Theory of Computation

(Algorithmic Solvability)

Pramod Ganapathi
Department of Computer Science
State University of New York at Stony Brook

January 24, 2021

How do we compute?

Problem

- What is an algorithm?

How do we compute?

Problem

- What is an algorithm?

Solution

- An algorithm is an effective/systematic/mechanical method for achieving the desired result for a given problem.

Problem

- What are the properties of an algorithm?

How do we compute?

Problem

- What is an algorithm?

Solution

- An algorithm is an effective/systematic/mechanical method for achieving the desired result for a given problem.

Problem

- What are the properties of an algorithm?

Solution

- It has a finite number of instructions.
- If carried out without error, it produces the desired result in a finite number of steps.
- It can be carried out by a human with only paper and pen.
- It requires no insight, intuition, or ingenuity, on the part of the human carrying out the method.

One big question

Problem

- Are Turing machines powerful enough to model any conceivable algorithm?

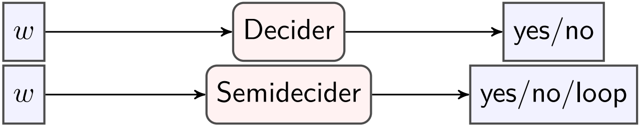
Approach

- To solve this problem, we need to formally define algorithm.
- Before attempting to define algorithm, we need to understand the capabilities and limitations of Turing machines.

What are the types of computational problems?

Types

- Decision problems:
  Problems with input w and output "yes" or "no" answer.
  ("yes": $w \in L$. "no": $w \notin L$.)
  e.g.: Given a specific chess configuration and it is your turn, can you win the chess game?



- Function computation:
  Problems with input w and output $f(w)$.
  e.g.: Given the Facebook graph, what is the minimum number of people connected between you and your role model?

What are Turing-decidable languages?

Definitions

- A Turing machine M accepts (or rejects) a given input string w iff the initial configuration yields the accepting (or rejecting) configuration for the given string w.
- A Turing machine M decides a language $L \in \Sigma^*$ iff for all strings $w \in \Sigma^*$,

$$\begin{cases} M \text{ accepts } w, & \text{if } w \in L, \\ M \text{ rejects } w, & \text{if } w \notin L. \end{cases}$$

- A language is called Turing-decidable or recursive iff there exists a TM that decides it.

- Does this mean that a Turing machine that decides a language never enters an infinite loop?

What are Turing-decidable languages?

Σ^*		✓,✗	$\mathbb{N}$	IsPrime?	✓,✗
$w_1 \in L$	$\longrightarrow$	✓	2	$\longrightarrow$	✓
$w_2 \notin L$	$\longrightarrow$	✗	3	$\longrightarrow$	✓
$w_3 \notin L$	$\longrightarrow$	✗	4	$\longrightarrow$	✗
$w_4 \in L$	$\longrightarrow$	✓	5	$\longrightarrow$	✓
$w_5 \in L$	$\longrightarrow$	✓	6	$\longrightarrow$	✗
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$w_n \notin L$	$\longrightarrow$	✗	97	$\longrightarrow$	✓
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

What are Turing-decidable languages?

Examples

- All regular languages
- All context-free languages
- Several non-context-free languages such as:
  $L = \{a^n b^n c^n \mid n \geq 0\}$
  $L = \{w \mid w = w^R \text{ and } w \in \{a, b\}^*\}$
  $L = \{ww \mid w \in \{a, b\}^*\}$
  $L = \{p \mid p \text{ is a prime}\}$

- What languages are Turing-undecidable languages?

What are Turing-computable functions?

Definitions

- The output of a TM for input string w is string w' iff

$$(q_0, \trianglerighteq w) \vdash^* (q_{\text{acc}}, \trianglerighteq w')$$

- Let function $f : \Sigma^* \to \Sigma^*$
- A Turing machine computes a function f iff for all strings $w \in \Sigma^*$,

$$M \text{ outputs } f(w), \text{ i.e.,}$$
$$(q_0, \trianglerighteq w) \vdash^* (q_{\text{acc}}, \trianglerighteq f(w))$$

- A function $f : \Sigma^* \to \Sigma^*$ is called Turing-computable or recursive iff there exists a TM that computes it.

- Why do we use the term recursive to describe both the languages decided by and the functions computed by Turing machines?

What are Turing-computable functions?

Σ^*	f	Σ^*
w_1	$\longrightarrow$	$f(w_1)$
w_2	$\longrightarrow$	$f(w_2)$
w_3	$\longrightarrow$	$f(w_3)$
w_4	$\longrightarrow$	$f(w_4)$
w_5	$\longrightarrow$	$f(w_5)$
$\vdots$	$\vdots$	$\vdots$
w_n	$\longrightarrow$	$f(w_n)$
$\vdots$	$\vdots$	$\vdots$

$\mathbb{N}$	Cube	$\mathbb{N}$
1	$\longrightarrow$	1
2	$\longrightarrow$	8
3	$\longrightarrow$	27
4	$\longrightarrow$	64
5	$\longrightarrow$	125
$\vdots$	$\vdots$	$\vdots$
10	$\longrightarrow$	1000
$\vdots$	$\vdots$	$\vdots$

What are Turing-semidecidable languages?

