DECIDABILITY.

CONTENTS

1. UNIVERSAL TURING MACHINES AND UNDECIDABILITY OF THE HALTING PROBLEM.

Let us first recall a few basic notions.
Let us consider a Turing Machine M and a word w in Σ^*. If given the input word w, M enters a final state, after performing zero or more transitions, and then M halts immediately afterwards, we say that M *accepts* w (or M *answers* "yes"). If given the input w, M halts after performing zero or more transitions, without ever entering a final state, we say that M *rejects* w (or M *answers* "no").

NOTE. The following two notions of acceptance of a language $L \subseteq \Sigma^*$ by a Turing Machine M are equivalent:
(1) for all words $w \in \Sigma^*$, $w \in L$ iff given the input w, M performs zero or more transitions and then enters a final state (nothing is said about what happens afterwards: it may be the case that M halts in that final state *or* M performs a finite number of transitions and then halts *or* M performs a infinite number of transitions, i.e., it never halts);
(2) for all words $w \in \Sigma^*$, $w \in L$ iff given the input w, M performs zero or more transitions, enters a final state, and then halts immediately afterwards in that final state. ■

We say that a language $L \subseteq \Sigma^*$ is *recursively enumerable* (r.e., for short), or L is a recursive enumerable subset of Σ^*, iff there exists a Turing Machine which *accepts* (or *recognizes*) L, that is, iff there exists a Turing Machine M such that for all words $w \in \Sigma^*$, $w \in L$ iff M accepts w. Thus, if M recognizes L and $w \notin L$ then given the input w, *either* M halts and does not answers "no" *or* M does not halt.

The language accepted (or recognized) by a Turing Machine M is denoted by L(M).

Given any recursively enumerable language L, there are $|N|$ Turing Machines each of which accepts (or recognizes) L.

We say that a language $L \subseteq \Sigma^*$ is *recursive*, or L is a recursive subset of Σ^*, iff there exists a Turing Machine M which always halts and accepts (or recognizes) L, that is, iff there exists a Turing Machine M such that for all words $w \in \Sigma^*$, given the input w, M enters a final state and then halts immediately afterwards in that final state iff $w \in L$. Thus, if L is recursive there exists a Turing Machine M such that for all words $w \in \Sigma^*$, M accepts the word w iff $w \in L$, and M rejects the word w iff $w \notin L$.

If a language L is not r.e. then L is not recursive, because the class of recursive languages is contained in the class of the r.e. languages.

In the sequel when we say that a language L is r.e., either (i) we mean that L is r.e. and it is not recursive, or (ii) we mean that L is r.e. and we say nothing about L being recursive or not. The reader should be careful in distinguishing between these two meanings.

Informally, we say that a problem is a *yes-no problem* if it can be viewed as a set of so called *instances*, and with each of these instances we associate a corresponding answer which is either “yes” or “no”.

For example, the Primality Problem is a yes-no problem, because it can be viewed as the set of natural numbers, each natural number being associated with a “yes” or “no” answer which tells us whether or not that number is a prime. Thus, an instance of the Primality Problem is simply a natural number.

Often we will say “problem”, instead of “yes-no problem”.

For any yes-no problem if there exists a computing device, such as a Turing Machine, which given an instance of that problem, always halts and provides the corresponding answer, we say that the yes-no problem is *solvable* or *decidable*.
A yes-no problem is said to be *unsolvable* or *undecidable* iff it is not solvable.

It may also be the case that given a yes-no problem, there exists a Turing Machine, say M, which halts and provides answers “yes” exactly for those instances of the problem which have answer “yes”. For the instances which have the answer “no”, M either does not halt or it halts and answers “no”. In this case we say that the yes-no problem is *semisolvable* or *semidecidable*.

With every semisolvable yes-no problem we may associate a Turing Machine and since this machine works on input words taken from some suitable set, say Σ^*, every instance of a yes-no problem should be encoded as a word of Σ^*.

Thus, given a problem P, we may define the associated *yes-language* L_P, subset of Σ^*, made out of every word which encodes an instance of the problem which is associated to the answer “yes”.

For example, in the case of the Primality Problem, we may choose Σ^* to be 01^*, because we may encode any given natural number n (≥ 0) into the word 01^n. The yes-language associated with the Primality Problem is $\{01^n \mid n \geq 0 \text{ and n is a prime}\}$.

Let us now look at a different yes-no problem, the so called Parsing Problem. We are given a grammar G, encoded as a word in $\{0,1\}^*$, which we will denote by [G], and a word w made out of terminal symbols of G. We look for the answer “yes” or “no” depending on whether or not w belongs to the language L(G) generated by G. (A way of encoding a grammar into a word in $\{0,1\}^*$ is to encode its productions, in the same way as a Turing Machine can be encoded into a word in $\{0,1\}^*$ by encoding its quintuples.)

The yes-language L_P associated with the Parsing Problem for the grammar G is the set $\{[G]\$w \mid w \text{ belongs to } L(G)\}$. Notice that, as it stands, L_P is a subset of the language $\{0,1\}^*\$T^*$, where T is the set of terminal symbols of G. (As usual, we have used the concatenation of regular expressions to mean the concatenation of the corresponding languages.) In order to transform L_P into a subset of $\{0,1\}^*$ we need to encode \$ and the elements of T as words in $\{0,1\}^*$.

Care should be taken when considering the encoding of the instances of a yes-no problem. We will not discuss this issue here. We want only to remark that the encoding has to be constructive (i.e., computable by a Turing machine which always halts), and it should not modify the solution of the problem itself (and this could happen when the problem is of syntactic nature as, for instance, when the problem depends on the syntactic way in which it is expressed in a particular language).

DEFINITION 1. A yes-no problem is said to be *decidable* iff under some suitable encoding of its instances into words of Σ^*, the associated yes-language is a recursive subset of Σ^*. ■

According to this definition every yes-no problem which has a finite number of instances is decidable.

In what follows we need the following facts.

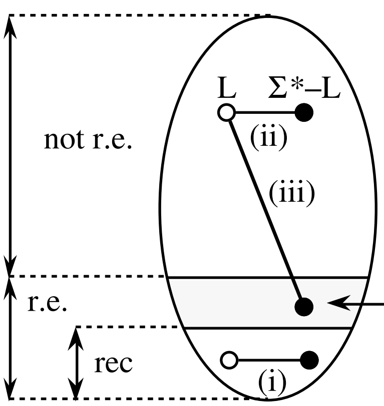
(i) The complement $\Sigma^* - L$ of a recursive set L is recursive.
In order to show this, it is enough to use the Turing Machine M for recognizing L. If M halts and answers "yes" (or "no") the Turing Machine M1 which recognizes $\Sigma^* - L$, halts and answers "no" (or "yes", respectively).

