Chapter 3

Computability and Complexity

3.1 Introduction

Consider a domain D of concrete objects, e.g., numbers, words in an alphabet, finite graphs, etc. We will refer to elements I of D as *instances*.

Definition 3.1.1 Given a set $P \subseteq D$ of instances, the *decision problem* for P is the following question:

> Is there an algorithm (*effective* or *mechanical procedure*) which will tell us, in finitely many steps, for each instance $I \in D$ whether $I \in P$ or $I \notin P$?

Example 3.1.2 Given a formula in propositional logic, is there an algorithm to determine whether it is satisfiable? One can also consider the decision problem where the formulas are now in first-order logic.

Example 3.1.3 Is there an algorithm that, given $n \in \mathbb{N}$ and

$$m \in \{0, 1, \ldots, 9\},$$

tells us whether the n-th digit in the decimal expansion of π is m?

Example 3.1.4 (The Hilbert 10th Problem) Is there an algorithm to verify whether an arbitrary polynomial (in several variables)

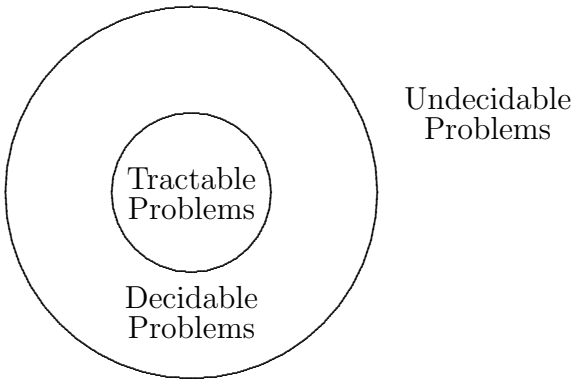
$$\sum a_{i_1 i_2 i_3 \ldots i_n} x_1^{i_1} x_2^{i_2} \ldots x_n^{i_n}$$

with integer coefficients $a_{i_1 i_2 \ldots i_n}$ has an integer solution?

Definition 3.1.5 If such an algorithm exists for the decision problem (given by) P, we will call P *decidable*. Otherwise we call it *undecidable*.

Example 3.1.6 The validity problem for formulas in propositional logic is decidable (use truth tables). The Hilbert 10th Problem is undecidable (Matyasevich, 1970).

Thus, modulo our rather vague definition of "algorithm," we have the first main classification of (decision) problems as decidable or undecidable. Our next goal will be to classify further the decidable problems according to the (time) complexity of their algorithms and distinguish between those for which an efficient (feasible) algorithm is possible, *the tractable problems*, and those for which it is not, the *intractable* problems.



In particular, this will lead to the study of an important class of problems widely believed to be intractable, but for which no proof has yet been found (this is the famous $\mathcal{P} = \mathcal{NP}$ problem).

We will now formalize these concepts and present an introduction to the study of decision problems and their complexity.

3.2 Decision Problems

We can represent concrete finite objects as words (strings) in a finite alphabet.

Definition 3.2.1 An *alphabet* is an arbitrary nonempty set, whose elements we call *symbols*.

We will be mainly dealing with finite alphabets $A = \{\sigma_1, \ldots, \sigma_k\}$.

Definition 3.2.2 A *word* (or *string*) for A is a finite sequence $a_1 a_2 \ldots a_n$, where each a_i is a symbol in A. If $w = a_1 a_2 \ldots a_n$ is a word, then $n = |w|$ is the *length* of w.

We denote by A^n the set of all words of length n from A. By convention we also have the *empty* word θ which has length 0. Thus $A^0 = \{\theta\}$. Finally we let

$$A^* = \bigcup_n A^n$$

be the set of all words from A (including θ).

Examples 3.2.3

(i) Natural numbers can be represented as words using decimal notation and the alphabet $\{0, 1, 2, \ldots, 9\}$, or in the alphabet $\{0, 1\}$ using binary notation, or even in the alphabet $\{1\}$ using unary ("tally") notation.

(ii) A finite graph $G = \langle V, E\rangle$, with set of vertices $V = \{v_1, \ldots, v_n\}$, can be represented by its adjacency matrix $A = (a_{ij})$, $1 \leq i, j \leq n$, where $a_{ij} \in \{0, 1\}$ and $a_{ij} = 1$ iff $(a_i, a_j) \in E$. This can be in turn represented by the word

$$a_{11} a_{12} \ldots a_{1n} a_{21} \ldots a_{2n} \ldots a_{n1} \ldots a_{nn}$$

in the alphabet $\{0, 1\}$.

Definition 3.2.4 A *decision problem* consists of a finite alphabet A and a set of words $P \subseteq A^*$.

Consider a finite first order language $L = \{f_1, \ldots, f_n;\ R_1, \ldots, R_m\}$. Representing the variable x_n by the string xn, where n is written in binary, we can view every wff in L as a word in the finite alphabet $A = L \cup \{x, 0, 1, \neg, \wedge, \vee, \Rightarrow, \Leftrightarrow, \exists, \forall, (,), =\}$. We let VALIDITY_L be the decision problem corresponding to (A, P), where

$$P = \{S : S \text{ is a sentence in } L \text{ and } \models S\}.$$

3.3 Turing Machines

Now that we have precisely defined decision problems, we will go on to make precise our previously vague concept of an "algorithm." We will do this by presenting a model of computation, using a formalized representation of a computer known as a Turing machine. There are many equivalent formalizations of the notion of "computability," but Turing machines were one of the earliest to be formulated and remain one of the easiest to understand and work with.