Definitions

- A Turing machine M semidecides a language $L \in \Sigma^*$ iff for all strings $w \in \Sigma^*$,

$$\begin{cases} M \text{ accepts } w, & \text{if } w \in L, \\ M \text{ rejects } w \text{ or runs forever}, & \text{if } w \notin L. \end{cases}$$

- A language is called Turing-semidecidable or recursively enumerable iff there exists a TM that semidecides it.

- Does this mean that a Turing machine that semidecides a language can enter an infinite loop?

What are Turing-semidecidable languages?

Σ^*		✓, ✗, run forever
$w_1 \in L$	$\longrightarrow$	✓
$w_2 \notin L$	$\longrightarrow$	✗, run forever
$w_3 \notin L$	$\longrightarrow$	✗, run forever
$w_4 \in L$	$\longrightarrow$	✓
$w_5 \in L$	$\longrightarrow$	✓
$\vdots$	$\vdots$	$\vdots$
$w_n \notin L$	$\longrightarrow$	✗, run forever

Programs	Correctness?	✓, ✗, run forever
P_1	$\longrightarrow$	✗
P_2	$\longrightarrow$	✓
P_3	$\longrightarrow$	run forever
P_4	$\longrightarrow$	run forever
P_5	$\longrightarrow$	✗
$\vdots$	$\vdots$	$\vdots$
P_n	$\longrightarrow$	run forever

Types of computational problems solved by TM's

Types

The three types of computational problems solved by TM's are:

- Turing-decidable languages
- Turing-computable functions
- Turing-semidecidable languages

- Can we formalize the notion of an algorithm using the computation ideas described above?

What might be algorithms?

Properties of algorithms

Intuitively, an algorithm has the following properties:

1. It is a sequence of steps that gives the correct result to a computational problem.
2. It should work for all input instances from a given domain.

What might be algorithms?

Properties of algorithms

Intuitively, an algorithm has the following properties:

1. It is a sequence of steps that gives the correct result to a computational problem.
2. It should work for all input instances from a given domain.

Describing algorithms

- The properties imply that an algorithm always halts.

Type of computation	Always halt?
TM's for decidable languages	✓
TM's for computable functions	✓
TM's for semidecidable languages	✗

- A TM for a Turing-decidable language or a Turing-computable function formalizes the intuitive notion of an algorithm.

What are algorithms?

Definitions

- Algorithm:
  Turing machine for a Turing-decidable language or Turing machine for a Turing-computable function.
- Algorithmic solvability:
  Turing-decidability or Turing-computability
- Algorithmic unsolvability:
  Turing-undecidability or Turing-noncomputability i.e.,
  Turing-semidecidability and not Turing-semidecidability

Examples of algorithms?

Examples

- Thousands of algorithms taught in the courses such as algorithms, data structures, programming, operating systems, networking, security, operations research, computer graphics, computer vision, etc
- The notion of algorithm is extended to include randomized algorithms, parallel algorithms, distributed algorithms, machine learning (or self-learning) algorithms, self-improving algorithms, quantum algorithms, etc

- Are Turing machines powerful enough to model any conceivable algorithm?

What is Church-Turing thesis?

Hypothesis

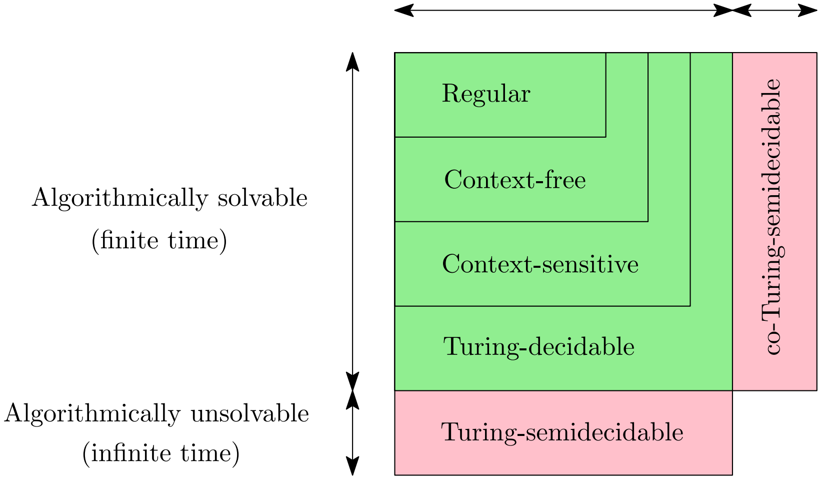
- Any algorithm can be executed by a Turing machine.
- Anything that can be computed can be computed by a Turing machine.
- A function on the natural numbers can be calculated by an effective method iff it is computable by a Turing machine.
- Turing machines can do anything that could be described as "purely mechanical".

