6 Unsolvable Problems and Computational Functions

Syllabus

Unsolvable problems and computable functions - Primitive recursive functions - Recursive and recursively enumerable languages - Universal Turing machine. Measuring and Classifying Complexity : Tractable and intractable problems - Tractable and possibly intractable problems - P and NP completeness - Polynomial time reductions.

Contents

6.1 Unsolvability

6.2 Primitive Recursive Functions

6.3 Recursive and Recursively Enumerable Languages
. . . . . **Dec.-03 ,04, 11,**
. . . . . **May-05, 06, 07, 09, 11, 12** · Marks 16

6.4 Universal Turing Machine. **May-05, 06, 07, 09, 12, 14,**
. . . . . **Dec.-03, 04, 11, 14** · · · · · · Marks 16

6.5 Tractable and Intractable Problems

6.6 Tractable and Possibly Intractable Problems : P and NP
. . . . . **Dec.-12, 13,** · · · · · · · · · · · · Marks 6

6.7 NP Completeness . . . . . **Dec.-11,** · · · · · · · · · · · · · · · · Marks 8

6.8 Polynomial Time Reductions

6.9 Proving NP Complete Problems

6.10 Post's Correspondence Problem **Dec.-07, 10, 11, 12, 14,**
. . . . . **May-13,** · · · · · · · · · · · · · · · · · Marks 8

6.11 Two Marks Question with Answers

6.12 University Questions with Answers (Long Answered Questions)

6.1 Unsolvability

In the theory of computation we often come across such problems that are answered either **yes** or **no**. The class of problems which can be answered as yes are is called solvable or decidable, otherwise the class of problems is said to be Unsolvable or Undecidable.

6.2 Primitive Recursive Functions

Recursive function is class of functions those are turing computable. The theory of recursive functions is just converse to Church's hypothesis. Recursive function theory begins with some very elementary functions that are intuitively effective. Then it provides a few methods for building more complicated functions from simpler functions. That means the computability of given function can be proved using the initial function and some building operations which are few in number.

Initial function : The initial functions are the elementary functions whose values are independent of their smaller arguments. The following functions comprise the class of recursive functions.

The zero function : z(x) = 0

The successor function : s(x) = successor of x (roughly , "x+1")

The identity function : id(x) = x

The zero functions returns zero regardless of its argument.

The successor functions returns the successor of its argument. Since successorship is a more primitive notion.

The zero and successor functions take only one argument each. But the identity functions is designed to take any number of arguments. When it takes one argument (as above) it returns its argument as its value. When it takes more than one argument, it returns one of them. That means,

$$id(x,y) = x$$

$$id(x,y) = y$$

Building operations : We will build more complex functions from the initial set by using only three methods that are,

i) Composition

ii) Primitive recursion

iii) Minimization.

i) Composition

We will start with the successor function,

$$s(x) = x + 1$$

Then we may replace its argument, x, with a function. If we replace the argument, x, with the zero function,

$$z(x)$$

then the result is the successor of zero,

$$s(z(x)) = 1$$

$$s(s(z(x))) = 2 \text{ and so on.}$$

In this way, with the help of the initial functions we can describe the natural numbers. This building operations is called "composition". It should be clear that when composition is applied to computable functions, only computable functions will result.

ii) Primitive recursion

The second building operation is called primitive recursion. Primitive recursion is a method of defining new functions from old function. The function h is defined through functions f and g by primitive recursion when

$$h(x, 0) = f(x)$$

$$h(x, s(y)) = g(x, h(x, y))$$

where f and g are known computable functions. There are two equations. When h's second argument is zero, the first equation applies; when it is not zero, we use the second. Use of successor function in second equation enforces the condition that the argument be greater than zero. Hence, the first equation applies in the minimal case and the second applied in every other case.

To solve the function by primitive recursion

1) When the second argument of h is zero, the function is equivalent to some known function f and we compute it;

2) Otherwise it is equivalent to some known function g and we compute it.

