Theory of Computation

Matteo Alberici

February 2022

What are the fundamental capabilities and limitations of computers?

Contents

1 Introduction

1.1 Automata, Computability, and Complexity

The theory of computation consists of three areas:

- **Complexity Theory**

  What makes problems computationally hard or easy?

- **Computability Theory**

  What problems can(not) computers solve?

- **Automata Theory**

  Deals with defining mathematical models of computation

1.2 Mathematical Notions and Terminology

1.2.1 Sets

A **set** is a group of unique unordered **elements** seen as a unit:

$$S = \{1,\ ciao,\ 3\},\ T = \{\emptyset\}$$

Set A is a **subset** of set B if every element of A is a member of B:

$$A \subseteq B$$

Moreover, A is a **proper subset** of B if A is not equal to B:

$$A \subset B$$

The **union** of sets A and B is the set containing all the elements in A and all those in B:

$$A \cup B$$

The **intersection** of sets A and B is the set containing the elements that are in both A and B:

$$A \cap B$$

The **complement** of set A is the set of all elements not in A:

$$\bar{A}$$

The **power set** $P(A)$ is the set of all the subsets of A.
The **Cartesian product** of sets A and B is the set of all ordered pairs wherein the first element is a member of A and the second one is a member of B:

$$A \times B$$

1.2.2 Sequences and Tuples

A **sequence** is an ordered list of objects:

$$(1, 2, 3)$$

A finite sequence is a **tuple**: a **k-tuple** contains k elements.

1.2.3 Graphs

A **graph** is a set of **nodes** connected with **edges**. We can label nodes and edges, obtaining a **labeled graph**. In a **directed graph**, edges have directions.
A **path** is a sequence of nodes connected by edges.

1.2.4 Strings and Languages

An **alphabet** Σ is a nonempty finite set of **symbols**:

$$\Sigma = \{a, b, c, ..., z\}$$

A **string over an alphabet** is a sequence of symbols from Σ:

$$abracadabra \text{ is a string over } \Sigma$$

The **empty string** ε is the string with a length of 0.
Given two strings x and y, the **concatenation** xy is obtained by appending y (suffix) to the end of x (prefix). A **language** L is a set of strings over a given alphabet.

1.2.5 Boolean Logic

Boolean logic is a mathematical system built around the **Boolean values** true and false, manipulated through **Boolean operations**:

- **Negation**: returns the opposite value (NOT - $\neg$)
- **Conjunction**: returns 1 if both values are 1 (AND - $\wedge$)
- **Disjunction**: returns 1 if either of the values is 1 (OR - $\vee$)
- **Exclusive or**: returns 1 if only one value is 1 (XOR - $\oplus$)
- **Equality**: returns 1 if both values have the same value ($\iff$)
- **Implication**: returns 0 iff the first operand is 1 and the second one is 0 ($\implies$)

The **distributive law** for AND and OR states that:

- $P \wedge (Q \vee R) \iff (P \wedge Q) \vee (P \wedge R)$
- $P \vee (Q \wedge R) \iff (P \vee Q) \wedge (P \vee R)$

1.3 Regular Languages

The theory of computation uses an idealized computer called a **computational model**.

1.3.1 Finite Automata

The simplest computational model is the **finite automaton** (**FA**). It receives a string in input and, by processing it, either **accepts** or **rejects** it. An FA is defined by the following **5-tuple**:

$$A = (Q, \Sigma, \delta, q_0, F), \text{ where}$$

- Q is the finite set of states
- Σ is the finite input alphabet
- $\delta : Q \times \Sigma \to Q$ is the transition function
- $q_0 \in Q$ is the initial state
- $F \subseteq Q$ is the accepting state

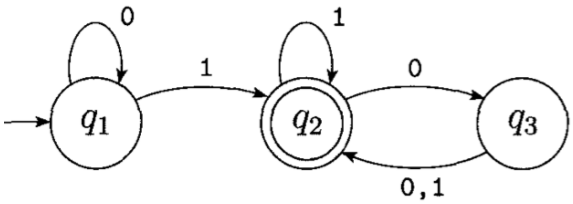


Figure 1: DFA example

A language is **regular** if an FA recognizes it. Given the set A of all strings accepted by a machine M, then A is the **language of machine** M:

$$L(M) = A$$

There are three **regular operations** concerning regular languages:

- **Union**: $A \cup B = \{x \mid x \in A \text{ or } x \in B\}$
- **Concatenation**: $AB = \{xy \mid x \in A \text{ and } y \in B\}$
- **Star**: $A* = \{x_1, x_2, ..., x_k \mid k \geq 0 \text{ and } x_i \in A\}$

1.4 Nondeterminism

In a **nondeterministic finite automaton** (**NFA**), several choices may exist for the next state:

- Multiple ways to proceed: the machine **splits** into copies
- No ways to proceed: the copy **dies**
- A copy is in an accepting state: the string is accepted
- An ε transition exists: the machine **splits immediately**

An NFA is defined by the following **5-tuple**:

$$FA = (Q, \Sigma, \delta, q_0, F), \text{ where}$$

- Q is the set of states
- Σ is the input alphabet
- $\delta : Q \times \Sigma_\varepsilon \to P(Q)$ is the **transition function**
- $q_0 \in Q$ is the **initial state**
- $F \subset Q$ is the **accepting state**

1.4.1 From NFA to FA

There exist an NFA for each FA:

1. Draw q_0 and the states reached from it along with an ε transition
2. Draw at least one accepting state
3. Draw all transitions
    - No label: the transition leads to the **dead state** $\emptyset$
    - ε transition: the transition leads to the subset of states reached along with it

