

**Solutions for Problem Set 1**  
**cs172**

Allison Coates

- 1.10 a. Show that if  $M$  is a DFA that recognizes language  $B$  swapping the accept and non-accept states in  $M$  yields a new DFA that recognizes the complement of  $B$ .

Consider a DFA  $M = \langle Q, \Sigma, \delta, q_0, F \rangle$ . There is exactly one transition path from a state  $q \in Q$  for each letter in  $\Sigma$ . If  $M$  accepts  $w$ , then there is exactly one path through  $M$  for  $w$ , ending in a state  $f \in F$ . If  $M$  does not accept  $w$ , then there is exactly one path through  $M$  for  $w$ , ending in a state  $f \in Q - F$ . A machine  $M' = \langle Q, \Sigma, \delta, q_0, Q - F \rangle$  accepts  $w$  if  $M$  does not, and does not accept  $w$  if  $M$  does.

- b. Show by giving an example that, if  $M$  is an NFA that recognizes language  $C$ , swapping the accept- and non-accept states in  $M$  doesn't necessarily yield a new NFA that recognizes the complement of  $C$ . Is the class of languages recognized by NFAs closed under complement? Explain your answer.

Example: Consider an NFA  $N = \langle Q, \Sigma, \delta, q_0, F \rangle$ , where:

$$\begin{aligned} Q &= q_0, q_1, q_2 \\ \Sigma &= \{0, 1\} \\ \delta &= \{\delta(q_0, 0) = \delta(q_0, 1) = q_1, \delta(q_1, 0) = \delta(q_1, 1) = q_2\} \\ &\quad q_0 \\ F &= \{q_1\} \end{aligned}$$

This NFA accepts  $\Sigma - \epsilon$ . If we simply swap accept and non-accept states, we build a machine  $N'$ , which accepts the language  $\Sigma$ . This is clearly not the complement of  $L(N)$ .

Yes, NFAs are closed under complement, but that complement isn't the simple "complement" of swapping accept and non-accept states. An NFA  $N$  accepts if any path accepts; a machine  $N'$  accepts the complement language if no path accepts those strings accepted by  $N$ . We can determinize an NFA and determine all possible paths that accept; if we then swap accept/non-accept on all possible paths, we have build a machine that accepts the complement.

- 1.15 For each of the following languages, gives two strings that are members and two strings that are *not* members— a total of four strings for each part.

Assume the alphabet  $\Sigma = \{a, b\}$  in all parts. I include three just to show more examples. you were required to show two.

- a.  $a^*b^*$   
Members:  $\epsilon$ , ab, aaa  
Not Members: ba, bba, bab
- b.  $a(ba)^*b$   
Members: ab, abab, ababab  
Not Members: a, b,  $\epsilon$
- c.  $a^* \cup b^*$   
Members: a, b,  $\epsilon$   
Not Members: ab, ba, aabb
- d.  $(aaa)^*$   
Members: aaa, aaaaaa,  $\epsilon$   
Not Members: b, a, aa

- e.  $\Sigma^*a\Sigma^*b\Sigma^*a\Sigma^*$   
 Members: aba, abbaba, babbbab  
 Not Members: aaaa, bbbb, *rmb\**
- f.  $aba \cup bab$   
 Members: aba, bab (That's it).  
 Not Members:  $a^*$ ,  $b^*$
- g.  $(\epsilon \cup a)b$   
 Members: b, ab  
 Not Members:  $a^*$ ,  $bbb^*$
- h.  $(a \cup ba \cup bb)\Sigma^*$   
 Members:  $a^*$ ,  $bbb^*$   
 Not Members: b,  $\epsilon$

1.25 Let

$$\Sigma_3 = \left\{ \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \dots, \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \right\}.$$

$\Sigma_3$  contains all size 3 columns of 0s and 1s. A string of symbols in  $\Sigma_3$  gives three rows of 0s and 1s. Consider each row to be a binary number and let

$$B = \{w \in \Sigma_3^* \mid \text{the bottom row of } w \text{ is the sum of the top two rows}\}.$$

For example,

$$\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} \in B, \text{ but } \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} \notin B.$$

Show that  $B$  is regular.

First consider the possible ways that two binary digits can be summed:

$$\begin{aligned} 0 + 0 &= 0 \\ 0 + 1 &= 1 + 0 = 1 \\ 1 + 1 &= 0 \text{ and a carry bit} \end{aligned}$$

If there is already a carry bit in play, then the numbers we can get are:

$$\begin{aligned} 1 + 0 + 0 &= 1 \\ 1 + 0 + 1 &= 0 \text{ and a carry bit} \\ 1 + 1 + 1 &= 1 \text{ and a carry bit} \end{aligned}$$

Intuitively, we can use an FSM to calculate this sum because we don't need to keep track of all the numbers we have seen already: we only need to know the next two digits in the binary numbers, and whether or not there is a carry bit. This suggests we can solve the problem using only finite states. In fact, all we need to know is whether or not there is a carry bit, because with this information, we can validate if a triple represents an appropriate sum.

To make this easy, we reverse the strings, so that we can compute with them by adding in binary. Here is an example: consider the following two strings as binary digits, padded to be of the same length:

```
10101010
00101011
11010101
```

Looking at the first two numbers, they sum to 1. If we match on the 1, (we do), then we update our state to reflect whether or not we have a carry bit (we don't.) Looking at the next two, they sum to 0 + carry. We do update our state. Continuing, the 0 matches, so we need to keep track of the carry bit. the next two sum to zero; that and the carry yields a 1. that matches, so we continue, etc.

Here's the picture:

