CS103
Spring 2020

Cynthia Bailey Lee
Keith Schwarz

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 SATURDAY, Feb 22nd at 2:30PM

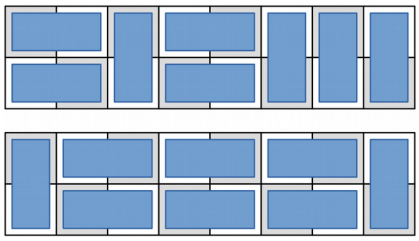
Problem One: Constructing DFAs

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 submit in a pair, in your GradeScope submission, let us know the SUNetID (e.g. htiek, but not 06001234) of the partner who submitted the DFAs.

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. There are many ways to tile a 2 × 8 checkerboard with dominoes, two of which are shown to the right. Notice that the horizontal dominoes must appear as stacked pairs (do you see why?) We can read each tiling from left to right as a string made from the characters `I` and `B`, where `I` denotes "a vertical domino" and `B` denotes "two horizontal dominoes." The top tiling here would be represented as `BIBIII` and the bottom tiling as `IBBBI`.



Let Σ be the alphabet {`B`, `I`}. Construct a DFA for the language { $w \in \Sigma^*$ | w represents a domino tiling of a 2 × 8 checkerboard }.

ii. You're taking a walk with your dog along a straight-line path. Your dog is on a leash of length two, so 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 Σ = {`y`, `d`}. A string in Σ* 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 you take two steps forward, then your dog takes two steps forward.

Let L be the language { $w \in \Sigma^*$ | w describes a series of steps where you and your dog are never more than two units apart }. Construct a DFA for L.

iii. Let Σ = {`a`, `b`, `c`, …, `z`}. Construct a DFA for the language L = { $w \in \Sigma^*$ | w contains the word `cocoa` as a substring }. For example, `mm`**`cocoa`**`mm` $\in L$ and **`cocoa`** $\in L$, but `c` $\notin L$, `chocolate` $\notin L$ (though `cocoa` is a sub***sequence*** of `c`**`h`**`oco`**`l`**`a`**`t`**`e`, it's not a sub***string***), and ε $\notin L$.

Fun fact: DFAs and their variants are often used in string-processing algorithms and data structures. Take CS166 for more details!

Test your automaton thoroughly. This one has some tricky edge cases.

iv. You can approximate the number of syllables in an English word by counting up the number of vowels in the word (including `y`), *except* for

- vowels that have vowels directly before them, and
- the letter `e`, if it appears by itself at the end of a word.

For example, the word `pr`**`o`**`gr`**`a`**`m` has two syllables; the word `p`**`ea`**`ch` has one syllable; the word
`d`**`e`**`d`**`u`**`ce` has two syllables, since the final `e` does not count as a syllable; the word **`o`**`b`**`oe`** has two syllables (although it ends in an `e`, that `e` is preceded by another vowel); the word `wh`**`y`** has

one syllable, since `y` counts as a vowel; and the word **`e`**`nq`**`ueue`** has two syllables. This approach isn't always correct – it will say that **`a`**`r`**`ea`** has two syllables – but it's still a good approximation.

Let Σ = {`a`, `b`, `c`, …, `z`}. Construct a DFA for the language { $w \in \Sigma^*$ | w has at least two syllables according to the above heuristic }.

The strings in this language don't necessarily have to be English words. For example, it contains the nonsense string `wekruvbsdf`.

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!***

As before, while you don't have to submit the smallest NFAs possible, we recommend that you try to keep your NFAs small both to make testing easier and for your own edification.

i. For the alphabet Σ = {`a`, `b`, `c`}, construct an NFA for { $w \in \Sigma^*$ | w ends in `a`, `bb`, or `ccc` }.

While it's possible to do this completely deterministically, it's a bit easier if you use the "guess-and-check" framework we talked about in class.

ii. Let Σ = {`a`, `b`, `c`, `d`, `e`}. Construct an NFA for the language L = { $w \in \Sigma^*$ | the letters in w are sorted alphabetically }. For example, `abcde` $\in L$, `bee` $\in L$, `c` $\in L$, and ε $\in L$, but `decade` $\notin L$.

*You could do this deterministically, but that will require a **lot** of transitions. You can dramatically reduce that number by using ε-transitions strategically.*

iii. For the alphabet Σ = {`a`, `b`, `c`, `d`, `e`}, construct an NFA for the language { $w \in \Sigma^*$ | the last character of w appears nowhere else in w, and $|w| \geq 1$ }.

This problem is all about embracing nondeterminism. Use the "guess-and-check" framework. What information would you want to guess? How would you check it? Please don't try to solve this problem by building a DFA for this language; you'll need at least 50 states if you try to approach things this way.

