CS/Math 240: Intro to Discrete Math 5/3/2011

Solutions to Homework 10

Instructor: Dieter van Melkebeek

Problem 1

There were five different languages in Problem 4 of Homework 9.

The Language D_0

Recall that D_0 is the language of ternary representations of multiples of zero. Strings with leading zeros are not allowed, and 0 is the only multiple of zero, so $D_0 = \{0\}$. A corresponding regular expression is 0.

The Language D_1

The only string in D_1 that starts with 0 is the string 0 because leading zeros are not allowed. Since any non-negative integer is a multiple of 1, and since every ternary string starting with a nonzero symbol is a ternary representation of some integer, any ternary string starting with a nonzero symbol is in D_1. One regular expresison that characterizes all ternary strings starting with a nonzero symbol is $(1 \cup 2)(0 \cup 1 \cup 2)^*$. We take the union of this regular expression with 0 to get a regular expression for the language D_1:

$$0 \cup (1 \cup 2)(0 \cup 1 \cup 2)^*.$$

Alternative solution: We can also use the automaton M_2 for D_1 we described on the previous homework, and use ideas from Lecture 24 to construct a regular expression that captures all strings that take M_2 to one of its accepting states. For completeness, we show M_2 in Figure 1.

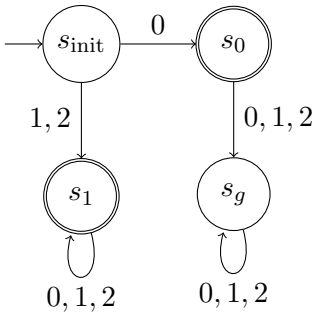

Figure 1: The finite state automaton M_2 for the language D_1.

The only string that takes M_2 from s_{init} to the accepting state s_0 is the string 0. Any other string that starts with a zero takes M_2 from the start state to the garbage state s_g.

To get to state s_1 from the start state, the first symbol of the input must be a 1 or a 2, which is characterized by the regular expression $(1 \cup 2)$. After that, no ternary string can make M_2 leave the state s_1, so the regular expression that captures all strings which take M_2 from the start state to the accepting state s_1 is $(1 \cup 2)(0 \cup 1 \cup 2)^*$.

Our final regular expression that characterizes the language D_1 is, therefore, $0 \cup (1 \cup 2)(0 \cup 1 \cup 2)^*$.

The Language D_2

As was the case for the language D_1, the only string in D_2 that starts with 0 is the string 0. Any other string that starts with a zero has other symbols after it, and is, therefore, not in D_2.

To find a regular expression that characterizes all strings that start with 1 or 2, we use a technique from Lecture 24 which we used for deriving regular expressions for the language of binary strings consisting of an odd number of ones and for the language of binary strings that start and end with the same symbol. Remember that a nonzero string in D_2 contains an even number of ones.

A nonzero string in D_2 can start with 2. After that, it could contain an arbitrary sequence of zeros and twos, which keeps the number of ones even. After reading a 1, there could be another sequence of zeros and twos. Finally, after the next 1, the number of ones in the string becomes even again. This pattern can repeat multiple times and is characterized by the regular expression $((0 \cup 2)^*1(0 \cup 2)^*1)^*$. After the last 1, the string can still contain an arbitrary number of zeros and twos, which is characterized by the regular expression $(0 \cup 2)^*$. Putting this all together, all strings in D_2 that start with a 2 are captured by the regular expression $2((0 \cup 2)^*1(0 \cup 2)^*1)^*(0 \cup 2)^*$.

If the first input symbol is 1, the string can contain an even number of only in the following way. After the first 1, the string can contain an arbitrary number of zeros and twos. After that, it contains another 1, which makes the number of ones even for the first time since the first symbol. The regular expression for this part of the string is $1(0 \cup 2)^*1$. From that point on, the characterization is the same as in the previous paragraph. Thus, the regular expression that characterizes all strings in D_2 starting with 1 is $1(0 \cup 2)^*1((0 \cup 2)^*1(0 \cup 2)^*1)^*(0 \cup 2)^*$.

Combining all the regular expressions we found, we get a regular expression that characterizes the language D_2.

$$0 \cup \Big(2((0 \cup 2)^*1(0 \cup 2)^*1)^*(0 \cup 2)^*\Big) \cup \Big(1(0 \cup 2)^*1((0 \cup 2)^*1(0 \cup 2)^*1)^*(0 \cup 2)^*\Big).$$

Like for the previous language, we can use the automaton M_3 for D_2 we described on the previous homework, and construct a regular expression out of it. For completeness, we show the automaton in Figure 2. The regular expression 0 characterizes all strings that take M_3 from state s_{init} to the accepting state s_0 is the string 0. The regular expressions $2((0 \cup 2)^*1(0 \cup 2)^*1)^*(0 \cup 2)^*$ and $1(0 \cup 2)^*1((0 \cup 2)^*1(0 \cup 2)^*1)^*(0 \cup 2)^*$ characterize all strings that take M_3 from s_{init} to s_0'. The union of the three regular expressions then gives a regular expression that characterizes all strings that take M_3 from the start state to one of its accepting states.

