Good start. For next phase, implement all the necessary rules for ≈
- test using random re
- space and time

4.5/5

Ethan Vynalek, Patrick Bacon-Blaber, Luis Tomazini
CSCI 341
10/29/15

PROJECT PHASE 3: REPORT

We implemented features from the "Using derivatives for RE matching" and "Using derivatives for DFA construction" sections of the article. We approached the problem of development with a "from the ground up" mindset. We started with the building blocks of the software, implementing Alphabet, Expression Tree, and Derivative classes. The Alphabet object is little more than a constructor initialized to create objects housing an alphabet of digits (0-9), lowercase letters, and uppercase letters. The Expression Tree class was created to manipulate and process input in the form of regular expressions (which, themselves, will be passed as Trees). It represents a traditional tree structured object made up of nodes, each with "value", "op", "left", and "right" fields, representing the value (leaf nodes), operation (non-leaf nodes), and left/right children of the node, respectively. The Derivative class is designed to simply compute the derivative of an RE given as a Tree with respect to an input character, as described in section 3.1 of the paper; we use the "helper function" v(r) to detect when an RE r is "nullable," and use Brzozowski's rules for computing the derivative (Owens, 177).

We also designed and implemented a node-based graph-type data type to represent the DFAs we construct for RE matching. In this structure, nodes are be states, and the DFA's transition function is given by the combination of each node's "Trans" object. This object is composed, itself, of an array of "Delta" objects, which are given by the template (current state, input char, next state).

In the next stage of the project, we will use the following experimental techniques to validate the soundness (1), completeness (2), and performance (3).

1) We will reexamine the primary RE matching algorithm implemented in REMatching.java and note, in comments, each step as it corresponds to its respective step in the proof given in the paper. When all steps have been accounted for in the right order, we will know the program is sound.
2) Using RandStrGen.java (generator of strings composed of randomly ordered characters from the given alphabet), we will write another program to test the software against each of the types to which we are, as part of the project, restricting input. Though technically not a measure of *actual* completeness, this will satisfy completeness for the universe in which the only inputs are of the types allowed.
3) When it is complete, we will examine our code and determine its theoretical complexity, then add timers to the central methods to measure its speed against inputs of different sizes. We should then be able to explain how efficient our program is based on the relationships between the runtimes for different inputs. Additionally, we will add space monitors to our code to track exactly how much memory we are consuming, ensure there are no leaks, and help us identify the parts of the software that use the most space.

Please find attached (in our project GitLab repository) the "functionsImplemented.txt" file, listing all functions written and giving short descriptions for each.