Syntactically comparing regex formulas

By Yacob Ben Youb

500672040

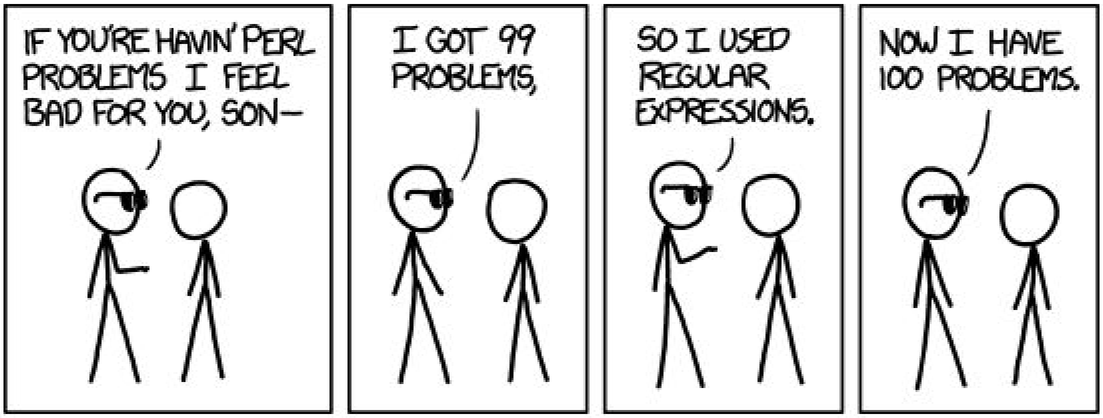

IF YOU'RE HAVIN' PERL PROBLEMS I FEEL BAD FOR YOU, SON—
I GOT 99 PROBLEMS,
SO I USED REGULAR EXPRESSIONS.
NOW I HAVE 100 PROBLEMS.

Table of contents:

Background

This research pertains an assignment to compare two differently written regex formulas which provide the exact same functionality. For this research it had initially been assumed that no external libraries or existing formulas were allowed to be used.

After a lot of background research a meeting was had with mr. Derriks about not being able to find a fitting solution. Together with mr. Derriks desk research was done, however sadly he was not able to find the solution he wanted me to implement and sent me on my way without any guiding knowledge, which means I did not know I was allowed to simply convert the regex to a DFA. It was not until after the project that Mr Derriks told me this, which came as a big surprise because this was far easier than what was created in this project.

Nonetheless, this report contains the knowledge obtained and a beginning structure of a purely syntactic regex comparison algorithm which to my current knowledge does not yet exist, and can provide new insight to the workings of regex syntax.

Research

Without a general guideline of how to solve a problem, even seemingly easy problems can become extremely difficult. Especially when a complex problem needs to be solved, which requires a lot of knowledge in a field to understand. Our problem with the subject at hand is that it required a lot of expertise in regex to find a solution and would solve the problem. As experts in a field might say "5% of the job is hitting the nail. 95% of the job is knowing where to hit.". For this research however we first needed to find out where to hit.

In the beginning of this study most of the time was spent on research of Google articles and some papers about finite state automata. Especially the papers provided a lot of insight into the rules of regex. At the time of conducting the study this information was all on a laptop which has been replaced, so these can sadly not be sourced, however to memory this contained almost every article regarding regex comparison up to the 5th page of Google search in 2015.

No knowledge as to syntactically comparing regex formulas however was available, which means that one needed to be created from scratch.

Rewriting regex

This chapter will discuss the process of discovering an algorithm to compare regex formulas, and what exactly the concept defines.

