Problem Set 6: Turing Machines
CS254, Fall 2019, Anna Rafferty

Solution: Please do not post these solutions or share them with anyone outside this class. Thanks!

You are welcome to talk to others about the problems on this problem set, but you should **read and attempt each problem prior do discussing it with others.** The write up must be your own work. That means that **you should write your answers by yourself, without looking at notes from sessions with collaborators, and you must be able to explain any answer you write down.** You should list anyone you collaborated with in your collaboration form.

Due dates: All problems are due on Wednesday, October 23, as Monday, October 21 is midterm break. I won't have my usual office hours on the Monday of midterm break; instead, I'll have office hours from 1:30-2:45PM on Tuesday. As usual, I suggest you start early and try to get at least some problems done by Monday. As for all homeworks, if a problem is due on a specific day it must be turned in by 9PM on that day. See homework handout (a copy is posted on Moodle) for more specific homework policies, and review the Problem Solving Tips if you're stuck or not sure how to get started. Failing to name your files properly may result in a loss of points and potentially receiving no credit for that problem; please name your files properly!

1. Some of you have asked about whether adding an additional stack to a PDA would result in a machine more powerful than a PDA, or whether adding even more stacks would make even more powerful machines. We'll define a 2-stack PDA as exactly like a regular PDA except it has two stacks - on every step it can pop and push both stacks (or skip popping/pushing by popping/pushing ϵ). Show that 2-stack PDAs are equivalent in power to Turing machines - that is, both recognize the same set of languages (the recursively enumerable languages). *Hint: Remember that you need to argue in both directions. For one direction, use the stacks to simulate a Turing machine tape.* If you construct a machine, be specific about the details of the construction so it's clear what is happening, and argue persuasively as to why your construction is correct. You do not need to provide an inductive proof of equivalence unless you would like to.

   Solution: We have two directions to show: if a language is recognizable by a 2-stack PDA, then it is recognizable by a TM, and if a language is recognizable by a TM, it is recognizable by a 2-stack PDA.

   Direction 1: if a language is recognizable by a 2-stack PDA, then it is recognizable by a TM. Since we know that TM are equivalent in power to non-deterministic TM and equivalent in power to multitape TM, we'll actually show it's recognized by a non-deterministic, multitape TM. Let the 2-stack PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$. We construct a non-deterministic, 3-tape Turing machine $M = (Q', \Sigma, \Gamma', \delta', q_0', q_{acc}, q_{rej})$ that mimics the behavior of P. The input tape is never written to. The other two tapes are used to store the stacks: stack1Tape and stack2Tape. If we have a transition $(p, b_1', b_2') \in \delta(q, a, b_1, b_2)$ for the 2-stack PDA, we have a transition in the NTM to mimic it: if we're in state q and read a on the input tape, b_1 on stack1Tape, and b_2 on stack2Tape, we transition to state p, move R on the input tape (overwriting the a with another a so the tape doesn't change), and overwrite b_1 with b_1' and b_2 with b_2'. We want to keep the head where it is, so we also move right and move left with each head to mimic staying in place (just as in the last problem set). If b_1 or b_2 is ϵ, we move right before writing. If b_1' or b_2' is ϵ, we move left after writing. We don't further move the head position on the input tape during these transitions (again, we can mimic this behavior if using a machine with only R, L). We add a transition $\delta(q, \sqcup) = (q_{acc}, R)$ for all $q \in F$ for the 2-stack PDA. Thus, if we've ever read the entire input and are in an accept state in the PDA, we accept. This construction recognizes the same language as the PDA, since it just uses the tapes to exactly track the stacks and mimic the transitions.

   Direction 2: if a language is recognizable by a TM, it is recognizable by a 2-stack PDA: Let the TM $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{acc}, q_{rej})$. We'll construct the 2-stack PDA to mimic it, basically using the stacks to record the position of the head. stack1 will have everything to the left of head. stack2 will have as it's top item the symbol currently pointed to by head, and everything to the right of that below it. The bottom of stack symbol on the right can be added to to put in more blanks; on the left, it indicates we can't move further. We first push a symbol ($\vdash$) to mark the bottom of stack1 and a \$ on stack2 (marking its bottom). Then we push all input on to stack1, and then transfer it all to stack2; we then transfer back the \$. Now, stack1 has \$ and stack2 has the input in order, where the top symbol is the leftmost symbol of the input. This is equivalent to where the TM starts: on the left of tape symbol (represented by \$), with all of its input to the right. Whenever we have a transition in the TM $\delta(q, a) = (p, b, R)$, we need add a symbol to the left stack and remove one from the right stack. We'll add a transition $\delta'(q, \epsilon, \epsilon, a) = \{(p, b, \epsilon)$. One issue that might occur is we move off the end of stack.

