9. UNDECIDABILITY

CHAPTER - 9
UNDECIDABILITY

9.1 INTRODUCTION

A problem whose language is recursive is said to be *decidable*. Otherwise, the problem is *undecidable*. That is, a problem is undecidable if there is no algorithm that takes as input an instance of the problem and determines whether the answer to that instance is *yes* or *no*.

Example 9.1

(i) Decidable problem

Given two DFSM's M_1 and M_2 is $L(M_1) \subseteq L(M_2)$?

That is an algorithm is exist for the problem to decide the answer as yes or no.

(ii) Undecidable problem

a) *Halting Problem* of TM. That is, given any TM M and any input string *w*, does M halt on *w*?

We want to write a program which will correctly output an answer to the question about termination (halting) of any given program.

This is called halting problem and since we can't develop an algorithm to solve this problem, this problem is an undecidable problem.

b) Can you develop an algorithm which will correctly tell us when a person will die? whatever you want may be taken as input.

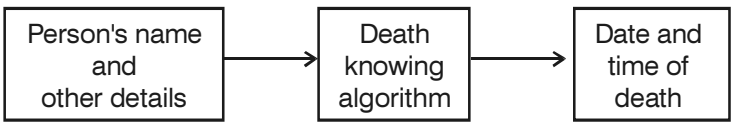


Figure 9.1 Undecidability of Death

9.2 DECISION PROBLEMS

Decision problems can be converted into membership problems for appropriate languages.

Example 9.2

Does a given DFSM M accept a given string *w*?

This problem is the same as testing whether the encoding of M and the string *w* belong to the language L defined as:

{"M" "*w*" : M is a DFSM that accepts string *w*}.

Problems, algorithms and languages

Decision problem	≡ Language
Algorithm	≡ TM that halts on all inputs
An algorithm for a problem	≡ A TM that decides corresponding language
Problem is decidable	≡ Language is recursive

Decidable problems for regular languages

The following problems are decidable

(i) Given a DFSM M and a string *w*, does M accept *w*?

(ii) Given a DFSM M, does M accept λ?

(iii) Given a DFSM M, is $L(M)=\phi$?

(iv) Given a DFSM M, is $L(M) = \Sigma^*$?

(v) Given two DFSMs M_1 and M_2, is $L(M_1)=L(M_2)$?

(vi) Given an NFSM M and a string *w*, does M accept *w*?

(vii) Given a regular expression α and a string *w*, does α generate *w*?

Theorem : The language L defined as {"M", "*w*" : M is a DFSM that accepts *w*} is recursive.

Proof

The TM M_D decides L: On input "M" "*w*", where M is a DFSM,

(i) M_D emulates M on input *w*.

(ii) If the emulation ends in a final state of M, then M_D halts in state *y*. If it ends in a non-final state of M, then M_D halts in state *n*.

One reasonable way to encode M is to adapt the TM encoding for FSMs and encode Q, Σ, δ, S and F. Before starting the emulation, M_D first checks whether the input string is a legal encoding of a DFSM and the DFSM's input string.

Theorem : The language L defined as {"M" : M is a DFSM and L(M) = ϕ} is recursive.

Proof

The following TM M_ϕ decides L: On input "M", where M is a DFSM.

(i) M_ϕ marks the start state of M.

(ii) Repeat until no new states are marked:

If $(p, \sigma, q) \in \delta$ and p is already marked, then mark q.

(iii) If no final state is marked, M_ϕ halts in y; otherwise, it halts in n.

Theorem: The language L defined as {"M_1" $\cup$ "M_2" : DFSMs M_1, M_2 and $L(M_1) \subseteq L(M_2)$} is recursive

Proof

The following TM M_E decides L: On input "M_1" and "M_2", where M_1 and M_2 are DFSMs.

(i) Construct DFSM M for the language $(L(M_1) \cap L(M_2))$.

(Note that $L(M) = \phi$ if and only if $L(M_1) \subseteq L(M_2)$)

(ii) Apply M_ϕ to the input string "M".

(iii) If M_ϕ halts in y, M_E halts in y.

If M_ϕ halts in n, ME halts in n.

9.3 RECURSIVE(R) AND RECURSIVELY ENUMERABLE(RE) LANGUAGES

The *R* and *RE* languages are called as *Turing decidable* and *Turing acceptable languages* respectively. In this topic, we will discuss about the languages associated with TM's and some of their restrictions. The non-trivial question is *"whether there are any languages that are not accepted by some TM."* To answer this question: *Show that; there are many languages than TMs, therefore for some languages for which there are TMs.*

9.3.1 Recursive Languages

Definition : A language L is Recursive (R) if and only if there is a TM that decides L.

Let M = (Q, Σ, δ, *S*, H) be a TM such that :

- H = {*y*, *n*}, where *y* means yes and *n* means no.
- $\Sigma_0 \subseteq \Sigma - \{g, \triangleright\}$ is the input alphabet of M
- $L \subseteq \Sigma_0^*$ is a language
- Assume that the initial configuration of the TM is (S, w), where w is an input string that contains no blanks and the underline indicates that the reader-writer is over the first character of w.

- M decides L if, for all strings $w \in \sum_0^*$:
  - either $w \in$ L, in which case M accepts w (M halts in state y)
  - or $w \notin$ L, then M rejects w (M halts in state n)

Example 9.3

L = $\{a^n b^n c^n : n>0\}$ a noncontext-free language.

