HW2CSE105F24: Homework assignment 2 solution

CSE105F24

Due: October 15th at 5pm, via Gradescope

In this assignment,

You will practice designing multiple representations of regular languages and working with general constructions of automata to demonstrate the richness of the class of regular languages.

Resources: To review the topics for this assignment, see the class material from Week 2. We will post frequently asked questions and our answers to them in a pinned Piazza post.

Reading and extra practice problems: Sipser Section 1.1, 1.2, 1.3. Chapter 1 exercises 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 1.10, 1.11, 1.12, 1.14, 1.15, 1.16, 1.17, 1.19, 1.20, 1.21, 1.22. Chapter 1 problem 1.51.

Assigned questions

1. **Automata design** (12 points): As background to this question, recall that integers can be represented using base b expansions, for any convenient choice of base b. The precise definition is: for b an integer greater than 1 and n a positive integer, the **base b expansion of** n is defined to be

$$(a_{k-1} \cdots a_1 a_0)_b$$

where k is a positive integer, $a_0, a_1, \ldots, a_{k-1}$ are nonnegative integers less than b, $a_{k-1} \neq 0$, and

$$n = \sum_{i=0}^{k-1} a_i b^i$$

Notice: *The base b expansion of a positive integer n is a string over the alphabet $\{x \in \mathbb{Z} \mid 0 \leq x < b\}$ whose leftmost character is nonzero.*

An important property of base b expansions of integers is that, for each integer b greater than 1, each positive integer $n = (a_{k-1} \cdots a_1 a_0)_b$, and each nonnegative integer a less than b,

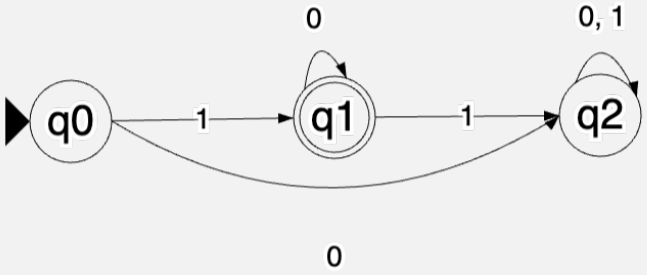
$$bn + a = (a_{k-1} \cdots a_1 a_0 a)_b$$

In other words, shifting the base b expansion to the left results in multiplying the integer value by the base. In this question we'll explore building deterministic finite automata that recognize languages that correspond to useful sets of integers.

(a) (*Graded for completeness*) [1] Design a DFA that recognizes the set of binary (base 2) expansions of positive integers that are powers of 2. A complete solution will include the state diagram of your DFA and a brief justification of your construction by explaining the role each state plays in the machine, as well as a brief justification about how the strings accepted and rejected by the machine connect to the specified language.

Hints: (1) A power of 2 is an integer x that can be written as 2^y for some nonnegative integer y, (2) the DFA should accept the strings 100, 10 and 100000 and should reject the strings 010, 1101, and ε (can you see why?).

Solution:



This DFA accepts strings over $\{0, 1\}$ that start with exactly a single 1 and end with 0s. Each state takes the following role:

- q_0 is the starting state, and its role is to not accept the empty string by not being one of the accept states.
- q_1 is the state that makes sure the string starts with exactly one 1 and none comes after. We enter q_1 if the first character is a 1. If all we see afterward are zeros, we accept. Otherwise, we exit this only accept state.
- q_2 is the trap state. If the string starts with a 0 or has more than two 1s the computation enters this non-accept state and doesn't leave until we finish reading the string, at which point we reject.

Now, we need to show why strings starting with exactly a single 1 and ending with 0s are exactly the set of binary strings representing powers of 2.

- ($\subseteq$) Take any string s with a single 1 at the front and 0's afterward, its value is determined by that first 1 since all the zero coefficients do not contribute to

[1] This means you will get full credit so long as your submission demonstrates honest effort to answer the question. You will not be penalized for incorrect answers. To demonstrate your honest effort in answering the question, we expect you to include your attempt to answer *each* part of the question. If you get stuck with your attempt, you can still demonstrate your effort by explaining where you got stuck and what you did to try to get unstuck.

the sum. Say $s = 1z_1z_2...z_n$ where $n \geq 0$, the value represented by s is $1 \cdot 2^n$, which is a power of 2.

- ($\supseteq$) Take any binary string s representing a power of 2. The first bit contributes 2^x for some $x \leq y$ to the sum. By induction, one can prove that $\sum_{i=0}^{n-1} 2^i = 2^n - 1$ (left as an exercise). To put it another way, even if all digits of s were 1, the value it represents is $\sum_{i=0}^{x} 2^i = 2^{x+1} - 1$, which is one less than 2^{x+1}, the next power of 2 greater than 2^x. This means that none of the bits after the first can be a 1 or else s will not represent a power of 2. This completes the proof.

(b) (*Graded for completeness*) Consider arbitrary positive integer m. Design a DFA that recognizes the set of binary (base 2) expansions of positive integers that are multiples of m. A complete solution will include the formal definition of your DFA (paramterized by m) and a brief justification of your construction by explaining the role each state plays in the machine, as well as a brief justification about how the strings accepted and rejected by the machine connect to the specified language.
Hints: (1) Consider having a state for each possible remainder upon division by m. (2) To determine transitions, notice that reading a new character will shift what we already read over by one slot.

Solution: A DFA that recognizes the set of binary (base 2) expansions of positive integers that are multiples of m will be denoted as $M = (Q, \{0,1\}, \delta, z, F)$ where we have the following:

- $Q = \{q_i \mid 0 \leq i < m\} \cup \{z, s\}$
- $\delta((q,x)) = \begin{cases} q_{((2i+x) \mod m)} & \text{for } q = q_i, 0 \leq i < m, x \in \{0,1\} \\ q_1 & \text{for } q = z, x = 1 \\ s & \text{for } q = z, x = 0 \\ s & \text{for } q = s, x \in \{0,1\} \end{cases}$
- $F = \{q_0\}$

The states z and s ensure that we only accept strings that are binary expansions (and therefore have a leading 1): z is the start state and is not in the set of accepting states so the empty string is rejected; s is the sink state the computation enters if the first bit is 0.

We can use the the hint given to us by having a running total of the remainder of the number we've read so far upon division by m. We can represent this running total by creating m states from q_0 to q_{m-1} such that each state q_i represents that i is the remainder of the current number read divided by m.

We can also notice that when we read a new character and "shift" what we've already read by one slot, this is equivalent to multiplying the current number by 2. We'll also have to add the character we're currently reading to add it to the running total. We can then mod this number by m to find the overall remainder of what we've read so far. Thus, we'll transition to the state that represents the remainder given the number we've read thus far. If we land in q_0, that means that the number we've read so far has a remainder of 0 when divided by m, which is the definition of being a multiple of m. If the string we've read doesn't land in q_0 that means the remainder when divided by m is not 0, which means the number is not a multiple of m and we should reject this string.

(c) (*Graded for correctness*) [2] Choose a positive integer m_0 between 5 and 8 (inclusive) and draw the state diagram of a DFA recognizing the following language over $\{0, 1, 2, 3\}$

$$\{w \in \{0, 1, 2, 3\}^* \mid w \text{ is a base 4 expansion of a positive integer that is a multiple of } m_0\}$$

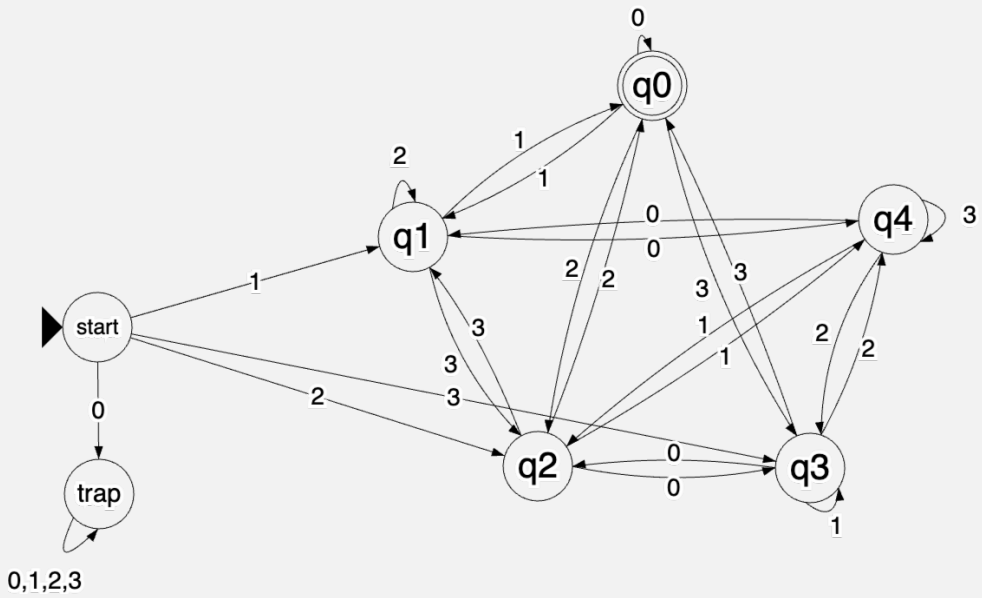
A complete solution will include the state diagram of your DFA and a brief justification of your construction by explaining the role each state plays in the machine, as well as a brief justification about how the strings accepted and rejected by the machine connect to the specified language.

Bonus extension to think about (ungraded): Which other languages related to sets of integers can be proved to be regular using a similar strategy?

Solution: You only need to have one of the following.