(ii) The union of two recursive languages is recursive. The union of two r.e. languages is r.e.

(iii) If the complement of an r.e. language L is r.e. then L is recursive (Post's Theorem).
It is enough to have the Turing Machine, say M1, for recognizing L, and the one, say M2, for recognizing $\Sigma^* - L$, running in parallel so that they perform their computations in an interleaved way. This means that, initially, M1 makes one computation step, then M2 makes one computation step, then M1 makes one more computation step, and so on, in an alternate way. One of the two machines will eventually halt (because given a word $w \in \Sigma^*$ either $w \in L$ or $w \notin L$), and it will give us the required answer. For instance, if it is M1 (or M2) which halts first and answers "yes", then the answer is "yes" (or "no", respectively).

Thus, given any set $L \subseteq \Sigma^*$ there are four mutually exclusive possibilities only:

Set of subsets of Σ^* :



(i) L is recursive and $\Sigma^* - L$ is recursive, or
(ii) L is not r.e. and $\Sigma^* - L$ is not r.e., or
(iii.1) L is r.e. and not recursive and $\Sigma^* - L$ is not r.e., or
(iii.2) L is not r.e. and $\Sigma^* - L$ is r.e. and not recursive.

As a consequence, in order to show that an r.e. language L is not recursive (and the associated problem is undecidable) it is enough to show that $\Sigma^* - L$ is not r.e.

Another technique for showing that a language is not recursive (and the associated problem is undecidable) is the *reduction technique*.

We say that a problem A whose associated yes-language is L_A, subset of Σ^*, is reduced to a problem B whose associated yes-language is L_B, also subset of Σ^*, iff there exists a total function, say r, from L_A to L_B such that for every word w in Σ^*, w is in L_A iff r(w) is in L_B.

Thus, if the problem B is decidable then the problem A is decidable, and if the problem A is undecidable then the problem B is undecidable.

Let us now consider the yes-no problem, called the *Halting Problem*. This problem can be formulated as follows. Given a Turing Machine M and a word w it should be determined whether or not M halts on the input w. The various instances of this problem are given by the instances of M and w.

We recall that given a language $R \subseteq \{0,1\}^*$ recognized by some Turing Machine M, there exists a Turing Machine M1 with tape alphabet $\{0,1,B\}$ such that R is recognized by M1.

Thus, if we prove that the Halting Problem is undecidable for Turing Machines with tape alphabet $\{0,1,B\}$ and words in $\{0,1\}^*$, then the Halting Problem is undecidable for any class of Turing Machines and words which includes this one.

We will show that the Halting Problem for one-tape Turing Machines, which are *binary*, that is, with tape alphabet $\{0,1,B\}$ and input words in $\{0,1\}^*$, is undecidable.

In order to do so, we first show that a Turing Machine, say M, with tape alphabet $\{0,1,B\}$ can be encoded into a word in $\{0,1\}^*$. Without loss of generality, we may assume that M is deterministic.

Let us assume that:

- the set of states of M is $\{q_i \mid i \geq 1\}$,
- the tape symbols 0, 1, and B are denoted by X_1, X_2, and X_3, respectively, and
- L (that is, the move to the left) and R (that is, the move to the right) are denoted by 1 and 2, respectively.

The initial and final states are assumed to be q_1 and q_2, respectively. Without loss of generality, we may also assume that there exists one final state only.

Then, each quintuple, say "q_i, $X_h \to q_j$, X_k, m", of M corresponds to a string of five positive numbers $\langle i, h, j, k, m \rangle$. It should be the case that $i,j \geq 1$, $1 \leq h,k \leq 3$, and $1 \leq m \leq 2$. Thus, the quintuple "q_i, $X_h \to q_j$, X_k, m" can be encoded by the sequence: $1^i 0 1^h 0 1^j 0 1^k 0 1^m$.

The various quintuples can be listed one after the other, in any order, so to get a sequence of the form:

000 code of the first quintuple 00 code of the second quintuple 00 … 000. (†)

Every sequence of the form (†) encodes one Turing Machine only, while a Turing Machine can be encoded by several sequences (recall that, for instance, when describing a Turing Machine the order of the quintuples is not significant).

We assume that the sequences of 0's and 1's which are not of the form (†) are encodings of a Turing Machine which recognizes the empty language, that is, a Turing Machine which does not accept any word in $\{0,1\}^*$.

Let us now consider the language:

$L_U = \{[M]w \mid \text{the Turing Machine M accepts the word } w\} \subseteq \{0,1\}^*$,

where [M] is the encoding of M as a word in $\{0,1\}^*$, w is a word in $\{0,1\}^*$, and [M]w is the concatenation of [M] and w. The leftmost symbol of the word w is immediately to the right of the three 0's in the rightmost position of [M].

THEOREM 2. L_U is recursive enumerable.

PROOF. We show that L_U is recursively enumerable by showing that there exists a binary Turing Machine M_U which recognizes L_U.

Indeed, we show the existence of a Turing Machine, say T, with three tapes, called Tape1, Tape2, and Tape3, respectively, which recognizes L_U. Then from T we construct the equivalent binary Turing Machine M_U, called *universal Turing Machine*, having one one-way infinite tape and 0, 1, and B as tape symbols.

The machine T works as follows.

Initially T has in Tape1 a sequence s in $\{0,1\}^*$. Then T checks whether or not s is equal to $s_1 s_2$ for some s_1 of the form (†).

If this is not the case, then T halts without entering a final state, and this indicates that M does not accept w (recall that we have assumed that a sequence not of the form (†) encodes a Turing Machine which recognizes the empty language).

Otherwise, if the sequence s can be viewed as the concatenation of a sequence of the form (†) which encodes a Turing Machine, say M, followed by a word w in $\{0,1\}^*$, then T writes w in its Tape2 and '1' in its Tape3 (denoting the fact that M starts its computation in its initial state q_1). The machine T continues working by simulating the behaviour of the machine M whose quintuples are on Tape1, on the input word w which is on Tape2, storing the information about the states of M on Tape3. In order to recall which symbol of the word w is currently scanned by M, the machine T uses a second track of Tape2. The only final state of M is q_2, which is encoded by 11.

