Unit 5 notes

Turing machine:

Informal Definition:

We consider here a basic model of TM which is deterministic and have one-tape. There are many variations, all are equally powerfull.

The basic model of TM has a finite set of states, a semi-infinite tape that has a leftmost cell but is infinite to the right and a tape head that can move left and right over the tape, reading and writing symbols.

For any input w with |w|=n, initially it is written on the n leftmost (continguous) tape cells. The infinitely many cells to the right of the input all contain a blank symbol, B whcih is a special tape symbol that is not an input symbol. The machine starts in its start state with its head scanning the leftmost symbol of the input w. Depending upon the symbol scanned by the tape head and the current state the machine makes a move which consists of the following:

- writes a new symbol on that tape cell, •
  moves its head one cell either to the left or to the right and
- (possibly) enters a new state.

The action it takes in each step is determined by a transition functions. The machine continues computing (i.e. making moves) until

- it decides to "accept" its input by entering a special state called accept or final state or
- halts without accepting i.e. rejecting the input when there is no move defined.

On some inputs the TM many keep on computing forever without ever accepting or rejecting the input, in which case it is said to "loop" on that input

Formal Definition :

Formally, a deterministic turing machine (DTM) is a 7-tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$, where

- Q is a finite nonempty set of states.
- Γ is a finite non-empty set of tape symbols, callled the tape alphabet of M.
- $\Sigma \subseteq \Gamma$ is a finite non-empty set of input symbols, called the input alphabet of M.
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L \times R\}$ is the transition function of M,

- $q_0 \in Q$ is the initial or start state.
- $B \in \Gamma \setminus \Sigma$ is the blank symbol
- $F \subseteq Q$ is the set of final state.

So, given the current state and tape symbol being read, the transition function describes the next state, symbol to be written on the tape, and the direction in which to move the tape head (L and R denote left and right, respectively).

Transition function : δ

- The heart of the TM is the transition function, δ because it tells us how the machine gets one step to the next.
- when the machine is in a certain state q$\in Q$ and the head is currently scanning the tape symbol $X \in \Gamma$, and if $\delta(q, x) = (p, Y, D)$, then the machine

1. replaces the symbol X by Y on the tape
2. goes to state p, and
3. the tape head moves one cell (i.e. one tape symbol) to the left (or right) if D is L (or R).

The ID (instantaneous description) of a TM capture what is going out at any moment i.e. it contains all the information to exactly capture the "current state of the computations".

It contains the following:

- The current state, q
- The position of the tape head,
- The constants of the tape up to the rightmost nonblank symbol or the symbol to the left of the head, whichever is rightmost.

Note that, although there is no limit on how far right the head may move and write nonblank symbols on the tape, at any finite

time, the TM has visited only a finite prefix of the infinite tape.

An ID (or configuration) of a TM M is denoted by $\alpha q \beta$ where $\alpha, \beta \in \Gamma^*$ and

- α is the tape contents to the left of the head
- q is the current state.
- β is the tape contents at or to the right of the tape head

That is, the tape head is currently scanning the leftmost tape symbol of β. (Note that if $\beta = \in$, then the tape head is scanning a blank symbol)

If q_0 is the start state and w is the input to a TM M then the starting or initial configuration of M is onviously denoted by $q_0 w$

Moves of Turing Machines

To indicate one move we use the symbol $\vdash$. Similarly, zero, one, or more moves will be represented by $\vdash$. A move of a TM

M is defined as follows.

Let $\alpha Z q X \beta$ be an ID of M where $X, Z \in \Gamma$, $\alpha, \beta \in \Gamma^*$ and $q \in Q$.

Let there exists a transition $\delta(q, X) = (p, Y, L)$ of M.

Then we write $\alpha Z q X \beta \vdash_M \alpha q Z Y \beta$ meaning that ID $\alpha Z q X \beta$ yields $\alpha Z q Y \beta$

- Alternatively , if $\delta(q, X) = (p, Y, R)$ is a transition of M, then we write $\alpha Z q X \beta \vdash \alpha z Y p \beta$ which means that the ID $\alpha z q X \beta$ yields $\alpha z Y p \beta$

- In other words, when two IDs are related by the relation $\vdash$, we say that the first one yields the second (or the second is the result of the first) by one move.
- If IDj results from IDi by zero, one or more (finite) moves then we write $\vdash$(If the TM M is understand, then the subscript M can be dropped from $\vdash$or $\vdash$)

Special Boundary Cases

- Let $q x \alpha$ be an ID and $\delta(q, x) = (p, Y, L)$ be an transition of M. Then $\vdash$. That is, the head is not allowed to fall off the left end of the tape.
- Let $q x \alpha$ be an ID and $\delta(q, x) = (p, Y, R)$ then figure (Note that $\alpha Y q$ is equivalent to $\alpha Y q B$)
- Let $q x \alpha$ be an ID and $\delta(q, x) = (p, B, R)$ then figure
- Let $\alpha z q x$ be an ID and $\delta(q, x) = (p, B, L)$ then figure

The language accepted by a TM $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$, denoted as $L(M)$ is

$L(M)$ = { w | $w \in \Sigma^*$ and figure for some $p \in F$ and $\alpha, \beta \in \Gamma^*$ }

In other words the TM M accepts a string $w \in \Sigma^*$ that cause M to enter a final or accepting state when started in its initial ID (i.e. $q_0 w$). That is a TM M accepts the string $w \in \Sigma^*$ if a sequence of IDs, $ID_1, ID_2, \cdots, ID_k$ exists such that

- ID_1 is the initial or starting ID of M
- $ID_i \vdash_M ID_{i+1}$ $1 \le i < k$

- The representation of IDk contains an accepting state.

