Undecidability by Rice Theorem

$\mathcal{RE}$-Completeness

Reductions by computational histories

Reflections on computability portion of the course

Sipser’s book, Chapter 5, Sections 5.1, 5.3
Rice’s Theorem

**Theorem:** Suppose $C$ is a proper, non-empty subset of the set of enumerable languages, $\mathcal{RE}$, then it is undecidable whether for a given encoding of a TM, $\langle M \rangle$, $L(M)$ is in $C$.

(See problem 5.22 in Sipser’s book)
Rice’s Theorem (Restated)

**Theorem:** Let $C$ be a proper non-empty subset of the set of enumerable languages. Denote by $L_C$ the set of all TMs encodings, $\langle M \rangle$, such that $L(M)$ is in $C$. Then $L_C$ is undecidable.

**Proof:** by reduction from $A_{TM}$.

Given $M$ and $w$, we will construct $M_0$ such that:
- If $M$ accepts $w$, then $\langle M_0 \rangle \in L_C$.
- If $M$ does not accept $w$, then $\langle M_0 \rangle \notin L_C$. 
Proof of Rice’s Theorem

Without loss of generality, $\emptyset \not\in C$.

(Otherwise, look at $\overline{C} = \mathcal{RE} \setminus C$, also proper and non-empty.)

Since $C$ is not empty, there exists some language $L \in C$. Let $M_L$ be a TM accepting this language (recall $C \subset \mathcal{RE}$ contains only recursively enumerable languages).

continued ...
Proof of Rice’s Theorem (cont.)

Given $\langle M, w \rangle$, construct $M_0$ such that:

- If $M$ accepts $w$, then $L(M_0) = L \in C$.
- If $M$ does not accept $w$, then $L(M_0) = \emptyset \notin C$.

$M_0$ on input $y$:

1. Run $M$ on $w$.
2. If $M$ accepts $w$, run $M_L$ on $y$.
   a. if $M_L$ accepts, accept, and
   b. if $M_L$ rejects, reject.

Claim: The transformation $\langle M, w \rangle \rightarrow \langle M_0 \rangle$ is a mapping reduction from $A_{TM}$ to $L_C$. 
Proof of Rice’s Theorem (cont.²)

Proof: $M_0$ on input $y$:

1. Run $M$ on $w$.
2. If $M$ accepts, run $M_L$ on $y$.
   a. if $M_L$ accepts, accept, and
   b. if $M_L$ rejects, reject.

The machine $M_0$ is simply a concatenation of two known TMs – the universal machine, and $M_L$.

Therefore the transformation $\langle M, w \rangle \rightarrow \langle M_0 \rangle$ is a computable function, defined for all strings in $\Sigma^*$.

(hey – what do we actually do with strings not of the form $\langle M, w \rangle$?)
Rice’s Proof (Concluded)

- If \( \langle M, w \rangle \in A_{TM} \) then \( M_0 \) gets to step 2, and runs \( M_L \) on \( y \).
- In this case, \( L(M_0) = L \), so \( L(M_0) \in C \).
- On the other hand, if \( \langle M, w \rangle \notin A_{TM} \) then \( M_0 \) never gets to step 2.
- In this case, \( L(M_0) = \emptyset \), so \( L(M_0) \notin C \).
- This establishes the fact that \( \langle M, w \rangle \in A_{TM} \) iff \( \langle M_0 \rangle \in L_C \). So we have \( A_{TM} \leq_m L_C \), thus \( L_C \) is undecidable.
Rice’s Theorem (Reflections)

- Rice’s theorem can be used to show undecidability of properties like
  - “does $L(M)$ contain infinitely many primes”
  - “does $L(M)$ contains a prime number”
  - “is $L(M)$ empty”

- Decidability of properties related to the encoding itself cannot be inferred from Rice. For example “does $\langle M \rangle$ has an even number of states” is decidable.

- Properties like “does $M$ reaches state $q_6$ on the empty input string” are undecidable, but this does not follow from Rice’s theorem.

- Rice does not say anything on membership in $\overline{\mathbb{RE}}$ of languages like “is $L(M)$ finite”.
Consider the language $L_{\text{infinite}} = \{ \langle M \rangle \mid L(M) \text{ is infinite} \}$.
By Rice Theorem, this language is not in $\mathcal{R}$.
We want to show that $L_{\text{infinite}} \notin RE$.