(α) Knowing the current state, say q_i, and the scanned symbol, say X_h, T checks whether or not 11 (denoting the accepting state q_2) is on the Tape3. If this is the case then T stops and accepts the word w. If it is not the case then T looks for a quintuple on the first tape of the form: $001^i01^h01^j01^k01^m00$, and

(α1) if there is one quintuple of that form then T makes the corresponding move by (i) placing 1^j on Tape3, (ii) writing 0 or 1 on the scanned cell of Tape2, if the value of k is 1 or 2, respectively, and (iii) changing the head marker on the second track of Tape2 to the left or to the right of the current one, if the value of m is 1 or 2, respectively. Then, T continues working as specified at Point (α) looking at the current state on Tape3.

(α2) If it does not exist a quintuple of the form $001^i01^h01^j01^k01^m00$ then T stops and it does not accept the input word because 11 is not on Tape3.

It should be clear from the above construction that the machine T simulates the machine M on the input w in the sense that: (i) T halts and accepts [M]w iff M halts and accepts w, (ii) T halts and does not accept [M]w iff M halts and does not accept w, and (iii) T does not halt iff M does not halt. ∎

THEOREM 3. L_U is not recursive.
PROOF. We show that L_U is not recursive, by showing first that the language L_D, we will define below, is not recursive. We will show that L_D is not recursive by showing that $\Sigma^* - L_D$ is not r.e. using a diagonal argument.

Then, by reducing L_D to L_U we will derive that L_U is not recursive.

The language L_D is constructed as follows. Let us consider a matrix W whose rows and columns are labeled by the words of $\{0,1\}^*$ in the canonical order, i.e., ε, 0, 1, 00, 01, 10, 11, 000, … In position of row i and column j we write "1" if the Turing Machine whose code is the i-th word in the canonical order, accepts the j-th word in the canonical order. (Recall that we have assumed that a sequence not of the form (†) encodes a Turing Machine which recognizes the empty language.) Otherwise we write "0".

Let L_D be $\{w \mid w$ is i-th word of the canonical order and $W[i,i] = 1\} \subseteq \{0,1\}^*$. That is, w belongs to L_D iff the Turing Machine with code w accepts the word w.

We now show that $\Sigma^* - L_D$ is not r.e. Indeed, we have that:
(i) w belongs to $\Sigma^* - L_D$ iff the Turing Machine with code w, does *not* accept w, and
(ii) if $\Sigma^* - L_D$ is r.e. then there exists a Turing Machine, say M, with code [M], such that M recognizes $\Sigma^* - L_D$.

Let us now ask ourselves whether or not the Turing Machine with code [M], accepts the word [M]. We have that the Turing Machine with code [M], accepts the word [M] iff (by definition of $\Sigma^* - L_D$) the Turing Machine whose code is [M] does not accepts [M].

This contradiction shows that $\Sigma^* - L_D$ is not r.e. Thus, L_D is not recursive.

To get the thesis, now it is enough to reduce L_D to L_U. We do this as follows. Given a word w in Σ^*, we compute the word ww (the first w plays the role of the code of a Turing Machine, and the second w is the word to be given as input to that Turing Machine). By construction we have that $w \in L_D$ iff $ww \in L_U$.

Since L_D is not recursive, we get that L_U is not recursive. ■

We have that the problem corresponding to the language L_U, called the Halting Problem, is not decidable, because L_U is not recursive.

Thus, to say that the Halting Problem is undecidable means that it does not exists any Turing Machine that: (i) halts for every input, and (ii) given the encoding [M] of a Turing Machine M and an input word w for M, answers "yes" iff M halts in a final state when starting initially with w on its tape.

By reduction of the Halting Problem to each of the following problems, one can show that these problems are undecidable:
(i) Blank Tape Halting Problem: given a Turing Machine M, determine whether or not M halts in a final state when its initial tape has blank symbols only,
(ii) Uniform Halting Problem (or Totality Problem): given a Turing Machine M, determine whether or not M halts in a final state for every input word in Σ^*, and
(iii) Printing Problem: given a Turing Machine M and a tape symbol b, determine whether or not M prints b during any of its computations.

Notice that it is undecidable whether or not a given binary Turing Machine M ever writes three consecutive 1's on its tape (the various instances of the problem being given by the choice of M), while it is decidable whether or not, given a binary Turing Machine M starting on a blank tape, there exists a cell scanned by M five or more times (the various instances of the problem being given by the choice of M) [Hopcroft-Ullman 79, page 193].

2. RICE'S THEOREM.

Let us consider the alphabet Σ and the set RE whose elements are the recursive enumerable languages, i.e., the recursive enumerable subsets of $\Sigma^* = \{0,1\}^*$.

Let us consider a subset S of RE, that is, a set of r.e. languages. We say that S is a *property* over RE. The property S is said to be *trivial* iff S is empty or S is RE itself.

Any subset S of RE corresponds to a yes-no problem and an instance of that problem is given by an r.e. language for which we want to know whether or not it is S. We need an encoding of an r.e. language as a word in $\{0,1\}^*$. This can be done by using the encoding of the Turing Machine which recognizes that r.e. language. Obviously, an r.e. language is given as input to a Turing Machine via the encoding of the Turing Machine which recognizes it.

The property $\varnothing$ (that is, the empty subset of RE) corresponds to the yes-no problem 'Is a given r.e. language not an r.e. language?', and the property RE corresponds to the yes-no problem 'Is a given r.e. language an r.e. language?'.

Now we ask ourselves whether or not a given set S, subset of RE, is itself r.e., i.e., whether or not there exists a Turing Machine M_S corresponding to S, such that given any r.e. set $X \subseteq \Sigma^*$, M_S accepts X iff $X \in S$.

THEOREM 4 (*Rice's Theorem for recursive properties over RE*). A set S of languages, subset of RE, is not recursive iff it is not trivial (that is, the yes-no problem of determining whether or not an r.e. language belongs to S is undecidable iff $S = \varnothing$ or $S = RE$).

PROOF. Let us consider a non-trivial subset S of languages in RE. We will prove that S is not recursive.

We may consider, without loss of generality, that the empty language (which is obviously an r.e. language) is not in S. Indeed, if the empty language belongs to S then we will first prove that RE – S is not recursive (and we have that the empty language does not belong to RE – S). Thus, if RE – S is not recursive we have that S is not recursive either. (Recall that if a set is not recursive, its complement is not recursive.)

