Decidability

Alyssa Lytle

Fall 2025

1 Review

In the previous lesson we talked about algorithms and decidability.

For motivation, we talked about the problem of finding the roots of $6x^3yz^2 + 3xy^2 - x^3 - 10$. We discussed that there is no possible *algorithm* that can be developed to determine the roots of a general polynomial. This is a strong claim, and it starts with first defining an algorithm.

In 1936, algorithms were concurrently defined in two different ways! Alonzo Church expressed them using λ-calculus and Alan Turing expressed them using Turing machines. Both representations are considered equivalent, but for this course, we'll focus on the Turing-Machine representation.

To express an algorithm using a Turing machine, we do the following:

1. Represent the problem we want to solve as a set/language.
2. Define a turing machine that *decides* it. (The "algorithm" is essentially the design of your machine.)

This also lead us to recall the difference between recognizability and decidability.

Definition: Turing Recognizable and Decidable

For a language L on machine M, It is Turing-recognizable iff it

- Accepts if the input is in L
- Rejects $l$oops forever if the input is not in L

Is is Turing-decidable iff it

- Accepts if the input is in L
- Rejects if the input is not in L

1.1 Input Notation

Let $\langle A \rangle$ represent the *string representation* of input A.

Since a Turing Machine takes string inputs, we must translate our input into a string representation.

So, for example, if we are encoding a directed graph G:

We could encode its vertices in a sequence followed by its edges in a sequence:

$\langle G \rangle = (1, 2, 3, 4)((1, 2), (2, 3), (3, 1), (1, 4))$

1.2 Decidable Problems + Regular Languages

Let's define an algorithm to determine whether a string is *accepted* by a DFA.

We do this by building a TM that decides it!

Let

$A_{DFA} = \{\langle B, w\rangle \mid B \text{ is a DFA that accepts input string } w\}$
For B's string representation, assume we have a string representation of the tuple, $B = (Q, \Sigma, \delta, s, F)$.
Let
$A_{DFA} = \{\langle B, w\rangle \mid B \text{ is a DFA that accepts input string } w\}$
High-level design of Turing Machine M:

1. Simulate B on input w - Take as input the string representations for B and w and confirm they are in proper format (otherwise reject). - Write on the tape to keep track of the changing state of B as we step through w. (*Read from the tape to find out the transitions defined in δ.) Continue until we reach the end of w.

2. If the simulation ends in an accept state in B, *accept*! If it ends in a nonaccepting state in B, *reject*! (Determine if the state is in F by reading F from the tape.)

Lemma: Deciability + Regular Languages

- $A_{DFA} = \{\langle B, w\rangle \mid B \text{ is a DFA that accepts input string } w\}$.
  A_{DFA} is a decidable language.

- $A_{NFA} = \{\langle B, w\rangle \mid B \text{ is an NFA that accepts input string } w\}$.
  A_{NFA} is a decidable language.

- $A_{REX} = \{\langle B, w\rangle \mid B \text{ is a regular expression that generates string } w\}$.
  A_{REX} is a decidable language.

What if we want to check if a DFA accepts *anything*? Is this decidable? Yes!
$E_{DFA} = \{\langle A\rangle | A \text{ is a DFA and } L(A) \neq \emptyset\}$

- General idea: We can't test all possible strings w because that could be infinite, so let's leverage the fact that the sets of states Q and transitions δ *are* finite!

- Starting with the start state, "mark" a state and follow all outgoing transitions. Mark every visited state. Repeat until either all states are marked or all transitions have been followed. If an accept state has been marked, *accept*. Otherwise, *reject*!

1.3 Decidable Problems + Regular CFLs

$E_{CFG} = \{\langle A\rangle | A \text{ is a CFG and } L(G) \neq \emptyset\}$
General idea:

- Similar to the previous example, we can't test all possible strings w, but we know we have *finite* rules.

- What does it mean to generate a string w. There has to be a mapping from a start variable to a string of all terminals.

- This algorithms works similarly to the previous one, but backwards in a way. Start on all terminal symbols, and "mark" them. Mark any variable that has a rule mapping to only marked symbols. Keep going until all variables or all rules have been marked. If the start state has been marked, *accept*. Otherwise, *reject*!

2 Undecidability

Talking about decidability, leads us to the bigger challenge of talking about undecidability. How do we *prove* that a problem is undecidable?

The main approach is to focus on one problem that we know to be undecidable which we will call A_{TM}, and if we want to prove any other problem is undecidable, we convert (or *reduce*) it to express it as an A_{TM} problem.

2.1 Input Acceptance + Halting on a Turing Machine

Previously, we talked about how we can check whether or not other automata accept a string, and how this is decidable. However, if we define:

$$A_{TM} = \{\langle M, w\rangle | M \text{ is a Turing Machine and accepts } w\}$$

A_{TM} is *NOT* decidable. (It *is* Turing-recognizable, though!)

Proof We won't prove this in-depth, but I will show you the basic reasoning of the proof. It is a proof by contradiction.

Since it's a proof by contradiction, that means we will assume A_{TM} is decidable. If A_{TM} is decidable, that means there exists a turing machine H that takes $\langle M, w\rangle$ as input and behaves such that if M accepts w, H accepts. Otherwise, H rejects.

The main contradiction is found in this idea: since every input to H maps to Accept and Reject, there should also exist a Turing Machine D that exhibits the opposite behavior (it Accepts when H Rejects and Rejects when H Accepts). We find a contradiction when we discover that no such D can exist!