**Idea:** Reduction from $H_{\text{TM}}$.
So we are after a reduction $f : \langle M, w \rangle \mapsto \langle M_0 \rangle$ such that
- If $M$ halts on $w$ then $L(M_0)$ is finite.
- If $M$ does not halts on $w$ then $L(M_0)$ is infinite.

This looks a bit tricky... Shown in recitation using controlled executions.
**$\mathcal{RE}$-Completeness**

**Question:** Is there a language $L$ that is **hardest** in the class $\mathcal{RE}$ of enumerable languages (languages accepted by some TM)?

**Answer:** Well, you have to **define** what you mean by “hardest language”.

**Definition:** A language $L_0 \subseteq \Sigma^*$ is called $\mathcal{RE}$-complete if the following holds

- $L_0 \in \mathcal{RE}$ (membership).
- For every $L \in \mathcal{RE}$, $L \leq_m L_0$ (hardness).
\textbf{RE-Completeness}

\textbf{Definition} A language $L_0 \subseteq \Sigma^*$ is called \textit{RE-complete} if the following holds

- $L_0 \in \mathcal{RE}$ (membership).
- For every $L \in \mathcal{RE}$, $L \leq_m L_0$ (hardness).

The second item means that for every enumerable $L$ there is a mapping reduction $f_L$ from $L$ to $L_0$. The reduction $f_L$ depends on $L$ and will typically differ from one language to another.
**RE-Completeness**

**Question:** Having defined a reasonable notion, we should make sure it is not vacuous, namely verify there is at least one language satisfying it.

**Theorem** The language $A_{TM}$ is $RE$-Complete.

**Proof:**

- The universal machine $U$ accepts the language $A_{TM}$, so $A_{TM} \in RE$.
- Suppose $L$ is in $RE$, and let $M_L$ be a TM accepting it. Then $f_L(w) = \langle M_L, w \rangle$ is a mapping reduction from $L$ to $A_{TM}$ (why?).
Reduction via Computation Histories

Important technique for proving undecidability.

- Useful for testing existence of some objects.
- For example, basis for proof of undecidability in Hilbert’s tenth problem,
- where "object" is integral root of polynomial.
- Other examples: Does a linear bounded TM accept the empty language?
- Does a context free grammar generate $\Sigma^*$?
Reminder: Configurations

Configuration:

\[ 1011_{q7}0111 \]

means:

- state is \( q_7 \)
- LHS of tape is 1011
- RHS of tape is 0111
- head is on RHS 0
Configurations

- configuration \( uaq_i bv \) yields \( uq_j acv \) if \( \delta(q_i, b) = (q_j, c, L) \)
- Of course, \( uaq_i bv \) yields \( uacq_j v \) if \( \delta(q_i, b) = (q_j, c, R) \)
- Special case (left end of tape): \( q_i bv \) yields \( q_j cv \) if \( \delta(q_i, b) = (q_j, c, L) \).
Computation Histories

Let $M$ be a TM and $w$ an input string.

- An **accepting** computation history for $M$ on $w$ is a sequence $C_1, C_2, \ldots, C_\ell$, where
  - $C_1$ is the starting configuration of $M$ on $w$
  - $C_\ell$ is an accepting configuration of $M$,
  - each $C_i$ yields $C_{i+1}$ according to the transition function.

- A **rejecting** computation history for $M$ on $w$ is the same, except
  - $C_\ell$ is a rejecting configuration of $M$. 
Remarks

- Computation sequences are finite.
- If $M$ does not halt on $w$, no accepting or rejecting computation history exists.
- Notion is useful for both deterministic (one history) and non-deterministic (many histories) TMs.
A CFG Question

SENTENCE

NOUN-PHRASE

ARTICLE

a

NOUN

boy

VERB

sees
Emptiness of CFGs

We have already seen an algorithm to check whether a context-free grammar is empty.

