Recitation 06: Computation History Method, Recursion Theorem and Midterm Review

In today’s recitation we will review the final topics in the computability theory part of the course: the computation history method and the recursion theorem. This will include another example problem for showing undecidability via the computation history method. Both of these concepts provide new approaches to showing undecidability, as we will see later. Afterwards, we will provide some review material for the midterm, mainly reviewing the relationship between different language classes and their closure properties.

Computation History Method

The computation history method is a powerful technique for proving reductions, based on the sequence of configurations that a Turing Machine goes through on a given input. We first revisit the definition of a configuration:

**Definition 1 (TM Configuration).** A configuration of a TM refers to a particular setting of its state, head position and tape. It can be represented as a triple \((q, p, t)\) with

- \(q\) = the state
- \(p\) = the head position
- \(t\) = the tape contents

We will often encode the configuration as a string \(t_1q t_2\), where \(t = t_1 t_2\) and the head is on the first symbol of \(t_2\)

**Example 1.** Let \((q_3, 3, aaaaabbbb)\) be a TM configuration. Then, we can encode this as the string \(aaaaq_3abbbb\).

Having defined configurations, we can then define an accepting computation history of a TM \(M\) on some input \(w\) as follows:
Definition 2 (Accepting Computation History). An accepting computation history is a sequence of configurations $C_1, C_2, \cdots , C_{\text{accept}}$ which $M$ passes through until accepting. We can encode a computation history as $C_1\#C_2\#\cdots \#C_{\text{accept}}$ (where each configuration is itself encoded as a string).

It is important to note that an accepting computation history of $M$ on $w$, $C_1\#C_2\#\cdots \#C_{\text{accept}}$, must satisfy the following three properties in order to be valid:

- $C_1$ is the start configuration, which means $C_1 = q_0w_1 \cdots w_n$
- $C_{\text{accept}}$ contains $q_{\text{accept}}$ somewhere, which means $C_{\text{accept}} = t_1q_{\text{accept}}t_2$
- Each $C_{i+1}$ follows from $C_i$ according to $M$’s transition function $\delta$. Intuitively, most of $C_i$ and $C_{i+1}$ should be identical, except for the region around the head which should be consistent with $\delta$.

Computation histories can be used to show undecidability for languages that check for the existence of some object. This could be some string accepted by an automaton in the case of $E_{LBA}$ or a matching for a set of dominoes in the case of PCP. In one common scenario, we are given some computational model $X$, which may be a variation of a DFA, PDA or TM and we want to show that either $\text{ALL}_X$ or $E_X$ is undecidable. This allows us to create a reduction from $A_{TM}$ by constructing a machine in the model $X$ which then allows checking whether there exists an accepting computation history of $M$ on $w$. We look at one such example in the next section.

Example: 2DFA and Undecidability of $E_{2DFA}$

In recitation, we introduced an automaton called a 2DFA:

Definition 3. A 2DFA is a variation of a DFA with two read-only heads which can move independently (these may have been defined as bidirectional or one-directional depending on the recitation section. It does not affect the rest of the example).

We will soon use the computation history method to show that $E_{2DFA}$ is undecidable. First, to get a feeling of how this 2DFA works, we work through the following exercise:

Example 2. Let $A_{2DFA} = \{\langle B, w \rangle \mid B \text{ is a 2DFA and } B \text{ accepts } w \}$

Solution 1. We will construct a decider similarly to how we constructed the decider for $A_{LBA}$. On any input $\langle B, w \rangle$, we can simulate $B$ on $w$ a finite number of steps given that a 2DFA has a limited number of configurations, which we will call $k$. This means that after $k$ steps, if $B$ has not halted, then it is looping and will never halt.
We now compute $k$. Note that the number of configurations is equal to the number of states $|Q|$ times the number of possible head positions, which on an input $w$ is equal to $|w|^2$. This yields $k = |Q||w|^2$. We can now describe a decider $M$ for $A_{2\text{DFA}}$:

\begin{itemize}
  \item \textbf{M}: On input $(B, w)$:
    \begin{enumerate}
      \item Run $B$ on $w$ for $|Q||w|^2$ steps
      \item If $B$ accepted, then \textbf{accept}
      \item If $B$ rejected or has not halted, then \textbf{reject}
    \end{enumerate}
\end{itemize}