Thus the function obtained will be computable in nature. For example, we can calculate the factorial function using recursion as :

Initially 1! = 1 and to calculate n!, if we multiply n by (n – 1) then it will generate a nonrecursive series. Instead of that we will multiply n by (n – 1)! We can express the calculation of factorial function by following two equations -

$$1! = 1$$

$$n! = n*(n-1)$$

A strict definition of the factorial function, f(n) then, consists of these two equations :

$$f(n) = 1 \quad \text{when } n = 1 \quad ...(6.2.1)$$

$$f(n) = n*(f(n-1)) \quad \text{when } n > 1 \quad ...(6.2.2)$$

Consider n = 5 then using equation (6.2.2) we will get,

$$f(5) = 5*f(4)$$

$$f(4) = 4*f(3)$$

$$f(3) = 3*f(2)$$

$$f(2) = 2*f(1)$$

By putting the value of equation (6.2.1) for calculating f(2) we will get,

$$f(2) = 2*1 = 2$$

Then $f(3) = 3*f(2)$

$= 3*2$

$= 6$

Then $f(4) = 4*f(3)$

$= 4*6$

$= 24$

Then $f(5) = 5*f(4)$

$= 5*24$

$= 120$

Primitive recursion is like mathematical induction. The first equation defines the basis, and the second defines the induction step.

iii) Minimization

If g(x) is a function that computes the least x such that f(x) = 0, then we know that g is computable. And then we can say that g is produced from f by minimization. But we can build g by minimization only if f(x) is already known to be computable.

For example : Suppose we want to obtain least x which makes f(x) = 0 then we will try the natural numbers 0,1,2,..... until we reach the first value that gives f(x) = 0. Now if such search for x never gets terminated then it is called unbounded minimization. And if we obtain such x within some tests then it is called bounded minimization. While

unbounded minimization has the disadvantages of a partial function which may never terminate, bounded minimization has the disadvantage of sometimes failing to minimize.

6.2.1 Class of Recursive Functions

The classification of recursive functions is as shown in Fig. 6.2.1.

1. Partial Recursive Function

The function is called partial recursive function if it can be obtained by applying composition, primitive recursion and minimization as building operations.

2. General Recursive Function

The function is called general recursive if it is obtained by applying composition, primitive recursive and an unbounded minimization that happen to terminate. The general recursive function is a larger class than partial recursive functions.

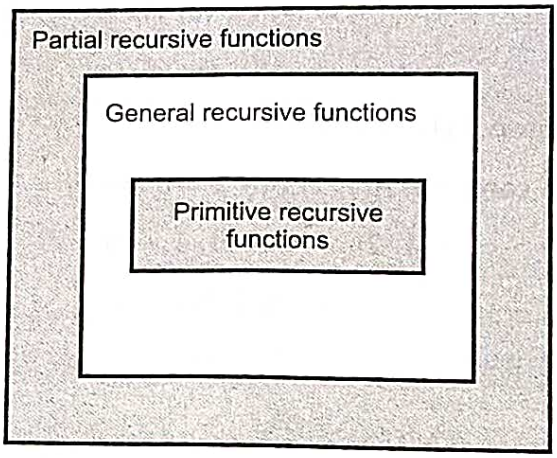


Fig. 6.2.1 Classification of recursive functions

3. Primitive Recursive Function

The function is called primitive recursive if it is obtained by applying composition, primitive recursion and unbounded minimization that does not terminate. The set of general recursive function is the same as the set of turing computable functions. The example of general recursive function is an Ackermann's function. The Ackermann's function can be defined as follows -

$$A(0, y) = y + 1$$

$$A(x+1, 0) = A(x, 1)$$

$$A(x+1, y+1) = A(x, A(x+1, y))$$

Then we can calculate A(x,y) for every pair of (x,y).