$m_0 = 5$:

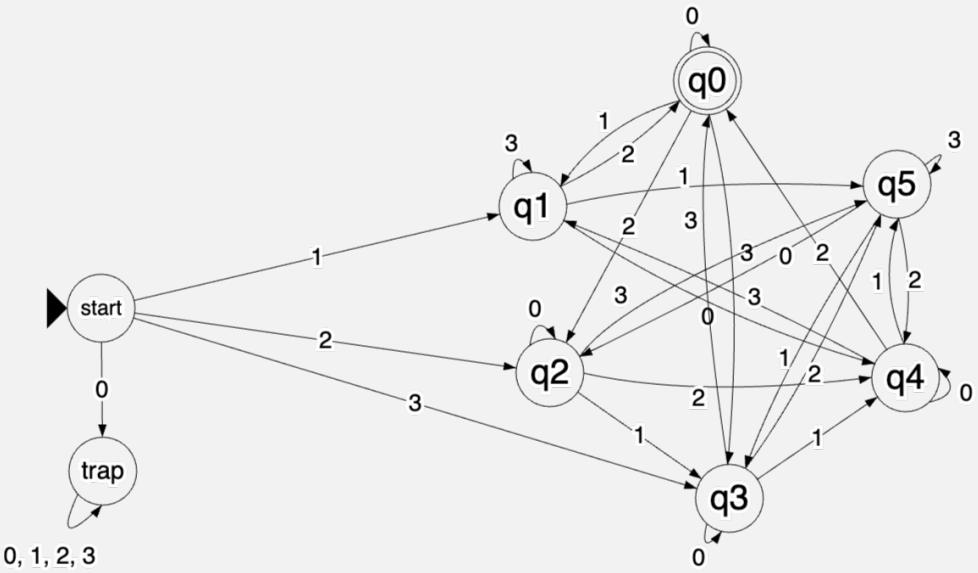
[2] This means your solution will be evaluated not only on the correctness of your answers, but on your ability to present your ideas clearly and logically. You should explain how you arrived at your conclusions, using mathematically sound reasoning. Whether you use formal proof techniques or write a more informal argument for why something is true, your answers should always be well-supported. Your goal should be to convince the reader that your results and methods are sound.



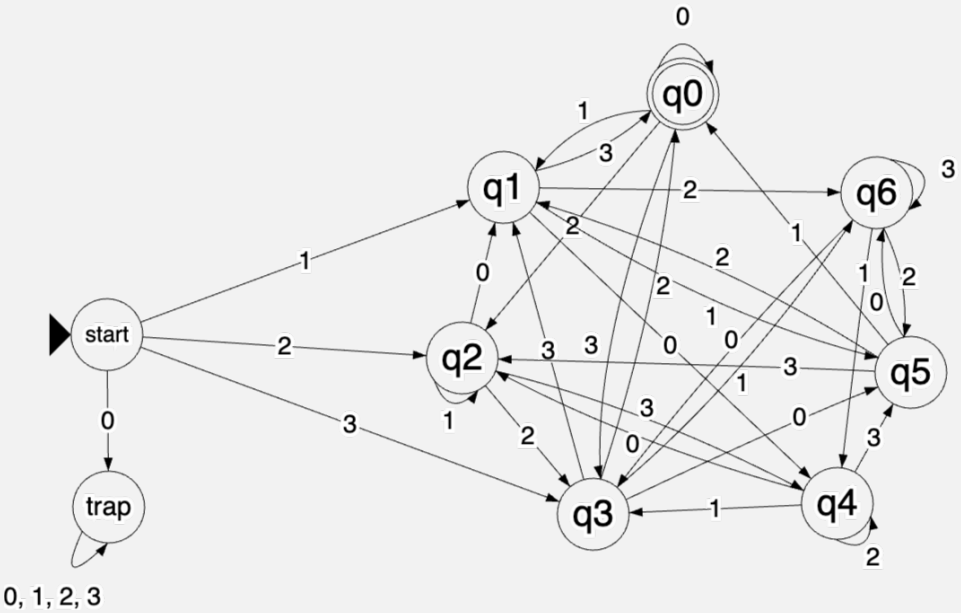
Justification:
The machine needs to accept strings that are base 4 expansions of a positive integer that is a multiple of m_0. The start state in combination with the trap state are included to reject any strings that start with a 0, as we require the first bit a_{k-1} to be nonzero. The role of the remaining states is to keep track of the current remainder when divided by m_0. More specifically, q_i represents that the remainder calculated so far is i, and q_0 is the accept state since when a computation ends in q_0, the remainder is 0, which means that the string is a base 4 expansion of a positive integer that is a multiple of m_0 and should be accepted. All other strings are rejected.

For example, in the above diagram, let $m_0 = 5$. Upon reading the first bit of the string 210, the current remainder will be updated to 2. When reading the next bit, we notice that the previous bit will be left shifted by 1, essentially multiplying the number by the base, 4. Thus, our current remainder will be updated to be $2 \cdot 4 + 1 \pmod 5 = 4$. Finally, after reading the final bit, we update the remainder to be $4 \cdot 4 + 0 \pmod 5 = 1$. Converting 210 in base 4 to 36 in decimal confirms that indeed, the remainder of 210 in base 4 divided by 5 is 1, thus we reject the string. Following a similar reasoning, the computation on string 203 ends in q_0. Converting 203 in base 4 to 35 in decimal confirms that indeed, the remainder of 203 in base 4 divided by 5 is 0, thus we accept the string.

