresponding expression. If the check fails, the individual is discarded and a new one is generated.

We used *two* fitness functions and implemented ranking between individuals by means of a standard *multi-objective optimization* algorithm—Non-Dominated Sorting Genetic Algorithm II (NSGA-II). The fitness functions to be minimized are: (i) the sum of the Levenshtein distances (also called *edit distances*) between each detected string and the corresponding desired string, and (ii) the length of the regular expression. In detail, we defined the fitnesses $f_d(R)$ and $f_l(R)$ of an individual R as follows:

$$f_d(R) = \sum_{i=1}^{n} d(s_i, R(t_i)) \tag{1}$$

$$f_l(R) = l(R) \tag{2}$$

where: t_i is the i-th example string in a set of n given examples, s_i is the substring to be found in t_i, $R(t_i)$ is the first string extracted by the individual R for the example t_i, $d(t', t'')$ is the Levenshtein distance between strings t' and t'', $l(R)$ is the number of characters in the individual R—i.e., the length of the regular expression represented by that individual.

The GP search is implemented by a software developed in our lab. The software is written in Java and can run different GP searches in parallel on different machines.

3.3 Observations

We remark that the fitness is *not* defined in terms of precision and recall, which are the performance metrics that really matter in the final result. Other prior works attempt to minimize the number of unmatched strings in the training corpus, thereby focusing more directly on precision and recall [4]. Our early experiments along this line did not lead to satisfactory results. Looking at the generated individuals, we found that this approach tends to be excessively selective, in the sense that individuals failing to match just a few characters are as important in the next evolutionary step as those that are totally wrong. We thus decided to use the Levenshtein distance (along the lines of [5]) and obtained very good results. A more systematic comparison between different fitness definitions is given in the experimental evaluation.

The choice of function set and terminal set has been influenced by the results of our early experiments, as follows. Regular expressions may include *quantifiers*, i.e., metacharacters that describe how many times a given group of characters shall repeat to be considered a match. Quantifiers can be grouped by their behavior in three macro groups: *greedy*, when they return the largest matching string, *lazy*, when they return the minimal match, and *possessive*, that are very similar to greedy quantifiers except that a possessive quantifier does not attempt to backtrack when a match fails. In other words, once the engine reaches the end of a candidate string without finding a match, a greedy quantifier would backtrack and analyze the string again, whereas a possessive quantifier will continue the analysis from the end of the candidate string just analyzed. Since greedy and lazy quantifiers have worst case exponential complexity, we decided to generate individuals that include only possessive quantifiers.

This design choice has been corroborated by the results of early experiments in which we allowed individuals to include either greedy or lazy quantifiers. The execution time of these experiments was way too long to be

practical—in the order of several tens of hours for generating a regular expression, as opposed to the minutes or few tens of minutes typically required when only possessive quantifiers are allowed (Section 4). Allowing regular expressions to contain only possessive quantifiers lead to results that cannot be handled directly by JavaScript engines included in major browsers. However, a simple mechanical transformation—which consists in replacing each possessive quantifier with an equivalent expression composed of group operators and a greedy quantifier—makes the resulting expression compatible with JavaScript. Our prototype implements this mechanical transformation and allows the user to opt for a regular expression which is compatible with JavaScript.

4 Experiments

4.1 Methodology

We considered 12 different extraction tasks on datasets in which we manually labelled all data (Table 1). We made our best to include in the evaluation all earlier proposals that address our problem.

We executed each experiment as follows:

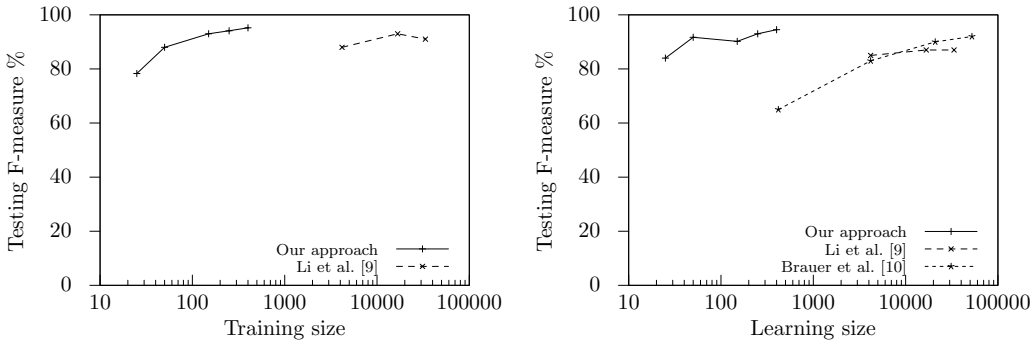
1. We split the dataset in three subsets selected randomly: a *training* set, a *validation* set and a *testing* set. The training set and the validation set are balanced, i.e., the number of positive examples is always the same as the number of negative examples. Those sets are used as *learning corpus*, as described below.

