PS7: Non-determinism and Closure

Collaboration: You should work on the problems yourself, before discussing with others, including your cohorts at your cohort meeting. In addition to discussing with your cohortmates, you may discuss the problems with any other current CS3102 students you want, and use any resources you want except for any materials from previous offerings of this course or complete solutions that might be available on the web, which are not permitted. Sharing and subsequently submitting any text, code, images, figures, etc. constitutes plagiarism, so make sure all submitted materials are created exclusively by members of your cohort.

Problem 1: Regular Expression Matching in Software (programming)

The most common way to do regular expression matching in practice is to convert the regular expression into a equivalent non-deterministic finite-state automaton, and then simulate the execution of that automaton on a particular string. While there is a wide diversity of techniques achieve the transformation and simulation (see Hyperscan, Grapefruit, for state-of-the-art examples), the core strategy is exactly the same.

I have written python code to perform regular expression matching using this strategy, employing the constructions exactly as presented in lecture. For this problem, your task is to use this tool to build a function that diagnoses Huntington's Disease.

Follow the instructions in the week 8 guide to complete this problem.

Problem 2: Non-Deterministic Finite State Automata

Give a non-deterministic finite state automaton which decides each of the languages described below, using no more than the number of states indicated. Give both a drawing and a description.

(a) $\{x \in \{0,1\}^* \mid x$ contains the substring 0101$\}$, using no more than five states.

(b) The language described by $0^*1^*0^*$, using no more than three states.

(c) The language described by $1^*(001^*)^*$, using no more than three states.

(d) The language described by $((0 \mid 00)^*(1|\varepsilon))^*$, using no more than one state.

Problem 3: NFA Size

Regular expressions are represented by strings. Consider that we have a regular expression that is n characters long (so if you used a string length method in Python/Java the return value would be n).

Show that the language represented by this regular expression can be computed by a NFA with $O(n)$ states (that is, the function from the length of the a regular expression to the number of states needed for an NFA that decides the language represented by that regular expression is in $O(n)$ where n is the length of the input).

I recommend that you prove this by structural induction. That is, show that the statement is true for the "simplest" regular expressions, then show that if the statement is true for a given regular expression (or expressions) then it is still true after applying any one regular expression operation.

Problem 4: Complement Construction for NFAs

In class we discussed (actually proved, but only verbally) that if a language L was computable with a DFA then $\overline{L}$ (i.e. L complement) was also computable with a DFA. The way we showed this was to take the

DFA for L and invert its final and non-final states (so if $F \subseteq Q$ was the set of final states of the DFA that computes L, the same machine but with $Q \setminus F$ as its set of final states would compute $\overline{L}$).

To summarize the proof of why, any string $w \in L$ will end in a final state when run on the DFA for L and any string $x \notin L$ will end in a non-final state. By inverting which states are final, w will afterwards end in a non-final state and x will end in a final state, so the new machine always has the opposite return value of the original.

We will consider what happens when we try to do the same as the above for NFAs:

– Show that performing the same construction on an NFA that computes language L does not necessarily result in a new NFA that computes the language $\overline{L}$. In other words, provide an NFA with the property that inverting the final vs. non-final states does not cause it compute the complement of its original language.

– Given an NFA to compute L, describe how we could construct a new NFA to compute $\overline{L}$. Justify your construction (informally is fine, a proof is not necessary here).

Problem 5: Operations on Languages

In class, we showed that the Regular languages are closed under various operations. We showed, for example, that when regular languages L_1, L_2 are given as operands to Complement ($\overline{L_1}$)), union ($L_1 \cup L_2$), and intersection ($L_1 \cap L_2$), the resulting language will also always be regular.

Show that the regular languages are closed under each of the operations below.

(a) XOR: a string x is in $XOR(L_1, L_2)$ if $\texttt{XOR}(x \in L_1, x \in L_2)$.

(b) Difference: a string x is in $SUBTRACT(L_1, L_2)$ if $x \in L_1$ and $x \notin L_2$.

(c) $\texttt{OneShorter}(L) = \{x \in \Sigma^* | \exists a \in \Sigma . xa \in L\}$. (In English, the language $\texttt{OneShorter}(L)$ is the set of all strings x for which there is at least one character a such that adding a to the end of the string x would make it a string in L. In other words, it is the set of all strings from L with their last characters removed. So if $1011 \in L$ then $101 \in \texttt{OneShorter}(L)$)

