Seminar : Automata Theory

Lec 2 : Halting Problem

Jinhee Paeng

Aug 29, 2023

Seoul National University

Table of Contents

Table of Contents

Turing Machine

Goal. In this section we define the machinery called *Turing Machine*.

Definition (Turing Machine)

The *Turing Machine*, or *TM*, is written as $M = (Q, \Sigma, \Gamma, \delta, q_0, H)$ where each is :

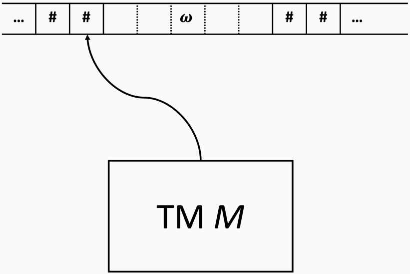
1. Q : Finite set of *States*.
2. Σ : The set of *Input Alphabets*.
3. Γ : The set of *Tape Alphabets*. Includes blank alphabet $\#$.
4. $\delta : (Q - H) \times \Gamma \to Q \times \Gamma \times \{Left, Stay, Right\}$.
5. $q_0 \in Q$: The *Initial state*.
6. $H \subseteq Q$: The *Halt states*.

Turing Machine

How TM works :

1. Initially, the input string $\omega \in \Sigma^*$ is written in *Tape*. The *blank alphabet* $\#$ is written elsewhere in the tape.
2. The Turing machine starts with *Initial state* q_0, with *Head* is positioned at the rightmost blank alphabet placed before ω.
3. After *Head* reads the alphabet on the current position, *State* changes, *Head* overwrites *Tape* and moves by δ function.
4. If current state is one of *Halt states*, the Turing machine halts.

Turing Machine

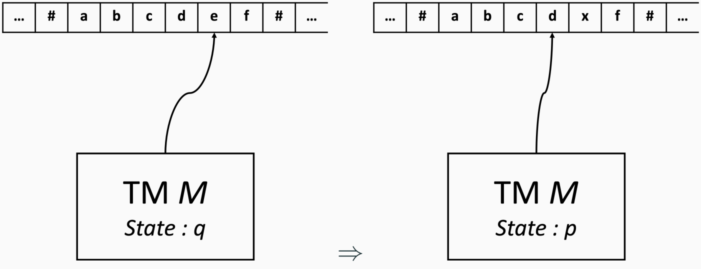


Initial starting configuration

Remark. We write current tape and head location using an underline at the head position. The diagram can be written as :

$$(q_0, \underline{\#}\omega)$$

Turing Machine



Update by δ function. It describes $\delta(q, e) = (p, x, L)$.

Remark. We can write the update in the figure as :

$$(q, \#abcd\underline{e}f) \vdash (p, \#abc\underline{d}xf)$$

Turing Machine

Goal. There are two Language classes related to the Turing machine : The *Recursively Enumerable Language* and *Recursive Language*.

Definition (Recursively Enumerable Language)

For a given Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, H)$, we define $L(M)$ as

$$L(M) = \left\{\omega \in \Sigma^* : (q_0, \underline{\#}\omega) \vdash^* (h, *), h \in H\right\}.$$

We call languages that can be defined as $L(M)$ for some TM M as *Recursively Enumerable Language*.

In other words, $L(M)$ is a set of strings that *Halts* the Turing machine M.

Turing Machine

Exmaples. Here's example of *Recursively Enumerable Language*.

- $\{0^n 1^n : n \geq 1\}$:
  Define TM $M = (\{q_0, q_1, \cdots, q_5\}, \{0, 1\}, \{0, 1, x, y, \#\}, \delta, q_0, \{q_5\})$, with the δ function as :

State	0	1	x	y	$\#$
q_0	(q_1, x, R)			(q_3, y, R)	$(q_0, \#, R)$
q_1	$(q_1, 0, R)$	(q_2, y, L)		(q_1, y, R)	
q_2	$(q_2, 0, L)$		(q_0, x, R)	(q_2, y, L)	
q_3				(q_3, y, R)	$(q_5, \#, S)$
q_4					

Remark. It replaces leftmost 0 by x, and 1 as y, by alternating one at a time. q_4 is designed to be a dead state.

Turing Machine

Definition (Recursive Language)

For a Language L, we say TM $M = (Q, \Sigma, \Gamma, \delta, q_0, \{y, n\})$ *decides* L if

- If $\omega \in L$ then, $(q_0, \underline{\#}\omega) \vdash^* (y, *)$.
- If $\omega \notin L$ then, $(q_0, \underline{\#}\omega) \vdash^* (n, *)$.