As a motivation for the definition which is to follow, consider a running computer program. At any given time, it is in some internal state (for example, which line of the program is being executed) and has some amount of stored data (in variables in memory and on disk). With each "tick" of the computer's clock, it moves to a new state based on its previous state and the contents of its stored data (for example, performing a conditional jump), and may modify the stored data as well.

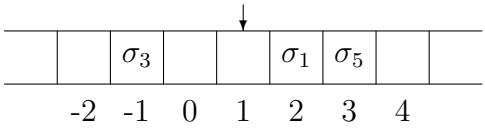
With this characterization in mind, we make the following definition.

Definition 3.3.1 A *Turing machine* M on an alphabet $A = \{\sigma_1, \sigma_2, \ldots, \sigma_k\}$ consists of the following:

(i) A finite set of states $\{Q_0, Q_1, \ldots, Q_m\}$. We call Q_0 the *start* state and Q_m the *stop* state.

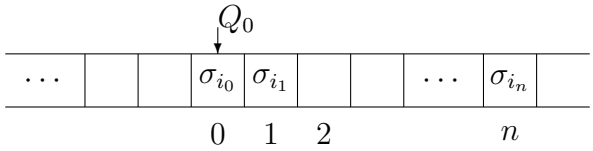
(ii) A table of $(m+1) \cdot (k+1)$ 5-tuples of the form (Q_i, a, b, s, Q_j), where $a, b \in A \cup \{*\}$. (Here $*$ is an additional symbol. whose meaning we explain later. $*$ is sometimes also denoted, for consistency, by $\sigma_0 = *$, so that $A \cup \{*\} = \{\sigma_0, \sigma_1, \ldots, \sigma_k\}$.) For each $0 \leq i \leq m$, $a \in A \cup \{*\}$, there must be exactly one entry in this table of the form (Q_i, a, b, s, Q_j). Finally we must have $s \in \{-1, 0, +1\}$.

The meaning of the set of states is clear from our discussion above, but the rest of the definition may be somewhat more obscure. Here is how to imagine the operation of a Turing machine. We have a two-way unlimited "tape" divided into squares, each of which may contain at most one symbol in A (or possibly be empty), and a read-write head that at any instant is positioned at exactly one square on the tape.



The head can move in one step along the tape at most one square at a time, right or left, it can write a symbol in the scanned square (replacing any symbol that may have been there) or erase the symbol in the scanned square.

Using this image, for any input $w \in A^*$, we can describe the computation of a Turing machine (TM) on this input w as follows: Let $w = \sigma_{i_0}\sigma_{i_1}\ldots\sigma_{i_n}$. Put this input on the tape as in the picture below and put the scanning head at the leftmost symbol in w (anywhere, if $w = \theta$) and assign to it state Q_0 (the start state):



We now follow the table of the TM by interpreting the entry

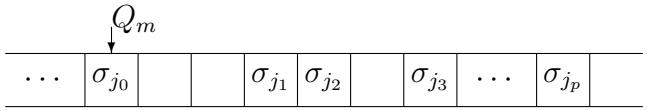
$$(Q_i, a, b, s, Q_j)$$

as expressing the instruction:

When the head is at state Q_i and scans the symbol a (if $a = *$ this means that it scans an empty square – thus $*$ is simply a symbol for the empty square), then prints the symbol b ($b = *$ means "erase what is present at the scanned square"). Then if $s = +1$, move the head one square to the right; if $s = -1$, move one square to the left; and if $s = 0$, leave the head where it is. Finally, change state to Q_j.

So in terms of our original characterization of a computer program, the symbol a represents the possible role of stored data in choosing the new state, while b is the possible change to the stored data. s is an artifact of how we have chosen to represent the stored data as an infinite tape. While it may seem like a limitation to be only able to examine and change one square of the tape at a time, this can in fact be overcome by the addition of a suitable number of intermediate internal states and transitions, and it simplifies the analysis considerably.

The computation proceeds this way step by step, starting from the input w. It terminates exactly when it reaches the stop state, Q_m. When the computation halts, whatever string of symbols from A remain on the tape to the right of the scanning head, say $v = \sigma_{j_0}\sigma_{j_1}\ldots\sigma_{j_p}$ (from left to right), we call the string v the *output* of the computation.



Note that a computation from a given input w may not terminate, because we need not ever reach the stop state Q_m.

Example 3.3.2 Consider the following TM on the alphabet $A = \{0, 1\}$:

states: Q_0, Q_1, Q_2

table:
$(Q_0, a, \bar{a}, +1, Q_0)$, for $a \in A$, where $\bar{a} = 1 - a$,
$(Q_0, *, *, -1, Q_1)$
$(Q_1, b, b, -1, Q_1)$, for $b \in A$,
$(Q_1, *, *, +1, Q_2)$
$(Q_2, a, a, 0, Q_2)$, $a \in A \cup \{*\}$

On an input of, say, $w = 1011101$, the machine terminates with output 0100010. In general, on input $w \in A^*$ the machine terminates with output $\bar{w}$ which is equal to w with 0s and 1s switched. (This operation is also known as the binary "ones complement".)

Example 3.3.3 $A = \{1\}$

states: Q_0, Q_1

table:
$(Q_0, 1, 1, +1, Q_0)$
$(Q_0, *, 1, +1, Q_0)$
$(Q_1, a, a, 0, Q_1)$, $a \in A \cup \{*\}$

Then no matter what the starting input is, say $w = 1\ldots1$ in A^*, this machine does not terminate: it just keeps adding 1's to the right of w forever.

Definition 3.3.4 We call a decision problem (A, P) *decidable* if there is a Turing machine M on some finite alphabet $B \supseteq A \cup \{Y, N\}$ such that for every word $w \in A^*$:

$w \in P \Rightarrow$ on input w the machine M halts with output Y
$w \notin P \Rightarrow$ on input w the machine M halts with output N

Otherwise, P is called *undecidable*.

3.4 The Church-Turing Thesis

We have now formalized the concept of decidability using the Turing machine model of computation. In this section we will discuss alternative ways to formalize this concept, and see that they are all equivalent.

3.4.A Register Machines

First we will discuss in detail one more model of computation which is closer to the operation of real-world computers: the so-called *register machines*.

Definition 3.4.1 A *register machine* N on an alphabet $A = \{\sigma_1, \ldots, \sigma_k\}$ is a program consisting of a finite list $1 : I_1, \ldots, n : I_n$ of consecutively numbered instructions, each of which is one of the following types:

ADD$_j$ Rm,	$1 \leq j \leq k$; $m = 1, 2, \ldots$
DEL Rm,	$m = 1, 2, \ldots$
CLR Rm,	$m = 1, 2, \ldots$
R$p \leftarrow$Rm,	$m, p = 1, 2, \ldots$
GO TO ℓ,	$1 \leq \ell \leq n$
IF FIRST$_j$ Rm GO TO ℓ	$1 \leq j \leq k$; $1 \leq \ell \leq n$; $m = 1, 2, \ldots$
STOP	

We assume moreover that $I_i = \text{STOP}$ if and only if $i = n$.

To understand the operation of this machine, imagine an infinite list of registers $R1, R2, \ldots$, each of which is capable of storing an arbitrary word in A^*. Then the meaning of each instruction in the above list is the following:

ADD$_j$ Rm:	add σ_j to the right of (the word in) Rm
DEL Rm:	delete the leftmost symbol in Rm
CLR Rm:	erase the word in Rm
R$p \leftarrow$Rm:	replace the word in Rp by the word in Rm
GO TO ℓ:	go the ℓth instruction $\ell : I_\ell$
IF FIRST$_j$ Rm GO TO ℓ:	if the word in Rm starts with σ_j, go to the ℓth instruction; otherwise go to the next instruction

STOP: halt the computation.

Using this, we can now describe the computation of the register machine (RM) N on any input $w \in A^*$: Put this input w in the first register R1 (with all other registers empty). Follow the instructions $1 : I_1, \ldots, \ell : I_\ell$ as above in order (unless of course a GO TO instruction is met). This computation terminates exactly when we reach (if ever) I_n : STOP. If $v \in A^*$ is the word in the register R1, when the computation stops, we call v the *output* of the computation. Again a computation from a given input w may not terminate. Also note that if the program for N mentions at most the registers R1, R2,..., Rm, then these are the only registers used in the computation on any input.

Example 3.4.2 Consider the following RM on the alphabet $A = \{1, \diamond\} = \{\sigma_1, \sigma_2\}$

1: IF FIRST$_1$ R1 GO TO **??**
2: DEL R1
3: IF FIRST$_1$ R1 GO TO **??**
4: GO TO **??**
5: DEL R1
6: ADD$_1$ R2
7: GO TO **??**
8: DEL R1
9: ADD$_1$ R2
10: GO TO **??**
11: R1←R2
12: STOP

Then on input $\underbrace{1\ 1\ldots1}_{n} \diamond \underbrace{1\ 1\ldots1}_{m}$ the machine terminates with output $\underbrace{1\ 1\ldots1}_{n+m}$ $(n, m \geq 1)$. Thus it performs addition in unary notation.

Example 3.4.3 Consider the alphabet $A = \{1\}$, and the machine:

1: ADD$_1$ R1
2: GO TO 1
3: STOP

Then this RM does not terminate on any input in A^*.

Although the TM and RM models of computation are quite different, it can be shown that they are equivalent, in the sense that whatever can be computed using one of the models can be computed with the other, and vice versa. More precisely, consider two finite alphabets A, B and a *partial function* $f : A^* \to B^*$, that is a function whose domain is a subset of A^* and which takes values in B^*. (Normally, when we write $g : X \to Y$ we mean that g is a *total* function with domain *exactly* X and values in Y. We will however use this notation below even if the domain of g is only a (possibly proper) *subset* of X, but in this case we will explicitly state that g is partial.)

Definition 3.4.4 $f : A^* \to B^*$ is TM-*computable* if there is a finite alphabet $C \supseteq A \cup B$ and a Turing Machine M on C such that for each input $w \in A^* (\subseteq C^*)$, M terminates on w iff w is the domain of f (i.e. $f(w)$ is defined), and in that case the output of M on input w is $f(w)$.

We define what it means for f to be RM-computable in a similar way. We can then show

Theorem 3.4.5 *TM-computable = RM-computable.*

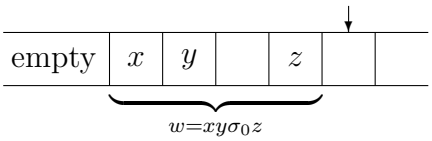
This is a programming exercise and we will not discuss it in detail here. First, we show that given a RM N on an alphabet C, one can construct a TM M on the alphabet $D = C \cup \{\diamond\}$, such that if we put $w \in C^*$ as input in N and the same input $w \in C^*$ in M, then N terminates on w iff M terminates on w, and for such w the outputs of N and M are the same.

