Theory of Computation: Midterm Exam Solutions

Arjun Chandrasekhar

1. (a) $L = \{a^{2n}b^{2m} | n, m \geq 0\}$ This language is regular. It can be described by the regular expression $R = (aa)^*(bb)^*$

   (b) (5 points) $L = \{a^{2n}b^{2n} | n \geq 0\}$ This language is not regular. AFSOC L is regular with pumping length p. Consider the string $w = a^{2p}b^{2p}$. This string is in the language and its length is at least p, so it should be pumpable. Let w be split up into three parts xyz such that $|xy| \leq p$ and $|y| > 0$. This means that x and y will only contain a's, and y will be non-empty. However, this means that if we pump y, the a's and b's will not be equal. Thus the new pumped string will not be in the language, which is a contradiction of the pumping lemma. We conclude that L is not regular.

   Note: It might be tempting to use a^pb^p as our string. However, if p is odd then this string is not in the language. Hence we use $a^{2p}b^{2p}$ because this is guaranteed to have an even number of each symbol.

   (c) $L = \{a^nb^nc^n | n \geq 3\}$ This language is not regular. AFSOC L is regular with pumping length p. Consider the string $a^{p+3}b^{p+3}c^{p+3}$. This string is in the language and its length is at least p, thus it should be pumpable. Let w be split into three parts xyz such that $|xy| \leq p$ and $|y| > 0$. This means that x and y will only contain a's and y will be non-empty. However, this means that if we pump y, the a's, b's, and c's will not be equal. Thus the new pumped string will not be in the language, which is a contradiction of the pumping lemma. We conclude that L is not regular.

   Note: It might be tempting to use $a^pb^pc^p$ as our string. However, if $p \leq 3$ then this string is not in the language. Hence, we use $a^{p+3}b^{p+3}c^{p+3}$ because this is guaranteed to have at least 3 copies of each symbol.

   (d) $L = \{a^nb^nc^n | n \leq 3\}$ This language is regular. It can be described by the regular expression $R = \epsilon \cup abc \cup aabbcc \cup aaabbbccc$. Alternately, we can write it as

   $$R = \bigcup_{i=0}^{3} a^ib^ic^i$$

   Note that this may look like the previous problem; however because n is constrained to be at most 3, the language is finite, which makes it regular. You may also worry that this language is not regular because of the pumping lemma. However we can set the pumping length to be $p = 10$. Every string in the language has length less than 10, so none of them are required to be pumpable.

2. We simply note that BOTHWAYS(L) $= L \cap L^R$. Because regular languages are closed under reversal and intersection, if L is regular then BOTHWAYS(L) must be regular.

3. (a) Suppose L is regular. We will construct a spooky expression S that also describes L.

Approach 1: Start with a regex R that describes L, and convert it into an equivalent spooky expression S inductively.

Base case:

- **case 1:** $R = a \in \Sigma$. Then $S = a$ is a spooky expression that describes L
- **case 2:** $R = \epsilon$. Then $S = \epsilon$ is a spooky expression that describes L
- **case 3:** $R = \emptyset$. Then $S = \emptyset$ is a spooky expression that describes L

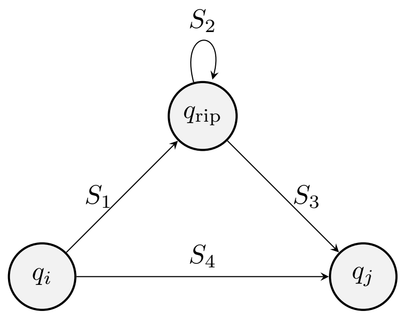
Inductive case: assume that every regular expression R_i of size $\leq n$ has an equivalent spooky expression S_i. Let R be a regular expression of size $n+1$ that describes L.

- **case 1:** $R = R_1 \cup R_2$. Then R_1 and R_2 both have size $\leq n$. Thus there are equivalent spooky expressions S_1 and S_2. Then the spooky expression $S = (S_1^c \cap S_2^c)^c$ is equivalent to R
- **case 2:** $R = R_1 \circ R_2$. Then R_1 and R_2 both have size $\leq n$. Thus there are equivalent spooky expressions S_1 and S_2. Then the spooky expression $S = S_1 \circ S_2$ is equivalent to R
- **case 3:** $R = (R_1)^*$. Then R_1 has size $\leq n$. Thus there is an equivalent spooky expressions S_1. Then the spooky expression $S = (S_1)^*$ is equivalent to R.

Approach 2: Start with a DFA D that describes L, and convert it into an equivalent spooky expression S. First, convert D into a GNFA G. Then, we will iteratively "rip" away states until there are just two states and one transition remaining; the one remaining transition will be labelled with a spooky expression S that describes L.

