CSE 311: Foundations of Computing I

Homework 8 (due December 4th at 11:00 PM)

Directions: *Write up carefully argued solutions to the following problems. Your solution should be clear enough that it should explain to someone who does not already understand the answer why it works. However, you may use results from lecture, the theorems handout, and previous homeworks without proof.*

1. State of the Art [Online] (10 points)

For each of the following, create an *NFA* that recognizes exactly the language described.

(a) [5 Points] Binary strings with at least three 1s **or** an odd number of 0s.

(b) [5 Points] Binary strings with at least three 1s **and** ending in 010.

Submit and check your answers to this question here:

https://grinch.cs.washington.edu/cse311/fsm

Think carefully about your answer to make sure it is correct before submitting. You have only 5 chances to submit a correct answer.

2. Regular as Clockwork [Online] (10 points)

For each of the following regular expressions, create an NFA that recognizes the same language.

(a) [5 Points] $((0 \cup 11)^*00)^*$

(b) [5 Points] $(1 \cup 01 \cup 001)^*(\varepsilon \cup 0 \cup 00)$

Submit and check your answers to this question here:

https://grinch.cs.washington.edu/cse311/fsm

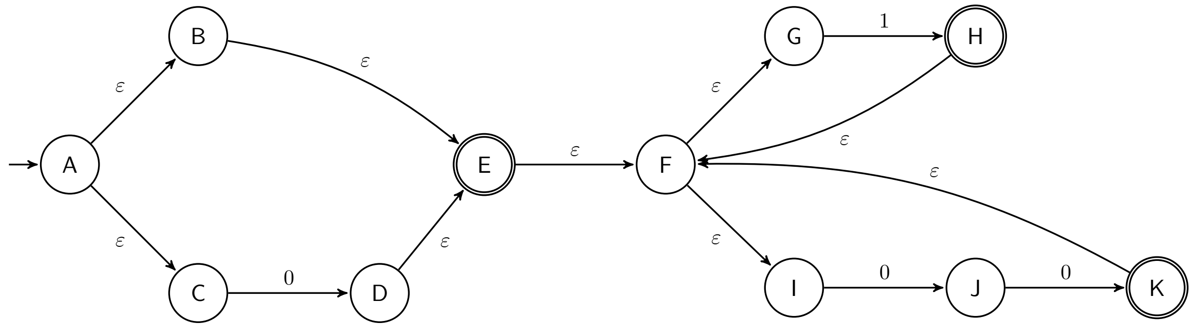
Think carefully about your answer to make sure it is correct before submitting. You have only 5 chances to submit a correct answer.

3. State's Evidence [Online] (20 points)

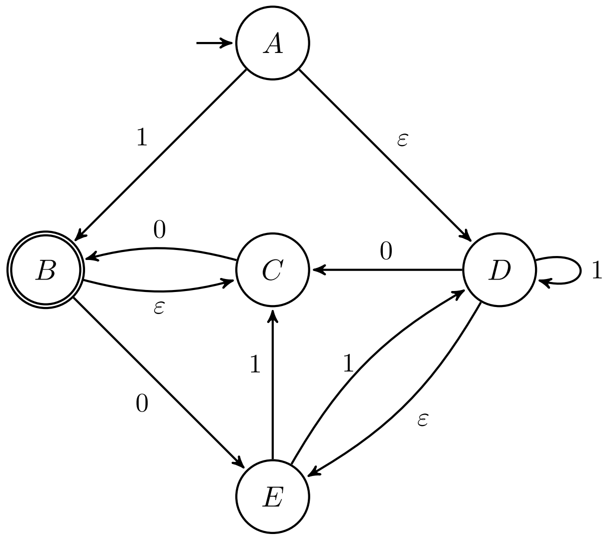
Use the construction from lecture to convert each of the following NFAs to DFAs.

Label each state of the DFA with the corresponding states of the NFA. For example, if the state corresponds to $\{q_0, q_1, q_3\}$, then label the state "q_0, q_1, q_3". The empty state can be labeled "empty set" or similar.