Example 6.2.1 *Compute A(1, 1) A(1, 2), A(2, 1), A(2, 2) using Ackermann's function.*

Solution : Let

$$A(0, y) = y + 1 \quad \ldots (1)$$

$$A(x+1, 0) = A(x, 1) \quad \ldots (2)$$

$$A(x+1, y+1) = A(x, A(x+1, y)) \quad \ldots (3)$$

Be the Ackermann's function,

To compute A(1, 1) put x = 0, y = 0, then

$$A(1, 1) = A(0+1, 0+1) \quad \text{...Using equation (3)}$$
$$= A(0, A(0+1, 0)) \quad \text{...Using equation (2)}$$
$$= A(0, A(0, 1)) \quad \text{... Using equation (1)}$$
$$= A(A(0, 1)) \quad \text{...Using equation (1)}$$
$$= A(0, 2)$$

Hence A(1, 1) = 3

To compute A(1, 2) put x = 0 and y = 1

$$A(1, 2) = A(0+1, 1 + 1) \quad \text{...Using equation (3)}$$
$$= A(0, A(1, 1))$$
$$= A(0, 3) \quad \text{...Using equation (1)}$$
$$= 4$$

To compute A(2, 1) put x = 1 and y = 0

$$A(2, 1) = A(1+1, 0+1)$$
$$= A(1, A(2, 0))$$
$$= A(1, A(1, 1))$$
$$= A(1, 3)$$
$$= A(0+1, 2+1)$$
$$= A(0, A(1, 2))$$
$$= A(0, 4)$$
$$= 5$$

To compute A(2, 2) put x = 1 and y = 1

$$A(2, 2) = A(1+1, 1+1)$$
$$= A(1, A(2, 1))$$
$$= A(1, 5)$$

Now we will compute A(1, 5) where in x = 0 and y = 4.

$$A(1, 5) = A(0+1, 4+1)$$
$$= A(0, A(1, 4))$$
$$= A(0, (A(0+1, 3+1)))$$

$= A(0,(A(0,A(1, 3))))$

$= A(0,(A(0,A(0+1,(2+1))))$

$= A\ (0,\ (A\ (0,A(0,A(1, 2))))$

$= A(0,(A(0,A(0,(4))))$

$= A(0,(A(0, 5)))$

$= A(0, 6)$

$A(1, 5) = 7$

Hence $A(2, 2) = A(1, 5)$

$A(2, 2) = 7$

6.3 Recursive and Recursively Enumerable Languages

AU : Dec.-03 ,04, 11, May-05, 06, 07, 09, 11, 12, Marks 16

The recursively enumerable language has two categories. The first category consists of all the languages that has some algorithm and an algorithm for language L can be modeled by a Turing machine. Naturally such turing machine always halts on valid input and enters in accept state, but on invalid input (the input that is not belonging to language L) it halts without entering in halt state.) Thus languages of this category possess a property of definiteness. The languages of this category are particularly called as recursive languages.

In the second category, the languages can be modeled by Turing machines but there is no guarantee that the TM will eventually halt. In the sense that the we can not predict that Turing machine will halt or will enter in an infinite loop for certain input. Such type of languages can be denoted by pair (M, w) where M is a Turing machine and w is an input string. These languages of this category are typically called as Recursively Enumerable languages (RE).

There are three categories of the languages -

1. Recursive language for which the algorithm exists.
2. Recursively enumerable language for which it is not sure that on which input the TM will ever halt. Such languages are not recursive.
3. The non recursively enumerable languages for which there is no Turing Machine at all.

Following Fig. 6.3.1 can show the relationship between these languages.

