Introduction to Languages and Computation

George Voutsadakis[1]

[1]Mathematics and Computer Science
Lake Superior State University

LSSU Math 400

Limits of Computablity

- We investigate the power of algorithms to solve problems.
- We demonstrate certain problems that can be solved algorithmically and others that cannot.
- Our objective is to explore the limits of algorithmic solvability.
- Why is studying unsolvability useful?
  - If a problem is known to be algorithmically unsolvable, then we know that it must be simplified or altered before an algorithmic solution can be found.
  - Even if we only deal with problems that are solvable, looking at the unsolvable can help gain an important perspective on computation.

Subsection 1

Decidable Languages

Decidable Problems on Regular Languages

- We show certain computational problems concerning finite automata are decidable.
- We give algorithms for :
  - testing whether a finite automaton accepts a string;
  - whether the language of a finite automaton is empty;
  - whether two finite automata are equivalent.
- We chose to represent computational problems by languages, since we already have terminology for dealing with languages.
- For example, the **acceptance problem** for DFAs of testing whether a particular deterministic finite automaton accepts a given string can be expressed as a language, A_{DFA}. This language contains the encodings of all DFAs together with strings that the DFAs accept:

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

- The problem of testing whether a DFA B accepts an input w is the same as the problem of testing whether $\langle B, w\rangle$ is a member of the language A_{DFA}.

Decidability of A_{DFA}

Theorem

A_{DFA} is a decidable language.

- We simply need to present a TM M that decides A_{DFA}.

  M: On input $\langle B, w \rangle$, where B is a DFA and w is a string,

  1. Simulate B on input w.
  2. If the simulation ends in an accept state, accept. If it ends in a non accepting state, reject.

  We look at a few implementation details:

  - The input $\langle B, w \rangle$ is a representation of a DFA B and of a string w. B's representation may be a list of its five components, Q, Σ, δ, q_0 and F.
  - M first determines whether it properly represents a DFA B and a string w. If not, M rejects.
  - Then M carries out the simulation directly. It keeps track of B's current state and B's current position in the input w by writing this information down on its tape.

The Language A_{NFA}

- Let $A_{NFA} = \{\langle B, w\rangle : B \text{ is an NFA that accepts input string } w\}$.

Theorem

A_{NFA} is a decidable language.

- We present a TM N that decides A_{NFA}.
- We could design N to directly simulate an NFA instead of a DFA.
- Instead, we have N use M as a subroutine.
- Because M works with DFAs, N first converts the input NFA to a DFA before passing it to M.
  N: On input $\langle B, w\rangle$, where B is an NFA, and w is a string,
  1. Convert NFA B to an equivalent DFA C, using the procedure for this conversion presented previously.
  2. Run the TM M of the preceding slide on input $\langle C, w\rangle$.
  3. If M accepts, accept; otherwise, reject.
- Running TM M in Stage 2 means incorporating M into the design of N as a subprocedure.

The Language A_{REX}

- Consider
  $A_{REX} = \{\langle R, w\rangle : R \text{ is a regular expression that generates string } w\}$.

Theorem

A_{REX} is a decidable language.

- The following TM P decides A_{REX}.

  P: On input $\langle R, w\rangle$, R a regular expression and w a string,

  1. Convert regular expression R to an equivalent NFA A by using the procedure for this conversion given previously.
  2. Run TM N of the preceding slide on input $\langle A, w\rangle$.
  3. If N accepts, accept; if N rejects, reject.

Emptiness Testing

- In the next proof we must determine whether a finite automaton accepts any strings at all.
- Let $\mathrm{E}_{\mathrm{DFA}} = \{\langle A \rangle : A \text{ is a DFA and } L(A) = \emptyset\}$.

Theorem

$\mathrm{E}_{\mathrm{DFA}}$ is a decidable language.

- A DFA accepts some string iff reaching an accept state from the start state by traveling along the arrows of the DFA is possible.
- To test this condition we can design a TM T that uses a marking algorithm similar to that used for graphs.
  T: On input $\langle A \rangle$, where A is a DFA,
  1. Mark the start state of A.
  2. Repeat until no new states get marked:
     3. Mark any state that has a transition coming into it from any state that is already marked.
  4. If no accept state is marked, accept; otherwise, reject.