A simple approach

Because no existing syntactical implementation had been created, it was assumed that the task was to create one by hand. This was mostly done by looking at formulas with similar results which were written differently. To start off, the following formula was used as a base comparison:

```
String regexOne = "xyz(xyz)*";
String regexTwo = "(xyz)*xyz";
```

At first observation we can see that in String regexOne the xyz gets repeated in the second part between the brackets which can be repeated infinitely. This means we can simply switch around matching character before brackets with matching characters inside of brackets. While this is seemingly a good solution, this switch would not work when there wasn't an exact repetition of characters before the brackets within the brackets that followed. An example where the algorithm above wouldn't work is:

```
String regexOne = "xyz(ayz)*";
String regexTwo = "x(yza)yz*";
```

The problem with our simple comparison algorithm is that while the output is similar, not all characters between the brackets are the same as the characters before them which means our simple matching method will now fail. Another solution needed to be found.

A difficult approach

Because regex contains many, many edge cases, it was incredibly difficult to come up with a working solution After more than a hundred hours of desk research, thinking and testing the movement of different characters in regex, many different comparison implementation were tested. Because of the lack of version control usage, most of these non-functional implementations have been deleted. However one implementation a basic application proving the validity of the concept has been created. It's important to take note that because of the sheer amount of special regex characters, a full implementation has not been created. Following the rules provided here, they can however easily be created.

The conceptual solution is based on two important rules which must be applied in chronological order, else they will not work. The rules are as followed:

1: Expand or retract both regex formulas as far as possible.

In order to syntactically compare two differently written regex formulas, they must first be written out as long or as short as possible. In this conceptual code implementation we will expand them. This particularly pertains quantifiers such as the regex "+" operator. Any regex + operator can be converted to a single repetition of the pattern it repeats, followed by that same pattern with a * behind it. For example:

```
String regex = "x(yzx)+yz";
```

Becomes

```
String regex = "xyz(zxy)(xyz)*z";
```

In the conceptual code demonstration of this, only the + quantifier has been converted, however a full code implementation would also need to convert quantifiers such as {n} by repeating the number in front of it with the same amount of repetitions, and other special quantifiers which might apply.

2: Compare and swap

The biggest issue of comparing similar regex formulas came with the introduction of regex groups (characters between brackets) which were not all similar to characters between the brackets. After a lot of failed implementations, an eventual breakthrough was discovered which allowed for the movement of single characters inside of brackets while keeping any regex consistent. Namely:

“When a character before any group is the same as the last character inside of the group, even with a quantifier after the group, the last character before the group can be moved to the first spot inside of the group, and the last character inside of the group can be moved after the group.”

A visual demonstration of this concept is given below:

1: The last letter (or group) before the brackets

```
xyz(ayz)*
```

2: Can be moved after its brackets

```
xy(zay)*z
```

3: This can be repeated

```
x(yza)*yz
```

4: Until a non-match is found.

```
x(yza)*yz
```

By using these rules, **any** regex written in different order can move its internal characters so that it will match a regex formula with the same functionality.

Implementing the algorithm

A code implementation of this concept also needed to be created. In the process of creating this code, the first rule of the algorithm had also been discovered. In total this code has been rewritten from scratch over 40 times. The code has been written in Java. Because of limited coding knowledge as a second year IT student, this cost far more time and is less expandable than a fully object oriented implementation.

First, a List is created which hold each character separately

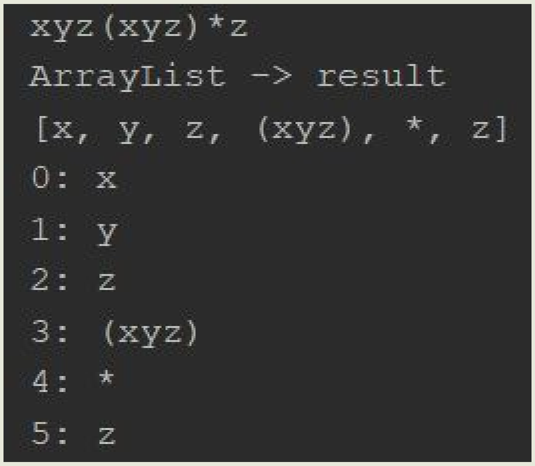
```
List<Character> regex = new ArrayList<Character>();
for (int i = 0; i < input.length(); i++) {
   regex.add(input.charAt(i));
}
```

