

## Problem Set 6

---

This sixth problem set explores the regular languages and their properties. This will be your first foray into computability theory, and I hope you find it fun and exciting!

As always, please feel free to drop by office hours, ask on Piazza, or email the staff list if you have any questions. We'd be happy to help out.

Good luck, and have fun!

**Due Friday, November 10 at 2:30PM**

## Problem One: Constructing DFAs

For each of the following languages over the indicated alphabets, construct a DFA that accepts precisely the strings that are in the indicated language. Your DFA does not have to have the fewest number of states possible, though for your own edification we'd recommend trying to construct the smallest DFAs possible.

***Please use our online tool to design, test, and submit your answers to this problem. Handwritten or typed solutions will not be accepted.*** To use the tool, visit the CS103 website and click the “DFA/NFA Editor” link under the “Resources” header. If you’re planning on submitting this assignment in a pair, in your GradeScope submission, please let us know the SUNetID (e.g. htiek, fideles) of the partner who submitted the DFAs so that we can match the problem set to the submitted answers.

Unlike the programming assignments, you will not be able to see the results of the autograder when you submit. As a result, ***be sure to test your solutions thoroughly before you submit!***

- i. For the alphabet  $\Sigma = \{a, b, c\}$ , construct a DFA for the language  $\{ w \in \Sigma^* \mid w \text{ contains exactly two } cs \}$ .
- ii. For the alphabet  $\Sigma = \{a, b\}$ , construct a DFA for the language  $\{ w \in \Sigma^* \mid w \text{ contains the same number of instances of the substring } ab \text{ and the substring } ba \}$ . Note that substrings are allowed to overlap, so  $aba \in L$  and  $babab \in L$ .
- iii. For the alphabet  $\Sigma = \{a, b, c, \dots, z\}$ , construct a DFA for the language  $\{ w \in \Sigma^* \mid w \text{ contains the word } "coco a" \text{ as a substring} \}$ .\*
- iv. Suppose that you are taking a walk with your dog along a straight-line path. Your dog is on a leash that has length two, meaning that the distance between you and your dog can be at most two units. You and your dog start at the same position. Consider the alphabet  $\Sigma = \{y, d\}$ . A string in  $\Sigma^*$  can be thought of as a series of events in which either you or your dog moves forward one unit. For example, the string “yydd” means that you take two steps forward, then your dog takes two steps forward. Let  $L = \{ w \in \Sigma^* \mid w \text{ describes a series of steps that ensures that you and your dog are never more than two units apart} \}$ . Construct a DFA for  $L$ .

## Problem Two: Constructing NFAs

For each of the following languages over the indicated alphabets, construct an NFA that accepts precisely the strings that are in the indicated language. ***Please use our online system to design, test, and submit your automata;*** see above for details. As before, ***please test your submissions thoroughly!***

- i. For the alphabet  $\Sigma = \{a, b, c\}$ , construct an NFA for the language  $\{ w \in \Sigma^* \mid w \text{ ends in } a, bb, \text{ or } ccc \}$ .
- ii. For the alphabet  $\Sigma = \{a, b, c, d, e\}$ , construct an NFA for the language  $\{ w \in \Sigma^* \mid \text{the last character of } w \text{ appears nowhere else in } w, \text{ and } |w| \geq 1 \}$ .
- iii. For the alphabet  $\Sigma = \{a, b\}$ , construct an NFA for the language  $\{ w \in \Sigma^* \mid w \text{ contains at least two } b's \text{ with exactly five characters between them} \}$ . For example, baaaaab is in the language, as is aabaabaaabb and abbbbbbabaaaaaab, but bbbbbb is not, nor are bbbab or aaabab.

\* DFAs are often used to search large blocks of text for specific substrings, and several string searching algorithms are built on top of specially-constructed DFAs. The *Knuth-Morris-Pratt* and *Aho-Corasick* algorithms use slightly modified DFAs to find substrings extremely efficiently.

### Problem Three: $\wp(\Sigma^*)$

Let  $\Sigma$  be an alphabet. Give a short English description of the set  $\wp(\Sigma^*)$ . Briefly justify your answer. (*We think that there is a single “best answer.” You should be able to describe the set in at most ten words*)

### Problem Four: Concatenation, Kleene Stars, and Complements

The regular languages are closed under a number of different operations. This problem explores some properties of those operations.