1.5 Regular Expressions

Regular expressions describe languages by using regular operations:

$$(0 \cup 1) \rightarrow (\{0\} \cup \{1\}) \rightarrow \{0, 1\} = \Sigma$$
$$\Sigma 0 \rightarrow \{00, 10\}$$
$$0* \rightarrow \{0*\} \rightarrow \{\varepsilon, 0, 00, ...\}$$

R is a regular expression over an alphabet Σ if it is at least one of:

1. a, for some a in Σ
2. ε
3. $\emptyset$
4. $R_1 \cup R_2$, where R_1 and R_2 are regular expressions
5. $R_1 R_2$, where R_1 and R_2 are regular expressions
6. R_1*, where R_1 is a regular expression

1.6 Nonregular Languages

To prove if a language is regular or not, we use the **pumping lemma**, which states that if L is a regular language, then there is a **pumping length** p such that, if w is a string in L of at least length p, then w may be divided into three pieces, x, y, and z, satisfying the following conditions:

1. $xy^i z \in L$ for each $i \geq 0$
2. $|y| > 0$
3. $|xy| \leq p$

We must choose a string $s \in L$ of at least length p, then show that no possible division satisfies all conditions.
Let us analyze an example with the following language:

$$L = \{w | w = 0^n 1^n \text{ for } n \geq 0\}$$

1. Assume language L is regular
2. A pumping length p must exist such that all strings in L of at least length p can be pumped
3. Choose a string w that cannot be pumped:

$$w = 0^p 1^p \rightarrow 00...011...1$$

4. Show that no division into xyz satisfying all conditions exists:
   (a) 3^{rd} condition: y must contain only 0s
   (b) 2^{nd} condition: y must contain at least one 0
   (c) 1^{st} condition: repeating 0s results in a string $\notin L$

1.7 Context-Free Languages

A **context-free grammar** (**CFG**) is a collection of **substitution rules** which are composed of a **variable**, a symbol, and a string, which again is composed of variables or **terminals**, built as follows:

1. Write the **start variable**:
2. Find a rule starting with a known variable
3. Replace the variable with the correspondent string

$$\begin{aligned} &A \rightarrow 0A1 && A \rightarrow 0A1 \rightarrow 00A11 \rightarrow 00B11 \rightarrow 00Z11 \\ &A \rightarrow B && A \rightarrow B \rightarrow Z \\ &B \rightarrow Z && L = \{0^n Z 1^n \mid n \geq 0\} \end{aligned}$$

A language generated by a CFG is a **context-free language** and is defined by the following **4-tuple**:

$$L = (V, \Sigma, R, S), \text{ where}$$

- V is the set of variables
- Σ is the set of terminals
- R is the set of rules
- $S \in V$ is the start variable

The following are some properties of CGFs:

1. Break a language into simpler languages:
   - Union:

$$S_1 \to 0S_1 1 \mid \varepsilon$$
$$S_2 \to 0S_2 1 \mid 01$$
$$S \to S_1 \mid S_2$$

   - Concatenation:

$$S_1 \to 0S_1 1 \mid \varepsilon$$
$$S_2 \to 0S_2 1 \mid 01$$
$$S \to S_1 S_2$$

2. Draw an FA and convert it to its CFG:
   (a) Design a variable R_i for every state Q_i
   (b) Add a rule $R_i \to aR_j$ for the $a-$transition from Q_i to Q_j
   (c) Add a rule $R_i \to \varepsilon$ if Q_i is a final state
   (d) Set the variable R_0 for the initial state Q_0

A sequence of substitutions is called a **derivation**. A string is **ambiguously derived** if it has many derivations; thus, a grammar is **ambiguous** if it generates at least one ambiguous string.

1.8 Pushdown Automata

A **pushdown automaton** (**PDA**) is an NFA equipped with a **stack** giving it an unbounded memory needed for some nonregular languages. A PDA is defined by the following **6-tuple**:

$$P = (Q, \Sigma, \Gamma, \delta, q_0, F), \text{ where}$$

- Q is the set of states
- Σ is the alphabet
- Γ is the **stack alphabet**
- $\delta : Q_i \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to P(Q_j \times \Gamma_\varepsilon)$ is the transition function, where:
  - Q_i is a state
  - Σ_ε is an input symbol
  - Γ is a stack symbol read
  - Q_j is the state reached
  - Γ_ε is the new symbol on the stack
- q_0 is the initial state
- $F \subseteq Q$ is the set of accepting states

The **transition labels** are written as follows:

$$A, B \to C,$$

meaning that if A is read from the input string and B is on top of the stack, then pop B and write C.

2 Introduction to Turing Machines

The **Turing machine** (**TM**) is a reasoning machine proposed by Alan Turing to compare problems and solutions. TMs process an infinite **tape** writing and reading through a **controller head** while moving left or right. Strings could be accepted or rejected before the end of the input is reached.

2.1 Turing Machine Definition

A Turing machine is defined by the following **7-tuple**:

$$M = (Q, \Sigma, G, \delta, q_0, q_{accept}, q_{reject}), \text{ where}$$

- Q is the finite set of states
- Σ is the input alphabet
- G is the **tape alphabet** and contains the blank symbol
- $\delta : Q \times G \to Q \times G \times \{L, R\}$ is the transition function
- q_0 is the initial state
- q_{accept} is the accepting state
- q_{reject} is the **rejecting state**

The **language of a TM** is the set of strings accepted by that TM. A TM **decides** a language L if it accepts all the strings $\in L$ and rejects all the strings $\notin L$; thus, a language is **Turing-decidable** if some TM decides it.
A Turing machine **recognizes** a language L if it accepts all and only those strings $\in L$; thus, a language is **Turing-recognizable** if some TM recognizes it.
Every Turing-decidable language is also Turing-recognizable. If L is a Turing-recognizable language, then the TM that recognizes L could suffer from the **halting problem** and may not find a solution while processing a string $\notin L$. In contrast, the problem does not concern Turing-decidable languages.

