Regular Expression Generator by using Genetic Algorithm

Ping-Jui Chuang
Department of Computer Science,
National Chiao Tung University
Email:maojui0427@gmail.com

Pin-Chih Chao
Department of Computer Science,
National Chiao Tung University
Email:a5180352@gmail.com

Han-Chang Lu
Department of Computer Science,
National Chiao Tung University
Email:j6e1n1n2y@gmail.com

ABSTRACT

While writing programs, we often encounter some problems while matching the similar strings. If you attempt to write regular expression manually, you will sometimes feel frustrate with that tons of unexpected strings. In this paper, we introduce a regular expression generator to automatically generate regular expressions for given strings by using genetic algorithm. Our system is composed by two fundamental components: a preprocessor and a genetic algorithm loop. We provide a process of our proposed, and believe it will be useful in somewhere.

CCS CONCEPTS

• **Theory of computation** → *Regular languages*;

KEYWORDS

Regular Expression, Genetic Algorithm, Automatic Generator

1 INTRODUCTION

Writing regular expressions (as known as *regex*) to match all strings in a huge set manually is seems to be impossible , and inefficiency even if it is possible. Therefore, generating regular expressions automatically by script or tools is more practical. Before implementing by ourselves, we attempted to find some related tools and papers on the Internet. We found almost nothing related to this articles, but only few commercial tools. There is no free or open-source tools on the Internet, so we try to implement one with genetic algorithm[1] to achieve the goal mentioned above by ourselves.

Our system is composed by two parts: a preprocessor and a genetic algorithm loop. In the first part, as the name implies, preprocessor will do some preprocess on the input strings set. It will first find some meaningful common sub-strings of the input set. Then it will replace all common sub-strings appear in every string by token with some permutations, because some common sub-strings may not appear just once. In this part, we use some well-known string operating algorithm to accelerate the process. After replacing all common parts in the input set, we will then pass the input to the genetic algorithm loop while voiding unneeded calculation cost on the common parts.

In genetic algorithm loop, we define a priority sequence as genotype. Each element in the sequence represent a common usage or a special character in regular expressions. Then we decode the input strings set to a single regular expression as phenotype with the order in sequence and some operations. We will explain the detail in §2. We define a fitness module to evaluate the regular expression generated by the previous step. In this module, users can modify the score in each items to match their preference, such as length of regular expressions, readability, and the penalty of using too much wildcard character ("." in *regex*). And we also use some technique like crossover and mutation to complete our genetic algorithm loop.

2 ARCHITECTURE

As this paper focuses on generating regular expression, the most relevant issue is find the relation between strings. In this section, we will explain the details about how our tool works.

2.1 Preprocessor

For a set of input strings, if there is a component which are totally the same in each string (as known as 'common sub-string'), then we *may* simply preserve and copy it as a part of the regular expression we generate. That is what our preprocessor purposes to.

2.1.1 Find Common Sub-strings. The first step in our preprocessor is to find all common sub-strings of the input strings set. We can not just use *Longest Common Sub-string* algorithm because there may be many common sub-strings. For example, "https://google.com" and "https://facebook.com" have two common sub-strings "https://" and ".com", and we should preserve both of them. So, in our way, we will list all common sub-strings by using dynamic programming, and then we use a filter to get common sub-strings which is not a sub-string of others, e.g. we will get ".com" from set {".", ".c", ".co", ".com"}.

2.1.2 Split Input Strings. While splitting the input strings with the common sub-strings we found, we quickly fall into another problem. There may be two or more common sub-strings, and each of them may appears not just once. So we have to try all permutation to split all input strings and check whether they are split similarly. For "www.jd.bd.jp" and "www.cd.ad.tw", they will be split as {"", 0, "j", 1, "b", 1, "jp"} and {"", 0, "c", 1, "a", 1, "tw"}, where 0 means "www" and 1 means "d.". They are both split in the same order with tokens 0, 1, 1, that means they have the common sub-strings with the similar position, and we can simply preserve them as a part of the regular expression as our purpose.

2.2 Genetic Algorithm Loop

After preprocessing, we will then use genetic algorithm to find the same characteristic of the remain parts of input strings. And we can generate regular expressions according to the same characteristics.

2.2.1 Definition of Genotype. Although we have modified the input strings, their lengths are still totally different. So it is hard for us to encode each strings to genotype directly. Another problem is that the the phenotype we decode to is regular expression, it seems to be strange that we decode from normal strings and then decode to regular expression.

