Homework 5. Due December 3rd, 9:59PM.

CS181: Fall 2020

Guidelines:

- Upload your assignments to Gradescope by 9:59 PM.
- Follow the instructions mentioned on the course webpage for uploading to Gradescope very carefully (including starting each problem on a new page and matching the pages with the assignments); this makes it easy and smooth for everyone. As the guidelines are simple enough, bad uploads will not be graded.
- You may use results proved in class without proofs as long as you state them clearly.
- Most importantly, make sure you adhere to the policies for academic honesty set out on the course webpage. The policies will be enforced strictly. Homework is a stepping stone for exams; keep in mind that reasonable partial credit will be awarded and trying the problems will help you a lot for the exams.
- All problem numbers correspond to our text 'Introduction to Theory of Computation' by Boaz Barak. So, exercise a.b refers to Chapter a, exercise b.

Campuswire: If you are having trouble or are stuck on a problem, don't hesitate to ask on campuswire!

1. Exercise 9.9. Please replace NAND-RAM program with a Turing machine for the problem. That is, consider the case where $F : \{0,1\}^* \to \{0,1\}$ takes two Turing machines P, M as input and $F(P, M) = 1$ if and only if there is some input x such that P halts on x but M does not halt on x. Prove that F is uncomputable. [1 point]

   Solution. Let us reduce from HALTONZERO. Let N_0 be the following program:

   def $N_0(x)$:

   (a) If $(x == 0)$: while() {}
   (b) Return 1.

   That is N_0 is a program that does not halt on 0 but halts on all other inputs. Let M be an input to HALTONZERO and define $\mathcal{R}(M) = (M, N_0)$. We claim that $HALTONZERO(M) = F(\mathcal{R}(M))$.

   Case 1: If $HALTONZERO(M) = 1$, then M halts on 0 and N_0 does not halt on 0 so $F(\mathcal{R}(M)) = 1$.

   Case 2: If $HALTONZERO(M) = 0$, then for any x either M does not halt or N_0 halts. So $F(\mathcal{R}(M)) = 0$.

The above gives a reduction from $HALTONZERO$ to F. As HALTONZERO is uncomputable, F is uncomputable too.

2. Exercise 9.13. Replace NAND-TM with just plain TM in the entire problem. [2 points]

   [Hint: For part (2), try to come up with a program whose description length is at most n but that takes $\omega(TOWER(n))$ steps to stop. I also highly recommend reading the two references in the problem.]

   Proof of part (1): We can use T_{BB} to compute HALTONZERO as follows: If $T_{BB}(P) = 0$, return 0, else return 1. As HALTONZERO is uncomputable, T_{BB} is uncomputable too.

   Proof of part (2): The actual proof does not use much about the TOWER function. The main idea is as follows. Suppose $f : \mathbb{N} \to \mathbb{N}$ is a computable function. Then, there is a Turing machine M_f takes n in binary on its input tape, and takes at least $f(n)$ steps on input n. This is achievable easily for any computable function: You can for instance first compute $f(n)$ and have another for loop that runs for $f(n)$ steps.

   Now, M_f has a fixed size description, say s_f. (It is a program and its description does not depend on the input length.). For every n, consider a new program P_n defined as follows:

   def P_n(x): RETURN $U_{TM}(M_f, n)$, where n is specified in binary and U_{TM} is a fixed constant size universal TM of size say c_u.

   The description length of P_n is $O(c_u) + O(s_f) + O(\log n) = O(1) + O(\log n)$. Thus, for some sufficiently big constant n_f, we would have $|P_n| < n$ for all $n \geq n_f$. This in turn implies that $NBB(n) \geq f(n)$ for all $n \geq n_f$. (As M_f takes at least $f(n)$ steps on input n, P_n takes at least $f(n)$ steps on input 0.)

   The above argument proves that for every computable function $f : \mathbb{N} \to \mathbb{N}$, there exists some constant n_f such that $f(n) \leq NBB(n)$ for all $n \geq n_f$.

   Now, returning to the problem, the main point is that $TOWER(n)$, however large it is, is computable. Moreover, even $n \cdot TOWER(n)$ is computable. For instance, if $f(n) = nTOWER(n)$, we can consider the following simple program: **def** nTOWER(n):

   (a) Set $a = 1$.
   (b) For $i = 1, 2, \ldots, n$, set $a \leftarrow 2^a$.
   (c) Set $b = 0$.
   (d) For $i = 1, \ldots, na$: $b = b + 1$.

   The number of steps taken by the above program on input n is at least $nTOWER(n)$. Further, for the case of the TOWER function as in the problem, the sequence of programs P_n could be the following: **def** $P_n(x)$:

   (a) Set $a = 1$.
   (b) For $i = 1, 2, \ldots, n$, set $a \leftarrow 2^a$.
   (c) Set $b = 0$.
   (d) For $i = 1, \ldots, n \cdot a$: $b = b + 1$.

