Department of Computer Science and Engineering
PES University, Bangalore, India

UE23CS243A: Automata Formal Language and Logic

Prakash C O, Associate Professor, Department of CSE

Definition of a decidable language.

A language L is called decidable if there exists an algorithm (or, equivalently, a Turing machine) that:

- given a word,
- returns "yes" or "no" depending on whether this word belongs to this language or not.

Decidable language -A decision problem P is said to be decidable (i.e., have an algorithm) if the language L of all yes instances to P is decidable.

Examples

1. (Acceptance problem for DFA) Given a DFA does it accept a given word?
2. (Emptiness problem for DFA) Given a DFA does it accept any word?
3. (Equivalence problem for DFA) Given two DFAs, do they accept the same language?

Note: Not all languages are decidable.

Halting Problem: The Unsolvable Problem

The halting problem can never be solved algorithmically.

Why the Halting Problem Matters in Theory of Computation

- The Halting Problem has an extensive scope and **plays a crucial role in understanding the limitations of computation**.
- It sets the boundary for what computers can and cannot solve and has substantial implications for multiple areas of study, from Artificial Intelligence to Cybersecurity.

Alan Turing and the Halting Problem

- Alan Turing, often referred to as the father of theoretical computer science and artificial intelligence, made significant contributions towards solving the Halting Problem.

- Turing's efforts were instrumental in proving that **a general algorithm that solves the Halting problem for all possible program-input pairs cannot exist.**
  As such, he demonstrated the restrictions of computers, thus establishing a limitation on the power of mechanical computation.

Definitions of Halting Problem

There are many equivalent definitions of Halting Problem:

- Given a TM M and string w, does M halt on w?. Note that M doesn't have to accept w; it just has to halt on w. As a formal language: **$HALT_{TM}$ = { ⟨M, w⟩ | M is a TM and M halts on w. }**

- The Halting Problem, in the simplest terms, is a statement about computational processes in computer science. **Is it possible to write a program that determines whether another program will halt or run indefinitely?**

- **Suppose we have a Turing machine that never halts. Can we make a Turing machine that can detect this? In other words, can we make an infinite loop detector?**

- **Does a TM, say H, exist which takes as its input description of another TM M and a string w and says if M halts on w or not (goes in an infinite loop)?**
  Going by the Church Turing thesis, a computer program/ algorithm is a set of steps that can be executed on a corresponding TM.

- So, this question is same as asking: **can we have a general program/algorithm to solve the halting problem for all possible program-input pairs** (the halting problem is the problem of determining, from a description of an arbitrary computer program and an input, whether the program will finish running, or continue to run forever)?

Undecidability of Languages

1) Prove that Language $HALT_{TM}$ = { <M, w> | M is a TM and M halts on input w } is undecidable

Given a TM M and string w, does M halt on w? Note that M doesn't have to accept w; it just has to halt on w.

Proof: By Self-Reference Method

Assume, for the sake of contradiction, language $HALT_{TM}$ is decidable. This means there exists a **Turing machine H** (i.e., H is a decider and L(H) = $HALT_{TM}$) **that takes as its input < M, w >,** where M is the description of the TM encoded in binary and w is a string, and provides the following output:

$$H(<M, w>) \begin{cases} \textbf{Accept:} & \textbf{if M halts on input w} \text{ (M halt at final state / M halt at non-final state on input w)} \\ \textbf{Reject:} & \textbf{if M goes into an infinite loop/ doesn't halt upon input w} \end{cases}$$

Decider H, "On input <M, w>: Run M on w. If M accepts, accept. If M rejects, accept." Then H accepts <M, w> if and only if M halts on w.

If decider H exists, then there exists a TM P such that when < M > (description of a TM M) is applied as an input to P, P calls the decider H and passes on to it an input in the form: **< M, < M >>**. The output of P is 'Accept' if H's output is 'accept'. But P goes in an infinite loop if H's output is 'reject'.

Thus, for the input < M > applied to P, P responds as follows:

$$P(<M>) \begin{cases} \textbf{Accept:} & \text{If M accepts/rejects for input } <M> \\ \textbf{Loop:} & \text{If M goes in an infinite loop/doesn't halt for input } <M> \end{cases}$$

The Turing Machine P calls the decider H internally as H(< M, <M> >) or The Turing machine P also simulates the decider H on input < M, < M >>

If decider H exists, also there exists another **TM N which does the reverse of P**, i.e, when < M > is applied as an input to N, N calls the aforementioned decider H and passes on to it an input in the form: < M, < M >>. The output of N is 'accept' if H's output is 'reject'. But N goes in an infinite loop if H's output is 'accept'.
Thus, for the input < M > applied to N, N responds as follows:

$$N(<M>) \begin{cases} \textbf{Accept:} & \text{If M goes in an infinite loop/ doesn't halt for input } <M> \\ \textbf{Loop:} & \text{If M accepts/rejects for input } <M> \end{cases}$$

The Turing Machine N calls the decider H internally as H(< M, <M> >) or The Turing machine N also simulates the decider H on input < M, < M >>

Note: While H is a decider, P and N are not deciders (they don't have to be), and so they can go in infinite loops.

What happens when the input to TM N is its own description < N >? i.e., N(< N >)

Now N calls the decider H as H(<N, < N >>)

- **If N goes in an infinite loop/doesn't halt for input < N >, N provides the output Accept** (for the input < N > applied to N). **This is a contradiction.**
- **If N accepts/rejects for input < N >, N goes into an infinite loop** (for the input < N > applied to N). **This is also a contradiction.**

So, for either case, there is a contradiction, and so TM N can't exist. But if decider H exists, TM N must exist.
So decider H can't exist. Hence, our initial assumption that $HALT_{TM}$ is decidable is wrong.
Thus, we prove by contradiction that $HALT_{TM}$ is undecidable.

Proof: By Cantor Diagonalization

Since TMs form a countable set, the following table can be constructed, with each row corresponding to a TM and each column corresponding to a TM description encoded in binary.

Each element (i, j) of the table corresponds to what TM M_i does when description of TM M_j (i.e., < M_j >) is fed to the TM M_i as an input string (i= 1, 2, 3, ...; j=1, 2, 3, ...).
There are three possibilities: 1) **accept,** 2) **reject,** 3) **loop (i.e., goes in an infinite loop/doesn't halt)**.
The table is shown below:

Table 1:

Turing Machine	$< M_1 >$	$< M_2 >$	$< M_3 >$	...	...	$< M_N >$	...
M_1	accept	loop	reject	...	...	...	...
M_2	loop	loop	accept	....	...	...	...
M_3	accept	accept	reject	...	...	...	...
...	...	...	...	...	...	...	...
...							
...	...	...	...	...	...	...	...
...							
M_N	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...

If language $HALT_{TM}$ is decidable, it means that there exists a decider H which takes as its input **< M_i, < M_j >>** and yields the following output:

Accept: if M_i halts upon input < M_j > (final state for M_i accept/ reject)

Reject: if M_i goes into an infinite loop/ doesn't halt upon input < M_j >

Corresponding to Table 1, the decider H's table is then as follows

Turing Machine	$< M_1 >$	$< M_2 >$	$< M_3 >$	...	...	$< M_N >$	...
M_1	accept	reject	accept	...	...	...	...
M_2	reject	reject	accept	....	...	...	...
M_3	accept	accept	accept	...	...	...	...
...	...	...	...	...	...	...	...
...							
...	...	...	...	...	...	...	...
...							
M_N	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...

Table 2: A table for the decider H corresponding to $HALT_{TM}$ (assuming $HALT_{TM}$ is decidable)

Using the diagonal in Table 2, we can construct TM P (this is equivalent to P in the self-reference method) which takes as an input < M_i> and acts as follows:

$$P(< M_i >) \begin{cases} \textbf{Accept:} & \text{If } M_i \text{ accepts/rejects} < M_i > \\ \textbf{Loop:} & \text{If } M_i \text{ goes in an infinite loop/doesn't halt for input} < M_i > \end{cases}$$

The Turing Machine P calls the decider H internally as **H(< M_i, < M_i >>)**or The Turing machine P also simulates the decider H on input **< M_i, < M_i >>**

