Regular Expression Generation through Grammatical Evolution

Ahmet Cetinkaya
Department of Computer Engineering
Istanbul Technical University
Maslak, Istanbul, 34469, Turkey
ahmet@itu.edu.tr

ABSTRACT

This study investigates automatic regular expression generation using Grammatical Evolution. The software implementation is based on a subset of POSIX regular expression rules. For fitness calculation, a multiline text file is supplied. Lines which are required to match with generated regular expressions are specified beforehand. Fitness is evaluated according to the successful match results. Using this fitness evaluation strategy, preliminary tests have been performed on different files. Results indicate that the Grammatical Evolution approach to automatic generation of regular expressions is promising.

Categories and Subject Descriptors

I.2.2 [**Artificial Intelligence**]: Automatic Programming—*program synthesis, program verification*

General Terms

Experimentation

Keywords

Regular Expressions, Grammatical Evolution

1. INTRODUCTION

In formal languages theory, regular expressions are notations for describing regular languages. They can describe simple patterns such as different floating point number representations. Regular expressions are strong tools which are used in a wide range of applications such as user input verification, text search and pattern investigation in large amounts of data.

Simple regular expressions are constructed by applying basic recursive rules on the components of a regular language [4]. Whereas users can employ more complex rule sets for generating complicated regular expressions, generating complicated regular expressions is an arduous process of trial and error. To reduce this difficulty, users are encouraged to use regular expression tools (testers, helpers) for assisting in the generation of regular expressions [2].

In this work, automatic generation of regular expressions through Grammatical Evolution is investigated. Tests are performed with the implementation which is based on a subset of POSIX regular expression rules.

The remainder of this paper is organized as follows. Major properties of regular expressions are covered in Section 2. Section 3 describes Grammatical Evolution and its use in regular expression generation. Preliminary test results are presented in Section 4. Section 5 contains conclusions and future work.

2. REGULAR EXPRESSIONS

Formal languages theory defines regular expressions as notations for describing regular languages [4]. POSIX, being a standard for achieving "portability across operating systems", specifies a standard for regular expressions and the tools that use regular expressions. Basic abilities of POSIX regular expressions are character classes, alternation, grouping and repetition.

Character classes are structural forms in which characters are listed [2]. A regular expression with a character class matches with any text containing any of the listed characters. For instance, a regular expression that is required to match with texts containing one of the ten digits can be constructed as "`[0123456789]`". Range expressions can achieve the same effect with a simpler form as in "`[0-9]`". The regular expression "`[a-z0-9]`" can be used for matching alphanumeric characters. Negation is also possible for character classes by using the '^' meta-character as in "`[^a-z0-9]`" which matches non-alphanumeric characters.

Alternation meta-character "|" combines multiple subexpressions into a single expression. For instance, "`this|that`" matches with both 'this' and 'that'. Grouping is mostly used together with alternation for matching one of several subexpressions inserted between '(' and ')' meta-characters.

'+' and '*' are two repetition operators. '+' allows matching one or more of the preceding item and '*' allows matching zero or more. For three numbers '11', '101' and '1001', '`10*1`' matches with '11', '101' '1001', whereas the '`10+1`' regular expression only matches with 101, 1001 and so on.

POSIX regular expressions use the '.' meta-character for matching any character. For example, "`pi.k`" regular expression will match with both of the words "pink" and "pick", or any 4 character word starting with 'pi' and ending with 'k'.

Matched portion of a text is the part with which the regular expression matches. For the sentence "New Pink Floyd album will be released soon.", "`P...`" expression matches with the portion "Pink".

Furthermore, both "`pink|pick`" and "`p[a-z][a-z][a-z]`" can also be used for matching with "pink". In conclusion, users have many different options that can be used for acquiring the same result. Generating a complex regular expression is considered to be confusing and difficult. One possible solution would be using regular expression assistance software such as regular expression testers; another solution is using automatic regular expression generation.

3. GRAMMATICAL EVOLUTION

Grammatical Evolution (GE) is defined as a grammar based genetic algorithm to generate programs in many different languages using the Backus-Naur Form (BNF) definitions [5]. BNF is the set of production rules that are used for expressing grammars of languages. BNF grammars consist of terminal and non-terminal symbols. Terminal symbols can not be expanded into further terminals or non-terminals. BNF grammars may express only a certain subset of a language that meets the needs of a problem.

