A Case Study on Grammatical-based Representation for Regular Expression Evolution

Antonio González[1], David F. Barrero[2], David Camacho[1], María D. R-Moreno[2]

Abstract Regular expressions, or simply regex, have been widely used as a powerful pattern matching and text extractor tool through decades. Although they provide a powerful and flexible notation to define and retrieve patterns from text, the syntax and the grammatical rules of these regex notations are not easy to use, and even to understand. Any regex can be represented as a Deterministic or Non-Deterministic Finite Automata; so it is possible to design a representation to automatically build a regex, and a optimization algorithm able to find the best regex in terms of complexity. This paper introduces both, a graph-based representation for regex, and a particular heuristic-based evolutionary computing algorithm based on grammatical features from this language in a particular data extraction problem.

Keywords: Regular Expressions, Grammatical-based representation, Evolutionary algorithms.

1 Introduction

Any Regular Expression, or *regex* [Friedl, 2002], can be described as a particular kind of notation for describing patterns of text. When a particular string is in the set described by a regex, it is said that the regex matches the string. The powerful pattern matching facilities provided by regex in different programming languages such as Perl, PHP, JavaScript, PCRE, Python, Ruby, or Java have not been conveniently exploited by the programmers or the computer scientists due the difficulty

[1] Departamento de Informática. Universidad Autónoma de Madrid.
C/Francisco Tomás y Valiente, n 11, 28049 Madrid, Spain.
{antonio.gonzalez,david.camacho}@uam.es

[2] Departamento de Automática. Universidad de Alcalá.
Ctra Madrid-Barcelona, Km. 33,6. 28871 Alcalá de Henares (Madrid), Spain.
{mdolores,david}@aut.uah.es

to write and understand the syntax, as well as the semantic meaning of those regular expressions.

Any regex can be represented as a Deterministic or Non-Deterministic Finite Automata, a complete discussion of this problem can be found at [Thompson, 1968], [Kleene, 1956], [Chang and Paige, 1992]. From an algorithmic and programming perspective the problem about how to represent and look for the optimal regex is an interesting, but usually a hard problem [Gold, 1978]. From the set of optimization and Machine Learning methods that can be used to represent and automatically search for regular expressions, the Evolutionary Computation (EC) [Eiben and Smith, 2008] provides a collection of algorithms inspired in the biological evolution whose characteristics make them promising to address this problem.

2 Graph-based representation for Regex

Different approaches can be defined to represent a regular expression as an evolutionary based individual, from GAs [Barrero et al., 2009] to GP [Dunay et al., 1994]. The final representation of this individual is a critical aspect in any EC algorithm. This work presents an initial approach based on graphs that uses the basic and syntactic considerations of the regex notation.

2.1 A brief introduction to Regular Expressions

Regular expressions [Cox and Russ, 2007] are a tool used for providing a compact representation of string patterns. They have been widely used by programmers and systems administrators since most of common languages such as Perl, Java, C or PHP support regex and many UNIX administration tools such as *grep* or *sed* use them. Regex are commonly used, for instance, to extract strings or validate user inputs. Regular expressions contain some special symbols called wildcards Following the basic wildcards from IEEE POSIX Basic Regular Expressions (BRE) standard, and POSIX Extended Regular Expressions (ERE) notation, 4 wildcards [1] are considered in this work which are *Plus*, *Star*, *Question mark* and *Pipe*. These wildcards are explained bellow and a graph-based representation are provided in Figure 1:

- **Plus +**. Repeats the previous item once or more.
- **Start ***. Repeats the previous item zero or more times.
- **Question mark ?**: Makes the preceding item optional.
- **Pipe** |: Causes the regex engine to match either the part on the left side, or the part on the right side. Can be strung together into a series of options.

---

[1] A complete syntax reference can be found at http://www.regular-expressions.info/reference.html

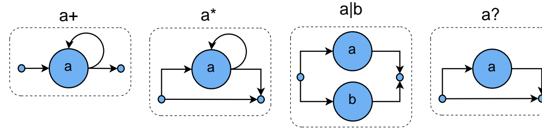


Fig. 1 Graph-based representation for main wildcards.