The Language L_1

The language L_1 is almost like D_2, except it additionally contains the empty string and does not contain 0. The nonzero nonempty strings in L_1 are the same as the same as the nonzero strings in

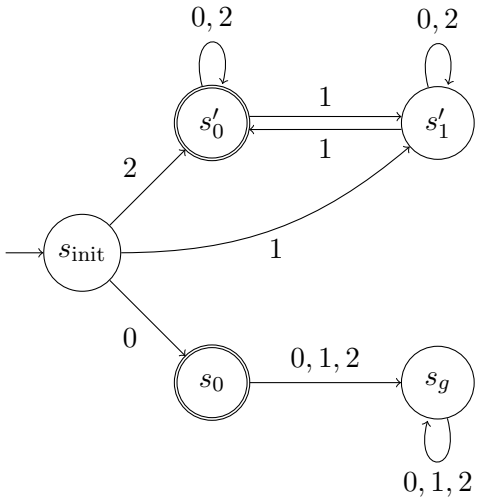

Figure 2: The finite state automaton M_3 for the language D_2.

D_2, and we know a regular expression for those. Since the regular expression for the empty string is ϵ, we get the following as one possible regular expression for L_1:

$$\epsilon \cup \Big(2((0 \cup 2)^*1(0 \cup 2)^*1)^*(0 \cup 2)^*\Big) \cup \Big(1(0 \cup 2)^*1((0 \cup 2)^*1(0 \cup 2)^*1)^*(0 \cup 2)^*\Big).$$

Alternative solution: We show the automaton M_4 for the language $L_1 = (D_2 \setminus \{0\}) \cup \{\epsilon\}$ in Figure 3. It looks almost the same as M_3, except the state s_0 is missing and the start state is now accepting.

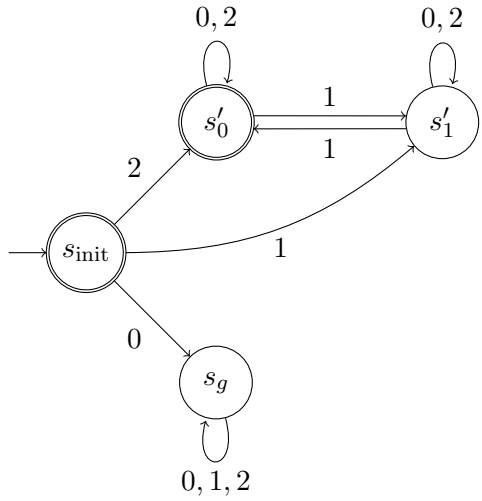

Figure 3: The finite state automaton M_4 for the language L_1.

The only way for M_4 to end in the accepting initial state is if the input is the empty string.

All paths in M_4 that lead to the accept state s_0' from the start state are also present in M_3, and there are no additional paths that lead from the start state to s_0', so the regular expression that captures all strings which take M_4 from the start state to the state s_0' is the same as for M_3.

Hence, the regular expression that characterizes L_1 is

$$\epsilon \cup \Big(2((0 \cup 2)^*1(0 \cup 2)^*1)^*(0 \cup 2)^*\Big) \cup \Big(1(0 \cup 2)^*1((0 \cup 2)^*1(0 \cup 2)^*1)^*(0 \cup 2)^*\Big).$$

The Language L_3

We showed on Homework 9 that $L_3 = D_6$, where D_6 is the language of all ternary representations of multiples of 6. We had an alternative characterization of strings in this language, namely that they contain an even number of ones (so they are multiples of 2), and they end with a zero (so they are also multiples of 3). Thus, the strings in this language are the string 0 and any string that starts with a 1 or a 2, ends with a zero, and contains an even number of ones. We know a regular expression that captures all strings of the latter kind, and we concatenate that regular expression with zero to get a regular expression that captures the nonzero strings in D_6. After that, take the union with the regular expression 0 to get a regular expression for all of D_6.

$$0 \cup \Big(2((0 \cup 2)^*1(0 \cup 2)^*1)^*(0 \cup 2)^*\Big)0 \cup \Big(1(0 \cup 2)^*1((0 \cup 2)^*1(0 \cup 2)^*1)^*(0 \cup 2)^*\Big)0.$$

Problem 2

Consider a finite automaton $M = (S, \Sigma, \nu, s_0, A)$ that accepts the language L. Using M, we would like to construct a new finite automaton that accepts all strings in L that have no nontrivial extension in L. Consider any string $x \in L$. When x is provided as input to M, the machine will end up in an accepting state S. If a path of positive length exists from S to another accepting state T (note that S can be equal to T), then the input along the path is a nonempty string y that can be appended to x to get a nontrivial extension xy of x that belongs to L. So, we would like that no such path exists between any two accepting states.