An informal description of the idea behind a TM program:

(i) Starting from the left end of the input string, in one pass find one *a*, followed later by one *b*, followed later by one *c*. Then, repeat this process until no new *a*'s, *b*'s or *c*'s are found.

(ii) Move to the right in search of one *a*; if an *a* is found, rewrite it as *d*; if a *b* or a *c* is found, then reject.

(iii) The TM halts in state *y* if all *a*'s, *b*'s and *c*'s are replaced by *d*'s. Otherwise it halts in state *n*.

(iv) Move further right in search of one *b*; if a *b* is found, rewrite it as a *d*; if no *b* is found, then reject.

(v) Move further right in search of one *c*; if a *c* is found, rewrite it as a *d*; if no *c* is found, then reject.

(vi) If all input has been replaced by *d*'s, then accept.

(vii) Return to the left end of the input string, goto 2.

Since this Turing Machine decides L, L is recursive.

Example 9.4

L = $\{a^i b^j c^k : i \times j = k \text{ and } i, j, k > 1\}$.

An informal description of the idea behind a TM program:

(i) Preprocess the input string from left to right to ensure that it is a string of the form *a*b*c**. If it is not of this form, then reject.

(ii) Return to the left end of the input string.

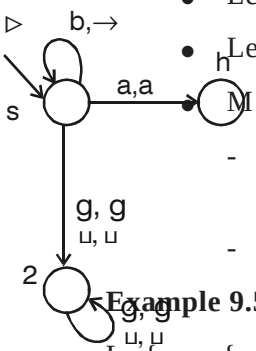
(iii) Check an *a* and scan to the right until it meets a *b*. Move back and forth between the *b*'s and the *c*'s, each time checking off one *b* and one *c* until all *b*'s are checked.

(iv) Uncheck the checked *b*'s and repeat 3 if there is another *a* to check. If all *a*'s are check, determine whether all *c*'s are also checked. If they are, then accept; otherwise, reject.

9.3.2 Recursively Enumerable Languages

Definition : L is Recursively Enumerable (RE) if and only if there is a TM that semidecides L.

Let M = (Q, Σ, δ, *S*, H) be a Turing Machine



- Let $\Sigma_0 \subseteq \Sigma - \{g, \triangleright\}$ be an alphabet
- Let $L \subseteq \Sigma_0^*$ be a language
- M semidecides L if, for all strings $w \in \Sigma_0^*$:
  - either $w \in L$, in which case M halts on input *w*. (Starting the TM with configuration (*S*, *w*) causes the machine to halt)
  - or $w \notin L$, in which case M does not halt (of course, it may stop or loop)

Example 9.5

L: $\{w \in \{a,b\}^* : w$ contains at least one $a\}$. The first TM semidecides L. (If *w* does not contain an *a*, then M does not stop) Thus, L is RE.

The second TM decides L.

Constructing a language that is RE but is not recursive is much more difficult.

- TMs that semidecide languages are NOT algorithms.
- TMs that decide languages are algorithms.

9.4 PROPERTIES OF RECURSIVE LANGUAGES

Theorem 1

The *union* of two *recursive languages* is *recursive*. The *union* of two *recursively enumerable* languages is *RE*.

Proof

I. Recursive language

(i) Let L_1 & L_2 be recursive languages accepted by M_1 and M_2.

(ii) We construct M, which first simulates M_1. If M_1 accepts, then M accepts.

(iii) If M_1 rejects then M simulates M_2 and accepts if and only if M_2 accepts.

(iv) *Conclusion :* Since M_1 and M_2 are algorithms, M is guaranteed to halt. Clearly M accepts $L_1 \cup L_2$.

Figure 9.2 Construction of union - Recursive languages

II. Recursively enumerable languages

For recursively enumerable languages the above construction does not work, since M_1 may not halt. Instead M can simultaneously simulate M_1 and M_2 on separate tapes. If either accepts, then M accepts. The following figure shows the construction of this theorem.

Figure 9.3 Construction of union - Recursively enumerable languages

Theorem 2

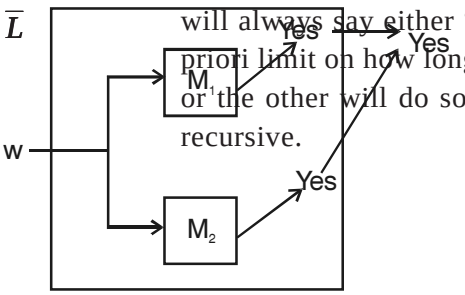
If a *language L* and its *complement* $\overline{L}$ are both *recursively enumerable*, then L (hence) is *recursive*.

Proof

(i) Let M_1 and M_2 accept L and respectively.

(ii) Construct M to simulate simultaneously M_1 and M_2. M accepts *w* if M_1 accepts *w* and rejects *w* if M_2 accepts *w*.

(iii) Since *w* is in either L or , we know that exactly one of M_1 or M_2 will accept. Thus M will always say either "yes" or "no," but will never say both. Note that there is no a priori limit on how long it may take before M_1 or M_2 accepts, but it is certain that one or the other will do so. Since M is an algorithm that accepts L, it follows that L is recursive.