Finally, we came up with an idea. We define 16 types of gene with constant 0x0 to 0xf, each of them represent a common usage

Target	Const	Symbol
numbers	0x0	\d
upper alpha	0x1	[A-Z]
lower alpha	0x2	[a-z]
alpha	0x3	[A-Za-z]
upper hexdigits	0x4	[0-9A-F]
lower hexdigits	0x5	[0-9a-f]
words	0x6	\w
space like	0x7	\s
space only	0x8	[ ]
anything	0x9	.
escape	0xa	[^{}$.|*+?]
symbol	0xb	[SYMBOLS]
range for all	0xc	[??-?]
range for letters	0xd	[??-?]
char or	0xe	[???]
string or	0xf	(??|???|?)

Table 1: Genotype - *Target*: describe the captured content; *Symbol*: what the *Const* will finally transform to.

or a special character in regular expressions. Shown as Table 1. For example, gene 0x03 represent "[A-Za-z]", and gene 0x06 represent "\w", which means "[A-Za-z0-9_]" in regular expressions. Then we define the permutation of these gene as genotype, e.g. "0xe385b1620fda9c47" and "0x6d07cfa3eb128945" is two kinds of genotypes. With this design, we can easily perform crossover and mutation on them because they have the same length as each other.

2.2.2 Decode to Phenotype. Next, how we decode genotype string to phenotype? First, we treat the genotype string as a ordered sequence. For instance, in "0x052e38b16fda9c47", the priority of "0x5" is higher than the priority of "0x8". Then, we will substitute all the characters in each input strings with gene by following the sequence. After substitution, we will transform the strings to value-length representation. Shown as listing 1.

```
Genotype = 0x052e38b16fda9c47
inputs = ['kakbb', 'e3new','aadmm']
# substitution
inputs = ["_____", "_0___", "_____"] # with 0x0
inputs = ["_5_55", "50_5_", "555__"] # with 0x5
inputs = ["25255", "50252", "55522"] # with 0x2
# value-length representation
inputs = [
    [('2',1), ('5',1), ('2',1), ('5',2)],
    [('5',1), ('0',1), ('2',1), ('5',1), ('2',1)],
    [('5',3), ('2',2)],
]
```

Listing 1: Substitute with the order of gene

Second, we will attempt to find the longest common sub-sequence of the inputs string. The purpose of this step is to find more characteristics appear in all input strings so that we can generate regular expressions more precisely. After that, we will randomly pick one possible position of common sub-sequence in each input strings, and replace the remain part with special tokens for the next step. Shown as listing 2.

```
inputs = [
    [('2',1), ('5',1), ('2',1), ('5',2)],
    [('5',1), ('0',1), ('2',1), ('5',1), ('2',1)],
    [('5',3), ('2',2)]
]
# longest common sub-sequence = '52'
# '*' means the position has been chosen
inputs = [
    [( '2',1), ('*5',1), ('*2',1), ('5',2)],
    [('*5',1), ( '0',1), ('*2',1), ('5',1), ('2',1)],
    [('*5',3), ('*2',2)]
]
# replace remain parts with tokens ('?')
inputs = [
    [('?',1), ('5',1), ('?',0), ('2',1), ('?',2)],
    [('?',0), ('5',1), ('?',1), ('2',1), ('?',2)],
    [('?',0), ('5',3), ('?',0), ('2',2), ('?',0)]
]
# now every input is in the same form "?5?2?"
```

Listing 2: Find longest common sub-sequence

Finally, all input strings have become the same value sequence in value-length form, which means we can now generate a single regular expression that can match all of them. Shown as the button of listing 2. Now, we generate the regular expression column by column. For those whose value is one kind of gene, we can decode it to a piece of string according to the table 1. For others whose value is special token, which means they actually have different characteristic in the origin strings, we have a rule to transform them to a regular expression. If there are less than three kinds of strings in their origin input strings, they have a probability with 50% to transform to *string-or*, e.g. "(abc|pqr|xyz)". Otherwise, they will become a wildcard (".."). Then we will add a quantification (e.g. "?", "+", "*", {n}, and {min,max}) if needed. And we will get our final result. Shown as listing 3.

```
inputs = [
    [('?',1), ('5',1), ('?',0), ('2',1), ('?',2)],
    [('?',0), ('5',1), ('?',1), ('2',1), ('?',2)],
    [('?',0), ('5',3), ('?',0), ('2',2), ('?',0)]
]
# for column 1 with value '?' and length 1, 0, 0
# the actual string in the origin is "k", "", ""
# less than 3 kinds, so it may become "(k||)"
# same as "k?". and same step on column 2 ~ 5
regex = "k?" + "[0-9a-f]{1,3}" + "3?" + "[a-z]{1,2}"
     + "(ew|bb)?"
```

Listing 3: Generate regular expression

2.2.3 Genetic Algorithm Parameters. Shown as listing 4.

```
Representation: Permutation of continuous numbers
Initialization: Random
Parent selection: Roulette wheel Selection with
    replacement
Recombination: Partially Mapped Crossover with
    probability 30%
Mutation: Random swap two genes with probability 5%
Replacement: All Parents Are Replaced
Population size: 100
Termination criterion: 20 generations
```

Listing 4: Genetic Algorithm Parameters

2.2.4 Fitness Function. The scoring mechanism we adopt is reward and punishment. The higher score in the final result means the better fitness. We list several factors that may affect the readability of regular expression as evaluation criteria below.