Specifically, we assume that all the registers mentioned in N are among R1, R2,..., Rm. If R1,..., Rm, at some stage in the computation, contain the words $w_1, \ldots, w_m$, respectively, then the idea is to represent their contents by the word $w_1 \diamond w_2 \diamond \cdots \diamond w_m \diamond$ in the tape of the TM M, with the head scanning the leftmost symbol of w_1 (or the first $\diamond$ if $w_1 = \theta$). Then for each instruction of N we will introduce a block of entries in the table of M, whose effect on $w_1 \diamond w_2 \diamond \cdots \diamond w_m \diamond$ will correspond to the effect of the instruction of N on the corresponding register values. In the case of GOTO instructions this simulation will preserve the same flow of control.

Conversely, we can show that given a TM M on an alphabet C, one can construct a RM N on some alphabet $D \supseteq C$ such that again if we put $w \in C^*$ as input in M and N, then M terminates on w iff N terminates on w, and for such w the outputs of M and N on w are the same. For example, one way to do that is to take $D = \{\sigma_0, \sigma_1, \ldots, \sigma_k\} \cup \{Q_0, \ldots, Q_m\}$,

if $C = \{\sigma_1, \ldots, \sigma_k\}$ and M has set of states $\{Q_0, \ldots, Q_m\}$. Now consider a tape snapshot wQ_iav, which indicates that the head is in state Q_i, scans the symbol a (which could be the empty square, i.e. $a = \sigma_0$), and to the left of it is the word $w \in \{\sigma_0, \sigma_1, \ldots, \sigma_k\}^*$ and to the right of it the word $v \in \{\sigma_0, \sigma_1, \ldots, \sigma_k\}^*$. Notice that at each stage of the computation of the machine M there will only be finitely many non-empty squares. Here w is the word which represents from left-to-right the contents of the tape starting from the left-most non-empty square to the square just to the left of the square scanned by the head. In case every square to the left of the head is empty, $w = \theta$. Similarly for v.

Example.



We can represent this tape snapshot in the register machine N by putting w in R1, Q_ia in R2, and v in R3. Then for each entry $(Q_i, a, -, -, -)$ in the table of M we can write a set of instructions (subroutine) for N such that the effect of this entry on wQ_iav is the same as the effect of the corresponding set of instructions on the corresponding register values. In this way, we can show that the notions of TM-computability and RM-computability are equivalent.

3.4.B The Church-Turing Thesis

As mentioned earlier, many alternative models of computation $\mathcal{M}$ have been proposed over the years, and for each one we have a corresponding notion of an $\mathcal{M}$-computable function. Two common ones which we will not discuss, but which you may encounter later, are the Lambda Calculus and the theory of Recursive Functions. Another one is the theory of cellular automata, an specific example of which is Conway's Game of Life. It turns out that in all these cases, one can again prove that $\mathcal{M}$-computable = TM-computable, so all these models are, in that sense, equivalent to the Turing machine model.

It is accepted today that the informal notion of algorithmic computability is represented correctly by the precise notion of TM-computability (= RM-computability = ...), in a way similar to the assertion that the formal ϵ-δ definition of continuity of functions of a real variable represents accurately

the intuitive concept of continuity. This (extramathematical) assertion that "computability = TM-computability" is referred to as the *Church-Turing Thesis*.

More explicitly, consider a partial function $f : A^* \to B^*$ (A, B finite alphabets). We call it *intuitively computable* if there is an algorithm which for each input $w \in A^*$ terminates (in finitely many steps) exactly when w is in the domain of f and in that case produces the output $f(w)$. Then the Church-Turing Thesis is the assertion:

$$\text{intuitively computable} = \text{TM-computable}.$$

We will simply say *computable* instead of TM-computable from now on.

Remark. Any decision problem (A, P) may be identified with the (total) function $f : A^* \to \{Y, N\}^*$ given by $f(w) = Y$ if $w \in P$ and $f(w) = N$ if $w \notin P$. Thus (A, P) is decidable iff f is computable.

3.5 Universal Machines

So far we have been discussing machines which represent the operation of a single computer program. But what about a computer itself, i.e. a single machine which can execute any program? It turns out that our existing formalization is sufficient to deal with this more general case as well. After a bit of thought, this should come as no surprise, since after all the inside of a computer is just a program which sequentially reads and executes the instructions of other programs. We will now describe how to formalize this notion.

To start with, we can effectively code the words in an arbitrary finite alphabet $A = \{\sigma_1, \ldots, \sigma_k\}$ by using words in the binary alphabet $\{0, 1\}$. For example, if $k = 22$ we can use five letter words from $\{0, 1\}$ to represent each individual symbol in A, and thus every word of length n from A can be encoded as a word of length $5n$ in the binary alphabet. (We could even use the unary alphabet $\{1\}$ instead of the binary alphabet, but this would require exponential increase in the length of the word if A has more than one symbol. This does not affect computability issues, but it can certainly affect the complexity issues discussed later.)

So let's assume that we have fixed some simple method for encoding words in a given alphabet A by binary words and refer to the binary word

$b(w)$ encoding the word $w \in A^*$ as the *binary* code of w. It is now not hard to see that given a TM M on A we can construct a TM M^* on $\{0,1\}$ such that for each $w \in A^*$ the input-output behavior of M on w is exactly the same on the input-output behavior of M^* on $b(w)$.

Thus, without any loss of generality, we can restrict ourselves to discussing TMs on the binary alphabet. Any such machine M is simply given by a finite table of 5-tuples which again can be easily coded as a word in the binary alphabet. We call this the *binary code* of M and denote it by $b(M)$. By carrying out a somewhat complicated programming project, one can now construct a *universal Turing Machine*. This is a TM U on $\{0,1\}$ such that for any TM M on $\{0,1\}$ and any input $w = \{0,1\}^*$, if we put $b(M)*w$ (where $* =$ empty square) as input in U (so that in the beginning the head scans the first symbol of $b(M)$), then U terminates on this input iff M terminates on w, and in this case their outputs are the same. So we have:

Theorem 3.5.1 (Existence of universal TM) *We can construct a TM U on $\{0,1\}$ such that for any TM M on $\{0,1\}$ with binary code $b(M)$, the input-output behavior of U on $b(M) * w$ ($w \in \{0,1\}^*$) is the same as the input-output behavior of M on w.*

Remark. The universal TM was conceived theoretically by Turing in the 1930's. Any personal computer today is essentially a practical implementation of this idea.

3.6 The Halting Problem

We will now use theorem **??** to show that the so-called *halting problem* for TM is undecidable. This is precisely formulated as:

$$\begin{aligned} A &= \{0,1\} \\ P &= \{b(M) : M \text{ is a TM on } \{0,1\} \text{ and } M \\ &\qquad \text{terminates on the empty input}\}. \end{aligned} \qquad \text{(HALTING)}$$

Theorem 3.6.1 (Turing) HALTING *is undecidable.*

Proof. Consider the universal TM U. For each $x \in \{0,1\}^*$ consider the TM U_x which on the empty input first prints $x * x$, moves the head over

the first symbol of the first x, and then follows U. The table of U_x is easily obtained from the table of U by adding some further entries depending on x.

Now, if HALTING were decidable, one could easily build a TM N_0 on $\{0,1\}$ such that for any $x \in \{0,1\}^*$, N_0 on input x would terminate with output 0 if U_x terminated on the empty input, and N_0 on input x would terminate with output 1 if U_x did not terminate on the empty input. By changing N_0 a little, we could then construct a TM M_0 on $\{0,1\}$ such that if N_0 on input x terminates with output 1, so does M_0, but if N_0 on input x terminates with output 0, then M_0 does *not* terminate in that input. To simplify the notation, let us write

$$M(w) \downarrow$$

if a TM M terminates on input w and

$$M(w) \uparrow$$

if it does not. Then we have for $x \in \{0,1\}^*$

$$\begin{aligned} M_0(x) \downarrow &\Leftrightarrow U_x(\theta) \uparrow \\ &\Leftrightarrow U(x * x) \uparrow . \end{aligned}$$

In particular, for any TM M on $\{0,1\}$,

$$\begin{aligned} M_0(b(M)) \downarrow &\Leftrightarrow U(b(M) * b(M)) \uparrow \\ &\Leftrightarrow M(b(M)) \uparrow \end{aligned}$$

Putting $M = M_0$ we get a contradiction, since it cannot be the case that both $M_0(b(M_0)) \uparrow$ and $M_0(b(M_0)) \downarrow$. ⊣

Similarly one can of course prove that the corresponding halting problem for RM is undecidable. More generally, given any programming language, it is impossible to design an algorithm that will decide whether an arbitrary program will terminate on a given input or not.

3.7 Undecidability of the Validity Problem

We have succeeded in showing that one problem—the halting problem—is undecidable, but the prospects are not good for applying the method we

used to other problems, since it depends on the halting problem being a sort of "meta-problem" *about* computability. Fortunately, we can leverage this success to show the undecidability of other problems, by using the method of computable reductions, as follows.

Definition 3.7.1 Suppose (A, P) and (B, Q) are two decision problems. A (total) function $f : A^* \to B^*$ is a *(computable) reduction* of P to Q if f is computable and for any $w \in A^*$,

$$w \in P \Leftrightarrow f(w) \in Q.$$

Notice that if P is reduced to Q via f and Q is decidable, so that there is a total computable function $g : B^* \to \{Y, N\}^*$ such that $v \in Q \Rightarrow g(v) = Y$, $v \notin Q \Rightarrow g(v) = N$, then $g \circ f = h$ is computable and $w \in P \Rightarrow h(w) = Y$, $w \notin P \Rightarrow h(w) = N$. Thus P is decidable as well.

So if P can be reduced to Q and Q is decidable, then P is decidable. This observation provides a general method for showing that decision problems are *un*decidable. Specifically, to show that (B, Q) is undecidable, choose an appropriate problem (A, P) that is known to be undecidable, and find a reduction of P to Q. For then if (B, Q) were decidable, so would (A, P) be, which we know is not the case. We will apply this method to show the undecidability of the validity problem for an appropriate language L.

Theorem 3.7.2 (Church) *There is a finite language L such that the decision problem* VALIDITY$_L$ *is undecidable.*

Proof. First consider a variation of the halting problem, HALTING$_U$. This is simply the halting problem for the universal TM U. It consists of all $w \in \{0, 1\}^*$ for which $U(w) \downarrow$. Since for any TM M on $\{0, 1\}^*$, $M(\theta) \downarrow$ iff $U(b(M) * \theta) \downarrow$ iff $U(b(M)) \downarrow$, clearly HALTING$_U$ is also undecidable.