Note that the input to the program P_n is x (which it ignores). The number of steps it takes on input 0 is at least $nTOWER(n)$. The description of length P_n is $O(1) + O(\log n)$ as we did in the general case.

Therefore from our earlier argument exists some n_f such that for all $n \geq n_f$, $nTOWER(n) \leq NBB(n)$. Thus, $\lim_{n\to\infty} TOWER(n)/NBB(n) = 0$ so that $TOWER = o(NBB)$.

3. Consider the grammar G with $V = \{R, X, S, T\}$, $\Sigma = \{0, 1\}$, $s = R$, with rules $R \to XRX|S$; $S \to 0T1|1T0$; $T \to XTX|X|\varepsilon$; $X \to 0|1$. Answer the following about the grammar: [1 point]

   (a) Give three strings in the language of G.

   (b) True or False: $T \Rightarrow^* 010$.

   (c) Give a description of the language of the grammar in english.

   (a) $\{01, 10, 0001\}$.

   (b) True.

   (c) The language is $\{x : x \text{ is } \textbf{not} \text{ a palindrome}\}$. The reason is that at some point the rule $R \to S$ must be applied and until then there are an equal number of symbols to the left and right of S. Now, S goes to $0T1$ or $1T0$. So no matter what T ends up being, the string will not be the same when looked at backward.

4. Design a context-free grammar for the following language: $L = \{x \in \{0, 1\}^* : \text{ x has more 1's than 0's}\}$. You can assume L has the empty string. [1 point]

   Let $E = \{x \in \{0, 1\}^* : \text{ x has an equal number of 1's and 0's}\}$.

   Let us first design a grammar for E: $A \to AA|0A1|1A0|\varepsilon$.

   Proof that the grammar above generates E: It is easy to check that every string generated from A has an equal number of 0's and 1's. Further, given any string x that has an equal number of 0's and 1's, it should fall into one of two cases: a) the first and last symbol are the same; b) the first and last symbol are different. In case (a), there must be a proper prefix where the number of 0's and 1's is the same so we can use the rule $A \to AA$ to generate the string. In case (b), we can use one of the other two rules $0A1$ or $1A0$.

   Now, going back to L, one idea for designing a grammar for L is to use the following. Let $x \in L$. Then, one of the the following cases holds: (1) $x \in y_1 e y_2$ with $y_1, y_2 \in L$, $e \in E$, (2) $x = y_1 e$, $y_1 \in L$, $e \in E$ or (3) $x = e y_1$, $y_1 \in L$, $e \in E$. Here, we assume y_1, y_2 are not the empty string. This allows us to recursively design the rules for L.

   Thus, given the rules we already have for E, we can do the following for getting a grammar for L:

$$S \to SAS|SA|AS|1T; \quad T \to 1T|\varepsilon; \quad A \to AA|0A1|1A0|\varepsilon.$$

5. Consider the function $EMPTY : \{0, 1\}^* \to \{0, 1\}$ that takes a DFA as input and outputs 1 if the language of the DFA is empty. That is, $EMPTY(D) = 1$ if D describes a DFA (under some encoidng - the representation is not important) that does not accept any string. Define $EQUIVALENT : \{0, 1\}^* \to \{0, 1\}$ as the function that takes two DFAs D, D' and checks

their equivalence: that is $EQUIVALENT(D, D') = 1$ if $D(x) = D'(x),\ \forall x$. Give a reduction from EQUIVALENT to EMPTY. [1 point]

[You can use high-level programming languages or pseudocode to describe your reduction.]

Proof: Given DFAs D, D', consider the function $f(x) = (D(x) \wedge NOT(D'(x))) \vee (NOT(D(x)) \wedge D'(x))$. Note that, by the closure properties of regular languages, f is regular: (1) NOT(D') is regular (as regular langauges are closed under complement), (2) $D \wedge NOT(D')$ is regular (as regular languages are closed under AND), (3) By same logic, $NOT(D) \wedge D'$ is regular, (4) $(D(x) \wedge NOT(D'(x))) \vee (NOT(D(x)) \wedge D'(x))$ is regular as regular functions are closed under taking OR.

Therefore, there exists a DFA D'' that computes f as above, and in fact we can compute the DFA for D'' from D, D' (using the arguments we saw for closure operations of regular languages). Define $\mathcal{R}(D, D') = D''$.

Finally, note that there exists an x such that $D''(x) = 1$ if and only if there exists an x such that $D(x) \neq D'(x)$. Therefore, $EQUIVALENT(D, D') = EMPTY(D'')$.