Report

Question 3 - DFA Minimization

Reference - https://www.youtube.com/watch?v=0XaGAkY09Wc

I understood the steps that we need to take to minimize DFAs from this video. It was pretty clear that we had to use a recursive method.

I first created somewhat of a matrix using the variable `adj`, where state `a` takes input `i` to go to state `adj[a][i]`. If `adj[a][i]` is -1, then it means that there is no such transition.

Initially, I was thinking of making use of a vector of sets, to keep track of what state is in what set. However, upon further thinking I felt like it would be harder to code it, and would make harder to understand the code as well. Therefore, I instead decided to use a map, which would indicate the id of the set to which a state belonged to.

To check if two states belonged to the same group, I checked if they were transitioning to the same state if they are given the same input, while making sure that if one doesn't have a specific transition, the other doesn't too.

In the recursive method, I would create a new "set of sets". If an element didn't belong to the new set, I would create a set and iterate over all elements and add all those who are equivalent to that state in this step.

If I end up with the same number of states as the previous recursive step, then I don't need to go further and terminate the recursion.

Afterwards, I go through all the original states and transitions, and create new minimized transitions between the sets created using them.

To find out the accepted states in the minimized DFA, I added all the sets to which an original accept state belonged to.

Question 4 - Regex to NFA

For this question, I made use of slightly harder to understand, method, by just pure parsing, and with no additional symbols (except for surrounding the regex with a pair of brackets), and no processing infix to postfix either. No reference was used

I made a struct called NFA, which consists of the start_state, end_state and a list of NFAs. I made use of many extra states, which helped in building the simplicity of the implementation of the code.

I recursively parsed the regex to create an NFA. In this recursion, I have "2 types of NFA/tokens", bracketed NFA, and simple NFA.

Bracketed NFA - When I come across a "(", I begin a "bracketed" NFA, which implies that it'll store NFAs in this manner

$$(NFA_1 + NFA_2 + NFA_3 + \cdots + NFA_n)$$

Basically, storing the NFAs that are separated by +. I created the start_state and the end_state as the opening and closing bracket respectively. To perform the union of all these NFAs to produce a single NFA, I did

$$\text{createTransition}(start\_state, NFA_i.start\_state, E)$$

$$\text{createTransition}(NFA_i.end\_state, end\_state, E)$$

For all values of i

Now to compute the "simple NFAs"

Simple NFA - A simple NFA only consists of concatenation of NFAs, and kleene stars. To start off with a simple NFA, I first created a dummy state, to reduce the edge cases that I might come across.

Then, I did some case work, and defined some instructions on how to react to every symbol.

- If we come across a symbol (a - z), then we need to just concatenate with the previous token in the simple NFA (here, the dummy node helps as we don't need to deal with the edge case that there are no previous tokens). We create a 2 state NFA for every symbol, which makes it easier to implement the $*$ operator. An epsilon transition was made between the end state of the previous token and the start state of the current token, while another transition was made between the start state and the end state of the curren NFA, which takes the corresponding character as input.
- If we come across an opening bracket, it indicates the starting of a bracketed NFA, and we go one level deeper into recursion to create the NFA for it.
- If we see a $*$, then we need to create a transition between the end state of the last NFA,, and the start state of the same NFA. We also need an epsilon transition between start state and end state (as $*$ also allows not taking the token even once). This works clearly for the bracketed NFA, as we have a starting dummy and ending dummy state. It also works for single tokens as well, as we create two states for every symbol.
- If we see a +, it means that the current NFA has completed, and the bracketed NFA that's one level above the simple NFA must go on to creating a new NFA.
- If we see a closing bracket, then it means that we've reached the end of the current simple NFA, as well as the end of the bracketed NFA one level above.