18.404/6.5400 Recitation 5

This recitation covers configurations and the computation history method, with examples.

1 TM Configurations

A *configuration* of a Turing machine (textbook p.168) completely represents the status of the TM. It consists of the current:

- state,
- location of the head, and
- tape contents.

To be able to write a configuration to a tape, we usually encode it as a string by inserting the state into the tape contents at the head location. In other words, we use uqv, where uv is the tape contents, q is the state, and the head is currently at the first symbol of v.

1.1 Each configuration can only yield one next configuration

Given a configuration of a deterministic TM, we can always figure out what the next configuration will be. This is because a configuration contains all the inputs needed to calculate the transition function of the TM.

1.2 A_{LBA} is Decidable

(textbook Theorem 5.9) Let a *linear bounded automaton* (LBA) be a Turing machine which is not allowed to move its head off its input. Then show that the language

$$A_{\mathsf{LBA}} = \{\langle M, w\rangle \mid \text{M is an LBA that accepts string } w\}$$

is decidable.

Proof. The key is to realize that **an LBA has a finite number of possible configurations.** One way of seeing this is to actually calculate how many there are:

- The number of states is $|Q|$.
- The number of possible head locations is the length of the input; call it n.
- The number of possible tape contents is $|\Gamma|^n$, where Γ is the tape alphabet.

So the number of possible configurations is exactly $|Q| \cdot n \cdot |\Gamma|^n$. Call this number K.

Now what happens if we simulate the LBA M for K steps? If after K steps it has halted, we will know whether it has accepted or not. If it has still not halted, then it has gone through $K + 1$ configurations. But since there are only K unique configurations, M must have repeated a configuration somewhere, which means it must be looping.

Finally, we construct a TM which decides A_{LBA}:
$L =$ " On input $\langle M, w\rangle$, where M is an LBA and w is a string:

1. Simulate M on w for K steps.
2. If it has halted, copy the output of M. Otherwise, *reject*."

□

2 Computation History Method

Often, our reduction proofs in this class have a step that looks like "Simulate M on w." But what if we are working with a model of computation which can't simulate Turing machines?

2.1 Computation Histories

Let the *computation history* of M on w be the sequence of configurations $C_1, C_2, \ldots, C_k$ that M goes through when given input w. Note that for a given deterministic M and input string w, there exists only one computation history of M on w.

2.2 The Idea

If M accepts w, then the computation history of M on w will satisfy the following three properties:

1. C_1 is of the form $q_0 w$, where q_0 is the start state.
2. C_k includes an accepting state.
3. For each $i \in [1, k-1]$, C_i yields C_{i+1} according to the transition function of M.

The big ideas at play are that:

- An accepting computation history of M on w ***exists*** if and only if M accepts w, and
- Checking whether a given computation history is a ***accepting/rejecting computation history of*** M ***on*** w is easier than simulating M on w.

2.3 When to use this method

The computation history method is often suitable for problems which:

- Deal with *existence*. Philosophically, this is because M accepts w if and only if there *exists* some accepting computation history of M on w.
- Involve models of computation which are not as powerful as TMs (and thus can't simulate them). When simulating M on w is not possible, it's natural to ask whether one can instead check *computation histories* of M on w. I like to describe this using the **pset grader mentality:**

  > *I might not remember how to solve this problem by myself, but I am 100% absolutely certain that this student's solution is wrong.*

Indeed, the examples we've seen using the computation history method fall roughly under these criteria: $E_{\mathsf{LBA}}, ALL_{\mathsf{CFG}}$, etc.

2.4 ALL_{PDA} is Undecidable

Let's try to use the compuation history method to prove that the following language is undecidable

$$ALL_{PDA} = \{\langle M \rangle \mid M \text{ is a PDA and } L(M) = \Sigma^*\}$$

Proof. Reduce from A_{TM}.

Assume for sake of contradiction that we have a decider for ALL_{PDA} call it D. Then construct a decider for A_{TM} as follows:
$L =$ " On input $\langle M, w \rangle$:

1. Construct a PDA, $P_{M,w}$, which accepts all string that are not an accepting computation history of M on w
2. Feed $\langle P_{M,w} \rangle$ into D.
3. If D accepts, that means that there's no accepting computation history of M on w , so we can *reject*. (otherwise $\langle P_{M,w} \rangle$ would have rejected that history, and hence D would have rejected as well)
4. Similarly, if D rejects, *accept*."

Hopefully the overall logic in this reduction is familiar. The only new component is the structure of $P_{M,w}$. The main idea behind the construction is that the PDA can nondeterministically check for all the possible errors in the history, and accept a string if it has any error

Recall that w is an accepting computation history if and only if:

1. $C_0 = q_0 w$
2. C_{i+1} follows by a single legal transition from C_i
3. C_{last} has an accept state

To check that (1) does not hold, $P_{M,w}$ can hard code the expected C_0 to check it and accept if it does not match. Similarly for (3), it can simply scan the input and accept it if it never sees an accept state of the Turing machine. Checking that (2) does not hold for some i is a little trickier.

To check for (2), we would hope to use the Stack to store C_i and compare it with C_{i+1}. After all, if the computation history is legitimate, they should be equal except for at most two consecutive places (the old and new locations of the head). The only issue is that when we push C_i into the stack it comes out reversed. To deal with this, we change the definition of the computation history so that it reverses every other configuration. In particular, instead of $C_0 \# C_1 \dots C_{last}$ the computation history is defined as $C_0 \# C_1^R \# C_2 \# C_3^R \dots$. That way when we push C_i into the stack, it comes out in the same orientation as C_{i+1}.

To summarize, $P_{M,w}$ accepts all string other than an accepting computation histories of M on w, where every other configuration is written in reverse. To do so, it nondeterministically pushes C_i on the stack, and looks for an error in the transition to C_{i+1} by comparing one element at a time. If any of those threads fine a discrepancy it accepts. It also has a hardcoded expectation for C_0 and it accepts if it does not match it, and it accepts if it never sees an accept state of the Turing machine.

Since the language of $P_{M,w}$ is all strings iff and only if M does not accept w, it allows us to use a decider for ALL_{PDA} to construct a decider for A_{ATM}, which proves that ALL_{PDA} is not decidable.

□