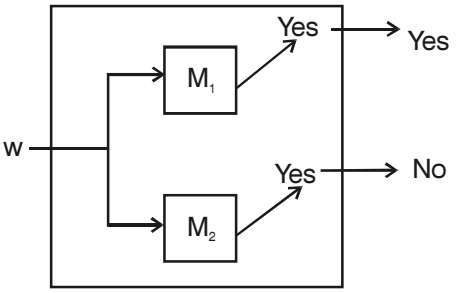


Figure 9.4 Construction for Theorem 2

Theorem 3

If *L* is *recursive*, then is *recursive* (recursive languages are closed under complementation)

Proof

Let M = (Q, ∑, δ, *S*, {*y*, *n*}) decide L

Define $\overline{M}$ = (Q, Σ, δ^1, *S*, {*y*, *n*}) such that is similar to M except that it swaps that halt states *y* and *n*. That is;

(i) If (*p*, *a*, *b*, *n*) $\in \delta$, then (*p*, *a*, *b*, *y*) $\in$.

(ii) If (*p*, *a*, *b*, *y*) $\in \delta$, then (*p*, *a*, *b*, *n*) $\in$.

(iii) All other transitions of the form (*p*, *a*, *b*, *q*) in δ, where $n \neq q \neq y$, are in .

Figure 9.5 Construction for Theorem 3

Theorem 4

If *L* is *recursive*, then *L* is *RE*.

Proof

Idea : Since L is recursive, there is a TM M that decides it. Make the rejected state of M a nonhalting state.

Let M = (Q, Σ, δ, *S*, {*y*, *n*}) decide L

Define M^1 = (Q, Σ, δ, *S*, {*y*})

- Either $w \in$ L, in which case M halts in state *y*. Thus, M^1 also halts in state *y*
- Or $w \notin$ L, in which case M halts in state *n*. Thus, M^1 stops in state *n*

Question:

Is the complement of an RE language always RE?

Solution :

No.

Question :

If L is RE, then is L always recursive?

Solution :

No.

9.5 THE DIAGONALIZATION LANGUAGE: A NON-RE LANGUAGE

Using previous encoding, we encoded the "*i*-th Turing machine" M_i as integer *i*, with code w_i, the i-th binary string. For invalid codes a TM halts immediately on any input, i.e., $L(M_i)=\phi$.

The diagonalization language, L_d, is the set of strings w_i such that $w_i \in L(M_i)$. That is, L_d consists of all strings *w* such that the TM M whose code is *w* does not accept when given *w* as input.

The diagonalization table construction is as follows:

(i). rows *i*-th TM M_i in the form of its binary encoding *i*,

(ii). columns binary encoding *j* of strings w_j,

(iii). table contents, so-called the characteristic vector for the language $L(M_i)$; that is, 1 means that the word is accepted by M_i, and 0 means that it doesn't.

$M_i = i \downarrow$	$w_j = j \rightarrow$ 1	2	3	4	…
1	0	1	1	0	…
2	1	1	0	0	…
3	0	0	1	1	…
4	0	1	0	1	…
.	.	.	.	.	. diagonal

The diagonal values tell whether M_i accepts w_j. To construct L_d, we complement the diagonal. For example, for the table, the complemental diagonal would begin with string 1000…. Thus, L_d would contain $w_1=\varepsilon$, not contain w_2 through w_4, which are 0, 1, and 00, and so on.

The complemental diagonal represents the characteristic vector of some language, namely L_d, but not characteristic vector of any $L(M_i)$ (it differs at least on 1 position with any possible row in the table).

Thus, the complement of the diagonal cannot be the characteristic vector of any Turing machine, i.e., $w_i \notin L(M_i)$, thus it belongs to L_d according to its definition.

Theorem

The diagonalization language L_d, is not a recursively enumerable language. That is, there is not Turing machine that accepts L_d.

Proof

Suppose L_d is RE. Then L_d=H(M) for some TM M.

Since the input alphabet of M is Σ={0,1}, i.e. M would be on the list of TM's we have constructed, since it includes all TM's with input alphabet {0,1}. Thus M is M_i for at least one value of *i*. Let w_i be the *i*-th string, i.e., w_i is *i* in binary (and on diagonal). Question: is w_i in L_d?

(i) If w_i is in L_d, then M_i accepts w_i. But then, by definition of L_d, w_i is not in L_d, because L_d contains only those w_j (has value 1 in the table for L_d's row) such that M_j does not accept w_j (has value 0 in the table - negated diagonal 1).

(ii) If w_i is not in L_d, then M_i does not accept w_i. Thus by definition of L_d, w_i is in L_d.

Since w_i cannot be at the same time in L_d and not to be in L_d, we conclude that there is a contradiction of our assumption that M exist. That is L_d is not RE.

9.6 UNIVERSAL TURING MACHINE (U) AND UNIVERSAL LANGUAGE (L_U)

Each Turing machine appears to be specialized at solving one particular problem.

We know that we can 'program' computers to solve many different problems-they are *general-purpose programmable machines*

Can we also 'program' a Turing machine?

Yes!

We will show that there are TM's, universal TM's, that can be programmed to solve any problem that can be solved by any Turing machine.

9.6.1 Universal Turing Machine U:

Input to U: The encoding "M" of a TM M and the encoding "*w*" of a string $w \in \Sigma^*$.

Behavior : U halts on input "M" "*w*" if and only if M halts on input *w* (U emulates M with input *w*). How do we encode a Turing machine as a string?

Crucial assumptions: We require that there are no transitions from any of the halt states of any given TM. Apart from the halt state, a given TM is total.