The set of strings that M accepts is the language of M, denoted $L(M)$, as defined above

More about configuration and acceptance

- An ID $\alpha q \beta$ of M is called an accepting (or final) ID if $q \in F$
- An ID $\alpha q x \beta$ is called a blocking (or halting) ID if $\delta(q, x)$ is undefined i.e. the TM has no move at this point.
- ID_j is called reactable from ID_i if $ID_i \vdash_M^* ID_j$
- $q_0 w$ is the initial (or starting) ID if $w \in \Sigma^*$ is the input to the TM and $q_0 \in Q$ is the initial (or start) state of M.

On any input string $w \in \Sigma^*$

either

- M halts on w if there exists a blocking (configuration) ID, I' such that $q_0 w \vdash_M^* I'$

There are two cases to be considered

- M accepts w if I is an accepting ID. The set of all $w \in \Sigma^*$ accepted by M is denoted as $L(M)$ as already defined
- M rejects w if I' is a blocking configuration. Denote by reject (M), the set of all $w \in \Sigma^*$ rejected by M.

or

- M loops on w if it does not halt on w.

Let $loop(M)$ be the set of all $w \in \Sigma^*$ on which M loops for.

It is quite clear that

$$L(M) \cup reject(M) \cup loop(M) = \Sigma^*$$

That is, we assume that a TM M halts

- When it enters an accepting ID_1 or
- When it enters a blocking ID_1 i.e. when there is no next move.

However, on some input string, , $w \notin L(M)$, it is possible that the TM M loops for ever i.e. it never halts

The Halting Problem

The input to a Turing machine is a string. Turing machines themselves can be written as strings. Since these strings can be used as input to other Turing machines. A "Universal Turing machine" is one whose input consists of a description *M* of some arbitrary Turing machine, and some input w to which machine *M* is to be applied, we write this combined input as *M* + *w*. This produces the same output that would be produced by M. This is written as
Universal Turing Machine (*M* + *w*) = *M* (*w*).

As a Turing machine can be represented as a string, it is fully possible to supply a Turing machine as input to itself, for example *M* (*M*). This is not even a particularly bizarre thing to do for example, suppose you have written a C pretty printer in C, then used the Pretty printer on itself. Another common usage is Bootstrapping—where some convenient languages used to write a minimal compiler for some new language L, then used this minimal compiler for L to write a new, improved compiler for language L. Each time a new feature is added to language L, you can recompile and use this new feature in the next version of the compiler. Turing machines sometimes halt, and sometimes they enter an infinite loop.

A Turing machine might halt for one input string, but go into an infinite loop when given some other string. The halting problem asks: "It is possible to tell, in general, whether a given machine will halt for some given input?" If it is possible, then there is an effective procedure to look at a Turing machine and its input and determine whether the machine will halt with that input. If there is an effective procedure, then we can build a Turing machine to implement it. Suppose we have a Turing machine "WillHalt" which, given an input string *M* + *w*, will halt and accept the string if Turing machine *M* halts on input *w* and will halt and reject the string if Turing machine *M* does not halt on input *w*. When viewed as a Boolean function, "WillHalt (*M*, *w*)" halts and returns "TRUE" in the first case, and (halts and) returns "FALSE" in the second.

Theorem

Turing Machine "WillHalt (*M*, *w*)" does not exist.

Proof: This theorem is proved by contradiction. Suppose we could build a machine "WillHalt". Then we can certainly build a second machine, "LoopIfHalts", that will go into an infinite loop if and only if "WillHalt" accepts its input:

```
Function LoopIfHalts (M, w):
if WillHalt (M, w) then
while true do { }
else
return false;
```

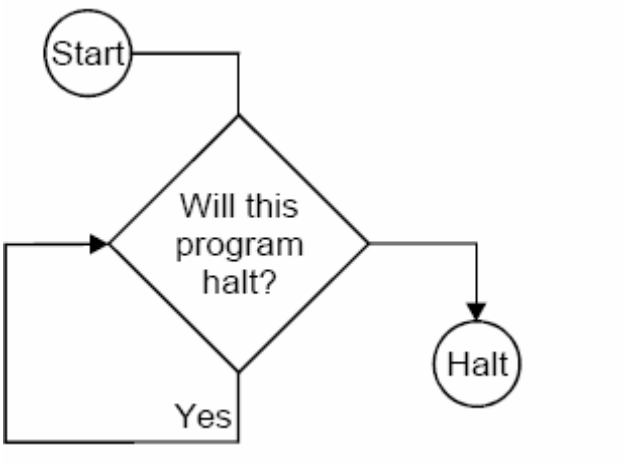
We will also define a machine "LoopIfHaltOnItSelf" that, for any given input *M*, representing a Turing machine, will determine what will happen if *M* is applied to itself, and loops if *M* will halt in this case.

```
Function LoopIfHaltsOnItself (M):
return LoopIfHalts (M, M):
```

Finally, we ask what happens if we try:

```
Func tion Impos sible:
return LoopIfHaltsOnItself (LoopIfHaltsOnItself):
```

This machine, when applied to itself, goes into an infinite loop if and only if it halts when applied to itself. This is impossible. Hence the theorem is proved.



Implications of Halting Problem

Programming

The Theorem of "Halting Problem" does not say that we can never determine whether or not a given program halts on a given input. Most of the times, for practical reasons, we could eliminate infinite loops from programs. Sometimes a "meta-program" is used to check another program for potential infinite loops, and get this meta-program to work most of the time.

The theorem says that we cannot ever write such a meta-program and have it work all of the time. This result is also used to demonstrate that certain other programs are also impossible.

The basic outline is as follows:

(i) If we could solve a problem *X*, we could solve the Halting problem

(ii) We cannot solve the Halting Problem

(iii) Therefore, we cannot solve problem *X*

A Turing machine can be "programmed," in much the same manner as a computer is programmed. When one specifies the function which we usually call δ for a Tm, he is really writing a program for the Tm.

