CS 181 - Homework 9

Kuruvilla Cherian, Jahan
UID: 104436427
Section 1B

November 30, 2017

Problem 3.8

b

When working with a Turing Machine, we work with the idea of an infinite tape with the ability to move the head back and forth as we please. Therefore, our general algorithm should essentially do the following on getting some input string w:

1. Scan the input and find the first unmarked 0, and mark it as read (using some delimiter such as a symbol X). If we don't find such a 0, then just skip to the final step.
2. Continue and scan the very next unmarked 0. This is essentially looking for the second 0 in our input. If one cannot be found, then immediately move to rejection. If one is found, then move the head back to the beginning of the tape.
3. Scan the tape forward looking for the first unmarked 1, and mark it as read. If one cannot be found then move to rejection.
4. Move the head to the front of the tape and repeat step 1.
5. Move the head to the front and scan the entire tape for unmarked 1's. If none are left, then we *accept*, else move to rejection.

Problem 3.10

If we want to be able to simulate an infinite number of writes, teh key intuition is to continually copy over the current cell with its modified value (a single write) to its corresponding spot to the right of the entire original input. What this means is that if the tape contains the input, the first copy of the input will be to the right of the original input, and so on and so forth. So essentially to work with this we are performing a write twice. The first write is to copy over the cell and then to modify its contents. Such a solution can be accomplished using a bi-tape Turing Machine. As discussed in class, such a Turing Machine can be represented by the original Turing Machine by simply using **two cells** to represent the write (modification) and the copy (like a boolean flag). However, because the input is not presented to the Turing Machine as having two cells per symbol, what we must do upon the very first copy, is to place the copy mark directly over the input symbol itself. Thus we show that a write-once Turing machine can be represented by the original Turing machine by utilizing cell copying, delineated by two cells per symbol in the original Turing Machine.

Problem 3.12

The given Turing Machine can recognize the class of Turing-recognizable languages if and only if it can be simulated by the original Turing Machine. There are two moves to consider for the *left reset TM*:

1. Move head to the right. This is the same operation in both the original TM and the left reset TM.

2. Move head to the left. This is a problem for the left reset TM as it will always move its head to the beginning of the tape. Thus to emulate a left move as is in the original TM, we first mark the position of the head currently in the left reset TM, reset the head back to the beginning and then copy over the entire tape one cell to the right, except for the marked cell, which remains in the same position. After doing this, reset the head back to the new beginning and scan until the mark, and keep the head there. This emulates the exact same semantics of a left move in the original Turing Machine.

Thus we see that the left reset Turing Machine can behave exactly like the original Turing Machine and thus can recognize the class of Turing-recognizable languages.

Problem 3.13

If a Turing Machine can only move to the right or stay put, then for it to be able to emulate a left move, means that it will have to memorize everything it has seen so far to be able to work with it as in effect with a move left. To be able to memorize everything it has seen so far, the language the Turing Machine recognizes must be finite, since we only have a finite amount of memory, and thus can only recognize **regular languages**. In this essence, a move left would have to essentially simulate a DFA for the language. Because this machine can only work with regular languages, it is not equivalent to the original Turing Machine which can recognize any infinite string because of its ability to read left or right.

Problem 3.16

a

Let L_1 and L_2 be Turing-recognizable languages recognized by Turing Machines M_1 and M_2 respectively. Let M' be the TM that recognizes $L_1 \cup L_2$. Thus on every input string w just run M_1 and M_2 on the string alternately step by step. If either of the machines accepts, then let M' accept. If both halt and reject, then M' also rejects. Note that if either of the machines reject by looping so will M'. Therefore, M' will arrive to an accepting state in a finite number of steps. Thus the union of Turing-recognizable languages is closed under **union** as we can construct a Turing Machine for it.

b

Let L_1 and L_2 be Turing-recognizable languages recognized by Turing Machines M_1 and M_2 respectively. Let M' be the TM that recognizes L_1L_2. On an input string w, run the following steps:

1. Non-deterministically split the input string into two parts, $w = w_1w_2$.
2. Run M_1 on w_1. If it *rejects*, then M' *rejects*.
3. Run M_2 on w_2. If it *accepts*, then M' will also *accept* since it recognizes both w_1 and w_2 at this point. If it *rejects*, then M' also *rejects*.

The idea here is that if both M_1 and M_2 accept some cut of $w = w_1w_2$, then w belongs in the concatenation of L_1 and L_2 and M' will accept this string in a finite number of steps. Thus is closed under **concatenation**.

c

Let L be a Turing-recognizable language recognized by Turing Machine M. Let M' be the TM that recognizes L^*. On an input string w, run the following steps:

1. Non-deterministically split the string $w = w_1w_2...w_n$
2. Run M on each split string w_i for all i. If at any point it *rejects* then M' also *rejects*. It it *accepts* all strings, then M' also *accepts*.

The idea here is that if M accepts some cut of $w = w_1w_2...w_n$, then M' will also accept the string in a finite number of steps. Thus is closed under **star**.

d

Let L_1 and L_2 be Turing-recognizable languages recognized by Turing Machines M_1 and M_2 respectively. Let M' be the TM that recognizes $L_1 \cap L_2$. On an input string w, run the following steps:

1. Run M_1 on w. If it *rejects*, then M' *rejects*. If it *accepts* then move onto the next step.
2. Run M_2 on w. If it *accepts*, then M' will also *accept*. If it *rejects*, then M' also *rejects*.

The idea here is that if both M_1 and M_2 accept w, then w belongs in the intersection of L_1 and L_2 and M' will accept this string in a finite number of steps. Thus is closed under **intersection**.