Patrick, Ethan, Luis
Phase 4 Project Report

Comments

For phase 4 of our project, we took the code from our phase 3 submission and reexamined it with the intention of fixing bugs and making it faster by simplifying code and revamping our algorithms to have lower complexity. After talking with our instructor, we realized that we had forgot to include a bunch of the rules for simplification that are given on page 180 of "Regular-expression derivatives re-examined" by Owens, Reppy, and Turon. A lot of the problems we were coming across and debugging one by one were solved by making these implementations. These rules covered the base cases for most of our derivative operations and provided a way to break infinite loops in most of the cases in which our programs were experiencing them.

To make our project faster, we also used the simplification rules and created a 'simplify' function that takes an ExpTree and returns the most simplified version of it. This ended up being really helpful, because our DFA would compare ExpTrees after calculating the derivative to see if a new state had to be created. We now simplify the derivative after calculation, which makes it so that if two states are equivalent, then they simplify to the same ExpTree and a new state no longer needs to be created.

We were also able to make improvements to the running time of our program by simplifying ExpTrees before the calculation of their respective derivatives. Since our derivative function runs on the order of 2 to the power of the depth of the tree, it's clear to see that the application of a simplification process at this step greatly reduces the depth of the tree and makes it easier to perform calculations.

Our team had a good dynamic working together. We would normally work as a group all at one computer, working through the problem together. Occasionally we would split up and divide and conquer the problem, but that was much less common than pair programming. When we weren't all working together, we each had the chance to work with both of our other group members, one after the other, in scenarios of co-development, and we were pleased with what we were able to accomplish by doing so.

Phase 4 API

Alphabet.java - *produces an alphabet string containing 0-9a-zA-Z.*

Derivative.java - *gets the derivative of an ExpTree.*

- `ExpTree getDerivative(char c, ExpTree expTree)` - *returns the ExpTree derivative of 'expTree' with respect to the character 'c' according to the rules on page 177 of "Regular-expression derivatives re-examined".*

- `int depth(ExpTree t)` - *returns the depth of an ExpTree 't'.*
- `ExpTree simplify(ExpTree t)` - *returns the simplified version of the ExpTree 't' according to the rules listed on page 180 of "Regular-expression derivatives re-examined".*
- `boolean v(ExpTree t)` - *the same v function from page 176 of "Regular-expression derivatives re-examined".*
- `String v2(ExpTree t)` - *a helper function for 'v' to return a string representing the null set or epsilon depending instead of true or false*

DFA.java - *represents a DFA.*

- `class Delta` - *a triple representing a transition, has an int 'current', a char 'letter', and an int 'next' where the ints represent states, and the letter represents the transition between states.*
- `class Trans` - *the list of transitions, contains an array of Deltas as well as the 'size'.*
  - void add(Delta item) - *adds a transition to the end of the list.*
- `class States` - *the list of states, contains an array of ExpTrees as well as the 'size'.*
  - void add(ExpTree item) - *adds a ExpTree state to the end of the list of states.*
- `class DFA` - *the DFA itself, contains a States object, an alphabet, a Transition object, and a Derivative object.*
  - void createDFA(ExpTree regExp) - *creates the DFA with regExp as the starting state.*

ExpTree.java - *a tree that represents a regular expression.*

- `ExpTree(String s)` - *initializes the ExpTree with a string value.*
- `ExpTree(Operation o)` - *initializes the ExpTree with an operation.*
- `boolean isEqual(ExpTree otherTree)` - *returns true if the two ExpTrees are equal, false otherwise.*
- `enum Operation` - `CONCAT`, `STAR`, `UNION`, `INTERSECT`.

