Harvard University
Computer Science 121

Problem Set 7

Due Friday, November 12, 2009 at 1:20 PM.
Submit a single PDF (lastname+ps7.pdf) of your solutions to cs121+ps7@seas.harvard.edu
Late problem sets may be turned in until Monday, November 15, 2009 at 1:20 PM with a 20% penalty.
See syllabus for collaboration policy.

Solution set.

PROBLEM 1 (10 points)

Let $L_1 = \{\langle M \rangle : M \text{ accepts } \langle M \rangle\}$ and $L_2 = \{\langle M \rangle : M \text{ rejects } \langle M \rangle\}$. Prove that there is no recursive language R such that $L_1 \subseteq R$ and $L_2 \subseteq \overline{R}$. (Hint: Suppose there were, and think about the TM that supposedly decides $\overline{R}$.)

(A) Proof by contradiction—assume there exists a recursive language R s.t. $L_1 \subseteq R$ and $L_2 \subseteq \bar{R}$. This language can decide whether a Turing machine accepts a description of itself. We can use this to solve the undecidable acceptance problem by cleverly encoding the problem with the TM. In particular, for any TM M and string x we create M' which deletes its input, writes x and then runs as before. Given the description of itself this TM ignores it. Its acceptance or rejection of itself is therefore a constant function decided by R, but this allows us to decide ATM, a contradiction.

PROBLEM 2 (10 points)

Let L_1 be a language. Prove that L_1 is r.e. if and only if there exists some recursive language L_2 such that $L_1 = \{x : \text{there exists } y \text{ such that } \langle x, y \rangle \in L_2\}$. (Hint: Imagine y gives you some information about an accepting computation on x, if one exists.)

(A) First we prove that if a language $L_1 = \{x : \text{there exists } y \text{ such that } \langle x, y \rangle \in L_2\}$ is r.e. then some L_2 is recursive. Let L_2 be a language which takes as input a string x and path y through the states of the TM which partially decides L_1. L_2 verifies the path through L_1 on x, ensuring it makes no improper transitions, and if the path arrives at an accept state it accepts. Otherwise it rejects. L_2 always halts since it's verifying a path of finite length, and strings can only be in L_1 if there exists a valid path from them to the accept state, which is membership in L_2. Finally, a string cannot be in L_1 if there is no path to the accept state, which is also the criteria for not being in L_2 (when considered matched with some string).

Now we prove if L_2 is recursive, $L_1 = \{x : \text{there exists } y \text{ such that } \langle x, y \rangle \in L_2\}$ is r.e. We construct the TM for L_1 by enumerating all strings in lexicographic order and simulating the TM for L_2 on the input and the generated string acting as a y, accepting if L_2 does. If there exists a string y where $(x, y) \in L_2$ this method will eventually find it and halt in the accept state. If there is no such string it will loop forever, satisfying the conditions for being r.e.

PROBLEM 3 (10 points)

A function $f : \Sigma^* \to \Sigma^*$ is *computable* if there exists a Turing Machine M that, when given $s \in \Sigma^*$ as input, halts with $f(s)$ written on the tape. The *range* of f is $\{f(x) : x \in \Sigma^*\}$. Prove that a nonempty language is r.e. if and only if it is the range of a computable function.

(A) First we prove that if a nonempty language is r.e. it is the range of a computable function. Recursively enumerable languages have enumerators. Consider a function which takes in a description of an enumerator and an integer, k; its output is the kth string created by the enumerator. We know this is a valid computable function by construction—the enumerator is a TM which we can run and we're counting strings up to an input value. So there exists a computable function with range equal to every language which has an enumerator, which is every r.e. language, so every r.e. language is the range of a computable function.

Now we prove that if a nonempty language is the range of a computable function it is r.e. We construct the enumerator directly. We add a "generator" to the TM M which computes f. This "generator" enumerates all input strings in lexicographic order. We then run this input through M, clear the result and repeat. This TM enumerates the range of f, and since an enumerator for a language implies it's r.e. we have that languages which are the ranges of computable functions are r.e.

