Automatic Generation of Regular Expressions from Examples with Genetic Programming

Alberto Bartoli
DI[3] - University of Trieste
Italy
bartoli.alberto@units.it

Giorgio Davanzo
DI[3] - University of Trieste
Italy
giorgio.davanzo@gmail.com

Andrea De Lorenzo
DI[3] - University of Trieste
Italy
andrea.delorenzo@phd.units.it

Marco Mauri
DI[3] - University of Trieste
Italy
marco.mauri@phd.units.it

Eric Medvet
DI[3] - University of Trieste
Italy
emedvet@units.it

Enrico Sorio
DI[3] - University of Trieste
Italy
enrico.sorio@phd.units.it

ABSTRACT

We explore the practical feasibility of a system based on genetic programming (GP) for the automatic generation of regular expressions. The user describes the desired task by providing a set of labeled examples, in the form of text lines. The system uses these examples for driving the evolutionary search towards a regular expression suitable for the specified task. Usage of the system should require neither familiarity with GP nor with regular expressions syntax. In our GP implementation each individual represents a syntactically correct regular expression. We performed an experimental evaluation on two different extraction tasks applied to real-world datasets and obtained promising results in terms of precision and recall, even in comparison to an earlier state-of-the-art proposal.

Categories and Subject Descriptors

H.3.3 [**Information Systems**]: Information Storage and RetrievalInformation Search and Retrieval

Keywords

genetic programming, regular expressions

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, e.g., extraction of bibliographic citations [2] and signal processing hardware design [7].

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. In this work we outline the design, implementation and preliminary experimental evaluation of a system based on genetic programming (GP) for the automatic generation of regular expressions. It is important to point out that all the user has to provide is just 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.

Essential components of our implementation include the following. First, individuals are generated so as to make sure that each individual represents a syntactically correct expression. Second, the fitness consists of a linear combination of two objectives to be minimized: the edit distance between each detected string and the corresponding examples, the size of the individual.

We performed a preliminary experimental evaluation of our proposal on two different real-world extraction tasks: phone numbers and HTML headings. We found that (i) we obtained very promising results for precision and recall even with just a few tens of examples; (ii) precision and recall on the validation set turned out to be good indicators of the performance on the (unknown) testing set.

The problem of synthesizing a regular expression from a set of examples is long-established (e.g., [1]) and has been studied from several points of view. Due to space constraints, we restrict our discussion to genetic programming based approaches and to recent proposals focused on practical application domains of text-extraction.

The automatic induction of deterministic finite automata from examples was proposed in [3], whereas the generation of regular expressions was proposed in [8]. Stochastic regular expressions were considered in [6]. Our approach follows the same lines of these works, in that regular expressions are directly encoded as program trees. On the other hand, the computing power available today enable us to place much stronger emphasis on real-world basic text processing problems, with regular expressions suitable to be input to widespread engines such as Java, PHP and so on.

Regular expressions are used in biological research for gene classification. Automatic generation by GP has been proposed in this context in [4]. The proposed algorithm is able to generate only a certain subset of regular expressions, tailored to the specific application domain—extraction of mRNA sequences—whereas we place instead no constraints

An approach that may be applied to a wide range of prac-

GECCO'12 Companion, July 7–11, 2012, Philadelphia, PA, USA.
ACM 978-1-4503-1178-6/12/07.

tical cases, but is *not* based on an evolutionary approach, is proposed in [5]. A crucial point of this algorithm is that the training phase 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 proposal is assessed on regular expressions for extracting phone numbers, university course names, software names, URLs. These datasets were publicly available and we included the first one (phone numbers) in our experimental evaluation.

2. OUR APPROACH

The user simply provides a set of examples, each composed by a pair of strings $\langle t, s \rangle$ where t is a text line and s is the substring of t that must be detected by the regular expression. Only one string per line may be captured. A pair where s is empty, meaning that no string must be extracted from t, is a negative example.

In our implementation every individual is a tree that represents a valid regular expression. We satisfy this requirement by defining, for each operator of regular expressions, which sub-trees are suitable for that operator.

The *function set* consists of regular expressions operators: (i) the *concatenator*, that is a binary node that concatenates other nodes or leaves, (ii) the *possessive quantifiers* "`*+`", "`++`", "`?+`", and "`{m,n}+`", (iii) the *group operator* "`()`" and (iv) the *character class* "`[]`" and "`[^]`". The *terminal set* consists of: (i) *constants*, i.e., a single character, a number or a string, (ii) *ranges*, i.e., "`a-z`" or "`A-Z`", (iii) *character classes*, i.e., "`\w`" or "`\d`" and (iv) the *wildcard*, i.e., the "`.`" character.