Since S is not empty (because S is not trivial) we have that an r.e. language, say L, belongs to S. Let us consider the Turing Machine M_L which recognizes the language L.

The proof of Rice's Theorem is by reduction of the Halting Problem (that is, the membership problem for the language $L_U = \{[M]w \mid \text{the Turing Machine M accepts the word } w\}$) to the problem of determining whether or not any given r.e. language belongs to S.

This reduction is done by showing that for any given Turing Machine M and any given word w there exists an r.e. language, say A, such that M accepts w iff A belongs to S, that is, for any given Turing Machine M and any given word w there exists a Turing Machine, say N_A, such that M accepts w iff the language accepted by N_A belongs to S.

Indeed, let us consider the following Turing Machine N_A (see also Fig. 1).

For any given input word v, N_A simulates the Turing Machine M on the input word w. If M accepts w then N_A starts working on the input v as the machine M_L. If M does not accept w then N_A does not accept v.

Thus, if M accepts w then N_A accepts L (which is a language in S), and if M does not accept w then N_A accepts no word, that is, N_A accepts the empty language (which is a language not in S). Therefore, M accepts w iff the language accepted by N_A belongs to S.

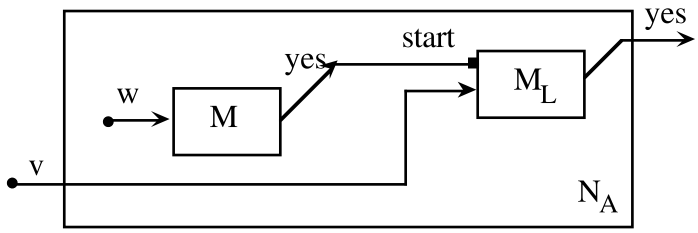


Figure 1. Construction of the Turing Machine N_A.
If M accepts w then N_A accepts the language L (which is in S).
If M does not accept w then N_A accepts the empty language (which is not in S).

From the above reduction of the Halting problem to the problem of determining whether or not any given r.e. language A belongs to S, and from the fact that the Halting Problem is undecidable (because the language L_U is not recursive) we get that the set S is not recursive. ■

As simple corollaries of Rice's Theorem, we have that:
(i) the set $\{[M] \mid [M]$ is an encoding of a Turing Machine M such that L(M) is empty$\}$ is not recursive,
(ii) the set $\{[M] \mid [M]$ is an encoding of a Turing Machine M such that L(M) is finite$\}$ is not recursive,
(iii) the set $\{[M] \mid [M]$ is an encoding of a Turing Machine M such that L(M) is regular$\}$ is not recursive, and
(iv) the set $\{[M] \mid [M]$ is an encoding of a Turing Machine M such that L(M) is context-free$\}$ is not recursive.

We state without proof the following result which is the analogous of the Rice's Theorem with respect to the r.e. properties, instead of the recursive properties.

THEOREM 5 (*Rice's Theorem for r.e. properties over RE*). A subset S of r.e. languages in RE is r.e. iff S satisfies the following three conditions:
(1) (r.e. upward closure) if $L1 \subseteq \{0,1\}^*$ is in S and $L1 \subseteq L2$ for some L2 which is r.e., then L2 is in S,
(2) (existential finite downward closure) if L is an infinite r.e. language in S, then there exists some finite r.e. language L_f subset of L such that L_f is in S, and
(3) the set of the finite languages which are elements of S, is r.e. ■

We have the following simple corollaries of Rice's Theorem for r.e. properties over RE:

(i) the set $\{[M] \mid [M]$ is an encoding of a Turing Machine M such that L(M) is empty$\}$ is *not* r.e. (because condition (1) does not hold),
(ii) the set $\{[M] \mid [M]$ is an encoding of a Turing Machine M such that L(M) is $\Sigma^*\}$, is *not* r.e. (because condition (2) does not hold),
(iii) the set $\{[M] \mid [M]$ is an encoding of a Turing Machine M such that L(M) is recursive$\}$ is

not r.e. (because condition (1) does not hold),

(iv) the set $\{[M] \mid [M]$ is an encoding of a Turing Machine M such that L(M) is not recursive$\}$ is *not* r.e. (because condition (1) does not hold),

(v) the set $\{[M] \mid [M]$ is an encoding of a Turing Machine M such that L(M) is regular$\}$ is *not* r.e. (because condition (1) does not hold), and

(vi) the set $\{[M] \mid [M]$ is an encoding of a Turing Machine M such that L(M) is made out of one word only$\}$ is *not* r.e. (because condition (1) does not hold), and

(vii) the set $\{[M] \mid [M]$ is an encoding of a Turing Machine M such that $L(M) - L_U$ is not empty$\}$ is *not* r.e. (because condition (3) does not hold. Indeed, take a finite set, say D, with empty intersection with L_U. The set $\Sigma^* - L_U$ is a superset of D and yet it is not r.e., because L_U is r.e. and not recursive).

We also have the following corollaries:

(viii) the set $\{[M] \mid [M]$ is an encoding of a Turing Machine M such that L(M) is not empty$\}$ is r.e. (and we also have that it is not recursive),

(ix) for any fixed k, the set $S_k = \{[M] \mid [M]$ is an encoding of a Turing Machine M such that L(M) has at least k elements$\}$ is r.e.,

(x) for any fixed word w, the set $S_w = \{[M] \mid [M]$ is an encoding of a Turing Machine M such that L(M) has at least the word w$\}$ is r.e.,

(xi) the set $\{[M] \mid [M]$ is an encoding of a Turing Machine M such that $L(M) \cap L_U$ is not empty$\}$ is r.e.

3. POST'S CORRESPONDENCE PROBLEM. UNDECIDABILITY AND DECIDABILITY PROPERTIES FOR CONTEXT-FREE LANGUAGES.

An instance of Post's Correspondence Problem, PCP for short, over the alphabet Σ, is given by two sequences of k words each, say $<u_1, \ldots, u_k>$ and $<v_1, \ldots, v_k>$, where the u_i's and the v_i's are elements of Σ^*. A given instance of the PCP has a solution iff there exists a sequence of indices taken from the set $\{1,\ldots,k\}$, say $<i1,\ldots, in>$, for $n \geq 1$, such that this equality between two words of Σ^*, holds:

$$u_{i1} \ldots u_{in} = v_{i1} \ldots v_{in}.$$

One can show that Post's Correspondence Problem is unsolvable (if the instances are taken using Σ and k such that $|\Sigma| \geq 2$ and $k \geq 2$) by showing that the Halting Problem can be reduced to it.