Some questions about algorithmic unsolvability

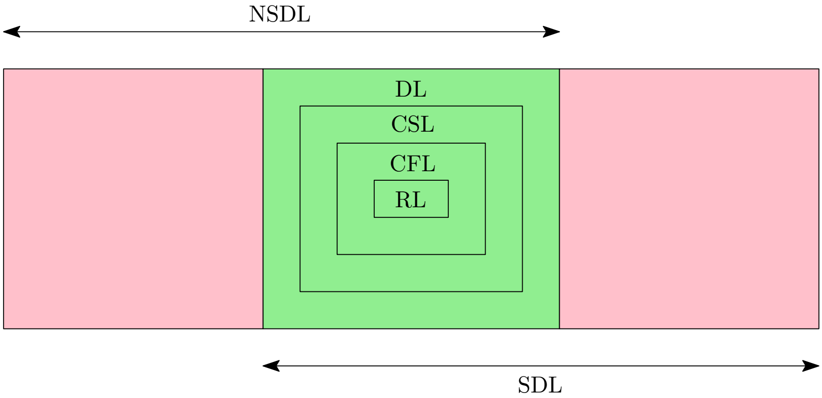
Some questions

- Why do we call Turing-decidable and Turing-semidecidable languages as recursive and recursively enumerable, respectively?
- What is the intuition behind algorithmic unsolvability?
- What is the relationship between recursive and recursively enumerable languages?
- What are the techniques to prove algorithmic unsolvability?
- What are some real-world problems that cannot be solved by human minds or real computers (from past, present, future)?

Chomsky hierarchy



Chomsky hierarchy



Some properties of languages

Properties

- If L is a Turing-decidable language, then $\overline{L}$ is a Turing-decidable language, too.
- If L is both Turing-semidecidable and Turing-undecidable (algorithmically unsolvable), then $\overline{L}$ is not Turing-semidecidable.

How can we prove algorithmic unsolvability?

Problem

- How can we prove that there are some computational problems that are algorithmically unsolvable?

Directions

A. Show that there are languages that are Turing-semidecidable but not Turing-decidable:

B. Show that there are languages that are not Turing semidecidable:

Approach	A	B
Show hypothetical examples	✓	✓
Prove that the set of decision problems/languages is bigger than the set of computer programs/TM's using uncountability	–	✓
Prove that the set of decision problems/languages is bigger than the set of computer programs/TM's using diagonalization	–	✓
Show real-world practical examples	✓	✓

Simple Turing machines that run forever

Problem

- Let's construct three non-halting Turing machines for $\Sigma = \{a\}$ and $\Gamma = \Sigma \cup \{\triangleright, \square\}$ with the following transition tables. Explain the working of these non-halting TM's M_1, M_2, and M_3.

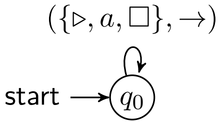
	Current symbol (Γ)		
Current state	$\triangleright$	a	$\square$
q_0	$(q_0, \rightarrow)$	$(q_0, \rightarrow)$	$(q_0, \rightarrow)$

	Current symbol (Γ)		
Current state	$\triangleright$	a	$\square$
q_0	$(q_0, \rightarrow)$	(q_0, a)	$(q_0, \square)$

	Current symbol (Γ)		
Current state	$\triangleright$	a	$\square$
q_0	$(q_0, \rightarrow)$	$(q_0, \leftarrow)$	$(q_0, \leftarrow)$

Simple Turing machines that run forever

Solution for M_1

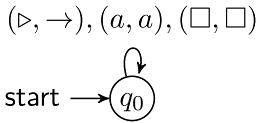


Time	State	Tape						
0	q_0	$\triangleright$	a	a	a	$\square$	$\square$	$\cdots$
1	q_0	$\triangleright$	a	a	a	$\square$	$\square$	$\cdots$
2	q_0	$\triangleright$	a	a	a	$\square$	$\square$	$\cdots$
3	q_0	$\triangleright$	a	a	a	$\square$	$\square$	$\cdots$
4	q_0	$\triangleright$	a	a	a	$\square$	$\square$	$\cdots$
5	q_0	$\triangleright$	a	a	a	$\square$	$\square$	$\cdots$

- The TM's tape head keeps moving right on the tape that has an infinite amount of memory.
- The TM never halts for any input string.

Simple Turing machines that run forever

Solution for M_2



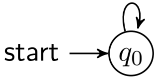
Time	State	Tape						
0	q_0	$\triangleright$	a	a	a	$\square$	$\square$	$\cdots$
1	q_0	$\triangleright$	a	a	a	$\square$	$\square$	$\cdots$
2	q_0	$\triangleright$	a	a	a	$\square$	$\square$	$\cdots$
3	q_0	$\triangleright$	a	a	a	$\square$	$\square$	$\cdots$
4	q_0	$\triangleright$	a	a	a	$\square$	$\square$	$\cdots$
5	q_0	$\triangleright$	a	a	a	$\square$	$\square$	$\cdots$

- The TM's tape head does not move, replaces the first character by itself, and stays in the same state.
- The TM never halts for any input string.