On input $\langle G \rangle$ where $G$ is a CFG:

1. Mark all terminal symbols in $G$.
2. Repeat until no new variables become marked:
   3. Mark any $A$ where
      $$A \rightarrow U_1 U_2 \ldots U_k$$
      and each $U_i$ has already been marked.
4. If start symbol marked, accept, otherwise reject.
Using Computation Histories for CFGs

So the language $\text{EMPTY}_{\text{CFG}}$ is decidable.

**Question:** What about the complementary question: Does a CFG generate all strings?

$$\text{All}_{\text{CFG}} = \{ \langle G \rangle | G \text{ is a CFL and } L(G) = \Sigma^* \}$$
Theorem: $\text{All}_{\text{CFG}}$ is undecidable.

Proof by reduction from $\text{A}_{\text{TM}}$ to $\text{All}_{\text{CFG}}$:

- Given $\langle M, w \rangle$, construct a coding of a CFG, $\langle G \rangle$
- $G$ generates all strings that are not accepting computation histories for $M$ on $w$
- If $M$ does not accept $w$, $G$ generates all strings
- If $M$ does accept $w$, $G$ does not generate the accepting computation history.
Does a CFG Generate All Strings?

An accepting computation history appears as 
\#C_1\#C_2\# \ldots \#C_\ell\#, where

- \( C_1 \) is the starting configuration of \( M \) on \( w \),
- \( C_\ell \) is an accepting configuration of \( M \),
- Each \( C_i \) yields \( C_{i+1} \) by transition function of \( M \).

A string is not an accepting computation history if it fails one or more of these conditions.
Does a CFG Generate All Strings?

Instead of the CFG, \( G \), we construct a PDA, \( D \) (recall equivalence).

\( D \) non-deterministically “guesses” which condition is violated.

- then verifies the guessed violation:
  - Is there some \( C_i \) that is not a configuration of \( M \) (number of \( q \) symbols \( \neq 1 \))?
  - Is \( C_1 \) not the starting configuration of \( M \) on \( w \)?
  - Is \( C_\ell \) not an accepting configuration of \( M \)?
  - Does \( C_i \) not yield \( C_{i+1} \) by the transition function of \( M \)?

- The last condition is the tricky one to check.
Does a CFG Generate All Strings?

- Does $C_i$ not yield $C_{i+1}$?

Idea:
- Scan input. Nondeterministically decide "violating configuration" $C_i$ was reached.
- Push $C_i$ onto the stack till $\#$.
- scan $C_{i+1}$ and pop matching symbols of $C_i$
  - check if $C_i$ and $C_{i+1}$ match everywhere, except . . .
  - around the head position,
  - where difference dictated by transition function for $M$. 

Slides modified by Benny Chor, based on original slides by Maurice Herlihy, Brown University.
When $D$ pops $C_i$ from stack, $C_i$ is in reverse order. Ignoring the local changes around head position, what we were trying to identify the language $x \# y$, with $x \neq y$.

While this can be done in principle by a non deterministic PDA (see problem 2.26 in Sipser’s book), there is a simpler way.

So far, we used a “straight” notion of accepting computation histories

\[
\begin{align*}
\# & \quad \rightarrow \quad \# & \quad \rightarrow \quad \# & \quad \rightarrow \quad \# & \quad \rightarrow \quad \# & \quad \cdots \quad \# & \quad \rightarrow \quad \# \\
&C_1 & & C_2 & & C_3 & & C_4 & & C_\ell
\end{align*}
\]
Does a CFG Generate All Strings?

So far, we used a “straight” notion of accepting computation histories

\[
\# \rightarrow \# \rightarrow \# \rightarrow \# \rightarrow \# \cdots \# \rightarrow \#
\]

But in this modern age, why not employ an alternative notion of accepting computation history, one that will make the life of our PDA much easier? **Solution:** Write the accepting computation history so that every other configuration is in reverse order.

\[
\# \rightarrow \# \leftarrow \# \rightarrow \# \leftarrow \# \cdots \# \leftarrow \#
\]

This takes care of difficulty in the proof.
Wrapping Things Up

Given $\langle M, w \rangle$, we constructed (algorithmically) a PDA, $D$, which rejects the string $z$ if and only if $z$ equals an accepting computation history of $M$ on $w$, written in the "alternating format".