There a variant of Post's Correspondence Problem, called the Modified Post's Correspondence Problem, where it is assumed that in the solution sequence i1 is 1. Also the Modified Post's Correspondence Problem is unsolvable.

If the alphabet Σ has one symbol only, then Post's Correspondence Problem is decidable. Indeed, let us denote by $|u|$ the number of symbols of Σ in a word u.

If there exists i, with $1 \leq i \leq k$, such that $u_i = v_i$ then the sequence $<i>$ is the solution. If for all i, $1 \leq i \leq k$, $u_i \neq v_i$ then there are two cases: (i) either for all i, $|u_i| > |v_i|$, or for all i, $|u_i| < |v_i|$, and (ii) there exist i and j, with $1 \leq i,j \leq k$, such that $|u_i| > |v_i|$ and $|u_j| < |v_j|$.

In case (i) there is no solution to Post's Correspondence Problem, and in case (ii) there is a solution which is given by the sequence of p (≥1) i's followed by q (≥1) j's such that:

$p\,(|u_i| - |v_i|) + q\,(|u_j| - |v_j|) = 0.$
We can always take p to be $-(|u_j| - |v_j|)$ and q to be $|u_i| - |v_i|$.

If k = 1 then Post's Correspondence Problem is decidable, and it has a solution iff $u_1 = v_1$.

Using the undecidability of the PCP, the following theorem shows that it is undecidable whether or not a given context-free grammar G is ambiguous, that is, it is undecidable whether or not there exists a word w generated by G such that w has two distinct leftmost derivations.

THEOREM 6. The ambiguity problem for context-free languages is undecidable.
PROOF. It is enough to reduce the PCP to the ambiguity problem of context-free grammars. Consider a finite alphabet Σ and two sequences of k (≥1) words, each word being an element of Σ^*:

$U = \langle u_1, \ldots, u_k\rangle$, and
$V = \langle v_1, \ldots, v_k\rangle$.

Let us also consider the set A of k new symbols $\{a_1, \ldots, a_k\}$ such that $\Sigma \cap A \neq \emptyset$, the following two languages which are subsets of $(\Sigma \cup A)^*$:

$UL = \{u_{i1}\, u_{i2} \ldots u_{ir}\, a_{ir} \ldots a_{i2}\, a_{i1} \mid r \geq 1 \text{ and } 1 \leq i1,i2,\ldots,ir \leq k\}$, and
$VL = \{v_{i1}\, v_{i2} \ldots v_{ir}\, a_{ir} \ldots a_{i2}\, a_{i1} \mid r \geq 1 \text{ and } 1 \leq i1,i2,\ldots,ir \leq k\}$.

A grammar G for generating the language $UL \cup VL$ is as follows: $\langle \Sigma \cup A, \{S, S_U, S_V\}, P, S\rangle$, where P is the set of productions:

$S \to S_U$, $\quad S_U \to u_i\, S_U\, a_i \mid u_i\, a_i \quad$ for any i = 1, …, k, and
$S \to S_V$, $\quad S_V \to v_i\, S_V\, a_i \mid v_i\, a_i \quad$ for any i = 1, …, k.

To prove the theorem we now need to show that the instance of the PCP for the sequences U and V has a solution iff the grammar G is ambiguous.
(⇒) If $u_{i1} \ldots u_{in} = v_{i1} \ldots v_{in}$ for some n≥1, then the word $w = u_{i1}\, u_{i2} \ldots u_{in}\, a_{in} \ldots a_{i2}\, a_{i1}$ is equal to $v_{i1}\, v_{i2} \ldots v_{in}\, a_{in} \ldots a_{i2}\, a_{i1}$, and w has two leftmost derivations: one derivation which first uses the production $S \to S_U$, and a second one, which first uses the production $S \to S_V$. Thus, G is ambiguous.
(⇐) Assume that G is ambiguous. Then there are two leftmost derivations for a word generated by G. Since every word generated by S_U has one leftmost derivation only, and every word generated by S_V has one leftmost derivation only (and this is due to the fact that the a_i's symbols force the uniqueness of the productions used when deriving a word from S_U or S_V), it must be the case that a word generated from S_U is the same as a word generated from S_V. This means that we have:

$$u_{i1}\, u_{i2} \ldots u_{in}\, a_{in} \ldots a_{i2}\, a_{i1} = v_{i1}\, v_{i2} \ldots v_{in}\, a_{in} \ldots a_{i2}\, a_{i1}$$

for some sequence <i1,…, in> of indexes with n≥1, each index being from the set {1,…,k}.

Thus,

$$u_{i1}\ldots u_{in} = v_{i1}\ldots v_{in},$$

and this means that the corresponding PCP has the solution $\langle i1,\ldots, in\rangle$. ■

We will now consider some more undecidability results about context-free languages. They can be proved using properties of the so-called "valid and invalid computations" of Turing Machines.

Given a Turing Machine $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$, a string of the form

$w_1\#w_2^R\#w_3\#w_4^R\#w_5\#\ldots\#w_n$ or

$w_1\#w_2^R\#w_3\#w_4^R\#w_5\#\ldots\#w_{n-1}\#w_n^R$,

where by the word w^R we mean "w reversed", is said to be a *valid computation* of M iff:

(i) each w_i is a configuration of M, that is, a string in $\Gamma^*Q\Gamma^*$, which does not end by a B,
(ii) w_1 is the initial configuration, that is, $w_1 = q_0u$ where u belongs to Σ^*,
(iii) w_n is a final configuration, that is, a string in $\Gamma^*F\Gamma^*$, and
(iv) for each i, $1 \leq i < n$, $w_i \rightarrow_M w_{i+1}$.

We also assume that # is not in $Q \cup \Gamma$, and $Q \cap \Gamma = \varnothing$.

A Turing Machine *invalid computation* is a string which is *not* a Turing Machine valid computation.

We have that:
(i) given a Turing Machine M the set of its valid computations is the intersection of two context-free languages (and the two corresponding context-free grammars can be effectively constructed from M), and
(ii) given a Turing Machine M the set of its invalid computations is a context-free language (and the corresponding context-free grammar can be effectively constructed from M).

