| | | |
|---|---|---|
| CSE 103 | Computational Models | |
| Fall 2021 | Prof. Fremont | HW 2 |

**(5 questions, 235 points total)**

1. **(50 pts.) Proving a DFA has a particular language**
Suppose we have a robot which moves around in a 1D world, moving either left or right at each time step. We can represent its trajectory as a string over the alphabet $\Sigma = \{\leftarrow, \rightarrow\}$: for example the string $\leftarrow\leftarrow\rightarrow$ would mean that the robot moves left twice, then right once. Suppose that if the robot moves more than $k$ steps left or right of its starting position, it will fall off a cliff and get stuck. Let $L_k$ be the language of all trajectories where the robot does *not* fall; then for example $L_1$ would contain $\rightarrow\leftarrow$ but not $\rightarrow\leftarrow\rightarrow\rightarrow\leftarrow$, since in the latter trajectory the robot falls off the cliff after the 4th movement.

We can build a DFA recognizing $L_k$, but we can't draw it as a graph since $k$ is arbitrary and we'll need a different DFA $M_k$ for each $k \geq 0$. So we'll define $M_k = (Q_k, \Sigma, \delta_k, q_0, F_k)$ mathematically, as follows:

- $Q_k = \{q \in \mathbb{Z} \mid |q| \leq k+1\} = \{-(k+1), -k, \ldots, -1, 0, 1, \ldots, k, k+1\}$
- $\Sigma = \{\leftarrow, \rightarrow\}$
- $\delta_k(q,s) = \begin{cases} q+1 & |q| \leq k \text{ and } s = \rightarrow \\ q-1 & |q| \leq k \text{ and } s = \leftarrow \\ q & |q| > k \end{cases}$
- $q_0 = 0$
- $F_k = \{q \in \mathbb{Z} \mid |q| \leq k\} = \{-k, \ldots, -1, 0, 1, \ldots, k\}$

Intuitively, this DFA works by using states to keep track of where the robot currently is, updating the state at each time step and getting stuck in a non-accepting state if the robot ever moves too far from the start. Notice how we've used integers as states so that we can add or subtract 1 to them to represent the robot moving from one position to the next: you can use any objects as states in a DFA.

Lets formally verify this intuition, and use it to prove the correctness of our DFA.

(a) (*40 pts.*) Prove that $\widehat{\delta_k}(q_0, x)$ (the state reached when running $M_k$ on input $x$) is the position of the robot after following the trajectory $x$, where we consider positions $k+1$ and $-(k+1)$ to indicate having fallen off the right and left cliffs respectively.
(*Hint:* Use induction on the length of $x$.)

**Solution:** We'll prove this by induction on the length of $x$. In the base case $x = \varepsilon$, we have $\widehat{\delta_k}(q_0, \varepsilon) = q_0 = 0$, which is indeed the position of the robot after following the trajectory $x$ (since the robot starts at position 0).

For the inductive case, suppose the claim holds for all strings of length $n$, and take any $x \in \Sigma^{n+1}$. Then writing $x = ya$ with $y \in \Sigma^n$ and $a \in \Sigma$ (so that $a$ is the last symbol of $x$), by the inductive hypothesis we have that $\widehat{\delta_k}(q_0, y)$ is the position of the robot after the trajectory $y$, which we denote $P(y)$. So $\widehat{\delta_k}(q_0, x) = \widehat{\delta_k}(q_0, ya) = \delta_k(\widehat{\delta_k}(q_0, y), a) = \delta_k(P(y), a)$. Now if $|P(y)| > k$, i.e., the robot has already fallen off a cliff after trajectory $y$, then $\delta_k(P(y), a) = P(y)$ by the definition of $\delta_k$, so $\widehat{\delta_k}(q_0, x) = P(y)$ as desired (since if we have fallen off the cliff, we get stuck, so that $P(ya) = P(y)$). If instead $|P(y)| \leq k$, so that the robot has not fallen off a cliff after $y$, then we have two cases: if $a = \rightarrow$, then $\delta_k(P(y), a) = P(y) + 1$, which is the correct position after $ya$ since we move one step to the right; if instead $a = \leftarrow$, then $\delta_k(P(y), a) = P(y) - 1$, which is also the correct position after $ya$ since we move one step to the left. So in all cases, $\widehat{\delta_k}(q_0, x) = \delta_k(P(y), a) = P(ya)$, the correct position after trajectory $ya$.