We call L as *Recursive Language* if there exist TM M that decides L.

Remark. Such TM always halts.

Remark. We sometimes refer *Recursive Language* as *Decidable*.

Theorem

If a language L is Recursive, then it is also Recursively Enumerable.

Proof.

Make n state as a dead state, looping itself infinitely. $\square$

Turing Machine

Definition (Computable function)

We say a function $f : \Sigma^* \to \Sigma^*$ is *Computable* if there exists a TM $M = (Q, \Sigma, \Gamma, \delta, q_0, \{h\})$ that satisfies :

$$(q_0, \underline{\#}\omega) \vdash^* (h, \underline{\#}f(\omega)).$$

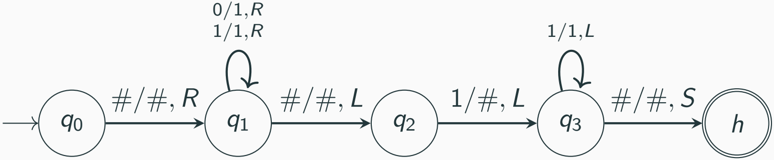
We say M *computes* f.

Remark. In this sense, we say TM as *Algorithm*.

Turing Machine

Example. *Addition* is computable. Perform $n+m$ with

$$f : 1^n 0 1^m \mapsto 1^{n+m}.$$



Remark. In this sense, we say TM as *Algorithm*.

Universal Turing Machine

Question. What is *computer*?

Observation. We can easily check that the set of Turing machines are countable. We can define a method of encoding $\langle \cdot \rangle$:

- TM M is encoded as $\langle M \rangle \in \{0,1\}^*$.
- String ω is encoded as $\langle \omega \rangle \in \{0,1\}^*$.
- Concatenation of $\langle M \rangle$ and $\langle \omega \rangle$ is denoted as $\langle M, \omega \rangle$.
- String $\langle M, \omega \rangle$ is unique among any pair of TM and string (M, ω).

Remark. Such encoding can be defined explicitly, but we omit the detail in this seminar. The idea is to write a n or n-th element as 0^n, and use 1 as a separating character.

Universal Turing Machine

Remark. With the encoding, we can have TM M as an input.

Definition (Universal Turing Machine)

A Turing machine M_u is called *Universal Turing Machine* if

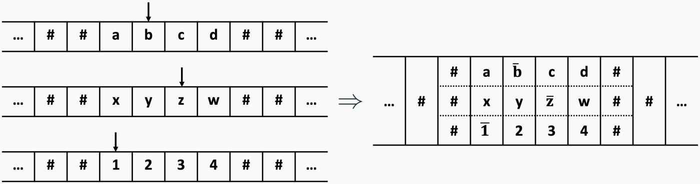
- it takes $\langle M, \omega \rangle$ as an input. It halts if an input is not in this form.
- it halts if M halts with input ω.
- it gives an output same to the output of M with ω.

Remark. We can build M_u using *3-tape TM*.

Remark. *Universal Turing Machine* is what we call a *general-purpose computer*.

Universal Turing Machine

Remark. We can build equivalent Turing machine of given *3-tape TM*.



Finding equivalent TM.

We use $\Gamma \cup (\overline{\Gamma} \cup \Gamma)^3$ as a new set of *Tape alphabet*. $\overline{\Gamma} = \{\bar{a} : a \in \Gamma\}$ indicates that *i*-th head is positioned here.

Universal Turing Machine

Definition (Turing-Completeness)

We call a computer P as *Turing-Complete* if it is equivalent to *Universal Turing Machine*. Or equivalently, a computer P that can simulate the *Universal Turing Machine* is called *Turing-Complete*.

Examples. Here are some examples of *Turing-Complete* systems.

- General-Purpose languages : C, C++, Python, R, Java, ...
- Some of other languages : Tex, Prolog, ...
- Conway's Game of Life : Simulating whole computer
- Microsoft Excel
- Some games :
  Cities: Skylines, Minecraft, Baba is you, Magic: The Gathering
- DNA and Enzyme system

Remark. Markdown language is not *Turing-Complete*.

Table of Contents

Halting Problem

Goal. In this section we cover the *Halting Problem*.

Theorem (Halting Problem)

It is impossible to build a Turing machine such that

- *takes* $\langle M, \omega \rangle$ *as an input.*
- *decides whether* $\omega \in L(M)$*, i.e.* ω *halts* M*.*
  (It halts in state y *if* $\omega \in L(M)$*, while halting in* n *if* $\omega \notin L(M)$*.)*

Remark. The proof takes a similar idea of *Set of all sets*.