1. Storage in finite Control

The finite control can be used to hold a finite amount of information. To do so, the state is written as a pair of elements, one exercising control and the other storing a symbol. It should be emphasized that this arrangement is for conceptual purposes only. No modification in the definition of the Turing machine has been made.

Example

Consider the Turing machine

Solution

$$T = (K, \{0, 1\}, \{0, 1, B\}, \delta, [q_0, B], F),$$

where K can be written as $\{q_0, q_1\} \times \{0, 1, B\}$. That is, K consists of the pairs $[q_0, 0]$, $[q_0, 1]$, $[q_0, B]$, $[q_1, 0]$, $[q_1, 1]$, and $[q_1, B]$. The set F is $\{[q_1, B]\}$. T looks at the first input symbol, records it in its finite control, and checks that the symbol does not appear elsewhere on its input. The second component of the state records the first input symbol. Note that T accepts a regular set, but T will serve for demonstration purposes. We define δ as follows.

1. a) $\delta([q_0, B], 0) = ([q_1, 0], 0, R)$
   b) $\delta([q_0, B], 1) = ([q_1, 1], 1, R)$
   (T stores the symbol scanned in second component of the state and moves right. The first component of T's state becomes q_1.)

2. a) $\delta([q_1, 0], 1) = ([q_1, 0], 1, R)$
   b) $\delta([q_1, 1], 0) = ([q_1, 1], 0, R)$
   (If T has a 0 stored and sees a 1, or vice versa, then T continues to move to the right.)

3. a) $\delta([q_1, 0], B) = ([q_1, B], 0, L)$
   b) $\delta([q_1, 1], B) = ([q_1, B], 0, L)$
   (T enters the final state $[q_1, B]$ if T reaches a blank symbol without having first encountered a second copy of the leftmost symbol.)

If T reaches a blank in state $[q_1, 0]$ or $[q_1, 1]$, it accepts. For state $[q_1, 0]$ and symbol 0 or for state $[q_1, 1]$ and symbol 1, δ is not defined, so if T ever sees the symbol stored, it halts without accepting.

In general, we can allow the finite control to have k components, all but one of which store information.

2. Multiple Tracks

We can imagine that the tape of the Turing machine is divided into k tracks, for any finite k. This arrangement is shown in Fig., with k = 3. What is actually done is that the symbols on the tape are considered as k-tuples. One component for each track.

Example

The tape in Fig. can be imagined to be that of a Turing machine which takes a binary input greater than 2, written on the first track, and determines if it is a prime. The input is surrounded by ¢ and $ on the first track.

Thus, the allowable input symbols are [¢, B, B], [0, B, B], [1, B, B], and [$, B, B]. These symbols can be identified with ¢, 0, 1, and $, respectively, when viewed as input symbols. The blank

symbol can be represented by [B, B, B]
To test if its input is a prime, the Tm first writes the number two in binary on the second track and copies the first track onto the third track. Then, the second track is subtracted, as many times as possible, from the third track, effectively dividing the third track by the second and leaving the remainder. If the remainder is zero, the number on the first track is not a prime. If the remainder is nonzero, increase the number on the second track by one.
If now the second track equals the first, the number on the first track is a prime, because it cannot be divided by any number between one and itself. If the second is less than the first, the whole operation is repeated for the new number on the second track. In Fig., the Tm is testing to determine if 47 is a prime. The Tm is dividing by 5; already 5 has been subtracted twice, so 37 appears on the third track.

3. Subroutines

VII. SUBROUTINES. It is possible for one Turing machine to be a "subroutine" of another Tm under rather general conditions. If T_1 is to be a subroutine of T_2, we require that the states of T_1 be disjoint from the states of T_2 (excluding the states of T_2's subroutine). To "call" T_1, T_2 enters the start state of T_1. The rules of T_1 are part of the rules of T_2. In addition, from a halting state of T_1, T_2 enters a state of its own and proceeds.

UNDECIDABILITY

Design a Turing machine to add two given integers.
Solution:

Assume that m and n are positive integers. Let us represent the input as $0^m B 0^n$.

If the separating B is removed and 0's come together we have the required output, $m + n$ is unary.

(i) The separating B is replaced by a 0.
(ii) The rightmost 0 is erased i.e., replaced by B.

Let us define $M = (\{q_0, q_1, q_2, q_3, q_4\}, \{0\}, \{0, B\}, \delta, q_0, \{q_4\})$. δ is defined by Table shown below.

	Tape Symbol	
State	0	B
q_0	$(q_0, 0, R)$	$(q_1, 0, R)$
q_1	$(q_1, 0, R)$	(q_2, B, L)
q_2	(q_3, B, L)	—
q_3	$(q_3, 0, L)$	(q_4, B, R)

M starts from ID $q_0 0^m B 0^n$, moves right until seeking the blank B. M changes state to q_1. On reaching the right end, it reverts, replaces the rightmost 0 by B. It moves left until it reaches the beginning of the input string. It halts at the final state q_4.

Some unsolvable Problems are as follows:
(i) Does a given Turing machine *M* halts on all input?
(ii) Does Turing machine *M* halt for any input?
(iii) Is the language *L*(*M*) finite?
(iv) Does *L*(*M*) contain a string of length *k*, for some given *k*?
(v) Do two Turing machines *M*1 and *M*2 accept the same language?
It is very obvious that if there is no algorithm that decides, for an arbitrary given Turing machine *M* and input string *w*, whether or not *M* accepts *w*. These problems for which no algorithms exist are called "UNDECIDABLE" or "UNSOLVABLE".