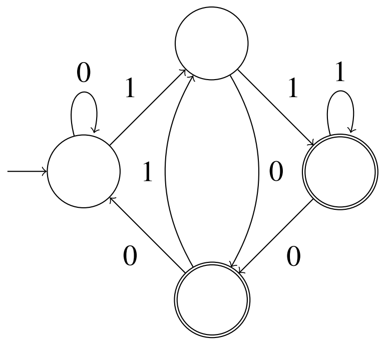
Therefore, by induction we have that $\widehat{\delta_k}(q_0, x)$ is the position of the robot after trajectory $x$ for all $x \in \Sigma^*$.

(b) (*10 pts.*) Using the result of part (a), prove that this DFA actually works, that is, that $L(M_k) = L_k$. (*Hint:* Use the lemma stated in class that $M_k$ accepts $x$ if and only if $\widehat{\delta_k}(q_0, x) \in F_k$.)

**Solution:** As shown in class, $M_k$ accepts $x$ if and only if $\widehat{\delta_k}(q_0, x) \in F_k$, which is equivalent to $|\widehat{\delta_k}(q_0, x)| \leq k$ by the definition of $F_k$. By our result in part (a) above, $\widehat{\delta_k}(q_0, x)$ is the position of the robot after trajectory $x$, so $M_k$ accepts $x$ iff the position of the robot after trajectory $x$ is between $-k$ and $k$. But since the robot gets stuck at position $k+1$ or position $-(k+1)$ if it falls off a cliff, this means that $M_k$ accepts $x$ iff the robot does *not* fall off a cliff when following trajectory $x$. This was the definition of $L_k$, so we have $L(M_k) = L_k$ as desired.

**2. (35 pts.) Defining a DFA symbolically**

For a given integer $k \geq 1$, let the language $L_k$ consist of the strings over the binary alphabet $\Sigma = \{0,1\}$ whose $k$th symbol from the end is 1 (so for example $0100 \in L_3$). We saw an NFA for $L_3$ in class; we can also build a DFA for $L_k$ (directly, without using the NFA-to-DFA conversion we'll cover later). For example, here is a DFA that recognizes $L_2$:



(a) *(30 pts.)* Define a DFA $M_k$ recognizing $L_k$. Since $k$ is arbitrary, you can't draw a graph; define the states $Q_k$, transition function $\delta_k$, etc. mathematically, as we did in question (1). (You do not have to prove that your DFA recognizes $L_k$, but if you'd like more practice with induction, give it a try!)

(*Hint:* The states in the set $Q$ don't have to be numbers; any distinct objects are fine. You could have states labeled by English words, for example, and then it would be well-defined to say something like "all states whose first letter is Z". So if it's convenient, feel free to have the "names" of the states store some useful information rather than just being numbers or letters.)

**Solution:** We will use states to remember the last $k$ symbols of the input string; this is enough to tell whether the string would be in $L_k$ if it ended now. So let $Q = \Sigma^k$ (so that we have one state for every string of length $k$); then given any $q \in Q$, we have $q = q_1 \dots q_k$ where each $q_i \in \{0,1\}$. Given an input symbol $s$, we'll update the history of the last $k$ symbols by "forgetting" the oldest symbol and adding $s$ as the newest one. So we define $\delta(q,s) = q_2 \dots q_k\, s$ (we have shifted the sequence $q$ to the left by one symbol, dropping $q_1$, and appending the new symbol $s$ on the end so that the sequence still has $k$ symbols). We start in $q_0 = 0^k$ to represent the fact that at the beginning of the input we have not seen any 1s so far. Finally, we should accept if the $k$th-to-last symbol so far is a 1, so we put $F = \{q \in Q \mid q_1 = 1\}$.