2.2 Turing Machines Configurations

A **configuration** shows the current state of a Turing machine:

$$C_1 = (uqv)$$

In C_1, the machine is in state q with strings u and v on the tape and the controller head on the first symbol of v. A configuration C_i yields another configuration C_j if the machine can legally go from C_i to C_j in a single step.
The **start configuration** q_0w indicates that the machine is in q_0 with the head at the leftmost position. The **accepting** and the **rejecting configurations** are called the **halting configurations** and do not yield further ones.
A machine accepts a string w if there exists a sequence of configurations $C_1, ..., C_k$ such that:

1. C_1 is the start configuration
2. C_i yields C_{i+1} with $(1 \leq i \leq k-1)$
3. C_k is an accepting configuration

3 Turing Machines Design and Variants

A Turing machine and its variants recognize the same languages: the invariance of a model to changes is called its **robustness**.

3.1 Turing Machine Variants

3.1.1 Stay Put TMs

In **stay-put Turing machines**, the controller's head can stay in the same position after reading or writing:

$$\delta : Q \times G \to Q \times G \times \{L, R, S\},$$

3.1.2 Multitape TMs

We can convert a single-tape Turing machine (STM) to a **multi-tape Turing machine** (**MTM**) to improve efficiency and simplify decision-making. Given the number of tapes k, we define the following transition function:

$$\delta : Q \times G^k \to Q \times G^k \times \{L, R, S\}^k,$$

which allows for reading, writing, and moving the heads on many tapes simultaneously. In order to convert an MTM to an STM, we must add a **delimiter** "#" to separate the contents of the different tapes and a mark "$\dot{}$" to remember the location of the heads.

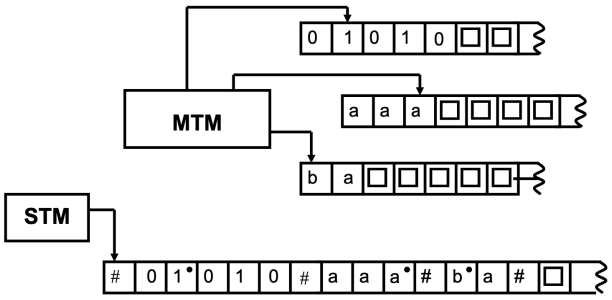


Figure 2: MTM to STM example

3.2 Nondeterministic TMs

In **nondeterministic Turing machines** (**NTM**), the transition function δ is the following:

$$\delta : Q \times G \to 2^{Q \times G \times \{L,R\}}$$

We obtain a tree whose branches correspond to possibilities: the machine accepts if some branch leads to an accepting state.
Every NTM has an equivalent deterministic TM (DTM): all branches are traversed with the Breadth-First Search algorithm (BFS), accepting if an accepting state is found; otherwise, the machine does not terminate.

3.2.1 Enumerator

An **enumerator** is a model which prints out a collection of strings: those strings form an **enumerated language**. A language is Turing-recognizable if some enumerator enumerates it.

4 Computability Theory

A problem is **solvable** if there exists a **decider** that decides it. A decider is a TM that always halts with any input.

4.1 Acceptance Testing

4.1.1 DFA Acceptance Testing

The **DFA acceptance testing** identifies a decider for A_{DFA}:

$$A_{DFA} = \{\langle B, \omega \rangle \mid B \text{ is a DFA that accepts string } \omega\}$$

We simulate B on ω keeping track of the DFA by writing information on the tape: if the simulation ends in an accepting state, then the machine accepts; otherwise, it rejects.

4.1.2 NFA Acceptance Testing

The **NFA acceptance testing** identifies a decider for A_{NFA}:

$$A_{NFA} = \{\langle B, \omega \rangle \mid B \text{ is a DFA that accepts string } \omega\}$$

We convert B to a DFA C and simulate it on ω keeping track of the NDFA by writing information on the tape: if the simulation ends in an accepting state, then the machine accepts; otherwise, it rejects.

4.1.3 REX Acceptance Testing

The **REX acceptance testing** determines if a regular expression generates a given string:

$$A_{REX} = \{\langle R, \omega \rangle \mid R \text{ is a REX that generates string } \omega\}$$

4.2 Emptiness Testing

The **emptiness testing** identifies a decider for E_{DFA}:

$$E_{DFA} = \{\langle A \rangle \mid A \text{ is a DFA and } L(A) = \emptyset\}$$

After marking the start state, we mark any state that has an in-transition from a marked state: if no accept state is marked, then the machine accepts; otherwise, it rejects.

4.3 Equivalence Testing

The **equivalence testing** identifies a decider for EQ_{DFA}:

$$EQ_{DFA} = \{\langle A, B\rangle \mid A \text{ and } B \text{ are DFAs and } L(A) = L(B)\}$$

We design an automaton C with the **symmetric difference feature**, meaning it accepts nothing when the languages are the same:

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

After marking the starting state of C, we mark any state that has an in-transition from a marked state: if no accept state is marked, then the machine accepts; otherwise, it rejects.

4.4 Decidability of CFLs

4.4.1 CFG Acceptance Testing

We must prove the decidability of language A_{CFG}.
The decidability results for CFGs are the same for PDAs.

4.4.2 CFG Emptiness Testing

We must prove the decidability of language E_{CFG}.
Given the CFG $\langle G\rangle$ in input, we mark all terminals in G, and then we mark any variable A where G has a rule $A \to U_1U_2...U_k$ with each symbol marked: if the start variable is not marked, the machine accepts; otherwise, it rejects.

4.4.3 CFG Equivalence Testing

We must prove the decidability of language EQ_{CFG}. We cannot reuse the symmetric difference idea since CFLs are not closed under complementation or intersection.

5 Undecidable Languages

A language is **undecidable** if there is not a decider for the language. A recognizable language can also be undecidable.