In GE, a sequence of 8 bit codons represent each individual. Each codon encodes an 8 bit number [6]. These numbers are used in the translation from genotype to phenotype [8]. This mapping process is based on selecting production rules. Starting from the beginning of the genotype, each codon's integer value is used for calculating the current production rule index. For the <meta-character>rule given below, there are two production rules from which to select. If the codon integer value being processed is 231, then modulo operation 231 MOD 2 = 1 would select rule (1) <meta-character>::=+.

```
<meta-character> ::= '*'      (0)
                   | '+'      (1)
```

Different integer values may represent the same production rule due to the use of the modulo operation as in 13 MOD 2 = 1 and 23 MOD 2 = 1. Hence, for the given example, 13, 23 and all other odd numbers would select the rule <meta-character>::=+. This is called genetic code degeneracy [7].

Before the phenotype can be completely constructed, the mapping process may reach the end of the codon sequence. The solution to this problem is "wrapping", which is looping back to the beginning and reusing the same codons [7]. Wrapping represents the "overlapping genes" property of evolution observed for many organisms [6]. The maximum number of wrapping is a parameter of the mapping process. If the mapping for an individual reaches the maximum number without producing a valid phenotype, the individual is classified as invalid. Invalid individuals are given the lowest possible fitness value [7].

3.1 Grammatical Evolution for Regular Expression Generation

Integer representation is used in the implementation [1]. This is where the implementation differs from the standard GE where adjacent 8 bit codons are replaced by their integer values and production rules are selected using these values and the MOD operation [5]. Whereas in this study, for simplifying the implementation, integer values are used for selecting appropriate production rules from BNF grammar and for representing the individuals as integer strings.

Before fitness evaluation, each individual is passed through the process of mapping. The result of this mapping procedure is a regular expression for each individual. Then, fitnesses for each of the regular expressions are calculated. For fitness calculation, users supply a file of n lines where they mark m of them specifically. A regular expression of optimum fitness is required to match all of these m lines and none of the other n-m lines.

The fitness of a regular expression is the sum of A and B where,

A = number of lines that the regular expression correctly matches

B = number of lines that the regular expression correctly not matches

Using this fitness evaluation strategy, the minimum fitness for a valid individual is 0 and the maximum is n. A regular expression that matches all lines of the file is evaluated with the value m. A regular expression that does not match any line has the fitness n-m. Invalid individuals have a fitness of -1.

A suitable Genetic Programming variant can also be employed for generating regular expressions. For instance, regular expressions are used in bioinformatics tools for classifying discovered protein and DNA sequences and Genetic Programming is used for generating a regular expression classifier [3].

4. EXPERIMENT RESULTS

4.1 The BNF Grammar

The BNF grammar given below has been used to represent a subset of POSIX regular expressions.

```
<reg-expression> ::= <term>
                   | <term> '|' <reg-expression>

<term>           ::= <factor>
                   | <factor> <term>

<factor>         ::= <atom>
                   | <atom><meta-character>

<atom>           ::= '.'
                   | '(' <reg-expression> ')'
                   | '[' <character-set> ']'
                   | '[^' <character-set> ']'

<character-set>  ::= <character-item>
                   | <character-item><character-set>

<character-item> ::= 'a' | 'b' | ... | 'z'
                   |  0  |  1  | ... |  9

<meta-character> ::= '*'
                   | '+'
```

4.2 Test Setup

As an example to test the approach, a web page source file of 266 lines was used for fitness evaluation. 83 of these 266 lines included links to other web pages. Each line with

Table 1: Generated Regular Expressions

Regular Expression	Generation Number
[h][r]+[ˆh-m]	240
[h][r]+.[ˆj-tl]+[ˆ1]	316
[e][l]*[f]+[v8-b]+	470
[f]+[7-ay0-0]+	494
[f][5-cxi]	1230

a link to other web pages contains HTML anchor tags. For instance, a line with a link may look like:

```
... <a href="otherpage.html">link to a page</a> ...
```

In this test, a regular expression with optimum fitness is required to match with only the lines with a link to other pages. As a result, the optimum fitness is 266.

An initial population of 100 individuals was constructed. Each individual of this initial population has a random integer string of length 100. The number of fitness evaluations is limited to 250000. The wrapping operator of GE was implemented for a maximum number of 5 rounds.