PROBLEM 4 (5+10+10 points)

In this problem you will define a function that grows faster than any computable function. Thus you will prove the existence of an uncomputable function directly, without relying on Turing's diagonalization argument. The **busy-beaver function** $\beta(n)$ is the the largest number of a's that can be printed by any n-state, two-symbol Turing machine that eventually halts when started from the empty tape.
(A) Show that adding more states increases the number of a's that can be written. Specifically, show that there is a constant t such that, for all natural numbers n and m,

$$\text{if } n \geq m + t, \text{ then } \beta(n) > \beta(m).$$

(B) Show that if $f : \mathbb{N} \to \mathbb{N}$ is computable, then there exists some constant k_f such that $\beta(n+k_f) \geq f(n)$ for all n. In other words, there is an $(n + k_f)$-state Turing machine M_n that writes at least $f(n)$ a's on a blank tape before halting. (*Hint:* First recall that any TM can be simulated by one with a two-symbol alphabet. Show that, for each n, there is an $(n + c)$-state TM N_n that write n a's, where c is a small positive constant independent of n. Combine such a TM with the fixed two-tape-symbol machine F which computes f. The overall constant k_f will represent the number of "extra" states c required to construct N_n plus the number of states of F.)

(C) Show that β is not computable.

(A) Consider the optimal TM using m states. We change the machine's halting state to scan right

until a blank symbol is found and write an additional 'a.' This state then transitions to a halting state. This prints one additional 'a' using one extra state, so $\beta(m+1) > \beta(m)$ by construction.

(B) We again construct a TM printing the desired number of a's. First we simulatte a machine which prints the input to the computable function, then move the TM's head back to the start of the input, then simulate the computable function. This produces $f(n)$.

Printing n a's can be done with a machine with n states—it is simply a line of states which move the head right and print another 'a.' To guarantee we print n a's precisely we may need to add a small constant number of states. We use this machine as the first part of our simulation, placing n $a's$ on the tape.

We now move the head back to the start of the input. The easiest way to do this is to have started the prior machine on the second tape position, leaving a blank to mark the start, and have it only print $n-1$ a's. Then we add a state which sweeps left looking for a blank and writes an 'a' there. This starts the head at the beginning of the input with n a's on the tape.

Finally we simulate the computable function of interest, which requires an additional constant number of states. This writes $f(n)$ a's to the tape.

So with $n+c$ states, where c is some states to move the head and perform accounting plus simulate a computable function, we have produced the same number of a's as any computable function, as required.

(C) Proof by contradiction. Assume β is computable.

The composition of two computable functions is computable. In fact, we can use a construction similiar to the above to see why this is true. This tell us $\beta(\beta(n))$ is a computable function implementable in a constant number of states.

From part (B) we know there exists some constant, c, s.t. $\beta(n+c) \geq \beta(\beta(n))$. And from part (A) we know $\beta()$ is strictly increasing. So $n+c \geq \beta(n)$. Since $\beta()$ is greater than or equal to any computable function, it must be greater than the function which doubles its input, which is computable by construction: for every character in the original replace it with a marker value, move right until a blank is found, move right again, write the original symbol, move left until a marker is found, move right and repeat until the symbol to be doubled is a blank; finally, replace all markered values with their original values. This implies $\beta(n) \geq 2n$ and therefore $n+c \geq 2n$ which is false for all $n > c$, a contradiction.

PROBLEM 5 (5 + 5 points)

In the near future you're working as an engineer at Google/Microsoft/Facebook when your manager asks you to write the following two programs. Is this a problem? Why or why not?

(A) Take another program's code as input and decide if that program is implemented in the fewest possible lines of code.

(B) Take another program's code and remove all inaccessible (dead) code from it.

(A) Turing machines can simulate any computation and minimizing a TM is undecidable, so the program cannot solve all cases.

Proof by contradiction. We assume this TM is decidable and demonstrate how using it we could decide ATM. We take the TM of interest and modify it so it erases its input and writes a new string of interest. It then runs as normal. This machine never alters its output—it is a constant function—so its minimal version is either an accepting or rejecting state. If the state is accepting then the original TM would have accepted, and if rejecting it would have rejected. This allows us to decide ATM, an undecidable language, a contradiction.

(B) Turing machines can simulate any computation and removing dead states from a TM is undecidable, so the program cannot solve all cases.

The proof is nearly identical to the above, except the dead states will be all off path states for a computation on a particular string. If the remaining machine, the input's path through the original, includes the halt and accept state we accept, otherwise reject.

PROBLEM 6 (**Challenge** + 1 points)

Given a particular method of encoding a Turing machine M into a string $\langle M \rangle$, define T_w to be the Turing machine encoded by the string w, or if w is not a proper encoding of any Turing machine, then define T_w to be an arbitrary fixed Turing machine. Prove that if f is any computable function, then there exists some string x such that $L(T_x) = L(T_{f(x)})$.

(A) See Sipser p.223, Theorem 6.8.