5.1 Membership Problem

The **membership problem** is unsolvable and is defined as follows:

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

Let us suppose that A_{TM} is decidable, then there exists a decider H for the language on input $\langle M, w\rangle$. Let L be a Turing-recognizable language accepted by the TM N, then let us build a decider for L on input $\langle N\rangle$.

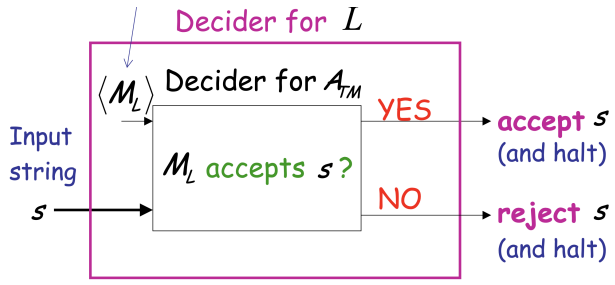


Figure 3: Decider for language L

Since L is decidable and arbitrarily chosen, then every Turing-recognizable language is decidable, which is a contradiction.

5.2 Halting Problem

5.2.1 Diagonalization

Let R be a relation between A and B, then R may have any of the following properties:

- $\forall x \in A$, there exists at least one $y \in B$, with $x, y \in R$
- $\forall x \in A$, there exists at most one $y \in B$, with $x, y \in R$
- $\forall y \in B$, there exists at least one $x \in A$, with $x, y \in R$
- $\forall y \in B$, there exists at most one $x \in A$, with $x, y \in R$

If a function $f : A \to B$ is a **correspondence**, meaning it is **one-to-one** and **onto**, then f has all the essential properties and establishes a "pairing" of the elements in A with those in B. Thus:

- **Existence** - R is onto if:

  $\forall y \in B$, there exists at least one $x \in A$, with $(x, y) \in R$

- **Uniqueness** - R is one-to-one if:

  $\forall y \in B$, there exists at most one $x \in A$, with $(x, y) \in R$

For functions from $\mathbb{R}$ to $\mathbb{R}$, the **horizontal line test** lets us see whether a function is one-to-one or onto or both. The horizontal line $y = b$ crosses the graph of $y = f(x)$ at the points where $f(x) = b$: f is one-to-one if no horizontal line crosses the graph more than once, and onto if every line crosses the graph at least once.
Two sets have the same number of elements if there exists a one-to-one and onto correspondence between them.

5.2.2 Countability

A set is **countable** if it is either finite or it is one-to-one and onto with the set of natural numbers $\mathbb{N}$. There are uncountably many languages and countably many TMs.

Theorem: the set of real numbers is uncountable.

Proof - Suppose there exists a correspondence between $\mathbb{N}$ and $\mathbb{R}$: if an x in $\mathbb{R}$ is not paired with $n \in \mathbb{N}$, then it leads to a contradiction.

5.2.3 Halting Problem Proof

The **halting problem** is unsolvable and occurs when a TM loops while simulating M on ω:

$$HALT_{TM} = \{\langle M, w\rangle \ : \ M \text{ is a TM that halts on input string } w\}$$

Let us suppose that $HALT_{TM}$ is decidable, then there exists a decider H for the language on input $\langle M, w\rangle$. Let us build a machine $\bar{H}$ by using H: $\bar{H}$ loops forever if M halts on input w; otherwise, it halts. Let us build a machine F: if M halts on input $\langle M\rangle$, then it loops forever; otherwise, it halts.

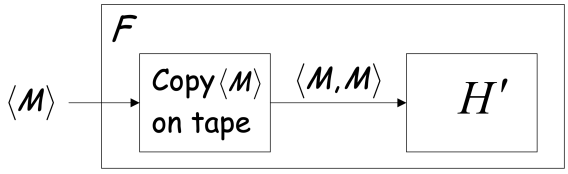


Figure 4: Machine F representation

Let us run F on input $\langle F \rangle$: if F halts, then F loops forever; otherwise, it halts. This leads to a contradiction.
By tabulating the results:

1. H accepts $\langle M, w \rangle$ when M accepts w
2. F rejects $\langle M \rangle$ when M accepts $\langle M \rangle$
3. F rejects $\langle F \rangle$ when F accepts $\langle F \rangle$

6 Reduction Approach

The **reduction approach** consists of reducing a problem X to a problem Y: if problem Y is solvable, then also problem X is solvable. We map an entity w in a domain D to a **result region** S:

$$w \in D \ \rightarrow \ f(w) \in S$$

6.1 Function Computability

A function f is **computable** if there exists a Turing machine M that halts on every input w with $f(w)$ on the tape:

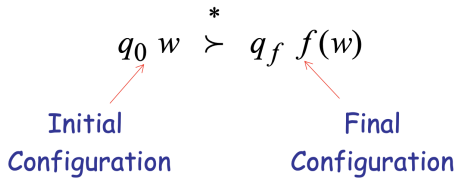


Figure 5: Function computability for all $w \in D$

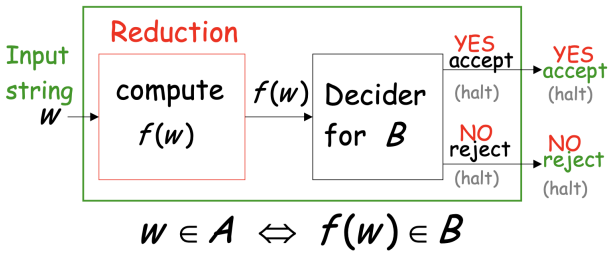
6.2 Reduction Definition

Language A is reduced to language B through f as follows:

$$A \leq_m B, \quad w \in A \iff f(w) \in B, \ \forall w$$

Reduction Theorem: if a language A is reduced to a language B which is decidable, then language A is decidable.