Thus, we also have a transition to add an additional blank: $\delta'(q, \epsilon, \epsilon, \$) = (q, \epsilon, \sqcup\$)$, where we know that we can modify the states to write two symbols to the stack one after another - this ends with stack2 still having a \$ on the bottom.

We also need to keep track of moving left. When we move left, we'll need to write to stack2 and shift a symbol from stack1 to stack2. Concretely, on transition $\delta(q, a) = (p, b, L)$, we pop a off of stack2, push b onto stack2, pop the symbol off the top of stack1, and push that symbol onto stack2. We end by transitioning to state p. If we ever cannot read a new symbol from the bottom of stack1, we simply pop and push ϵ - i.e., we no longer move leftward.

Finally, we handle accept by maintaining the same accept state as in the TM. Since we read all of the input at the beginning, if we ever enter this state, we'll immediately accept. This 2-stack PDA exactly mimics how the TM accepts, and thus accepts the same language.

2. Sipser 3.15b, c *Note: You should know how to do (d) and (e) from class.* As usual, you should justify why any construction works: that is, why it accepts all and only the strings it should and for these closures, why the machine is a decider.

Solution: b. To show that the decidable languages are closed under concatenation, consider a language L_1 that is decided by a TM M_1 and a language L_2 that is decided by TM M_2. We construct a Turing machine M that decides $L_1 \circ L_2$. On any input w, the Turing machine begins by simulating M_1 on the portion of the input that is marked (to begin, ϵ). If M_1 accepts, then this machine simulates M_2 on the unmarked portion of the input. If M_2 accepts, then this machine rejects. If in either part M_1 or M_2 rejects, this machine marks the next unmarked character in the input and repeats. If there is no unmarked character, it rejects. This machine is checking every possible division of the string into a first part from L_1 and a second part from L_2 and accepts only if it can find a valid division. It never loops forever because it simulates $|w| + 1$ ways to split the string, and testing each way requires running M_1 and (sometimes) M_2. Since both of these machines are deciders, the machine always accepts or rejects.

c. To show that the decidable languages are closed under star, we consider a language L_1 that is decided by a TM M_1 and construct a non-deterministic Turing machine M with two tapes that decides L^*. M will consider every possible way of splitting up an input w. M works as follows on an input w:

- (Nondeterministically) select a non-empty leftmost part of the remaining part of the input w. Copy this to the second tape and delete the copied part from this tape.
- Simulate M_1 on this portion of the input. If M_1 accepts and the other tape has no input left on it (i.e., all input has been selected), then accept. If M_1 accepts and there is still input on the other tape, return to the first step. If M_1 rejects, no further computation on this branch.

We know that nondeterministic TM are equivalent in power to TM and this NTM decides the language (since M_1 cannot loop forever and there are a finite number of ways to divide any string into substrings of length at least 1), so M is a decider. The language of M is exactly L^*: If there is some way of breaking up a string $w = w_1 \dots w_k$ such that each $w_i \in L_1$, then each of the w_is will be accepted by M_1. Thus, some branch of the NTM will accept. If there is no way to break up the string w so that each part is in L_1, then every branch of the NTM must halt without accepting (because M_1 will reject some component string for every possible way of breaking up the input).

3. Sipser 3.16b, c *Note: You should know how to do (d) from class.*

Solution: b. To show that r.e. languages are closed under concatenation, we consider a language L_1 that is recognized by a TM M_1 and a language L_2 that is recognized by TM M_2, and construct a Turing machine M that recognizes $L_1 \circ L_2$. We use the marking strategy described in the previous problem, but augment it with a count of how many steps we've run (similar to the proof showing that enumerators and TMs recognize the same set of languages). Thus, we repeat the following for $i = 1, 2, 3, \dots$:

- For $j = 0, 1, \dots, |w|$:
  1. Simulate M_1 for i steps on the [0,j]th portion of w, where the [0,0]th portion is ϵ and $[0, |w|]$ is w.
  2. If M_1 accepts, simulate M_2 for i steps on the $[j, |w|]$th portion of w. If M_2 accepts, accept. If M_1 or M_2 rejects or is still running, go back to step 1.