Code for Turing Machine:

Our next goal is to devise a binary code for Turing machines so that each TM with input alphabet $\{0,1\}$ may be thought of as a binary string. Since we just saw how to enumerate the binary strings, we shall then have an identification of the Turing machines with the integers, and we can talk about "the ith Turing machine, M_i." To represent a TM $M = (Q, \{0,1\}, \Gamma, \delta, q_1, B, F)$ as a binary string, we must first assign integers to the states, tape symbols, and directions L and R.

- We shall assume the states are $q_1, q_2, \ldots, q_r$ for some r. The start state will always be q_1, and q_2 will be the only accepting state. Note that, since we may assume the TM halts whenever it enters an accepting state, there is never any need for more than one accepting state.

- We shall assume the tape symbols are $X_1, X_2, \ldots, X_s$ for some s. X_1 always will be the symbol 0, X_2 will be 1, and X_3 will be B, the blank. However, other tape symbols can be assigned to the remaining integers arbitrarily.

- We shall refer to direction L as D_1 and direction R as D_2.

Since each TM M can have integers assigned to its states and tape symbols in many different orders, there will be more than one encoding of the typical TM. However, that fact is unimportant in what follows, since we shall show that no encoding can represent a TM M such that $L(M) = L_d$.

Once we have established an integer to represent each state, symbol, and direction, we can encode the transition function δ. Suppose one transition rule is $\delta(q_i, X_j) = (q_k, X_l, D_m)$, for some integers i, j, k, l, and m. We shall code this rule by the string $0^i10^j10^k10^l10^m$. Notice that, since all of i, j, k, l, and m are at least one, there are no occurrences of two or more consecutive 1's within the code for a single transition.

A code for the entire TM M consists of all the codes for the transitions, in some order, separated by pairs of 1's:

$$C_1 11 C_2 11 \cdots C_{n-1} 11 C_n$$

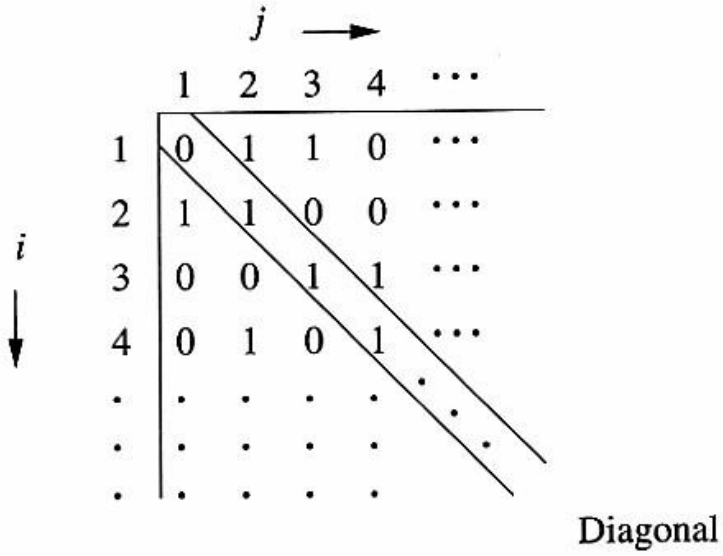
where each of the C's is the code for one transition of M.

Diagonalization language:

- The language L_d, the *diagonalization language*, is the set of strings w_i such that w_i is not 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 reason L_d is called a "diagonalization" language can be seen if we consider Fig. 9.1. This table tells for all i and j, whether the TM M_i accepts input string w_j; 1 means "yes it does" and 0 means "no it doesn't."[1] We may think of the ith row as the *characteristic vector* for the language $L(M_i)$; that is, the 1's in this row indicate the strings that are members of this language.



This table represents language acceptable by Turing machine

The diagonal values tell whether M_i accepts w_i. To construct L_d, we complement the diagonal. For instance, if Fig. 9.1 were the correct table, then the complemented diagonal would begin $1, 0, 0, 0, \ldots$. Thus, L_d would contain $w_1 = \epsilon$, not contain w_2 through w_4, which are 0, 1, and 00, and so on.

The trick of complementing the diagonal to construct the characteristic vector of a language that cannot be the language that appears in any row, is called *diagonalization*. It works because the complement of the diagonal is

Proof that L_d is not recursively enumerable:

Theorem 9.2: L_d is not a recursively enumerable language. That is, there is no Turing machine that accepts L_d.

PROOF: Suppose L_d were $L(M)$ for some TM M. Since L_d is a language over alphabet $\{0,1\}$, M would be in the list of Turing machines we have constructed, since it includes all TM's with input alphabet $\{0,1\}$. Thus, there is at least one code for M, say i; that is, $M = M_i$.

Now, ask if w_i is in L_d.

- 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 such that M_j does *not* accept w_j.
- Similarly, 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 can neither be in L_d nor fail to be in L_d, we conclude that there is a contradiction of our assumption that M exists. That is, L_d is not a recursively enumerable language. □

Recursive Languages:

We call a language L *recursive* if $L = L(M)$ for some Turing machine M such that:

1. If w is in L, then M accepts (and therefore halts).
2. If w is not in L, then M eventually halts, although it never enters an accepting state.

A TM of this type corresponds to our informal notion of an "algorithm," a well-defined sequence of steps that always finishes and produces an answer. If we think of the language L as a "problem," as will be the case frequently, then problem L is called *decidable* if it is a recursive language, and it is called *undecidable* if it is not a recursive language.