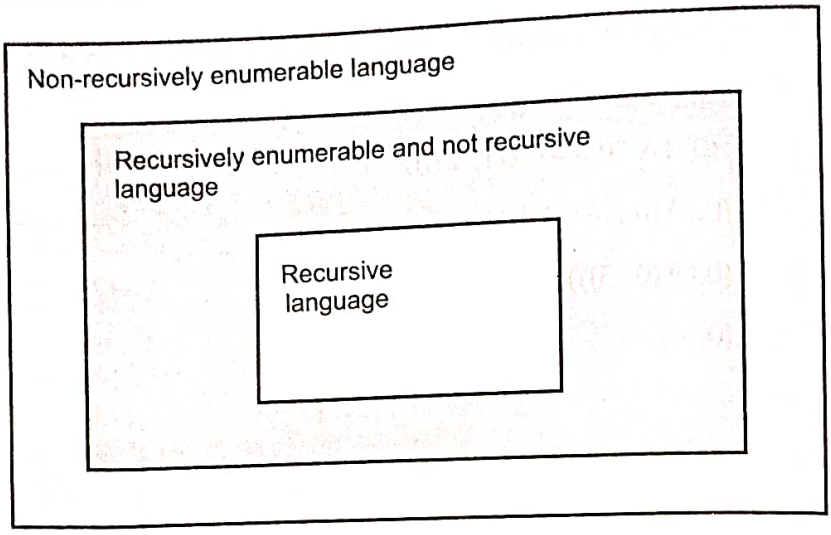


Fig. 6.3.1

6.3.1 Properties of Recursive and Recursively Enumerable Languages

Decidable and Undecidable Languages - If a language is recursive then it is called decidable languages and if the language is not recursive then such a language is called undecidable language.

Let us discuss some important theorems that prove the important closure properties of recursive languages.

Theorem 1 : If L is recursive language then L' is also a recursive language.

AU : May-05, 11, Marks 2; May-09, Marks 6

Proof : Let there will be some L that can be accepted by Turing machine M. Hence we can denote language L by L(M). On acceptance of L(M) the machine M always halts. Now, we construct a TM M' such that L' = (M'). for construction of M' following steps are followed -

1. The accepting steps of M are made non-accepting states of M' and there is no transition from M'. That means we have created the states such that M' will halt without accepting.
2. Now create a new accepting state for M' say r and there is no transition from r.
3. In machine M, for each of the transition with combination of non accepting state and input tape symbol, make the same transition having the combination of accepting state and input tape symbol for machine M'.

Since M is guaranteed to halt M' is also guaranteed to halt. In

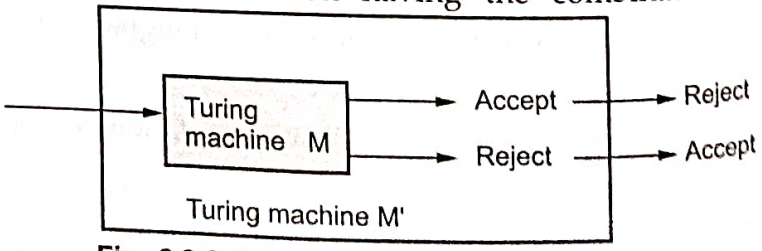


Fig. 6.3.2 Construction of M ' accepting L'

fact, M' accepts exactly those strings that M does not accept. Thus we can say that M' accepts L'

Theorem 2 : If a language L and its complement L' both are RE then L is a recursive language. **AU : Dec.-03, May-07, Marks 6; Dec.-04, May-06, Marks 4; May-05, Marks 16**

Proof : Consider a Turing machine M made up of two Turing machines M1 and M2. The machine M2 is complement of machine M1. We can also denote that L(M) = L(M1) and L(M2). Both M1 and M2 are simulated in parallel by machine M. Machine M is a two tape TM, which can be made one tape TM for the ease of simulation. This One tape then will consist of tape of machine M1 and machine M2. The states of M consists of all the states of machine M1 and all the states of machine M2. The machine M made up of M1 and M2 is as shown in Fig. 6.3.3.