One can show the following facts.
(A) It is undecidable whether or not given two context-free grammars, say G_1 and G_2, we have that $L(G_1) \cap L(G_2)$ is empty. The proof is based on the fact that the problem of determining whether or not a given Turing Machine recognizes the empty language, can be reduced to this problem. (Recall Property (i) of the valid computations of a Turing Machine.)

(B) It is undecidable whether or not given two context-free grammars they generate the same context-free language.

(C) It is undecidable whether or not a given context-free grammar generates Σ^+ (or Σ^* if we assume the possibility of the production $S \rightarrow \varepsilon$).

It is decidable whether or not two deterministic context-free languages generate the same language (ICALP 1997: paper by Géraud Sénizergues).

Languages which are generated by LL(k) grammars for some k≥0, are deterministic context-free languages, but not all deterministic context-free languages are generated by LL(k) grammars. For those languages we have that, given two LL(k) grammars G_1 and G_2, for some k≥0, it is decidable whether or not $L(G_1) = L(G_2)$.

Given the context-free grammar G, it is undecidable whether or not L(G) is a linear context-free language, that is, it can be generated by a context-free grammar whose productions have right hand sides with at most one occurrence of a non-terminal.

It is undecidable whether or not a context-sensitive grammar generates a context-free language. [Aho-Ullman: Theory of Parsing, Translation, and Compiling. Vol. I, 1972, page 208]

It is undecidable whether or not a context-free grammar generates a deterministic context-free language.

A language L is said to be *prefix-free* (or to have the *prefix property*) iff no word in L is a proper prefix of another word in L.

It is undecidable whether or not a context-free grammar generates a prefix-free language [Harrison 78, page 262].

It is decidable whether or not given k (≥ 0) and a context-free grammar G, G is LR(k).

It is decidable whether or not given k (≥ 0) and a context-free grammar G, G is LL(k).

It is undecidable whether or not given a context-free grammar G there exist k (≥ 0) and an LR(k) grammar G1 such that G and G1 generate the same language.

It is undecidable whether or not given a context-free grammar G there exist k (≥ 0) and an LL(k) grammar G1 such that G and G1 generate the same language.

DECIDABLE PROPERTIES OF DETERMINISTIC CONTEXT-FREE LANGUAGES, WHICH ARE UNDECIDABLE FOR CONTEXT-FREE LANGUAGES

Let Σ be a given alphabet. We assume that $|\Sigma| \geq 2$. We list now some decidable properties for deterministic context-free languages, which are undecidable for context-free languages.

Given a deterministic context-free grammar G whose corresponding language is L, and a given regular language R, it is decidable whether or not:

D1) $L = R$,
D2) $R \subseteq L$,
D3) $L = \Sigma^*$ (that is, the complement of L is empty),
D4) $\Sigma^* - L$ is a context-free language,

D5) L is a regular language (that is, it is decidable whether or not there exists a regular langauge R1 such that L =R1).

D6) It is decidable whether or not a deterministic context-free grammar G generates a language which is prefix-free [Harrison 78, page 355]. ■

UNDECIDABLE PROPERTIES OF DETERMINISTIC CONTEXT-FREE LANGUAGES, WHICH ARE UNDECIDABLE ALSO FOR CONTEXT-FREE LANGUAGES

We list below some undecidable properties for deterministic context-free languages which are undecidable also for context-free languages.

Given the deterministic context-free grammars G_1 and G_2, whose corresponding languages are L_1 and L_2, respectively, it is undecidable whether or not:

U1) $L_1 \cap L_2 = \varnothing$,
U2) $L_1 \subseteq L_2$,
U3) $L_1 \cap L_2$ is a deterministic context-free language,
U4) $L_1 \cup L_2$ is a deterministic context-free language,
U5) $L_1\, L_2$ is a deterministic context-free language ,
U6) $L_1{}^*$ is a deterministic context-free language,
U7) $L_1 \cap L_2$ is a context-free language. ■

Given the context-free grammar G = <T, V, P, S> (where T is the set of terminal symbols, V is the set of non-terminal symbols, P is the set of productions, and S is the axiom), it is decidable whether or not L(G) is infinite. Indeed, it is enough to derive an equivalent grammar, say CG, in Chomsky normal form, without useless symbols, and to construct a directed graph whose nodes are the non-terminals of CG.

(Recall that a symbol X in the set V of the non-terminal symbols, is useful iff there is a derivation $S \rightarrow^* \alpha X \beta \rightarrow^* w$ for some α, β in $(T \cup V)^*$, and some w in T^*. X is useless iff it is not useful.)

In that graph we then construct an edge from node n_1 to node n_2 iff there exists a production in CG such that $n_1 \rightarrow n_2 v$ or $n_1 \rightarrow v n_2$ for some non-terminal v.

L(G) is infinite iff in that graph there exists a cycle ([Hopcroft-Ullman 79], page 137).

We now present a theorem which is the analogous of Rice's Theorem for context-free languages. We need first two definitions.

DEFINITION 7. We say that a class *C* of languages is *effectively closed* under concatenation with regular sets and union iff given the encoding (as a string in $\{0,1\}^*$) of any two languages, say L_1 and L_2, in *C*, and the encoding of a regular language, say R, we have that the languages (i) $R \bullet L_1$, (ii) $L_1 \bullet R$, and (iii) $L_1 \cup L_2$ are in *C*, and it is possible to produce the encodings of those languages. ■

DEFINITION 8. Given an alphabet Σ, a language $L \subseteq \Sigma^*$, and a symbol $b \in \Sigma$, we say that the set $\{w \mid wb \in L\}$ is the *quotient language* of L w.r.t. b. ■

THEOREM 9 (*Greibach's Theorem*). Let us consider a class C of languages which is effectively closed under concatenation with regular sets and union. Let us assume that for that class C the problem of determining whether or not $L = \Sigma^*$ for any sufficient large cardinality of Σ, is undecidable. Let P be a non-trivial property of C, that is, P is a non-empty subset of C different from C itself.

If P holds for all regular sets and it is preserved under quotient w.r.t. any symbol in Σ, then P is undecidable for C. ■

We have that:
- the class of context-free languages is effectively closed under concatenation with regular sets and union, and for context-free languages it is undecidable the problem of determining whether or not $L = \Sigma^*$ for $|\Sigma| \geq 2$,
- the class of regular languages is a non-trivial subset of the context-free languages,
- the property of being a regular language obviously holds for all regular languages, and
- the class of regular languages is closed under quotient w.r.t. any symbol in Σ (it is enough to delete the final symbol in the corresponding regular expression).