The Language $\mathrm{EQ}_{\mathrm{DFA}}$

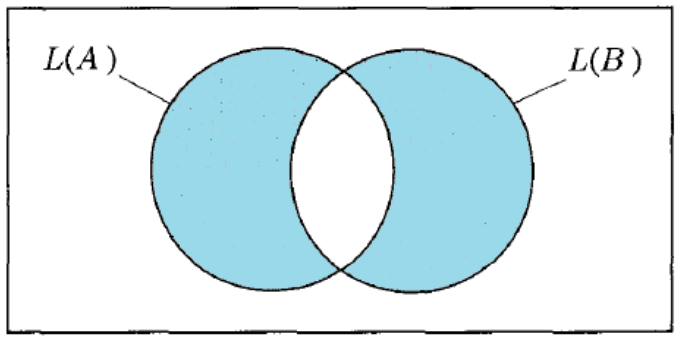
- The next theorem states that determining whether two DFAs recognize the same language is decidable.
- Let $\mathrm{EQ}_{\mathrm{DFA}} = \{\langle A, B\rangle : A \text{ and } B \text{ are DFAs and } L(A) = L(B)\}$.

Theorem

$\mathrm{EQ}_{\mathrm{DFA}}$ is a decidable language.

- To prove this theorem we use emptiness testing.
- We construct a new DFA C from A and B, where C accepts only those strings that are accepted by either A or B but not by both. Thus, if A and B recognize the same language, C will accept nothing. The language of C is

$$L(C) = (L(A) \cap \overline{L(B)}) \cup (\overline{L(A)} \cap L(B)).$$



This expression is called the **symmetric difference** of $L(A)$ and $L(B)$.

Algorithm for $\mathrm{EQ}_{\mathrm{DFA}}$

- The symmetric difference is useful because $L(C) = \emptyset$ if and only if $L(A) = L(B)$.
- We can construct C from A and B with the constructions for proving the class of regular languages closed under complementation, union, and intersection. These constructions are algorithms that can be carried out by Turing machines.
- Once we have constructed C we can use emptiness testing to test whether $L(C)$ is empty. If it is empty, $L(A)$ and $L(B)$ must be equal.

  F: On input $\langle A, B \rangle$, where A and B are DFAs,

  1. Construct DFA C as described.
  2. Run TM T on input $\langle C \rangle$.
  3. If T accepts, accept. If T rejects, reject.

The Language A_{CFG}

- Let $A_{CFG} = \{\langle G, w\rangle : G \text{ is a CFG that generates string } w\}$.

Theorem

A_{CFG} is a decidable language.

- For CFG G and string w we want to determine if G generates w.
- One idea is to use G to go through all derivations to determine whether any is a derivation of w.
- Since infinitely many derivations may have to be tried, this is unworkable.
- If G does not generate w, this algorithm would never halt.
- This idea gives a Turing machine that is a recognizer, but not a decider, for A_{CFG}.
- To make this Turing machine into a decider we need to ensure that the algorithm tries only finitely many derivations.

Algorithm for A_{CFG}

- It is possible to show that, if G is in Chomsky normal form, any derivation of w has $2n-1$ steps, where n is the length of w.
- In that case checking only derivations with $2n-1$ steps to determine whether G generates w is sufficient.
- We can convert G to Chomsky normal form by using the procedure given previously.

  S: On input $\langle G, w \rangle$, where G is a CFG and w is a string,

  1. Convert G to an equivalent grammar in Chomsky normal form.
  2. Let n be the length of w.
     - If $n \neq 0$, then list all derivations with $2n-1$ steps;
     - If $n = 0$, then list all derivations with 1 step.
  3. If any of these derivations generate w, accept; if not, reject.
- Since we have given procedures for converting back and forth between CFGs and PDAs, all results on the decidability of problems concerning CFGs apply also to PDAs.

CFL Emptiness Testing

- We can show that the problem of determining whether a CFG generates any strings at all is decidable.
- Let $\mathrm{E}_{\mathrm{CFG}} = \{\langle G \rangle : G \text{ is a CFG and } L(G) = \emptyset\}$.

Theorem

$\mathrm{E}_{\mathrm{CFG}}$ is a decidable language.

- To find an algorithm for this problem we might attempt to use TM S for string generation. It states that we can test whether a CFG generates some particular string w.
- To determine whether $L(G) = \emptyset$ the algorithm might try going through all possible w's, one by one. But there are infinitely many w's to try, so this method could end up running forever.
- In order to determine whether the language of a grammar is empty, we need to test whether the start variable can generate a string of terminals.