Encoding of TM's and their input strings

Let M = (Q, {0, 1}, {0, 1, B}, δ, q_1, B,{q_n})

Where Σ = {0, 1} - input alphabet

Γ = {0, 1, B} - tape symbol

Q = {q_1, q_2..... q_n} - set of states

q_1 = start state

q_n = final state

Generic Move :

1. We call symbols 0, 1, B by X_1, X_2, X_3 respectively.

   (i.e.,) $0 = X_1$

   $1 = X_2$

   $B = X_3$

2. The directions L and R are called by synonyms D_1, D_2 respectively.

   (i.e.,) $L = D_1$

   $R = D_2$

3. The generic move.

   $\delta(q_i, X_j) = (q_k, X_l, D_m)$ is encoded by the binary string has

   $0^i\ 10^j\ 10^k\ 10^l\ 10^m$ ..... (1)

   The binary code of Turing Machine M is :

   111 $code_1$ 11 $code_2$11.... 11 $code_r$ 111 .....(2)

   where each $code_i$ is of the form (1) .

4. Any such code of M is denoted by <M>

Note :

(i) Each decoding is unique, (because no string of the form (1) has two 1's in a row, so code is found directly.

(ii) If a string fails to begin and end with exactly three 1's, the string represents no TM.

(iii) The pair M and *w* is represented by a string of the form (2) followed by *w*. It is denoted as <M, *w*>

Example 9.6

Obtain the code for <M, 1011> where M = ($\{q_1, q_2, q_3\}$, {0,1}, {0, 1, B}, δ, q, B, $\{q_2\}$)

have moves

$\delta(q_1, 1) = (q_3, 0, R)$

$\delta(q_3, 0) = (q_1, 1, R)$

$\delta(q_3, 1) = (q_2, 0, R)$

$\delta(q_3, B) = (q_3, 1, L)$ **(Nov/Dec 2003)**

Solution :

1. We call tape symbols 0, 1, B by synonyms X_1, X_2, X_3 respectively.

   i.e., $0 = X_1$

   $1 = X_2$

   $B = X_3$

2. We give directions L and R by synonyms D_1, D_2 respectively.

   $L = D_1$

   $R = D_2$

3. Generic move $\delta(q_i, X_j) = (q_k, X_r, D_m)$ is encoded by binary string

   $0^i \quad 10^j \quad 10^k \quad 10^l \quad 10^m$

   Separator

   $\therefore$ for the given moves the binary coding is as follows:

   $\delta(q_1, 1) = (q_3, 0, R)$ it is of the form

   $\delta(q_1, X_2) = (q_3, X_1, D_2)$ (because of synonyms)

   $\therefore$ binary code is $0^1 10^2 10^3 10^1 10^2$

   Similarly for $\delta(q_3, 0) = (q_1, 1, R)$ is of form

   $\delta(q_3, X_1) = (q_1, X_2, D_2)$

   binary code = $0^3 10^1 10^1 10^2 10^2$

   Similarly for $\delta(q_3, 1) = (q_2, 0, R)$ is of form

   $\delta(q_3, X_2) = (q_2, X_1, D_2)$

   binary code is $0^3 10^2 10^2 10^1 10^2$

   Similarly for $\delta(q_3, B) = (q_3, 1, L)$ is of form

   $\delta(q_3, X_3) = (q_3, X_2, D_1)$

   binary code is $0^3 10^3 10^3 10^2 10^1$

   $\therefore$ binary code of TM is

   111 $code_1$ 11 $code_2$ 11 $code_3$ 11 $code_4$ 111

   $\therefore$ <M, *w*> is represented as

   111 $code_1$ 11 $code_2$ 11 $code_3$ 11 $code_4$ 111*w*

$\therefore$ <M, 1011> is represented as

111 0100100010100 11 000101010010011

00010010010100 11 000100010001001011111011

Construction of universal turing machine

It is easiest to describe the Universal Turing Machine U as a multitape TM with tapes and operation (moves) described below:

(i) Tape 1-keeps the transitions δ of M encoded in binary, that is tape symbol X_i of M is represented by 0^i, and tape symbols will be separated by single 1's along with the string *w*.

(ii) Tape 2 - keeps a binary string *w* (input of M) where M's 0's are encoded as 10, 1's as 100.

(iii) Tape 3 - holds the state of M, with state q_i represented by *i* 0's.

Other tapes are scratch tapes. The operation of U can be summarized as follows:

(i) Examine the Tape 1 to make sure that the code for M is a legitimate code for some TM. If not, U halts without accepting. Since invalid codes are assumed to represent the TM with no moves, and such a TM accepts no inputs, this action is correct.

(ii) Initialize Tape 2 for the input string w where the blanks of M (encoded as 1000) could be replaced by U's own blank symbol.

(iii) Place 0, as the start state of M, on Tape 3, and move the head of U's second tape to the first simulated cell.

(iv) To simulate a move of M, U searches on its first tape for a transition $0^i10^j10^k10^l10^m$, such that 0^i is the state on tape 3, and 0^j is the tape symbol of M that begins at the position on tape 2 scanned by U. This transition $0^i10^j10^k10^l10^m$ is the one M would next make. U should:

(a) Change the contents of tape 3 to 0^k (next state). To do so, U first erases all the 0's on tape 3 to blanks, and the copies 0^k from tape 1 to tape 3.