Halting Problem

Before moving on to the detailed proof, let's first take observation on the *Recursively Enumerable Language* and *Recursive Language*.

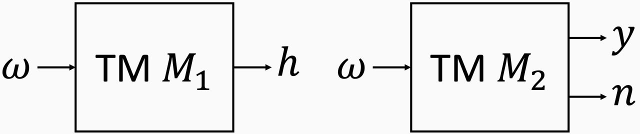


Diagram of M_1 and M_2

TM M_1 that defines *Recursively Enumerable Language* $L_1 = L(M_1)$ and TM M_2 that decides *Recursive Language* L_2 are described as above.

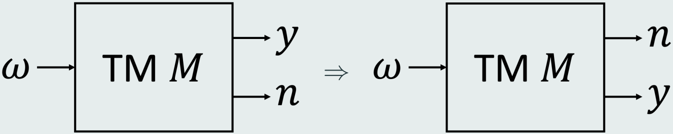
Halting Problem

Theorem

Complement of Recursive Language L, L^C, is also a Recursive Language.

Proof.

Change y, n states, and it will decide L^C.



□

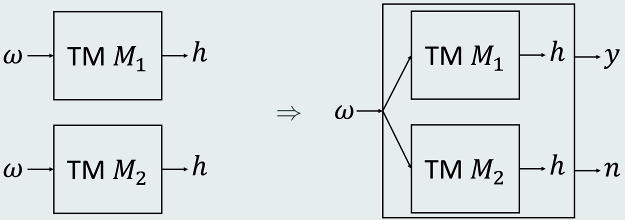
Halting Problem

Theorem

When a language L and its complement L^C are both Recursively Enumerable Language, than L is Recursive Language.

Proof.

Suppose $L = L(M_1)$ and $L^C = L(M_2)$. Then build :



It defines a new TM that decides L, making it *Recursive Language*. □

Halting Problem

Observation. Thus, we can say that a language L and it complement L^C fall into following three cases :

- L and L^C are both *Recursive Language*.
- Both L and L^C are not *Recursively Enumerable Language*.
- One of L or L^C is not *Recursively Enumerable Language*, while other is *Recursively Enumerable Language* but not *Recursive*.

Remark. We later use this observation to prove that a language corresponding to the Halting problem is not *Recursive*.

Halting Problem

Definition (L_u)

We can describe *Halting problem* as a language :

$$L_u = \{\langle M, \omega \rangle : \omega \in L(M)\}.$$

The Halting problem is equivalent to L_u being not *Recursive*.

Theorem

L_u is Recursively Enumerable Language.

Proof.

The *Universal Turing Machine* M_u defines L_u. i.e. $L_u = L(M_u)$. $\square$

Halting Problem

Definition (L_d)

A language L_d is defined as :

$$L_d = \{\langle M \rangle : \langle M \rangle \in L(M)\}.$$

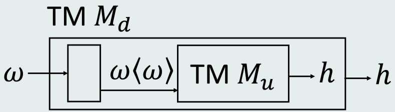
L_d is a set of encoded TM, that halts by encoded string of itself.

Theorem

L_d is Recursively Enumerable Language.

Proof.

TM M_d of the following diagram defines L_d. i.e. $L_d = L(M_d)$.



$\square$

Halting Problem

Theorem

L_d^C is not a Recursively Enumerable Language.

Proof.

Suppose L_d^C is *Recursively Enumerable*, thus $L_d^C = L(M)$ for some M.

- If $\langle M \rangle \in L_d^C$, then $\langle M \rangle \in L_d$. Thus contradiction.
- If $\langle M \rangle \notin L_d^C$, then $\langle M \rangle \notin L_d$. Thus contradiction.

$\square$

Remark. This concludes that L_d is *Recursively Enumerable* but not *Recursive*.

Remark. L_d^C is an analogous concept of $\{A : A \notin A\}$.

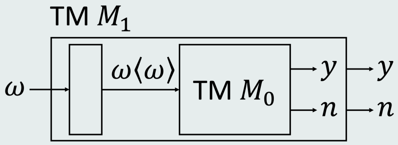
Halting Problem

Theorem

L_u is not a Recursive Language.

Proof.

Proof by contradiction : Suppose L_u is *Recursive*, and TM M_0 decides L_u. Then, it is possible to build TM M_1 that decides L_d.



However, it contradicts with L_d not being a *Recursive Language*. □

Remark. Thus, we can conclude the *Halting Problem*. It is impossible to generally answer whether the given Turing machine and input string halts.