Next a new list is made, this time of strings. Whenever an opening bracket is found, it looks for a closing bracket to group all the strings inside of. After this it adds each character string to their respective index.

```
List<String> result = new ArrayList<String>();
String temp = "";
for (int i = 0; i < regex.size(); i++) {
   if (regex.get(i).equals('(')) {
       temp += '(';
       while (i + 2 <= regex.size() - 1 && regex.get(i + 1) != ')') {
           i++;
           temp += regex.get(i);}
       temp += ')';
       result.add(temp);
       i += 2;
   }
}
```

Now a String List of grouped characters has been created. An example of the output has been created and it looks as following:

```
System.out.println("ArrayList -> result");
System.out.println(result);
int index = 0;
for (String s : result)
   System.out.println((index++) + ": " + s);
```



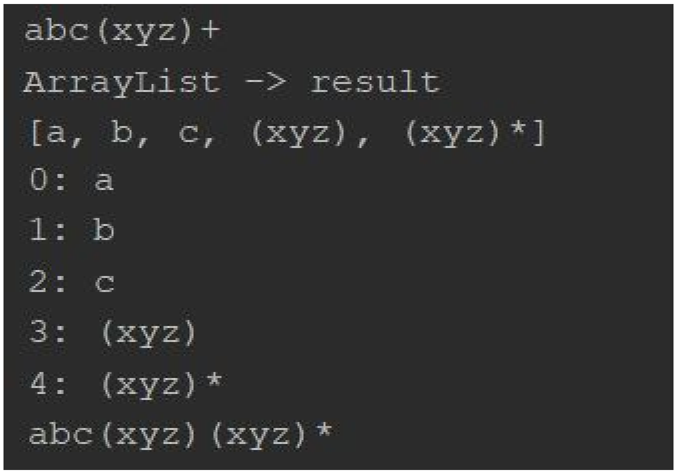
```
xyz(xyz)*z
ArrayList -> result
[x, y, z, (xyz), *, z]
0: x
1: y
2: z
3: (xyz)
4: *
5: z
```

After separating all these indeces, another factor which needs to be accounted for is the + quantifier. Whenver a plus quantifier is detected, it copies the previous index and adds a zero or more quantifier.

```
result.add(regex.get(i).toString());
             if (regex.get(i).equals('+')) {

             if (temp.contains("(")) {
                    result.remove(result.size() - 1);
             } else {
                    result.remove(result.size() - 1);
                    temp = result.get(result.size() - 1);
             }
             temp += '*';
             result.add(temp);
             temp = "";
          }
```

Sample output using the regex abc(xyz)+



```
abc(xyz)+
ArrayList -> result
[a, b, c, (xyz), (xyz)*]
0: a
1: b
2: c
3: (xyz)
4: (xyz)*
abc(xyz)(xyz)*
```

Becomes abc(xyz)(xyz)*

Swapping characters

The last part of the code is written to swap the character before a group, and the last character inside of the group if they are the same. After this the group and character switch index places, so the last character in the group ends up after the group itself.

```
for (int i = 0; i < result.size() - 1; i++) {
           //Get index with (
           if (i > 0 && (result.get(i).charAt(0) == '(')) {
               //last character before the )
//                  if (result.get(i-1).charAt(result.get(i).length()-2) ==  )
               if ((result.get(i).charAt(result.get(i).length() - 2)) ==
(result.get(i - 1).charAt(result.get(i - 1).length() - 1))) {
                   System.out.println((result.get(i).charAt(result.get(i).length()
- 2)));
                   System.out.println((result.get(i - 1).charAt(result.get(i -
1).length() - 1)));
                   //Switch last character before ")" and last character in slot
before, and then switch the entire slots.
                   StringBuilder charSwapper = new StringBuilder();
                   charSwapper.append(result.get(i - 1).charAt(result.get(i -
1).length() - 1));
                   charSwapper.append(result.get(i).substring(1,
result.get(i).length() - 2));
                   charSwapper.insert(0, '(');
                   charSwapper.append(")");
                   result.set(i, String.valueOf(result.get(i -
1).charAt(result.get(i - 1).length() - 1)));
                   result.set(i-1, charSwapper.toString());
                   System.out.println(charSwapper.toString());
                   i=0;
               }
           }
       }
```