(b) *(5 pts.)* How many states does your automaton have?

**Solution:** $|Q| = |\Sigma^k| = |\Sigma|^k = 2^k$.

(We'll see later how to prove that this is in fact the *minimum* number of states necessary for any DFA recognizing $L_k$ — there's no way to get around having to remember all possibilities for the last $k$ symbols. This contrasts with the situation for NFAs, where by generalizing the example for $L_3$ we saw in class we can build an NFA for $L_k$ with only $k+1$ states. So this example shows that although NFAs are equivalent to DFAs in terms of what languages they can recognize, they can be exponentially more compact.)

**3. (35 pts.) Does a DFA accept everything?**

(a) *(20 pts.)* Describe an efficient algorithm to decide whether a DFA $D$ accepts all strings, i.e. whether $L(D) = \Sigma^*$. You may use standard graph algorithms as subroutines.

**Solution:** Observe that $L(D) = \Sigma^*$ if and only if $\overline{L(D)} = \emptyset$. So we can first swap the accepting/rejecting states of $D$ to get a DFA $D'$ for its complement, and then apply the algorithm described in class to test emptiness of the language of a DFA. That algorithm simply performs a depth-first search in $D'$ from the start state to see if any accepting states are reachable: if so, $L(D')$ is nonempty and so $L(D) \neq \Sigma^*$; otherwise, $L(D')$ is empty and so $L(D) = \Sigma^*$.

**Alternate Solution:** Another way to view the same algorithm is as a search for reachable non-accepting states: if a non-accepting state is reachable from the start state, then $D$ rejects the word corresponding to the path to that state, so $L(D) \neq \Sigma^*$; conversely, if no non-accepting state is reachable, then every path must end in an accepting state and so $L(D) = \Sigma^*$. So we can perform a depth-first search (or other graph traversal) to see if any non-accepting states are reachable; if so, $L(D) \neq \Sigma^*$, and otherwise $L(D) = \Sigma^*$.

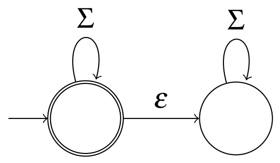
(b) *(5 pts.)* What is the asymptotic runtime of your algorithm, assuming $D$ has $n$ states and $|\Sigma| = m$?

**Solution:** Complementing $D$ takes $\Theta(n)$ time, since we simply go through all states and swap whether they are accepting or rejecting (in fact, we don't even need to do this explicitly, but can simply use DFS to search for non-accepting states as explained in the alternate solution above). Depth-first search on a graph with $V$ vertices and $E$ edges takes $\Theta(V+E)$ time; for $D$ we have $V = n$ and $E = nm$ (since there are $m$ transitions coming out of each state), so DFS takes $\Theta(nm)$ time. Therefore the total runtime of the procedure is $\Theta(nm)$.

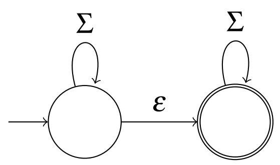
(Note that we would consider this to be a linear-time algorithm, because the encoding of $D$ as a graph has size at least $nm$ since there are $nm$ transitions to specify.)

(c) *(10 pts.)* Would your algorithm also work for an NFA? If so, explain why. If not, give an example of an NFA where your algorithm returns the wrong answer.

**Solution:** This algorithm would not work for an NFA, since the complement construction only works for DFAs (we could add an extra step of converting the NFA to a DFA, but that would make the runtime exponential). For example, consider the following NFA $N$:



$N$ clearly accepts all strings ($L(N) = \Sigma^*$), since we can just stay in the start state. Exchanging accepting/rejecting states gives the NFA $N'$:



However, $L(N') \neq \overline{L(N)}$: in fact, $N'$ also accepts all strings, since we can just take the $\varepsilon$-transition and then remain in the accepting state. Depth-first search on $N'$ will find that the accepting state is reachable from the start state, so our algorithm will wrongly conclude that $L(N) \neq \Sigma^*$.