- i. Prove or disprove: if  $L$  is a nonempty, finite language and  $k$  is a positive natural number, then  $|L^k| = |L|^k$ . Here, the notation  $|L|^k$  represents “the cardinality of  $L$ , raised to the  $k$ th power,” and the notation  $|L^k|$  represents “the cardinality of the  $k$ -fold concatenation of  $L$  with itself.”
- ii. Prove or disprove: there is a language  $L$  where  $\overline{(L^*)} = (\overline{L})^*$ .

### Problem Five: Monoids and Kleene Stars

The Kleene star operator is one of the more unusual operators we’ve covered over the course of the quarter. This problem explores one of its fundamental properties.

Let  $\Sigma$  be an arbitrary alphabet. A **monoid over  $\Sigma$**  is a set  $M \subseteq \Sigma^*$  with the following properties:

$$\varepsilon \in M \quad \forall x \in M. \forall y \in M. xy \in M.$$

Let  $L$  be an arbitrary language over  $\Sigma$  and let  $M$  be an arbitrary monoid over  $\Sigma$ . Prove that if  $L \subseteq M$ , then  $L^* \subseteq M$ .

In the course of writing this proof, please call back to the formal definition of language concatenation and the Kleene star, and use induction as appropriate. Here’s a refresher on the definitions:

$$L_1 L_2 = \{ wx \mid w \in L_1 \text{ and } x \in L_2 \}$$

$$L^0 = \{ \varepsilon \} \quad L^{n+1} = LL^n$$

$$L^* = \{ w \mid \exists n \in \mathbb{N}. w \in L^n \}$$

The result that you’ve shown here is a building block toward a larger result: the Kleene star of a language  $L$  is the smallest monoid containing all the strings in  $L$ .

## Problem Six: Hard Reset Sequences

A *hard reset sequence* for a DFA is a string  $w$  with the following property: starting from any state in the DFA, if you read  $w$ , you end up in the DFA's start state.

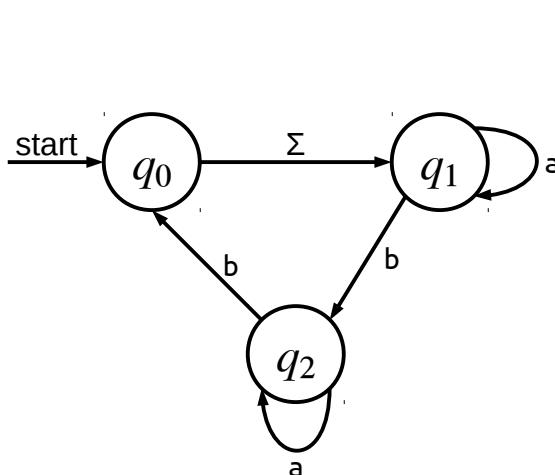
Hard reset sequences have many practical applications. For example, suppose you're remotely controlling a Mars rover whose state you're modeling as a DFA. Imagine there's a hardware glitch that puts the Mars rover into a valid but unknown state. Since you can't physically go to Mars to pick up the rover and fix it, the only way to change the rover's state would be to issue it new commands. To recover from this mishap, you could send the rover a hard reset sequence. Regardless of what state the rover got into, this procedure would guarantee that it would end up in its initial configuration.

Here is an algorithm that, given any DFA, will let you find every hard reset sequence for that DFA:

1. Add a new start state  $q_s$  to the automaton with  $\epsilon$ -transitions to every state in the DFA.
2. Perform the subset construction on the resulting NFA to produce a new DFA called the *power automaton*.
3. If the power automaton contains a state corresponding solely to the original DFA's start state, make that state the only accepting state in the power automaton. Otherwise, make every state in the power automaton a rejecting state.

This process produces a new automaton that accepts all the hard reset sequences of the original DFA. It's possible that a DFA won't have any hard reset sequences (for example, if it contains a dead state), in which case the new DFA won't accept anything.

Apply the above algorithm to the following DFA and give us a hard reset sequence for that DFA. For simplicity, please give the subset-constructed DFA as a transition table rather than a state-transition diagram. We've given you space for the table over to the right, and to be nice, we've given you exactly the number of rows you'll need.



	a	b

Sample hard reset sequence: \_\_\_\_\_

## Problem Seven: Complementing NFAs

In lecture, we saw that if you take a DFA for a language  $L$  and flip all the accepting and rejecting states, you end up with a DFA for  $\bar{L}$ .

Draw a simple NFA for a language  $L$  where flipping all the accepting and rejecting states does not produce an NFA for  $\bar{L}$ . Briefly justify your answer; you should need at most a sentence or two here.

## Problem Eight: DFAs, Formally

When we first talked about graphs, we saw them first as pictures (objects connected by lines), but then formally defined a graph  $G$  as an ordered pair  $(V, E)$ , where  $V$  is a set of nodes and  $E$  is a set of edges. This rigorous definition tells us what a graph actually is in a mathematical sense, rather than just what it looks like.