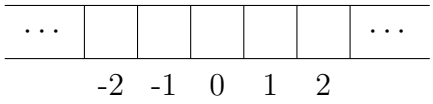
Say the states of U are $\{Q_0, Q_1, \ldots, Q_m\}$. We take L to consist of the following symbols:

$$\begin{array}{rl} 0 & : \text{constant} \\ S & : \text{unary function} \\ < & : \text{binary relation} \\ H & : \text{binary relation} \\ T_*,\ T_0,\ T_1 & : \text{binary relations} \\ R_0, \ldots, R_m & : \text{unary relations.} \end{array}$$

Intuitively, we have in mind the following interpretation for these:

(i) $0, S, <$ will be the 0, successor and order on $\langle \mathbb{Z}, 0, S, < \rangle$.

(ii) Variables will vary over $\mathbb{Z}$. They will represent both the time (i.e. stage in the computation), when restricted to ≥ 0 values (so 0 will be the beginning, 1 the next step, 2 the next one, etc.), and also the location of a square on the tape.



(iii) $H(t, x)$ will be true iff $t, x \in \mathbb{Z}$, $t \geq 0$ and the head at time t scans the square x.

(iv) $T_*(t, x)$ will be true iff $t, x \in \mathbb{Z}$, $t \geq 0$ and the tape at time t contains $*$ (i.e. is empty) at the square x, and similarly for T_0 and T_1.

(v) $R_i(t)$ will be true iff $t \in \mathbb{Z}$, $t \geq 0$, and at time t the machine is in state Q_i.

We will now assign to each $w \in \{0,1\}^*$ a sentence σ_w in L such that

$$U(w) \downarrow \quad \text{iff} \quad \sigma_w \text{ is valid.}$$

It will be clear from our construction that the map $w \mapsto \sigma_w$ is computable. Thus it will follow that HALTING_U can be reduced to VALIDITY_L, so VALIDITY_L must be undecidable.

We will first construct some preliminary sentences in L:

(i) Z is the conjunction of the following sentences:

$$\forall x(\neg x < x)$$
$$\forall x \forall y \forall z(x < y \wedge y < z \Rightarrow x < z)$$
$$\forall x \forall y(x < y \vee x = y \vee y < x)$$
$$\forall x \exists y(S(y) = x)$$
$$\forall x(x < S(x))$$
$$\forall x \forall y(x < y \Rightarrow y = S(x) \vee S(x) < y)$$

(expressing the usual properties of $\langle \mathbb{Z}, 0, S, < \rangle$).

(ii) NOCONFLICT is the conjunction of the following sentences:

$$\forall t(0 = t \vee 0 < t \Rightarrow R_0(t) \vee \cdots \vee R_m(t))$$
$$\forall t(0 = t \vee 0 < t \Rightarrow \neg R_i(t) \vee \neg R_{i'}(t)),\ 0 \leq i < i' \leq m$$
$$\forall t(0 = t \vee 0 < t \Rightarrow \exists x H(t, x))$$
$$\forall t \forall x \forall x'((0 = t \vee 0 < t) \wedge x \neq x' \Rightarrow \neg H(t, x) \vee \neg H(t, x'))$$
$$\forall t \forall x(0 = t \vee 0 < t \Rightarrow T_*(t, x) \vee T_0(t, x) \vee T_1(t, x))$$
$$\forall t \forall x(0 = t \vee 0 < t \Rightarrow (\neg T_*(t, x) \vee \neg T_0(t, x)))$$
$$\forall t \forall x(0 = t \vee 0 < t \Rightarrow (\neg T_0(t, x) \vee \neg T_1(t_1, x))$$
$$\forall t \forall x(0 = t \vee 0 < t \Rightarrow (\neg T_1(t, x) \vee \neg T_*(t, x))$$

expressing that at each time t, M is in exactly one state, and scans exactly one square, and for each time t and each square x there is exactly one symbol on that square (possibly empty).

(iii) START_w is the conjunction of the following sentences, where $w = w_0 \dots w_{n-1}$ with $w_i \in \{0, 1\}$, and we use the abbreviation

$$\bar{i} = S(S(\dots S(0)) \dots)$$

(i times) for all $0 \leq i \in \mathbb{Z}$:

$$R_0(0)$$
$$H(0, 0)$$
$$T_{w(i)}(0, \bar{i}),\ 0 \leq i < n$$
$$\forall x(\bar{n} < x \vee \bar{n} = x \Rightarrow T_*(x))$$
$$\forall x(x < 0 \Rightarrow T_*(x))$$

(expressing that the machine starts in state Q_0 with the head on the 0th square and on the input w).

(iv) NONSTOP is the sentence

$$\forall t(0 = t \vee 0 < t \Rightarrow \neg R_m(t))$$

expressing that the machine does not stop.

(v) STEP is the sentence constructed as follows:

Let (Q_i, a, b, s, Q_j) be an entry in the table of U. Say $s = +1$. (Appropriate changes have to be made in the formula below in case $s = 0$ or $s = -1$.) Assign to this entry the following sentence:

$$\forall t \forall x [0 = t \vee 0 < t \Rightarrow (R_i(t) \wedge H(t,x) \wedge T_a(t,x) \Rightarrow \\ R_j(S(t)) \wedge H(S(t), S(x)) \wedge T_b(S(t), x))]$$

expressing the action of the machine following this entry. Then STEP is the conjunction of all these sentences for all entries in the table.