RandStrGen.java - *used to generate a random string*

- `RandStrGen(Alphabet A)` - *creates a random string with the full alphabet from Alphabet.java.*
- `RandStrGen(String s)` - *creates a random string with the alphabet given as a string.*
- `String genString(int length)` - *generates a random string of length 'length'.*
- `String[] genStrings(int numStrings, int length)` - *generates 'numStrings' random strings of length 'length'.*

REMatcher.java - *used to see if a string matches a regular expression.*

- `boolean isMatch(String s)` - *returns true if 's' matches an RE given in the constructor, false otherwise.*

Tester.java - *used to test our DFA creation and string matching.*

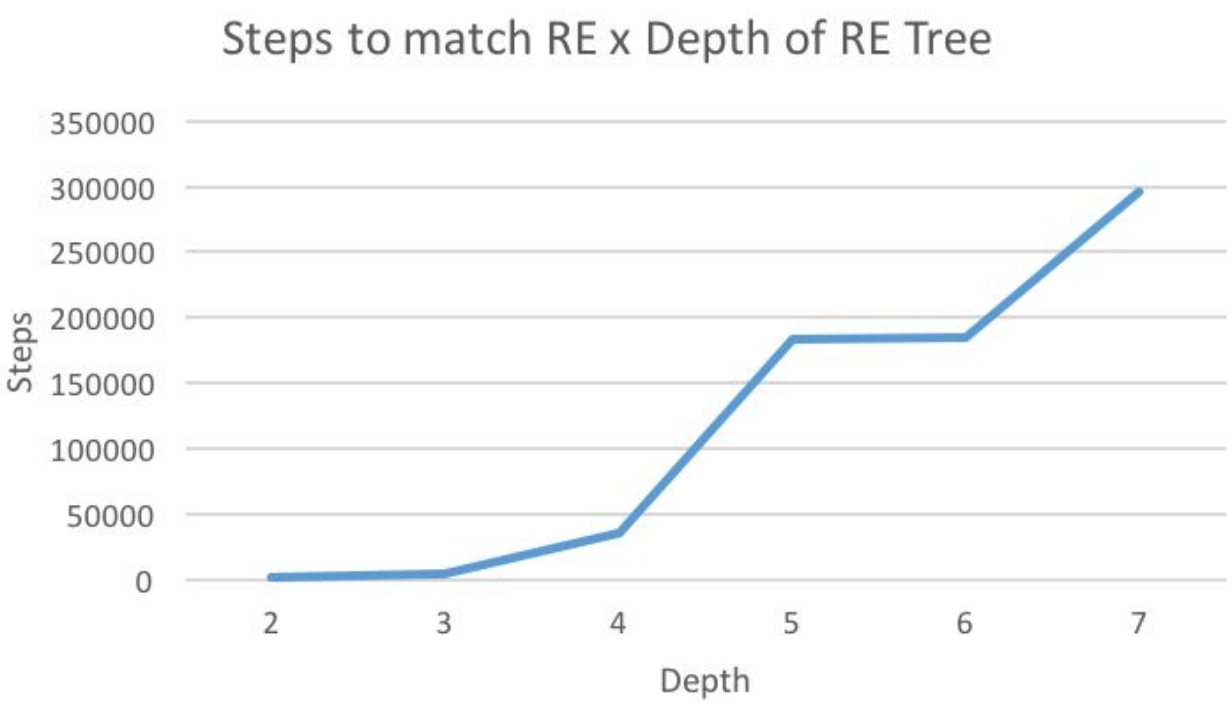
- `MAX_POWER` - *the max length of strings to test*.
- `DEPTH` - *the maximum depth of the randomly generated ExpTree*
- `ALPHABET` - *a string that represents the alphabet for random generation*
- `ExpTree randomRE(int depth, int prob)` - *generates a random regular expression of maximum depth 'depth', 'prob' should be passed as the same as depth when using, it is only there for recursive calls.*

Experimental Results

For the experimental analysis, we decided that we would run 10 trials for each experiment being performed and then take the average of the result founds. This approach was necessary to ensure we did not make assumptions about the speed or memory consumption of our code based on only one case, which could be an outlier. We decided not to count the results of tests run with a DFA of only 2 states because that DFA represents a language that rejects all strings, so the DFA was not as complicated because of our simplification function.