Simple Turing machines that run forever

Solution for M_3

$$(\triangleright, \rightarrow), (a, \leftarrow), (\Box, \leftarrow)$$



Time	State	Tape						
0	q_0	$\triangleright$	a	a	a	$\Box$	$\Box$	$\cdots$
1	q_0	$\triangleright$	a	a	a	$\Box$	$\Box$	$\cdots$
2	q_0	$\triangleright$	a	a	a	$\Box$	$\Box$	$\cdots$
3	q_0	$\triangleright$	a	a	a	$\Box$	$\Box$	$\cdots$
4	q_0	$\triangleright$	a	a	a	$\Box$	$\Box$	$\cdots$
5	q_0	$\triangleright$	a	a	a	$\Box$	$\Box$	$\cdots$

- The TM's tape head oscillates between the left end symbol and the first character.
- The TM never halts for any input string.

Most problems are algorithmically unsolvable

Problem

- Prove that the set of all decision problems or languages is bigger than the set of Turing machines or computer programs using countability/uncountability.

Solution

Prerequisite: Learn from
`https://www3.cs.stonybrook.edu/~pramod.ganapathi/doc/discrete-mathematics/Functions.pdf`

1. Prove that the set of decision problems is uncountable.
2. Prove that the number of Turing machines is countable.

This proves that most decision problems or languages are not Turing-semidecidable.

Most problems are algorithmically unsolvable

Solution (continued)

Part 1. Prove that the set of decision problems is uncountable.

- A decision problem can be represented as a number in $[0, 1]$.
- E.g.: The function below represents $0.0110001\ldots$.

Strings		$\{0, 1\}$
ϵ	$\longrightarrow$	0
0	$\longrightarrow$	1
1	$\longrightarrow$	1
00	$\longrightarrow$	0
01	$\longrightarrow$	0
10	$\longrightarrow$	0
11	$\longrightarrow$	1
$\vdots$		$\vdots$

- Set of all decision problems (or functions $\Sigma^* \rightarrow \{0, 1\}$) can be represented by the set of all real problems in $[0, 1]$.
- The set of all real numbers in $[0, 1]$ is uncountable.
- Hence, the set of all decision problems is uncountable.

Most problems are algorithmically unsolvable

Solution (continued)

Part 2. Prove that the set of all Turing machines is countable.

- A TM can be represented as a finite string.
- A finite string in ASCII can be represented as a binary string.
- The set of all TM's represents the set of all binary strings.
- The set of all binary strings is countable.
- Hence, the set of all TM's is countable.

Most problems are algorithmically unsolvable

Problem

- Prove that the set of all decision problems or languages is bigger than the set of Turing machines or computer programs using diagonalization.

Most problems are algorithmically unsolvable

Problem

- Prove that the set of all decision problems or languages is bigger than the set of Turing machines or computer programs using diagonalization.

Solution

- Suppose $M_1, M_2, M_3, \ldots$ are the TM's.
  Suppose $w_1, w_2, w_3, \ldots$ are strings in Σ^*.
- Construct a table with TM's as rows and strings as columns.

	Strings					
TM	w_1	w_2	w_3	w_4	w_5	$\cdots$
M_1	1	0	0	1	0	$\cdots$
M_2	0	0	1	0	0	$\cdots$
M_3	0	1	1	1	1	$\cdots$
M_4	1	1	0	1	0	$\cdots$
M_5	0	1	0	0	0	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Most problems are algorithmically unsolvable

Solution (continued)

- Construct a TM that accepts language $L_d = \{w_i \mid w_i \notin L(M_i)\}$ i.e., $L_d = d_1 d_2 d_3 \ldots$, where
$$d_i = \begin{cases} 1 & \text{if table}_{ii} = 0, \\ 0 & \text{if table}_{ii} = 1. \end{cases}$$
- For the example below, $L_d = 01001\ldots$

TM	Strings w_1	w_2	w_3	w_4	w_5	$\cdots$
M_1	1	0	0	1	0	$\cdots$
M_2	0	0	1	0	0	$\cdots$
M_3	0	1	1	1	1	$\cdots$
M_4	1	1	0	1	0	$\cdots$
M_5	0	1	0	0	0	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$
M_d	0	1	0	0	1	$\cdots$

Most problems are algorithmically unsolvable

Solution (continued)

Proof by contradiction.

- Suppose L_d is Turing-semidecidable. Then there exists TM M_k such that $L_d = L(M_k)$.
- Case 1. M_k accepts w_k.
  $\implies w_k \notin L_d$ $\quad$ ($\because$ defn. of L_d)
  $\implies w_k \notin L(M_k)$ $\quad$ ($\because L_d = L(M_k)$)
  $\implies M_k$ does not accept w_k $\quad$ ($\because$ defn. of $L(M_k)$)
- Case 2. M_k does not accept w_k.
  $\implies w_k \in L_d$ $\quad$ ($\because$ defn. of L_d)
  $\implies w_k \in L(M_k)$ $\quad$ ($\because L_d = L(M_k)$)
  $\implies M_k$ accepts w_k $\quad$ ($\because$ defn. of $L(M_k)$)
- Contradiction! Hence, L_d is not Turing-semidecidable.
  There is a decision problem or language that is not Turing-semidecidable.