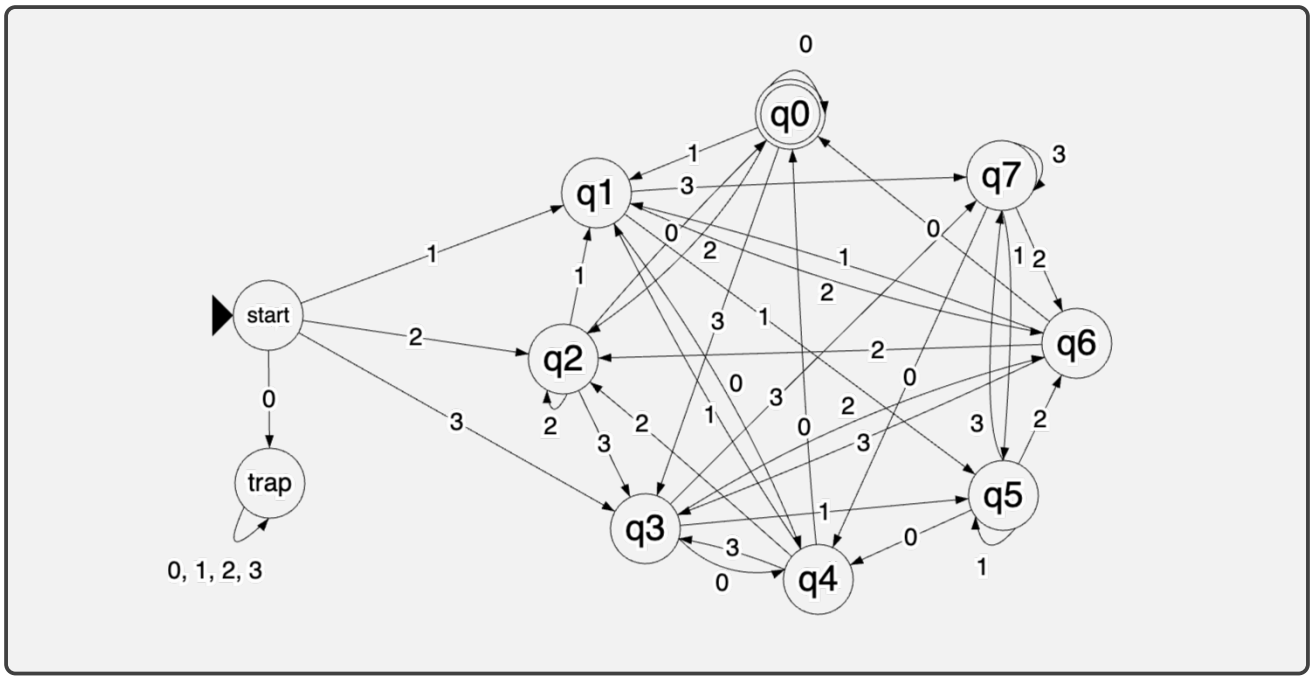
Below are the state diagrams for different values of m_0.
$m_0 = 6$:



$m_0 = 7$:



$m_0 = 8$:



2. **Nondeterminism** (15 points): For this question, the alphabet is $\{a, b\}$.

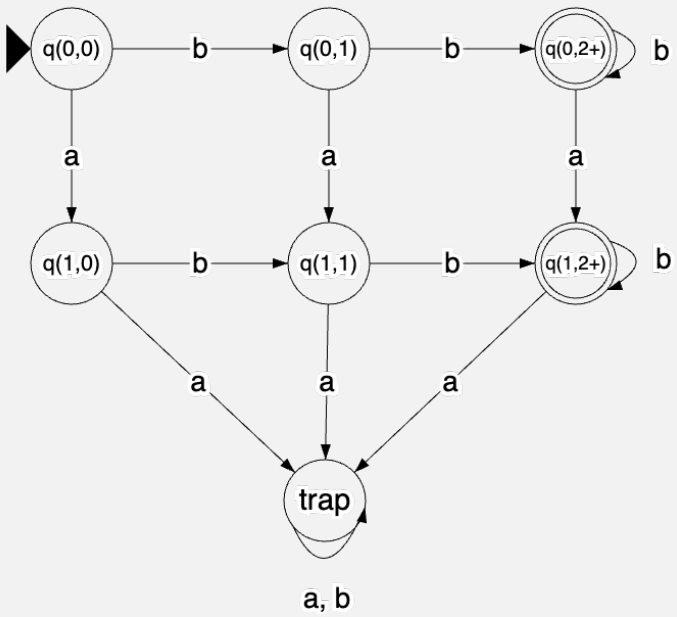
(a) (*Graded for completeness*) Design a DFA that recognizes the language

$$\{w \in \{a, b\}^* \mid w \text{ contains at most one } a \textbf{ and } \text{at least two } b\text{s}\}$$

You can design this DFA directly or use the constructions from class (and the footnote to Theorem 1.25 in the book) to build this DFA from DFA for the simpler languages that are intersected to give this language.