Now let ρ_w be the conjunction of all the sentences Z, NOCONFLICT START$_w$, NONSTOP, STEP, and put $\sigma_w = \neg\rho_w$. Then we have

$$U(w) \downarrow \quad \text{iff} \quad \sigma_w \text{ is valid.}$$

To see this, first notice that if $U(w) \uparrow$, then σ_w is not valid, i.e. ρ_w has a model. For we can take as such a model

$$\mathcal{M} = \langle \mathbb{Z}, 0, S, <, H^{\mathcal{M}}, T_*^{\mathcal{M}}, T_0^{\mathcal{M}}, T_1^{\mathcal{M}}, R_0^{\mathcal{M}}, \ldots, R_m^{\mathcal{M}} \rangle,$$

where $0, S, <$ have their usual meaning and $H^{\mathcal{M}}, \ldots, R_m^{\mathcal{M}}$ are interpreted as in (3)-(5) before.

Conversely, assume that $U(w) \downarrow$, say the computation on input w terminates at time $N \geq 0$. If σ_w failed to be valid (working towards a contradiction), ρ_w would have some model

$$\mathcal{A} = \langle A, 0^{\mathcal{A}}, S^{\mathcal{A}}, <^{\mathcal{A}}, H^{\mathcal{A}}, T_*^{\mathcal{A}}, T_0^{\mathcal{A}}, T_1^{\mathcal{A}}, R_0^{\mathcal{A}}, \ldots, R_m^{\mathcal{A}} \rangle.$$

Since $\mathcal{A} \models Z$, clearly A contains a copy of $\mathbb{Z}$, with $0^{\mathcal{A}}$ corresponding to 0, $S^{\mathcal{A}}(0^{\mathcal{A}})$ to 1, etc. Denote by $\mathbf{n}$ the element of A corresponding to $n \in \mathbb{Z}$. (In general, A contains many more other elements than these $\mathbf{n}$.) Then it is easy to see that for $t \in \mathbb{Z}$ with $t \geq 0$ and $x \in \mathbb{Z}$, $H^{\mathcal{A}}(\mathbf{t}, \mathbf{x})$ will be true iff the head at time t scans the square x, and similarly for $T_*^{\mathcal{A}}, \ldots, R_m^{\mathcal{A}}$. This can be proved, for example, by induction on t. Thus $R_m^{\mathcal{A}}(\mathbf{N})$ is true and this contradicts that $\mathcal{A} \models$ NONSTOP. $\dashv$

3.8 The Hilbert Tenth Problem

Using the method of reduction it has been also shown that the Hilbert 10th Problem (example **??**) is undecidable. More precisely, consider the alphabet

$$A = \{x, 0, 1, \cdot, +, -, (,), \hat{}\}.$$

Encoding the variable x_n by $x(n$ written in binary), natural numbers by their binary notation, and using $\hat{}$ for exponentiation, any polynomial in several variables with integer coefficients can be encoded by a word in this alphabet.

Example 3.8.1 $x_1^2 - 7x_2^2 - 1$ will be encoded by the word

$$(x(1)\hat{}(10) + (-111)(x(10)\hat{}(10) + (-1).$$

Let now DIOPHANTINE EQUATIONS be the decision problem (A, P) where

$$P = \{w \in A^* : w \text{ encodes a polynomial (with integer coefficients in several variables) which has an integer solution}\}.$$

We now have

Theorem 3.8.2 (Matyasevich) DIOPHANTINE EQUATIONS *is undecidable.*

So this gives a negative answer to the Hilbert 10th Problem.

3.9 Decidable Problems

We will now discuss some examples of decidable problems. We will present them informally, by giving the instances together with the question defining the set of instances that constitutes the decision problem. It will be assumed that then these can be encoded in some reasonable way as formal decision problems (A, P) as in section **??**.

Example 3.9.1 ELEMENTARY ALGEBRA is the following decision problem:

Instance. A sentence σ in the first-order language $L = \{0, 1, +, \cdot\}$.

Question. Is σ true in $\langle \mathbb{R}, 0, 1, +, \cdot \rangle$, i.e. is $\sigma \in \mathrm{Th}(\mathbb{R}, 0, 1, +, \cdot)$?

Tarski in 1949 has shown that ELEMENTARY ALGEBRA is decidable.

Remark. On the other hand, if we consider the corresponding problem ELEMENTARY ARITHMETIC for $\mathrm{Th}(\mathcal{N}, 0, S, +, \cdot, <)$, then Church has shown in the 1930's that it is undecidable (this also follows easily from **??**).

Example 3.9.2 SATISFIABILITY is the following decision problem:

Instance. A finite set $U = \{u_1, \ldots, u_k\}$ of propositional variables and $C = \{c_1, \ldots, c_m\}$ a finite set of clauses $c_i = \{\ell_{i,1}, \ldots, \ell_{i,n_i}\}$, where each $\ell_{i,j}$ is a literal from U, i.e. a variable in U or its negation. (As usual we write $\bar{u}_i$ instead of $\neg u_i$ in this context.)

Question. Is C satisfiable (i.e. is there a valuation $\nu : U \to \{T, F\}$ which makes every clause c_i true, recalling that the clause c_i as above represents the disjunction $\ell_{i,1} \vee \cdots \vee \ell_{i,n_i}$)?

