Overall good project.
With an additional it would have been Perfect. You would have fixed the termination issues and revised the tests

Patrick, Ethan, Luis
Phase 5 Project Report

Comments

For phase 5 of our project, we took the code from our phase 4 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 size of the tree, it's clear to see that the application of a simplification process at this step greatly reduces the size of the tree and makes it easier to perform calculations.

revise that

?? Sure

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.*
- `SIZE` - *the maximum size of the randomly generated ExpTree*
- `ALPHABET` - *a string that represents the alphabet for random generation*
- `ExpTree randomRE(int size)` - *generates a random regular expression of maximum size 'size'*

Simplify

One of the most important functions that our app is using is the simplify function. This function is used to implement the similarity rules on page 180 of "Regular-expression derivatives re-examined" and simplify some of our regular expressions. While these were not originally included in our phase 3, we were quickly able to realize that the inclusion of this function would solve almost all of the problems that we were having with our DFA creation.

left associative

With our simplify function, one of the main things we were doing was "stacking" the left side of our expression trees. By this, we mean that we would use the communicative properties of some of the regular expressions to make it so that the left side of our regular expression tree was always larger than the right. We would also make sure that if there were two unions, intersections, or concatenations next to each other in the tree, they would be set up so that the "lower size" of the operations was always cascading to the left. This allowed for us to simplify at an even faster rate, and ended some infinite loops that were resulting in some of the previous regular expressions (like (a U aa)*).

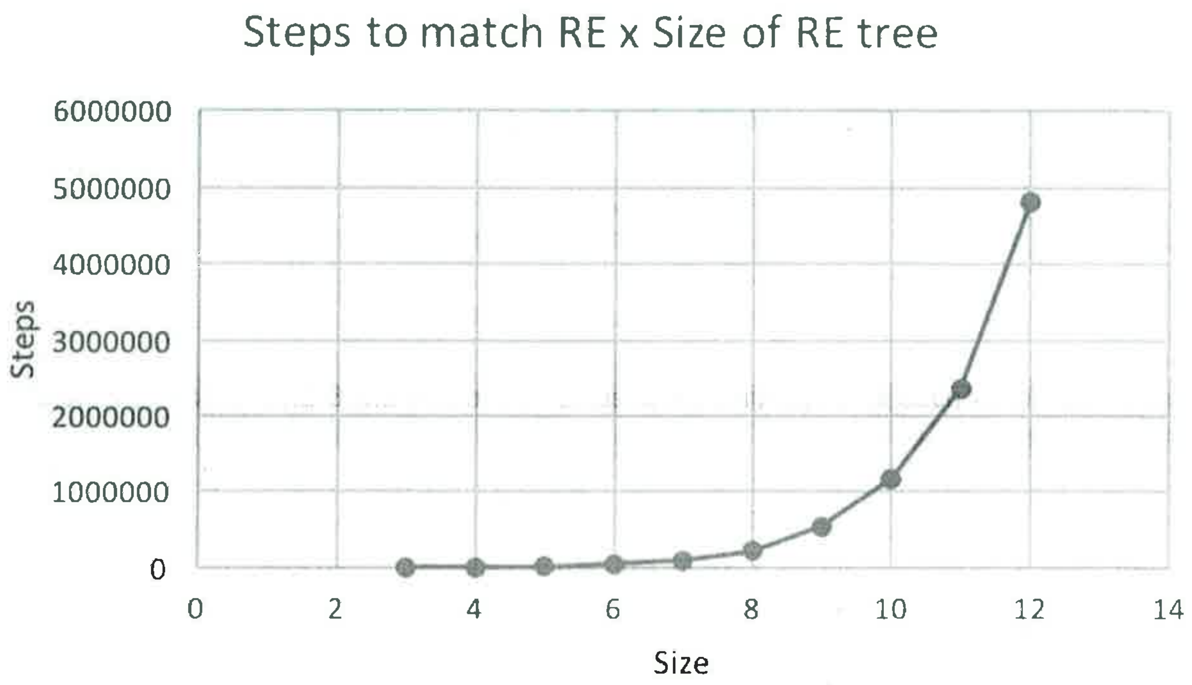
Experimental Results

For the experimental analysis, we decided that we would run 100 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.

