CSE 103 Computational Models
Fall 2021 Prof. Fremont HW 8

(4 questions, 260 points total)

1. (80 pts.) Mapping reductions and proving a language is not RE
Let $L = \{\langle M \rangle \mid M \text{ is a TM and } \varepsilon \notin L(M)\}$.

(a) Prove that $E_{TM} \leq_m L$.

Solution: Given a TM M, we need to build a TM N such that $\langle M \rangle \in E_{TM} \iff \langle N \rangle \in L$, i.e., M has empty language if and only if N does *not* accept the empty string. Equivalently, if M accepts *something* then we need to ensure that N accepts ε, and if M accepts nothing, then N should not accept ε. We can arrange this by defining N as follows:

1. Simulate M "in parallel" on all possible inputs. (As usual, we do this by running a finite number of simulations for a finite time each, iteratively increasing the time bound and the number of inputs.)
2. If any of the simulations end with M accepting, accept.

Now if $L(M) \neq \emptyset$, then one of the parallel simulations will complete with M accepting, so N will accept regardless of what its input was, and in particular it will accept ε. If instead $L(M) = \emptyset$, then none of the simulations will accept, so step (1) will run forever as it tries more and more inputs, and therefore N will not accept ε. So we have $\langle M \rangle \in E_{TM} \iff \langle N \rangle \in L$, as desired.

Furthermore, given $\langle M \rangle$, we can compute $\langle N \rangle$ (the "source code" for N just embeds the description of M, plus some extra code to orchestrate the parallel simulations), so setting $f(\langle M \rangle) = \langle N \rangle$ defines a computable function. Therefore f is a mapping reduction from E_{TM} to L.

(*Note:* Again, I've ignored the case where the input to f is not of the form $\langle M \rangle$ with M a TM. We can handle this in the usual way: f detects when the input x is not of this form, and outputs some constant string that is not in L. Then we will have $f(x) \notin L$, which is what we want since $x \notin E_{TM}$.)

(b) Prove that E_{TM} is not RE, which together with (a) implies that L is not RE either.

Solution: Observe that E_{TM} is co-RE: if a TM has *nonempty* language, we can detect that by simulating it in parallel on all possible inputs as above, accepting if any parallel simulation does. This gives a recognizer for $\overline{E_{TM}}$, so it is RE and therefore E_{TM} is co-RE. Now if E_{TM} were also RE, then it would be both RE and co-RE and so decidable. But we know E_{TM} is undecidable, so therefore it must not be RE.

Alternate Solution: Another way to show that E_{TM} is not RE is by reducing a known non-RE problem to it via a mapping reduction: we showed in class that if $A \leq_m B$ and A is not RE, neither is B (note that we proved this result only for mapping reductions: it's not true in general for Turing reductions). We can mapping-reduce $\overline{HALT}$ to E_{TM} as follows. Given $\langle M, x \rangle$, let N be the following TM:

1. Simulate M on x.
2. Accept.

Then if M does not halt on x, N will run forever on all inputs and so have empty language; otherwise it will accept all strings. So $\langle M, x \rangle \in \overline{HALT} \iff \langle N \rangle \in E_{TM}$, and therefore setting $f(\langle M, x \rangle) = \langle N \rangle$ gives a mapping reduction from $\overline{HALT}$ to E_{TM}. Therefore since $\overline{HALT}$ is not RE, neither is E_{TM}.

2. (40 pts.) A polynomial-time decidable problem

Let $L = \{\langle D, n\rangle \mid D \text{ is a DFA with } n \text{ states that accepts some string of length } n-1\}$. Prove that $L \in \mathsf{P}$.

Solution: We'll start by checking if D has n states, and rejecting if not (also rejecting if the input isn't of the form $\langle D, n\rangle$ at all). To see if D accepts some string of length $n-1$, we can't simply test all of these strings, since there are exponentially many of those (as long as the alphabet Σ of D has at least 2 symbols). Instead, we can check for the existence of a string in $L(D)$ of length $n-1$ by building a DFA accepting only those strings, and then checking whether it has empty language. To do this, first build a DFA A that accepts exactly the strings of length $n-1$: we can use a chain of $n+1$ states to keep track of how many symbols have been seen so far, from 0 up to n (once we get to n we don't need to keep counting, since we'll reject in any case). Then use the product construction on D and A to get a DFA P such that $L(P) = L(D) \cap L(A) = L(D) \cap \Sigma^{n-1}$. Finally, we can use the algorithm for E_{DFA} on P to check whether $L(P) = \emptyset$: if so, then $\langle D, n\rangle \notin L$, and otherwise $\langle D, n\rangle \in L$.