Tournament selection with a tournament size of 2 and two-point crossover with probability 0.9 is used. For mutation, random-reset and creep mutations were considered. Random-reset for integer representations is defined as, setting a new random integer value for each gene with user-defined probability p [1]. Integer values before and after the mutation may represent same rule due to the use of the modulo operation in Grammatical Evolution mapping process. In order to increase the probability of rule change in a possible mutation, creep mutation was employed. Mutation was implemented using the definition of creep mutation in [1], as adding -1 or +1 to each gene with a probability of 1/N, where N is the size of integer string being used to represent individuals. Since chromosome length for the implementation is 100, creep mutation was used with probability 0.01. Elitism is used to keep the best individual of the previous generation.

4.3 Results

Results for the examples are acquired from the software that has been implemented in this study, using the functional programming language, Haskell. These results are evaluated and reported over 20 runs of the algorithm for the same test file. Each of these runs ended up with an optimum individual. Individuals with optimum fitness value were found in an average 65960 fitness evaluations. Table 1 shows 5 of the evolved regular expressions with optimum fitnesses and the generations at which they were found.

Each regular expression from Table 1 was evaluated again using the same web page source file. The matched portions of the lines are listed in Table 2. The second column of Table 2 shows the target portion of the corresponding regular expression. Table 2 also shows that evolved regular expressions share similar characteristics. Each of them targets similar portions of the lines.

The average of best fitnesses of 20 runs are presented in Figure 1.

The best regular expressions found in 20 runs were also tested with different web page source files. These validation web pages contain 498, 1317 and 1919 lines. Each file has lines containing 43, 82 and 88 links to other web pages, respectively. The best regular expressions of 20 runs are evaluated for their fitness using the validation files. Table 3 shows 5 best regular expressions and the results of their fitness for the corresponding web pages. Three of the regular expressions (marked with *) reached best possible fitness for at least one of the pages.

Table 2: Matched Portions for Regular Expressions

Regular Expression	Matched Portion
[h][r]+[ˆh-m]	"hre"
[h][r]+.[ˆj-tl]+[ˆ1]	"href="..."
[e][l]*[f]+[v8-b]+	"ef="
[f]+[7-ay0-0]+	"f="
[f][5-cxi]	"f="

Table 3: Results for Other Web Pages

Regular Expression	WP 1	WP 2	WP 3
[h][r]+[ˆh-m]	498 *	1316	1917
[h][r]+.[ˆj-tl]+[ˆ1]	498 *	1316	1919 *
[e][l]*[f]+[v8-b]+	498 *	1316	1919 *
[f]+[7-ay0-0]+	492	1299	1911
[f][5-cxi]	492	1283	1891

Further experiments have been conducted for a relatively more complex problem. This time, a file of 273 lines was used where 232 of the lines contained the words 'http' or 'ftp' in them. 5 runs were performed and 4 of them resulted in an optimum solution before the 250000th fitness evaluation. The failing run resulted with a best individual of fitness 269.

Table 4 shows the evolved regular expressions with optimum fitnesses and the generations to which they belong. All regular expressions reflect similar characteristics. The matched portions for all of them is "tp". This indicates that the similar parts between "http" and "ftp" were discovered by the search.

The average of best fitnesses of 5 runs are presented in Figure 2.

5. CONCLUSIONS AND FUTURE WORK

In this study, Grammatical Evolution is applied to the problem of automatic generation of regular expressions. Implemented software was used for preliminary tests on web-page source files. Experiments indicate that the GE approach to automatic generation is promising. However, more work is required to make additional experiments and tune the parameters of evolution for improving the success rate.

The current evolution scheme for automatic regular expression generation is based on a random initial population. As mentioned in [7], the characteristics of initial population is effective on the results. For instance, reducing the number of invalid individuals in the initial population could improve the performance. "Sensible initialization" suggested in [7]

Table 4: Generated Regular Expressions

Regular Expression	Generation Number
[t][p]	882
[v]*[ˆqhuak]*([t])[p]	412
[ct][py]	854
[t][p]+	2073