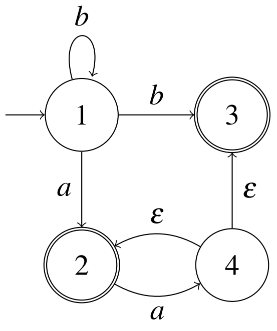
In fact, we'll prove later in the class that unless $\mathsf{P} = \mathsf{NP}$, there is no polynomial-time algorithm to decide whether an NFA accepts everything.

**Alternate Solution:** We can also see the failure of the algorithm from the “search for a reachable non-accepting state” view: in the NFA $N$ above, there is a reachable non-accepting state and yet the language is all of $\Sigma^*$. The problem is that an input word can have multiple possible paths corresponding to it, so just because there is a rejecting path doesn’t mean that there is a rejecting word: the word corresponding to the path may have a different path that is accepting.

**4. (55 pts.) Working with an NFA**
Consider the NFA $M = (Q, \Sigma, \delta, q_0, F)$ where:

- $Q = \{1,2,3,4\}$
- $\Sigma = \{a,b\}$
- $\delta(q,s) = \begin{cases} \{2\} & q = 1 \text{ and } s = a \\ \{1,3\} & q = 1 \text{ and } s = b \\ \{4\} & q = 2 \text{ and } s = a \\ \{2,3\} & q = 4 \text{ and } s = \varepsilon \\ \emptyset & \text{otherwise} \end{cases}$
- $q_0 = 1$
- $F = \{2,3\}$

(a) *(20 pts)* Draw $M$ as a graph.



(b) *(8 pts.)* What are the $\varepsilon$-closures $E(\{1\})$, $E(\{2\})$, $E(\{4\})$, and $E(\{1,4\})$?

**Solution:**

- $E(\{1\}) = \{1\}$
- $E(\{2\}) = \{2\}$
- $E(\{4\}) = \{2,3,4\}$
- $E(\{1,4\}) = \{1,2,3,4\}$

(c) *(15 pts.)* Which of the following strings does $M$ accept: $\varepsilon, aa, b, ba, aaabb$? Give an accepting path (just a sequence of states) for each accepted string.

**Solution:**

$\varepsilon$**:** rejected; the only possible path is $(1)$, and 1 is not an accepting state

$aa$**:** accepted; $(1,2,4)$ is an accepting path ($(1,2,4,3)$ and many other paths also work)

$b$**:** accepted; $(1,3)$ is the only accepting path

$ba$**:** accepted; $(1,1,2)$ is the only accepting path

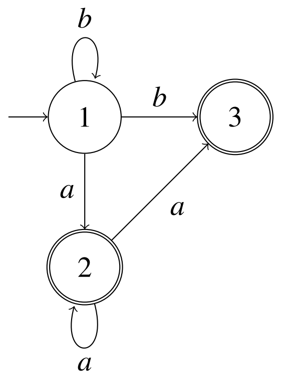
$aaabb$**:** rejected; the only possible partial path is $(1,1,4,2,4)$ (for reading in $aaa = aa\varepsilon a$), at which point we need to read a $b$ and get stuck

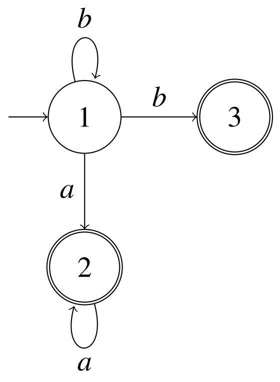
(d) *(12 pts.)* What is the language $L(M)$ of $M$?

