Recitation 05: Configurations and Computation Histories

In this recitation, we look at computation histories, which provide a powerful method of proving the undecidability of certain languages.

TM Configurations

A *configuration* of a Turing machine 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 uqv, where uv is the tape content, 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 ab$q_5$cd. For a deterministic TM (or any deterministic machine in general), given the current configuration, we can always figure out the next configuration by simulating one step of the machine.

A_{LBA} is Decidable

Recall that an LBA, or linearly bounded automaton, is an automaton that cannot move its head off the input string.

Theorem 1. *$A_{\mathsf{LBA}} = \{\langle B, w\rangle \mid B$ is an LBA 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, since a configuration depends only on the current state (of which there are $|Q|$), the head location (of which there are $n = |w|$), and the tape contents (of which there are $|\Gamma|^n$), we can bound the number of possible configurations as $K = |Q| \cdot n \cdot |\Gamma|^n$.

Note how we specify K is an upper bound on the number of possible configurations, not the number of configurations of the LBA: there is no reason that the LBA must go through all K configurations.

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 at most 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 $\langle B, w\rangle$:

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

□

Computation History Method

So far in this course, if we want to prove a language involving a particular model of computation (e.g. TMs, multi-tape TMs) is undecidable, we often reduce from A_{TM}. In the proof, we often create an instance of that model of computation and include a step that looks like "simulate a TM M on w using my model of computation in order to check whether M accepts/rejects on w," leading to a contradiction as this decides A_{TM}. But what if we are working with a model of computation which can't simulate Turing machines? It turns out that the model of computation may still have the capability of checking whether the work done by the TM, or the *computation history*, leads to an accept – this is the *computation history method*.

Definition 1. *Let the* computation history *of M on w be the sequence of configurations $C_1, C_2, \ldots, C_k$ that M goes through on 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 a computation history on a tape, we often encode it as $c_1\#c_2\#\ldots\#c_k$, where c_i are the string encodings of the configurations C_i and # is a delimiter symbol that we assume was not part of M's alphabet.

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 q_0w: 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 $i \in \{1, \ldots, k-1\}$, c_i yields c_{i+1} according to the transition function of M.

For M to not accept w, then there exists no accepting computation history of M on w. Checking whether a string is not an accepting computation history amounts to checking that any one of the three properties are false, which often is doable with a simple model of computation.

In the remaining examples, we review two of the computation history proofs we've seen in lecture already, E_{LBA} and ALL_{PDA}, before looking at a new example $E_{\mathsf{2WAY-PDA}}$.

E_{LBA} *is Undecidable*

Example 1. *Show that*

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

is undecidable.

Solution 1. *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 exists some accepting computation history, and the* LBA *will accept that string. If M does not accept w, then there is no accepting computation history, and the language of the* LBA *will be empty.*

Again, note how we can assume the input string to the LBA is in the form of a computation history (i.e. it has the right encoding).

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 as $c_1 = q_0w$.*

2. *Each transition from c_i to c_{i+1} is correct.*

3. *c_k contains the accept state q_{acc}.*

So our goal is to design a LBA *(given $\langle M, w\rangle$) that can check all three conditions. For the first condition, we can check that c_1 matches the finite string q_0w using the states of the* LBA*. For the second condition, 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.*

Note how we are designing a LBA specific to $\langle M, w\rangle$, so we can "hardcode" q_0w using a finite number of states, as q_0w is independent of the input to the LBA.

Summing it all together: We proceed via a reduction 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 by checking each of the three conditions as described above.*
2. *Feed $\langle B_{M,w}\rangle$ into D.*
3. *If D accepts, then $L(B_{M,w})$ is empty and there is no accepting computation history of M on w, so we reject.*
4. *If D rejects, then $L(B_{M,w})$ is nonempty and there is an accepting computation history of M on w, so we accept.*

ALL_{PDA} *is Undecidable*

In the previous example problem, we saw how we can check for an accepting computation history using LBAs. Here, we work with a more basic model of computation, PDAs, which we are only able to show have the capability for checking whether a string is *not* an accepting computation history. However, since we are instead looking at the language being Σ^* rather than $\emptyset$, this is enough to create the reduction from A_{TM}.