2. We executed a GP search as follows: (i) we ran J different and independent GP evolutions (*jobs*), each on the training set (without the examples in the validation set) and with the same values for GP-related parameters (in particular, a population size of 500 and a number of generations of 1000); (ii) we selected the individual with the best fitness on the training set for each job; (iii) among the resulting set of J individuals, we selected the one with the best F-measure on the validation set and used this individual as the final regular expression R of the GP search.

3. We evaluated precision, recall and F-measure of R on the testing set. In detail, we count an *extraction* when some (non empty) string has been extracted from an example and a *correct extraction* when exactly the (non empty) string associated with a positive example has been

Table 1: Extraction Tasks and Datasets

Task	Extracted item	Corpus description	#Examples	
			Pos.	Neg.
ReLIE URL	URL	Collection of web-pages obtained from the publicly available University of Michigan Web page collection (used by [9, 10])	2820	1057
Cetinkaya HREF	HREF attribute	HTML source of a set of 3 web pages (used by [4])	211	3205
Cetinkaya URL	URL	The same as Cetinkaya HREF	466	767
Hashtag/Cite	Hashtags and citations	Twitter messages collected using the Twitter Streaming API	34879	15121
LogIP	IP addresses	Log from our lab gateway server running the vuurmuur firewall software	5000	5000
LogMAC	Ethernet card address	The same as LogIP	5000	5000
Italian SSN	Italian SSNs	Partly composed of synthetically generated examples including some form of noise and partly obtained by OCR processing of low quality printed documents	2783	2724
Email Header IP	IP addresses	Headers of 50 emails from personal mail boxes of our lab staff. This task is more challenging than LogIP because email headers typically contain strings closely resembling to IP addresses, such as serial numbers, unique identification numbers or timestamps	480	1728
Website Email	Email addresses	HTML source of the address book page from the website of a local nonprofit association	1095	24495
Website Heading	HTML headings	HTML source of a set of pages taken from Wikipedia and W3C web sites	566	48947



(a) ReLIE URL task (b) ReLIE Phone Number task

Figure 1: Analysis of tasks ReLIE URL and ReLIE Phone Number. Performance comparison between our approach and earlier state-of-the-art proposals.

extracted. Accordingly, the *precision* of a regular expression is the ratio between number of correct extractions and number of extractions; the *recall* is the ratio between number of correct extractions and number of positive examples; *F-measure* is the harmonic mean of precision and recall.

4. We repeated steps 1–3 five times.

5. We averaged the results for precision, recall and F-measure across the five repetitions.

4.2 Results

We executed a first suite of experiments with a learning corpus size of 100 elements, 50 training examples and 50 validation examples, and $J = 128$ jobs. The learning corpus is always a small portion of the full dataset, around 1-4% except for the Cetinkaya URL task in which it is 8.1%. The results were very good in all tasks as we always obtained values of precision, recall, and F-measure around or higher than 90% (see Table 2). The only exceptions are the precision indexes for Cetinkaya HREF and the ReLIE Phone precision. However, even these results constitute a significant improvement over earlier works as discussed in detail in the following.

Tasks ReLIE URL and ReLIE Phone Number have been used in earlier relevant works [9, 10]. We repeated our experiments with different sizes for the training set (as clarified in more detail below) and plotted the average F-measure of the generated expressions on the testing set against the learning

set size. The results are in Fig. 1(a) for ReLIE URL and in Fig. 1(b) for ReLIE Phone Number. The figures show also curves for the corresponding F-measure values as reported from the cited works. It seems fair to claim an evident superiority of our approach—note the logarithmic scale on the x-axis.

The performance indexes of our approach are obtained, as described in the previous section, as the average performance of the best expressions generated in each of the five repetitions, where the best expression for each repetition is chosen by evaluating $J = 128$ individuals on the validation set. We analyzed *all* the 5×128 individuals that compose the final populations of the five repetitions and verified that the very good performance which we obtain is not the result of a bunch of lucky individuals (the corresponding performance distributions across the individuals can be found in [12]): our approach manage to generate systematically a number of different expressions with high values of precision, recall and F-measure.