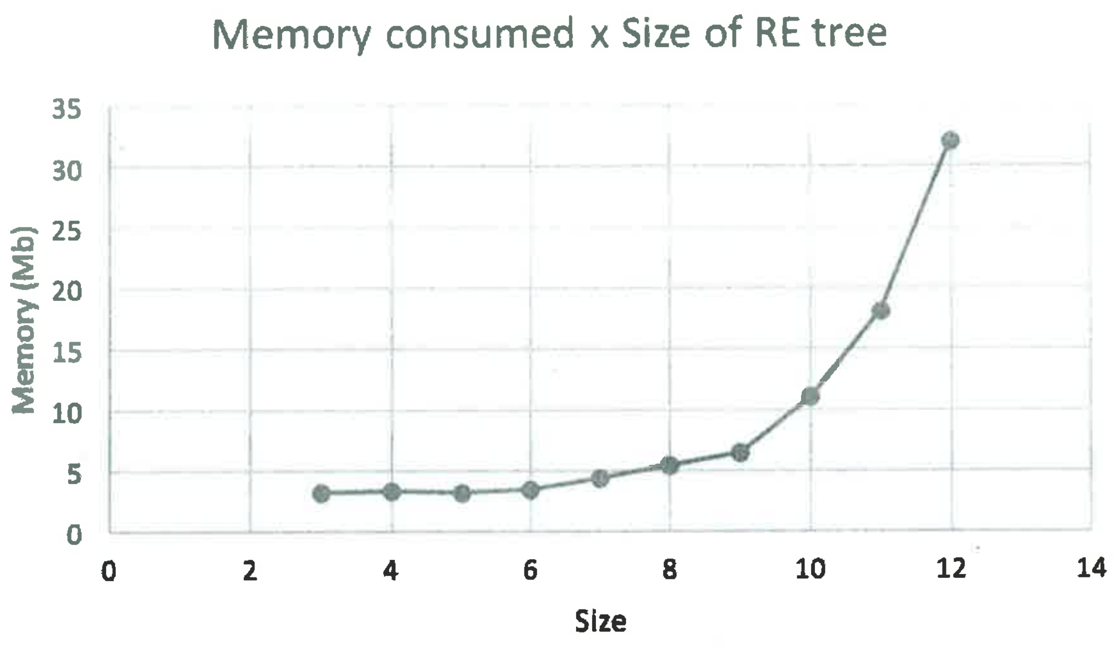
The first variable that we found relevant for our analysis was the time necessary to compute the test for all the different regular expression tree sizes. Unfortunately, the time to run all of the 10 different RE sizes was less than 1 second.

The second 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 size, 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 size to 4 showed a slightly increment in the number of steps required to match the strings to the regular expression. For size 5 and 6 the number of steps were very close to each other, but it rose significantly in relation to the size 4. For a size 7 or higher the steps needed continued to grow exponentially. ?