This machine will eventually accept if there's some way of dividing the string into a part from L_1 and a part from L_2, since we assume that M_1 and M_2 recognize these languages and thus accept any string in them in a finite number of steps. If there is no way to divide up the string into a part from L_1 and a part from L_2, then M won't accept because for every split and number of steps we consider, at least one of M_1 and M_2 will not accept. Thus, M accepts w iff $w \in L_1 \circ L_2$..

c. To show that r.e. languages are closed under star, we can actually use the same exact construction as above. We consider a language L_1 that is recognized by a TM M_1, and construct a non-deterministic Turing machine M that recognizes L_1^*. We'll have two tapes in our NTM M. M works as follows on an input w:

1. (Nondeterministically) select a non-empty leftmost part of the remaining part of the input w. Copy this to the second tape and delete the copied part from this tape.
2. Simulate M_1 on this portion of the input. If M_1 accepts and the other tape has no input left on it (i.e., all input has been selected), then accept. If M_1 accepts and there is still input on the other tape, return to the first step. If M_1 rejects, no further computation on this branch.

This machine tries all the ways of dividing up a string into parts, selecting among the choices non-deterministically so that all versions are run in parallel. If there is some way of breaking up a string $w = w_1 \dots w_k$ such that each $w_i \in L_1$, then each of the w_is will be accepted by M_1. Thus, some branch of the NTM will accept. If there is no way to break up the string w so that each part is in L_1, then every branch of the NTM will either loop (if M_1 loops on one of the strings that we're considering) or halt without accepting. Thus, M accepts w iff $w \in L_1^*$.

4. Sipser 4.29. In your answer, you can describe the Turing machine at a high level, and assume that we have a reasonable encoding for a CFG in binary form. You should argue convincingly why your Turing machine decides this language, although as above, you do not need to provide an inductive proof unless you would like to.

Solution: We want to design a TM M such that $L(M) = C_{CFG}$. M works as follows on input $\langle G, k \rangle$. Let $G = (V, \Sigma, R, S)$. Let b be the maximum number of symbols in the right-hand side of a rule; if this results in $b = 1$, then we set $b = 2$. First, M checks every string of length $\leq b^{|V|+1}$ to see if it is generated by G, and keeps a count c_1 of the number of strings that are generated by G. (We know this is decidable because A_{CFG} is decidable.) If $c_1 = k = 0$, then we accept. If $c_1 > k$, we reject. Otherwise, we continue and check every string of length $> b^{|V|+1}$ and $\leq 2 * b^{|V|+1}$, keeping a count that we'll call c_2. If $c_2 > 0$, then we accept if $k = \infty$ and reject if $k \neq \infty$. If $c_2 = 0$, then we accept if $c_1 = k$.

We need to show that this machine is a decider and that it accepts all strings in C_{CFG} and rejects all strings not in C_{CFG}. First, it is a decider because there are a finite number of strings for it to check - once it has checked all strings of length up to $\leq 2 * b^{|V|+1}$, it goes to an accept or reject state, and thus it always terminates in finite steps.

Second, it decides the language C_{CFG}. There are two options to consider: G is finite (i.e., generates a finite number of strings) or G is infinite. If G is finite, it cannot generate any strings longer than the pumping length. (Imagine it did generate strings longer than the pumping length. Then by the pumping lemma, we could pump those strings with any i, resulting in an infinite number of strings that must be in the language. Thus, G cannot generate any strings longer than the pumping length if it is finite.) From our proof of the pumping lemma, we know $b^{|V|+1}$ is a valid pumping length. Thus, if G is finite, we will count all of the strings it generates and since we will find no strings of length $> b^{|V|+1}$ and $\leq 2 * b^{|V|+1}$ that are generated, we will accept exactly when k is equal to the number of strings.