Using truth tables it is clear that SATISFIABILITY is decidable.

Example 3.9.3 TRAVELING SALESMAN is the following decision problem:

Instance. A finite set $\{c_1, \ldots, c_m\}$ of "cities", a distance function $d(c_i, c_j) \in \{1, 2, \ldots\}$ for $i \neq j$, and a bound $B \in \{1, 2, \ldots\}$.

Question. Is there a tour of all the cities with total length $\leq B$, i.e. an ordering $c_{\pi(1)}, c_{\pi(2)}, \ldots, c_{\pi(m)}$ of $\{c_1, \ldots, c_m\}$ such that

$$\sum_{i=1}^{m-1} d(c_{\pi(i)}, c_{\pi(i+1)}) + d(c_{\pi(m)}, c_{\pi(1)}) \leq B?$$

Again, by listing all possible orderings and calculating the above sum for each one of these, it is trivial to see that this problem is decidable.

Example 3.9.4 PRIMES is the following decision problem:

Instance. An integer $n \geq 2$.

Question. Is n prime?

This is also decidable as we can go through all the integers between 2 and $n - 1$ and check whether they divide n.

3.10 The Class $\mathcal{P}$

We have so far been discussing what problems are computable *in principle*. However, a question of more practical interest is the following: given a problem which *is* computable, can we compute it *efficiently*? The study of efficiency is also called *computational complexity*. We will concentrate here on time complexity (how long it takes to solve the problem), but one can also discuss, for example, space complexity (how much "memory" storage is used in solving it). A natural measure of time complexity for us is the number of steps in the computation of a TM that decides a given problem.

Since different problems require different amounts of input, in order to compare the complexities of different algorithms and problems, we must measure complexity as a function of the input size. In addition, with computer speeds increasing rapidly, small instances of a problem can usually be solved no matter how difficult the problem is. For this reason, and also to eliminate the effect of "overhead" time, we will consider only the *asymptotic* complexity as the size of the input increases. First we recall the "big-O notation," which gives a rigorous way of comparing asymptotic growth rates.

Definition 3.10.1 Given two functions $f, g : \mathbb{N} \to \mathbb{N}$ we define

$$f = O(g) \text{ iff } \exists n_0 \, \exists C \, \forall n \geq n_0 (f(n) \leq Cg(n)).$$

For example, if $p(n)$ is a polynomial, then $p(n) = O(n^d)$ for large enough d.

Definition 3.10.2 Given any $T : \mathbb{N} \to \mathbb{N}$, we let TIME($T$) consist of all decision problems (A, P) which can be decided by a TM in time t for some $t = O(T)$.

More precisely, a decision problem (A, P) is in TIME(T) if there is $t = O(T)$ and a TM M on some alphabet $B \supseteq A \cup \{Y, N\}$ such that for $w \in A^*$:

(i) $w \in P \Rightarrow$ on input w, M halts in $\leq t(|w|)$ steps with output Y.

(ii) $w \notin P \Rightarrow$ on input w, M halts in $\leq t(|w|)$ steps with output N

(here $|w|$ = length of the word w).

What sort of growth rate should we require of an algorithm to consider it manageable? After all, we must expect the time to increase somewhat with the input size. It turns out that the major distinctions in complexity are between "polynomial time" algorithms and faster-growing ones, such as exponentials.

Definition 3.10.3 A decision problem is in the class $\mathcal{P}$ (or it is *polynomially decidable*) if it is TIME(n^d) for some $d \geq 0$, i.e. it can be decided in polynomial time.

Problems in the class $\mathcal{P}$ are considered *tractable* (efficiently decidable) and the others *intractable*. It is clear that the class $\mathcal{P}$ provides an upper limit for problems that can be algorithmically solved in realistic terms. If a problem is in $\mathcal{P}$, however, it does not necessarily mean that an algorithm for it can be practically implemented, for example, it could have time complexity of the order of $n^{1,000,000}$ or of the order n^3 but with enormous coefficients. However, most natural problems that have been shown to be polynomially decidable have been found to have efficient (e.g., very low degree) algorithms. Moreover, the class of problems in $\mathcal{P}$ behaves well mathematically, and is independent of the model of computation, since any two formal models of computation can be mutually simulated within polynomial time.

Remark. Time complexity as explained here is a worst case analysis. If a problem P is intractable, then there is no polynomial time algorithm which for all n and all inputs of length n will decide P. But one can still search for approximate algorithms that work well on the average or for most practical (e.g., small) instances of the problem or give the correct answer with high probability, etc.

Example 3.10.4 ELEMENTARY ALGEBRA is intractable (Fisher-Rabin 1974). In fact it is in TIME($2^{2^{cn}}$) for some $c > 0$ but not in TIME($2^{2^{dn}}$) for any $d > 0$; that is, it is *super-exponential*.

Example 3.10.5 LINEAR PROGRAMMING is the decision problem given by:

Instance. An integer matrix (v_{ij}) for $1 \leq i \leq m$ and $1 \leq j \leq n$, along with integer vectors $D = (d_i)_{i=1}^m$ and $C = (c_j)_{j=1}^n$, and an integer B.

Question. Is there a rational vector $X = (x_j)_{j=1}^n$ such that $\sum_{j=1}^n v_{ij}x_j \leq d_i$, for $1 \leq i \leq m$, and $\sum_{j=1}^n c_j x_j \geq B$?

LINEAR PROGRAMMING turns out to be in $\mathcal{P}$ (Khachian, 1979).