In order to fulfill the above constraint, we change the set of accepting states, A, of M to A' where A' contains all those states in A from which there is no path of positive length to any other state in A. The resulting machine, $M' = (S, \Sigma, \nu, s_0, A')$ is a finite automaton, which means that the language of all strings in L with no nontrivial extensions in L is a regular language.

Problem 3

Part a

We start by creating a state for each equivalence class; these will be denoted $[x]$, where x is any member of that class (generally, we will use the smallest member, and the first in lexographical order among members of the same length). For example, the equivalence class which contains the empty string is denoted $[\epsilon]$. Our goal is to define a set of transitions between these states, such that for any input x, $\nu(s_0, x) = [x]$. Note that by setting $z = \epsilon$ in the definition of R_L, we have that each equivalence class of R_L either entirely falls within L, or else is disjoint from L. Then, by making the accepting states all equivalence classes that are in L, we have an automaton M_L that accepts L.

We note that for any state $[x]$, $\nu([x], a) = [xa]$; any string in the same equivalence class as x will lead to the equivalence class containing xa when a is appended. This is because for any strings x and y that belong to the same equivalence class, we have by the definition of R_L that $(\forall z)\, xz \in L \iff yz \in L$, which implies that $(\forall z')\, xaz' \in L \iff yaz' \in L$), which means that xa and ya belong to the same equivalence class of R_L. Setting $[\epsilon]$ to be the start state, yield the desired automaton M_L.

Part b

Assume that on an arbitrary finite automaton, for two strings x and y, $\nu(s_0, x) = \nu(s_0, y)$. This means that after processsing x and y, this automaton will be in the same state s_k. Any additional input z from this state will lead to $\nu(s_k, z)$. So $\nu(s_0, xz) = \nu(s_0, yz) = \nu(s_k, z)$. This final state can either be accepting or not accepting; either way xz is accepted (and is thus in L) if and only if yz is accepted. This is our definition for members of the same equivalence class in R_L.

Part c

To show that L is regular if and only if the number of equivalence classes of R_L is finite, we will use the result shown in lecture that a language L is regular iff there exists a finite automaton M that accepts L. If we can show that this occurs if and only if the number of equivalence classes of R_L is finite, we will have proven the original statement due to the transitivity of 'if and only if'.

From our construction in (a), we see that if the number of equivalence classes of R_L is finite, it is possible to create an automaton M_L that accepts L where the states of M are the equivalence classes of R_L. Since the number of equivalence classes of R_L is finite, the number of states of M_L is as well; so it is possible to create a finite automaton that accepts L.

Now, assume that we have a finite automaton M that accepts L, but the number of equivalence classes of R_L is infinite. This means there is at least one state in M that contains members of two different equivalence classes of R_L (we will denote these x and y). But then, $\nu(s_0, x) = \nu(s_0, y)$, which from (b) we know implies that x and y belong to the same equivalence classes of $R_{L(M)}$. This is a contradiction; as such, it is not possible for a finite automaton M to accept L and the number of equivalence classes of R_L to be infinite. In other words, if we have a finite automaton M that accepts L, the number of equivalence classes of R_L is finite.

We have shown that there exists a finite automaton M that accepts L if and only if the number of equivalence classes of R_L is finite; this combined with the fact that a language L is regular if and only if there exists a finite automaton M that accepts L proves our original statement.

To argue that if the number of equivalence classes of R_L is finite, the minimum number of states of any finite automaton accepting L equals the number of equivalence classes of R_L, we use a similar argument to our second direction above. If a finite automaton has less states than the number of equivalence classes of R_L, there are two members of different equivalence classes x and y in the same state. But then $\nu(s_0, x) = \nu(s_0, y)$, a contradiction. So our original statement must hold. Moreover, the automaton M_L from part (a) is an FA accepting L with the allotted number of states.

Part d

We recall that D_6 is the set of ternary strings with no leading zeros that represent multiples of 6. A more useful defintion of this set is that it contains the string 0 and strings with an odd number of 1s that don't start with 0 and end with 0.

We now construct equivalence classes of the relation R_{D_6} and use them to design a finite state automaton for D_6. We first handle strings that start with a zero, and then move on to strings that start with nonzero symbols. For strings that start with a nonzero symbol, we make a distinction between those that are in D_6 and those that are out. As in part (a), the names of the classes will be $[x]$ where x is some string (generally the shortest one) in its equivalence class.