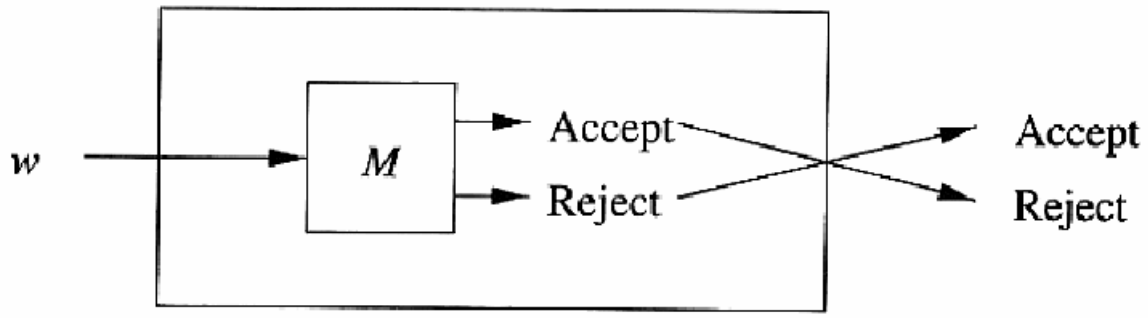
Theorem 9.3: If L is a recursive language, so is $\overline{L}$.

PROOF: Let $L = L(M)$ for some TM M that always halts. We construct a TM $\overline{M}$ such that $\overline{L} = L(\overline{M})$ by the construction suggested in Fig. 9.3. That is, $\overline{M}$ behaves just like M. However, M is modified as follows to create $\overline{M}$:

1. The accepting states of M are made nonaccepting states of $\overline{M}$ with no transitions; i.e., in these states $\overline{M}$ will halt without accepting.

2. $\overline{M}$ has a new accepting state r; there are no transitions from r.

3. For each combination of a nonaccepting state of M and a tape symbol of M such that M has no transition (i.e., M halts without accepting), add a transition to the accepting state r.



Since M is guaranteed to halt, we know that $\overline{M}$ is also guaranteed to halt. Moreover, $\overline{M}$ accepts exactly those strings that M does not accept. Thus $\overline{M}$ accepts $\overline{L}$. □

Theorem 9.4: If both a language L and its complement are RE, then L is recursive. Note that then by Theorem 9.3, $\overline{L}$ is recursive as well.

PROOF: The proof is suggested by Fig. 9.4. Let $L = L(M_1)$ and $\overline{L} = L(M_2)$. Both M_1 and M_2 are simulated in parallel by a TM M. We can make M a two-tape TM, and then convert it to a one-tape TM, to make the simulation easy and obvious. One tape of M simulates the tape of M_1, while the other tape of M simulates the tape of M_2. The states of M_1 and M_2 are each components of the state of M.

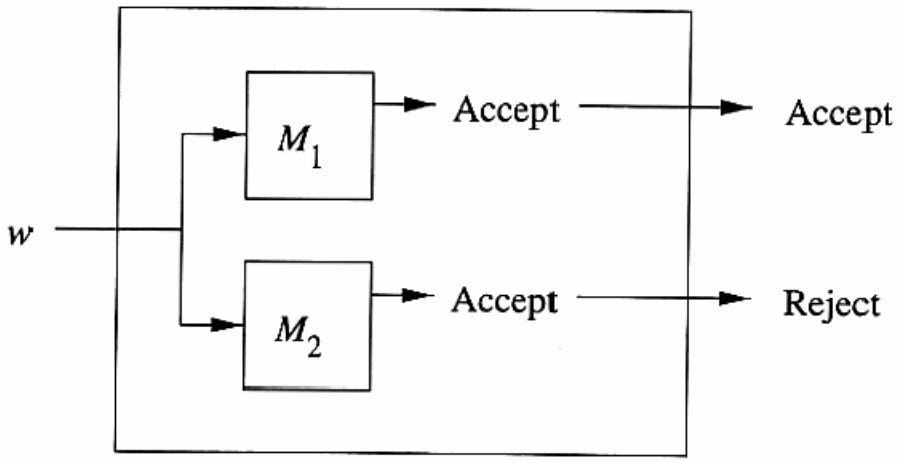


Figure 9.4: Simulation of two TM's accepting a language and its complement

If input w to M is in L, then M_1 will eventually accept. If so, M accepts and halts. If w is not in L, then it is in $\overline{L}$, so M_2 will eventually accept. When M_2 accepts, M halts without accepting. Thus, on all inputs, M halts, and $L(M)$ is exactly L. Since M always halts, and $L(M) = L$, we conclude that L is recursive. □

Universal Language:

We define L_u, the *universal language*, to be the set of binary strings that encode, in the notation of Section 9.1.2, a pair (M, w), where M is a TM with the binary input alphabet, and w is a string in $(\mathbf{0}+\mathbf{1})^*$, such that w is in $L(M)$. That is, L_u is the set of strings representing a TM and an input accepted by that TM. We shall show that there is a TM U, often called the *universal Turing machine*, such that $L_u = L(U)$. Since the input to U is a binary string, U is in fact some M_j in the list of binary-input Turing machines we developed in

Undecidability of Universal Language:

Theorem 9.6: L_u is RE but not recursive.

PROOF: We just proved in Section 9.2.3 that L_u is RE. Suppose L_u were recursive. Then by Theorem 9.3, $\overline{L_u}$, the complement of L_u, would also be recursive. However, if we have a TM M to accept $\overline{L_u}$, then we can construct a TM to accept L_d (by a method explained below). Since we already know that L_d is not RE, we have a contradiction of our assumption that L_u is recursive.