When we initially convert the DFA to a GNFA, first we'll add a special start state with an ϵ transition to the original start state and $\emptyset$ transitions to all other states. We will also add a special accept state; the original accept states will have ϵ transitions to this special accept state, and all other states will have $\emptyset$ transitions to the special accept state. We will then make sure that there is a transition between every pair of original states (including a loop from each original state back to itself), and we need to make each of these transitions into a spooky expression. If there is a transition between q_i and q_j that is labelled by only one character σ, we will label the transition in the GNFA with the spooky expression $S = \sigma$. If the transition between q_i and q_j is labelled by multiple characters $\sigma_1, \sigma_2, \ldots, \sigma_n$, we will label the transition in the GNFA with the spooky expression $S = (\sigma_1^c \cap \sigma_2^c \cap \cdots \cap \sigma_n^c)^c$ (which is equivalent to $\sigma_1 \cup \sigma_2 \cup \cdots \cup \sigma_n$). If the transition between q_i and q_j is missing in the original DFA, we will add an $\emptyset$ transition in the GNFA.

Now we need to explain how to rip away states. Recall that for regular expressions, when we "rip" away a state q_{rip}, we need to repair the transitions between each pair of remaining states q_i and q_j. Before we rip away q_{rip}, the neighborhood around q_i and q_j looks like this:



Here, S_1 is the spooky expression that takes us from q_i to q_{rip}, S_2 is the spooky expression that takes us from q_{rip} back to itself, S_3 is the spooky expression that takes us from q_{rip} to q_j, and S_4 is the spooky expression that takes us from q_i to q_j directly.

When we "rip" the state q_{rip}, we need to repair the transition from q_i to q_j so that the same set of strings take us from q_i to q_j. With regular expressions, we would repair it as follows:

$$\begin{tikzcd} q_i \arrow[r, "(S_1 \circ S_2^* \circ S_3) \cup S_4"] & q_j \end{tikzcd}$$

However, $\cup$ is not a spooky expression operation. Luckily, we can use complement and intersection to achieve the same effect as union. In particular, $A \cup B = (A^c \cap B^c)^c$ So when we repair the transition from q_i to q_j, we will do so as follows:

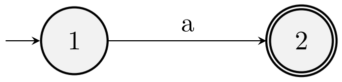
$$\begin{tikzcd} q_i \arrow[r, "((S_1 \circ S_2^* \circ S_3)^c \cap S_4^c)"] & q_j \end{tikzcd}$$

Thus, we will keep "ripping" states and repairing each remaining transition until there is just one transition remaining. The spooky expression labelling this transition describes the language of the original DFA. Thus, every regular language can be described by a spooky expression.

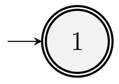
(b) Suppose L is described by a spooky expression S. We will show how to inductively (or if you prefer, recursively) construct an NFA to recognize L.

Base case:

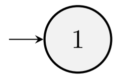
- **case 1:** $S = a \in \Sigma$. Then the following NFA will recognize L



- **case 2:** $S = \epsilon$. Then the following NFA will recognize L



- **case 3:** $S = \emptyset$. Then the following NFA will recognize L



Inductive case: Assume that for every spooky expression S_i of size n, we can construct an NFA N_i (and by extension, a DFA D_i) that recognizes $L(S_i)$. Let L be described by a spooky expression of size $n + 1$

- **case 1:** $S = S_1 \cap S_2$. Inductively, we can construct DFAs D_1 and D_2 that recognize $L(S_1)$ and $L(S_2)$, respectively. Then we can use the Cartesian product technique to construct a DFA D to recognize $L(D_1) \cap L(D_2) = L(S_1) \cap L(S_2) = L(S)$
- **case 2:** $S = S_1^c$. Inductively, we can construct a DFA D_1 that recognizes $L(S_1)$. Then we can flip the accept and reject states to construct a DFA D that recognizes $L(D)^c = L(S_1)^c = L(S)$
- **case 3:** $S = S_1 \circ S_2$. Inductively, we can construct DFAs D_1 and D_2 that recognize $L(S_1)$ and $L(S_2)$, respectively. We can then construct an NFA N to recognize $L(D_1) \circ L(D_2)$. We will add ϵ transitions from the accept states of D_1 to the start state of D_2. The start state of N is the start state of D_1 and the accept states of N are the accept states of D_2. This NFA recognizes $L(D_1) \circ L(D_2) = L(S_1) \circ L(S_2) = L(S)$
- **case 4:** $S = (S_1)^*$. Inductively, we can construct a DFA D_1 that recognizes $L(S_1)$. We can then construct an NFA N that recognizes $L(D_1)^*$. We will add a special start state s_0 that is also an accept state. This start state will have an ϵ transition to the start state of D_1. Every accept state of D_1 will have an ϵ transition to the start state of D_1. This NFA recognizes $L(D_1)^* = L(S_1)^* = L(S)$