Stuck? Try reducing the alphabet to two or three letters and see if you can solve that version.

iv. For the alphabet Σ = {`a`, `b`}, construct an NFA for the language { $w \in \Sigma^*$ | w contains at least two `b`'s with exactly five characters between them }. For example, `baaaaab` is in the language, as is `aabaabaaabbb` and `abbbbbabaaaaaaab`, but `bbbbb` is not, nor are `bbbab` or `aaabab`.

The smallest DFA for this language is pretty big, so please don't solve this one deterministically. Embrace the nondeterminism! What would you like to guess? How would you check that your guess is right?

Problem Three: 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 $\overline{L}$.

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

ii. Explain why your result from part (i) doesn't contradict the fact that the regular languages are closed under complementation.

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

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

i. Let L be an arbitrary language over Σ and let M be an arbitrary monoid over Σ. 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. Here's a refresher on the definitions:

$$L_1L_2 = \{\ wx \mid w \in L_1 \text{ and } x \in L_2\ \}$$

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

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

To get yourself acquainted with monoids, try giving three examples of monoids. At least one of them should be finite, and at least one of them should be infinite.

This problem is all about finding the right way to formalize things. Think about, ultimately, what it is that you need to prove. Before you start trying to prove that, break the task down into smaller pieces and make sure you organize everything in a way that makes the logical flow easy to read and rigorously covers all cases. Once you have the setup put together, dive in and fill out each section.

If you'd like to use any properties of Kleene stars, language concatenation, etc. that aren't given in the definition, you'll need to prove them first.

Problem Five: 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.

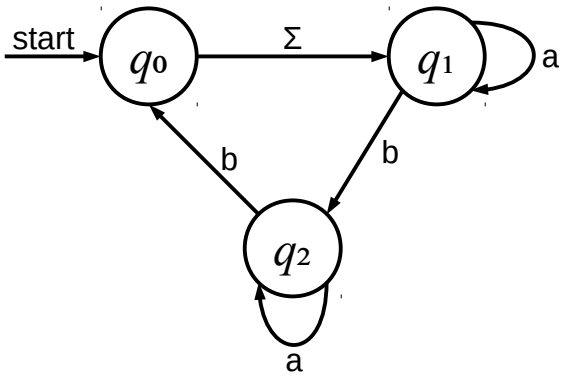
Hard reset sequences have many practical applications. For example, suppose you're remotely controlling a Mars rover whose state you're modeling as a DFA. Imagine there's a hardware glitch that puts the Mars rover into a valid but unknown state. Since you can't physically go to Mars to pick up the rover 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 ε-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: _____

Finding a hard reset sequence for this DFA is a lot easier if you take a few minutes to think about what the power automaton does.

Problem Six: DFAs, Formally

When we talked about graphs, we saw them first as pictures (objects connected by lines), 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*;
- Σ is a finite, nonempty set, the elements of which we call *characters*;
- $\delta : Q \times \Sigma \to Q$ 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 δ 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.

*To define an object using a 5-tuple, use this template: "Let $D = (Q, \Sigma, \delta, q_0, F)$, where $Q = \ldots$, $\Sigma = \ldots$," etc. Defining a DFA requires you to define a transition function δ. To help you see how our study of functions this quarter applies here, we'd like you to formally define δ either using a fixed rule or as a piecewise function. For the purposes of this problem, please **do not** give a picture or table for δ, even though in general these could be valid ways of describing a function. If you've having trouble doing so, it might mean that you picked too complex of a DFA and might want to search for something simpler.*

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.

We've included several optional fun problems here. Feel free to work on any number of them, but please submit at most one of them with your problem set. If you submit more than one problem, we'll select which one to grade arbitrarily and capriciously. ☺

Optional Fun Problem One: Why Finite? (Extra Credit)

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 Nine, except that Q must be an infinite set. Since DIAs have infinitely many states, they're mostly an object of purely theoretical study. You couldn't actually build one in the real world.

Prove that if L is an arbitrary language over some alphabet Σ, 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.

Optional Fun Problem Two: Edit Distances (Extra Credit)

The ***edit distance*** between two strings w and x is the minimum number of edits that need to be made to w to convert it into x. Here, an ***edit*** consists of either adding a character somewhere into w, deleting a character somewhere from w, or replacing a character of w with another. For example, `cat` and `dog` have edit distance 3, `table` and `maple` have edit distance 2, and `edit` and `distance` have edit distance 6.

Let $\Sigma = \{$`w`, `h`, `i`, `m`, `s`, `y`$\}$. Design an NFA for $\{ w \in \Sigma^* \mid$ the edit distance of w and `whimsy` is at most three $\}$. Submit your answer online, and let us know in your written assignment who made the submission.