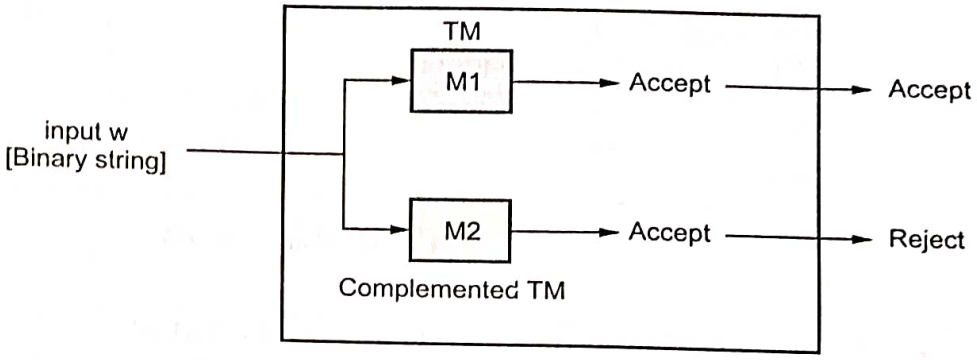


Fig. 6.3.3

If the input w of language L is given to M then M1 will eventually accept and therefore M will accept L and halt. If w is not in L, then it is in L'. So M2 will accept and therefore M will half without accepting. Thus on all inputs, M halts. Thus L(M) is exactly L. Since M always halts we can conclude that L(M) mean L is a recursive language. Thus a recursive language can be recursively enumerable but a recursively enumerable language is not necessarily be recursive.

6.4 Universal Turing Machine

AU : May-05, 06, 07, 09, 12, 14, Dec.-03, 04, 11, 14, Marks 16

The universal language L_u is a set of binary strings which can be modeled by a Turing machine. The universal language can be represented by a pair (M, w) where M is a TM that accepts this languages and w is a binary sring in$(0 + 1)^*$ such that w belongs to L(M). Thus we can say that any binary string belongs to a universal language. From this the concept of universal Turing machine came up. The universal language can be represented by L_u = L(U) where U is a universal Turing Machine. In fact U is a binary string. This binary string represents various codes of many Turing machines. Thus the universal Turing machine is a turing machine which accepts many Turing machines. The TM U is a multitape Turing machine which can be as shown in Fig. 6.4.1.

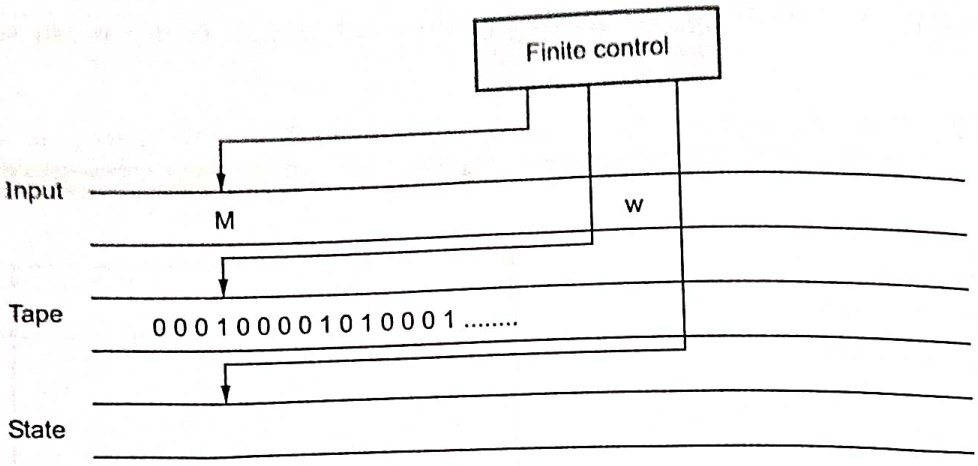


Fig. 6.4.1 Multitape TM

- The universal Turing machine U accepts the TM M
- The transitions of M are stored initially on first tape along with the string w.
- On the second tape the simulated tape of M is placed. Here the tape symbol X_i of M will be represented by 0^i and tape symbols are separated by single 1's.
- On the third tape various states of machine M are stored. The state q_i is represented by i 0's.