Now we can move on to the emptiness problem for a 2DFA:

\textbf{Example 3.} Let $E_{2\text{DFA}} = \{\langle B \rangle | B$ is a 2DFA and $L(B) = \emptyset\}$. Show that $E_{2\text{DFA}}$ is undecidable.

\textbf{Solution 2.} We start with some intuition for our approach. We will show a reduction from $A_{\text{TM}}$ to $E_{2\text{DFA}}$, by assuming we have a decider $R$ for $E_{2\text{DFA}}$ and using it to construct a decider $S$ for $A_{\text{TM}}$. Note that the inputs to $R$ have the form $\langle B \rangle$, where $B$ is a 2DFA, so we want to construct a machine $B_{M,w}$ such that running $R$ on this input tells us whether $M$ accepts $w$.

Now, running $R$ on $B_{M,w}$ will tell us whether $L(B_{M,w})$ is empty. Then, recall that $M$ only has an accepting computation history on $w$ if $M$ accepts $w$. Conversely, the set of accepting computation histories of $M$ on $w$ is nonempty if and only if $M$ accepts $w$. Thus if the language of $B_{M,w}$ consists of accepting computation histories of $M$ on $w$, then the decider $R$ would tell us whether or not such a computation history exists. In turn, this would allow us to decide $A_{\text{TM}}$. Based on this reasoning, we construct our decider $S$:

\begin{itemize}
  \item \textbf{S}: On input $\langle M, w \rangle$:
    \begin{enumerate}
      \item Construct the following 2DFA $B_{M,w}$:
        \begin{itemize}
          \item \textbf{B}_{M,w}: On input $x$:
            \begin{enumerate}
              \item Use one head to check if $C_1$ is a valid start state of $M$ on $w$.
              \item For every $i$, place one head at the start of $C_i$ and the other at the start of $C_{i+1}$. Alternate moving each head forward to check that $C_{i+1}$ follows from $C_i$ according to $M$’s transition function.
              \item Once one head reaches the end of the input, use the second head to check that $C_{\text{accept}}$ contains $q_{\text{acc}}$.
            \end{enumerate}
        \end{itemize}
      \item Run $R$ on input $\langle B_{M,w} \rangle$
      \item If $R$ rejects, \textbf{accept}.
      \item If $R$ accepts, \textbf{reject}
    \end{enumerate}
\end{itemize}

This reduction from $A_{\text{TM}}$ to $E_{2\text{DFA}}$ shows that $E_{2\text{DFA}}$ is undecidable.
There are a couple of interesting takeaways from both this example and the LBA example from lecture. One is that even if a computational model is not powerful enough to simulate a TM, it may be powerful enough to check for an accepting computation history of a TM. This would then imply that some problem relating to the languages of these machines is undecidable, since we can construct a machine that only accepts/rejects valid computation histories. The other is that one can think of simulating a TM as producing a computation history, so this tells us that it is easier to check whether a candidate computation history is valid than to actually create one.

Recursion Theorem

In lecture, we talked about the idea of self-replicating Turing Machines, and how this leads to the result that a Turing Machine can access its own description. We did not cover the recursion theorem extensively in recitation. Many problems that can be solved using the Recursion Theorem can also be solved using general reductions, but using this theorem can often shorten/simplify the proofs. Recall that we saw an example in lecture which was an alternative proof for the undecidability of $A_{TM}$:

**Theorem 1.** $A_{TM}$ is undecidable

**Proof.** For the sake of contradiction, assume there exists a decider $H$ for $A_{TM}$. Then, we can construct the following TM $R$ for which $H$'s computation is incorrect:

\[
\begin{array}{l}
R: \text{On input } w: \\
1. \text{Get own description } \langle R \rangle \text{ via the Recursion Theorem} \\
2. \text{Run } H \text{ on input } \langle R, w \rangle \\
3. \text{If } H \text{ accepts, then reject} \\
    \quad \text{If } H \text{ rejects, then accept}
\end{array}
\]

Thus, $H$ does not decide $A_{TM}$ and we have a contradiction.