The datasets of tasks Cetinkaya HREF and Cetinkaya URL were also used in earlier relevant works [4] in the context of a flagging problem: a positive example is counted as correct when some string is extracted, irrespective of the string. We assessed the performance of our result, where an example is correct when exactly the desired string is extracted, and of the regular expressions described in [4] according to this metric—i.e., we used all these expressions for solving a flagging problem on our testing set. We obtained with our results a flagging accuracy of 100% and 99.64% on the Cetinkaya HREF and Cetinkaya URL tasks, respectively, whereas the results of [4] are 99.97% and 76.07%. Our results thus exhibit better performance, which is interesting because: (i) the regular expressions in [4] were generated with 266 and 232 learning examples for the two tasks, whereas our result used 100 learning examples; (ii) our GP search aimed at optimizing a different (stronger) metric.

Having ascertained the good performance of the previous configuration, we investigated other dimensions of the design space in order to gain insights into the relation between quality of the generated expressions and size of the training set. We executed a large suite of experiments by varying the size of the learning set, as summarized in Table 2. This table reports, for each task, the number of learning examples, the percentage of the learning corpus with respect to the full dataset and the number of training examples. It can be seen that the quality of the generated expression is very good in nearly all cases, even when the learning corpus is very small. Not surprisingly, for some tasks a learning corpus composed of only 25–50 examples turns out to be excessively small—e.g., Cetinkaya HREF. Even in these cases, however,

Table 2: Experiment results with different learning size

Task	Dataset			Results (%)			Time
	Learning	%	Training	Prec.	Recall	F-m.	(min)
ReLIE URL	25	0.7	12	77.3	82.5	78.3	2
	50	1.3	25	79.9	98.1	88.0	4
	100	2.6	50	88.6	98.1	93.0	6
	250	6.4	150	89.7	99.0	94.1	10
	400	10.3	300	92.0	98.6	95.2	23
ReLIE Phone Number	25	0.1	12	80.9	90.9	84.0	2
	50	0.1	25	85.4	99.2	91.7	5
	100	0.2	50	83.2	98.7	90.2	7
	250	0.6	150	87.7	99.1	93.0	11
	400	1.0	300	90.2	99.1	94.5	28
Cetinkaya HREF	25	0.7	12	34.5	94.8	46.9	5
	50	1.5	25	72.2	94.4	81.6	10
	100	2.9	50	81.3	99.9	89.6	17
	250	7.3	150	85.6	99.2	91.8	30
	400	11.7	300	88.1	100.0	93.5	41
Cetinkaya URL	25	2.0	12	79.4	89.6	83.4	3
	50	4.1	25	87.6	98.4	92.7	7
	100	8.1	50	90.6	99.7	94.9	12
	250	22.3	150	95.0	99.8	97.3	22
	400	32.4	300	97.1	99.8	98.5	29
Twitter Hashtag/Cite	25	0.1	12	98.7	91.2	94.8	1
	50	0.1	25	99.1	95.6	97.3	2
	100	0.2	50	100.0	100.0	100.0	3
	250	0.5	150	99.9	100.0	100.0	8
	400	0.8	300	99.8	99.9	99.9	13
Twitter URL	25	0.5	12	95.4	99.5	97.3	1
	50	0.9	25	97.3	99.4	98.3	2
	100	1.9	50	96.6	99.7	98.1	7
	250	4.7	150	96.5	99.6	98.0	12
	400	7.5	300	97.4	99.4	98.4	24
Log IP	25	0.3	12	100.0	100.0	100.0	4
	50	0.5	25	100.0	100.0	100.0	7
	100	1.0	50	100.0	100.0	100.0	9
	250	2.5	150	100.0	100.0	100.0	7
	400	4.0	300	100.0	100.0	100.0	30
Log MAC	25	0.3	12	100.0	100.0	100.0	4
	50	0.5	25	100.0	100.0	100.0	7
	100	1.0	50	100.0	100.0	100.0	10
	250	2.5	150	100.0	100.0	100.0	19
	400	4.0	300	100.0	100.0	100.0	29
Italian SSN	25	0.5	12	95.6	99.6	97.6	1
	50	0.9	25	90.7	99.7	94.9	2
	100	1.8	50	94.7	99.7	97.1	2
	250	4.5	150	98.6	99.7	99.2	3
	400	7.3	300	98.5	99.6	99.1	6
Email Header IP	25	1.1	12	84.2	99.8	91.3	2
	50	2.3	25	86.1	99.4	92.4	4
	100	4.5	50	87.0	99.4	92.8	6
	250	11.3	150	89.5	98.1	93.6	9
	400	18.1	300	89.8	99.9	94.6	20
Website Email	25	0.1	12	75.3	99.2	81.0	2
	50	0.2	25	88.3	99.8	92.3	5
	100	0.4	50	89.0	98.1	91.8	7
	250	1.0	150	99.1	100.0	99.6	10
	400	1.6	300	99.1	100.0	99.6	23
Website Heading	25	0.1	12	79.9	100.0	88.7	6
	50	0.1	25	72.4	91.4	78.7	10
	100	0.2	50	89.8	95.4	92.4	15
	250	0.5	150	90.6	89.9	89.2	28
	400	0.8	300	92.7	100.0	96.2	42