Table of Contents

Undecidable Problems

Goal. In this section we cover some *Undecidable Problem*.

Definition (Undecidable Problem)

We call a language L *Undecidable* if there's no TM that *decides* L. For a *Undecidable* L, a question

$$\text{For a given string } \omega \text{, is } \omega \in L?$$

is called *Undecidable Problem*.

Remark. The *Halting problem* is one of the *Undecidable Problem*.

Undecidable Problems

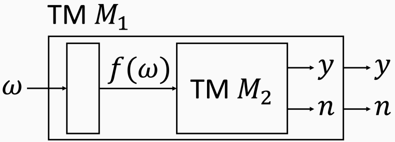
Recall. A function f is called *Computable* if there exists a TM M that always halts for any input ω and gives an output of $F(\omega)$.

Definition (Problem Reduction)

A *Reduction* of language L_1 to L_2 is a *computable* function f such that

$$\omega \in L_1 \text{ if and only if } f(\omega) \in L_2.$$

Remark. *Reduction* is useful when proving a language is *Undecidable*. When L_1 is *Undecidable*, so is L_2.



Undecidable Problems

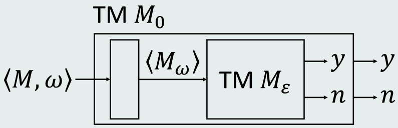
Example. "Does M halts with input ϵ?" is *Undecidable Problem*.

$$L_\epsilon = \{\langle M \rangle : \epsilon \in L(M)\}$$

Equivalently, L_ϵ is not *Recursive Language*.

Proof.

Make a *Computable function* of $\langle M, \omega \rangle \mapsto \langle M_\omega \rangle$, where M_ω is a TM that when an input ϵ is given, writes ω on the *Tape* and runs TM M.



If L_ϵ is decided by some TM M_ϵ, then M_0 of the diagram decides L_u, which contradicts to the *Halting Problem*. □

Theorem

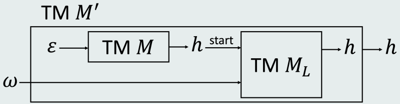
When C is a nonempty, proper subset of the collection of Recursively Enumerable Language, then the question "$L(M) \in C$?" is Undecidable.

$$L_C = \{\langle M \rangle : L(M) \in C\}$$

Equivalently, L_C is not Recursive Language.

Proof.

Without loss of generality, assume $\emptyset \notin C$. (If not, consider C^C).
Then we may choose a language $L \in C$ that $L \neq \emptyset$. Say $L = L(M_L)$.
For any TM M, define a new TM M' depending on M and M_L :



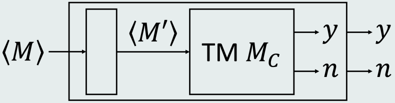
Undecidable Problems

Proof continued.

Note that $\langle M \rangle \mapsto \langle M' \rangle$ is *Computable*. Also,

- If $\langle M \rangle \in L_\epsilon$, then $L(M') = L(M_L) = L$. Thus $L(M') \in C$.
- If $\langle M \rangle \notin L_\epsilon$, then $L(M') = \emptyset$. Thus $L(M') \notin C$.

To prove by contraction, assume TM M_C decides L_C. Build a TM :



It decides whether M halts with input ϵ. Thus, it contradicts with the previous example of L_ϵ being not *Recursive Language*. $\square$

Undecidable Problems

Remark. It is possible to decide whether a given language can be defined by some Turing machine or not. This is the reason why we excluded the case where C being empty or every *Recursively Enumerable Language*.

Remark. What *Rice Theorem* implies is that it is impossible to decide whether a given language can be defined by a Turing machine when extra condition is given. For example, it is impossible to computationally check whether two given TMs define a same language.

Example. Examples of *Undecidable Problems* as corollary of *Rice Thm*.

- "Given TM M, does M not halt for any input?"
- "Given TM M, is $L(M)$ a *Regular Expression*?"

Table of Contents

Busy Beaver

Goal. It is section we cover a set of functions related to TM. We will also go through their characteristics including noncomputability.

Definition (Busy Beaver)

A function $BB : \mathbb{N} \to \mathbb{N}$, named *Busy Beaver*, is a maximum number of 1 we can write on the tape using TM with n number of states that halts with input ϵ. To be explicit, it is defined as :

$$BB(n) = \max_{\substack{\epsilon \in L(M) \\ M=(\{q_1,\cdots,q_n\},\{0,1\},\Gamma,\delta,q_1,H)}} \text{score}(M),$$

where score(M) is defined as a number of 1's in the *Tape* when M halts.