Loop here means P goes in an infinite loop/ doesn't halt (again, P doesn't need to be a decider).
Thus, from Table 1 and Table 2, **P(< M_1 >)= accept, P(< M_2 >)= loop** (because P is not a decider)**, P(< M_3 >)= accept,....** Hence, it is to be noted that **P is not exactly the diagonal of Table 2 but derived from it ('accept' in the diagonal is 'accept' for P and 'reject' in the diagonal is 'loop' for P).**

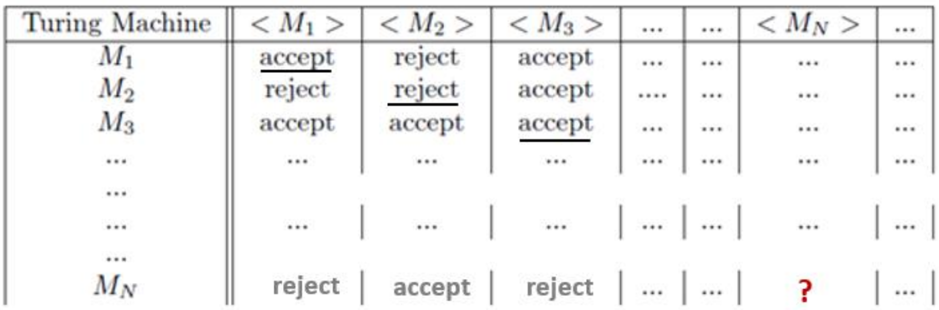
We took the diagonal in Table 1 and took its complement, here also **we can do the same for Table 2** and **construct another TM M_N** (By our assumption, H is a TM and so is M_N. Therefore, it must occur on the list M_1, M_2 , . . . of all TMs).

M_N takes as an input < M_i> and does the opposite of P.

Thus, M_N acts as follows:
Accept: If M_i goes in an infinite loop/ doesn't halt for input < M_i >
Loop: If M_i accepts/ rejects on input < M_i >

Thus, from Table 1 and Table 2, **M_N(< M_1 >)= loop, M_N (< M_2 >)= accept, M_N (< M_3 >)= loop,....** Just like complement of the diagonal in Table 1, cannot be placed in Table 1, TM M_N cannot be placed in Table 1.

- If M_N is placed at the Nth row of Table 2 as shown below, **TM M_N accepts < M_N >, if M_N goes in an infinite loop upon input < M_N >. This is a contradiction.**
- Similarly, **TM M_N goes in an infinite loop upon input < M_N >, if M_N accepts/rejects input < M_N >. This is also a contradiction.**



Turing Machine	$< M_1 >$	$< M_2 >$	$< M_3 >$	...	...	$< M_N >$	...
M_1	accept	reject	accept	...	...	...	...
M_2	reject	reject	accept	....	...	...	...
M_3	accept	accept	accept	...	...	...	...
...	...	...	...	...	...	...	...
...							
...	...	...	...	...	...	...	...
...							
M_N	reject	accept	reject	...	...	?	...

Table 2: A table for the decider H corresponding to $HALT_{TM}$ (assuming $HALT_{TM}$ is decidable)

But since M_N is a TM, it should correspond to a row in Table-2. But since it can't due to the contradiction, we conclude that decider H can't exist.
Thus, we prove by Cantor's diagonalization technique that decider H doesn't exist and thus $HALT_{TM}$ is undecidable.

Note: No matter what M_N does, it is forced to do the opposite, which is obviously a contradiction. Thus, neither TM M_N nor TM H can exist.

2) Prove that Language A_{TM} = {< M, w > | M is a TM and M accepts input w } is undecidable

The problem of determining whether a Turing machine accepts a given input.

Proof: By Cantor Diagonalization

Assume that a Turing Machine U decides language A_{TM}.

$$U(< M, w >) \begin{cases} \textbf{Accept:} & \text{if M accepts an input w (M halt at final state)} \\ \textbf{Reject:} & \text{if M goes into an infinite loop/ halt at non-final state on input w.} \end{cases}$$

We construct a new Turing machine N which takes <M> and calls U internally as U(<M, <M>>).

N(< M >)
- **Accept:** if M goes into an infinite loop/ halt at non-final state on input <M>.
- **Reject:** if M accepts an input <M> (M halt at final state)

Finally, run N on itself, that is, N(<N>) and N calls internally U as U(<N, <N>>)

Thus, the machines take the following actions, with the last line being the contradiction.