(a) [10 Points] The NFA below, which we get by applying the construction described in class[1] to the regular expression $(\varepsilon \cup 0)(1 \cup 00)^*$:



(b) [10 Points] The NFA below:



Submit and check your answers to this question here:

https://grinch.cs.washington.edu/cse311/fsm

Think carefully about your answer to make sure it is correct before submitting. You have only 10 chances to submit a correct answer.

[1]The only simplification performed was using three states rather than four to represent the sub-expression 00.

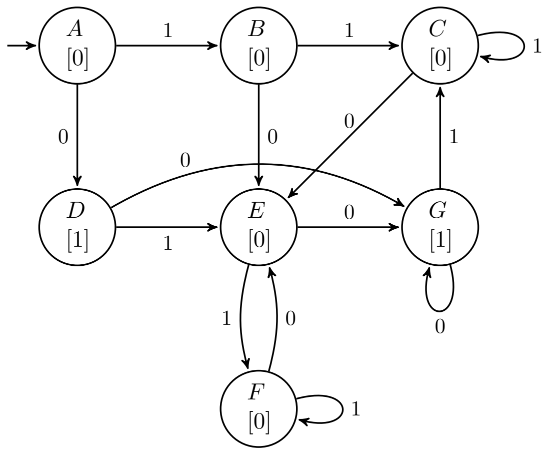
4. Enemy of the State (20 points)

Use the algorithm for minimization that we discussed in class to minimize the each of the following DFAs.

(a) [10 Points] Your solution to Problem 3 (a), which should be a DFA. (Make sure you first submit your solution to that problem and ensure that it is correct before solving this problem.)

You may want to start by renaming the states of that machine since we originally gave them large names, corresponding to the subset of NFA states that they represent.

(b) [10 Points]



Submit and check your answers to this question here:

https://grinch.cs.washington.edu/cse311/fsm

Think carefully about your answer to make sure it is correct before submitting. You have only 10 chances to submit a correct answer.

5. Not Again (25 points)

Let $\mathcal{A} = \{\ldots, p, q, r, \ldots\}$ be a fixed set of atomic propositions. We then define the set $\mathbf{Prop}$ as follows:

Basis Elements For any $p \in \mathcal{A}$, $\texttt{Atomic}(p) \in \mathbf{Prop}$.

Recursive Step If $A, B \in \mathbf{Prop}$, then $\mathsf{NOT}(A) \in \mathbf{Prop}$ and $\mathsf{XOR}(A, B) \in \mathbf{Prop}$.

The set $\mathbf{Prop}$ represents parse trees of propositions that use only the operations of negation (represented by NOT) and exclusive or (represented by XOR).

Next, we define a function $\mathcal{T}$ that takes a parse tree (an element of $\mathbf{Prop}$) as input and returns the proposition that it represents. Formally, we define

$$\begin{aligned} \mathcal{T}(\mathsf{Atomic}(p)) &= p && \text{for any } p \in \mathcal{A} \\ \mathcal{T}(\mathsf{NOT}(A)) &= \neg\mathcal{T}(A) && \text{for any } A \in \mathbf{Prop} \\ \mathcal{T}(\mathsf{XOR}(A, B)) &= (\mathcal{T}(A)) \oplus (\mathcal{T}(B)) && \text{for any } A, B \in \mathbf{Prop} \end{aligned}$$

(Note that this setup is very similar to that of Problem 5 from Section 8. The only differences are that these parse trees support "$\oplus$" rather than "$\wedge$" and "$\vee$".)

Now, let $p \in \mathcal{A}$ be any atomic proposition. The function neg_p takes a parse tree as input and returns another parse tree with all nodes representing p replaced by nodes representing $\neg p$ instead:

$$\begin{aligned} \mathsf{neg}_p(\mathsf{Atomic}(q)) &= \mathsf{NOT}(\mathsf{Atomic}(q)) && \text{if } q = p \\ \mathsf{neg}_p(\mathsf{Atomic}(q)) &= \mathsf{Atomic}(q) && \text{for any } q \in \mathcal{A} \text{ with } q \neq p \\ \mathsf{neg}_p(\mathsf{NOT}(A)) &= \mathsf{NOT}(\mathsf{neg}_p(A)) && \text{for any } A \in \mathbf{Prop} \\ \mathsf{neg}_p(\mathsf{XOR}(A, B)) &= \mathsf{XOR}(\mathsf{neg}_p(A), \mathsf{neg}_p(B)) && \text{for any } A, B \in \mathbf{Prop} \end{aligned}$$

(a) [15 Points] Prove the following: for any $A \in$ **Prop** and any $p \in \mathcal{A}$, we have either $\mathcal{T}(\mathsf{neg}_p(A)) \equiv \mathcal{T}(A)$ or $\mathcal{T}(\mathsf{neg}_p(A)) \equiv \neg\mathcal{T}(A)$.