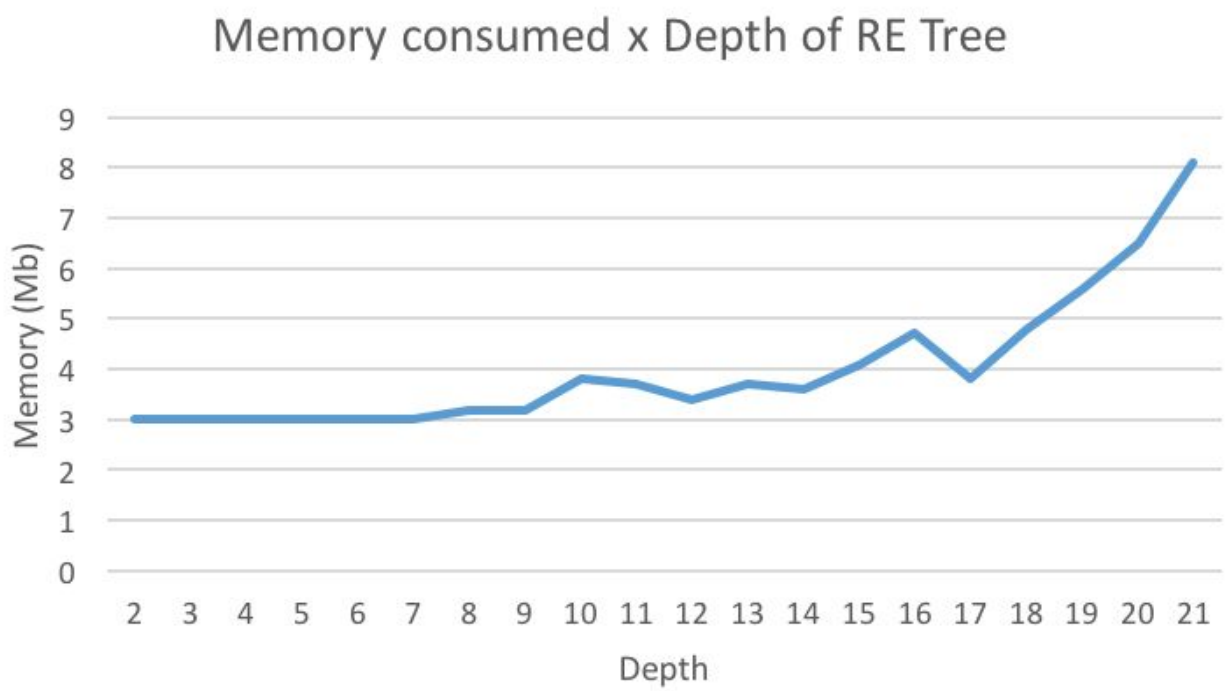
We implemented a function to create a list of strings to be matched to a regular expression, which was randomly generated. For our test we had a list with all strings in an alphabet containing *a*, *b* and *c* up to a maximum length of 5 characters, i.e. a, b, c, aa, ab, ac,, …, cccca, ccccb, ccccc.

The first analysis that we made was about the relation between the number of the states of the DFA and the number of steps necessary to match our strings in the list. For regular expressions trees of a small depth, such as 2 and 3, the number of steps necessary were not high and did not have any remarkable difference between their results. Increasing the depth to 4 showed a slightly increment in the number of steps required to match the strings to the regular expression. For depths 5 and 6 the number of steps were very close to each other, but it rose significantly in relation to the depth 4. For a depth 7 the steps needed continued to grow exponentially.



Graph 1 - Steps necessary to match a list of strings to a regular expression tree by the depth of the RE Tree

The second analysis we made was considering the memory consumed to match the regular expression tree to the list of strings. For depths of regular expression trees less than 10 the memory consumed was always close to 3Mb. After that the trend was always to increase the amount of memory consumed, reaching 4.1Mb with depth 15 and 6.5 with depth 20. The general trend was to increase exponentially the memory used by the tester method. There were some cases that the DFAs had fewer states and so less memory was required to match, because their regular expressions got more complex and thus recognized fewer strings, as in depth 17.



Graph 2 - Memory consumed to match a list of strings to a regular expression tree by the depth of the RE Tree

Improvements

One of the most significant improvements we added to our project from phase 3 to phase 4, was the inclusion of a simplification function. The equivalent relations for REs can be found on the section 4.1 of the paper. Their inclusion helped us in two ways. The first was that we could remove a significant amount of base cases in our getDerivative method. We no longer needed to check for null sets nor epsilon characters, which gave us the possibility to write a much more readable code, implementing strictly the rules given in section 3.1 of the paper for derivatives. The second improvement was to have simpler regular expressions trees, as we could remove unnecessary nodes, as they would still recognize the same language.