Homework 5 Solutions

COMS W3261, Summer B 2021

This homework is due **Monday, 8/2/2021, at 11:59PM EST**. Submit to GradeScope (course code: X3JEX4).

Grading policy reminder: LaTeX is preferred, but neatly typed or handwritten solutions are acceptable. Your TAs may dock points for indecipherable writing. Proofs should be complete; that is, include enough information that a reader can clearly tell that the argument is rigorous.

The tool http://madebyevan.com/fsm/ may be useful for drawing finite state machines.

If a question is ambiguous, please state your assumptions. This way, we can give you credit for correct work. (Even better, post on Ed so that we can resolve the ambiguity.)

1 Problem 1 (15 points)

Use the context-free pumping lemma to prove that the following languages are not context-free. (Recall that the CFPL has the form "In any context-free language, all sufficiently long strings have some decomposition that satisfies three properties". To show that a language is not context-free, we need to contradict this statement; i.e., show that "In this particular language, there exists at least one long string such that *no* decomposition of this string satisfies all three properties.")

1. (5 points).

$$L_1 = \{a^n b^n a^n b^n \mid n \geq 1\}.$$

Assume for contradiction that L_1 is context-free. Then L_1 satisfies the context-free pumping lemma: there exists a pumping length p such that for every string $s \in L_1$ such that $|s| \geq p$, there exists a division into five pieces $s = uvxyz$ such that $|vy| > 0$, $|vxy| \leq p$, and $uv^i xy^i z \in L_1$ for all $i \geq 0$.

Consider the string $s = a^p b^p a^p b^p$. We proceed to show that every possible division of s into five substrings fails the conditions. First, we observe that in any division that satisfies $|vxy| \leq p$, the string vxy contains elements from at most two of the four substrings a^p, b^p, a^p and b^p that compose s. Thus the string $uvvxyyz$ increases the length of at most two of these four substrings. Because we must satisfy $|vy| > 0$, $uvvxyyz$ must increase the length of at least one of the four. However, this implies that two of the four substrings must be of unequal length after pumping, which is a contradiction.

Thus s cannot be pumped, which contradicts our assumption that L_1 is context-free.

2. (10 points).

$$L_2 = \{a^i b^j c^k \mid i, j, k \geq 1; ij = k\}.$$

Assume for contradiction that L_2 is context-free, in which case L_2 satisfies the CFPL and has some pumping length p. We consider the string $s = a^p b^p c^{p^2}$ and proceed to show that s cannot be pumped. We consider every possible division of s into strings $uvxyz$ in two cases:

   (a) v and y contain only a's and b's or only c's. In this case, using the assumption that $|vy| > 0$, pumping v and y increases either the size of ij (if v and y contain only a's and b's) or the size of k (if v and y contain only c's) but not both. Thus $ij \neq k$ after pumping.

   (b) v and y contain some b's and some c's but no a's. (We can rule out the possibility that v and y contain both a's and c's using the requirement that $|vxy| \leq p$.) Consider the string $uvvxyyz$ created by pumping v and y once. Because v and y contain at least one b, j increases by at least 1. Letting i and j denote the new number of a's and b's, we have $ij \geq p(p+1) = p^2 + p$.

   Using the assumption $|vxy| \leq p$, because v and y contain at least one b, we know that v and y contain at most $p-1$ c's. Thus letting k denote the new number of c's, we have $k \leq p^2 + p - 1$. Combined with the equation above, this contradicts $ij = k$ and thus s cannot be pumped.

2 Problem 2 (8 points)

Provide *implementation-level descriptions* of Turing Machines that recognize the following languages.

(Recall that an implementation-level description is less detailed than a *formal description*, in which we specify every state, transition function, and so on. You do not need to specify specific states or the transition function for this question. An implementation-level description is more detailed than a *high-level description*, in which we describe an algorithm but ignore all details of how we move the machine head or manage storage on the tape. For this question, you should describe in prose how the TM moves its head around and manages memory. Answers should be a few paragraphs (certainly more than one sentence and less than two pages.))

(Example answer for the language $\{w\#w^R : w \in \{0,1\}^*\}$: First, we read the input string from left to right and ensure it contains the # symbol. We reject if it does not. Then, beginning at the hash symbol, we shuttle back and forth between the two substrings on either side. Every time we go to the left, we cross out and remember the first uncrossed symbol we come to. If it matches the first uncrossed symbol on the right, we cross out this symbol also and continue. If it does not match, or if there are no more uncrossed symbols on the right, we reject. Once we have completed this procedure for all symbols on the left substring, we accept if and only if we have crossed out all symbols on the right substring.)

1. (4 points).

$$L_3 = \{0^n 1^n 2^n \mid n \geq 1\}$$

First, we read the input string from left to right and check to ensure that it matches the regular expression $0^+1^+2^+$. This is a regular expression, so we can see if the input is a string in this language by simulating a simple DFA. We reject if our input fails this preliminary check; otherwise, we return to the beginning of the tape.

Next, we shuttle back and forth between the substring of 0's and the substring of 1's, crossing them out in turn. We reject if we don't cross out the last 0 and the last 1 in the same step, i.e., if the number of 0's and 1's is not the same.

If the number of 0's is the same as the number of 1's, we scan across the string of crossed-out 1's and uncross each 1. We then perform the same check as in the previous step to ensure that the number of 1's is the same as the number of 2's. We accept if this condition succeeds and reject if it fails.

2. (4 points).

$$L_4 = \{a^n b^{n^2} \mid i \geq 1\}$$

First, we read the input string from left to right and check to ensure it matches the regular expression a^+b^+ as in the previous part. We then reset the tape head.