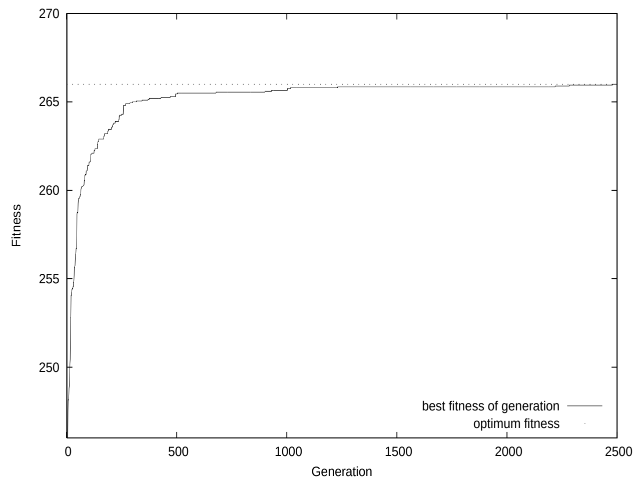

Figure 1: Average of best fitnesses of 20 runs for the first experiment.

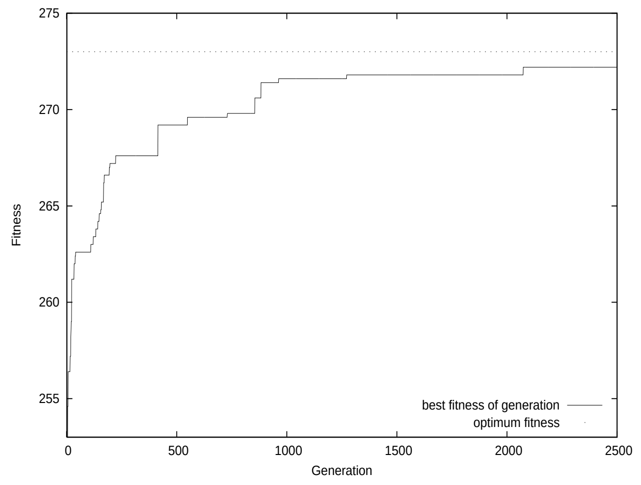

Figure 2: Average of best fitnesses of 5 runs for the second experiment.

can be implemented and used for creating the initial population. Moreover, different crossover and mutation operators can be investigated.

A subset of POSIX regular expressions was used in the tests. After implementing the mapping procedure for the rest of the POSIX rules, new tests can be performed covering a larger portion of the standard.

Furthermore, an algorithm can be implemented and used for correcting invalid regular expressions that would not be accepted by POSIX regular expression engine.

6. ACKNOWLEDGEMENTS

This work is based on the author's senior project under the supervision of Lecturer H. Turgut Uyar from the Computer Engineering Department of Istanbul Technical University, Turkey.

7. REFERENCES

[1] A. E. Eiben and J. E. Smith. *Introduction to Evolutionary Computing.* Springer, 2003.

[2] J. E. Friedlt. *Mastering Regular Expressions.* Addison-Wesley Publishing Company, Sebastopol, CA, 1997.

[3] A. Heddad, M. Brameier, and R. M. MacCallum. Evolving regular expression-based sequence classifiers for protein nuclear localisation. In G. R. Raidl, S. Cagnoni, J. Branke, D. W. Corne, R. Drechsler, Y. Jin, C. R. Johnson, P. Machado, E. Marchiori, F. Rothlauf, G. D. Smith, and G. Squillero, editors, *Applications of Evolutionary Computing, EvoWorkshops 2004*, volume 3005 of *LNCS*, pages 31–40, Coimbra, Portugal, 5-7 Apr. 2004. Springer Verlag.

[4] P. Linz. *An Introduction to Formal Languages and Automata.* John and Barlett Publishers, Sudbury, Massachusetts, 2000.

[5] M. O'Neill and C. Ryan. Under the hood of grammatical evolution. In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, pages 1143–1148, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann.

[6] M. O'Neill and C. Ryan. Grammatical evolution. *IEEE Transactions on Evolutionary Computation*, 5(4):349–358, Aug. 2001.

[7] M. O'Neill and C. Ryan. *Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language.* Kluwer Academic Publishers Group, Norwell, Massachusetts, 2003.

[8] C. Ryan, J. J. Collins, and M. O Neill. Grammatical evolution: Evolving programs for an arbitrary language. In W. Banzhaf, R. Poli, M. Schoenauer, and T. C. Fogarty, editors, *Proceedings of the First European Workshop on Genetic Programming*, volume 1391, pages 83–95, Paris, 14-15 1998. Springer-Verlag.