- U accepts ⟨M, w⟩ exactly when M accepts w. **i.e., when you call U(<M, w>)**
- N rejects ⟨M⟩ exactly when M accepts input ⟨M⟩. **i.e., when you call N(<M>)** and **N calls internally U(<M, <M>>)**
- N rejects ⟨N⟩ exactly when N accepts input ⟨N⟩ **i.e., when you call N(<N>)** and **N calls internally U as U(<N, <N>>)**

We list all TMs down the rows, M1, M2, … , and all their descriptions across the columns, ⟨M1⟩, ⟨M2⟩, … The entries tell whether the machine in a given row accepts the input in a given column.
The entry is *accept* if the machine accepts the input and entry is *blank* if it rejects or loops on that input.

	<M1>	<M2>	<M3>	<M4>	….
M1	accept		accept		….
M2	accept	accept	accept	accept	….
M3					….
M4	accept	accept			….
….	…. ….	…. ….	…. ….	….	….

If we run Turing Machine U on inputs corresponding to above table, U rejects input ⟨M3, ⟨M2 ⟩⟩. U accepts input ⟨M4, ⟨M1 ⟩⟩. Hence, table for U becomes:

	<M1>	<M2>	<M3>	<M4>	….
M1	accept	reject	accept	reject	….
M2	accept	accept	accept	accept	….
M3	reject	reject	reject	reject	….
M4	accept	accept	reject	reject	….
….	…. ….	…. ….	…. ….	….	….

By our assumption, U is a TM and so is N. Therefore, it must occur on the list M1, M2, . . . of all TMs.
Note that N computes the opposite of the diagonal entries.
The contradiction occurs at the point of the question mark where the entry must be the opposite of itself.

	<M1>	<M2>	<M3>	<M4>	….	<N>	….
M1	accept	reject	accept	reject	….	accept	….
M2	accept	accept	accept	accept	….	accept	….
M3	reject	reject	reject	reject	….	reject	….
M4	accept	accept	reject	reject	….	accept	….
….	….	….	….	….	….	….	….
N	reject	reject	accept	accept	….	???	….
….	….	….	….	….	….	….	….

No matter what N does, it is forced to do the opposite, which is obviously a contradiction. Thus, neither TM N nor TM U can exist.

Exercise:

1. Prove that L^{Φ}_{TM} = {<M>|M is a TM and L(M) = Φ} is undecidable

2. Prove that L^{EQ}_{TM} = { <M_1, M_2> | M are TMs and $L(M_1) = L(M_2)$} is undecidable

Is everything about Turing machines undecidable?

1. **L={<M> | L(M) is regular}**
2. **L={<M> | L(M) is context-free}**
3. **L={<M> | L(M) is finite}**
4. **L={<M> | M has 5 states}**
5. **L={<M> | L(M) is a language}**
6. **L={<M> | M makes at least 5 moves on some input}**

Reducibility (or Reductions)

- We will explore now the phenomenon of undecidability outside the realm of problems concerning automata and Turing machines, i.e., we will look at a Post Correspondence Problem (PCP). We will ultimately show its connection with Turing machines and show that solving PCP is equivalent to deciding A_{TM}.
- Since we already know that A_{TM} is undecidable, by equivalence/reduction, we show here that PCP is non-computable for some cases.

What is Reduction (or Reducibility)?

- **A reduction is a process that converts one problem into another so that the solution of the second problem can be used to solve the first. We say the first problem reduces to the second problem.**
  In other words, a reduction is a way to change a problem that needs to be solved to a problem that is already known how to solve.
- **Reducibility always involves two problems, which we call A and B. If A reduces to B, we can use a solution of B to solve A.**
- Reducibility plays an important role in classifying problems by decidability, and later in complexity theory as well. **When A is reducible to B, solving A cannot be harder than solving B because a solution to B gives a solution to A.**

In terms of computability theory,

- **If A reduces to B and B is decidable, A also is decidable.**
- **Equivalently, if A reduces to B and A is undecidable, then B is also undecidable.**

Note: We say A is reducible to B, and we denote it by $A \leq_m B$. (Mapping reduction)

Definition of the PCP Problem:

Let Σ be an alphabet with at least two symbols. The input of the problem consists of 2-finite lists $x_1, x_2, \ldots, x_n$ and $y_1, y_2, \ldots, y_n$ of n-words over Σ.