We used as *fitness* function a linear combination of: the sum of the Levenshtein distances (also called *edit distances*) between each detected string and the corresponding desired string, and the length of the regular expression. In detail, we defined the fitness $f(R)$ of an individual R as follows:

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

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 string extracted by the individual R for the example t_i, $d(x, y)$ is the Levenshtein distance between strings x and y, α is a parameter, $L(R)$ is the number of characters in the individual R—i.e., the length of the regular expression represented by that individual. We set α to 0.01 after a few exploratory experiments. However, we found that our system appears to be quite robust with respect to ample variations of this parameter.

3. EXPERIMENTAL EVALUATION

We considered two different datasets: D_h, a set of 49513 lines of HTML sources taken from web pages of Wikipedia and W3C web sites, from which to extract the HTML headings (i.e., `h1`, ..., `h6`); and D_p, a set of 41832 lines of emails from which to extract phone numbers. The latter dataset is used also in [5].

We split the datasets in two subsets selected randomly, one to use as learning set and the other to use as testing set. The learning set is composed of 400 lines and further split in a training set (300 lines) and a validation set (100 lines).

Table 1: Experiment results

Dataset	Training	Validation	Testing	Results (%)	
	Pos./Neg.	Pos./Neg.	Pos./Neg.	Prec.	Recall
D_h	151/149	50/50	505/48608	96.1	100.0
D_p	27/273	10/90	4788/36644	96.2	95.9

Table 2: Found regular expressions.

Dataset	Regular expression
D_h	`<h\d[^Z]++`
D_p	`\(?+\d\d\d[^0]?+\d\d\d[^0]\d\d\d\d`

We executed the GP search for each dataset as follows: (i) we ran 128 different and independent GP evolutions, each on the training set; (ii) we selected the individual with the best fitness on the training set, obtaining a final population of 128 individuals; (iii) among the resulting 128 individuals, we selected the one with the best F-measure on the validation set and used this individual as the final regular expression for the dataset. Finally, we evaluated precision and recall of the regular expression on the testing set.

The results are summarized in Table 1: we obtained promising figures for precision and recall, corresponding to F-measures equals to 98.1% and 96.0%, respectively for D_h and D_p. Concerning D_p, the approach of [5] scores a F-measure equals to 85% using 4183 lines as learning set, whereas we only used 400 lines.

The regular expressions generated for the two datasets as described above and without any manual post-processing are shown in Table 2.

4. REFERENCES

[1] A. Brāzma. Efficient identification of regular expressions from representative examples. In *Proceedings of the sixth annual conference on Computational learning theory*, volume 1, pages 236–242. ACM, 1993.

[2] C.-C. Chen, K.-H. Yang, C.-L. Chen, and J.-M. Ho. BibPro: A Citation Parser Based on Sequence Alignment. *IEEE Transactions on Knowledge and Data Engineering*, 24(2):236–250, Feb. 2012.

[3] B. Dunay, F. Petry, and B. Buckles. Regular language induction with genetic programming. In *Evolutionary Computation, 1994. IEEE World Congress on Computational Intelligence., Proceedings of the First IEEE Conference on*, volume 1, pages 396–400. IEEE, 1994.

[4] W. B. Langdon, J. Rowsell, and A. P. Harrison. Creating regular expressions as mrna motifs with gp to predict human exon splitting. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, GECCO '09, pages 1789–1790, New York, NY, USA, 2009. ACM.

[5] Y. Li, R. Krishnamurthy, S. Raghavan, S. Vaithyanathan, and A. Arbor. Regular Expression Learning for Information Extraction. *Computational Linguistics*, (October):21–30, 2008.

[6] B. Ross. Probabilistic pattern matching and the evolution of stochastic regular expressions. *Applied Intelligence*, pages 285–300, 2000.

[7] I. Sourdis, J. a. Bispo, J. a. M. P. Cardoso, and S. Vassiliadis. Regular Expression Matching in Reconfigurable Hardware. *Journal of Signal Processing Systems*, 51(1):99–121, 2007.

[8] B. Svingen. Learning Regular Languages Using Genetic Programming. In J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, and R. Riolo, editors, *Genetic Programming 1998 Proceedings of the Third Annual Conference*, pages 374–376. Morgan Kaufmann, 1998.