Proof - Let us build a decider for language A by using the decider for language B:



Negated Reduction Theorem: if an undecidable language A is reduced to a language B, then language B is undecidable.

Proof - Let us suppose that language B is decidable, then build the decider for language A by using the decider for language B: if B is decidable, also A is decidable, which is a contradiction.

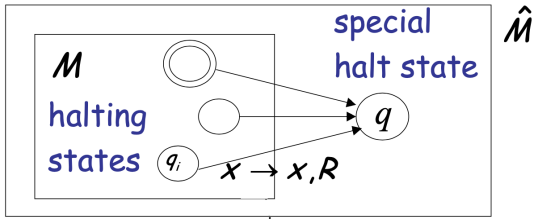
6.3 State-entry Problem

Does a machine M enter state q while processing an input string w?

$$\text{STATE}_{TM} = \{\langle M, w, q\rangle : M \text{ enters state } q \text{ on input string } w\}$$

State Theorem: the language STATE_{TM} is undecidable.

Proof - Let us reduce language HALT_{TM} to language STATE_{TM}: if language STATE_{TM} is decidable, then language HALT_{TM} is decidable, which is a contradiction. We construct $\langle \hat{M}\rangle$ from $\langle M\rangle$, designing a transition for every unused symbol x of q_i:



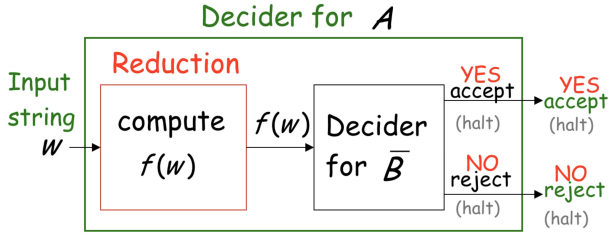
$\hat{M}$

$\langle M\rangle$ halts on the input string w; therefore, $\langle \hat{M}\rangle$ halts on state q on the input string w. Equivalently:

$$\langle M, w\rangle \in \text{HALT}_{TM} \iff \langle \hat{M}, w, q\rangle \in \text{STATE}_{TM}$$

Reduction Theorem 3: if an undecidable language A is reduced to $\bar{B}$, then B is undecidable.

Proof - Let us suppose that B is decidable. Then, also $\bar{B}$ is decidable. Now we build the decider for A by using the decider for $\bar{B}$, which leads to a contradiction:



$w \in A \Leftrightarrow f(w) \in \bar{B}$

6.4 Blank-tape Halting Problem

Does a machine M halt if it starts with a blank tape?

$$\text{BLANK}_{TM} = \{\langle M \rangle \ : \ M \ \text{ halts if it starts on blank tape}\}$$

In order to show that language BLANK_{TM} is undecidable, we reduce HALT_{TM} to BLANK_{TM}: if BLANK_{TM} is decidable, then HALT_{TM} is decidable, which is a contradiction.

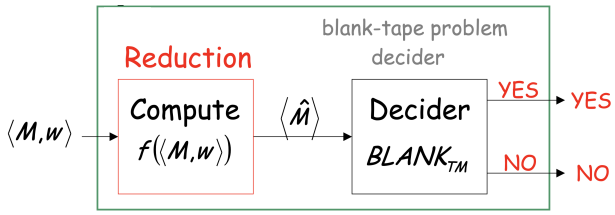


Figure 6: Decider for HALT_{TM}

Now let us build the reduction $\langle \hat{M} \rangle = f(\langle M, w \rangle)$ such that:

$$\langle M, w \rangle \in \text{HALT}_{TM} \iff \langle \hat{M} \rangle \in \text{BLANK}_{TM}$$

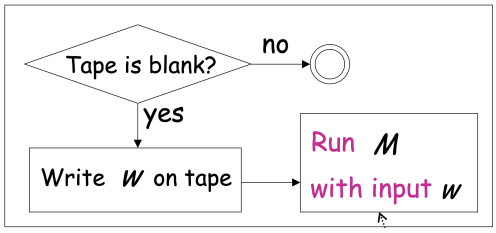


Figure 7: $\langle \hat{M} \rangle$ representation

6.5 Undecidable Problems for Recognizable Languages

Let L be a recognizable language and take in input a machine M:

- EMPTY_{TM} is undecidable:

$$\text{EMPTY}_{TM} = \{\langle M \rangle : \\ M \text{ is a TM that accepts the empty language } \emptyset\} \rightarrow \text{A}_T M \leq_m \\ \text{EMPTY}_T M$$

- REGULAR_{TM} is undecidable:

$$\text{REGULAR}_{TM} = \{\langle M \rangle : \\ M \text{ is a TM that accepts a regular language}\} \rightarrow \text{A}_T M \leq_m \\ \text{REGULAR}_T M$$

- SIZE2_{TM} is undecidable:

$$\text{SIZE2}_{TM} = \{\langle M \rangle : \\ M \text{ is a TM that accepts a exactly two strings}\} \rightarrow \text{A}_T M \leq_m \\ \text{SIZE2}_T M$$

6.6 Language Properties

A **non-trivial property** P is possessed by some Turing-recognizable languages, but not all:

$P_1 : L$ is regular?

YES: $L = \emptyset$

NO: $L = \{a^n | n \geq 0\}$

NO: $L = \{a^n b^n | n \geq 0\}$

A **trivial property** P is possessed by all Turing-recognizable languages:

$P_2 : L$ is accepted by some TM?