Strings that start with zero (and the empty string): Before we start, note that for any string x starting with a 1 or a 2, there exists a string z such that $xz \in D_6$. If x has an even number of ones, let $z = 0$ and then xz is a string with an even number of ones that ends with zero, which makes it a string in D_6. In fact, if x itself ends with 0, we can just let $z = \epsilon$. Finally, if x contains an odd number of ones, let $z = 10$.

On the other hand, no matter what string we append to a string that starts with a zero and has length at least 2, we end with a string that is out of the language D_6 because D_6 does not allow strings with leading zeros. Call the set of all such strings [00], and note that any two strings in [00] are related by R_{D_6} by the previous sentence. The previous paragraph also implies that strings starting with 1 or 2 belong to different equivalence classes than strings in [00]. Also note that $0 \notin [00]$ because $0 \in D_6$, so $0\epsilon \in D_6$ whereas $x\epsilon \notin D_6$ for any $x \in [00]$. Finally, $\epsilon \notin B_g$ because $\epsilon 0 \in D_6$ whereas $x0 \notin D_6$ for any $x \in C_g$. Thus, B_g is an equivalence class.

Now note that 0 and ϵ are not related by R_{D_6} because $00 \notin D_6$ whereas $\epsilon 0 \in D_6$. Thus, they are in different equivalence classes. We already showed that neither string is in [00], and we now show that their equivalence classes are singleton sets. If x starts with a 1 or a 2, we can extend it to a string in D_6 by appending 00 or 010 depending on whether x contains an even or an odd number of ones. Appending either of those strings to ϵ or 0 yields a string with a leading zero, which is not in D_6. Thus, ϵ and 0 are not related by R_{D_6} to any other strings besides themselves, which means they form singleton equivalence classes under this relation. Call the equivalence classes $[\epsilon]$ and [0], respectively.

Strings that start with 1 or 2: A string that starts with 1 or 2 is in D_6 if and only if it contains an even number of ones and ends with a zero. Let [20] be the set of such strings. Observe that appending any string z to $x \in [20]$ gives us a string with value $3^{|z|}\text{Val}(x) + \text{Val}(z)$, which is divisible by 6 if and only if $\text{Val}(z)$ is divisible by 6. This holds for any string $x \in [20]$, so all strings in [20] are related by R_{D_6}. On the other hand, if $y \notin D_6$, $y\epsilon \notin D_6$ whereas $x\epsilon \in D_6$ for any $x \in [20]$, and we showed earlier that 0 is not related to any string in [20], so [20] is actually an equivalence class.

For the remaining strings that start with a 1 or a 2, we make a distinction between strings with an even and odd number of 1s. Let [2] be the set of strings that start with a nonzero symbol, have an even number of ones, and don't end with a zero. Also let [1] be the set of strings that start with a nonzero symbol, have an odd number of ones, and end with a zero. Appending 0 to a string in [2] yields a string with an even number of ones that ends with zero (so it's in D_6), whereas appending 0 to a string in [1] yields a string with an odd number of zeros (which is not in D_6). Thus, no element of [2] is related to any other element of [1], and we know from earlier that no elements of [2] or [1] are related by R_{D_6} to elements of the other equivalence classes we have found so far.

What remains to show is that all elements of [2] are mutually related, and that all elements of [1] are mutually related.

First consider $x \in [2]$, which is a string that ends with a nonzero (otherwise it would be in [20]) and has an even number of 1s. It is not in D_6, and $x0^r$ is in D_6 for any $r \geq 1$. Now consider extending x with a string z that has at least one nonzero symbol in it. We have $\text{Val}(xz) = 3^{|z|}\text{Val}(x) + \text{Val}(z)$. Since z contains at least one symbol in it and $\text{Val}(x)$ is even, $3^{|z|}\text{Val}(x)$ is divisible by 6. Hence, xz represents a multiple of 6 if and only if $\text{Val}(z)$ is a multiple of 6. Hence, z fully determines whether $xz \in D_6$ for any $x \in [2]$, so all elements of [2] are related by R_{D_6}, which means that [2] is an equivalence class. Here we also remark that the strings z that extend strings in [2] to strings in D_6 are actually strings from the language D_6' that represent multiples of 6 with leading zeros allowed.

Finally, let $x \in [1]$, and consider any string z. Since x contains an odd number of ones, z can make xz an element of D_6 only if it contains an odd number of ones. In that case, consider the first occurrence of 1 in z, and call the portion of z up to the first 1 z', and the remainder of the string z''. Note that xz' is now in [2], and, therefore, $xz \in D_6$ if and only if $z'' \in D_6'$ by the previous paragraph. We see again that z determines whether xz belongs to D_6 or not, so all elements in [1] are related.

Automaton: This completes the construction of equivalence classes for the relation R_{D_6}. We summarize all equivalence classes in Table 1.