Note that some facts proven in Problem 4 of Section 2 may be useful here.

(b) [5 Points] Let $A \in$ **Prop** be an expression using only the atomic variables $p, q \in \mathcal{A}$. Describe what the truth table of $\mathcal{T}(A)$ must look like in the case that $\mathcal{T}(\mathsf{neg}_p(A)) \equiv \mathcal{T}(A)$? How about if $\mathcal{T}(\mathsf{neg}_p(A)) \equiv \neg\mathcal{T}(A)$?

(c) [5 Points] We saw in lecture that $\neg$, $\vee$, and $\wedge$ together can express any proposition (e.g., by writing it in sum-of-products form). De Morgan's Law tells us that $\wedge$ can be written instead with $\vee$, so any proposition can be expressed with only $\neg$ and $\vee$. Prove that the same is not true of $\neg$ and $\oplus$.

Specifically, use the results of the earlier parts to show that $p \wedge q$ cannot be represented by any expression using only $\neg$ and $\oplus$.

6. Just Irregular Guy (30 points)

Use the method described in lecture to prove that each of the following languages is **not regular**.

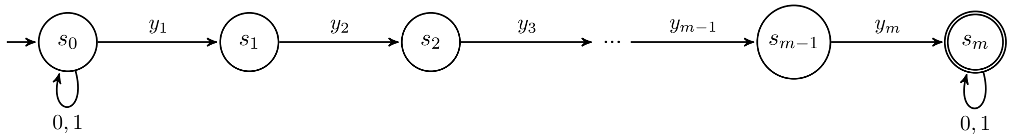
(a) [15 Points] The set of binary strings of the form $\{0^x 1^m 0^y \mid x, m, y > 1 \text{ and } x \equiv y \pmod{m}\}$.

(b) [15 Points] Unicode strings that are syntactically valid Java Script Object Notation (JSON).

(This simple language does not include arithmetic expressions but is already irregular.)

7. Extra Credit: Pratt-Pratt-Pratt (0 points)

Suppose we want to determine whether a string x of length n contains a string $y = y_1 y_2 \ldots y_m$ with $m \ll n$. To do so, we construct the following NFA:



(where the ... includes states $s_3, \ldots, s_{m-2}$). We can see that this NFA matches x iff x contains the string y.

We could check whether this NFA matches x using the parallel exploration approach, but doing so would take $O(mn)$ time, no better than the obvious brute-force approach for checking if x contains y. Alternatively, we can convert the NFA to a DFA and then run the DFA on the string x. *A priori*, the number of states in the resulting DFA could be as large as 2^m, giving an $\Omega(2^m + n)$ time algorithm, which is unacceptably slow. However, below, you will show that this approach can be made to run in $O(m^2 + n)$ time.

(a) Consider any subset of states, S, found while converting the NFA above into a DFA. Prove that, for each $1 \leq j < m$, knowing $s_j \in S$ *functionally determines* whether $s_i \in S$ or not for each $1 \leq i < j$.

(b) Explain why this means that the number of subsets produced in the construction is at most $2m$.

(c) Explain why the subset construction thus runs in only $O(m^2)$ time (assuming the alphabet size is $O(1)$).

(d) How many states would this reduce to if we then applied the state minimization algorithm?

(e) Explain why part (c) leads to a bound of $O(m^2 + n)$ for the full algorithm (without state minimization).

(f) Briefly explain how this approach can be modified to count (or, better yet, find) *all* the substrings matching y in the string x with the same overall time bound.

Note that any string matching algorithm takes $\Omega(m + n) = \Omega(n)$ time in the worst case since it must read the entire input. Thus, the above algorithm is optimal whenever $m^2 = O(n)$, or equivalently, $m = O(\sqrt{n})$, which is the case for normal inputs circumstances.