Simulate program is algorithmically impossible

Problem

- Prove that it is impossible to design an algorithm to simulate the working of a given computer program on a given input string.

Simulate program is algorithmically impossible

Problem

- Prove that it is impossible to design an algorithm to simulate the working of a given computer program on a given input string.

Solution

Language $= \{\langle M, w \rangle \mid \text{TM } M \text{ accepts input string } w\}$
Let's call the hypothetical method as SIMULATE.

1. Prove that SIMULATE is Turing-semidecidable.
2. Prove that SIMULATE is algorithmically impossible.

Simulate program is algorithmically impossible

Solution (continued)

Part 1. Prove that SIMULATE is Turing-semidecidable.

- Consider the following generic procedure.

SIMULATE($\langle M, w \rangle$)

1. Simulate TM M on input string w
2. if M accepts w then
3. accept
4. elseif M rejects w then
5. reject

- Case 1: If M accepts w, then SIMULATE accepts.
  Case 2: If M rejects w, then SIMULATE rejects.
  Case 3: If M runs forever on w, then SIMULATE runs forever.
- So, SIMULATE is Turing-semidecidable.

Simulate program is algorithmically impossible

Solution (continued)

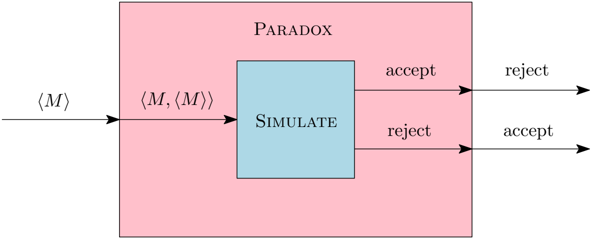
Part 2. Prove that SIMULATE is algorithmically impossible.

Proof by contradiction.

- Let's assume that SIMULATE is algorithmically possible i.e., SIMULATE always halts giving a correct answer.
- Then, we construct the PARADOX algorithm as follows.

PARADOX($\langle M \rangle$)

1. $result \leftarrow$ SIMULATE($\langle M, \langle M \rangle \rangle$)
2. if $result =$ accept then reject
3. elseif $result =$ reject then accept



Simulate program is algorithmically impossible

Solution (continued)

Part 2. Prove that SIMULATE is algorithmically impossible.

PARADOX($\langle M \rangle$)

Input: Source code of a computer program
Output: Accept or reject
Require: Invoke PARADOX($\langle$PARADOX$\rangle$)
1. $result \leftarrow$ SIMULATE($\langle M, \langle M \rangle \rangle$)
2. if $result =$ accept then reject
3. elseif $result =$ reject then accept

- Case 1. PARADOX accepts $\langle$PARADOX$\rangle$
  $\implies$ SIMULATE rejects $\langle$PARADOX, $\langle$PARADOX$\rangle\rangle$
  $\implies$ PARADOX rejects $\langle$PARADOX$\rangle$.
- Case 2. PARADOX rejects $\langle$PARADOX$\rangle$
  $\implies$ SIMULATE accepts $\langle$PARADOX, $\langle$PARADOX$\rangle\rangle$
  $\implies$ PARADOX accepts $\langle$PARADOX$\rangle$.
- Contradiction! Hence, SIMULATE is algorithmically impossible.

Halt program is algorithmically impossible

Problem

- Prove that it is impossible to design an algorithm to check if a given computer program halts on a given input string.

Halt program is algorithmically impossible

Problem

- Prove that it is impossible to design an algorithm to check if a given computer program halts on a given input string.

Solution

Language $= \{\langle M, w \rangle \mid$ TM M halts on input string $w\}$
Let's call the hypothetical method as $\textsc{Halt}$.

1. Prove that $\textsc{Halt}$ is Turing-semidecidable.
2. Prove that $\textsc{Halt}$ is algorithmically impossible.

Halt program is algorithmically impossible

Solution (continued)

Part 1. Prove that HALT is Turing-semidecidable.

- Consider the following generic procedure.

HALT($\langle M, w \rangle$)

1. Simulate TM M on input string w
2. if M accepts w or M rejects w then
3.   accept
4. else if M runs forever then
5.   reject

- Case 1: If M accepts w, then HALT accepts.
  Case 2: If M rejects w, then HALT accepts.
  Case 3: If M runs forever on w, then HALT runs forever.
- So, HALT is Turing-semidecidable.

Halt program is algorithmically impossible

Solution (continued)

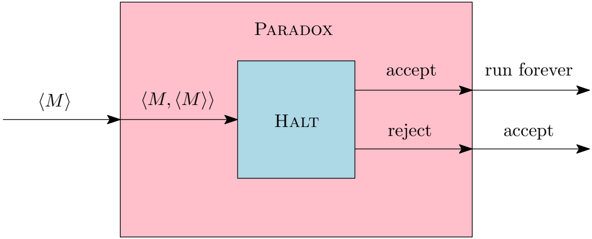
Part 2. Prove that HALT is algorithmically impossible.