This type of reasoning also leads us to the following Theorem:

Theorem:

A language is decidable iff it is Turing-recognizable *and* its complement is Turing recognizable.

3 Reducibility

The primary method used to prove that problems are unsolvable is *reducibility*.

Definition: Reduction

Given two problems A and B, a *reduction* is a way of converting problem A to problem B in such a way that a solution to B can be used to solve A. If this is the case then you can say "A is reducible to B"

Reducibility has some powerful ramifications.

If A is reducible to B and B is decidable, then that means A is decidable! Similarly, if A is reducible to B and A is undecidable, the B is undecidable. This second line of reasoning shows how we will prove the undecidability of other problems.

3.1 The Halting Problem

Another common question is: "Will the machine *halt* on this input or will it loop forever?"

This can be expressed as:

$$HALT_{TM} = \{\langle M, w\rangle | M \text{ is a Turing Machine and halts on input } w\}$$

$HALT_{TM}$ is also not Turing-decidable. Let's prove it using the idea of reducibility.

Proof:

We know that A_{TM} is undecidable. So, we need to show that A_{TM} is reducible to $HALT_{TM}$. This is a proof by contradiction in a way, because proving A_{TM} is reducible to $HALT_{TM}$ means assuming a solution exists to $HALT_{TM}$ and showing that it can be used to find a solution to A_{TM}.
In other words, assume we have a TM R that decides $HALT_{TM}$. We will use R to construct a TM S that decides A_{TM}.

$S =$ "On input $\langle M, w\rangle$:

1. Run TM R on input $\langle M, w\rangle$
2. If R rejects, *reject*.
3. If R accepts, simulate M on w until it halts.
4. If M has accepted, *accept*; if M has rejected, *reject*."

Other common problems can be shown to be undecidable in a similar way.

Theorem: Undecidable Problems

The following problems are undecidable:

- $HALT_{TM} = \{\langle M, w\rangle | M$ is a Turing Machine and halts on input $w\}$
- $E_{TM} = \{\langle M\rangle | M$ is a Turing Machine and $L(M) \neq 0\}$
- $REGULAR_{TM} = \{\langle M\rangle | M$ is a Turing Machine and $L(M)$is a regular language$\}$
- $EQ_{TM} = \{\langle M_1, M_2\rangle | M_1$ and M_2 are Turing Machines and $L(M_1) = L(M_2)\}$

3.2 Reductions via Computation Histories

We've previously talked about *configurations* on Turing machines.

Recall:

The *configuration* of the Turing Machine is the current state, tape contents, and head location. It is represented as uqv where q is the current state and uv are the current contents of the tape, with the first character of v being the current head location.

- The *start configuration* on input w would be sw.
- The *accepting configuration* is the configuration that contains the state q_{accept}
- and the *rejecting configuration* is the configuration that contains the state q_{reject}

A Turing machine M *accepts* input w if there are a sequence of configurations that begin at the start configurations and finish in an accepting configuration.

The computation history of a machine is defined in terms of these configurations.

Definition: Computation History

Let M be a Turing Machine and w an input string. An *accepting computation history* for M is a sequence of configurations $C_1, C_2, \ldots, C_l$, where C_1 is the start configuration of M on w, C_l is an accepting configuration of M, and each C_i legally follows from C_{i-1} according to the rules of M.
A *rejecting computation history* for M on w is defined similarly, except that C_l is a rejecting configuration. [Sip96]

Definition: Linear Bounded Automaton

A *linear bounded automaton* (LBA) is a type of Turing Machine with a finite memory, determined by the size of the input tape.

The problem of whether or not a string is accepted by an LBA is actually decidable!

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

A_{LBA} is decidable because, since our memory is now finite, there are a finite number of configurations possible over M. Therefore, we can bound our machine to stop after a certain number of configurations.

However, what about the question of whether or not an LBA accepts any strings?

$$E_{LBA} = \{\langle M\rangle | M \text{ is an LBA where } L(M) \neq 0\}$$

This is, in fact, undecidable.

Proof:

For this proof, we will take a similar approach as we did before. We will show A_{TM} is reducible to E_{LBA} by assuming E_{LBA} is decidable and showing how that could be used to solve A_{TM}.
So, we assume there exists a TM R, that decides E_{LBA}.
Now, let's think of an LBA that we can give as input to R that would help solve A_{TM}.
First, let's again think about configuration histories. We can reframe A_{TM} in term of configuration histories. If the set of accepting configuration histories for M on w is non-empty, then that means there's an accepting configuration on M for w. (In other words, M accepts w!)
So, if we construct an LBA B such that $L(B)$ is the set of accepting *computation histories* for M on w, then running R on B would tell us if there are any accepting configurations on M for w.
Essentially, B would work by following our definition of computation histories. It would take some input string x which is a string representation of a computation history of M over input w: $C_1, C_2, \ldots, C_l$.
It would verify that:

- C_1 is the start configuration of M on w
- C_l is an accepting configuration of M
- and each C_i legally follows from C_{i-1} according to the rules of M

R can be used to confirm whether or not B is nonempty, therefore successfully reducing A_{TM} to E_{LBA}.

References

[Sip96] Michael Sipser. Introduction to the theory of computation. *ACM Sigact News*, 27(1):27–29, 1996.