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

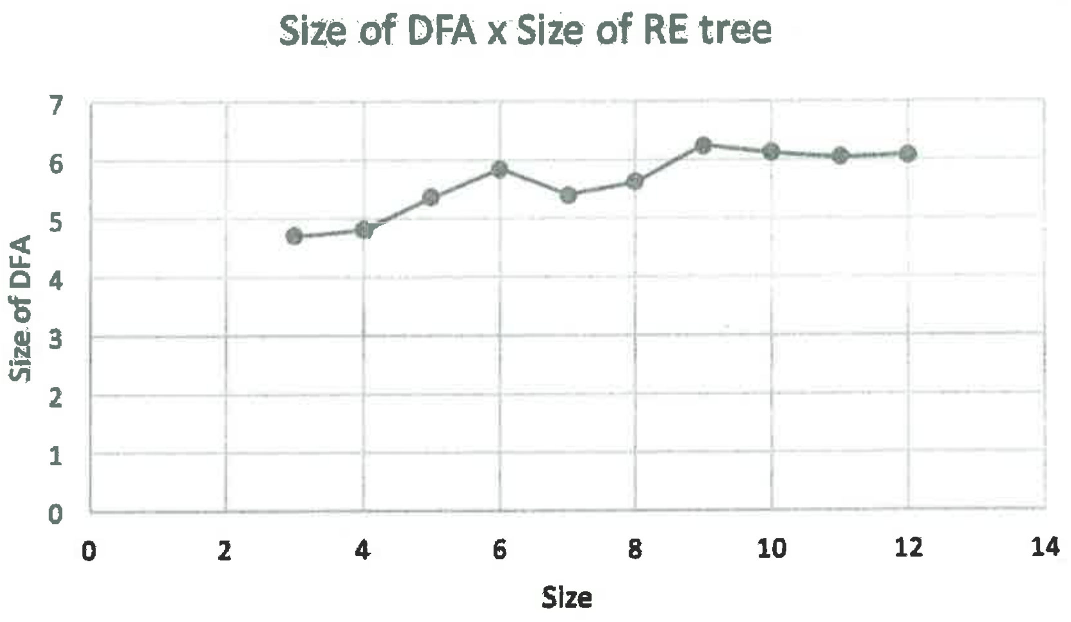
The third analysis we made was considering the memory consumed to match the regular expression tree to the list of strings. For size of regular expression trees less than 10 the memory consumed was always close to 3Mb. After that the trend was always to increase the amount of memory consumed, reaching 11.1Mb with size 10 and 32.1 with size 12. 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 size 12.



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

A fourth variable that we found interesting to analyze was the size of the DFA generated in relation to the size of the regular expression tree. The size of the DFA did not have a significant increase as the size of the RE tree grew. For RE sizes of 3 and 4 the DFA had approximately 4 states. From 5 to 8 the DFAs generated had a size of 5 in average. And with regular expression trees with size greater or equal to 9 the DFA had 6 states in general.

??



Graph 3 - Size of the DFA generated to match a list of strings to a regular expression tree by the size of the RE Tree

Soundness and Completeness

Another important part of our work on this phase has been proving the soundness of our code. In this case, it suffices to show that for an exhaustive number of randomly generated regular expressions, the output of our algorithm exactly matches the output of an external RE matching tool we trust to be correct. We chose to use Java's Regex library to accomplish this task. Our methodology was simple enough. First, we generate 50 random strings of length 10. We then loop over the number of random strings, generating a random regular expression as a tree at each step, simplifying it, converting it to a string RE, and then checking to see if the random string at that step's index in the array was accepted by our tree RE. If the output of our RE matching function differs at any step from the output of Java's Regex pattern matching (`java.util.regex.Pattern`, `java.util.regex.Matcher`), we know that our program is not sound and that our results are flawed, as they do not match the output from our trusted source. However, we are happy to report that after many hundreds of iterations of the abovementioned testing procedure, we never encountered a difference between the behavior of our DFA-based Tree RE matching algorithm and that of Java's Regex library. Thusly, we conclude that our algorithm is sound.

Furthermore, another point we set out to cover in the transition to this phase from the last was the demonstration of the completeness of our algorithm. It is simple enough to see from our code that the function of real importance - `REMatcher.isMatch` - always returns either true (in the case where the `REMatcher` object's stored regular expression accepts the input string) or false (in the case where the input string is rejected). Thusly, as this function always returns, it is complete. Where we continue to run into trouble, however, is in the code upon which `REMatcher` depends - namely, somewhere in the creation and/or running of the constructed DFA. We have now implemented all the simplification rules as set forth by Owens et al. in section 4.1 of their research paper, where previously we had implemented all but the associative rule for unions [(r+s)+t=r+(s+t)]. By rights, we believe this means that our code should now be able to simplify any of the regular expression trees we generate to be able to properly check for equality, but for whatever reason, our program still encounters infinite loops. These scenarios seem to be arising in situations where we generate random RE trees with a good number (>10) of nodes and iterate over large numbers of randomly generated strings used for soundness testing. Because of these situations, we cannot truly say that our algorithm satisfies the requirements for completeness, as it does not produce any output when it encounters an infinite looping scenario. However, these crashes are few and far between, and occur only once or twice per hundred random strings compared against the randomly generated regular expressions.

| YES

??