Thus, by Theorem 9, it is undecidable whether or not a given context-free grammar defines a regular language (see property D5 above).

Theorem 9 allows us to show that also inherent ambiguity for context-free languages is undecidable. We recall that a context-free language L is said to be inherently ambiguous iff every context-free grammar G generating L is ambiguous (that is, there is a word of L which has two distinct leftmost derivations according to G).

4. ARITHMETICAL HIERARCHY.

Let us consider properties, that is, subsets, of the set of natural numbers N. We may also consider properties of arity n>1, in which case they can be viewed as subsets of N^k. However, by the bijection between N^k and N, we may consider any property of arity n≥1 to be a subset of N.

We say that a property (or a predicate) R(x) of N is *decidable* (or recursive, or solvable) iff there exists a Turing Machine M that always halts and for any input value x in N, M tells us whether or not R(x) holds by entering or not entering a final state.

This notion of decidability of a property is consistent with the notion of decidability of a problem. Indeed, the language corresponding to a problem is a subset of Σ^* and it can be viewed as a property which is enjoyed by some elements of Σ^*.

We say that a property P(x) of N is *semidecidable* (or semirecursive, or r.e., or semisolvable) iff there exists a Turing Machine M such that for any input value x in N for which P(x) holds, we have that M enters a final state and halts.

Semidecidability is a particular "degree of undecidability". There are, indeed, various degrees of undecidability, and they can be structured in the so-called Arithmetical Hierarchy (see Fig. 2).

We say that the subset A of N is a Σ_n set (or simply Σ_n) iff for some $n \geq 0$, there exists a recursive predicate $R(x,y_1,\ldots,y_n)$ on N^{n+1}, such that

$A = \{x \mid \exists y_1\ \forall y_2 \ldots R(x, y_1, y_2, \ldots, y_n)\}$. (All y_i's are alternatively quantified.)

We say that the subset A of N is a Π_n set (or simply Π_n) iff for some $n \geq 0$, there exists a recursive predicate $R(x,y_1,\ldots,y_n)$ on N^{n+1}, such that

$A = \{x \mid \forall y_1\ \exists y_2 \ldots R(x, y_1, y_2, \ldots, y_n)\}$. (All y_i's are alternatively quantified.)

In order to understand the definition of the classes Σ_n and Π_n, the reader should also recall that a contiguous sequence of universal quantifications is equivalent to one universal quantification only. The same holds for existential quantifications. Thus, in the definitions of the classes Σ_n and Π_n, it is important both (i) the nature of the outermost quantification, and (ii) the number of alternations in the quantifications.

As a consequence of the above definitions we have that an r.e. subset of N is an element of Σ_1. Thus, the following is an alternative definition of an r.e. set.

A set A is said to be r.e. iff there exists a decidable predicate R(x,y) such that $x \in A$ iff $\exists y.$ R(x,y). The predicate R can be taken to be primitive recursive [Cutland 80, page 124].

Given an r.e. set $\{x \mid P(x)\}$, the set $\{x \mid \mathrm{not}(P(x))\}$ is co-r.e. The set Π_1 is by definition the set of all co-r.e. subsets of N.

We have that $\Sigma_0 \cap \Pi_0$ is the set of the recursive subsets of N.

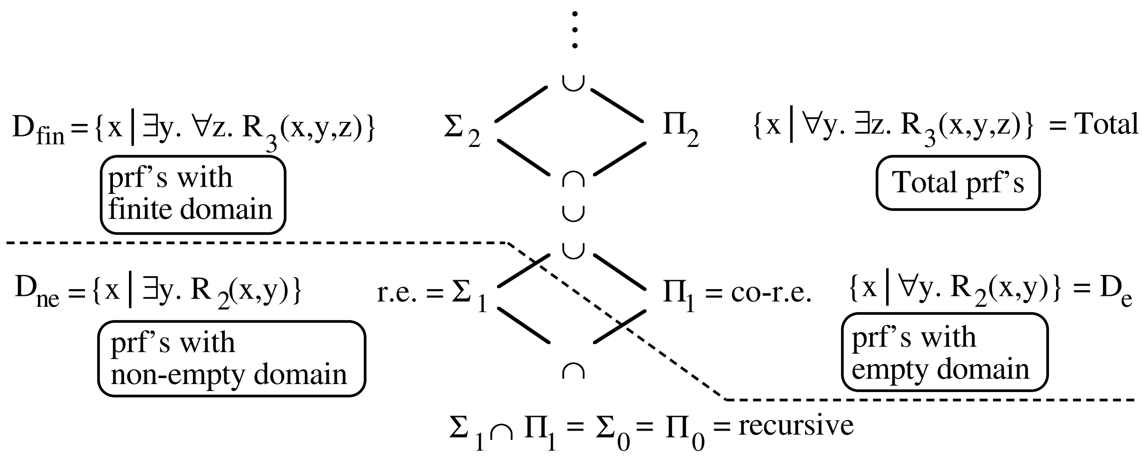


Figure 2. The Arithmetical Hierarchy. R_2 and R_3 are recursive predicates.
Turing Computability is below the dashed line. prf's stands for "partial recursive functions".
Containments are all strict.

For any given $n \geq 1$, we have that ([Rogers 67], page 305) (see also Fig. 2):

(i) $\{x \mid P(x)\}$ is in Σ_n iff $\{x \mid \text{not}(P(x))\}$ is in Π_n,
(ii) Σ_n, Π_n are both proper subsets of $\Sigma_n \cup \Pi_n$,
(iii) $\Sigma_n \cup \Pi_n$ is a proper subset of $\Sigma_{n+1} \cap \Pi_{n+1}$, and
(iv) $\Sigma_{n+1} \cap \Pi_{n+1}$ is a proper subset of both Σ_{n+1} and Π_{n+1}.

An example of a recursive predicate is the so-called Turing Predicate T(i,x,y,t). T(i,x,y,t) means that the Turing Machine whose encoding is the natural number i, given the input x, after exactly t steps, enters a final state and produces the output y.

Given a recursive property $R_2(x,y)$ we get a semirecursive property by considering its existential closure w.r.t. one of the two variables.