Remark. *Busy Beaver* function is a well-defined function.

Remark. Some call this function as *Radó's sigma function*, $\Sigma(n)$.

Example. $(n = 4)$, (BaBa is you ver.)

Busy Beaver

Definition (Frantic Frog)

A function $FF : \mathbb{N} \to \mathbb{N}$, named *Frantic Frog*, is a maximum number of *head shift* in TM with n number of states that halts with input ϵ. To be explicit, it is defined as :

$$BB(n) = \max_{\substack{\epsilon \in L(M) \\ M=(\{q_1,\cdots,q_n\},\{0,1\},\Gamma,\delta,q_1,H)}} \text{score}(M),$$

where score(M) is defined as a number of *head shift* until M halts.

Remark. *Frantic Frog* function is also a well-defined function.

Remark. Some call this function as *Maximum shifts function*, $S(n)$.

Busy Beaver

Application. Knowing $FF(43)$ solves *Goldbach's conjecture*.

Application. Knowing $FF(744)$ solves *Riemann Hypothesis*.

Explanation. Build a TM that searches the counterexamples. If it halts, the statement is false. If it does not halt, the statement if true. It is possible to build such TM with 43 states for *Goldbach's conjecture*, and 744 states for *Riemann Hypothesis*.

Explicit form. github link

Busy Beaver

Observation. $BB(n)$ and $FF(n)$ are both increasing functions.

Observation. $BB(n) \leq FF(n)$ since writing one 1 takes one shift.

Theorem

$BB(n)$ and $FF(n)$ is asymptotically faster than any other computable functions.

$$\lim_{n\to\infty} \frac{BB(n)}{f(n)} = \infty, \lim_{n\to\infty} \frac{FF(n)}{f(n)} = \infty$$

for any computable function f.

Remark. This result directly implies that both $FF(n)$ and $BB(n)$ are noncomputable.

Busy Beaver

Proof.

Suppose there exists a computable function f such that $FF(n) \leq f(n)$. Then, we can build a TM that decides L_ϵ as :

1. When an input $\langle M \rangle$ is given, get the number of states n.
2. Compute $f(n)$.
3. Run M with a blank input, while counting the *Head shift*.
4. If *Head shift* count exceeds $f(n)$, conclude $\langle M \rangle \notin L_\epsilon$.
   If it halts, conclude $\langle M \rangle \in L_\epsilon$.

$\square$

Busy Beaver

Known Results. Very few values are currently known.

- $BB(1) = 1$ and $FF(1) = 1$
- $BB(2) = 4$ and $FF(2) = 6$
- $BB(3) = 6$ and $FF(3) = 21$
- $BB(4) = 13$ and $FF(4) = 107$

Remark. The TM that obtains maxima in FF and BB are not identical.

Question. Why is it so hard to compute?
Answer. There's no general method to compute $BB(n)$ or $FF(n)$. Only way is to check every possible TMs, and we have to prove whether each TM halts. Due to the *Halting Problem*, it is impossible to build an algorithm to tell that a given algorithm halts or not. Also, number of TM grows exponentially, ex) 6975757441 when $n = 4$.

Busy Beaver

Theorem

It is impossible to calculate the value of $FF(748)$ in ZFC. Even providing an upper bound is impossible.

Proof.

1. It is possible to build a TM with 748 states that halts if and only if *ZFC* is inconsistent.
2. Knowing $FF(748)$ can prove or disprove inconsistency of *ZFC*.
3. However, *The Incompleteness Theorem* by *Kurt Gödel* states that it is impossible to prove system's consistency within the system.

□

Busy Beaver

Remark. Such statements foreshadows that getting an upper bound for $FF(744)$ is much harder than solving *Riemann Hypothesis*.

Remark. Note that even if you manage to get an upper bound of $FF(744)$, keep in mind that it actually take *eons* to compute until the iteration reaches $FF(744)$.

Remark. Proving that the TM (that proves *Goldbach's conjecture* or *Riemann Hypothesis*) does not halt is much easier problem. It is actually a subproblem that need to be solved to get $FF(43)$ or $FF(744)$.

Next week preview

Preview : Next week, I hope to cover topics :

- Algorithm classes including $P, NP, NP - Complete$.
- Proof of Hilbert's 10th problem :
  Given a Diophantine equation with any number of unknown quantities and with rational integral numerical coefficients: To devise a process according to which it can be determined in a finite number of operations whether the equation is solvable in rational integers.