CFL Emptiness Testing: The Algorithm

- The algorithm determines, for each variable, whether it is capable of generating a string of terminals. When the algorithm has determined that a variable can generate some string of terminals, the algorithm keeps track of this information by placing a mark on that variable.
  - First, the algorithm marks all the terminal symbols in the grammar.
  - Then, it scans all the rules of the grammar. If it ever finds a rule that permits some variable to be replaced by some string of symbols all of which are already marked, the algorithm knows that this variable can be marked, too.
  - The algorithm continues in this way until it cannot mark any additional variables.

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

1. Mark all terminal symbols in G.
2. Repeat until no new variables get marked:
   3. Mark any variable A where G has a rule $A \to U_1 U_2 \cdots U_k$ and each symbol $U_1, \ldots, U_k$ has already been marked.
4. If the start variable is not marked, accept; otherwise, reject.

The Language $\mathrm{EQ}_{\mathrm{CFG}}$

- Next we consider the problem of determining whether two context-free grammars generate the same language.
- Let $\mathrm{EQ}_{\mathrm{CFG}} = \{\langle G, H\rangle : G \text{ and } H \text{ are CFGs and } L(G) = L(H)\}$.
- The algorithm that decides the corresponding language $\mathrm{EQ}_{\mathrm{DFA}}$ for finite automata is based on the closedness of regular languages under complementation, intersection and union.
- The class of context-free languages is not closed under complementation or intersection, and the technique cannot be applied.
- In fact, $\mathrm{EQ}_{\mathrm{CFG}}$ is not decidable.
- On the other hand, every context-free language is decidable.

Decidability of Context-Free Languages

Theorem

Every context-free language is decidable.

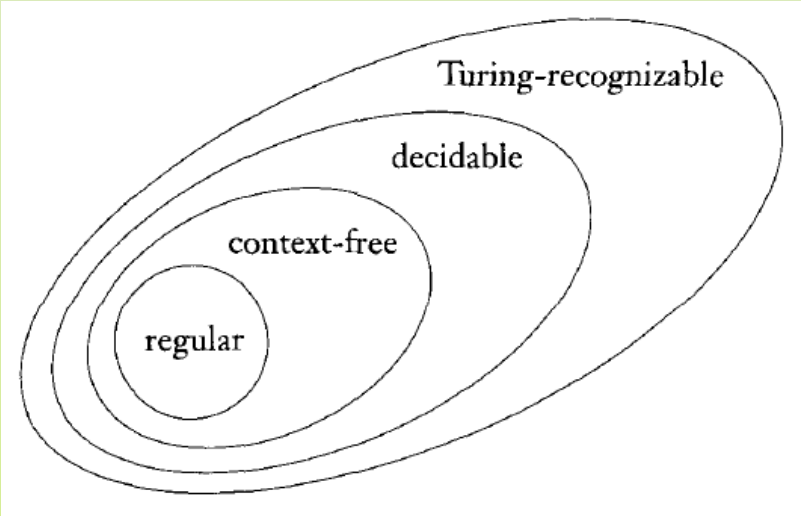
- Let A be a CFL. Our objective is to show that A is decidable.
- A bad idea is to convert a PDA for A directly into a TM.
  - That is not hard to do, since simulating a stack with the TM's more versatile tape is easy.
  - The PDA for A may be nondeterministic, but we can convert it into a nondeterministic TM and we know that any nondeterministic TM can be converted into an equivalent deterministic TM.
- The difficulty is that some branches of the PDA's computation may go on forever, reading and writing the stack without ever halting.
- The simulating TM then would also have some non-halting branches in its computation, and so the TM would not be a decider.

Deciding a Context-Free Language A: The Algorithm

- The TM S deciding $\mathrm{A}_{\mathrm{CFG}}$ is used.
- Let G be a CFG for A.
- We design a TM M_G that decides A by including a copy of G into M_G:

  M_G: On input w
  1. Run TM S on input $\langle G, w \rangle$
  2. If this machine accepts, accept; if it rejects, reject.

- The relationship among the four classes of regular, context free, decidable and Turing-recognizable languages:



Subsection 2

The Halting Problem

Algorithmic Unsolvability

- One of the most philosophically important theorems of the theory of computation is the following:

  There is a specific problem that is algorithmically unsolvable.