True for all Turing-recognizable languages

7 Complexity Theory

Complexity theory seeks to understand what makes problems algorithmically difficult to solve. A decidable problem may not be solvable since the solution could require an inordinate amount of time or memory. The **running time** of an algorithm as a function of the length of the string representing the input.
Let us consider a deterministic Turing machine M deciding a language L: for any string w, the computation of M terminates in a finite amount of transitions, which is the **decision time**. $T_M(n)$ is the maximum time required to decide any string of length n. We estimate the running time since the exact one is complex. In **asymptotic analysis**, we seek to understand the running time of an algorithm with large inputs.
Let us see the following example:

$$A = \{0^k 1^k \mid k \geq 0\} \quad TM \text{ on input string } w$$

1. Scan the tape and reject if a 0 is found to the right of a 1: $O(n)$
2. Repeat if both 0s and 1s remain on the tape: $O(n^2)$
3. Scan the tape, crossing off a single 0 and a single 1
4. If 0s remain after the 1s have been crossed off, or if 1s still remain after all the 0s have been crossed off, reject: $O(n)$
5. Otherwise, accept

The complexity is evaluated as follows:

$$O(n) + O(n^2) + O(n) = O(n^2) \Rightarrow A \in TIME(n^2)$$

Let us analyze the last example on a two-tapes TM: each step has an execution time of $O(n)$.
The following are some time classes:

- **Deterministic**

  All the languages decidable by a DTM that run in $O(T(n))$.

- **Polynomial**

  Tractable algorithms with results decided fastly for small ks.

- **Exponential**

  Intractable algorithms that take an enormous amount of time.

7.1 Post Correspondence Problem

We are given two sets of n strings defined as follows:

$$A = w_1, w_2, ..., w_n$$
$$B = v1_v2, ..., v_n$$

There exists a **post correspondence solution** if there is a sequence such that:

$$w_i w_j ... w_k = v_i v_j ... v_k$$

Indices may be repeated. Let us see the following example:

$$A: \quad w_1 = 100 w_2 = 11 \quad w_3 = 111$$
$$B: \quad v_1 = 001 w_2 = 111 \quad w_3 = 11$$

PC-Solution: $2, 1, 3 \ \rightarrow \ w_2 w_1 w_3 = v_2 v_1 v_3$

Rule #1: if every top string is longer than the corresponding bottom one, there can't be a match.

Rule #2: if there is a domino with the same string on the top and on the bottom, then there is a match.

The **Post Correspondence Problem** (**PCP**) consists of finding a match given a set of dominos:

$$\text{PCP} = \{\text{P} \mid \text{P is a set of dominos with a match } \}$$

Language PCP is undecidable.
The **Modified Post Correspondence Problem** (**MPCP**) is defined as follows:

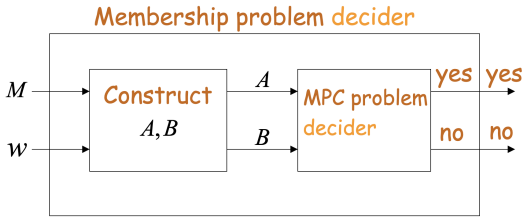
$$A = w_1, w_2, ..., w_n$$
$$B = v1_v2, ..., v_n$$

MPC-Solution: $1, i, j, ..., k \ \rightarrow \ w_1 w_i w_j ... w_k = v_1 v_i v_j ... v_k$

The MPCP is undecidable, proved by reducing the membership problem to it:

1. Membership problem (alternatively): given an unrestricted grammar G and a string w, $w \in L(G)$?
2. Suppose there exists a decider for the MPCP

3. Build a decider for the membership problem by using the MPCP decider as a subroutine



4. Convert Turing machine M and string w to the sets of strings A and B:

$$M \text{ accepts } w \Leftrightarrow \text{MPC solution for } A \text{ and } B$$

5. Since membership problem is undecidable, MPCP is undecidable

PCP Theorem: PCP is undecidable.
We can prove it by reducing the MPCP to the PCP.

1. Suppose we have a decider for the PCP
2. Build a decider for the MPCP by using the decider for the PCP as a subroutine
3. Convert the input instances

7.2 Undecidable Problems for CFGs

7.2.1 Empty-intersection Problem

Given to CFGs G_1 and G_2, the **empty-intersection problem** is defined as follows:

$$L(G_1) \cap L(G_2) = \emptyset$$

This problem is undecidable:

1. Suppose we have a decider for the empty-intersection problem
2. Build a decider for the PC problem using the decider for the empty-intersection problem as a subroutine
3. Reduce the PC problem to the empty-intersection problem

7.2.2 Ambiguity Problem

Ambiguity Theorem: for a context-free grammar G, it is undecidable to determine if G is ambiguous.
The theorem is proved by reducing the PC problem to the ambiguity problem:

1. Suppose we have a decider for the ambiguity problem
2. Build a decider for the PC problem using the decider for the ambiguity problem as a subroutine
3. Reduce the PC problem with inputs A and B to the ambiguity problem by building CFGs as follows:
   (a) Define S_A as the starting variable of G_A
   (b) Define S_B as the starting variable of G_B
   (c) Define S as the starting variable of G:

$$S \to S_A | S_B$$

Finally, we can state what follows:

(A, B) has a PC solution $\Leftrightarrow L(G_A) \cap L(G_B) \neq \emptyset \Leftrightarrow G$ is ambiguous

8 Polynomial vs Non-polynomial

P is the class of languages that are decidable in **polynomial time** on a DTM:

$$P = \cup_{k>0} \text{TIME}(n^k)$$

P is invariant for all models of computation that are **polynomially equivalent** to the DTM, and represents the class of problems that are realistically solvable on a computer.
NP is the class of languages that are verified in **non-polynomial time** on a NTM:

$$NP = \cup_{k>0} \text{NTIME}(n^k)$$

8.1 Complexity Relation

Every $t(n)$ time MTM has an equivalent $O(t^2(n))$ time STM.

8.1.1 Polynomial Difference

Let N be a NTM that is a decider with a running time of $f : N \to N$, where $f(n)$ is the maximum number of steps that N uses on any branch given an input of length n. The running time of a NTM characterizes the complexity of the NP class.

