Program Synthesis Final Project Report

David Klingler

April 27, 2023

Abstract

This project presents a method of generating regular expressions through the use of examples. The user can specify examples to be matched by the output and negative examples to not be matched in order to guide the search. The approach uses version spaces to compactly represent possible regular expressions. These version spaces are combined in order to discover shared structure between two version spaces. Once no more version spaces can be combined, the possible regular expressions each version space represents is generated and merged with regular expressions from the other version spaces to generate a potential output regular expression.

1 Introduction

The goal of my project is to synthesize regular expressions. From my own experience (and the anecdotal evidence from my friends), I have found that a surprising number of computer science students know about regular expressions and how to use them but lack the expertise in creating them. This is most likely due to the fact that regular expressions do not contain enough complexity to justify their own class and do not fit nicely into the curriculum of any core class. As such, many computer science students have been exposed at some point to regular expressions but have never been formally taught the concepts behind them. This awareness of regular expressions without the ability to create them is the motivation behind this project. There are many Stack Overflow questions about how to create a regular expression. These questions serve as a useful source of benchmarks. This project presents a method to allow its users to quickly and conveniently synthesize regular expressions.

2 Specification

In user of the program specifies the characteristics of the regular expression they desire by providing two kinds of examples. The first kind is called a matching example. these are strings the user wants the synthesized regular expression to match. The second kind is called a negative example. These are strings the user does not want the synthesized regular expression to match.

3 Algorithm

The algorithm for generating regular expressions from examples consists of four main steps: version space generation, version space combination, regular expression combination, and ranking. The first step entails generating a version space for each example. The data structure behind a version space is a graph, which was inspired from the FlashFill paper [1]. Each edge in the graph represents a set of simpler regular expressions that match that portion of the example. The next stage of the algorithm combines version spaces by taking the intersection of the two graphs very similarly to the intersection operation in the FlashFill paper [1]. This process is repeated until the intersection between all version spaces is the empty set. Next, a set of all possible regular expressions for each version space is generated. Then final candidate regular expressions are generated by combining one regular expression from each version space. This combination uses some rules to increase the conciseness of the produced final regular expressions. The last step is to rank the generated regular expressions. This is done effectively by a simple heuristic.

3.1 Version Space Generation

Let m be a matching example with ℓ characters. A version space of m is a directed acyclic graph $v = (n, e)$. Where n is a set of $\ell + 1$ nodes each assigned a number from 0 to n and e is a set of directed edges. Each edge in e must have a destination greater than its source. An edge from n_x to n_y $(x < y)$ stores a set of regular expressions that match the sub-string of m starting at index x and ending at index $y - 1$. For example given the example "abc", a valid edge from n_1 to n_2 would be the set $\{[b], [a - z], [a - zA - Z]\}$.

In order to generate a version space for an example, a set of simple regular expressions are enumerated. Then each of these regular expressions is tested on the example. If it matches any part of the example graph, it is added to the appropriate edge in the graph. How this simple set of regular expressions is generated is discussed in section 3.5.

3.2 Version Space Combination

Two version spaces are combined by taking their intersection. Let the first version space be $v_1 = (n_1, e_1)$ and the second version space be $v_2 = (n_2, e_2)$. The intersection operation defined as: $v_i = (n_i, e_i)$ where $n_i = n_1 \times n_2$ and $v_i = \{\langle(n_{1x}, n_{1y}), (n_{2x}, n_{2y})\rangle \mid (n_{1x}, n_{1y}) \in e_1,$ $(n_{2x}, n_{2y}) \in e_2\}$. The value of $\langle(n_{1x}, n_{1y}), (n_{2x}, n_{2y})\rangle$ is the intersection of the values of (n_{1x}, n_{1y}) and (n_{2x}, n_{2y}).

Any path of edges from $(n_{1,0}, n_{2,0})$ to $(n_{1,|n_1|}, n_{2,|n_2|})$ represents a set of regular expressions that exist in v_1 and v_2. If such as path exists then the intersection of v_1 and v_2 is non-empty. Two versions spaces are combined by taking their intersection and keeping the resulting version space as long as it is non-empty. When the intersection of two graphs is taken, some edges may not lie along any path from the start node to the end node. These edges are pruned by performing a BFS traversal from the start node and a reverse BFS traversal from the end node. Only edges that are visited in both traversals exist along a path from the start to the end. All other edges are discarded.

Version spaces are naively combined until the intersection between all remaining versions sets is the empty set. This process captures the most generic common structure between all initial version spaces. This causes a problem in that this always produces the regular expression .+ (where . is shorthand for a character class that matches everything except the new line). This is where negative examples are used to eliminate certain possibilities.

Just like matching examples, a version space is generated for each negative example. Before matching version spaces are combined, each negative example is subtracted from each matching example. The subtraction operation first performs an intersection between a matching version space (v_1) and a negative version space (v_2). For each edge $e'_i = \langle (n_{1x}, n_{1y}), (n_{2x}, n_{2y}) \rangle$ in the intersection that lies on path from the start node in v_i to the end node in v_i there is a corresponding edge $e'_1 = (n_{1x}, n_{1y})$ in the matching version space v_1. Let $v(e'_i)$ and $v(e'_1)$ be the set of regular expressions of e'_i and the set of regular expressions of e'_1 respectively. If $|v(e'_i)| < |v(e'_1)|$ or $y - x > 1$ for both (n_{1x}, n_{1y}) and (n_{2x}, n_{2y}), then $v(e'_1)$ is set to $v(e'_1) - v(e'_i)$. This removes values from edges in v_1 that could generate a regular expression that matches the negative example.