Now, we need to show that if G is infinite, this TM will accept iff $k = \infty$. Our claim is that if G is infinite, it generates at least one string of length $> b^{|V|+1}$ and $\leq 2 * b^{|V|+1}$. We will show this by contradiction. Imagine there is no string of length $> b^{|V|+1}$ and $\leq 2 * b^{|V|+1}$. Then G must have infinitely many strings longer than $2 * b^{|V|+1}$. Choose the shortest such string w. We know we can write $w = uvxyz$ with $|vyx| \leq p$. We choose $i = 0$. Then $uxz \in L(G)$, and $|uxz|$ is at a minimum $|w| - p$ and at a maximum $|w| - 1$. Since $p = b^{|V|+1}$ and $|w| \geq 2 * b^{|V|+1} + 1$, this means that the minimum value for $|uxz|$ is $2 * b^{|V|+1} + 1 - b^{|V|+1} = b^{|V|+1} + 1$. The maximum value for $|uxz|$ must be $2 * b^{|V|+1}$, since we chose w to be the smallest string that was in the language and of length greater than $2 * b^{|V|+1}$. But this means that $uxz \in L$ and its length is $> b^{|V|+1}$ and $\leq 2 * b^{|V|+1}$. Thus, we have a contradiction and there is a string of length $> b^{|V|+1}$ and $\leq 2 * b^{|V|+1}$ if G is infinite. Since we count how many strings are in this range, we will determine if G is infinite by our check as to whether $c_2 = 0$. If it is not, then we accept iff $k = \infty$, and if it is, then we have verified that it is not infinite and we accept exactly when $k = c_1$.

5. Let $EQ_{DFA} = \{\langle A, B \rangle \mid A \text{ and } B \text{ are DFAs and } L(A) = L(B)\}$. Describe a Turing machine that decides EQ_{DFA} by testing A and B on all strings up to a certain length. Your answer should make clear exactly which length works

and why, and how your Turing machine computes that length from the input; your length does not necessarily have to be the minimum possible length (i.e., you must show that your length works, but not that all smaller lengths do not work). You can describe the Turing machine at a high level, and assume that we have a reasonable encoding for a DFA in binary string form. *Hint: Think about pumping lengths. Note that while Sipser provides a proof the EQ_{DFA} is decidable, this question is asking you to prove this in a different way.*

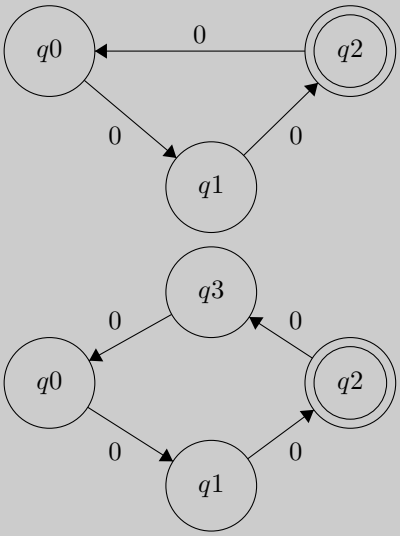
Solution: We'll use a bound for the length of strings we need to test that isn't necessarily tight, but that isn't too bad to prove it works. We let Q_A be the states in DFA A and Q_B be the states in DFA B. We know that $|Q_A|$ is a valid pumping length for A and $|Q_B|$ is a valid pumping length for B, based on our proof of the pumping lemma. We'll let our bound be $|Q_A| \cdot |Q_B|$; we know we can compute this since we've shown TMs that do multiplication, and our DFA string encodings will give the number of states in each (likely as a string of 0s).

Then, we seek to show that $L(A) = L(B)$ iff for all w such that $|w| \leq |Q_A| \cdot |Q_B|$, $w \in L(A)$ iff $w \in L(B)$. We construct a new machine P that accepts strings that are in $L(A)$ but not in $L(B)$ or in $L(B)$ but not in $L(A)$: $L(P) = \{w \mid w \in L(A) \iff w \notin L(B)$. This can be done by following the product construction that we used for intersection, but accepting iff exactly one of the states in the final pair is a final state. Recall that our algorithm for this construction created states that were pairs, with one state from A and one from B; thus, DFA P has $|Q_A| \cdot |Q_B|$ states. Let w be the shortest string in $L(P)$. If we can show that $|w| \leq |Q_A| \cdot |Q_B|$, then we have shown that if $L(P)$ has any strings (i.e., $L(A) \neq L(B)$), then it must have at least one that is less than $|Q_A| \cdot |Q_B|$ and thus testing all strings up to that length will suffice to show that $L(A) = L(B)$. Assume the shortest string $w \in L(P)$ has length $|w| > |Q_A| \cdot |Q_B|$. To accept or reject w, P visits $|w| + 1$ states; thus, it visits at least $|Q_A| \cdot |Q_B| + 1$ states, so by the pigeonhole principle, at least one state must be repeated. That means there's some substring of s of length at least 1 that takes P from q back to q; we could omit this substring from w and still end in the accept state, since we'd only have omitted the loop. But this is a contradiction, since we've assumed w is the shortest string in $L(P)$. Thus, our assumption is wrong and the shortest string in $L(P)$ has length less than or equal to $|Q_A| \cdot |Q_B|$. Hence, our decider checks all strings up to this length in both A and B, and accepts if each returns the same answer from both machines and rejects otherwise. (We again rely in having a canonical order to be able to move through all strings up to a certain length.)