Suppose the number of a's in the input is m. We implement a subroutine that crosses off m b's for each a in the input. Thus we can accept if this procedure crosses off every b (without overshooting) and reject otherwise.

Mark each a in the input with a dot in turn. Every time we mark an a with a dot, do the following:

- Shuttle back and forth between the string of a's and the string of b's, crossing off one a and one b each time. (Cross off all the a's, including the marked ones.)
- Uncross all the a's (leaving the marks).

Once all a's have been marked, we've crossed off m^2 b's. Reject if there are uncrossed b's or if the procedure aborts early; otherwise, accept.

3 Problem 3 (12 points)

Provide *high-level descriptions* of Turing Machines that recognize the following languages.

(Recall that a high-level description is an algorithm for a Turing Machine described in prose, ignoring implementation details such as the way information is encoded or where the head needs to move. However, your TM's behavior should still be *precisely specified*: it should be clear what the Turing Machine does in every case. These answers should be on the order of a few sentences.)

(Example for the language $\{\langle G\rangle \mid \langle G\rangle$ encodes a connected graph.$\}$: First, check to see if the input encodes a graph and reject if not. Then, begin a list with an arbitrary vertex v. Add every neighbor of v to the list, then every neighbor's neighbor, and so on. Repeat this process until no new vertices are added and accept if the list contains every vertex.)

1. (4 points). The Fibonacci sequence F begins by defining $F_0 = 0$ and $F_1 = 1$, and then recursively defines F_i for all $i \geq 2$ as $F_i = F_{i-1} + F_{i-2}$.

   $$L_5 = \{w \mid w \text{ is a number in the Fibonacci sequence}\}.$$

   Our TM will enumerate the Fibonacci sequence. We begin by writing down 0 and 1. At each step, given F_i and F_{i+1}, we add them to get F_{i+2} and replace F_i. Every time we write down a new member of the Fibonacci sequence, we check to see if it matches w and accept if so. Thus if $w = F_n$, we accept after checking n numbers.

2. (4 points). Let D be any DFA.

   $$L_D = \{w \mid D \text{ accepts the input string } w\}.$$

   D is finite by definition, so our TM can contain a 'hard-coded' procedure for writing down an encoding of D. We then simulate D on w and accept if and only if D accepts.

3. (4 points). The following language consists of strings over $\{0, 1\}$ and the symbols $\cup$, *, $^+$, etc., that we usually use to write down regular expressions.

   $$L_R = \{w \mid w \text{ is a well-defined regular expression over } \{0, 1\}\}.$$

   By definition, any regular expression is $a \in \Sigma$, ε, $\emptyset$, or the union, concatenation, or star of other regular expressions.

   Our TM will recursively decompose the input using this definition. At any point, we will store an active list of potential regular expressions to be decomposed into base cases ($a \in \Sigma, \varepsilon$, or $\emptyset$). At the beginning of execution, the list contains w. At each step, we consider the first string A on our active list. If $A = R_1 \cup R_2$, $R_1 R_2$, or R_1^* for any two strings R_1, R_2, we replace A with its component parts and add these to the list. If A consists of a single symbol, ε, or $\emptyset$, we remove A from the list.

   We accept if our list eventually becomes empty. If we encounter a list item that does not meet our recursive definition of a regular expression, we reject.

4 Problem 4 (Extra credit, up to 10 points)

1. (2 points). P and NP refer to classes of languages that we will define next week. The question of whether $P = NP$ is a famous unresolved question in theoretical computer science.

   Consider the language L_6 defined as follows.

   $$L_6 = \begin{cases} \{0\} \text{ if } P \neq NP \\ \{1\} \text{ if } P = NP \end{cases}.$$

   Is this language decidable? (Hint: you don't need to know what P and NP are to answer this question.)

   While we don't know the contents of L_6, it is true either that $L_6 = \{0\}$ or that $L_6 = \{1\}$. Both of these languages are decided by some Turing Machine, so L_6 is decidable.

2. (3 points). Give a high-level description of a Turing machine that recognizes the following language.

   $$L_7 = \{\langle M \rangle \mid \langle M \rangle \text{ encodes a TM that accepts the input string 'ACCEPT ME'}\}.$$

   First, we check to make sure that the input encodes some TM. We then simulate the action of M on 'ACCEPT ME' and accept if the simulation accepts.

3. (1 point). In the previous part, you showed that L_7 was Turing-recognizable. Why is it harder to show that L_7 is Turing-decidable?

   The TM we described in the previous question clearly accepts if M accepts on 'ACCEPT ME'. However, we don't know how long M might compute before reaching an accept state. M might also run forever. To decide to reject, it seems like we have to be able to distinguish a machine that takes an arbitrarily long time to compute from one that runs forever. (As we'll see next week, this turns out to be impossible.)

4. (4 points). Recall that an *enumerator* is a Turing Machine with an attached printer that can print the contents of its tape at any point. The language of an enumerator is the set of all strings that it prints. Let G be a context-free grammar. Give a high-level description of an enumerator that writes down every string in the language of G.

   For $i = 1, 2, 3, ...$, our enumerator will simulate every possible derivation that takes at most i steps and then write down any derivation that ends in a string of terminals. As every string in the language of G can be derived in a finite number of steps, this enumerates all strings.

   (Note that this strategy implicitly explores the tree of derivations in a breadth-first manner. A depth-first search might fail, as some sequence of rules might cause our computation to loop infinitely.)