Proof by contradiction.

- Let's assume that HALT is algorithmically possible i.e., HALT always halts giving a correct answer.
- Then, we construct the PARADOX algorithm as follows.

PARADOX($\langle M \rangle$)

1. $result \leftarrow$ HALT($\langle M, \langle M \rangle \rangle$)
2. if $result =$ accept then run forever
3. elseif $result =$ reject then accept



Halt program is algorithmically impossible

Solution (continued)

Part 2. Prove that HALT is algorithmically impossible.

PARADOX($\langle M \rangle$)

Input: Source code of a computer program
Output: Accept or reject
Require: Invoke PARADOX($\langle$PARADOX$\rangle$)
1. $result \leftarrow$ HALT($\langle M, \langle M \rangle \rangle$)
2. if $result =$ accept then run forever
3. elseif $result =$ reject then accept

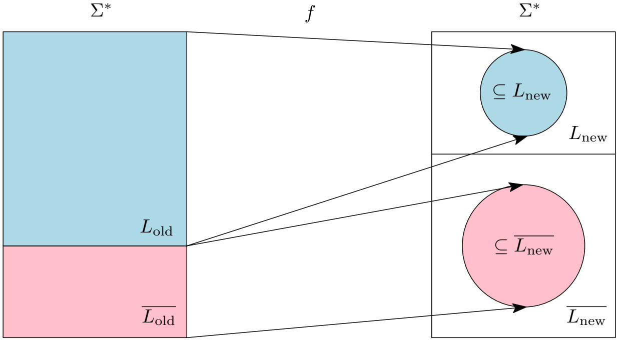
- Case 1. PARADOX accepts $\langle$PARADOX$\rangle$
  $\Longrightarrow$ HALT rejects $\langle$PARADOX, $\langle$PARADOX$\rangle\rangle$
  $\Longrightarrow$ PARADOX runs forever on $\langle$PARADOX$\rangle$.
- Case 2. PARADOX runs forever on $\langle$PARADOX$\rangle$
  $\Longrightarrow$ HALT accepts $\langle$PARADOX, $\langle$PARADOX$\rangle\rangle$
  $\Longrightarrow$ PARADOX accepts $\langle$PARADOX$\rangle$.
- Contradiction! Hence, HALT is algorithmically impossible.

What is reduction?

Definition

- Given two languages $L_{\text{old}}, L_{\text{new}} \in \Sigma^*$, we say that L_{old} reduces to L_{new}, meaning L_{new} is at least as hard as L_{old}, denoted as $L_{\text{old}} \leq_m L_{\text{new}}$ if there exists a computable function f such that for all $x \in \Sigma^*$

$$x \in L_{\text{old}} \iff f(x) \in L_{\text{new}}$$



What is reduction?

Properties

- Notation. In $L_{\text{old}} \leq_m L_{\text{new}}$,
  the 'm' letter in $\leq_m$ represents many-to-one function.
- Meaning. If $L_{\text{old}} \leq_m L_{\text{new}}$, then
  L_{new} is at least as hard as L_{old}.
- Intuition. If $L_{\text{old}} \leq_m L_{\text{new}}$, then
  the reduction should turn:
  - Instance of L_{old} with yes to instance of L_{new} with yes.
  - Instance of L_{old} with no to instance of L_{new} with no.
- Consequences. If $L_{\text{old}} \leq_m L_{\text{new}}$, then
  If L_{old} is undecidable, then so is L_{new}.
  If L_{old} is not Turing-semidecidable, then so is L_{new}.
  If L_{new} is decidable, then so is L_{old}.

Halt program is algorithmically impossible

Problem

- Prove that it is impossible to design an algorithm to check if a given computer program halts on a given input string, using reduction.

Halt program is algorithmically impossible

Problem

- Prove that it is impossible to design an algorithm to check if a given computer program halts on a given input string, using reduction.

Solution

- $L_{\textsf{sim}} = \{\langle M, w \rangle \mid \text{TM } M \text{ accepts input string } w\}$
  $L_{\textsf{halt}} = \{\langle M, w \rangle \mid \text{TM } M \text{ halts on input string } w\}$
- Proof by contradiction and proof by reduction.
  Let's call the hypothetical method as Halt.
  We show that if Halt is algorithmically possible,
  then Simulate is algorithmically possible, too.

Halt program is algorithmically impossible

Solution (continued)

Prove that HALT is algorithmically impossible.

- Let's assume that HALT is algorithmically possible. Then, we construct the SIMULATE algorithm as follows.

SIMULATE($\langle M, w \rangle$)

1. $result \leftarrow$ HALT($\langle M, w \rangle$)
2. if $result =$ reject then reject ▷ M runs forever on w
3. elseif $result =$ accept then
4. Simulate M on w
5. if M accepts w then accept ▷ M accepts w
6. elseif M rejects w then reject ▷ M rejects w

- If HALT is an algorithm, then SIMULATE is an algorithm too, which terminates in all cases.
- We know that SIMULATE is algorithmically impossible. Hence, HALT is algorithmically impossible, too.

Hofstadter's ac puzzle

Problem

Starting with the string ab, can you derive ac, using the following productions?

1. Add a c to the end of any string ending in b.
   i.e., $xb \to xbc$
2. Double the string after the first character a.
   i.e., $ax \to axx$
3. Replace any bbb with a c.
   i.e., $xbbby \to xcy$
4. Remove any cc.
   i.e., $xccy \to xy$

Hofstadter's ac puzzle

Solution (core idea)

- The problem cannot be solved.
- Invariant: $n = (\#b$'s in a string$)$ is not divisible by 3.
- The invariant is true for the starting string ab.
  The invariant is true for all strings derivable from ab.
- The invariant is false for ac.
- Hence, the string ac cannot be derived from the string ab.

Hofstadter's ac puzzle

Solution (continued)

- [Starting string.] For the starting string ab, $n = 1$.
  Hence, the invariant is true.
  [Rules 1 and 4.] The rules do not change $\#b$'s.
  Hence, the invariant is true.
  [Rule 2.] Doubling a number that is not divisible by 3 does not make it divisible by 3. Hence, the invariant is true.
  [Rule 3.] Subtracting 3 from a number that is not divisible by 3 does not make it divisible by 3. Hence, the invariant is true.
- The desired string ac cannot be derived because $n = 0$.
  And 0 is divisble by 3.

- Reference: `https://en.wikipedia.org/wiki/MU_puzzle`

Decision problems involving TM's

Decision problems

Algorithmically solvable.

- Given a TM M, does M have at least 481 states?
- Given a TM M, does M take more than 481 steps on input ϵ?
- Given a TM M, does M take more than 481 steps on some input?
- Given a TM M, does M take more than 481 steps on all inputs?
- Given a TM M, does M ever move its head more than 481 tape cells away from the left endmarker on input ϵ?

Decision problems involving TM's

Decision problems

Algorithmically unsolvable.

- Given a TM M and an input string w, is $w \in L(M)$?
- Given a TM M, is $L(M)$ nonempty?
- Given a TM M, is $L(M) = \Sigma^*$?
- Given a TM M, is $L(M)$ a regular language?
- Given a TM M, is $L(M)$ a CFL?
- Given a TM M, is $L(M)$ a recursive language?
- Given a TM M, is $L(M)$ recursively enumerable?
- Given two TM's M_1 and M_2, is $L(M_1) = L(M_2)$?
- Given two TM's M_1 and M_2, is $L(M_1) \subseteq L(M_2)$?
- Given two TM's M_1 and M_2, is $L(M_1) \cap L(M_2)$ nonempty?
- Given a TM M and an input string w, does M use a finite amount of tape?

Post correspondence problem (PCP)

Problem

- Given the set of dominos, is it possible to list these dominos (repetitions permitted) so that the string of symbols on top is the same as the string of symbols on the bottom?

$$(D_1, D_2, D_3, D_4) = \left(\left[\frac{b}{ca}\right], \left[\frac{a}{ab}\right], \left[\frac{ca}{a}\right], \left[\frac{abc}{c}\right] \right)$$

Post correspondence problem (PCP)

Problem

- Given the set of dominos, is it possible to list these dominos (repetitions permitted) so that the string of symbols on top is the same as the string of symbols on the bottom?

$$(D_1, D_2, D_3, D_4) = \left(\begin{bmatrix} b \\ ca \end{bmatrix}, \begin{bmatrix} a \\ ab \end{bmatrix}, \begin{bmatrix} ca \\ a \end{bmatrix}, \begin{bmatrix} abc \\ c \end{bmatrix} \right)$$

Solution

- Yes!
- A solution: $[D_2 D_1 D_3 D_2 D_4]$.

$$\begin{bmatrix} a \\ ab \end{bmatrix}\begin{bmatrix} b \\ ca \end{bmatrix}\begin{bmatrix} ca \\ a \end{bmatrix}\begin{bmatrix} a \\ ab \end{bmatrix}\begin{bmatrix} abc \\ c \end{bmatrix} = \begin{bmatrix} abcaaabc \\ abcaaabc \end{bmatrix} = \text{match}$$

Post correspondence problem (PCP)

Problem

- Given the set of dominos, is it possible to list these dominos (repetitions permitted) so that the string of symbols on top is the same as the string of symbols on the bottom?

$$(D_1, D_2, D_3) = \left(\begin{bmatrix} abc \\ ab \end{bmatrix}, \begin{bmatrix} ca \\ a \end{bmatrix}, \begin{bmatrix} acc \\ ba \end{bmatrix} \right)$$

Post correspondence problem (PCP)

Problem

- Given the set of dominos, is it possible to list these dominos (repetitions permitted) so that the string of symbols on top is the same as the string of symbols on the bottom?

$$(D_1, D_2, D_3) = \left(\begin{bmatrix} abc \\ ab \end{bmatrix}, \begin{bmatrix} ca \\ a \end{bmatrix}, \begin{bmatrix} acc \\ ba \end{bmatrix} \right)$$

Solution

- No!
  Every top string is greater than its bottom string.