2.2 A particular example on web data extraction

The extraction of simple web data, as URL addresses, is a very common and useful task achieved by a huge type of programs that crawling and searching the World Wide Web for useful information. Using the representation shown in previous section, any regex could be represented as a Finite Automata. Next regex could be used to retrieve a generic URL link (it only considers most current usual protocols as http, https or ftp) from a web page:

(https|http|ftp)://[a-zA-Z]+(\.[a-zA-Z]+)*\.(com|net|org)

Figure 2 shows in detail the graphical representation of this regex. Note that the aim of the graph is to accept the string that matches the regex, not to accept the string that represents the regex. The graph must accept, for example, ”*http://my.url.com*” but not the regex itself. Also note the initial node is represented as a node with a double border and without any label.

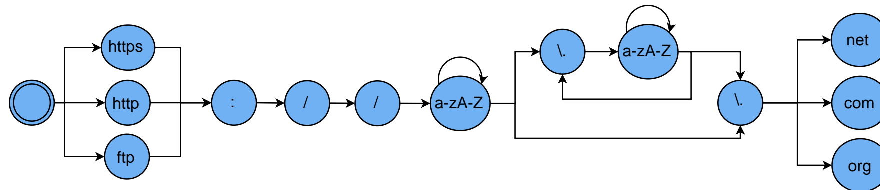


Fig. 2 Graph-based representation for a regex.

2.3 Nodes representation

A major problem in a graph representation is the combinatorial explosion that might happen if each character is represented by a node. In order to avoid it, the representation must reduce the size of the graphs using nodes to represent composed symbols instead of simple ones. In this way the search space is drastically reduced, however a new problem arises: how to automatically construct the composed alphabet.

Our approach uses the Zipf Law [Zipf, 1935] to construct the alphabet. Zipf observed that given a corpus, a small subset of the words concentrates a high frequency of occurrences, while most words appears few times in the corpus. This fact is used

to design a heuristic able to identify important strings, and use it to build the alphabet. For instance, given a set of URL examples, it is possible to identify ”http” and ”ftp” as common strings, and thus use them as a symbol.

The graph that is evolved is composed by three types of nodes. Each node is able to accept a composed symbol such as ”http”. Nodes of type N_1 contain the symbols that generated Zipf valid strings, while nodes of type N_2 are the strings extracted using Zipf Law. Symbols N_3, N_4 and N_5 are the especial characters from regex with an increasing semantics, so the wildcard ”*” represents a cyclic graph (see previous section), whereas symbol ”(” is used to maintain the syntax correctness of the regex. The information given from these kind of nodes will be used by an evolutionary algorithm to guide both the mutation and crossover operations.

Node	Elements
N_1	{:, /, \.}
N_2	{http, https, ftp, com, es, org, net, 8080}
N_3	{\d, \w, a-z, A-Z, |}
N_4	{(,), {, }, [,]}
N_5	{*, + , ?}

Table 1 Categorization of Nodes

2.4 Population representation and initial population generation

Each individual in the population represents a graph whose phenotype is a regex. The graph is represented by a variable length list of interconnected nodes. In order to generate always a syntactically correct regex the set of N_5 nodes has been extended with three more elements, which are +’, *’ and ?’. These nodes do not belong to the vocabulary about regex nor the vocabulary about URL, but they are needed to restrict the effect of the elements +, * and ?, respectively. For example, the regex abc(de)* will be represented as abc*(de)*’, and the program will know that only the string *ed* is affected by the operator *.

Figure 3 shows the genotype of an individual that represents the regex (http|ftp):(\w)*.

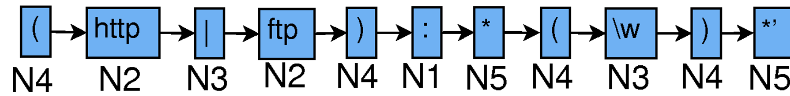


Fig. 3 Regular expression for URL address represented as a list of N_j nodes.

Once an individual can be represented, it has been defined some heuristics to guide the generation of the initial population. These heuristics allow to create well-formed (syntactically and grammatically) individuals. In order to do that, a maximum length of the genotype has been defined. Note, that this length is a constraint for the generation of the initial population, for that reason its possible to obtain, in

any generation, any individual whose genotype exceeds this maximum length. A manually generated regex able to accept all the patterns of this study has 55 nodes. The maximum length of the genotype of the initial population will be 70 nodes. This value allows initial population to exceed the length of the valid regex in, approximately, a 25%.

3 Mutation and crossover operations for regex evolution

From our previous nodes classification, two different types of mutation have been defined:

- **Random**. This is equivalent to the uniform mutation operator found in GA or point mutation in GP. Once a particular node *i* that belongs to class j ($n_{i,j}$) in the graph is selected, it is changed by any randomly token available in our sets of nodes. Therefore, any node in the graph can be substituted by any other.
- ***Guided***. This mutation takes randomly the node $n_{i,j} \in N_j$ from the graph, and it only can be changed by other token (randomly selected) which belongs to the class N_j. This guided mutation tries to use the statistical knowledge acquired from the analysis of natural language, so the token **http** could be mutated into **ftp**, and the grammatical knowledge from the regex notation, so the wildcard ***** could be mutated into +. Note, that in the case of the node belongs to class N_4 and N_5, the same mutation has to be performed twice due to nodes belonging to these classes appear in pairs. That means, if a node * is mutated into +, then the corresponding node ***'**, which represent the end of the operator, must change into **+'**.

The guided mutation allows one to reuse grammatical and syntactical knowledge from the language used to generate regex, it allows to easily generate correct individuals. However, this produces a second important bias in the searching process of individuals.

The crossover operation using a graph and grammatical-based representation (see section 2) needs to consider the inner structure of this graph to select part of the graphs that can be later used in the offspring generation. To compare how the representation works, some crossover operations have been defined:

- **One-point**. This is the equivalent crossover of the traditional one-point crossover found in GA. A point is randomly selected from both parents splitting its genotype in two parts. The new individual generated will inherit two parts, one from each parent.
- **Two-point**. Two points inside the graph (both randomly) are selected, then both sections of the graph are interchanged. It is the equivalent of two-points GA crossover.
- **Grammatical-based selection**. In order to explain this operation, a new concept *typed block* is defined. Let be B_x, where $x \in \{N_4, N_5\}$, a block of variable

length limited by nodes of class x, B_x is named as *block of type x*. For example: *+(http|ftp)+'* is a block limited by nodes belonging to N_5 class, thus is a B_5 block. Grammatical-based selection is heuristically guided and it works as follows:

1. One point in the graph is selected.
2. If the selected node is contained into a B_4 block, then the entire block is selected.
3. If the selected node is not included into a B_4 block, the algorithm will try to determine the B_5 block that contains the selected node. If that block exists, then it is selected.
4. Finally, if the selected node does not belong to a B_5 block nor a B_4 block, then only the selected node will be exchanged.

4 Experimental Results

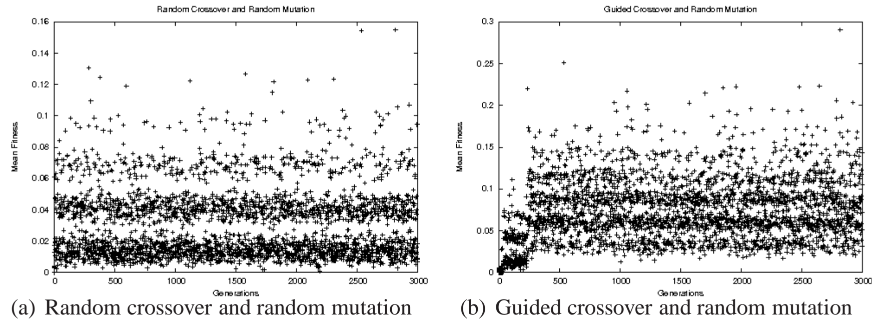
Genetic Algorithms (GA) need an objective function which allows to evaluate the population and to differentiate its individual. This function is called *fitness function*. In order to evaluate each regex generated by the program, a file with different URLs is used. With this two notions and the context of this case study, it is easy to identify that a possible fitness function could be the number of URLs that the generated regex is able to match. Nevertheless, this fitness function is very restrictive and this makes the evolution of the individual difficult.
The fitness function used in this case study is based on the proportion of character that the regex is able to match from the whole test file. Given an individuals i, the fitness of i is defined by equation 1.

$$\mathfrak{F}(i) = \frac{\sum_{j=1}^{N} \frac{m_{ij}}{l_j}}{N} \tag{1}$$

Where N is the number of URLs in the test file, m_{ij} is the number of characters that the individual i matches from the URL number j and l_j is the total length of the URL number j.
There are some parameters which must be specified to execute the GA, these parameters are the number of parents in the population, the number of offspring population and the mutation rate. These values have been set with the following values: the parent population is 30, the offspring population is 45 and the mutation rate is $0,05$. With these values, the program has been executed taking into account all possible combinations of Mutation and Crossover operations, that means trying to launch the program using random crossover and random mutation, guided crossover and random mutation, random crossover and guided mutation and, guided crossover and guided mutation.