- Computers are so powerful that one may believe that all problems can be solved by a computer.
- However, computers are limited in a fundamental way.
- Even some ordinary problems are computationally unsolvable:
  Given a computer program and a precise specification of what that program is supposed to do, the task is to verify that the program performs as specified, i.e., that it is correct.
  - Because both the program and the specification are mathematically precise objects, the hope was to automate the process of verification by feeding these objects into a suitably programmed computer.
  - However, the general problem of software verification is not solvable by computer.

The Problem A_{TM}

- We look at types of problems that are unsolvable and learn techniques for proving unsolvability.
- We start with the problem of determining whether a Turing machine accepts a given input string:

$$A_{TM} = \{\langle M, w\rangle : M \text{ is a TM and } M \text{ accepts } w\}.$$

Theorem

A_{TM} is undecidable.

- We first observe that A_{TM} is Turing-recognizable.
- A consequence is that recognizers are more powerful than deciders, i.e., requiring a TM to halt on all inputs restricts the kinds of languages that it can recognize.

A Turing Machine Recognizing A_{TM}

- The following Turing machine U recognizes A_{TM}:
  U: On input $\langle M, w \rangle$, where M is a TM and w is a string,
  1. Simulate M on input w.
  2. If M ever enters its accept state, accept; if M ever enters its reject state, reject.
- This machine loops on input $\langle M, w \rangle$, if M loops on w, which is why this machine does not decide A_{TM}.
- If the algorithm had some way to determine that M was not halting on w, it could reject, whence A_{TM} is sometimes called the **halting problem**.
- The Turing machine U is called **universal** because it is capable of simulating any other Turing machine when provided with the description of that machine.

Cantor's Counting Method

- The proof of the undecidability of the halting problem uses a technique called **diagonalization** (Georg Cantor, 1873).
- Cantor was concerned with the problem of measuring the sizes of infinite sets.
- If we have two infinite sets, how can we tell whether one is larger than the other or whether they are of the same size?
  - For a finite set, we simply count its elements and the resulting number is its size.
  - If we try to count the elements of an infinite set, we will never finish!
- Example: Take the set of even integers and the set of all strings over $\{0, 1\}$. Both sets are infinite and thus larger than any finite set, but is one of the two larger than the other? How can we compare their relative size?
- Cantor observed that two finite sets have the same size if the elements of one set can be paired with the elements of the other set.
- This method compares the sizes without resorting to counting.

Correspondences

Definition (Correspondence)

Assume that we have sets A and B and a function f from A to B.

- Say that f is **one-to-one** if it never maps two different elements to the same place, i.e., if $f(a) \neq f(b)$ whenever $a \neq b$.
- Say that f is **onto** if it hits every element of B, i.e., if for every $b \in B$, there is an $a \in A$, such that $f(a) = b$.
- Say that A and B are the **same size** if there is a one-to-one, onto function $f : A \to B$.
- A function that is both one-to-one and onto is called a **correspondence**. In a correspondence every element of A maps to a unique element of B and each element of B has a unique element of A mapping to it, i.e., a correspondence is simply a way of pairing the elements of A with the elements of B.

Illustrating the Definition

- Let $\mathbb{N}$ be the set of natural numbers $\{1, 2, 3, \ldots\}$ and let $\mathbb{E}$ be the set of even natural numbers $\{2, 4, 6, \ldots\}$.
- Using Cantor's definition of size we can see that $\mathbb{N}$ and $\mathbb{E}$ have the same size.
- The correspondence f mapping $\mathbb{N}$ to $\mathbb{E}$ is simply

$$f(n) = 2n.$$

We can visualize f more easily with the help of a table:

n	$f(n)$
1	2
2	4
3	6
$\vdots$	$\vdots$

- This example may seem bizarre, since $\mathbb{E}$ seems smaller than $\mathbb{N}$ because $\mathbb{E}$ is a proper subset of $\mathbb{N}$.

Countable Sets

Definition (Countable Set)

A set A is **countable** if either it is finite or it has the same size as $\mathbb{N}$.