Since A has $n+1 = O(|D|)$ states and the same alphabet as D, its size is polynomial in $|D|$. So the product construction on D and A will also run in polynomial time, and P will have polynomial size. Therefore the algorithm for E_{DFA} will run in polynomial time too, and so the entire procedure will take polynomial time.

(*Note:* It's critical to the runtime here that D had n states, since the size of A was linear in n, not in the size of the binary encoding of n. If we allowed D to have any number of states, the algorithm would still work, but it could take exponential time: suppose D had only a single state, and we let n grow. The runtime will grow linearly with n, but the length of $\langle D, n\rangle$ will only grow *logarithmically* with n; so in terms of the length of its input, the algorithm will take exponential time.)

Alternate Solution: D accepts a string of length $n-1$ if and only if there is a path of that length from the start state to an accepting state. So we can use a variant of breadth-first search (BFS) to find all states which are reachable in exactly $n-1$ steps from the start state, and see if any of them are accepting states. Note that ordinary BFS will *not* work: BFS usually does not visit a vertex more than once, so that if for example some state q is reachable in 2 steps but *also* reachable in 5 steps along a different path, we will only visit q along the path of length 2 and fail to discover that it can also be reached along a path of length 5. To fix this problem, we'll allow a vertex to be visited multiple times, iteratively computing for $i = 0, 1, 2, \ldots n-1$ the set of states S_i reachable along a path of length i. We start out with $S_0 = \{q_0\}$, since only the initial state is reachable in 0 steps. Then having computed S_i, we can compute S_{i+1} by taking all states in S_i and finding all destinations of their outgoing transitions (formally, $S_{i+1} = \{\delta(q, s) \mid q \in S_i \text{ and } s \in \Sigma\}$). Once we have S_{n-1}, we accept if it contains any accepting state and otherwise reject.

Each iteration iterates over all states in S_i, of which there are at most $|D|$, and all the transitions from each such state, of which there are also at most $|D|$; so each iteration takes polynomial time. In total we need n iterations to compute $S_0, S_1, \ldots, S_{n-1}$, and as noted in the previous solution we have $n = O(|D|)$, so the overall time of the procedure is polynomial in the size of the input $\langle D, n\rangle$.

(*Note:* As in the previous solution, the runtime is linear in n, so the algorithm could take exponential time in the length of its input if we didn't require D to have n states.)

(*Note:* While these two solutions may seem quite different, they're basically doing the same thing under the hood. The states of the product automaton P in the first solution are of the form (q, i) where q is a state of D and $i \in \{0, \ldots, n\}$, and the start state is $(q_0, 0)$. For every transition from q to q' in D, there are corresponding transitions from (q, i) to $(q', i+1)$ in P for all $i < n$. So if the DFS performed by the E_{DFA} algorithm on P reaches state (q, i), that means that state q is reachable in D in exactly i steps from q_0. Therefore the reachable states in P for a given value of i are exactly the members of the set S_i in the second solution. Since the accepting states of P are of the form $(q, n-1)$ with q an accepting state of D, the DFS in the first solution finds a path to an accepting state if and only if S_{n-1} contains an accepting state.)

3. (60 pts.) Short answer

Answer each of the following questions, giving a 1-3 sentence justification.

Solution Note: I'm writing out my solutions in more detail than we asked for here, to help you better understand them.

(a) Consider the function $f : \{0,1\}^* \to \{0,1\}^*$ which converts binary to unary, i.e., if x is a binary representation of an integer n, then $f(x) = 1^n$ (so for example $f(101) = 11111$, $f(01) = 1$, and $f(000) = \varepsilon$). Is it possible to compute f in polynomial time?

Solution: No, f cannot be computed in polynomial time. This is because the output of f sometimes has to be exponentially-longer than its input: for example, a binary encoding of the integer $n = 2^m$ is the string $x = 10^m$, which has length $|x| = m+1$. But $f(x) = 1^n$, which has length $|f(x)| = n = 2^m$. So even writing the output of f on the tape in this case requires at least 2^m steps, which is not polynomial in the length of the input x. Therefore, f cannot be computed in polynomial time.

(b) What is wrong with the following argument? The language $L = 0^*1^*$ is regular, so it is in $\mathsf{TIME}(n)$ and thus in P. Now for any language $R \in \mathsf{P}$, there is a reduction $R \leq_P L$, so $R \in \mathsf{TIME}(n)$ as well. Therefore any problem solvable in polynomial time is solvable in linear time.