A complete solution will include the state diagram of your DFA and a brief justification of your construction either by explaining the role each state plays in the machine, as well as a brief justification about how the strings accepted and rejected by the machine connect to the specified language, or by justifying the design of the DFA for the simpler languages and then describing how the Theorem was used.

Solution:



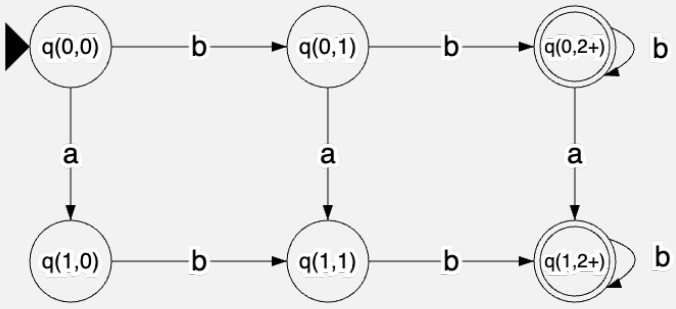
Justification:
The above DFA recognizes the given language. Each state remember the number of a's and b's we've seen so far: $q_{(0,0)}$ means zero a and zero b, $q_{(0,1)}$ means zero a and one b, $q_{(0,2+)}$ means zero a and two or more b's, $q_{(1,0)}$ means one a and zero b, $q_{(1,1)}$ means one a and one b, $q_{(1,2+)}$ means one a and two or more b's. The *trap* state means that we have seen more than one a. By making $q_{(0,2+)}$ and $q_{(1,2+)}$ accept states, we ensure that strings with zero or one a and two or more b's will be accepted, while the computation on other strings will not end in an accept state and thus other strings will be rejected. Therefore, the DFA recognizes $\{w \in \{a,b\}^* \mid w$ contains at most one a **and** at least two bs$\}$.

(b) (*Graded for correctness*) Design a NFA with at most 6 states that recognizes the language

$$\{w \in \{a,b\}^* \mid w \text{ contains at most one } a \textbf{ and } \text{at least two } b\text{s}\}$$

A complete solution will include the state diagram of your NFA and a brief justification of your construction by explaining the role each state plays in the machine, as well as a brief justification about how the strings accepted and rejected by the machine connect to the specified language. Give one example string in the language and explain the computation of the NFA that witnesses that the machine accepts this string. Also, give one example string not in the language and explain why the NFA rejects this string.

Solution:



Justification:
The above NFA recognizes the given language. Each state remember the number of a's and b's we've seen so far: $q_{(0,0)}$ means zero a and zero b, $q_{(0,1)}$ means zero a and one b, $q_{(0,2+)}$ means zero a and two or more b's, $q_{(1,0)}$ means one a and zero b, $q_{(1,1)}$ means one a and one b, $q_{(1,2+)}$ means one a and two or more b's. By making $q_{(0,2+)}$ and $q_{(1,2+)}$ accept states, we ensure that strings with zero or one a and two or more b's will be accepted. Computations of the NFA on strings with more than one a will get stuck, since there is no outgoing arrow labeled by a from $q_{(1,0)}$, $q_{(1,1)}$, or $q_{(1,2+)}$. Computations of the NFA on strings with less than two b's will end in one of the four non-accept states if the number of a is at most 1, or will get stuck if the number of a's is more than 1. Therefore, the NFA rejects the strings that have more than one a or less than two b's. Thus the NFA recognizes the given language $\{w \in \{a,b\}^* \mid w$ contains at most one a **and** at least two bs$\}$.

Notice that this is almost the same state diagram as in part (a): we get to save a state by not including the "trap" state since NFAs do not have to have outgoing edges from each state labelled with each input character.

One example string in the language would be bb. The computation of the NFA that witnesses that the machine accepts this string will be: start from $q_{(0,0)}$, read b and move to $q_{(0,1)}$, read b and move to $q_{(0,2+)}$, then the string is fully processed and accepted.

One example string not in the language would be aa. Start from q_0, we read the first a and move to $q_{(1,0)}$. However, we cannot process the second a, since there is no outgoing arrow labeled with a from $q_{(1,0)}$. Therefore, there is no computation of the NFA on the string aa that can process the whole string and ends in an accept state. Therefore, the NFA rejects string aa.