Name	Description
$[\epsilon]$	ϵ
[0]	0
[00]	$0(0 \cup 1 \cup 2)(0 \cup 1 \cup 2)^*$
[1]	Strings that don't start with 0 and have an odd number of ones.
[2]	Strings that don't start with 0, have an even number of ones, and don't end with 0.
[20]	Strings that don't start with 0, have an even number of ones, and end with 0.

Table 1: Summary of equivalence classes of R_{D_6}.

Our finite state machine for D_6 has a single state for each equivalence class defined above. The transitions for these states follow from the arguments as well. We show the automaton M_{D_6} in Figure 4. The states are labeled by the name of the equivalence classes.

The class [1] contains all strings starting with a nonzero symbol and containing an odd number of 1s; [2] contains all strings starting with a nonzero symbol, not ending with 0, and containing an even number of 1s; and [20] contains all strings starting with a nonzero symbol, ending in 0, and containing an even number of 1s. We see that, in comparison to the finite state machine for L_1 in Homework 9, this automaton requires only one additional state. This is the state to track whether a string with an even number of 1s ends in 0 or 1. Since D_6 was originally given as $L_1^*\{0\}$, this extension appears reasonable.

In comparison to the finite automaton M_{L_3} in the model solutions for Homework 9, our given automaton has condensed the information represented in the previous solution to keep only what is necessary. Essentially, the information held in the state sets $\{s_1', s_3', s_1'\}$ and $\{s_2', s_4'\}$ is redundant; the former are states with an odd number of 1s, and the latter states with an even number of 1s that end in 0.

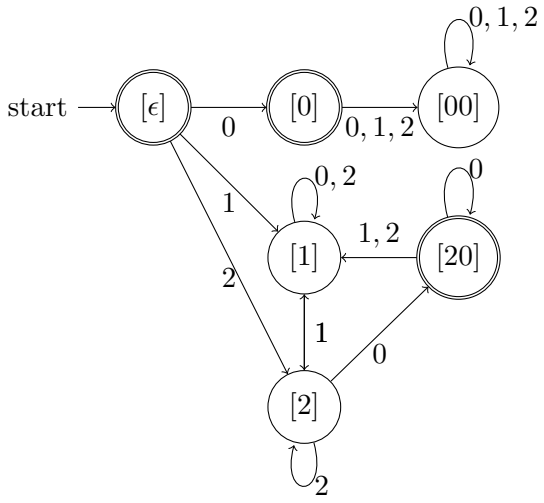

Figure 4: The finite state automaton M_{D_6}.

Problem 4

We start by recalling our result from problem 3; namely, that the minimum number of states of any finite automaton accepting L equals the number of equivalence classes of R_L.

To determine these equivalence classes, consider two arbitrary input strings, x and y. Let us say that x contains the symbol i while y does not. Treating x and y as substrings of a larger input, we see that if we append x or y with a string z that contains all symbols but i, xz should be rejected by a finite automaton while yz should be accepted. This is not possible if the inputs x and y belong to the same equivalence class.

On the other hand, if x and y contain the same subset of symbols, we note that it is the case that $(\forall z)\, xz \in L \iff yz \in L$. We can thus express an equivalence class of R_L by the symbols present and absent in each of the members of that class. For example, all the strings containing only the symbol 1 will be in the same equivalence class (since they will only not be in L when appended with a string containing all other symbols).

The information about what symbols a string contains can be captured using a bit string of length k, where the bit $x_i = 1$ indictates the presence of a symbol i in the string (likewise, $x_i = 0$ indicates its absence). We see immediately that there are 2^k possible such bit strings. So there are 2^k equivalence classes of R_L, and the minimum number of states of any finite automaton accepting L equals 2^k.

For a nondeterministic finite automaton, we recall from our definition that an NFA accepts a given string if and only if there exists a path on the NFA leading to an accepting state. To make a NFA that accepts the language of all strings that does not contain a specific symbol i in the alphabet, then, we can simply exclude any transitions on the input i. Such a NFA would look as follows:

However, this does not quite give us what we desire. We should accept some strings that contain the symbol i; but only those which do not contain some other symbol. By combining k separate states, $s_1, s_2, ..., s_k$, which match the format specified above, we can capture all strings that do not contain every single symbol from 1....k. To connect these states, we need an initial state that leads

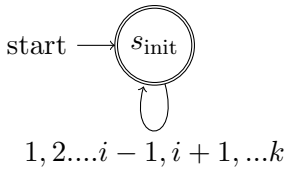

Figure 5: The finite state automaton M_1 for the language $\{x \mid x \text{ does not contain } i\}\}$.

to each of these k states. Such an automaton would look as follows:

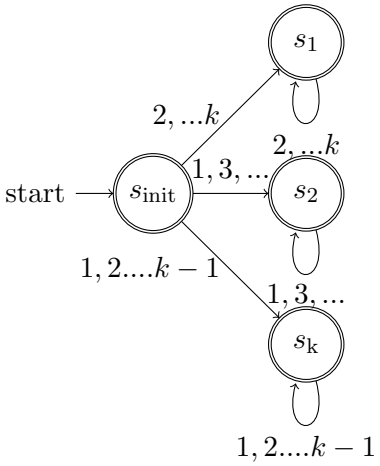

Figure 6: The finite state automaton M_2 for the language $\{x|x \text{ does not contain all of } \{1, 2, ...k\}$.