(b) Replace 0^j on tape 2 by 0^l; that is, change the tape symbol of M. If more or less space is needed (i.e., i^1j), use the scratch tape and shift contents appropriately to manage the spacing.

(c) Move the head on tape 2 to the position of the next 1 to the left or right, respectively, depending on whether m=1 (move left) or m=2 (move right). Thus, U simulates the move of M to the left or to the right.

(v) If M has no transition that matches the simulated state and tape symbol, then in step 4, no transition will be found. Thus, M halts in the simulated configuration, and U must do likewise.

(vi) If M enters its accepting state, then U accepts.

In this manner, U simulates M on w.

Theorem

U accepts the coded pair <M,*w*> if and only if M accepts *w*, i.e., L_u=L(U) is RE.

Tape 1: encoded simulated TM M

Tape 2: encoded its input *w*

Tape 3: encoded its state *q*

Tape 4: and others: scratch tapes

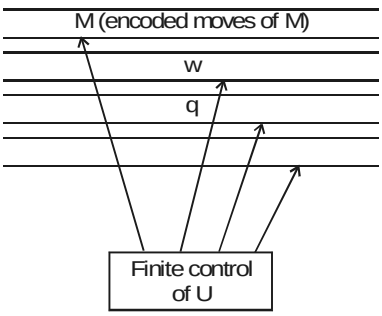


Figure 9.6 Organization of the Universal Turing Machine U as a Multitape TM

9.6.2 The Universal Language (Lu)

The *Universal Language* L_u (of the UTM U) consists of the set of binary strings in the form of pairs <M,*w*>, where M is a TM encoded in binary and *w* is its binary input string, such that $w \in L(M)$, i.e., L_u=L(U), where $<M,w> \in L(U)$.

Since the input to U is a binary string, U is in fact some M_j in the list of binary-input Turing machines from the table developed in the construction of the diagonalization language.

Universal Turing Machine describes the notion of the universal (or Turing universal) computability.

9.7 UNDECIDABLE PROBLEMS

9.7.1 Reduction Technique

Reduction: If we have an algorithm to convert instances of a problem P_1 to instances of a problem P_2 that have the same answer, then we say that P_1 reduces to P_2 (i.e., P_1 becomes a special case/subset of P_2)

Reduction of L_u to P: Knowing that the Universal Language L_u is undecidable (i.e., not a recursive language) is useful by applying the reduction of L_u to another problem P to prove that there is no algorithm to solve P, regardless of whether or not P is RE.

Reduction of L_d to P: This is only possible if P is not RE, so L_d cannot be used to show undecidability for those problems that are RE but not recursive. On the other hand, if we want to show a problem not to be RE, then only L_d can be used; L_u is useless since it is RE.

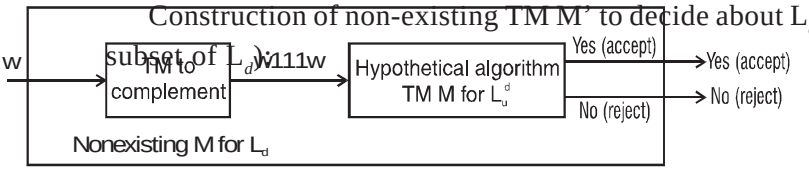
Theorem

L_u is RE but not recursive, i.e., undecidable.

Proof

By simulation of the UTM by a multitape TM we showed that L_u is RE. Suppose that L_u were recursive. Then by definition, its complement L_u^d would also be recursive. We prove by reduction from L_d to L_u^d that L_u^d is not recursive.

Construction of non-existing TM M' to decide about L_d: reduction of L_d to L_u^d (it becomes subset of L_u):



(i) Transfer *w* (input of M' and of UTM) to complementary value (complementary UTM), e.g., *w*111*w* belongs to complement of arbitrary w. Put *w*111*w* as an input of TM M (complementary UTM).

(ii) M' simulates M on the new input. If *w* is w_i in enumeration for L_d, then M' determines whether M_i accepts w_i. Since M accepts L_u^d :, i.e., complement of $w=w_i$, it will accept iff M_i of UTM does not accept w_i, i.e., w_i is in L_d (by definition of L_d).

Thus, M' accepts *w* if and only if *w* is in L_d. Since we know M' cannot exist, we conclude that L_u is not recursive.

9.7.2 Rice's Theorem

Any nontrivial property about the language recognized by a Turing machine is undecidable.

A property about Turing machines can be represented as the language of all Turing Machines, encoded as strings, that satisfy that property. The property P is *about the language recognized by Turing machines* if whenever L(M)=L(N) then P contains (the encoding of) M iff it contains (the encoding of) N. The property is non-trivial if there is at least one Turing Machine that has the property, and at least one that hasn't.

Proof

Without limitation of generality we may assume that a Turing machine that recognizes the empty language does not have the property P. For if it does, just take the complement of P. The undecidability of that complement would immediately imply the undecidability of P.

In order to arrive at a contradiction, suppose P is decidable, i.e., there is a halting Turning machine B that recognizes the descriptions of Turing machines that satisfy P. Using B we can construct a Turning machine A that accepts the language {(M, w)} M is the description of a Turing machine that accepts the string *w*}. As the latter problem is undecidable this will show that B cannot exists and P must be undecidable as well.

Let MP be a Turing machine that satisfies P (as P is non-trivial there must be one). Now A operates as follows:

(i) On input (M, *w*), create a (description of *a*) Turing machine C(M, *w*) as follows:

(a) On input *x*, let the Turing machine M run on the string w until it accepts (so if it doesn't accept C(M,*w*) will run forever).

