CSE 103 Computational Models
Fall 2021 Prof. Fremont HW 7

(4 questions, 270 points total)

From this point in the course on, when defining a decision procedure, enumerator, etc., describing your algorithm is enough. You don't need to show how to actually implement it on a TM: we will assume the Church-Turing thesis. Just make sure to be precise: you don't need to give formal pseudocode (although you can if you think it would be clearer), but each step of your algorithm should be explained in enough detail that it's clear how the algorithm could be implemented on a computer.

1. (40 pts.) Short answer

For each of the following statements, state whether it is true or false and give a 1-2 sentence justification.

(a) If a language L and its complement $\overline{L}$ are both co-Turing-recognizable, then L is decidable.

Solution: This is true. Since L is co-Turing-recognizable, it is co-RE, as we've proved that a language is RE iff it is Turing-recognizable. Since $\overline{L}$ is also co-Turing-recognizable, its complement $\overline{\overline{L}} = L$ is Turing-recognizable and so RE. Therefore L is both RE and co-RE, and so it is decidable.

(b) If a language L is RE, then every subset of L is also RE.

Solution: This is false. For any alphabet Σ, the language Σ^* is RE: in fact it is decidable (by a TM which always accepts). So if the claim were true, then *every* language would be RE, whereas we've proved in class that some languages are not RE (e.g. $\overline{HALT}$).

2. (80 pts.) Decidable languages

Argue why each of the following languages is decidable.

(a) $L = \{\langle N_1, \ldots, N_k \rangle \mid \text{each } N_i \text{ is an NFA with alphabet } \Sigma \text{ and } \cup_i L(N_i) = \Sigma^*\}$

Solution: First, we check if the input has the form $\langle N_1, \ldots, N_k \rangle$ with each N_i being an NFA with the same alphabet; if not, we reject. Then, using our NFA constructions for union and complement (which can both be implemented as algorithms on a TM), we can build an NFA N such that $L(N) = \overline{\cup_i L(N_i)}$. Now $\cup_i L(N_i) = \Sigma^*$ if and only if $L(N) = \emptyset$, and we can decide the latter using the algorithm for E_{NFA} (the emptiness problem for NFAs).

Note: This argument shows that $L \leq_m E_{NFA}$: the reduction is given by the function $f(\langle N_1, \ldots, N_k \rangle) = N$, which is both computable and satisfies $\langle N_1, \ldots, N_k \rangle \in L \iff f(\langle N_1, \ldots, N_k \rangle) \in E_{NFA}$. (Strictly speaking, f also needs to be defined on inputs which aren't of the correct form $\langle N_1, \ldots, N_k \rangle$: for those, we can have f output some fixed NFA that has nonempty language.)

Alternate Solution: You could also check whether $\cup_i L(N_i) = \Sigma^*$ by constructing an NFA for the union, converting it to an equivalent DFA (which you had to do anyway above in order to take the complement), and then performing DFA minimization (which we did not cover in class, and which I do *not* expect you to know an algorithm for). Since the smallest DFA for the language Σ^* consists of a single accepting state with a self-loop, if this DFA is the result of minimization then $\cup_i L(N_i) = \Sigma^*$, and otherwise not.

(b) $L = \{\langle M, n \rangle \mid M \text{ is a TM that always halts within } n \text{ steps, on any input}\}$

(*Hint:* There are infinitely-many possible inputs, so you can't check them all. But do you need to check the inputs of length greater than n?)

Solution: Since the head of a TM reads one symbol and moves one cell left or right in each step of execution, in n steps it can only read the first n symbols of its input. So if a TM halts within n steps on an input x of length n, it also halts within n steps on any input string that begins with x (with arbitrary symbols afterward). Therefore, to check whether M halts within n steps on all inputs, it suffices to check whether M halts within n steps on all inputs of length at most n. We can decide this in finite time by going through the finitely-many strings of length at most n and simulating M on each one for up to n steps to see whether it halts within that time.

3. (100 pts.) Recursively-enumerable languages

Argue why each of the following languages is RE.

(Note: remember our theorem that being RE and being Turing-recognizable are equivalent; for each language below, it may be easier to prove one than the other.)

(a) Let L be the language of pairs of CFGs whose languages have at least three words in common, i.e. $\{\langle G_1, G_2\rangle \mid G_1 \text{ and } G_2 \text{ are CFGs and } |L(G_1) \cap L(G_2)| \geq 3\}$. Show that L is RE.