(c) (*Graded for correctness*) Design a NFA with at most 6 states that recognizes the language

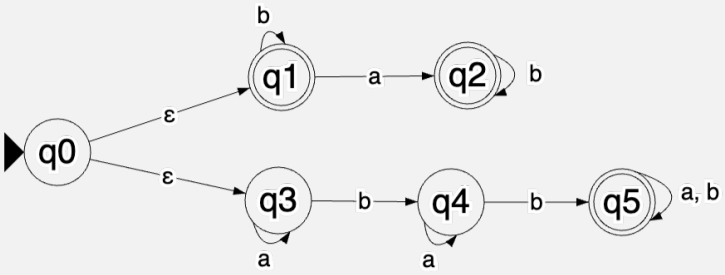
$$\{w \in \{a,b\}^* \mid w \text{ contains at most one } a \textbf{ or } \text{at least two } b\text{s}\}$$

A complete solution will include the state diagram of your NFA and a brief justification of

your construction by explaining the role each state plays in the machine, as well as a brief justification about how the strings accepted and rejected by the machine connect to the specified language. Give one example string in the language and explain the computation of the NFA that witnesses that the machine accepts this string. Also, give one example string not in the language and explain why the NFA rejects this string.

Bonus extension to think about (ungraded): Did you need all 6 states? Could you design DFA with 6 states that recognize each of these langauges?

Solution:



Justification:
From the start state q_0, the computation can spontaneously move to the upper part (consists of q_1 and q_2) or the lower part (consists of q_3, q_4 and q_5). The upper part simulates an NFA that recognizes the set of strings with at most one a, where q_1 means that we have not seen an a, and q_2 means that we have seen exactly one a. The lower part simulates an NFA that recognizes the set of strings with at least two b's, where q_3 means that we have not seen a b, q_4 means that we have seen exactly one b, and q_5 means that we have seen two or more b's. Therefore, only strings with at most one a or at least two b's will be accepted by this NFA.

One example string in the language would be ab. The computation of the NFA that witnesses that the machine accepts this string will be: start from q_0, spontaneously move to q_1, read a and move to q_2, read b and stay in q_2, then the string is fully processed and accepted.

One example string not in the language would be aa. Start from q_0, if we choose to spontaneously move to q_1, then we will read an a and transit to q_2, then when we try to process the second a, the computation is stuck. On the other hand, start from q_0, if we choose to spontaneously move to q_3, then we will read the two a's and stay in q_3 which is not an accept state. Therefore, there is no computation of the NFA on the string aa that can process the whole string and ends in an accept state. Therefore, the NFA rejects string aa.

3. **General constructions** (15 points): In this question, you'll practice working with formal general constructions for NFAs and translating between state diagrams and formal definitions.

(a) (*Graded for correctness*) Consider the following general construction: Let $N_1 = (Q, \Sigma, \delta_1, q_1, F_1)$ be a NFA and assume that $q_0 \notin Q$. Define the new NFA $N_2 = (Q \cup \{q_0\}, \Sigma, \delta_2, q_0, \{q_1\})$ where

$$\delta_2 : (Q \cup \{q_0\}) \times \Sigma_\varepsilon \to \mathcal{P}(Q \cup \{q_0\})$$

is defined by

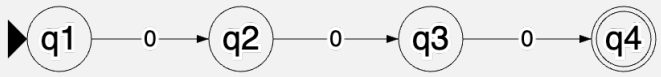
$$\delta_2(q, a) = \begin{cases} \{q' \in Q \mid q \in \delta_1(q', a)\} & \text{if } q \in Q,\ a \in \Sigma_\varepsilon \\ F_1 & \text{if } q = q_0,\ a = \varepsilon \\ \emptyset & \text{if } q = q_0,\ a \in \Sigma \end{cases}$$

Illustrate this construction by defining a specific example NFA N_1 and applying the construction above to create the new NFA N_2. Your example NFA should

- Have exactly four states (all reachable from the start state),
- Accept at least one string and reject at least one string, and
- Not have any states labelled q_0.

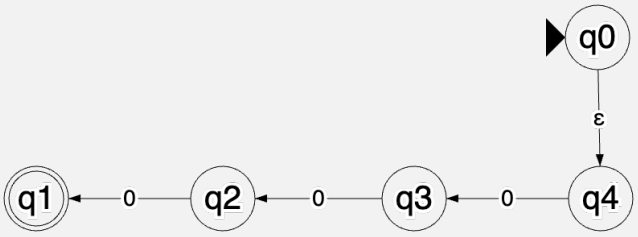
Apply the construction above to create the new NFA. A complete submission will include the state diagram of your example NFA N_1 and the state diagram of the NFA N_2 resulting from this construction and a precise and clear description of $L(N_1)$ and $L(N_2)$, justified by explaining the role each state plays in the machine, as well as a brief justification about how the strings accepted and rejected by the machine connect to the language.

Solution: We choose the alphabet $\{0\}$. Let the original NFA N_1 be as follows:



We can see that $L(N_1) = \{000\}$. Each state has a role: q_1 means we have not seen a 0, q_2 means we have seen exactly one 0, q_3 means we have seen exactly two 0's, and q_4 means we have seen exactly three 0's. Since q_4 is the accept state, only strings with exactly three 0's will be accepted, i.e. 000. Computation on a string with less than three 0's will not end in q_4 (for example, computation on string 00 will end in q_3), and computation on a string with more than three 0's will stuck after reaching q_4, therefore they will not be accepted.

