Joey Robinson
Nick Flouty
Jonathan Gil

Testing Membership of a String in a Regular Expression

Problem Statement:

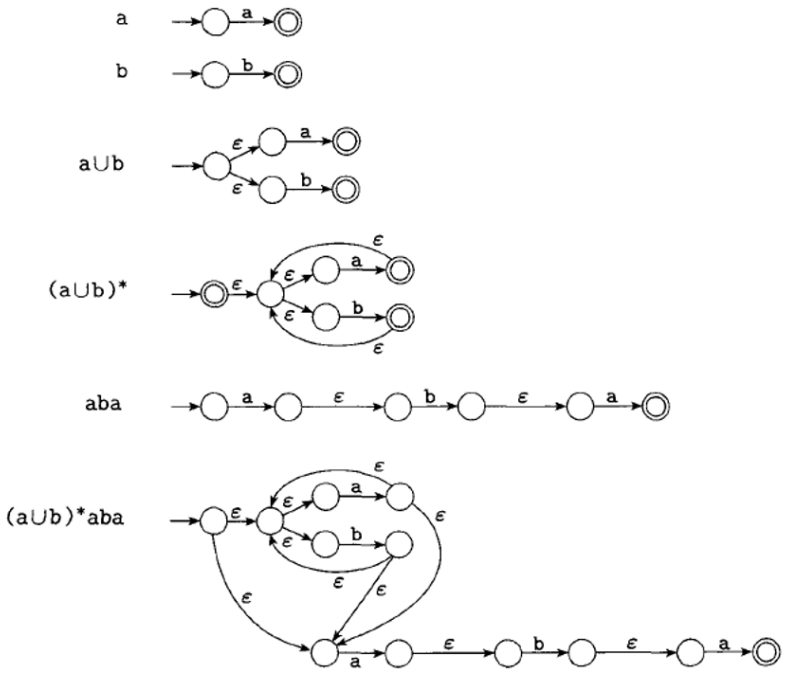
We want to be able to test if a given input string is part of a given regular expression. Based on Kleene's theorem, we know that any regular language is accepted by a finite automaton. This also means that any language accepted by a finite automaton is regular. This information can be used to convert any regular expression into a nondeterministic finite automaton. The objective of this project is to take the user inputted regular expression (using any letter character as the symbols, and using parenthesis, "*", ".", and "+" for operator symbols) convert it into an epsilon NFA, then create an epsilon-free NFA to then be able to test a user inputted string to see if it belongs to the regular expression

Solution Outline:

We are using the Python language to solve this problem.

There are three primary steps to solving our problem:

1. Take as input an infix regular expression, and convert it into postfix and push each symbol onto a stack
2. As the symbols are read, we want to create an e-NFA for every character symbol we read on the stack, and for operating symbols, we want to either concatenate(.), union(+), or use Kleene star operations on the e-NFA's that are in between given parenthesis. In Hopcroft's text *Introduction to Automata Theory, Languages and Computation*, a good example of this process is laid out in images for the regular expression (a+b)*aba. (Remember here, "+" is the same as "U")



3. Our next step is to take the final e-NFA, and figure out the epsilon closure of each state. We can then create new edges based on the epsilon closure of each state so that all epsilon edges are removed, and we are left with an NFA that can be traversed based on the given input string.

Data Structures and Methods:

inputExpression():

This function takes in the user input regular expression, and builds the postfix version of it into an array. Once we have the postfix expression into an array, we can walk through the array to correctly generate the e-NFA

e-NFA Methods:

Our e-NFA is written as a 2D array, with each index in the array having these qualities: [CurrentState, SymbolRead, NextStateFromSymbol].

or_(v1, v2): This function takes as input two e-NFA's (which are popped off the main nfa_stack array) and converts them into a single e-NFA that accepts v1 or v2 (Union Operation).

star_(v1): This function takes as input a single e-NFA and creates a new start state (which also becomes an accepting state), and creates an epsilon move from that new state to the original start state. It then adds epsilon transitions from the original accepting state to the original start state (Kleene Star Operation).

concat_(v1, v2): This function takes as input two e-NFA's and creates epsilon moves from the accept states of v1 to accept states of v2 and make just the v2 accept states accepting. This way, the string must be accepted by v1 AND by v2 in order to constitute the string as accepting from concatenation.