The set Dom_{ne} of partial recursive functions with non-empty domain, is in Σ_1, because

$$\text{Dom}_{ne} = \{i \mid \exists x\ \exists y\ \exists t.\ T(i,x,y,t)\}.$$

The set Dom_e of partial recursive functions with empty domain, is in Π_1, because

$$\text{Dom}_e = \{i \mid \forall x\ \forall y\ \forall t.\ \text{not}(T(i,x,y,t))\}.$$

The set Dom_{fin} of partial recursive functions with finite domain, is in Σ_2, because

$$\text{Dom}_{fin} = \{i \mid \exists max\ \forall x\ \forall y\ \forall t.\ \text{not}(T(i,x,y,t)) \text{ or } x<max\},$$

because $\exists max\ \forall x.$ if $(\exists y\ \exists t.\ T(i,x,y,t))$ then $x<max$ iff
$\exists max\ \forall x\ \forall y\ \forall t.\ (\text{not}(T(i,x,y,t)) \text{ or } x<max)$.

The set Total of partial recursive functions which are total, is in Π_2, because

$$\text{Total} = \{i \mid \forall x\ \exists y\ \exists t.\ T(i,x,y,t)\}.$$

The set R_{Bound} of total partial recursive functions with finite image, is in $\Sigma_3 \cap \Pi_3$, because

$$R_{Bound} = \{i \mid (\exists max\ \forall x\ \exists y\ \exists t.\ T(i,x,y,t) \text{ and } y<max) \text{ and } \forall x\ \exists y\ \exists t.\ T(i,x,y,t)\}.$$

As it is written, the predicate relative to the definition of R_{Bound}, indicates that R_{Bound} is in $\Sigma_3 \cap \Pi_2$, but since $\Pi_2 \subseteq \Pi_3$, it is also true that R_{Bound} is in $\Sigma_3 \cap \Pi_3$.

We have seen that the above sets Dom_e, Dom_{fin}, and Total are examples of Π_1, Σ_2, and Π_2 sets. They are, so to speak, significant examples, as we now illustrate.

Given two subsets, say A and B, of N, we say that A is 1-*reducible* to B, and we write $A \leq_1 B$, iff there exists an injective function f such that
$\forall a \in A.\ a \in A$ iff $f(a) \in B$.

Given a class *C* of subsets of N, and a reducibility relation $\leq$, we say that a set A is *hard* w.r.t. *C* and $\leq$ (or *C*-*hard* w.r.t. $\leq$, or $\leq$-*hard* in *C*) iff $\forall\ X \in C.\ X \leq A$.

Given a class C of subsets of N, and a reducibility relation $\leq$, we say that a set A is *complete* w.r.t. C and $\leq$ (or *C-complete* w.r.t. $\leq$, or *$\leq$-complete* in C) iff $\forall$ X $\in C$. X $\leq$ A and A $\in C$.

We have that ([Wagner-Wechsung 90], page 43):
(i) $K = \{i \mid \exists y\, \exists t.\ T(i,i,y,t)\}$ is Σ_1-complete w.r.t. $\leq_1$,
(ii) Dom_e is Π_1-complete w.r.t. $\leq_1$,
(iii) Dom_{fin} is Σ_2-complete w.r.t. $\leq_1$, and
(iv) Total is Π_2-complete w.r.t. $\leq_1$.

The notion of hardness and completeness w.r.t. a class C of subsets of N and the reducibility relation $\leq_1$, can also be defined w.r.t. other classes of sets and other (reflexive and transitive) reducibility relations.

For instance, in the literature one finds the notions of NP-hard problems or NP-complete problems. They are given w.r.t. the class of NP problems and the polynomial-time (or logarithmic-space) reductions between the languages corresponding to those problems [Hopcroft-Ullman 79].

Fig.3 below shows that in different classes of sets, like PRF (indexes, or encodings of partial recursive functions from N to N), CF (context-free languages over $\Sigma=\{0,1\}^*$), and REG (regular languages over $\Sigma=\{0,1\}^*$), the 'relative difficulty' of some problems is preserved. For instance,

(i) in PRF the Membership Problem is in Σ_1, while the 'more difficult' Totality Problem is in Π_2,
(ii) in CF the Membership Problem is in Σ_0 (that is, it is decidable), while the 'more difficult' problem of determining whether or not a given language L is equal to Σ^*, is in Π_1, and
(iii) in REG the Membership Problem can be solved in polynomial time, while the 'more difficult' problem of determining whether or not a given language L is equal to Σ^*, is polynomial-space complete w.r.t. logarithmic-time reductions ([Hopcroft-Ullman 79], page 370).

$f \in$ PRF		$L \in$ CF		$L \in$ REG	
	f is total : Π_2		$L = \Sigma^*$: Π_1		$L = \Sigma^*$: PSC
$x \in Dom(f)$: Σ_1	f is always undefined : Π_1	$w \in L$: Σ_0	$L = \varnothing$: Σ_0	$w \in L$: PT	$L = \varnothing$: PT

PSC : polynomial-space complete (w.r.t. polynomial-time reductions)
PT : polynomial time

Figure 3. Comparing problem difficulties in various classes of sets. The time and space complexities PT and PSC are w.r.t. the size of the regular expression defining L.
Recall that PSC $\subseteq$ DTIME($2^{p(n)}$) where p(n) is a polynomial in n.

REFERENCES

[Cutland 80] Cutland, N. J.: “Computability: An Introduction to Recursive Function Theory”, Cambridge University Press, 1980.

[Davis 58] Davis, M.: “Computability and Unsolvability” McGraw-Hill, New York, 1958.

[Harrison 78] Harrison, M.A.: “Introduction to Formal Language Theory” Addison-Wesley, 1978.

[Hermes 69] Hermes, H.: “Enumerability, Decidability, Computability”, Springer Verlag, 1969.

[Hopcroft-Ullman 79] Hopcroft, J. E., and Ullman, J. D.: “Introduction to Automata Theory, Languages, and Computation” Addison-Wesley, 1979.

[Mendelson 87] Mendelson, E.: “Introduction to Mathematical Logic” 3rd Edition, Wadsworth, Monterey, California, 1987.

[Rogers 67] Rogers, H. Jr.: “Theory of Recursive Functions and Effective Computability”, McGraw Hill, 1967.

[Wagner-Wechsung 90] Wagner, K., and Wechsung, G.: “Computational Complexity”, D. Reidel Publishing Co. Dordrecht, 1990.