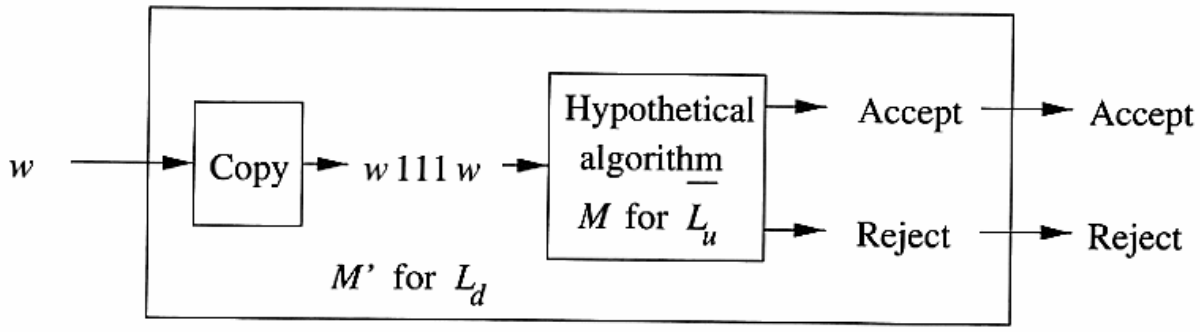


Figure 9.6: Reduction of L_d to $\overline{L_u}$

Suppose $L(M) = \overline{L_u}$. As suggested by Fig. 9.6, we can modify TM M into a TM M' that accepts L_d as follows.

1. Given string w on its input, M' changes the input to $w111w$. You may, as an exercise, write a TM program to do this step on a single tape. However, an easy argument that it can be done is to use a second tape to copy w, and then convert the two-tape TM to a one-tape TM.

2. M' simulates M on the new input. If w is w_i in our enumeration, then M' determines whether M_i accepts w_i. Since M accepts $\overline{L_u}$, it will accept if and only if M_i does not accept w_i; i.e., w_i is in L_d.

Thus, M' accepts w if and only if w is in L_d. Since we know M' cannot exist by Theorem 9.2, we conclude that L_u is not recursive. □

Problem -Reduction :
If P_1 reduced to P_2,
Then P_2 is at least as hard as P_1.
Theorem: If P_1 reduces to P_2 then,

- If P_1 is undecidable the so is P_2.
- If P_1 is Non-RE then so is P_2.

Post's Correspondence Problem (PCP)

A post correspondence system consists of a finite set of ordered pairs $(x_i, y_i),\ i = 1, 2, \cdots, n,$ where $x_i, y_i \in \Sigma^+$ for some alphabet Σ.

Any sequence of numbers $i_1, i_2, \cdots, i_k \quad s - t.$

is called a solution to a Post Correspondence System.

$x_{i_1}, x_{i_2}, \cdots, x_{i_k} = y_{i_1}, y_{i_2}, \cdots, y_{i_k}$ The Post's Correspondence Problem is the problem of determining whether a Post Correspondence system has a solutions.

Example 1 : Consider the post correspondence system

$\{(aa, aab), (bb, ba), (abb, b)\}$ The list 1,2,1,3 is a solution to it.

Because

$x_1 x_2 x_1 x_3 = y_1 y_2 y_1 y_3$

$\underbrace{aa}_{x_1}\underbrace{bb}_{x_2}\underbrace{aa}_{x_1}\underbrace{abb}_{x_3} = \underbrace{aab}_{y_1}\underbrace{ba}_{y_2}\underbrace{aab}_{y_1}\underbrace{b}_{y_3}$

$aabbaaabb = aabbaaabb$

i	x_i	y_i
1	aa	aab
2	bb	bb
3	abb	b

(A post correspondence system is also denoted as an instance of the PCP)

Example 2 : The following PCP instance has no solution

i	x_i	y_i
1	aab	aa
2	a	baa

This can be proved as follows. (x_2, y_2) cannot be chosen at the start, since than the LHS and RHS would differ in the first symbol (a in LHS and $'b'$ in RHS). So, we must start with (x_1, y_1) . The next pair must be (x_2, y_2) so that the 3 rd symbol in the RHS becomes identical to that of the LHS, which is a . After this step, LHS and RHS are not matching. If (x_1, y_1) is selected next, then would be mismatched in the 7 th symbol

(γ in LHS and a in RHS). If (x_2, y_2) is selected, instead, there will not be any choice to match the both side in the next step.

Example3 : The list 1,3,2,3 is a solution to the following PCP instance.

i	x_i	y_i
1	1	101
2	10	00
3	011	11

The following properties can easily be proved.

Proposition The Post Correspondence System

$\left\{\left(a^{i_1}, a^{j_1}\right), \left(a^{i_2}, a^{j_2}\right), \cdots, \left(a^{i_n}, a^{j_n}\right)\right\}$ has solutions if and only if

$\exists k$ such that $i_k = j_k$ or
$\exists k$ and l such that $i_k > j_k$ and $i_l < j_l$

Corollary : PCP over one-letter alphabet is decidable.

Proposition Any PCP instance over an alphabet Σ with $|\Sigma| \geq 2$ is equivalent to a PCP instance over an alphabet Γ with $|\Gamma| = 2$

Proof : Let $\Sigma = \{a_1, a_2, \cdots, a_k\}, k > 2.$

Consider $\Gamma = \{0,1\}$ We can now encode every $a_i \in \Sigma, 1 \leq i \leq k$ as $10^i 1.$ any PCP instance over Σ will now have only two symbols, 0 and 1 and, hence, is equivalent to a PCP instance over Γ

Theorem : PCP is undecidable. That is, there is no algorithm that determines whether an arbitrary Post Correspondence System has a solution.

Proof: The halting problem of turning machine can be reduced to PCP to show the undecidability of PCP. Since halting problem of TM is undecidable (already proved), This reduction shows that PCP is also undecidable. The proof is little bit lengthy and left as an exercise.

Some undecidable problem in context-free languages

We can use the undecidability of PCP to show that many problem concerning the context-free languages are undecidable. To prove this we reduce the PCP to each of these problem. The following discussion makes it clear how PCP can be used to serve this purpose.

Let $\{(x_1,y_1),(x_2,y_2),\cdots,(x_n,y_n)\}$ be a Post Correspondence System over the alphabet Σ. We construct two CFG's G_x and G_y from the ordered pairs *x,y* respectively as follows.