Example 2. *Show that*

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

is undecidable.

Solution 2. *We first outline our reduction, which is similar in flavor to the previous example, but we instead need a* PDA *that accepts all strings* except *an accepting computation history of M on w.*

We 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 reject.*

4. *Similarly, if D rejects, we accept."*

The big question now is how to design $P_{M,w}$. As previously noted, when checking accepting computation histories, we need multiple conditions (i.e. that of the start state, transitions, and accept state) to all hold. However, when checking for non-acceptance, we only need one of the conditions to fail. Since PDAs *are non-deterministic, we can non-deterministically go down a branch checking a single condition, and we accept on that branch (and thus overall) if the condition fails.*

In a rough sense, this corresponds to how non-determinism allows us to compute logical ORs (between several conditions) really easily but not logical ANDs.

Recall that $c_1\#c_2\dots\#c_k$ is an accepting computation history if and only if

1. *$c_1 = q_0w$,*
2. *c_{i+1} follows by a single legal transition from c_i for all i, and*
3. *c_k has an accept state.*

To check that (1) does not hold, $P_{M,w}$ can hard code the expected c_1 (i.e. write it into its states) and accept if it does not match. For (3), it can scan the input and accept if it never sees an accept state of the Turing machine. To check that (2) does not hold, we can non-deterministically choose a particular i and focus on the specific case that checks whether (2) does not hold for that i.

Our hope is 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), and we can check whether this transition is correct based on finite state machine of M. The only issue is that when we push c_i into the stack, then pop it to compare to c_{i+1}, c_i comes out reversed, which stops us from comparing the strings. To deal with this, we encode our computation history in a more convenient fashion: we reverse every other configuration. In particular, instead of $c_1\#c_2\#c_3\#c_4\#\dots$, the computation history is encoded as $c_1\#c_2^R\#c_3\#c_4^R\#\dots$. That way, when we push c_i into the stack, it comes out in the same orientation as c_{i+1}. Given c_i and c_{i+1} in the correct order, we can design a PDA that performs the check of whether c_{i+1} does not follow from c_i or not.

Note how we have not really changed the essence of a computation history; we have only changed its encoding.

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 non-deterministically goes down any of the following three branches:

1. *Accept if c_1 does not match q_0w.*
2. *Accept if for some non-deterministically chosen i, c_{i+1} does not follow from a transition from c_i (use the stack to store c_i, and compare it to c_{i+1} by popping it out, noting that we reversed the directions properly to check the strings in the same order).*
3. *Accept if c_k does not have an accept state.*

Meta-analysis of the Computation History Method

A common question is when to use the computation history method to prove undecidability. There really is no hard and fast rule for this, but we can have some priors on when the method is suitable:

- The problem should roughly deal with *existence*. Philosophically, this is because M accepts w if and only if there *exists* some accepting computation history of M on w.
- The problem should 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. One way to describe this is using a (potential) grader's 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 two examples this recitation (E_{LBA} and ALL_{PDA}) fall under these criteria.

Another thing to note is that for E_{LBA}, we made a machine that only accepts accepting computation histories of M on w, while for ALL_{PDA}, we made a machine that accepts everything *except* accepting computation histories of M on w. This is because the first machine's language is empty if and only if M rejects w (and thus dealing with the emptiness criterion makes sense), while the second machine's language is everything if and only if M rejects w (and thus dealing with the ALL criterion makes sense).

We can refer to them in the plural sense of "computation histories" if we want to allow M to be non-deterministic, but more precisely for A_{TM}, where we deal with deterministic machines, we only need to refer to the (single, if it exists) accepting computation history."