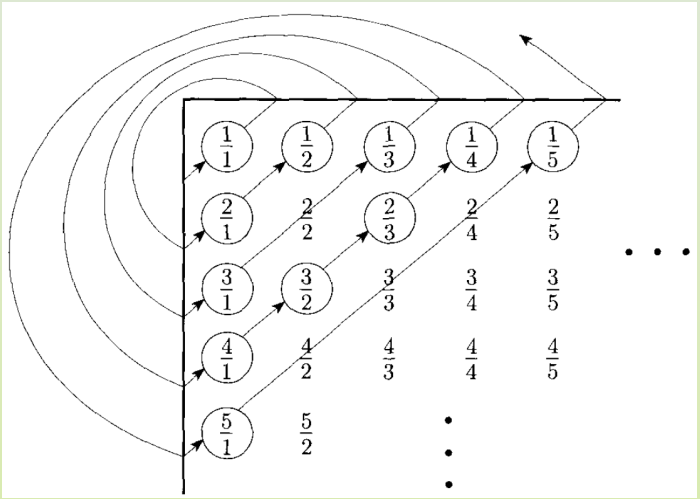
- Example: Let $\mathbb{Q} = \{\frac{m}{n} : m, n \in \mathbb{N}\}$ be the set of positive rational numbers.
  - $\mathbb{Q}$ seems to be much larger than $\mathbb{N}$.
  - Yet these two sets are the same size according to our definition.

  We give a correspondence with $\mathbb{N}$ to show that $\mathbb{Q}$ is countable. One easy way to do so is to list all the elements of $\mathbb{Q}$. Then, we pair:
  - the first element on the list with the number 1 from $\mathbb{N}$;
  - the second element on the list with the number 2 from $\mathbb{N}$;
  - and so on.

  Every member of $\mathbb{Q}$ should appear only once on the list.

Making a List of the Elements of $\mathbb{Q}$

- We make an infinite matrix containing all the positive rational numbers:



  - The ith row contains all numbers with numerator i and the jth column has all numbers with denominator j. So the number $\frac{i}{j}$ occurs in the ith row and jth column.
  - To turn this matrix into a list, we list the elements on the diagonals, starting from the corner.
    - The first diagonal contains the single element $\frac{1}{1}$;
    - The second diagonal contains the two elements $\frac{2}{1}$ $\frac{1}{2}$.
    - The third diagonal contains $\frac{3}{1}, \frac{2}{2}$ and $\frac{1}{3}$. If we simply added these to the list, we would repeat $\frac{1}{1} = \frac{2}{2}$. We avoid doing so by skipping an element when it would cause a repetition, i.e., we add only $\frac{3}{1}$ and $\frac{1}{3}$.
    - Continuing in this way we obtain a list of all the elements of $\mathbb{Q}$.

Uncountable Sets

- After seeing this correspondence of $\mathbb{N}$ and $\mathbb{Q}$, one might think that any two infinite sets have the same size.
- For some infinite sets no correspondence with $\mathbb{N}$ exists.
- Such sets are called **uncountable**.
- The set of real numbers is an example of an uncountable set.
- A real number is one that has a decimal representation, e.g., $\pi = 3.1415926\cdots$ and $\sqrt{2} = 1.4142135\cdots$ are real numbers.
- We use $\mathbb{R}$ to denote the set of real numbers.
- Cantor proved that $\mathbb{R}$ is uncountable and in doing so he introduced the diagonalization method.

Uncountabilty of $\mathbb{R}$: Outline of Proof

Theorem

$\mathbb{R}$ is uncountable.

- We show that no correspondence exists between $\mathbb{N}$ and $\mathbb{R}$. The proof is by contradiction. Suppose that a correspondence f exists between $\mathbb{N}$ and $\mathbb{R}$. f must pair all the members of $\mathbb{N}$ with all the members of $\mathbb{R}$. We find an $x \in \mathbb{R}$ that is not paired with anything in $\mathbb{N}$, which will give a contradiction.
  The way we find this x is by actually choosing each digit of x to make x different from one of the real numbers that is paired with an element of $\mathbb{N}$. In the end we are sure that x is different from any real number that is paired.

Uncountabilty of $\mathbb{R}$: Illustration of the Idea

- Suppose that the correspondence f exists. Let $f(1) = 3.14159\ldots$, $f(2) = 55.55555\ldots$, $f(3) = 0.12345\ldots$, $\ldots$. Then f pairs the number 1 with $3.14159\ldots$, the number 2 with $55.55555\ldots$, and so on. The following table shows a few values of a hypothetical correspondence f between $\mathbb{N}$ and $\mathbb{R}$:

n	$f(n)$
1	$3.14159\ldots$
2	$55.55555\ldots$
3	$0.12345\ldots$
4	$0.50000\ldots$
$\vdots$	$\vdots$

The desired x is a number between 0 and 1, such that $x \neq f(n)$, for any n.