Solution: We can build a recognizer R for L as follows. Given an input string, reject if it is not of the form $\langle G_1, G_2\rangle$. Otherwise, enumerate all words in Σ^*: for each one, run the CYK algorithm (equivalently, the decision procedure for A_{CFG}) twice to check whether it is in both $L(G_1)$ and $L(G_2)$; if it is, increment a counter. If the counter reaches 3, accept; otherwise, continue enumerating words.

Any word in $L(G_1) \cap L(G_2)$ will eventually be enumerated, so if $|L(G_1) \cap L(G_2)| \geq 3$ then our counter will reach 3 and R will accept. If instead $|L(G_1) \cap L(G_2)| < 3$, then the counter will never reach 3, so R will run forever. Therefore R recognizes L, so L is Turing-recognizable and therefore RE.

(*Note:* It would not work to try to build a CFG whose language is the intersection of those of G_1 and G_2, since as we showed on HW 5, the context-free languages are not closed under intersection. Therefore it is possible that $L(G_1) \cap L(G_2)$ is *not* context-free.)

(b) *(60 pts.)* A *rewriting system* is a generalization of a CFG with the same components, namely a set of variables V, terminal symbols Σ, rules (or productions) P, and start symbol S, but now allowing the rules to have arbitrary strings on their left-hand sides instead of a single variable. For example, we could have a rule $xB \to xC$ which would allow rewriting the string $zxBz$ into $zxCz$ (notice that this rule is not "context-free", since it allows us to rewrite B only when it is preceded by an x), and even more complicated rules like $ABC \to DE$ are allowed.

As in the case of CFGs, the language $L(R)$ of a rewriting system R is the set of strings over Σ^* that can be derived from the start symbol S by applying the rules. Show that $L(R)$ is RE for any rewriting system R.

Solution: We can build an enumerator for $L(R)$ as follows: we'll keep track of a set D of all strings (of terminal and nonterminal symbols) we've derived so far. Initially we set $D = \{S\}$. For every string $w \in D$, we go through the rules P, applying them to w in every possible way: i.e., if the rule has the form $\alpha \to \beta$, we find all occurrences of α in w; we derive a new string for each one by replacing the occurrence of α with β. If the new string consists only of terminal symbols, we print it out. Finally, we add all newly-derived strings to D, and repeat this process.

Since we apply all the rules in all possible ways, any string that can be derived from S will eventually be added to D, and printed out if it consists only of terminals. Conversely, any string which cannot be derived from S will never be printed out. So this procedure will print out exactly the strings in $L(R)$, and is therefore an enumerator for $L(R)$. This shows that $L(R)$ is RE.

(*Note:* While $L(R)$ has to be RE, it *doesn't* have to be decidable, as was the case for context-free grammars. For any TM, you can come up with a rewriting system whose language is the same as the TM: use the rewrite rules to simulate how configurations of the TM change in each step of execution. This shows that if you take away the "context-free" restriction on grammars and allow arbitrary rewrite rules, they become as powerful as Turing machines. In particular, since *HALT* is Turing-recognizable, you could build a rewriting system for it, and this would be an example of a rewriting system with an undecidable language.)

4. (50 pts.) Using reductions to prove undecidability

Prove the following problem is undecidable by reducing one of the undecidable languages we saw in class to it: given a TM M, does M write more 1s than 0s on its tape when running on an empty input?

Solution: We will give a reduction from the halting problem, $HALT$. Given an input $\langle M,x\rangle$ for the halting problem, we construct a TM N that does the following:

(1) Simulate M on input x, but using new symbols not used by M in place of 0 and 1 so that the simulation does not require us to write any 0s or 1s.

(2) If the simulation finishes, write a single 1 on the tape and then accept.

Now if M halts on input x, then N will write a single 1 on its tape and no 0s. If instead M does not halt on input x, then N will not write any 0s or 1s on its tape. So we can decide whether $\langle M,x\rangle \in HALT$ by constructing N and checking whether it writes more 1s than 0s on its tape when run on empty input. Since $HALT$ is undecidable, so is the latter.

Note: We described the reduction above as a Turing reduction: a way to decide $HALT$ given an oracle for the language described in the problem statement (let's call it L). But it is also a mapping reduction: defining $f(\langle M,x\rangle) = \langle N\rangle$, the function f is computable, and we have $\langle M,x\rangle \in HALT \iff f(\langle M,x\rangle) \in L$ (here, as usual we've swept under the rug the detail that f needs to be defined for *all* inputs, not just those of the form $\langle M,x\rangle$: for malformed inputs we can just have f output some constant string that is not in L). Therefore $HALT \leq_m L$.