Now we apply the construction, creating q_0 as the new start state and connecting it to all original final states, which is just q_4. Additionally, we reverse all the edges and make q_1 the accept state. This gives N_2:



We can see that $L(N_2) = \{000\}$. Each state has a role: q_0 and q_4 means we have not seen a 0, q_3 means we have seen exactly one 0, q_2 means we have seen exactly two 0's, and q_1 means we have seen exactly three 0's. Since q_1 is the accept state, only strings with exactly three 0's will be accepted, i.e. 000. Computation on a string with less than three 0's will not end in q_1 (for example, computation on string 00 will end in q_2), and computation on a string with more than three 0's will stuck after reaching q_1, therefore they will not be accepted.

(b) In Week 2's review quiz, we saw the definition that a set X is said to be **closed under an operation** if, for any elements in X, applying to them gives an element in X. For example, the set of integers is closed under multiplication because if we take any two integers, their product is also an integer .

Recall the definitions we have: For each language L over the alphabet $\Sigma_1 = \{0, 1\}$, we have the associated set of strings

$$EXTEND(L) = \{w \in \Sigma_1^* \mid w = uv \text{ for some strings } u \in L \text{ and } v \in \Sigma_1^*\}$$

We will prove that the collection of languages over $\{0, 1\}$ that are each recognizable by some NFA is closed under the $EXTEND$ operation.

i. (*Graded for completeness*) As a helpful tool in our construction[3], prove that every NFA can be converted to an equivalent one that has a single accept state. Note: this is exercise 1.11 in the textbook.

Solution: Given some NFA $N = (Q, \Sigma, \delta, q_0, F)$, consider the following construction of an NFA N' such that $L(N) = L(N')$ and N' has only one accept state:

$N' = (Q \cup \{q_{acc}\}, \Sigma, \delta', q_0, \{q_{acc}\})$, where $q_{acc} \notin Q$, and the transition function $\delta' : Q' \times \Sigma_\varepsilon \to \mathcal{P}(Q')$ is defined by

$$\delta'((q,a)) = \begin{cases} \delta((q,a)) & \text{if } q \in Q,\ q \notin F \text{ and } a \in \Sigma_\varepsilon \\ \delta((q,a)) & \text{if } q \in F \text{ and } a \in \Sigma \\ \delta((q,a)) \cup \{q_{acc}\} & \text{if } q \in F \text{ and } a = \varepsilon \\ \emptyset & \text{if } q = q_{acc} \text{ and } a \in \Sigma_\varepsilon \end{cases}$$

[3]A result that is proved in order to work towards a larger theorem is called a Lemma.

The big takeaway from this construction is to add a new accept state q_{acc} to the original NFA N, make all old accept states to be non-accepting, and add ε transitions from all old accept states to q_{acc}.

Proof of correctness: Consider some string w accepted by N. Then there exists an N's computation on w that processes the whole string and ends in some state $p \in F$, i.e an accept state of N. Now consider N''s computation on w. Based on the definition of N''s transition function, there will also exist an N''s computation on w that processes the whole string and arrives at the state $p' = p \in F$. Here, we specified an ε transition to q_{acc} in δ' when a state in N' was an accept state in N. So, p' will have an ε transition to q_{acc}. Thus, once N''s computation processed the whole string w and lands in p', we can take the ε transition to q_{acc} and accept w, so $w \in L(N')$, and $L(N) \subseteq L(N')$.

Conversely, consider some string w accepted by N'. Then by the transition function, when N''s computation on w ends, there is a state p' that the machine was at before taking an ε transition to q_{acc}. This p' state is formerly an accept state of N. Because N''s transition function is dependent on N's transition function, when N runs on w, there would also exist a computation that ends in p', which is an accept state of N, so N accepts w. $w \in L(N)$, and $L(N') \subseteq L(N)$. Thus we proved that $L(N) = L(N')$.

ii. (*Graded for correctness*) Prove that the collection of languages over $\{0,1\}$ that are each recognizable by some NFA is closed under the $EXTEND$ operation. You can assume that you are given a NFA with a single accept state $N = (Q, \{0,1\}, \delta, q_0, \{q_{acc}\})$ and you need to define a new NFA, $N_{new} = (Q_{new}, \{0,1\}, \delta_{new}, q_{new}, F_{new})$, so that $L(N_{new}) = EXTEND(L(N))$.
A complete solution will include precise definitions for $Q_{new}, \delta_{new}, q_{new}$, and F_{new}, as well as a a brief justification of your construction by explaining why these definitions work, referring specifically to the definition of $EXTEND$ and to acceptance of NFA.