Uncountabilty of $\mathbb{R}$: Illustration of the Idea (Cont'd)

- We construct x:
  - To ensure that $x \neq f(1)$ we let the first digit of x be anything different from the first fractional digit 1 of $f(1) = 3.14159\ldots$. Arbitrarily, we let it be 4.
  - To ensure that $x \neq f(2)$ we let the second digit of x be anything different from the second fractional digit 5 of $f(2) = 55.555555\ldots$. Arbitrarily, we let it be 6.
  - The third fractional digit of $f(3) = 0.12345\ldots$ is 3, so we let x be anything different, say, 4.
  - Continuing in this way down the diagonal of the table for f, we obtain all the digits of x.

  We know that x is not $f(n)$ for any n because it differs from $f(n)$ in the nth fractional digit. We overcome the problem that arises because certain numbers, such as $0.1999\ldots$ and $0.2000\ldots$, are equal even though their decimal representations are different, by never selecting the digits 0 or 9 when we construct x.

Existence of non-Turing Recognizable Languages

- The preceding theorem has an important application to the theory of computation: It shows that some languages are not decidable or even Turing recognizable, for the reason that there are uncountably many languages, but only countably many Turing machines.
- Because each Turing machine can recognize a single language and there are more languages than Turing machines, some languages are not recognized by any Turing machine.
- Such languages are not Turing-recognizable, as we state in the following corollary:

Corollary

Some languages are not Turing-recognizable.

Proof of the Corollary

- To see that the set of all Turing machines is countable, first observe that the set of all strings Σ^* is countable, for any alphabet Σ.

  There are finitely many strings of each length, so we may form a list of Σ^* by writing down all strings of length 0, length 1, length 2, and so on.

The set of all Turing machines is countable because each Turing machine M has an encoding into a string $\langle M \rangle$. If we simply omit those strings that are not legal encodings of Turing machines, we can obtain a list of all Turing machines.

To show that the set of all languages is uncountable, we first observe that the set of all infinite binary sequences is uncountable. An infinite binary sequence is an unending sequence of 0s and 1s. Let B be the set of all infinite binary sequences. The uncountability of B may be shown by diagonalization, as was done before for $\mathbb{R}$.

Uncountability of the Set of all Languages over Σ

- Let $\mathcal{L}$ be the set of all languages over alphabet Σ. We show that $\mathcal{L}$ is uncountable by giving a correspondence with B, thus showing that the two sets are the same size. Let $\Sigma = \{s_1, s_2, s_3, \ldots\}$. Each language $A \in \mathcal{L}$ has a unique sequence in B. The ith bit of that sequence is a 1 if $s_i \in A$ and a 0 if $s_i \notin A$, which is called the **characteristic sequence** of A. E.g., if A were the language of all strings starting with a 0 over the alphabet $\{0, 1\}$, its characteristic sequence χ_A would be

$$\begin{array}{rcccccccccccc} \Sigma^* & = & \{ & \varepsilon, & 0, & 1, & 00, & 01, & 10, & 11, & 000, & 001, & \cdots & \}; \\ A & = & \{ & & 0, & & 00, & 01, & & & 000, & 001, & \cdots & \}; \\ \chi_A & = & & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & \cdots & \end{array}$$

  The function $f : \mathcal{L} \to B$, where $f(A)$ equals the characteristic sequence of A, is one-to-one and onto and hence a correspondence. Therefore, as B is uncountable, $\mathcal{L}$ is uncountable as well. Thus, the set of all languages cannot be put into a correspondence with the set of all Turing machines.

Undecidability of the Halting Problem

Theorem (Undecidability of the Halting Problem)

$\mathrm{A_{TM}} = \{\langle M, w\rangle : M \text{ is a TM and } M \text{ accepts } w\}$ is undecidable.

- We assume that $\mathrm{A_{TM}}$ is decidable and obtain a contradiction. Suppose that H is a decider for $\mathrm{A_{TM}}$. On input $\langle M, w\rangle$, where M is a TM and w is a string, H halts and accepts, if M accepts w, and H halts and rejects, if M fails to accept w. In other words, we assume that H is a TM, where

$$H(\langle M, w\rangle) = \begin{cases} \text{accept}, & \text{if } M \text{ accepts } w \\ \text{reject}, & \text{if } M \text{ does not accept } w \end{cases}.$$

  Now we construct a new Turing machine D with H as a subroutine. This new TM calls H to determine what M does when the input to M is its own description $\langle M\rangle$. Once D has determined this information, it does the opposite, i.e., it rejects, if M accepts, and accepts, if M does not accept.