Post correspondence problem (PCP)

Problem

Given the set of dominos, is there a match?

- $(D_1, D_2, D_3) = \left(\begin{bmatrix} a \\ baa \end{bmatrix}, \begin{bmatrix} ab \\ aa \end{bmatrix}, \begin{bmatrix} bba \\ bb \end{bmatrix} \right)$
- $(D_1, D_2, D_3, D_4) = \left(\begin{bmatrix} b \\ bab \end{bmatrix}, \begin{bmatrix} abb \\ b \end{bmatrix}, \begin{bmatrix} aba \\ a \end{bmatrix}, \begin{bmatrix} bbaaa \\ babaaa \end{bmatrix} \right)$
- $(D_1, D_2, D_3) = \left(\begin{bmatrix} bb \\ b \end{bmatrix}, \begin{bmatrix} ab \\ ba \end{bmatrix}, \begin{bmatrix} c \\ bc \end{bmatrix} \right)$
- $(D_1, D_2, D_3) = \left(\begin{bmatrix} bbab \\ b \end{bmatrix}, \begin{bmatrix} abba \\ bb \end{bmatrix}, \begin{bmatrix} b \\ bba \end{bmatrix} \right)$

Post correspondence problem (PCP)

Problem

Given the set of dominos, is there a match?

- $(D_1, D_2, D_3) = \left(\frac{a}{baa}, \frac{ab}{aa}, \frac{bba}{bb} \right)$
- $(D_1, D_2, D_3, D_4) = \left(\frac{b}{bab}, \frac{abb}{b}, \frac{aba}{a}, \frac{bbaaa}{babaaa} \right)$
- $(D_1, D_2, D_3) = \left(\frac{bb}{b}, \frac{ab}{ba}, \frac{c}{bc} \right)$
- $(D_1, D_2, D_3) = \left(\frac{bbab}{b}, \frac{abba}{bb}, \frac{b}{bba} \right)$

Solution

- Yes! Infinite solutions: $[D_3 D_2 D_3 D_1]^+$.
- Yes! Infinite solutions: $[D_1 D_2 D_3 D_1]^+$
- Yes! Infinite solutions: $[D_1 D_2^+ D_3]^+$.
- Yes! Shortest solution has length 252.

Post correspondence problem (PCP)

Problem

Given the set of dominos, is there a match?

- $(D_1, D_2, D_3) = \left(\begin{bmatrix} bb \\ abb \end{bmatrix}, \begin{bmatrix} ab \\ a \end{bmatrix}, \begin{bmatrix} aab \\ bba \end{bmatrix} \right)$

Post correspondence problem (PCP)

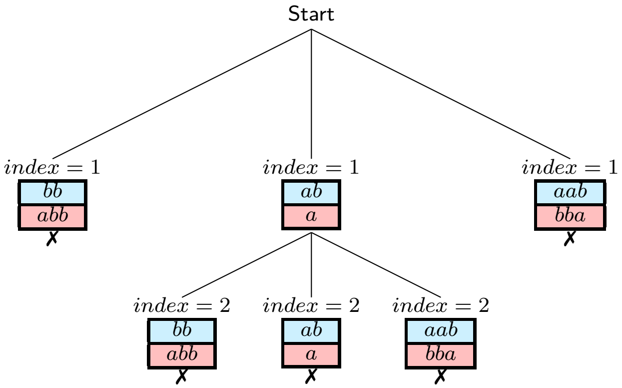
Problem

Given the set of dominos, is there a match?

- $(D_1, D_2, D_3) = \left(\frac{bb}{abb}, \frac{ab}{a}, \frac{aab}{bba} \right)$

Solution

- No!



Post correspondence problem (PCP)

Problem

- Discovered by Emil Post in 1940's.
- You are given the set P of dominos
  $$\left(\begin{bmatrix} t_1 \\ b_1 \end{bmatrix}, \begin{bmatrix} t_2 \\ b_2 \end{bmatrix}, \ \dots, \begin{bmatrix} t_k \\ b_k \end{bmatrix} \right)$$
  A match is a sequence $i_1 i_2 \dots i_\ell$, where
  $t_{i_1} t_{i_2} \cdots t_{i_\ell} = b_{i_1} b_{i_2} \cdots b_{i_\ell}$.
- Is there an algorithm to determine if P has a match?

Post correspondence problem (PCP)

Problem

- Discovered by Emil Post in 1940's.
- You are given the set P of dominos
  $$\left(\left[\frac{t_1}{b_1}\right], \left[\frac{t_2}{b_2}\right], \ldots, \left[\frac{t_k}{b_k}\right] \right)$$
  A match is a sequence $i_1 i_2 \ldots i_\ell$, where
  $t_{i_1} t_{i_2} \cdots t_{i_\ell} = b_{i_1} b_{i_2} \cdots b_{i_\ell}$.
- Is there an algorithm to determine if P has a match?

Solution

- No! The problem is algorithmically unsolvable.
- Let PCP $= \{\langle P \rangle \mid P$ is a domino set that has a match$\}$.
  PCP is not Turing-decidable.