Operations on Turing Machine

1. The first tape is observed carefully to check whether the code for M is valid or not. If the code is not valid then U halts without accepting. As invalid codes are assumed to represent the TM with no moves, such TM accepts no inputs and halts.
2. Initialize the second tape for placing the tape values of M in encoded form that means for each 0 of the input w place 10 on the second tape and for each 1 place 100 there. The blank letters of tape of machine M are represented by 1000. But actually blank appears at the end of input w for the machine U. However the blanks of M are simulated by 1000.
3. As the third tape stores states of machine M, place 0 there as the start state of machine M. The head of third tape will now position at start state (i.e. at 0) and the head of third tape will now position at start state (i.e. at 0) and the head for the tape 2 will position at the first simulated cell of second tape.
4. Now the move of M can be simulated. As on the first tape transitions are stored U first searches the tape 1 to know the transitions which are always given in the form $0^i\ 10^j\ 10^k\ 10^l\ 10^m$, as we have already seen that such a Turing machine code consists of

   0^i means current state,

0^j means current input tape symbol,

0^k means changed state,

0^l means changed input tape symbol,

0^m means direction in which the head is to be moved.

Now to simulate the move of M change the content of tape 3 to 0^k that is, simulate the state change of M. Then replace 0^j on tape 2 by 0^l, that is simulate the change in input. Then move the tape head on tape to the position of next 1. The tape head moves to left or right depending upon the value of m. that means if m = 1 then move left and if m = 2 move right. Thus U simulates the move of M to left or to the right.

5. If M has no transition then no transition is performed. Therefore M halts in the simulated configuration and hence U halts.

6. If M enters in accept state, then U accepts.

Thus the TM U simulates M on binary input string w where a pair (M, w) is also coded in binary form.

6.4.1 Undecidability of the Universal Languages

AU : May-14, Marks 14

The universal language L_u is a recursively enumerable language we have to prove that it is undecidable (means not recursive).

Theorem : L_u is RE but not recursive.

Proof : Consider that language L_u is recursively enumerable language. We will assume that L_u is recursive. Then the complement of L_u that is L'_u is also recursive. However if we have a TM M to accept L'_u then we can construct a TM L_d. But L_d the diagonalization language is not RE. Thus our assumption that L_u is recursive is wrong (not RE means not recursive). Hence we can say that L_u is RE but not recursive. The construction of M for Ld is as shown in Fig. 6.4.2.

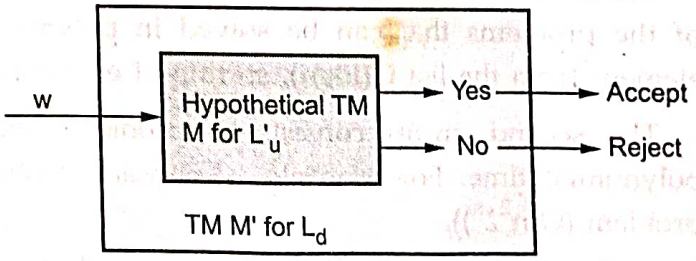


Fig. 6.4.2 Construction of M' for L_d

Review Questions

1. *Show that the complement of a recursive language is recursive.*

AU : Dec.-03; May-09, Marks 6

2. *If a language L and its complement $\bar{L}$ are both recursively enumerable then show that L and hence $\bar{L}$ is recursive.*

AU : Dec.-03, May-07, Marks 6, Dec.-04, May-06, Marks 4; May-05, Marks 16

3. *Write short note on recursive and recursively enumerable languages.*

AU : Dec.-11, Marks 8

4. *Discuss the properties of recursive langauges.*

AU : May-12, Marks 8

5. *Prove that the universal language is recursively enumerable.*

AU : May-14, Marks 10

6. *Define the language L_u and show that L_u is RE language.*

AU : Dec.-14, Marks 4

6.5 Tractable and Intractable Problems

- The class of solvable problems is known as **decidable problems**. That means decidable problems can be solved in measurable amount of time or space.
- The tractable problems are the class of problems that can be solved within reasonable time and space.
- The intractable problems are the class of problems that can be solved within polynomial time. This has lead to two classes of solving problems - P and NP class problems.