Matching is the most important requirement. If the regular expression can not match all of the input strings, which is an unreasonable result, then it will get the score with minus infinite. Besides, user can also provide a set of strings that they *do not* want to match. If the regular expression match any of them, it will also get the same score above.

Length is the most obvious factor. Longer regular expressions usually get lower readability, so we will minus the score with the string length as penalty. The less amount of *Or* in regular expressions usually means better readability, so we will reward for those using less *Or*. On the contrary, we will also give punishment if there are too many *Or* in them. The reason is that regular expressions will become complex and meaningless if using a lot of *Or*.

Then, let's discuss about the genes. Here we will give different penalties for each gene. For a wider range, such as wildcard ("."), it should be given a significant penalty than hex-digits ("[0-9a-f]"). So deducting all the genotypes that have appeared in the evaluation standard for this part.

Complexity is also a factor to judge regular expression. If it can use fewer components (*columns* mention in §2.2.2) to build our final result, we except that it will be more simple and readable.

The last part of fitness function is that the number wildcard being used. We all know that ".*" and ".+" can be applied to almost all situations. Regular expressions will loss precision if they contain too many ".*" and ".+". So we will give them great punishment and hope to reduce the occurrence in the next generation.

3 EXPERIMENT AND RESULT

Our current architecture can rapidly find out precise regular expressions on the string set with obvious features. After several generations, the average of fitness has grown up. However, the reason for the increment is that most of the individuals with lower fitness has become fewer. On the other hand, the individual with the best fitness appears in the first or second generation and cannot become stronger in the following generations.

In addition, during the testing process, we have found that our system cannot get precise result with the following case.

Input data	Best result
#fff	^#?([0-9a-f]){3}$
#ffffff	^#?([0-9a-f]){6}$
#fff, #ffffff	^#?([0-9a-f]){3,6}$

Table 2: Missing precision - the result will also match strings like "fffff" if the input data are "fff" and "ffffff".

(1) Unable to distinguish data with multiple structures: Cause of the design of our system, we expected the input data has similar format. If the input data consist by two or more structure, our tool will try to find a solution to match all of them. Here's an example for matching hex-digits shown in Table 2.
(2) Unable to get deeper meanings beyond literal character: Because of the way we use is to replace the characters, integrate the common parts, and then output the result. Therefore, the program can not get deeper meanings beyond literal character. For example, the only result we can get from the input strings like IP addresses is "\d{1,3}.\d{1,3}.\d{1,3}.\d{1,3}". Our system will never know that the legal IP address is only consist by four numbers between 0 to 255.

Here a piece of output of our system below:

```
Target :
  https://kakbb.nctu.edu.tw/dcspc/?p=9872
  https://e3new.nctu.edu.tw/dcspc/?id=85596
  https://aadmm.nctu.edu.tw/

1 Generation :
  1763  ^https://(e3)?[a-z]{3,5}\.nctu\.edu\.tw/.*
  1722  ^https://.{5}\.nctu\.edu\.tw/.*
  ...
```

Listing 5: Execute result

4 FUTURE WORK

In future work, we may try to add more gene types and improve fitness calculation methods to generate more precisely. And we can even combine it with machine learning to preprocess and classify the input data. Then we can make the regular expressions more meaningful with different input data type. We believe that these methods can truly increase the quality of the output result.

5 CONCLUSION

In this paper, we produce a system to automatically generate regular expressions for a given string set by using genetic algorithm. First, we find out some meaningful common parts which obviously the same in each strings. Then we use genetic algorithm to help us to cover the rest parts of the strings with appropriate regular expressions.

Although the result may not be precise enough in some situations, it help us to generate regular expressions in an efficient way

while avoiding doing this complicated work manually. And the result can actually match all strings we provided.

Our project is open-source, all codes are available at Github: https://github.com/maojui/Regex-Generator.

REFERENCES

[1] *Wikipedia - Genetic algorithm.* URL: https://en.wikipedia.org/wiki/Genetic_algorithm.