Solution: Assume that $q_{extend} \notin Q$ (or relabel if it is). Consider the construction $N_{new} = (Q_{new}, \{0,1\}, \delta_{new}, q_{new}, F_{new})$, where:
$Q_{new} = Q \cup \{q_{extend}\}$
$q_{new} = q_0$
$F_{new} = \{q_{extend}\}$

$\delta_{new} : Q_{new} \times \Sigma_\varepsilon \to \mathcal{P}(Q_{new})$ is defined by

$$\delta_{new}((q,a)) = \begin{cases} \delta((q,a)) & \text{if } q \in Q,\ q \neq q_{acc} \text{ and } a \in \Sigma_\varepsilon \\ \delta((q,a)) & \text{if } q = q_{acc} \text{ and } a \in \Sigma \\ \delta((q,a)) \cup \{q_{extend}\} & \text{if } q = q_{acc} \text{ and } a = \varepsilon \\ \{q_{extend}\} & \text{if } q = q_{extend}, a \in \Sigma_\varepsilon \end{cases}$$

Similar to the previous part, we add a new accept state q_{extend} to the NFA N, connect the previous accept state q_{acc} to q_{extend} via spontaneous transition in N_{new}, and make q_{acc} a non-accept state in N_{new}. Additionally, q_{extend} also has a self loop for every symbol in Σ and a spontaneous self loop (even though that is superfluous). The q_{extend} state represents the string $v \in \Sigma_1^*$ in the definition of $EXTEND$.

Proof of correctness: Assume the string w is accepted by N_{new}. Based on the construction of N_{new}, there exists a computation of N_{new} on w that reaches q_{acc} and then ends in q_{extend}. This means that w can be decomposed into $w = uv$ such that u is accepted by N, and $v \in \Sigma_1^*$. Therefore, $w \in EXTEND(L(N))$, and $L(N_{new}) \subseteq EXTEND(L(N))$.

Conversely, assume the string $w = uv$ is in $EXTEND(L(N))$ where $u \in L(N)$ and $v \in \Sigma_1^*$. Again, there exists an N's computation on u that ends in q_{acc}, so there also exists an N_{new}'s computation on u that arrives at q_{acc}. Per the construction, q_{acc} will have a spontaneous transition to q_{extend} in N_{new}. The transition will be taken and q_{extend} will run the rest of the computation on v. Since q_{extend} has self-loops for all symbols of Σ and is an accept state, q_{extend} will "process" and accept v no matter what it is (i.e $v \in \Sigma_1^*$), and as a result, w is accepted by N_{new}. Therefore $w \in L(N_{new})$, so $EXTEND(L(N)) \subseteq L(N_{new})$.

4. **Multiple representations** (8 points): For any language $L \subseteq \Sigma^*$, recall that we define its *complement* as

$$\overline{L} := \Sigma^* - L = \{w \in \Sigma^* \mid w \notin L\}$$

That is, the complement of L contains all and only those strings which are not in L. Our notation for regular expressions does not include the complement symbol. However, it turns out that the complement of a language described by a regular expression is guaranteed to also be describable by a (different) regular expression.[4]

For example, over the alphabet $\Sigma = \{a, b\}$, the complement of the language described by the regular expression $\Sigma^* b$ is described by the regular expression $\varepsilon \cup \Sigma^* a$ because any string that does not end in b must either be the empty string or end in a.

[4]We'll see that this is connected to the result we proved in class that the complement of each language recognizable by a DFA is recognizable by a(nother) DFA.

For each of the regular expressions R over the alphabet $\Sigma = \{a, b\}$ below, write the regular expression for $\overline{L(R)}$. Your regular expressions may use the symbols $\varnothing$, ε, a, b, and the following operations to combine them: union, concatenation, and Kleene star.

Briefly justify why your solution for each part works by giving plain English descriptions of the language described by the regular expression and of its complement and connecting them to the regular expression via relevant definitions. An English description that is more detailed than simply negating the description in the original language will likely be helpful in the justification.

Alternatively, you can justify your solution by first designing a DFA that recognizes $L(R)$, using the construction from class and the book to modify this DFA to get a new DFA that recognizes $\overline{L(R)}$, and then applying the constructions from class and the book to convert this new DFA to a regular expression.

For each part of the question, clearly state which approach you're taking and include enough intermediate steps to illustrate your work.

(a) (*Graded for correctness*) $(a \cup b)^* a (a \cup b)^*$

Solution: b^*
First, we translate this regular expression to plain English. Looking at the structure of the regex, we notice that it describes the set of strings with at least one a symbol. The complement of this language would be the set of strings that do not contain any a, which means the set of all strings that contain only b's. An equivalent regular expression that describes the complement would be b^*.

(b) (*Graded for correctness*) $(a \cup b)(a \cup b)(a \cup b)$

Solution: $\varepsilon \cup \Sigma \cup \Sigma\Sigma \cup \Sigma\Sigma\Sigma\Sigma\Sigma^*$
The given regular expression describes the set of strings of length 3. The complement of this language would be the set of all strings over the alphabet whose lengths are not 3. An equivalent regular expression that describes the complement would therefore be $\varepsilon \cup \Sigma \cup \Sigma\Sigma \cup \Sigma\Sigma\Sigma\Sigma\Sigma^*$, which describes the union of the sets of strings with length 0, 1, 2, and 4 or more.