Therefore $L(D)$ is either $\Sigma^*$ or $\Sigma^* \setminus \{z\}$.

This $D$ has an equivalent (and efficiently described) CFG, $G$, namely $L(D) = L(G)$. So $L(G)$ is either $\Sigma^*$ or $\Sigma^* \setminus \{z\}$. The mapping $\langle M, w \rangle \mapsto \langle G \rangle$ is thus a reduction from $A_{TM}$ to $\text{All CFG}$.

Since $A_{TM} \not\in R$ we get $\text{All CFG} \not\in R$.

As the class $R$ is closed under complement, we conclude that $\text{All CFG} \not\in R$. ♠
Linear Bounded Automata

- A restricted form of TM.
- Cannot move off portion of tape containing input
- Rejects attempts to move head beyond input
- Size of input determines size of memory

![Diagram showing a Turing machine and a linear bounded automaton]
Linear Bounded Automata

**Question:** Why linear?

**Answer:** Using a tape alphabet larger than the input alphabet increases memory by a constant factor.
Linear Bounded Automata

Believe it or not, LBAs are quite powerful. The deciders for

- $A_{DFA}$ (does a DFA accept a string?)
- $A_{CFG}$ (is string in a CFG?)
- $\text{EMPTY}_{DFA}$ (is a DFA trivial?)
- $\text{EMPTY}_{CFG}$ (is a CFL empty?)

are all LBAs.

Every CFL can be decided by a LBA.

Not too easy to find a natural, decidable language that cannot be decided by an LBA.
Acceptance for LBAs

Define

\[ A_{\text{LBA}} = \{ \langle M, w \rangle | M \text{ is an LBA that accepts } w \} \]

**Question:** Is \( A_{\text{LBA}} \) decidable?

**Answer:** Unlike \( A_{\text{TM}} \), the language \( A_{\text{LBA}} \) is decidable!
Lemma:

Let $M$ be a LBA with

- $q$ states
- $g$ symbols in tape alphabet

On an input of size $n$, LBA has exactly $qng^n$ distinct configurations, because a configuration involves:

- control state ($q$ possibilities)
- head position ($n$ possibilities)
- tape contents ($g^n$ possibilities)
Theorem: $A_{\text{LBA}}$ is decidable

Idea:

- Simulate $M$ on $w$ (if $M$ tries to “trespass” the leftmost blank, halt and reject).

- But what do we do if $M$ loops?

- Must detect looping and reject.

- $M$ loops if and only if it repeats a configuration.

- Why? And is this also true of “regular” TMs?

- By pigeon hole, if our LBA $M$ runs long enough, it must repeat a configuration!
Theorem: $A_{\text{LBA}}$ is decidable

On input $\langle M, w \rangle$, where $M$ is an LBA and $w \in \Sigma^*$,

1. Simulate $M$ on $w$,

2. while maintaining a counter.

3. Counter incremented by 1 per each simulated step (of $M$).

4. Keep simulating $M$ for $qng^m$ steps, or until it halts (whichever comes first)

5. If $M$ has halted, accept $w$ if it was accepted by $M$, and reject $w$ if it was rejected by $M$.

6. reject $w$ if counter limit reached ($M$ has not halted).
More LBAs

Surprisingly though, LBAs do have undecidable problems too!

Here is a related problem.

\[ \text{Non-EMPTY}_{LBA} = \{ \langle M \rangle | M \text{ is an LBA and } L(M) \neq \emptyset \} \]

**Question:** Is \( \text{Non-EMPTY}_{LBA} \) decidable?
Non-EMPTY_{LBA} = \{ \langle M \rangle | M \text{ is an LBA and } L(M) \neq \emptyset \}

Theorem: Non-EMPTY_{LBA} is undecidable.

Proof by reduction from \( A_{TM} \), using computation histories.
More LBAs

Given $M$ and $w$, we will construct an LBA, $B$.

- $L(B)$ will contain exactly all accepting computation histories for $M$ on $w$.
- $M$ accepts $w$ iff $L(B) \neq \emptyset$. 
More LBAs

It is not enough to show that $B$ exists.

We must show that the mapping from $\langle M, w \rangle$ to $\langle B \rangle$ is computable.