8.1.2 Exponential Difference

Every $t(n)$ time NTM has an equivalent $2^{O(t(n))}$ time DTM.

8.1.3 Nondeterminism

A NTM decides each string of length n in time $O(t(n))$. The language class is $NTIME(T(n))$.

8.2 Satisfiability Problem

Consider the following language:

$$L = \{w \mid \text{expression } w \text{ is satisfiable}\} \quad L \in \text{TIME}(2^{n^k})$$

The needed algorithm searches exhaustively all the possible binary values of the variables. This problem is an NP problem.

8.3 Polynomial Time Verifiability

A **verifier** for a language A is an algorithm V, where:

$$A = \{w \mid V \text{ accepts } \langle w, c\rangle \text{ for some string } c\}$$

The string c is called **certificate** and is used to verify that a string w is a member of A.
The time of a verifier is only measured in terms of the length of w: a polynomial time verifier runs in polynomial time in the length of w.

8.4 NP-completeness

Np-complete problems represent a subset of the NP class. If a polynomial time algorithm exists for any of these problems, all problems in NP could be solved in polynomial time. There are two theoretical implications:

1. To prove $P = NP$ if one finds a polynomial time algorithm for NP-complete problem
2. To prove $P \neq NP$ if any problem in NP requires more than polynomial time, an NP-complete does

Furthermore, there is one practical implication:

1. Do not have to search for non-existing polynomial time algorithms

8.4.1 Polynomial Time Reductions

A **polynomial time reduction** is performed when there exists a DTM M such that for any string w it computes a **polynomial computable function** $f(w)$ in polynomial time $O(|w|)$:

$$w \in A \Leftrightarrow f(w) \in B$$

Since M cannot use more than $O(|w|^k)$ tape space in that time, we obtain the following relation:

$$|f(w)| = O(|w|^k)$$

Theorem - A is polynomial reducible to B; if $B \in P$, then $A \in P$. Let machine M be the decider for B in polynomial time and machine M' the decider for A in polynomial time. On input string w:

1. Compute $f(w)$
2. Run M on input $f(w)$
3. If $f(w) \in B$, then accept

Let us see the following example in which we reduce the 3CNF-satisfiability problem to the CLIQUE problem:

$$\text{3CNF-SAT} = \{w : w \text{ is a satisfiable 3CNF formula}\}$$
$$\text{CLIQUE} = \{\langle G, k\rangle : \text{ graph } G \text{ contains } k\text{-clique}\}$$

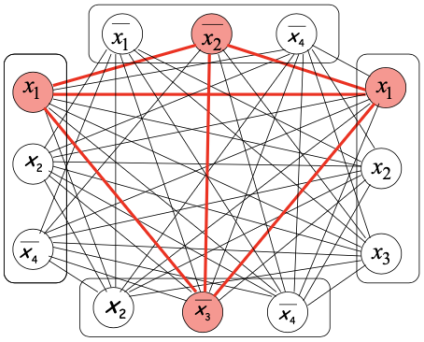
The first step consists in transforming the formula into a graph by creating one cluster of nodes for each clause in the formula, then add a link from a literal x to a all the other clauses' literals but each complement $\bar{x}$.
Let us have a look at the following formula

$$(x_1 \cup x_2 \cup \overline{x_4}) \cap (\overline{x_1} \cup \overline{x_2} \cup \overline{x_4}) \cap (x_1 \cup x_2 \cup x_3) \cap (x_2 \cup \overline{x_3} \cup \overline{x_4}) = 1$$

The values for which the formula is satisfied are the following:

$$x_1 = 1 \quad x_2 = 0 \quad x_3 = 0 \quad x_4 = 1$$

Therefore, the created graph is the following:



Finally, the formula is satisfied iff the graph has a 4-clique.

A language L is NP-complete if $L \in NP$ and every language in NP is reduced to L in polynomial time.

Cook-Levin Theorem: language SAT is NP-complete.
Since SAT $\in NP$, we only need to reduce all NP languages to the SAT problem in polynomial time.

Theorem: If language A is NP-complete, language $B \in NP$, and A is polynomial time reducible to B, then B is NP-complete.

8.4.2 Vertex Cover Problem

The **vertex cover** of a graph is a subset of nodes S such that every edge in the graph touches one node in S:

VERTEX-COVER= $\{G, k$: graph G contains a vertex cover of size $k\}$

Theorem: VERTEX-COVER is NP-complete.
After having proved that VERTEX-COVER is in NP, we reduce in polynomial time language 3CNF-SAT to VERTEX-COVER:

1. Let φ be a 3CNF formula with m variables and l clauses

$$\varphi = (x_1 \cup x_2 \cup x_3) \cap (\bar{x_1} \cup \bar{x_2} \cup \bar{x_4}) \cap (\bar{x_1} \cup \bar{x_2} \cup \bar{x_4})$$

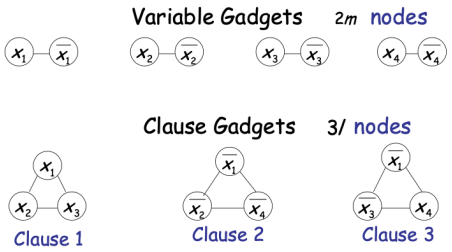


Figure 8: Construction blocks

2. Connect the variables through edges

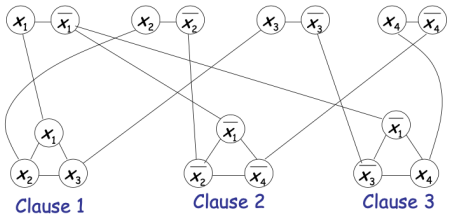


Figure 9: Connecting variables

3. First direction: if φ is satisfied, then G contains a cover of size k:

$$k = m + 2l = 10$$

4. Put every satisfying literal in the cover

5. Select one satisfying literal in each clause gadget and include the remaining ones in the cover This is a vertex cover since every edge is adjacent to a chosen node
   The first direction is proved

6. Second direction: if G contains a vertex-cover of size $k = m+2l$, then φ is satisfiable

7. Choose one literal in each variable gadget

8. Choose 2 nodes in each clause gadget

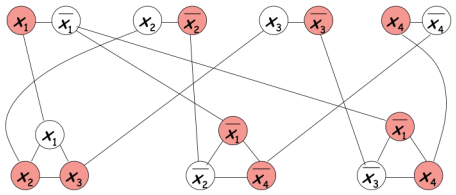


Figure 10: Vertex cover of G

9. For variable assignment, choose the literals in the cover from variable gadgets The second direction is proved

For general case:

- Edges in variable gadgets are incident to at least one node in cover
- Edges in clause gadgets are incident to at least one node in cover, since two nodes are chosen in a clause gadget

The general approach for proving that a problem is NP-complete is the following:

1. Show that problem B belongs to NP
2. Find a known NP-complete problem A and show $A \leq_p B$

If we can only complete step 2, then the problem is **NP-hard**.
The following are three proof techniques:

1. **Restriction**

   Show that A is a sub-problem of B

2. **Local Replacement**

   Show that every basic unit in an instance of A can be replaced by a different structure in a uniform way to obtain an instance of B

3. **Component Design**

   Show that the constituents of an instance of A can be used to "design" components that can be combined to "realize" instances of B

9 Appendix A: SAT Solvers

9.1 Propositional Logic

Let us consider boolean variables shirt and tie:

- One should not wear a tie without a shirt: $\neg$ tie $\vee$ shirt
- Not wearing a tie nor a shirt is impolite: tie $\vee$ shirt
- Wearing a tie and a shirt is overkill: $\neg$(tie $\wedge$ shirt)

The **syntax** of a Boolean formula, also known as **well-formed formula** (**wff**), defines how to write such a formula, while the **semantics** concerns interpreting the symbols in a wff to obtain a statement which is either true or false.
A **propositional language** is a set L of symbols called **propositional variables** p_i. Every $p \in L$ is a wff called an **atomic formula**. An **interpretation** I assigns one truth value to every propositional variable.
Given a wff F and an interpretation I, we say that $I| = F$ if F evaluates to true under interpretation I. The following are some properties of wffs:

1. **Satisfiability**: there exists at least one I such that $I| = F$
2. **Validity**: for every interpretation $I \Rightarrow I| = F$
3. **Unsatisfiability**: for every interpretation $I \Rightarrow I/| = F$

9.2 SAT Problem

The **SAT problem** is an NP-complete problem defined as follows: given a Boolean formula ϕ, check if ϕ is satisfiable, meaning if a set of variables can be assigned to make the formula evaluate to true. NP is a class of problems whose solution can be verified in polynomial time on a deterministic Turing machine. Every problem in NP can be reduced to an NP-complete problem with a deterministic Turing machine in polynomial time.

9.3 SAT Solvers

A **SAT solver** is a program that decides whether a Boolean formula is satisfiable or not: if the formula is satisfiable, then it returns a

model known as **satisfying assignment**, while if the formula is unsatifsiable, then it returns a proof. State-of-the-art solvers are based on the Conflict-Driven Clause Learning (CDCL) algorithm.
A **literal** is either a variable or a negated variable. A **clause** is a disjunction of literals. Boolean formulas can be expressed in their **Conjunctive Normal Form**, which is a conjunction of clauses.
SAT solvers accept formulas written in the **DIMACS** format:

- First line: `p cnf variables_number clauses_number`
- Comments: start a line with `'c'`
- Lines: represent clauses and terminate with a 0
- Variables: positive (true) and negative (false) numbers

9.4 Pigeon-Hole SAT Problem

There are n pigeons and k holes:

1. A pigeon cannot stay inside two holes
2. Each hole can contain only one pigeon
3. Each pigeon must stay inside some hole
4. Example: "pigeon 2 does not like hole 3"

We introduce $n \cdot k$ variables as follows:

$$inHole(p, h)$$

Let us define the problem in a SAT solver:

1. Encode variables a and b that are not true at the same time:

$$\neg(a \wedge b) \equiv (\neg a \vee b)$$

9.5 Space Complexity

The **space complexity** of a Turing machine T is the function Space_T such that $\text{Space}_T(x)$ is the number of distinct tape cells read during computation $T(x)$. If $T(x)$ does not halt, then $\text{Space}_T(x)$ is undefined.
For any function f, we say that the space complexity of a decidable language L is in $O(f)$ if there exists.

10 Appendix B: More on Reductions

A is undecidable
A $\leq_m$ B
thus B is undecidable
use decider for B into the decider for A after the reduction in which the input for A is transformed into the input for B
Ex 2 (similarly) Reduce $A_T M$ to $HALT_T M$ $\rightarrow A_T M$ <=$HALT_T M$
$A_T M$: on input $\langle M, w \rangle$ where M accepts w
$HALT_T M$: on input $\langle M, w \rangle$ where M halts on w

$f(\langle M, w \rangle) = \langle M', w' \langle$
f: M': on input x
1) run M on x
2) if M accepts $\rightarrow$ accept
3) if M rejects $\rightarrow$ reject
Output: $\langle M', w \rangle$

Define the formal parameter x:
M' on input x:
1) Run M on input x $\rightarrow$ run M on w and M accepts 2) if M accepts $\rightarrow$ accept $\Rightarrow$ condition satisfied and M' accepts w and halts
M' halts on w, then M accepts w
M does not accept w, then M' does not halt on w

M' runs on w:
1) M runs on w