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. A configuration consists of the Turing machine's current:

- state q of the finite control,
- location/position of the head p, and
- tape contents t.

When writing out configurations as a string, we usually use a standard encoding. We write out the tape contents, then insert the state into the tape contents right before the symbol that the head is at. 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. For example, if the tape contains the string `abcd`, the state of the machine is q_5, and the head is currently above `c`, we insert q_5 right before `c` to get the encoding $\texttt{ab}q_5\texttt{cd}$.

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. Specifically, if we initialize the Turing machine with the current configuration, then simulate one step, this gives the next configuration.

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. Show that the language

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

is decidable.

Proof. The key is to realize that **a LBA has a finite number of possible configurations.** A general Turing machine can access an arbitrary number of tape cells, but a LBA can only move on and modify the tape area containing the input. Specifically, to calculate the number of possible configurations:

- The number of possible 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 B 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, B must have repeated a configuration somewhere, which means it must be looping.

We can therefore construct a decider L for A_{LBA}:
$L =$ " On input $\langle B, w\rangle$, where B is a LBA and w is a string:

- Simulate B on w for K steps, where $K = |Q| \cdot n \cdot |\Gamma|^n$, and n is the length of w.
- If the simulation accepts, *accept*.
- If the simulation rejects or has not halted, *reject*.

$\square$

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.

When we write out computation histories on a tape, we encode each configuration C_i as a string c_i. We then join together the strings c_i, separating them by a marker symbol # (which we assume by convention is not in any of the original strings c_i). Then, our entire encoded history looks like $c_1\#c_2\#\ldots\#c_k$.

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 the correct start configuration: the machine is in the start state q_0, the head is at the start of the tape, and the tape contains the input w.
2. C_k is an accepting configuration: the machine is in the accept state q_{acc}.
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 Turing machines (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 every step in this student's solution is correct (or that a certain step in 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 $E_{\mathbf{LBA}}$ is Undecidable

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

$$E_{\mathsf{LBA}} = \{\langle B\rangle \mid \text{B is a } \mathsf{LBA} \text{ and } L(B) = \emptyset\}$$

To show undecidability, we want to reduce from A_{TM}. To do so, we construct a LBA which tests if a given computation history for M on w is accepting. If M accepts w, then there is some accepting computation history, and the LBA will accept this string. If M does not accept w, then there is no accepting computation history, and the language of the LBA will be empty.

To check whether a computation history $c_1\#c_2\#\ldots\#c_k$ is accepting, we need to check:

1. The start configuration is encoded correctly: $c_1 = q_0w$ (start state q_0, input w on tape).
2. Each transition from c_i to c_{i+1} is correct.
3. c_k contains the accept state q_{acc}.

For our constructed LBA to exist, we need to show that a LBA can test all these conditions, given the computation history as input. For the start configuration, we need to check that c_1 matches the finite string q_0w; as this is a finite amount of information, we can encode it into the states of the LBA. To check a transition $c_i \to c_{i+1}$, the LBA can sweep across the two configurations, comparing matching portions of them and checking that the transition is valid. For the last configuration, the LBA just needs to check that q_{acc} is on the tape.

After arguing we can construct an LBA to check computation histories, we can write out our reduction:

Proof. Reduce from A_{TM}.

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

1. Construct a LBA, $B_{M,w}$, which tests if a given computation history for M on w is accepting or not.
2. Feed $\langle B_{M,w} \rangle$ into D.
3. If D accepts, then $L(B)$ is empty and there is no accepting computation history of B on w, so we can *reject*.
4. If D rejects, then $L(B)$ is nonempty and there is an accepting computation history of B on w, so we can *accept*.

□

2.5 $ALL_{\textbf{PDA}}$ is Undecidable

Let's use the computation history method to show another language is undecidable:

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

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 strings 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*."

The overall logic in this reduction is very similar to the previous problem. The only new component is the structure of $P_{M,w}$: instead of checking for accepting computation histories, we check for rejecting ones.

When checking accepting computation histories, we need multiple conditions (start state, transitions, accept state) to all hold. However, when checking for non-acceptance, we only need one of the conditions to fail. We can then split the checking into threads, with each thread checking a single condition.

The main idea behind the construction is that the PDA can non-deterministically 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. $c_1 = q_0 w$
2. c_{i+1} follows by a single legal transition from c_i
3. c_k has an accept state

To check that (1) does not hold, $P_{M,w}$ can hard code the expected c_1 and accept if it does not match. Similarly for (3), it can scan the input and accept 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 $c_1\#c_2\#c_3\#c_4\#\ldots$, the computation history is defined as $c_1\#c_2^R\#c_3\#c_4^R\#\ldots$. 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 strings other than an accepting computation history of M on w, where every other configuration is written in reverse. To do so, it none-deterministically 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 finds a discrepancy it accepts. It also has a hardcoded expectation for c_1 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 if and only if M does not accept w, it allows us to use a decider for ALL_{PDA} to construct a decider for A_{TM}, which proves that ALL_{PDA} is not decidable.

$\square$