Methods For Removing e-moves:

The e-NFa looks like: [[4e0],[0a1],[1e2],[2b3],[3e5],[5e0],[4e5]] for input (a.b)*. [4e0] represents state 4 with and epsilon edge to state 0.

The e-free NFa looks like: [[4a1],[0a1],[1b3],[2b3],[3a1],[5a1],[4a1]]

Our build_delta function takes an e-NFA as an argument and returns the same list of states but with epsilon-free edges. For each state with an epsilon edge, the algorithm traces through the next state and looks at the outgoing edge that state has. If that next state has an outgoing edge of epsilon we check its next state outgoing edge, until the outgoing edge is not epsilon. When the outgoing edge is not epsilon, we retrace our steps to first state with the epsilon edge and make that edge the first non-epsilon character we saw in that trace, and the state that edge leads to the state that the traced state went to on that edge. The e-free NFA gets passed into another function which removes the duplicate states.

Time Complexity: O(nlog(n)) where n=size of e-NFA

Testing the Input String:

The newly built NFA, user input test string, the starting state, iterator i, accepting states list, and a states_change counter all get passed into def walk_through_nfa() which returns a state arrived at, after reading all of the input string. Starting from the first input, by passing 0 for ‘i’ on first function call, we check if the current_state, the state passed in the function, has an edge corresponding to that index position in input string. If that statement is true, current_state would become the next state. Because this is an NFA, and NFA’s can have states with multiple outgoing edges of the same input, we must account for possible paths it could take on an input. We accounted for it by having a for loop that creates a current_state each iteration. Many cases it would run once, so it would create a new current_state, then move on to the next read input. But in the cases where there there’s multiple options for the state to take, it would take one, then check if that path leads to an accepting state with the remaining input. Whichever path leads to an accepting state with the remaining input, then current_state would become its next_state that lead to the accepting state.

Time Complexity: O(size of test string * size of nfa)

Examples Inputs/Outputs:

```
Enter the infix notation of the Regular Expression : (a.b)*
Enter a test string for the Regular Expression: abababab
This string is Accepted
```

```
Enter the infix notation of the Regular Expression : (a.b)*
Enter a test string for the Regular Expression: abababa
This string is Not Accepted
```

```
Enter the infix notation of the Regular Expression : (a+b).c
Enter a test string for the Regular Expression: ac
This string is Accepted
```

```
Enter the infix notation of the Regular Expression : (a+b).c
Enter a test string for the Regular Expression: cab
This string is Not Accepted
```

```
Enter the infix notation of the Regular Expression : (a.b)*.(c.a)*
Enter a test string for the Regular Expression: abababcaca
This string is Accepted
```

```
Enter the infix notation of the Regular Expression : (a+b)*.c*
Enter a test string for the Regular Expression: bbbbbccccc
This string is Accepted
```

Summary and Conclusion:

For our group, it seemed like a good idea to be able to traverse through a list of the transitions from certain states to determine if a string is accepted by the epsilon-free NFA, which in turn is accepted by the given Regular Expression. This array of transitions can be thought of as the Delta function for the NFA, which is the common method of determining the different transitions from each state. Some of the complications that rose when solving this problem were the following:

- Having multiple transitions from one state to another on the same input symbol
- Figuring out the epsilon closure of states to figure out the new transitions on certain input symbols.

These problems were the most time consuming for solving, especially if the case of the two are combined. We were able to solve these problems by using recursive functions to go through the entire array of transitions and figuring out what needs to happen when we come across these conditions. In the future, this program could feature graphical interpretation of the e-NFA and show the epsilon moves being removed.