Table 3: Regular expressions obtained with a training set of 50 elements. For each task, we report only the shortest expression among those obtained in the five repetitions.

Task	Regular expression
Twitter Hashtag/Cite	`[@#]\w++`
Twitter URL	`\w++[^\w]*+\w\.\w\w[^#]\w*+`
Log IP	`\d++\.\d++\.\d++\.\d++`
Italian SSN	`([A-Z]{4,8}+(\w\w\w)*+[A-Z])*+`
Email Header IP	`\d*+\.\d*+\.\d*+\.\d*+`
Website Email	`(\-?+\w*+@*+\.*+\w++)*+`
Log MAC	`\w*+:\w*+:\w\w:\w\w:\w\w:\w\w`
Website Heading	`\<h[^X]*+`
ReLIE URL	`((\w++:)?+/*+\w++\.[a-z]\w([^1]\w)?+\w(\.([^1]\w++)++)?+)++`
ReLIE Phone Number	`([^\)]\d)++[^:][^:]\d++[^:]\d\d[^:]\d`
Cetinkaya HREF	`h[r][^\.]*+(([^1][^h][^1]*+\w*+[^1])*+)*+/*+(\.*+\w\w*+/*+[^1])*+\w*+`
Cetinkaya URL	`([/\w:]*+\.([^:][/\w\.]++)*+)`

enlarging the learning corpus does improve performance and 100 examples always suffice to achieve F-measure greater than 90%.

The table also reports the average execution time for each repetition. We executed our experiments on 4 identical machines running in parallel, each powered with a quad-core Intel Xeon X3323 (2.53 GHz) and 2GB of RAM. Execution time is in the order of a few minutes, which seems practical. Indeed, although constructing the learning corpus is not immediate, the size of such a corpus is sufficiently small to be constructed in a matter of minutes as well. Most importantly, though, this job does not require any specific skills to be accomplished.

We also explored the possibility of reducing the number of jobs $J = 128$, in order to save computing resources. We repeated each of the experiments in Table 2 twice, with $J = 64$ and $J = 32$. We found that performance does not degrade significantly even when the number of jobs drops from 128 to 32—which roughly corresponds to dividing the execution time in Table 2 by four. In this perspective, we decided to set $J = 32$ in the prototype of our system available at `http://regex.inginf.units.it`.

We believe that our fitness definition plays a crucial role in determining the very good results. In order to gain further insights into this issue, we executed further experiments with different fitness definitions. First, we defined a linear combination of the objectives in Eqn. (1) and (2):

$$f(R) = \sum_{i=1}^{n} d(s_i, R(t_i)) + \alpha l(R) \tag{3}$$

Next, we focused on the experiment of the Twitter URL task with learning corpus of 400 examples and executed this experiment with the following fitness definitions.

MO [*Edit*, *Length*] the multi-objective fitness function of our approach (Section 3.2).

MO [*Edit*, *Depth*] a multi-objective fitness function in which the length of the regular expression is replaced by the *depth* of the tree representing the individual.

$Edit + \alpha Length$ a linear combination of the objectives, with varying values for the α parameter (Eqn. 3);

$Edit + \alpha Depth$ the same as the previous definition, but using the depth of the tree instead of the length of the expression;

Errors a set of four fitness definitions obtained from the four above by counting the number of missed examples rather than the sum of the edit distances between each detected expression and the corresponding example.