**Solution:** Let's consider the two accepting states in turn. To get to state 2, we can read an arbitrary number of $b$s from the start state 1, followed by an $a$; afterward we can return to state 2 by reading an $a$ to get to state 4 and taking the $\varepsilon$-transition back. So the strings with an accepting path ending at state 2 are of the form $b^*aa^*$, and these are also the strings which can end at state 4. To get to the other accepting state 3, we either can take a $b$ from state 1, or the $\varepsilon$-transition from state 4, but then we get stuck at 3 and cannot read any more symbols. So the strings with an an accepting path ending at state 3 are of the form $b^*b \mid b^*aa^*$. Overall, we have $L(M) = b^*aa^* \mid (b^*b \mid b^*aa^*) = b^*aa^* \mid b^*b = b^*(aa^* \mid b)$. In words, this is the language of all strings consisting of zero or more $b$s, followed by either one or more $a$s or a single $b$. Another way of describing the same language is all nonempty strings which do not contain the substring $ab$ (note we need to specify *nonempty* strings since $\varepsilon$ is rejected).

**Alternate Solution:** Another way to view the argument above is to think about progressively simplifying $M$, similar to the construction for converting NFAs to regular expressions. Observe that from state 4 the only outgoing transitions are $\varepsilon$-transitions to states 2 and 3, and the only incoming transition is the $a$-transition from state 2; so we can delete state 4 and re-route the $a$-transition from 2 to go to either 2 or 3 without changing the language, giving the NFA on the left below.

Observe that the only outgoing transition of state 2 is an $\varepsilon$-transition to 4, so we can merge it into 4 without changing the language, giving the NFA on the left below. We can then note that the transition from 2 to 3 does not allow any additional strings to be accepted, since we might as well stay in the accepting state 2. So we can simplify to the NFA on the right below, from which we can directly read off the regular expression $b^*(aa^* \mid b)$ as above.



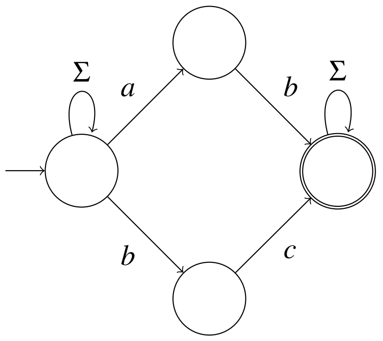


**5. (60 pts.) Designing NFAs**

For each of the following languages, draw an NFA (as a graph) recognizing it. Your NFAs should have no more than 4 states.

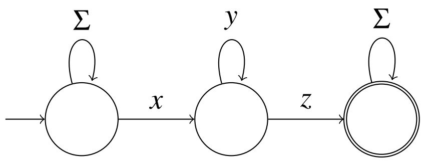
(a) All strings over $\Sigma = \{a, b, c, \ldots, z\}$ containing either the substring *ab* or the substring *bc*.

**Solution:**



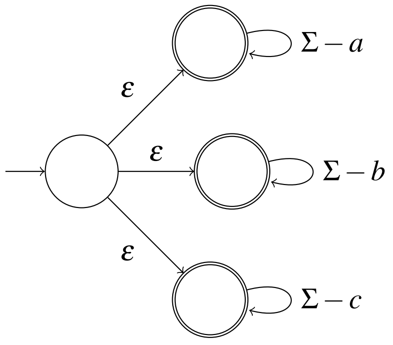
(b) All strings over $\Sigma = \{a, b, c, \ldots, z\}$ which include a pair of $x$ and $z$ with only $y$s in between (e.g. *axyyzb* and *xz* are OK but *xaz* and *xy* are not).

**Solution:**



(c) All strings over $\Sigma = \{a, b, c\}$ which do not contain all three symbols (e.g. *aab* and *bbba* are OK but *bca* is not).

**Solution:** For this language, the most natural approach (to me) would be to build a DFA which has states to remember which symbols we've seen so far; unfortunately, this wouldn't fit in 4 states, since we'd have $\binom{3}{2} = 6$ different combinations we'd have to track before being able to detect the final missing symbol and rejecting. Instead, the key to this problem is to use the NFA's ability to guess: we can guess which of the three symbols will be missing, and then check that that symbol in fact never occurs.



(This language, generalized to use an alphabet of size $n$, turns out to be another example where NFAs are more concise than DFAs: our construction above needs only $n+1$ states, whereas the smallest DFA for the language needs exponentially-many states.)