Problem 5

Let's start with some examples. Consider the regular expression $R = aa^* \cup ab^*$. The languages $L(aa^*)$ and $L(ab^*)$ are not disjoint. Both of them contain the string a. We can modify R to get another regular expression $R' = aa^* \cup abb^*$ such that the languages $L(aa^*)$ and $L(abb^*)$ are disjoint. One can verify that $L(R) = L(R')$.

The second example that we have is more involved. Consider the regular expression $R = R_1 \cup R_2 = a^*b^*a \cup ab^*a^*$. The languages generated by R_1 and R_2 have many more strings in common than the earlier example. Let's work out a regular expression R' such that $L(R) = L(R')$ but all the unions appearing in R' are amongst disjoint regular expressions. Further, let's keep R_1 unchanged. Then, we need to suitably modify R_2 so that it does not generate strings represented by R_1. The strings generated from R_2 with no contribution from b^* are also contained in $L(R_1)$. So, b^* should contribute at least one b. So we now need to modify the expression $R'_2 = abb^*a^*$. We can divide R'_2 into disjoint sub expressions – $R'_2 = abb^* \cup abb^*aa^*$. In the first expression a^* does not contribute any characters. The strings generated by it are not in $L(R_1)$ since all strings in $L(R_1)$ end with an a. The strings generated by the second expression in R'_2 can be generated using R_1 if a^* does contribute any characters. Hence, we add another a to the expression to get $R'_2 = abb^* \cup abb^*aaa^*$. Hence, $R' = R_1 \cup R'_2 = a^*b^*a \cup abb^* \cup abb^*aaa^*$.

From the examples discussed above, it seems that it is always possible to rewrite a regular

expression such that all unions involved are disjoint. In order to prove this conjecture, induction is a possible candidate strategy. In this case, it is tricky to set up the induction scheme. The problem is the choice of the induction parameter. The standard choice as suggested by the recursive definition of a regular expression, namely the number of regular operations in the expression, does not work. It is possible to get induction to work but there is a nicer approach.

In class, we saw constructions to go from a regular expression R to a finite automaton M such that $L(M) = L(R)$, and vice versa. Given a regular expression R, applying the first construction, and then the second, yields a regular expression R' that is equivalent to R, i.e., $L(R) = L(R')$. R' will typically be much more complicated than R but it (almost) has the disjointness property we want. The reason for the latter is that R' is constructed by breaking up the strings in $L(M)$ based on the type of path they induce in the automaton M.

It turns out we need a minor modification to the construction from class. Let us label the states of M as $1, 2, \ldots, |S|$ as before. We now construct regular expressions $R_{i,j,k}$ that describe all nonempty strings that lead M from state i to state j using only states with labels at most k as intermediate states. The minor difference with the construction from class is that we discard the empty string.

As before, we construct the expressions $R_{i,j,k}$ inductively. The base cases $R_{i,j,0}$ now either consist of a single alphabet symbol, or are empty. The induction step

$$R_{i,j,k+1} = R_{i,j,k} \cup R_{i,k+1,k}(R_{k+1,k+1,k})^* R_{k+1,j,k}$$

still holds (verify this yourself!). The union involved is disjoint because the first term corresponds to paths from i to j that only use states up to k as intermediate states, and the second term to paths from i to j that use state $k+1$ and no state higher than $k+1$ as an intermediate state. The nonempty strings in $L(M)$ are described by the regular expression $\cup_{s \in A} R_{s_0,s,|S|}$, which is a disjoint union because each term corresponds to paths that start from the same state s_0 but end in different states s. Finally, we include the empty string if needed using one more disjoint union.

As an exercise, you should convince yourself that we do need to exclude the empty string from the sets R_{ijk} to realize our goal.

Extra Credit

We make use of the tools developed in Problem 3. There, we constructed an automaton with six states for the language D_6 using equivalence classes of the relation R_{D_6} in part (d) of Problem 3. Now we describe the equivalence classes of R_{D_k} for any k. The number of equivalence classes gives us a lower bound on the number of states a finite automaton that recognizes D_k must have by part (c) of Problem 3.

