Due:Dec 6

Implementation Remember at the start of the semester, you learned about NFAs and DFAs—as you saw in lecture 20 these are graphs, so now that you know about graphs, you can apply your knowledge of graph algorithms to solve problems on regular languages! We are going to solve one particular problem: are two regular expressions equivalent (do they accept the same set of strings) which turns out to combine several graph problems! We have broken this problem down into 3 steps. In each step you solve a different graph problem, and all of them together give you a program which tests two regular expressions for equality. Note that we **strongly** recommend that you test your code extensively in between in each step! We have given you code to help you with the testing.

1. Convert $Regex_1$ and $Regex_2$ to NFA_1 and NFA_2
   We have provided to you code to parse regular expressions (`parse_re`), which takes a regular expression string, and returns a `Regex` object. In `regex.py` you have classes `Regex`, `ConcatRegex`, `StarRegex`, `OrRegex`, `SymRegex`, and `EpsilonRegex` that it builds up a tree. Note that we picked `&` as a symbol for epsilon. To transform Regex to NFA you are going to recurse over this tree and build a graph (We have added a method called `transformToNFA` to all the classes in `regex.py` that you need to implement). `transformToNFA` should create a new object and return it. The ones that recurse will recurse on their children, then modify one of the returned results to return it. You should also complete the NFA class in `nfa.py`. NFA has three fields, `states`, `is_accepting` and `alphabet`. State in `state.py`

has two fields, `id` and `transitions`. We assume that NFA state ids always starts from 0. The method that you need to implement in `nfa.py` are `addTransition`, `addStateFrom`, and `epsilonClose`. You need `addTransition`, and `addStateFrom` for the transformation of `ConcatRegex`, and `OrRegex` to NFA. `addTransition` should add a transition from one state of the NFA to the other input state of the NFA. `addStateFrom` should add all the states from another NFA, and return a mapping of (state number in old NFA $\rightarrow$ state number in this NFA) as a dictionary.

We have provided a method in NFA called `isStringInLanguage` that gets a string and returns True if the string is in the language of this NFA. `testNFA` function in `main.py` gets a regular expression string, a string, and a boolean. You can use it to test if your regex to NFA conversion is correct and accepts the strings in the language of the regex and rejects the strings not in the language of the regex.

The order we recommend you implementing and testing is as follow:

(a) Implement `transformToNFA` in `EpsilonRegex` and test it using `testNFA('\&','', True)`, etc.

(b) Implement `transformToNFA` in `SymRegex` and test it using `testNFA`.

(c) Implement `transformToNFA` in `ConcatRegex` which you need to also implement `addTransition` and `addStateFrom` and then test it using `testNFA`.

(d) Implement `transformToNFA` in `StarRegex` and test it.

(e) Implement `transformToNFA` in `OrRegex` and test it.

(f) Test on regular expressions that have the mixure of all subclasses of regex.

2. Next, we want to compute two NFAs: $NFA_1 - NFA_2$ and $NFA_2 - NFA_1$. If both of these are empty, the NFAs are equivalent. If either accepts any string, that string shows the difference between the two. We can compute $NFA_1 - NFA_2$ and $NFA_2 - NFA_1$ by $\overline{\overline{NFA_1} \cup NFA_2}$ and $\overline{NFA_1 \cup \overline{NFA_2}}$. Doing so requires conversion of the NFA to a DFA, which is another graph algorthm. You use BFS for this conversion. As said in class you can change the goal of BFS according to the problem that you are using it for. You need to implement the `nfaToDFA` function in `main.py`. You can review the NFA to DFA conversion algorithm in the book. Part of the algorithm is that you need to compute epsilon closure of a state which is a set of states which are reachable from this state on epsilon transitions. So you need to implement the method `epsilonClose` in `nfa.py`. It takes a state and returns the epsilon closure of it.

   After you have the DFA you need to stop and test your conversion. So you need to implement the `isStringInLanguage` function in `dfa.py` and then from `main` in `main.py` call the function `testDFA` to test the correctness of your DFA (similiar to the test that you did for NFAs). After you know that your NFA to DFA conversion is correct you can compute the complement of the DFA. You should complete the method called `complement` in your DFA class (`dfa.py`).

   Test your complement functions. You can use `testDFA` again but this time pass the complemented DFA as input. After you are your complement function pass your tests then you can proceed. Remember, it is easier to complement a DFA and it is

easier to union two NFAs. So now that we can compute the complement of DFA_1 and DFA_2, we need to convert the complements to NFAs to do the union. Then convert the resulting NFA to DFA and complement it. After every conversion you need to test the returned NFA or DFA.

3. As you saw in lecture 20, we can run BFS algorithm to find if any string is accepted by DFA. Here if any string is accepted by either $\overline{NFA_1} \cup NFA_2$ or $NFA_1 \cup \overline{NFA_2}$ (the results of the previous step) then it means that $Regex_1$ and $Regex_2$ are not equivalent. The last step to solve the regex equivalence problem is to implement the BFS algorithm to find out if any string is accepted.

   You should implement `shortestString` in `dfa.py`. This function runs BFS on this DFA and returns the shortest string accepted by it. You can use `isStringInLanguage` function to double check if the returned string is in the language of the first regex or the second regex. You should also complete the function `equivalent` in `main.py`. This function gets two regex and then returns a boolean value; `True` if the two regex are equivalent and `False` if they are not. You need to call the functions that you implemented before such as `transformToNFA`, `nfaToDFA`, `dfaToNFA`, `complement` , `union`, `shortestString`.