Note: Some students wrote to use the maximum of $|Q_A|$ and $|Q_B|$ (the maximum of the minimum valid pumping length for each machine). This doesn't quite work because of the fact that two different strings longer than the pumping length may "pump down" to the same string: i.e., one language has string $w = xy_1z$ and the other has string $w' = xy_2z$, so $xy_1^0z = xz = xy_2^0z$: they agree for shorter strings, but not for longer strings.

Here's a more formal counterexample: Let $\Sigma = \{0\}$. Consider a machine M_A with states $q_0, \ldots, q_n$ ($n \geq 2$) and transitions from each q_i to q_{i+1}, with a loop back from q_n to q_0. Then let M_B have states $q_0, \ldots, q_{n+1}$ and transitions from each q_i to q_{i+1}, with a loop back from q_{n+1} to q_0. In both machines, the only accept state is q_n. Then the machines agree on strings 0^k for $0 \leq k \leq 2n$, but disagree on 0^{2n+1}. However, the max states is $n + 2$, which is less than $2n + 1$ for all $n \geq 2$.

Here's what those machine look like if $n = 2$:



If we consider strings up to length 4 (the maximum number of states in either machine), then both machines accept only the string 00. However, their languages are not the same: the first machine accepts 00000, but the second machine does not (it accepts 000000 while the other machine does not).

6. Let $\Sigma = \{0, 1\}$. Let's fix an ordering of all of the strings in Σ^*: we list all strings in Σ^0 in alphabetical order (i.e., just ϵ), then all strings in Σ^1 in alphabetical order (i.e., 0, 1), then all strings in Σ^2 in alphabetical order (i.e., 00,

01, 10, 11), etc. Call this the canonical order of Σ^*. Prove that a language $A \subseteq \Sigma^*$ is decidable if and only if there is an enumerator E that enumerates A in canonical order. (As above, no inductive proof is needed unless desired.)

Solution: First, we show that if we have a decider M with $L(M) = A$, we can build an enumeration machine E that enumerates A in the canonical order. For each string $w \in \{0,1\}^*$ taken in the canonical order, simulate M on w and enumerate w if M accepts. By the assumption that M is a decider, we know that E enumerates exactly A and we are considering all strings, and thus the strings in A, in canonical order.

Next, we show that if we have an enumeration machine E that enumerates A in the canonical order, then there must exist a Turing machine that decides A. There are two cases: A is finite, or A is infinite. If A is finite, then it is in fact regular as we could build a regular expression of the union of all the finite strings, which would match exactly those strings. All regular languages are decidable. If A is infinite, then there is some string $y \in A$ that comes after w in the canonical order. Our decider for infinite A will work as follows: we simulate E until either w is enumerated or some string y that comes after w in the canonical order is enumerated. In the former case, we accept; in the latter, we reject. By assumption, eventually such a string w or y must be enumerated, and therefore our Turing machine is a decider. By the assumption about Es operation, we can conclude that our TM accepts exactly A.

7. **Optional** supplemental problems: Some of you have asked for more practice problems (including additional problems that are more difficult), while others feel overwhelmed by too much homework. Please note that you do not need to do the additional practice problems; they're simply there as a supplement. The ones I'm suggesting this week are ones with answers that I think can cement your understanding of designing TMs and what it means to be decidable (we'll save undecidability for the next problem set). For extra practice, you might try 3.5, 3.22, 4.10, 4.12, or 4.23 (this one is somewhat tougher than the others, I think).