(b) Next run MP on *x*. Accept iff MP does.

Note that C(M, *w*) accepts the same language as MP if M accepts *w*; C(M, *w*) accepts the empty language if M does not accept *w*.

Thus if M accepts w the Turing machine C(M, *w*) has the property P, and otherwise it doesn't.

(ii) Feed the description of C(M, w) to B. If B accepts, accept the input (M,w); if B rejects, reject.

9.7.3 Halting Problem

The halting problem for TMs is : Given any TM M and any input string *w*, does M halt on *w*?

This problem is undecidable.

An equivalent problem : Given the language H = {"M" "*w*" : TM M halts on string *w*}, is H recursive? As a first step, is H recursively enumerable?

Yes, the UTM U semidecides it. (since U halts on input "M" "*w*" whenever M halts on *w*).

Theorem

The halting problem is undecidable; that is

H={"M" "*w*" : TM M halts on string *w*} is not recursive.

Proof Idea

- Assume that H is recursive.
- By the definition of recursive languages, there is a TM M_H that decides H
- Design a new TM that uses M_H as a submachine to decide H_1
- But this construction contradicts that nonrecursiveness of H_1! Hence, H is not recursive
- We say H_1 is *reduced* to H

Proof

- Assume that H is recursive
- Then, there is a TM (or algorithm) M_H that decides H; that is either M_H ("M" "*w*") halts in state *y*, if M halts on *w* or M_H ("M" "*w*") halts in state *n*, if M does not halt on *w*.
- We construct the following TM M_{H_1} that decides H_1. M_{H_1} has the following behavior.

  (i) If an input string *x* is not the encoding of a TM, then it halts in state *n*.

  (ii) If an input string *x* is the encoding of a TM M (*x*= "M"), then apply M_H to the input string *x* "*x*" (M_H(*x* "*x*"))

  If M_H halts in state *y*, then M_{H_1} halts in state *y*.

  If M_H halts in state *n*, then M_{H_1} halts in state *n*
- This behavior contradicts the nonexistence of M_{H_1}
- Hence, H is not recursive

Correctness Verification

Verify the correctness of M_{H_1}.

On input *x*, M_{H_1} behaves as follows:

- $x \in H_1$. Then, by the definition of H_1, *x* = "M", for some TM M, and M halts on input "M". Thus, M_H(*x* "*x*") halts in state *y*. By the construction of M_{H_1}, $\overline{M}_{H_1}$ also halts in state *y*

- $x \notin H_1$.
  - x is not an encoding of any TM. Then, M_{H_1} halts in state n
  - x = "M", for some TM M, and M does not halt on "M". Then, $M_H(x$ "x") halts in state n.

By the construction of M_{H_1}, M_{H_1} also halts in state n

Alternative approach to halting problem proof

Here is an equivalent way to present the proof:

- Assume that problem H is decidable
- Then, there is an algorithm AlgoH(M, w) that decides it. That is, AlgoH(M, w) halts in state y if M halts on w and halts in state **no** if M does not halt on w.
- We can now construct an algorithm for problem H_1 (which is known to be unsolvable) as follows:

$AlgoH_1(x)$

(i) If x is not an encoding of any TM, then $AlgoH_1$ halts in state n.

(ii) If x is an encoding of some TM, call AlgoH(M, "M")

(iii) If AlgoH halts in state y, then $AlgoH_1$ halts in state y.

(iv) If AlgoH halts in state n, then $AlgoH_1$ halts in state n.

9.8 RELATION BETWEEN RECURSIVE, RE AND NON-RE LANGUAGES

From the 9 possible ways to place a language L and its complement L_d in the above diagram only 4 are possible:

(i). For recursive languages: both L and L_d are recursive, i.e., in the inner ring. Both L and L_d are decidable.

(ii). For RE languages: L is RE but not recursive, and L_d is not RE, i.e., one is in the middle ring (L) and the other is in the outer ring (L_d). L is decidable and L_d is undecidable.

(iii). For RE languages: L_d is RE but not recursive, and L is not RE, i.e., one is in the middle ring (L_d) and the other is in the outer ring (L). L_d is decidable and L is undecidable.

(iv). For not RE languages: both L and L_d are in the outer ring. Both L and L_d are undecidable.

9.9 POST'S CORRESPONDENCE PROBLEM

An instance of Post's Correspondence Problem (PCP) consists of two lists $A = w_1, w_2, ..., w_k$ and $B = x_1, x_2, ... x_k$ of strings over some alphabets $\sum$. This instance of PCP has a solution if there is any sequence of integers $i_1, i_2, ... i_m$, with $m \geq 1$, such that

$w_{i1}, w_{i2}, ..., w_{im} = x_{i1}, x_{i2}, ... x_{im}$.

The sequence $i_1, i_2, ... i_m$ is a solution to this instance of PCP.

Example 9.7

Let $\sum = \{0, 1\}$. Let A and B be lists of three strings each, defined below in the Fig.PCP has a solution 2,1,1,3.

	List A	*List B*
i	w_i	x_i
1	1	111
2	10111	10
3	10	0

Fig. An instance of PCP

Let $m = 4$, $i_1 = 2$, $i_2 = 1$, $i_3 = 1$, $i_4 = 3$. Now

$w_2w_1w_1w_3 = 101111110$

$x_2x_1x_1x_3 = 101111110$

Represent the string w_i on the top half and the string x_i on the bottom half as:

10111	1	1	0
10	111	111	0

It is easy to verify the solution sequence beginning

10111	...
10	.....