Another example of the recursion theorem can be found in the textbook, and it is the problem of showing that determining whether a TM $M$ is the TM with the shortest description $\langle M \rangle$ recognizing $L(M)$.

**Definition 4.** A TM $M$ is called **minimal** if no TM with a shorter description can recognize $L(M)$. Let $MIN_{TM} = \{ \langle M \rangle | M \text{ is a minimal TM} \}$.

**Theorem 2.** $MIN_{TM}$ is not Turing-Recognizable
Proof. Recall that a language is T-Recognizable if and only if it is T-Enumerable. Then, assume for the sake of contradiction that there is an enumerator $E$ for $MIN_{TM}$.

We then construct the following TM $R$:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>R:</strong> On input $w$:</td>
<td></td>
</tr>
<tr>
<td>1. Get own description $\langle R \rangle$ via the Recursion Theorem.</td>
<td></td>
</tr>
<tr>
<td>2. Run $E$ until we get some $\langle B \rangle$ with $</td>
<td>\langle R \rangle</td>
</tr>
<tr>
<td>3. Run $B$ on $w$</td>
<td></td>
</tr>
<tr>
<td>4. If $B$ accepts, then <strong>accept</strong></td>
<td></td>
</tr>
<tr>
<td>If $B$ rejects, then <strong>reject</strong>.</td>
<td></td>
</tr>
</tbody>
</table>

This means that $B$ is not a minimal TM, so $E$ is not an enumerator for $MIN_{TM}$ and we have reached a contradiction.

These examples show how the recursion theorem can be used to show undecidability and unrecognizability via proof by contradiction.
**Midterm Review**

We start with the following Venn Diagram of Language Classes, which is a useful reference to understand the relationship between Regular, Context-Free, Turing-Recognizable and Turing-Decidable languages. Some closure properties, which we will put in a table later, can also be understood from this.

Many of these languages are useful when proving a language is decidable, as well as constructing general reductions and mapping reductions, as we will see in some tips and tricks given later in this section.

Next we move on to the following table with closure properties. In it, each entry either indicates that the language is closed under said property, or a counterexample from either lecture or past Problem Sets.

<table>
<thead>
<tr>
<th>Class</th>
<th>Union (∪)</th>
<th>Intersection (∩)</th>
<th>Concat (⊙)</th>
<th>Star (*)</th>
<th>Complement (L)</th>
<th>Reversal (L^R)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regular</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>CFL</td>
<td>Yes</td>
<td>PSet2</td>
<td>Yes</td>
<td>Yes</td>
<td>A_TM</td>
<td>Yes</td>
</tr>
<tr>
<td>T-Recognizable</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Decidable</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>A_TM</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Another important closure property is that $CFL \cap REG = CFL$. In recitation, we quickly reviewed the arguments for why these closure properties hold, and we recommend it as a review exercise that you do this yourself. This may help you find some knowledge gaps you may have with each computation model.

Finally, here is a quick overview of the techniques we have studied so far to solve different problems, as well as tips and tricks for each

Note that these tricks are subjective and largely my own intuition for approaching these problems.
Let’s first review the general approach we take when showing a given language \( L \) belongs to a certain class:

- **Showing Regular:** The easiest solution is often to construct an NFA recognizing \( L \), although closure properties can come in handy as well. The NFA allows you to use nondeterminism, which will often simplify your solution.
  
  Conversely, if a problem tells you that a language \( A \) is regular, it is usually more helpful to take a DFA \( M \) for this language. The determinism allows you to have more guarantees about \( M \)’s computation, which you can then use to construct another automaton.