Figure 4(a) shows the results of launching the program with random crossover and random mutation. As it can be seen, there is no evolution in the population due to data dispersion. These bad results are expected because with random mutation

and random crossover, there is no guarantee of generating well-formed individual. Therefore, the probability of generating any individual with a good fitness value is very low. Another execution uses guided crossover and random mutation in order to obtain better results than the previous one. The results of this execution are shown in Figure 4(b). As it can be seen, this experiment provides better results than the previous one, but they are not useful since it only recognizes the 30% of the examples. The experiment with random crossover and guided mutation does not work due to guided mutation needs the creation of well-formed individual (syntactically and grammatically), and this restriction is not always true by the random crossover.



(a) Random crossover and random mutation (b) Guided crossover and random mutation

Fig. 4 Mean fitness of graph evolution.

Finally, with guided mutation and guided crossover the best result are obtained. All the generated individuals are well-formed (syntactically and grammatically) due to the heuristic applied in the operators (see Section 3). Figure 5 shows the result of this experiment. The evolution of the population is represented in the increase of the fitness of the population for each generation. The chart shows how the mean fitness increases and it is closer to the optimal value. This value is 1 which represents that it is matched a 100% of the examples in the test file. An example of individual generated by the program is the following: $[\backslash w * |ftp * \backslash d/?/A-Z]$

5 Conclusions

This work shows a representation and evolutionary algorithm case study based on both, graphs and grammatical features from the syntax and grammar standards used to define regex. It have been defined two different kind of specific evolutionary operators, mutation and crossover guided by these grammatical considerations. Although the experimental results provide an initial evaluation of the genetic algorithm, more experiments must be carried out in the future and other parameters, such as precision, need to be considered in order to measure the efficiency of the algorithm.

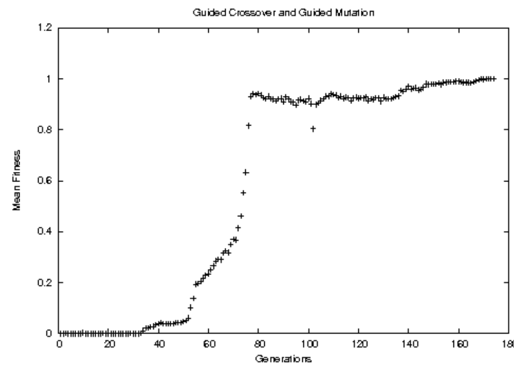


Fig. 5 Results of execution with guided crossover and guided mutation.

Acknowledgements

This work has been partially supported by the Spanish Ministry of Science and Innovation under the projects Castilla-La Mancha project PEII09-0266-6640, COMPUBIODIVE (TIN2007-65989), and by HADA (TIN2007-64718).

References

[Barrero et al., 2009] Barrero, D. F., Camacho, D., and R-Moreno, M. D. (2009). *Data Mining and Multiagent Integration*, chapter Automatic Web Data Extraction Based on Genetic Algorithms and Regular Expressions. Springer.

[Chang and Paige, 1992] Chang, C.-H. and Paige, R. (1992). From regular expressions to dfa's using compressed nfa's. pages 90–110.

[Cox and Russ, 2007] Cox and Russ (2007). Regular expression matching can be simple and fast.

[Dunay et al., 1994] Dunay, B. D., Petry, F., and Buckles, B. P. (1994). Regular language induction with genetic programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, pages 396–400, Orlando, Florida, USA. IEEE Press.

[Eiben and Smith, 2008] Eiben, A. E. and Smith, J. E. (2008). *Introduction to Evolutionary Computing (Natural Computing Series)*. Springer.

[Friedl, 2002] Friedl, J. E. F. (2002). *Mastering Regular Expressions*. O'Reilly & Associates, Inc., Sebastopol, CA, USA.

[Gold, 1978] Gold, E. M. (1978). Complexity of automaton identification from given data. *Inform. Control*, 37:302–320.

[Kleene, 1956] Kleene, S. C. (1956). Representation of events in nerve nets and finite automata. In Shannon, C. E. and McCarthy, J., editors, *Automata studies*, volume 34, pages 3–40.

[Thompson, 1968] Thompson, K. (1968). Regular expression search algorithm. *Comm. Assoc. Comp. Mach.*, 11(6):419–422.

[Zipf, 1935] Zipf, G. (1935). The psycho-biology of language. Houghton Mifflin, Boston, MA.