9.9.1 Modified Post's Correspondence Problem

Definition : In the modified PCP, there is an additional requirement on a solution that the first pair on the A and B lists must be the first pair in the solution. More formally, an instance of MPCP is two lists A = $w_1, w_2, ...w_k$ and B = $x_1, x_2, ... z_k$, and a solution is a list of 0 or more integers $i_1, i_2, ... i_m$ such that

$$w_1 w_{i_1} w_{i_2} ..... w_{i_m} = x_1 x_{i_1} x_{i_2} .....x_{im}$$

Proof of PCP undecidability

In the reduction of L_u to MPCP, given a pair (M, *w*), we construct an instance (A, B) of MPCP such that TM M accepts input *w* if and only if (A, B) has a solution.

The essential idea is that MPCP instance (A, B) simulates, in its partial solutions, the computation of M on input *w*. That is, partial solutions will consist of strings that are prefixes of the sequence of ID's of M: $\#\alpha_1 \#\alpha_2 \#\alpha_3 \#.....$, where α_1 is the initial ID of M with input *w*, and $\alpha_i \vdash \alpha_{i+1}$ for all *i*. The string from the B list will always be one ID ahead of the string from the A list, unless M enters an accepting state. In that case, there will be pairs to use that will allow the A list to "catch up" to the B list and eventually produce a solution. However, without entering an accepting state, there is no way that these pairs can be used, and no solution exists.

To simplify the construction of an MPCP instance, we assume that TM never prints a blank and never moves left from its initial head position. In that case, an ID of the Turing machine will always be a string of the form $\alpha q \beta$, where α and β are strings of nonblank tape symbols and *q* is a state. However, we shall allow β to be empty if the head is at the blank immediately to the right of α, rather than placing a blank to the right of the state. Thus, the symbols of α and β will correspond exactly to the contents of the cells that held the input, plus any cells to the right that the head has previously visited.

Let M = {Q, Σ, Γ, δ, q_0, B, F) be a TM and let *w* in Σ^* be an input string. We construct an instance of MPCP as follows. To understand the motivation behind our choice of pairs, remember that the goal is for the first list to be one ID behind the second list, unless M accepts.

(i) The first pair is:

List A	List B
#	$\#q_0w\#$

This, pair which must start any solution according to the rules of MPCP, begins the simulation of M on input *w*. Notice that initially, the B list is a complete ID ahead of the A list.

(ii) Tape symbols and the separator # can be appended to both lists. The pairs

List A	List B	
X	X	for each X in Γ
#	#	

allow symbols not involving the state to be "copied". In effect, choice of these pairs lets us extend the A string to match the B string, and at the same time copy parts of the previous ID to be end of the B string. So doing helps to form the next ID in the sequence of moves of M, at the end of the B string.

(iii) To simulate a move of M, we have certain pairs that reflect those moves. For all *q* in Q–F (i.e. *q* is a nonaccepting state), *p* in Q, and X, Y and Z in Γ we have:

List A	List B	
*q*X	Y*p*	if $\delta(q, X) = (p, Y, R)$
Z*q*X	*p*ZY	if $\delta(q, X) = (p, Y, L)$; Z is any tape symbol
q#	Y*p*#	if $\delta(q, B) = (p, Y, R)$
Z*q*#	*p*ZY#	if $\delta(q, B) = (p, Y, L)$; Z is any tape symbol

Like the pairs of (2), these pairs help extent the B string to add the next ID, by extending the A string to match the B string. However, these pairs use the state to determine the change in the current ID that is needed to produce the next ID. These changes – a new state, tape symbol, and head move – are reflected in the ID being constructed at the end of the B string.

(iv) If the ID at the end of the B string has an accepting state, then we need to allow the partial solution to become a complete solution. We do so by extending with "ID's" that are not really ID's of M, but represent what would happen if the accepting state were-allowed to consume all the tape symbols to either side of it. Thus, if *q* is an accepting state, then for all tape symbols X and Y, there are pairs:

List A	List B
X*q*Y	*q*
X*q*	*q*
*q*Y	*q*

(v) Finally, once the accepting state has consumed all tape symbols, it stands alone as the last ID on the B string. That is, the remainder of the two strings (the suffix of the B string that must be appended to the A string to match the B string) is $q\#$. We use the final pair:

List A	List B
$q\#\#$	#

to complete the solution.

Example 9.8

Consider the Turing Machine M and $w = 01$, where $M = (\{q_1, q_2, q_3\}, \{0,1\}, \{0,1,B\}, \delta, q_1, B, \{q_3\})$ and δ is given by

q_i	$\delta(q_i,0)$	$\delta(q_i,1)$	$\delta(q_i,B)$
q_1	$\delta(q_2,1, R)$	$\delta(q_2,0,L)$	$\delta(q_2,1,L)$
q_2	$\delta(q_3,0,L)$	$\delta(q_1,0,R)$	$\delta(q_2,0,R)$
q_3	–	–	–

Reduce the above problem to Post's Correspondence Problem and find whether that PCP has a solution or not.

Solution :

An instance of MPCP with lists A and B as follows with $w = 01$.