$G_x = (N_x, \Sigma_x, P_x, S_x)$ and

$G_y = (N_y, \Sigma_y, P_y, S_y)$ where

$G_y = (N_y, \Sigma_y, P_y, S_y)$

$N_x = \{S_x\}$ and $N_y = \{S_y\}$

$\Sigma_x = \Sigma_y = \Sigma \bigcup \{1,2,\cdots,n\}$,

$P_x = \{S_x \to x_i S_x i,\ S_x \to x_i i \mid i = 1,2,\cdots,n\}$

and $P_y = \{S_y \to y_i S_y i,\ S_y \to y_i i \mid i = 1,2,\cdots n\}$

it is clear that the grammar G_2 generates the strings that can appear in the LHS of a sequence while solving the PCP followed by a sequence of numbers. The sequence of number at the end records the sequence of strings from the PCP instance (in reverse order) that generates the string. Similarly, G_y generates the strings that can be obtained from the RHS of a sequence and the corresponding sequence of numbers (in reverse order).

Now, if the Post Correspondence System has a solution, then there must be a sequence

$i_1 i_2, \cdots, i_k$ *s.t*

$x_{i_1} x_{i_2} \cdots x_{i_k} = y_{i_1} y_{i_2} \cdots y_{i_k}$

According to the construction of G_2 and G_y

$S_x \overset{*}{\underset{G_x}{\Rightarrow}} x_{i_1} x_{i_2} \cdots x_{i_k} i_k i_{k-1} \ldots i_2 i_1$ and

$S_y \overset{*}{\underset{G_y}{\Rightarrow}} y_{i_1} y_{i_2} \cdots y_{i_k} i_k i_{k-1} \cdots i_2 i_1$

In this case

$$x_{i_1} x_{i_2} \cdots x_{i_l} i_k \cdots i_2 i_1 = y_{i_1} y_{i_2} \cdots i_{i_l} i_k \cdots i_2 i_1 = w(\text{say})$$

Hence , $w \in L(G_x)$ and $w \in L(G_y)$ implying

$$L(G_x) \bigcap L(G_y) \neq \phi$$

Conversely, let $w \in L(G_x) \bigcap L(G_y)$

Hence, *w* must be in the form w_1w_2 where $w_1 \in \Sigma^*$ and w_2 in a sequence $i_k i_{k-1} \cdots i_2 i_1$(since, only that kind of strings can be generated by each of G_2 and G_y).

Now, the string $w_1 = x_{i_1} x_{i_2} \cdots x_{i_l} = y_{i_1} y_{i_2} \cdots y_{i_l}$ is a solution to the Post Correspondence System.

It is interesting to note that we have here reduced PCP to the language of pairs of CFG,s whose intersection is nonempty. The following result is a direct conclusion of the above.

Theorem : Given any two CFG's G_1 and G_2 the question "Is $L(G_1) \bigcap L(G_2) = \phi ?$ " is undecidable.

Proof: Assume for contradiction that there exists an algorithm *A* to decide this question. This would imply that PCP is decidable as shown below.

For any Post Correspondence System, *P* construct grammars G_2 and G_y by using the constructions

elaborated already. We can now use the algorithm *A* to decide whether and $L(G_x) \bigcap L(G_y) \neq \phi$ Thus, PCP is decidable, a contradiction. So, such an algorithm does not exist.

If G_2 and G_y are CFG's constructed from any arbitrary Post Correspondence System, than it is not difficult to show that $\overline{L(G_x)}$ and $\overline{L(G_y)}$ are also context-free, even though the class of context-free languages are not closed under complementation.

$L(G_x), L(G_y)$ and their complements can be used in various ways to show that many other questions related to CFL's are undecidable. We prove here some of those.

Theorem : Foe any two arbitrary CFG's $G_1 \ \& G_2$, the following questions are undecidable

i. Is $L(G_1) = \Sigma^* ?$
ii. Is $L(G_1) = L(G_2) ?$

iii. Is $L(G_1) \subseteq L(G_2)$?

Proof :

i. If $L(G_1) = \Sigma^*$ then, $\overline{L(G_1)} = \phi$

Hence, it suffice to show that the question “Is $L(G_1) = \phi$?" is undecidable.

Since, $\overline{L(G_x)}$ and $\overline{L(G_y)}$ are CFl's and CFL's are closed under union, $L = \overline{L(G_x)} \cup \overline{L(G_y)}$ is also context-free. By DeMorgan's theorem, $\overline{L} = L(G_x) \cap L(G_y)$

If there is an algorithm to decide whether $L(G_1) = \phi$ we can use it to decide whether $\overline{L} = L(G_x) \cap L(G_y) = \phi$ or not. But this problem has already been proved to be undecidable.

Hence there is no such algorithm to decide or not. $L(G_1) = \phi$

ii.

Let *P* be any arbitrary Post correspondence system and G_2 and G_y are CFg's constructed from the pairs of strings.

$L_1 = \overline{L(G_x)} \cup \overline{L(G_y)}$ must be a CFL and let G_1generates L_1. That is,

$$L_1 = L(G_1) = \overline{L(G_x)} \cup \overline{L(G_y)} = \overline{L(G_x) \cap L(G_y)}$$

by De Morgan's theorem, as shown already, any string, $w \in L(G_x) \cap L(G_y)$ represents a solution to the PCP. Hence, $L(G_1)$ contains all but those strings representing the solution to the PCP.

Let $L(G_2) = (\Sigma \cup \{1, 2, \cdots, n\})^*$ for same CFG G_2.

It is now obvious that $L(G_1) = L(G_2)$ if and only if the PCP has no solutions, which is already proved to be undecidable. Hence, the question “Is $L(G_1) = L(G_2)$?" is undecidable.

iii.