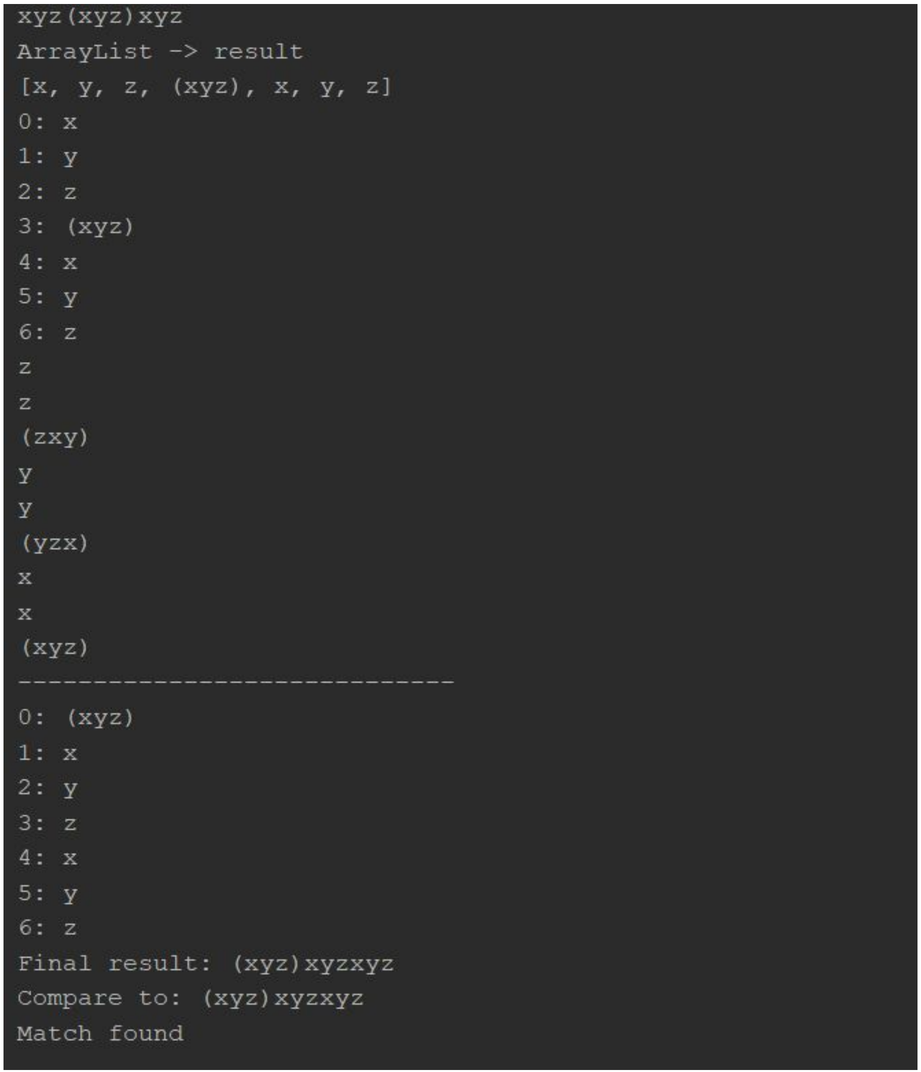
Now that all the indeces are ordered, they can be concatinated in order and then compared to a second regex string. In this example the algorithm has not been used on the second string, but it can be by turning the algorithm into methods and simply passing the regex string as a parameter.

```
StringBuilder finalString = new StringBuilder();
index = 0;
for (String s : result) {
   finalString.append(s);
   System.out.println((index++) + ": " + s);
}

System.out.println("Final result: "+finalString);
System.out.println("Compare to: "+ secondRegex);

if (secondRegex.equals(finalString.toString())){
   System.out.println("Match found");
}
```