First pair $\rightarrow$	List A	List B
	#	# q_1 01 #

Note: # indicates blank in list A, #q_1 01# indicates start state followed by w and blank.

Group I	List A	List B
	0	0
	1	1
	#	#

Note: Group I represents, each symbol belongs to Γ.(i.e.,) $X \in \Gamma$.

$$\left.\begin{array}{ll} 0q_1\# & q_2 01 \\ 1q_1\# & q_2 11\# \end{array}\right\} \quad \because \delta(q_1,\mathrm{B}) = (q_2,1,\mathrm{L})$$

$$\left.\begin{array}{ll} 0q_2 0 & q_3 00 \\ 1q_2 0 & q_3 10 \end{array}\right\} \quad \because \delta(q_2,0) = (q_3,0,\mathrm{L})$$

$$q_2 1 \quad 0q_1 \quad \because \delta(q_2,1) = (q_1,0,\mathrm{R})$$

$$q_2\# \quad 0q_2\# \quad \because \delta(q_2,\mathrm{B}) = (q_2,0,\mathrm{R})$$

Note: Group II is constructed for each q in Q-F, p in Q and X,Y and Z in Γ. For the transition

- List A is constructed with left hand side of the transition state - input (i.e) $q_1 0$ and list B is constructed with right hand side of the transition symbol written - state (i.e) $1q_2$ for the head movement on the right side.

If the transition is $\delta(q_1,1) = (q_2,0,\mathrm{L})$, that is the head is moved to the left side of the tape, then 2 steps are constructed . In list A symbol of ∑-state-input (i.e) [$0q_1$ 1, $1q_1 1$ where ∑={0,1} and in list B state - symbol of ∑-symbol written (i.e) [$q_2 00$, $q_2 10$ where ∑={0,1}

If the transition is $\delta(q_2,\mathrm{B}) = (q_2,0,\mathrm{R})$, that is on blank , an # symbol is added at the end of list A and list B instances.

Similarly other instances of list A and list B are constructed from the given transition table.

Group IV List A List B

$\delta(q_1,0) = (q_2,1,R)$ q_3 ## #

Group III	List A	List B
	$0q_3 0$	q_3
	$0q_3 1$	q_3
	$1q_3 0$	q_3
	$1q_3 1$	q_3
	$0q_3$	q_3
	$1q_3$	q_3
	$q_3 0$	q_3
	$q_3 1$	q_3

Note: Group III is constructed for each q in F and x, y in Γ,

Note: Group IV is constructed for each q in F.

For the input w = 01, the computation is :

$q_1\ 01$
$\vdash 1\, q_2\ 1$
$\vdash 10\, q_1\ B$
$\vdash 1\, q_2\ 01$
$\vdash q_3\ 101$
$\vdash q_3\ 01 \vdash q_3\ 1 \vdash q_3$

Since q_3 is a final state, a solution to the instance of MPCP is written from list A and list B of follows:

9.10 P AND NP PROBLEMS

The languages are said to be tractable if it can be recognized within reasonable time and space constraints. A tractable problem to be solvable in polynomial time. The common problems for which no polynomial time algorithms require exponential run time.

The languages recognizable in deterministic polynomial time is P class of problems that can be solved efficiently.

There are number of problems that do not appear to be in P but have efficient non deterministic algorithms is NP.

The difference between P and NP is the difference between efficiently finding a proof of a statement ("this graph has a Hamilton Circuit") and efficiently verifying a proof ("checking that a particular circuit is Hamilton").

The deterministic polynomial and non-deterministic polynomial's, time and space are defined by:

$$\text{P} = \bigcup_{i \geq 1} \text{DTIME}(n^i)$$

$$\text{NP} = \bigcup_{i \geq 1} \text{NTIME}(n^i)$$

$$\text{PSPACE} = \bigcup_{i \geq 1} \text{DSPACE}(n^i)$$

$$\text{NPSPACE} = \bigcup_{i \geq 1} \text{USPACE}(n^i)$$

Definition

A language L' is said to be polynomial time reducible to L if there is a polynomial time bounded *TM* that for each input x produces an output y that is in L if and only if x is in L'.

Lemma

Let L' be polynomial-time reducible to L. Then

a) L' is in NP if L is in NP,

b) L' is in P if L is in P.

Proof

Assume that the reduction is $p_1(n)$ time bounded and that L is recogniazable in time $p_2(n)$, where p_1 and p_2 are polynomials. Then L' can be recognized in polynomial time as follows. Given input x of length n, produce y using the polynomial-time reduction. As the reduction is $p_1(n)$ time bounded, and at most one symbol can be printed per move, if follows that $|y| \leq p_1(n)$. Then, we can test if y is in L' in time $(p_2(p_1(n))$. Thus the total time to tell whether x is in L is $p_1(n) + p_2(p_1(n))$, which is polynomial in n. Therefore, L is in P.

NP-complete and NP-hard Problems

It is obvious the P$\subseteq$NP. It is not known that whether there exists some language L in NP, which is also accepted by deterministic TM.

A language L is said to be NP-hard if $L_1 \leq$ PL for every $L_1 \subseteq$NP. The language L is NP-complete if $L \subseteq$NP and L is NP-hard.

Examples for NP-complete problems

(i) L_{vc}, the vertex cover problem.

(ii) L_{dh}, the directed Hamilton circuit problem.

(iii) L_h, the Hamilton circuit problem for undirected graph.

Example for NP-hard problem

(i) Integer linear programming.