Let G_1 be a CFG generating the language $(\Sigma \cup \{1, 2, \cdots n\})^*$ and G_2 be a CFG generating $\overline{L(G_x)} \cup \overline{L(G_y)}$ where G_2 and G_y are CFG.s constructed from same arbitrary instance of PCP.

$$L(G_1) \subseteq L(G_2) \text{ iff } \overline{L(G_x)} \cup \overline{L(G_y)} = (\Sigma \cup \{1, 2, \cdots n\})^*$$

i.e. iff the PCP instance has no solutions as discussed in part (ii).

Hence the proof.

Theorem : It is undecidable whether an arbitrary CFG is ambiguous.

Proof : Consider an arbitrary instance of PCP and construct the CFG's G_2 and G_y from the ordered pairs of strings.

We construct a new grammar *G* from G_2 and G_y as follows.

$G = (N, \Sigma, P, S)$ where

$N = \{S, S_x, S_y\}$,

Σ is same as that of G_2 and G_y.

$$P = \{P_x, \cup P_y \cup \{S \to S_x | S_y\}$$

This constructions gives a reduction of PCP to the ---------of whether a CFG is ambiguous, thus leading to the undecidability of the given problem. That is, we will now show that the PCP has a solution if and only if *G* is ambiguous. (where *G* is constructed from an arbitrary instance of PCP).

Only if Assume that $i_1, i_2, \cdots i_k$ is a solution sequence to this instance of PCP.

Consider the following two derivation in $i_1, i_2, \cdots i_k$.

$$\begin{aligned} S &\overset{1}{\underset{G}{\Rightarrow}} S_x \overset{1}{\underset{G}{\Rightarrow}} x_{i_1} S_x i_1 \overset{1}{\underset{G}{\Rightarrow}} x_{i_1} x_{i_2} S_x i_2 i_1 \\ &\overset{*}{\underset{G}{\Rightarrow}} x_{i_1} x_{i_2} \cdots x_{i_{k-1}} S_x i_{k-1} \cdots i_2 i_1 \\ &\overset{1}{\underset{G}{\Rightarrow}} x_{i_1} x_{i_2} \cdots x_{i_{k-1}} x_{i_k} i_k i_{k-1} \cdots i_2 i_1 \end{aligned}$$

$$S \overset{1}{\underset{G}{\Rightarrow}} S_y \overset{1}{\underset{G}{\Rightarrow}} y_{i_1} S_y i_1 \overset{1}{\underset{G}{\Rightarrow}} y_{i_1} y_{i_2} S_y i_2 i_1$$
$$\overset{*}{\underset{G}{\Rightarrow}} y_{i_1} y_{i_2} \cdots x y_{i_{k-1}} S_y i_{k-1} \cdots i_2 i_1$$
$$\overset{1}{\underset{G}{\Rightarrow}} y_{i_1} y_{i_2} \cdots y_{i_{k-1}} y_k i_k i_{k-1} \cdots i_2 i_1$$

But ,

$$x_{i_1} x_{i_2} \cdots x_{i_k} = y_{i_1} y_{i_2} \cdots y_k, \text{ since } \cdots i_1, i_2, \cdots i_k$$

is a solution to the PCP. Hence the same string of terminals $\left(x_{i_1} x_{i_2} \cdots x_{i_k}\right)$ has two derivations. Both these derivations are, clearly, leftmost. Hence *G* is ambiguous.

If It is important to note that any string of terminals cannot have more than one derivation in G_2 and G_y Because, every terminal string which are derivable under these grammars ends with a sequence of integers $i_k i_{k-1} \cdots i_2 i_1$ This sequence uniquely determines which productions must be used at every step of the derivation.

Hence, if a terminal string, $w \in L(G)$, has two leftmost derivations, then one of them must begin with the step.

then continues with derivations under G_y

In both derivations the resulting string must end with a sequence $i_p i_{p-1} \cdots i_2 i_1$ for same $p \geq 1$ The reverse of this sequence must be a solution to the PCP, because the string that precede in one case is $x_1 x_2 \cdots x_{i_{p-1}} x_{i_p}$ and $y_1 y_2 \cdots y_{i_{p-1}} y_{i_p}$ in the other case. Since the string derived in both cases are identical, the sequence $i_1, i_2, \cdots i_{p-1}, i_p$

must be a solution to the PCP.

Hence the proof

Class p-problem solvable in polynomial time:

A Turing machine M is said to be of *time complexity* $T(n)$ [or to have "running time $T(n)$"] if whenever M is given an input w of length n, M halts after making at most $T(n)$ moves, regardless of whether or not M accepts. This definition applies to any function $T(n)$, such as $T(n) = 50n^2$ or $T(n) = 3^n + 5n^4$; we shall be interested predominantly in the case where $T(n)$ is a polynomial in n. We say a language L is in class $\mathcal{P}$ if there is some polynomial $T(n)$ such that $L = L(M)$ for some deterministic TM M of time complexity $T(n)$.

Non deterministic polynomial time:

A nondeterministic TM that never makes more than p(n) moves in any sequence of choices for some polynomial p is said to be non polynomial time NTM.

- NP is the set of languags that are accepted by polynomial time NTM's
- Many problems are in NP but appear not to be in p.
- One of the great mathematical questions of our age: is there anything in NP that is not in p?

NP-complete problems:

If We cannot resolve the "p=np question, we can at least demonstrate that certain problems in NP are the hardest , in the sense that if any one of them were in P , then P=NP.

- These are called NP-complete.
- Intellectual leverage: Each NP-complete problem's apparent difficulty reinforces the belief that they are all hard.

Methods for proving NP-Complete problems:

- Polynomial time reduction (PTR): Take time that is some polynomial in the input size to convert instances of one problem to instances of another.
- If P1 PTR to P2 and P2 is in P1 the so is P1.
- Start by showing every problem in NP has a PTR to Satisfiability of Boolean formula.
- Then, more problems can be proven NP complete by showing that SAT PTRs to them directly or indirectly.