These experiments have three key outcomes (we omit the corresponding detailed results, that can be found in [12], for space reasons). First, fitness definitions aimed at minimizing the number of missed examples do not work. Indeed, this observation is perhaps the reason why the earlier approaches shown in the Fig. 1(a) and Fig. 1(b) need a much larger training set. Second, when minimizing the sum of the edit distances, the various fitness flavors have essentially no effect on precision and recall, but they do have a strong impact on the complexity, and thus on readability, of the generated expression. Third, a multi-objective framework avoids the problem of estimating the linearization coefficients, but a broad range of values for α provide expressions that are shorter and of comparable quality.

Finally, we show a sample of the expressions generated by our system in Table 3. The table has one row for each of the previous experiments with training set of 50 elements. Each row shows the shortest expression generated across the corresponding five repetitions. The expressions have not been manipulated and are exactly as generated by our machinery.

5 Concluding remarks

We have proposed an GP-based approach for the automatic generation of regular expressions for text extraction. The approach requires only a set of

labeled examples for describing the extraction task and it does not require any hint about the regular expression that solves that task. No specific skills about regular expressions are thus required by users.

We assessed our proposal on 12 datasets from different application domains. The results in terms of precision and recall are very good, even if compared to earlier state-of-the-art proposals. The training corpus was small, in a relative sense (compared to the size of the testing set), in an absolute sense and in comparison to earlier proposals. The execution time is sufficiently short to make the approach practical.

Key ingredients of our approach are: (i) a multi-objective fitness function based on the edit distance and the length of the candidate regular expression, (ii) the enforcement of syntactical constraints on all the individuals constructed during the evolution, (iii) the choice of speeding up fitness evaluation by constructing individuals that may include only possessive quantifiers.

As for any automatic generation method based on examples, our approach effectiveness crucially depends on how representative the examples are for the extraction task. This issue might become relevant when the task includes semantic constraints that cannot be described by means of a few examples (e.g., IP addresses).

Although our approach has certainly to be investigated further on other datasets and application domains, we believe that our results are highly promising toward the achievement of a practical surrogate for the specific skills required for generating regular expressions, and significant as a demonstration of what can be achieved with GP-based approaches on modern IT technology.

References

[1] John R Koza. Genetic Programming: On the Programming of Computers by Means of Natural Selection (Complex Adaptive Systems). 1992.

[2] A. Bràzma. Efficient identification of regular expressions from representative examples. In *Conference on Computational learning theory*, volume 1, pages 236–242. ACM, 1993.

[3] Simon M Lucas and T Jeff Reynolds. Learning deterministic finite automata with a smart state labeling evolutionary algorithm.

IEEE Transactions on Pattern Analysis and Machine Intelligence, 27(7):1063–1074, 2005.

[4] Ahmet Cetinkaya. Regular expression generation through grammatical evolution. In *International Conference on Genetic and evolutionary computation*, GECCO, pages 2643–2646, New York, NY, USA, 2007. ACM.

[5] D.F. Barrero, David Camacho, and M.D. R-Moreno. Automatic Web Data Extraction Based on Genetic Algorithms and Regular Expressions. *Data Mining and Multi-agent Integration*, pages 143–154, 2009.

[6] David F. Barrero, María D. R-Moreno, and David Camacho. Adapting searchy to extract data using evolved wrappers. *Expert Systems with Applications*, 39(3):3061–3070, February 2012.

[7] Tianhao Wu and W.M. Pottenger. A semi-supervised active learning algorithm for information extraction from textual data. *Journal of the American Society for Information Science and Technology*, 56(3):258–271, 2005.

[8] Efim Kinber. Learning regular expressions from representative examples and membership queries. *Grammatical Inference: Theoretical Results and Applications*, pages 94–108, 2010.

[9] Yunyao Li, Rajasekar Krishnamurthy, Sriram Raghavan, Shivakumar Vaithyanathan, and Ann Arbor. Regular Expression Learning for Information Extraction. *Computational Linguistics*, (October):21–30, 2008.

[10] Falk Brauer, Robert Rieger, Adrian Mocan, and W.M. Barczynski. Enabling information extraction by inference of regular expressions from sample entities. In *ACM International Conference on Information and knowledge management*, pages 1285–1294. ACM, 2011.

[11] Wiliam B. Langdon, J. Rowsell, and A. P. Harrison. Creating regular expressions as mrna motifs with gp to predict human exon splitting. In *International Conference on Genetic and evolutionary computation*, GECCO, pages 1789–1790, New York, NY, USA, 2009. ACM.

[12] Automatic synthesis of regular expressions from examples (supplemental material), at `http://machinelearning.inginf.units.it/data-and-tools/automatic-synthesis-of-regular-expressions-from-examples`.