Write $k = 3^\ell \cdot m$ where 3 does not divide m. Let $x = x_n x_{n-1} \ldots x_1 x_0$ be a ternary string, and let the integer value corresponding to the string x be

$$\mathrm{Val}\,(x) = \sum_{i=0}^{n} x_i 3^i. \tag{1}$$

First observe that the terms in (1) with $i \geq \ell$ are all divisible by 3^ℓ, and that the sum $\sum_{i=0}^{\ell-1} x_i 3^i$ is divisible by 3^ℓ if and only if $x_i = 0$ for all $i \in \{0, \ldots, \ell - 1\}$ (this is because the largest this sum

can be is $3^\ell - 1$ when all the x_i are equal to 2; you can prove this by induction). It follows that the ternary representation of every nonzero integer that is a multiple of k ends with ℓ zeros.

Second, suppose $\mathrm{Val}\,(x)$ is a multiple of k, so it has the form $y0^\ell$. Note that $\mathrm{Val}\,(x) = kq = 3^\ell mq$, so $\mathrm{Val}\,(x)\,/3^\ell = mq$, which is divisible by m. Also, $\mathrm{Val}\,(y) = \sum_{i=\ell}^{n} x_i 3^{i-\ell}$, and x_0 through $x_{\ell-1}$ are all zero, so

$$\frac{\mathrm{Val}\,(x)}{3^\ell} = \frac{\sum_{i=0}^{n} x_i 3^i}{3^\ell} = \sum_{i=\ell}^{n} x_i 3^{i-\ell} = \mathrm{Val}\,(y)\,.$$

It follows that $y \in D_m$. Since x was an arbitrary string in D_k, it follows that every string in D_k except for the string 0 is a concatenation of a string in D_m with a string of ℓ zeros.

Finally, the converse of the last two paragraphs also holds. If $\mathrm{Val}\,(x)$ is divisible by m, so is $3^\ell \mathrm{Val}\,(x)$, and the ternary representation of this number is the string x followed by ℓ zeros. Thus, we have shown that a string that is not zero represents a multiple of k if and only if it is a concatenation of a string that represents a multiple of m with a string of ℓ zeros.

Now let's describe the equivalence classes of the relation R_{D_k} using the observations we've made. We summarize them in Table 2. The strings in C_i represent all possible multiples of m, and are separated into $\ell + 1$ categories depending on the number of zeros they have at the end. The strings in C'_i represent numbers that either aren't multiples of m (when $i > 0$), or that are multiples of m (when $i = 0$) but don't end with a zero.

There are a total of $3 + \ell + m$ classes and they form a partition of the set of all ternary strings. Also observe that $D_k = [0] \cup C_\ell$ and $D_m = \bigcup_{i=0}^{\ell} C_i$.

	Name	Description
	$[\epsilon]$	ϵ
	$[0]$	0
	$[00]$	$0(0 \cup 1 \cup 2)(0 \cup 1 \cup 2)^*$
For $i \in \{0, \ldots, \ell - 1\}$:	C_i	Strings in D_m that don't start with 0 and end with exactly i zeros.
	C_ℓ	Strings in D_m that don't start with 0 and end with at least ℓ zeros.
	C'_0	Same as C_0. Not a new class; defined only for convenience
For $i \in \{1, \ldots, m - 1\}$:	C'_i	Strings x that don't start with 0 and with $\mathrm{Val}\,(x) \bmod m = i$

Table 2: The equivalence classes for the relation R_{D_k}.

We need to show that it is possible to extend any string that starts with a 1 or a 2 to a string that belongs to D_k. Suppose x is one such string. We construct a ternary string of length $\lceil \log_3(m) \rceil$ that does the job. If z has $\mathrm{Val}\,(z) \in \{0, \ldots, m - 1\}$. Thus, one possibility for z yields a string xz such that $\mathrm{Val}\,(xz) \bmod m = 0$, and $xz \in D_m$. In particular, we can pick z so that $\mathrm{Val}\,(z) = m - (3^{\lceil \log_3(m) \rceil} \mathrm{Val}\,(x) \bmod m)$. Also observe that now we can concatenate xz with ℓ zeros to get a string in D_k.

We now show that two elements of any given class are related by R_{D_k}. This is obvious for the classes $[\epsilon]$ and $[0]$ since they only contain one element each. Any string with a leading zero followed by at least one more digit is not in the language, so no matter how we extend it, we cannot get a string in D_k. Thus, all pairs of elements of $[00]$ are also related by R_{D_k}.

Now consider two strings $x, y \in C_i$ where $i \in \{0, \ldots, \ell\}$. There are two cases to consider.

Case 1: z consists only of zeros. Then the strings xz and yz are both strings in D_m that end with $i + |z|$ zeros, and note that the number of trailing zeros in a string in D_m determines whether it is in D_k or not. Thus, for all such strings z, $xz \in D_k \iff yz \in D_k$.