The Turing Machine D

- The Machine D is described as follows:

  D: On input $\langle M \rangle$, where M is a TM,

  1. Run H on input $\langle M, \langle M \rangle \rangle$.
  2. Output the opposite of what H outputs: if H accepts, reject; if H rejects, accept.

  In summary, $D(\langle M \rangle) = \begin{cases} \text{accept}, & \text{if } M \text{ does not accept } \langle M \rangle \\ \text{reject}, & \text{if } M \text{ accepts } \langle M \rangle \end{cases}$.

  When we run D with its own description $\langle D \rangle$ as input, we get

$$D(\langle D \rangle) = \begin{cases} \text{accept}, & \text{if } D \text{ does not accept } \langle D \rangle \\ \text{reject}, & \text{if } D \text{ accepts } \langle D \rangle \end{cases}.$$

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

Illustration of the Diagonalization

$M_i(\langle M_j \rangle)$

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\cdots$
M_1	accept		accept		
M_2	accept	accept	accept	accept	
M_3					$\cdots$
M_4	accept	accept			
$\vdots$		$\vdots$			

$H(\langle M_i, \langle M_j \rangle \rangle)$

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\cdots$
M_1	accept	reject	accept	reject	
M_2	accept	accept	accept	accept	
M_3	reject	reject	reject	reject	$\cdots$
M_4	accept	accept	reject	reject	
$\vdots$		$\vdots$			

Adding Machine D in the Last Table

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\cdots$	$\langle D \rangle$	$\cdots$
M_1	accept	reject	accept	reject		accept	
M_2	accept	accept	accept	accept		accept	
M_3	reject	reject	reject	reject	$\cdots$	reject	$\cdots$
M_4	accept	accept	reject	reject		accept	
$\vdots$		$\vdots$			$\ddots$		
D	reject	reject	accept	accept		???	
$\vdots$		$\vdots$					$\ddots$

Characterization of Decidability

- We exhibited a language, namely A_{TM}, that is undecidable.
- Now we demonstrate a language that is not even Turing-recognizable.
- Recall that A_{TM} was shown to be Turing-recognizable.
- The following theorem shows that, if both a language and its complement are Turing-recognizable, the language is decidable.
  - Thus, for any undecidable language, either it or its complement is not Turing-recognizable.
- A language is **co-Turing-recognizable** if it is the complement of a Turing-recognizable language.

Theorem (Characterization of Decidability)

A language is decidable if and only if it is both Turing-recognizable and co-Turing-recognizable.

Proof of the Characterization Theorem

- If A is decidable, both A and $\overline{A}$ are Turing-recognizable:
  - Any decidable language is Turing-recognizable;
  - The complement of a decidable language also is decidable.
- If both A and $\overline{A}$ are Turing-recognizable, let M_1 be the recognizer for A and M_2 the recognizer for $\overline{A}$. The following TM M decides A:
  M: On input w
  1. Run both M_1 and M_2 on input w in parallel.
  2. If M_1 accepts, accept; if M_2 accepts, reject.

  Running the two machines in parallel means that M has two tapes, one for simulating M_1 and the other for simulating M_2. M takes turns simulating one step of each machine until one of them accepts. M decides A: Every string w is either in A or $\overline{A}$. Therefore either M_1 or M_2 must accept w. Because M halts whenever M_1 or M_2 accepts, M always halts and so it is a decider. Furthermore, it accepts all strings in A and rejects all strings not in A. So M is a decider for A, and, thus, A is decidable.

A Non Turing Recognizable Language

Corollary

$\overline{\mathrm{A}_{\mathrm{TM}}}$ is not Turing-recognizable.

- We know that A_{TM} is Turing-recognizable.
  If $\overline{\mathrm{A}_{\mathrm{TM}}}$ also were Turing recognizable, A_{TM} would be decidable.
  But the preceding theorem shows that A_{TM} is not decidable.
  We conclude that $\overline{\mathrm{A}_{\mathrm{TM}}}$ is not Turing-recognizable.