$E_{\mathsf{2WAY-PDA}}$ *is Undecidable*

We look at one final example of using computation histories on a new model of machine, the $\mathsf{2WAY-PDA}$.

Definition 2. *A* 2-way pushdown automaton *(*$\mathsf{2WAY-PDA}$*) is a non-deterministic pushdown automaton that has a single stack and that can move its input head in* <u>*both*</u> *directions on the input tape. In addition, we assume that a* $\mathsf{2WAY-PDA}$ *is capable of detecting when its input head is at either end of its input tape. A* $\mathsf{2WAY-PDA}$ *accepts its input by entering an accept state.*

As an exercise, prove that $\mathsf{2WAY-PDA}$s are more powerful than normal PDAs. Hint: can you design a $\mathsf{2WAY-PDA}$ that can recognize the language $\{a^n b^n c^n \mid n \geq 0\}$?

Example 3. *Show that*

$$E_{\mathsf{2WAY-PDA}} = \{\langle P\rangle | P \text{ is a } \mathsf{2WAY-PDA} \text{ and } L(P) = \emptyset\}$$

is undecidable.

Solution 3. *This problem setup motivates us to use computation histories: we're dealing with the existence (of a string in the language of P), and we're dealing with a machine that is a weaker model of computation than a Turing Machine. In addition, since we're dealing with emptiness, we are motivated to make a machine that only accepts accepting computation histories of M on w in our reduction from A_{TM}.*

How can a $\mathsf{2WAY-PDA}$ check for an accepting computation history of M on w? Let's assume that we are given the computation history in its standard encoding of $c_1\#c_2\#\ldots\#c_k$. We want to design P to check the three conditions that we've seen before:

1. *Check that $c_1 = q_0w$. This can be easily done by hardcoding q_0w into the state control of P or (equivalently) pushing a copy of q_0w onto the stack and checking c_1 in reverse, popping the characters off the stack one-by-one.*
2. *Check that c_{i+1} follows from c_i according to the transition function of M. The issue we had with PDAs (that we dealt with in ALL_{PDA}) is that in order to compare c_i and c_{i+1}, we had to push c_i on the stack and then pop the characters off one-by-one, which meant it came out in reverse order. For the ALL_{PDA} situation this motivated us to reverse adjacent configurations in our encoding. However, we don't need to do that here: because P is a $\mathsf{2WAY-PDA}$, we can read the configuration c_{i+1} in backwards order, and then we don't need to change any encoding (alternatively, we can pop c_i onto the stack in backwards order).*
3. *Check that c_k is an accepting configuration: we check that the state in the configuration is an accepting one.*

All of this looks good, with one minor caveat. After checking the $c_i \to c_{i+1}$ transition, recall that with a PDA we had problems because we no longer had a copy of c_{i+1} to check with c_{i+2} for the next transition, and we couldn't go backwards to "re-read" c_{i+1}. For a $\mathsf{2WAY-PDA}$ we have no such problem: we simply reset the head back to the start of c_{i+1}.

This is roughly why a computation history proof can't work for E_{PDA}. In fact, recall that it's a decidable language!

Finally, we can write up precisely how the reduction works from A_{TM} in a very similar fashion to E_{LBA}:

We proceed via a reduction from A_{TM}. Assume for sake of contradiction that we have a decider for $E_{\mathsf{2WAY-PDA}}$; call it D. Then construct a decider for A_{TM} as follows:

***L**: On input $\langle M, w\rangle$:*

1. *Construct a $\mathsf{2WAY-PDA}$, $P_{M,w}$, which tests if a given computation history for M on w is accepting or not by checking each of the three conditions as described above.*
2. *Feed $\langle P_{M,w}\rangle$ into D.*
3. *If D accepts, then $L(P_{M,w})$ is empty and there is no accepting computation history of M on w, so we reject.*

4. *If D rejects, then $L(P_{M,w})$ is nonempty and there is an accepting computation history of M on w, so we accept.*