Solution: The only error in the argument is the inference that $R \in \mathsf{TIME}(n)$. A reduction $R \leq_P L$ means that an instance of R can be converted to an instance of L with the same answer in *polynomial time*, not necessarily linear time. The algorithm to solve R using this reduction would first apply the reduction to get an instance of L, then use the linear-time algorithm for L. The total time taken by this algorithm is not necessarily linear, since applying the reduction could take (say) n^2 or n^{10} time.

(*Note:* In particular, the claim that $R \leq_P L$ for all $R \in \mathsf{P}$ is correct: there are polynomial-time reductions between *any*[1] two problems in P. This is for the same reason there is a mapping reduction between any two decidable problems: the reduction can just *solve* the given problem and output an appropriate constant instance of the other problem based on whether the answer is yes or no.)

(c) Let $\mathsf{NP}_{\log}$ be a modified version of NP where the certificate for an input x is required to be of length $O(\log |x|)$. Argue that $\mathsf{P} = \mathsf{NP}_{\log}$.

Solution: Take any language $L \in \mathsf{NP}_{\log}$; since the length of the certificate for any input x is $O(\log |x|)$, it is $\leq c \log_2 |x|$ for some constant c (which could depend on L). But then there are at most $2^{c \log_2 |x|} = |x|^c$ possible values for the certificate, so we can go through all of them in polynomial time and for each one run the polynomial-time verifier for L to see if it accepts, accepting if so. This gives a polynomial-time algorithm for L, since if $x \in L$ then we will find a certificate value which the verifier accepts, and otherwise we will reject.

[1] Unless the right-hand side is $\emptyset$ or Σ^*, since then you wouldn't have anything to map yes/no answers to respectively, as we saw for mapping reductions.

4. (80 pts.) Scheduling is hard

Consider the following scheduling problem:

- There are students $S_1, \ldots S_n$, and classes $C_1, \ldots, C_m$ they can take.
- Each student has a different set of major requirements. There are three possible types of requirements:
  1. A requirement to take a specific class.
  2. An elective requirement, where the student must take at least one of a given set of classes.
  3. An incompatibility requirement, where a student is not allowed to take both of two given classes (e.g. if they are variants of the same class).

Given the students, classes, and requirements, the *schedule feasibility problem* is to determine whether it is possible to assign students to classes so that all requirements are satisfied. Let's prove that this problem is NP-complete.

(a) *(20 pts.)* Prove that the feasibility problem is in NP.

Solution: Given an instance of the problem, the certificate is just the assignment of students to classes: a verifier can then check whether every student satisfies their requirements. This can be done in polynomial time, since there are only linearly-many students, classes, and requirements, and checking a single requirement can clearly be done in polynomial time. So the feasibility problem is in NP.

(b) *(60 pts.)* Prove that the feasibility problem is NP-hard by constructing a polynomial-time reduction from *3SAT* to it.

(*Hint:* For each variable in the *3SAT* formula, create two classes: one where having a student in the class represents the variable being assigned true, and another representing the variable being assigned false. Then set up the class requirements so that an assignment of students to classes corresponds to a variable assignment that ensures at least one literal in each clause is assigned true.)

Solution: Suppose we have a 3SAT formula φ. Create a single student S, and classes C_x and $C_{\overline{x}}$ for each variable x in φ, which will represent the corresponding literals. We add an incompatibility requirement between C_x and $C_{\overline{x}}$ to prevent S from taking both classes, and an elective requirement saying it must take at least one. Then for each variable x the student must take exactly one of C_x and $C_{\overline{x}}$, representing whether we set x to true or false. Now to enforce the clauses, for each clause $c = \ell_1 \vee \ell_2 \vee \ell_3$ we add an elective requirement saying that S must take one of the three classes C_{ℓ_1}, C_{ℓ_2}, or C_{ℓ_3}. Then in any assignment of classes, at least one literal of c will be satisfied, and since this is true for each clause of φ, the assignment will satisfy φ. Conversely, if we have a satisfying assignment, the corresponding assignment of classes satisfies all our requirements. Therefore, our instance of the scheduling problem has a solution if and only if φ has a satisfying assignment. We can clearly construct the instance in polynomial time, since there are only polynomially-many students, classes, and requirements. So this gives a polynomial-time reduction from *3SAT* to the feasibility problem.

(*Note:* This shows that the problem is NP-hard even if you only allow a single student! On the other hand, the construction requires that student to take an arbitrarily-large number of classes, which isn't very realistic. If you generalize the problem to allow classes to have enrollment caps, then it remains NP-hard even if students only need to take 2 classes. More realistic versions of the problem with limited time slots, etc. would be at least as hard, so they would be NP-hard too.)