The end result of this entire process is a fully shuffled String which can be compared to another string:



```
xyz(xyz)xyz
ArrayList -> result
[x, y, z, (xyz), x, y, z]
0: x
1: y
2: z
3: (xyz)
4: x
5: y
6: z
z
z
(zxy)
y
y
(yzx)
x
x
(xyz)
-----------------------------
0: (xyz)
1: x
2: y
3: z
4: x
5: y
6: z
Final result: (xyz)xyzxyz
Compare to: (xyz)xyzxyz
Match found
```

Areas of improvement

Because of time constraints (and at the time limited coding knowledge) it was not possible to write out special functionality of every character and the current implementation does have quite a few flaws. These are mostly flaws which have sprung from the inability at the time to create a well object oriented class to swap the regex characters, and especially keep certain groups of regex tokens together. When rewriting the algorithm, it would be best suited to create custom comparator functions which can be called in order to swap characters, which would only compare certain characters inside of a group.

Groups:

First of all, all characters need to be grouped better together, and always take a remaining quantifier(*) with it. This could be implemented by using regex itself and implementing a method which searches for alphanumeric letters inside of a group, ignores characters such as '*()' and scans using the index itself. With this it can move the entire index including any related quantifiers or special characters without falsely comparing them to other characters.

It should be kept in mind that characters before a group must fully correlate to the last character, and when the last character in a group is also a group they should only be compared to other groups.

Brackets: [a-z0-9] and or statements

There is a big similarity between brackets and OR-statements, because everything between brackets is simply a long OR statement.
Because brackets offer a choice between characters, every possible character between brackets needs to be concatenated into a group while being separated with '|' signs.

For example: [abc] is the same as (a|b|c)

Caution should be taken with "-" signs, which need to convert into OR signs separating the entire possible range of choices between the two characters

Groups inside of groups:

This is a very uncommon edge case where things get rather tricky. In order to find groups inside of groups a regex algorithm needs to be used which creates an array inside of the existing array recursively for every ‘(‘ up until the last ‘)’ in that index.

It should then remove the outer brackets and do the same for every inner group inside of that in a recursive manner.

This sorting algorithms should happen after the initial List quantifiers have been written out, but before the initial sorting has happened, as it is important to sort out inner groups in indices before comparing them to other indices. When problems arise with the maximum amount of inner groups, a new list should be created using the maximum count of previously found inner groups as dimensions.

Conclusion

While it would take a lot of time to properly code a syntactical regex comparator by writing out every edge case, it is certainly possible using the basic concept covered in this research. First all regex operators which can be turned into another by writing them out, need to fully converted so all their possible syntax is displayed. Then characters and groups need to be compared with one another to see which positions can be switched. In the end this results to a fully written and maximally swapped regex algorithm which any other exactly similar regex algorithm would also be turned in to.

Reflection

The biggest lesson that has been learned from this research is that it is very important to write conceptual code and think of possible edge cases, and also to test conceptual code on multiple scenarios be before starting development. These lesson have remained with me throughout my education and aided a lot in the process of creating coding structures. This reduces development time by weeding out unnecessary code before having even written it.

Another important lessing is making code modular and object oriented. It is far easier to modify existing code functionality without having to completely rewrite it. This lesson had proven especially useful in my thesis, in which I created a machine vision framework in a modular way so that certain poorly performing classifiers could later on easily be replaced.

As for coding knowledge, hindsight is always 20/20. Now that I have far more coding experience I could say that for this project custom comparator functions would have been extremely useful, as well as comparing characters in groups and excluding special characters, however this would (once again) require the entire code to be rewritten from the ground up, as has happened many times before, which is not the point of this research.