We are now going to describe the linear bounded machine, $\langle B \rangle$. It will be clear that indeed $\langle B \rangle$ is computable from $\langle M, w \rangle$.

Assume an accepting computation history is presented as a string:

$$\# \ C_1 \ # \ C_2 \ # \ C_3 \ # \cdots \ # \ C_\ell \ #,$$

with descriptions of configurations separated by $#$ delimiters.
The LBA

The LBA, $B$, works as follows:

On input $x$, the LBA $B$:

- breaks $x$ according to the # delimiters
- identifies strings $C_1, C_2, \ldots, C_\ell$.
- then checks that all the following conditions hold:
  - Each $C_i$ are a configuration of $M$
  - $C_1$ is the start configuration of $M$ on $w$
  - Every $C_{i+1}$ follows from $C_i$ according to $M$
  - $C_\ell$ is an accepting configuration
The LBA

- Checking that each \( C_i \) is a configuration of \( M \) is easy: All it means is that \( C_i \) includes exactly one \( q \) symbols.

- Checking that \( C_1 \) is the start configuration on \( w \), \( q_0 w_1 w_2 \cdots w_n \), is easy, because the string \( w \) is “wired into” \( B \).

- Checking that \( C_\ell \) is an accepting configuration is easy, because \( C_\ell \) must include the accepting state \( q_a \).

- The only hard part is checking that each \( C_{i+1} \) follows from \( C_i \) by \( M \)’s transition function.
The Hard Part

Checking that for all $i$, $C_{i+1}$ follows from $C_i$ by $M$’s transition function:

- $C_i$ and $C_{i+1}$ almost identical, except for positions under head and adjacent to head.

- These positions should updated according to transition function.

Do this verification by

- zig-zagging between corresponding positions of $C_i$ and $C_{i+1}$.
- use “dots” on tape to mark current position
- all this can be done inside space allocated by input $x$. Thus $B$ is indeed a LBA.
The LBA, $B$, accepts the string $x$ if and only if $x$ equals an accepting computation history of $M$ on $w$. Therefore $L(B)$ is either empty or a singleton $\{x\}$.

We construct $B$ so that $L(B)$ is non-empty iff $M$ accepts $w$. Thus $\langle M, w \rangle \in A_{TM}$ iff $\langle B \rangle \in \text{Non-EMPTY}_{LBA}$.

Namely $A_{TM} \leq_m A_{LBA}$, so $\text{Non-EMPTY}_{LBA} \notin \mathcal{R}$. ♠

BTW, is $\text{Non-EMPTY}_{LBA} \in \mathcal{RE}$?
Unrestricted Grammars

Unrestricted grammars are similar to context free ones, except left hand side of rules can be strings of variables whose lengths are greater than one.

- To non-deterministically generate a string according to a given unrestricted grammar:
  - Start with the initial symbol
  - While the string contains at least one non-terminal:
    - Find a substring that matches the LHS of some rule
    - Replace that substring with the RHS of the rule

An example of an unrestricted grammar generating the language \( \{0^n1^n2^n\} \) – on board.
Unrestricted Grammars

Let $UG$ be the set of languages that can be described by an Unrestricted Grammar:

$UG = \{ L : \exists \text{ Unrestricted Grammar } G \text{ such that } L[G] = L \}$

Claim: $UG = RE$

To Prove:
- Show $UG \subseteq RE$
- Show $RE \subseteq UG$
Given any Unrestricted Grammar $G$, we create a Turing Machine $M$ that accepts $L[G]$.

$M$ will be non-deterministic, simulating derivations of $G$.  

$UG \subseteq RE$
Given any language $L \in \mathcal{RE}$, let $M$ be a deterministic Turing Machine that accepts it. We can create an Unrestricted Grammar $G$ such that $L[G] = L$.

Grammar: Generates a string

Turing Machine: Works from string to accept state

Two formalisms work in different directions

Simulating Turing Machine with a Grammar can be difficult.
Simulating Turing Machine with a Grammar can be difficult.

Requires working backwards.

Derivations works from short, accepting configuration, to initial configuration of $M$, and finally to the bare string, $w \in \Sigma^*$. 

$RE \subseteq UG$