Case 2: z contains at least one nonzero symbol. Say $z = 0^r z'$ where the first symbol of z' is nonzero. Then the strings $x' = x0^r$ and $y' = y0^r$ satisfy $xz = x'z'$ and $yz = y'z'$, and also $x', y' \in D_m$ because $\text{Val}(x') = 3^r \text{Val}(x)$, so it's a multiple of m because $\text{Val}(x)$ is. The same holds for y'.

We now make two observations about z' in this case. Since $x', y' \in D_m$, m divides both $\text{Val}(x')$ and $\text{Val}(y')$, and also $3^{|z'|}\text{Val}(x')$ and $3^{|z'|}\text{Val}(y')$. We also have $\text{Val}(x'z') = 3^{|z'|}\text{Val}(x') + \text{Val}(z')$ and $\text{Val}(x'z') = 3^{|z'|}\text{Val}(x') + \text{Val}(z')$, so m divides $\text{Val}(x'z')$ and $\text{Val}(y'z')$ if and only if $\text{Val}(z')$ is a multiple of m, or, in other words, if $z' \in D_m$. Also, since z' starts with a nonzero symbol, the number of zeros at the end of z' determines whether $x'z'$ and $y'z'$ belong to D_k. Thus,

$$xz \in D_k \iff x'z' \in D_k \iff z' \in D_k \iff y'z' \in D_k \iff yz \in D_k,$$

and we see that any two elements in C_i are related by R_{D_k}.

Now suppose $x, y \in C'_i$ where $i \in \{1, \ldots, m-1\}$, and consider any string z. Recall that $\text{Val}(x) \bmod m = i$ and $\text{Val}(y) \bmod m = i$. We have $\text{Val}(xz) = 3^{|z|}\text{Val}(x) + \text{Val}(z)$ and $\text{Val}(yz) = 3^{|z|}\text{Val}(y) + \text{Val}(z)$, and this equality also holds modulo m. Thus, $xz \in D_m \iff yz \in D_m$. Also note that z contains at least one nonzero symbol in it when $xa \in D_m$ because otherwise concatenating x with z would yield an integer of the form $3^r\text{Val}(x)$ which is not divisible by m because $\text{Val}(x)$ is not divisible by m and 3 does not divide m. Thus, we only get $xz \in D_k$ if z ends with ℓ zeros, and in that case we also have $yz \in D_k$. It follows that any two strings in C'_i are related by R_{D_k}.

Finally, we need to show that the partition classes we described are actually equivalence classes. That is, we show that no two elements of different partition classes are related by R_{D_k}.

We argued in the solution to part (d) of Problem 3 that $[\epsilon]$, $[0]$, and $[00]$ are equivalence classes. The same argument applies here.

Next, consider two classes C_i and C_j with $0 \le i < j \le \ell$. Let $x \in C_i$ and $y \in C_j$, and let $z = 0^{\ell - j}$. Then $xz \notin C_\ell$ whereas $yz \in C_\ell$, so $xz \notin D_k$ whereas $yz \in D_k$. It follows that x and y are not related by R_{D_k}.

Now consider classes C_i and C'_j where $i \in \{0, \ldots, \ell\}$ and $j \in \{1, \ldots, m-1\}$, and let $x \in C_i$ and $y \in C'_j$. Pick $z = 0^{\ell - i}$, and note $xz \in C_\ell$ whereas $yz \notin D_m$ for some $r \in \{1, \ldots, m-1\}$. It follows that $xz \in D_k$ whereas $yz \notin D_k$, and that x and y are not related by R_{D_k}. So now we know that the classes C_i for $i \in \{0, \ldots, \ell\}$ are also equivalence classes.

Finally, consider classes C'_i and C'_j with $1 \le i < j \le m-1$. Let $x \in C'_i$ and $y \in C'_j$. By definition, we have $\text{Val}(x) \bmod m \neq \text{Val}(y) \bmod m$. We can extend any string that starts with a nonzero symbol into a string that represents a multiple of m, so pick z such that $xz \in D_m$. Since multiplication by a constant relatively prime to m as well as addition of any integer are injective functions on the set of integers modulo m, we get $yz \notin D_m$. Now concatenate both these strings with ℓ zeros to get $xz0^\ell \in D_k$ and $yz0^\ell \notin D_k$, thus showing that x and y are not realted.

It now follows by part (c) of Problem 3 that the minimum number of states needed by a finite state automaton that decides D_k is $\ell + m + 3$ where $k = 3^\ell \cdot m$ and 3 does not divide m.

Also note the following correspondence between our equivalence classes and the classes from part (d) of Problem 3: $[1] = C'_1$, $[2] = C'_0 = C_0$, and $[20] = C_1$.