- **Showing Context-Free:** Approaching these problems using PDAs tends to be easier than using CFGs, since PDAs allow you to think in terms of computation rather than trying to construct a grammar for \( L \). The PDA’s stack is often used for counting/comparing certain amounts, and the nondeterminism can be used to make necessary guesses.
  
  If a problem tells you that some language \( A \) is a CFL, there are situations where taking the CFG \( G \) for \( A \) will yield a simpler solution than using a PDA \( P \) that recognizes \( A \).

- **Showing Turing-Recognizable:** Here, you can construct a Turing Machine \( M \) and you only have to argue why it always halts for inputs \( x \in L \), and that any \( x \) not in \( L \) is not accepted. There is no need to worry about whether \( M \) halts on inputs not in the language.

- **Showing Decidable:** One approach is to construct a Turing Machine \( D \) and argue why it halts on inputs that are both in the language and not in the language. As we saw in PSet 2 Problem 5, one way to guarantee this is to somehow upper-bound the number of steps for which we run \( D \) on its input.
  
  If \( L \) is a language about DFAs, NFAs, CFGs or PDAs, it can also be useful to use a decider for languages we already proved decidable (found in the Venn Diagram above), including \( A_{DFA}, E_{DFA}, EQ_{DFA}, A_{CFG}, E_{CFG} \). This will be similar to a reduction, where you modify your input to get something that can be passed into the decider for one of these languages, and it will often make it much easier to argue that your decider \( D \) halts on all inputs.

We finish off this review by reviewing some tips for showing that a language \( L \) **does not** belong to one of the classes we have studied:

- **Showing Non-Regular:** This is most commonly done using the Pumping Lemma, where we assume the language is regular and
thus has a pumping length $p$. We then show that there is some string $s \in L$ that violates the lemma. Remember that you only need to give one string $s$, constructed for some general $p$, but you need to argue that there is no way to split it up into $s = xyz$ such that $xy^i z \in L$ for every $i$. In other words, you have to argue that no matter how the string gets cut up, there is some $i$ for which $xy^i z \notin L$. Sometimes it will be easier to “pump up” ($i > 1$) and others to “pump down” ($i = 0$).

Sometimes, it is possible to use closure properties to show that if $L$ is regular then some other $L'$ (which is known to be non-regular) is also regular, which gives a contradiction.

- **Showing Non-CFL**: The Context-Free Pumping Lemma can be used here, similarly to the Regular case. Nonetheless, it often happens that the Context-Free version requires some more case work, mainly because there are more ways to create the $s = uvxyz$ partition. Look out for cases in which this casework can be simplified by using closure properties, or the fact that $CFL \cap REG = CFL$. More specifically, if you want to show $L$ is not a CFL, you can construct a language $L'$ by taking the intersection of $L$ with a regular language. Then you use the Pumping Lemma to show $L'$ is not a CFL, which implies that $L$ isn’t either.

- **Showing Undecidable**: The approach we have used most frequently for showing undecidability is reducing $A_{TM}$ to $L$. We studied two main ways to do this:
  
  - Assume there exists a decider $R$ for $L$, use it to construct a decider $S$ for $A_{TM}$.
  - Use the Computation History Method. To determine if $\langle M, w \rangle \in A_{TM}$, assume there is a decider $R$ for $L$, and use it to determine whether there exists an accepting computation history of $M$ on $w$.

  We can also use the recursion theorem to reach a contradiction, as seen in an earlier proof of $A_{TM}$.

- **Showing Turing-Unrecognizable**: The most common approach we studied was giving a mapping reduction from $\overline{A_{TM}}$ to $L$, namely showing that $A_{TM} \leq_M L$. It suffices to construct a computable function $f$ such that $x \in \overline{A_{TM}} \iff f(x) \in L$.

  This concludes our review of problem-solving techniques and also our midterm review. Note that reading through these is not enough practice to succeed in the exam, and we strongly recommend you try the practice problems to get a feeling of what concepts are less clear to you.