A solution to this problem is a sequence of indices, such that the concatenation of the words from the respective list in that sequence yields the same resultant string.

The decision problem then is to decide whether such a solution exists or not, for the given two lists above. We will prove here that this problem is unsolvable by any algorithm.

We can express the PCP as a language as:

L_{PCP} = {< P >|P is an instance of the PCP with a solution/match}

Example 1: Proving Undecidability of L_{PCP} by reducing PCP to empty Intersection problem (i.e., $L_1 \cap L_2 = \emptyset$?)

Solution:

Take a PCP problem, and then write two grammars (corresponding to two lists of PCP).

Consider a PCP Problem

	List-**A**	List-**B**
Word-**1**	1	111
Word-**2**	10111	10
Word-**3**	10	0

Write grammars i.e., CFGs for both the List-A and List-B.

CFG G_A for List-A: $S_A \rightarrow 1\ S_A\ 1 \mid 10111\ S_A\ 2 \mid 10\ S_A\ 3 \mid -$

CFG G_B for List-B: $S_B \rightarrow 111\ S_B\ 1 \mid 10\ S_B\ 2 \mid 0\ S_B\ 3 \mid -$

Numbers 1, 2 and 3 after S_A or S_B keeps track of the sequence numbers (way to remember sequence numbers).

We know this PCP has a solution: **2 1 1 3**

Let's generate the string **101111110** using $S_A \rightarrow 1\ S_A\ 1 \mid 10111\ S_A\ 2 \mid 10\ S_A\ 3 \mid -$

$S_A \Rightarrow 10111S_A2$
$\Rightarrow 101111S_A12$
$\Rightarrow 1011111S_A112$
$\Rightarrow 10111110S_A3112$
$\Rightarrow$ 101111110-3112 (reverse 3112 as 2113 because sequence numbers 1, 2 and 3 are nested in RHS of rules)
$\Rightarrow$ **101111110-2113**

Let's generate the string **101111110** using $S_B \rightarrow 111\ S_B\ 1 \mid 10\ S_B\ 2 \mid 0\ S_B\ 3 \mid -$

$S_B \Rightarrow 10S_B2$
$\Rightarrow 10111S_B12$
$\Rightarrow 10111111S_B112$
$\Rightarrow 101111110S_B3112$
$\Rightarrow$ 101111110-3112 (reverse 3112 as 2113 because sequence numbers 1, 2 and 3 are nested in RHS of rules)
$\Rightarrow$ **101111110-2113**

The above derivations of both G_A and G_B are generating the same string and hence the constructed grammars for List-A and List-B are correct.

Grammars G_A and G_B are recursive grammars and represent infinite set context free languages **L(G_A)** and **L(G_B)**. Hence, PCP is reduced to empty intersection problem i.e., **L(G_A)∩L(G_B)=∅?**

We know that, the empty intersection of two CFLs is undecidable, i.e., $L_1 \cap L_2 = \emptyset$?

As there is no algorithmic way to solve L(G_A)∩L(G_B)=∅?, hence L_{PCP} is also undecidable(not solvable).

In general PCP is known to be Undecidable, that is, there is no particular algorithm that determines whether any PCP has solution or not.

Exercises:

1. **Proving Undecidability of PCP by reducing A_{TM} to PCP**

   We have already seen that A_{TM} is an undecidable language. In order to prove that PCP is undecidable, it suffices to show that we can reduce A_{TM} to PCP. Thus, we will show that if we can decide PCP, then we can decide A_{TM}. Given any Turing machine M and input w, we construct an instance $P_{M,w}$ such that $P_{M,w}$ has a solution index sequence (or match) if M accepts w. That is, $P_{M,w}$ has a match if there is an accepting CH of M on w. So, if we can determine whether the instance has a match, we would be able to determine whether M accepts w.

   ……

2. **Proving Undecidability of empty Intersection problem by reducing it to PCP**

References:

- https://web.iitd.ac.in/~debanjan/
- Undecidable(PCP,UTM,Reduction).pdf by Prof. Preet Kanwal, PES University.