3.3 Regular Expression Combination

Once no more version spaces can be combined, the set of regular expressions represented by each version space is enumerated. Then a final regular expression is generated by simple taking a regular expression from each version space and then taking their union. This takes the form $r_f = (r_1|r_2|r_3|...)$ where r_1 is a regular expression from v_1, r_2 is a regular expression from v_2 and so on.

However, this method is not very concise. In order to increase the conciseness of the final regular expressions, a few rules are used to combine regular expressions that share a common structure. Below are examples of some of the rules (some of the reverse rules are left out for brevity). In the table below a, b, and c can be any regular expression (not just a character class with a single character).

Regex 1	Regex 2	Combined Regex
ab	a	ab?
ab+	a	ab*
ab	ac	a(b|c)
aba	aca	a(b|c)a
aba	aa	ab?a
ab+a	aa	ab*a

This takes advantage of the shared structure of regular expressions and in practice greatly reduces the size of the final regular expressions.

3.4 Ranking

The final step of the process is to rank all of the final regular expressions. A heuristic rank function $rank(r)$ where r is a regular expression is used to determine which regular expression to output. The regular expression with the lowest rank is chosen as the output. The rank function is simply the sum of the size of each character class used in the regular expression where the size of the character class is the number of characters matched by the character class.

3.5 Enumeration

The enumeration of simple regular expressions used to create the versions spaces in the current implementation is quite simple. It first creates a single character class for each unique character in both the matching and negative examples. It also adds a set of common character classes such as lower case letters, numbers, whitespace, etc to the set of simple regular expressions. The enumeration does effect what regular expressions the algorithm is capable of producing. Users in the specification can optionally provide a list of regular expressions to be used in the creation of the version spaces. A major limitation to the types of regular expressions this algorithm can produce is the fact that if some part of a regular expression r_i is not in the set used to generate version spaces, then the algorithm will not include $(r_i)+$, $(r_i)*$, or $(r_i)?$.

4 Evaluation

All benchmarks were ran on an Intel Core i7-11375H with 32 GB of memory. The benchmarks consist of eleven different regular expressions that were either commonly asked on StackOverflow, from my personal use other classes this semester, or that were found while testing the capabilities of the system.

Benchmark	# Matching Exs	# Negative Exs
Floating point, 2 precision	4	2
Hex Number	4	1
Match Either	2	1
Pascal Float	6	2
Pascal String	3	1
Phone Number	4	4
Fixed Area Code	2	2
Social Security Number	2	1
Word or prefix	2	1
Nested Optionals	6	1
Variable Middle	6	1

The table above shows that the algorithm is capable of generating non-trivial regular expressions with both few matching and negative examples. I used both the pascal float and pascal string regular expressions in my compilers course this semester.

The table below shows the time required to complete each benchmark as well as if the algorithm could still produce the correct regular expression without the negative examples.

Benchmark	Time in msec	Passed w/o Neg Exs
Floating point, 2 precision	40	N
Hex Number	45	N
Match Either	113	N
Pascal Float	47	N
Pascal String	42	N
Phone Number	56	N
Fixed Area Code	123	N
Social Security Number	87	N
Word or prefix	386	N
Nested Optionals	34	N
Variable Middle	207	N

The fact that no benchmark can be passed with no negative examples demonstrates their importance. The version spaces are always combined as long as there exists at least one satisfying regular expression. In practice this means that the algorithm always picks the most general regular expression in order to minimize the number of versions spaces. This means that without any negative examples, the algorithm almost always returns the regular expression .+ (which means one or more of any character so basically match anything). Negative examples are required to limit the generality of the regular expressions in the version spaces.

However, not all negative examples are required for correctness. For instance, the phone number benchmark has four negative examples, of which only two are necessary for correctness. With four negative examples it takes 56 milliseconds to find the correct regular expression. With only two negative examples, it takes eighteen seconds. This is because more negative examples reduces the size of the initial version spaces which greatly reduces the size of the second half of the algorithm. These negative examples do require some knowledge in order to be able to construct them effectively.

Overall, this algorithm is able to effectively synthesize non-trivial, useful regular expressions from a few examples in an efficient manner. Future work could include converting the combination of regular expressions in the second to last step as an algorithm that works on version spaces instead of individual regular expressions. This could greatly reduce the cost of the program. In addition, the current implementation is completely single threaded and there is large amounts of parallelism baked in the algorithm that could be exploited for even faster results.

5 Related Works

This project was heavily inspired by the FlashFill paper by Gulwani [1]. The use of version spaces and the intersection between version spaces in this project is very similar to FlashFill.

These ideas were adapted to regular expressions. Unlike FlashFill which must discover conditionals to differentiate between forms, this project takes advantage of the union operator to combine regular expressions that could not be combined via the intersection operation.

6 References

[1] Gulwani, Sumit. "Automating string processing in spreadsheets using input-output examples." ACM Sigplan Notices 46.1 (2011): 317-330.

7 Appendix A

Below are the benchmarks with their associated regular expressions:

```
Floating point, 2 precision: ([0-9])+(\.[0-9][0-9])?
Hex Number: 0(x|X)([0-9a-fA-F])+
Match Either: (apple|banana) - this benchmark was suprising hard
Pascal Float: ([0-9])+((e[-]?|\.)([0-9])+)?
Pascal String: '(([^']|'')+)'
Phone Number: (\([0-9][0-9][0-9]\)|[0-9][0-9][0-9])-[0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]
Fixed Area Code: 512-[0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]
Social Security Number: [0-9][0-9][0-9]-[0-9][0-9]-[0-9][0-9][0-9][0-9]
Word or its prefix: s(eason)?
Nested optionals: a+(b+c*)?
Variable middle: a+(b+|A+|[0-9]+)a+
```