We've been talking about DFAs for a while now and seen how to draw them both as a collection of states with transitions (that is, as a state-transition diagram) and as a table with rows for states and columns for characters. But what exactly *is* a DFA, in a mathematical sense?

Formally speaking, a DFA is a 5-tuple  $(Q, \Sigma, \delta, q_0, F)$ , where

- $Q$  is a finite set, the elements of which we call *states*;
- $\Sigma$  is a finite, nonempty set, the elements of which we call *characters*;
- $\delta : Q \times \Sigma \rightarrow Q$  is the ***transition function***, described below;
- $q_0 \in Q$  is the start state;
- $F \subseteq Q$  is the set of accepting states.

The transition function warrants a bit of explanation. When we've drawn DFAs, we've represented the transitions either by arrows labeled with characters or as a table with rows and columns corresponding to states and symbols, respectively. In this formal definition, the transition function  $\delta$  is what ultimately specifies the transition. Specifically, for any state  $q \in Q$  and any symbol  $a \in \Sigma$ , the transition from state  $q$  on symbol  $a$  is given by  $\delta(q, a)$ .

This question explores some properties of this rigorous definition.

- i. Is it possible for a DFA to have no states? If so, define a DFA with no states as a 5-tuple, explaining why your 5-tuple meets the above requirements. If not, explain why not.
- ii. Is it possible for a DFA to have no *accepting* states? If so, define a DFA with no accepting states as a 5-tuple, explaining why your 5-tuple meets the above requirements. If not, explain why not.
- iii. In class, we said that a DFA must obey the rule that for any state and any symbol, there has to be exactly one transition defined on that symbol. What part of the above definition guarantees this?
- iv. Is it possible for a DFA to have an unreachable state (that is, a state that is never entered regardless of what string you run the DFA on)? If so, define a DFA with an unreachable state as a 5-tuple, explaining why your 5-tuple meets the above requirements. If not, explain why not.

Going forward, in CS103, we won't use the formal definition of DFAs in our proofs, not because it's not useful, but because it often makes the reasoning a bit harder to follow. However, we thought you should at least see the definition, since it's useful for formalizing what a DFA actually is!

## Problem Nine: Why the Extra State?

In our proof that the regular languages are closed under the Kleene closure operator (that is, if  $L$  is regular, then  $L^*$  is regular), we used the following construction:

1. Begin with an NFA  $N$  where  $\mathcal{L}(N) = L$ .
2. Add in a new start state  $q_{\text{start}}$ .
3. Add an  $\epsilon$ -transition from  $q_{\text{start}}$  to the start state of  $N$ .
4. Add  $\epsilon$ -transitions from each accepting state of  $N$  to  $q_{\text{start}}$ .
5. Make  $q_{\text{start}}$  an accepting state.
6. Make every state besides  $q_{\text{start}}$  a rejecting state.

You might have wondered why we needed to add  $q_{\text{start}}$  as a new state to the NFA. It might have seemed more natural to do the following:

1. Begin with an NFA  $N$  where  $\mathcal{L}(N) = L$ .
2. Add  $\epsilon$ -transitions from each accepting state of  $N$  to the start state of  $N$ .
3. Make the start state of  $N$  an accepting state.
4. Make every other state of  $N$  a rejecting state.

Unfortunately, this construction does not work correctly.

Find a regular language  $L$  and an NFA  $N$  for  $L$  such that using the second construction does not create an NFA for  $L^*$ . Justify why the language of the new NFA isn't  $L^*$ .

## Optional Fun Problem: Why Finite? (1 Point Extra Credit)

The term “finite” in finite automata refers to the fact that each automaton can only have finitely many states. It turns out there's a good reason for that.

We'll say that a **deterministic infinite automaton**, or **DIA**, is a generalization of a DFA in which the automaton has infinitely many different states. Formally speaking, a DIA is given by the same 5-tuple definition as a DFA from Problem Eight, except that  $Q$  must be an infinite set. Since DIAs have infinitely many states, they can't actually be built, and so are mostly an object of purely theoretical study.

Prove that if  $L$  is an arbitrary language over an alphabet  $\Sigma$ , then there is a DIA that accepts  $L$  (that is, the DIA accepts every string in  $L$  and rejects every string not in  $L$ .) To do so, show how to start with a language  $L$ , formally define a 5-tuple corresponding to a DIA for  $L$ , then formally prove that that DIA accepts all and only the strings in  $L$ .