4. **Approach 1:** Construct a state machine. Let $D_A = (Q_A, \Sigma, \delta_A, S_A, F_A)$ be a DFA that recognizes A, and let $D_B = (Q_B, \Sigma, \delta_B, S_B, F_B)$ be a DFA that recognizes B. We will construct a DFA $D = (Q, \Sigma, \delta, S, F)$ to recognize XOR-SHUFFLE(A, B) as follows:

   - $Q = Q_A \times Q_B \times \{A, B\}$. Every state is a combination of three variables: a state from machine A, a state from machine B, and a counter that keeps track of whose turn it is to read a symbol.
   - The alphabet $Sigma$ stays the same
   - The transition function is defined as follows:
     - $\delta((q_A, q_B, A), \sigma) = (\delta_A(q_A, \sigma), q_B, B)$. If it's A's turn to read, we transition A's state while keeping B the same. To transition A's state, we apply A's transition function. After reading the character, it will be B's turn to read,
     - $\delta((q_A, q_B, B), \sigma) = (q_A, \delta_B(q_B, \sigma), A)$. If it's B's turn to read, we transition B's state while keeping A the same. To transition B's state, we apply B's transition function. After reading the character, it will be A's turn to read.
   - $S = (S_A, S_B, A)$. Initially, A starts in it's start state, B starts in its start state, and it's A's turn to read a character
   - $F = \{(q_A, q_B, A) | q_A \in F_A \text{ or } q_B \in F_B \ \underline{\text{but not both}}\}$. We accept if exactly one of the two machines finished in an accept state. Additionally, it should be A's turn to read a character at the end.

   Thus, because there is a DFA to recognize XOR-SHUFFLE(A, B), we conclude that XOR-SHUFFLE(A, B) is regular, which establishes that regular languages are closed under XOR-SHUFFLE.

   Approach 2: Appeal to known regular language closure properties. We note that

   $$\text{XOR-SHUFFLE}(A, B) = \text{PERFECT-SHUFFLE}(A, B^c) \cup \text{PERFECT-SHUFFLE}(A^c, B)$$

   Regular languages are closed under complement and PERFECT-SHUFFLE. Thus, regular languages are closed under XOR-SHUFFLE.

5. (a) First we prove the forward direction: if L is regular, then L can be recognized by an rNFA. If L is regular, then there exists a DFA D that recognizes L. We simply note that D *is* a rNFA that simply chooses not to have non-determinism. A DFA only has one computation path, so it only accepts if exactly one computation path accepts.

   Next we prove the backward direction: if L can be recognized by an rNFA then L is regular. Let $R = (Q_R, \Sigma, \delta_R, \S_R, F_R)$ be an rNFA that recognizes L. We will design a DFA $D = (Q, \Sigma, \delta, S, F)$ as follows:

   - $Q = \mathcal{P}(Q_R)$ - Every DFA state is a unique combination of rNFA states. The DFA keeps track of all of the possible states the rNFA *could* currently be in based on the characters it has read and choices that were available at each step.
   - The alphabet Σ stays the same
   - $\delta(T, \sigma) = E\left(\bigcup_{t \in T} \delta_R(t, \sigma)\right)$. The input to the transition function is a combination of states $T \subseteq Q_R$, as well as a symbol $\sigma \in \Sigma$. We loop through every individual state $t \in T$, and apply the rNFA transition function to t and σ. We collect the union of all of these results into one list of possible "next states". Finally, we take the ϵ closure of this set to see where the rNFA could go after reading the symbol σ as well as any number of ϵ characters.
   - $S = E(\{S_R\})$. The start state of the DFA is the ϵ closure of the start state of the rNFA. This reflects all of the states that the rNFA could possibly reach before it reads any characters.

- $F = \{T \subseteq Q_R | T$ contains $\underline{\text{exactly one}}$ accept state of $R\}$. Every combination that contains one and only one accept state is an accepting combination. This indicates that after reading the string, the rNFA can reach one and only one accept state.

By construction, D is a DFA that simulates every computation path of R, and accepts a string if and only R would have accept it. Thus, D recognizes L, so L is regular.

(b) We can use rNFA's to show that regular languages are closed under $\oplus$. Suppose A and B are regular. Then there are DFAs D_A, D_B to recognize A and B, respectively. To recognize $A \oplus B$, we construct an rNFA R that has a special start state S_0, along with ϵ transitions to the original start states of D_A and D_B. This creates exactly two computation paths: one path checks if D_A accepts w, and the other path checks if D_B accepts w. The rNFA accepts if exactly one of these two paths accepts. This means R accepts w if either $w \in A$ or $w \in B$, but not both. By definition, this means R accepts w if and only if $w \in A \oplus B$.