6.6 Tractable and Possibly Intractable Problems : P and NP

AU : Dec.-12, 13, Marks 6

There are two groups in which a problem can be classified. The first group consists of the problems that can be solved in polynomial time. For example: searching of an element from the list O(logn), sorting of elements O(logn).

The second group consists of problems that can be solved in non-deterministic polynomial time. For example : Knapsack problem $O(2^{n/2})$ and Travelling Salesperson problem $(O(n^2 2^n))$.

- Any problem for which answer is either yes or no is called decision problem. The algorithm for decision problem is called **decision algorithm.**
- Any problem that involves the identification of optimal cost (minimum or maximum) is called optimization problem. The algorithm for optimization problem is called **optimization algorithm.**
- **Definition of P** - Problems that can be solved in polynomial time. ("P" stands for polynomial).

  Examples - Searching of key element, Sorting of elements, All pair shortest path.
- **Definition of NP** - It stands for "non-deterministic polynomial time". Note that NP does not stand for "non-polynomial".

Examples - Travelling Salesperson problem, Graph coloring problem, Knapsack problem, Hamiltonian circuit problems.

- The NP class problems can be further categorized into NP-complete and NP hard problems.

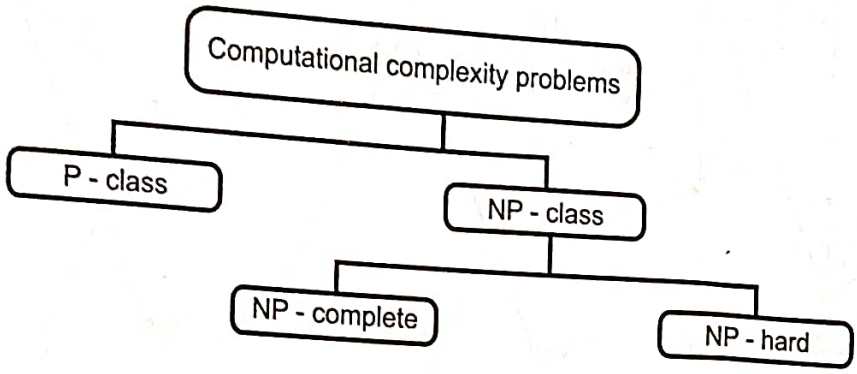


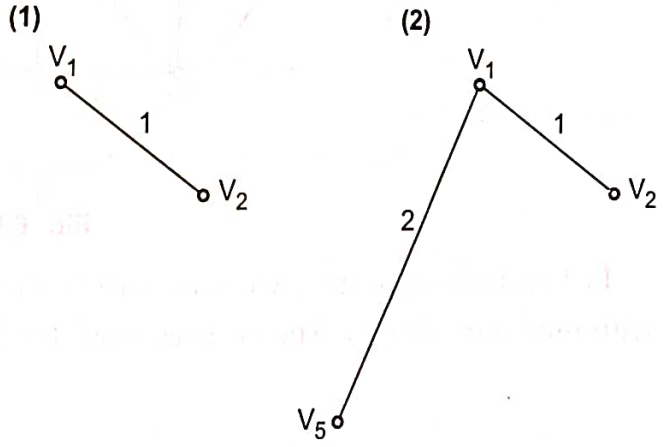
Fig. 6.6.1

- A problem D is called NP-complete if -
  i) It belongs to class NP
  ii) Every problem in NP can also be solved in polynomial time.
- If an NP-hard problem can be solved in polynomial time then all NP-complete problems can also be solved in polynomial time.
- All NP-complete problems are NP-hard but all NP-hard problems can not be NP-complete.
- The NP class problems are the decision problems that can be solved by non-deterministic polynomial algorithms.

6.6.1 Example of P Class Problem

Kruskal's Algorithm: In Kruskal's algorithm the minimum weight is obtained. In this algorithm also the circuit should not be formed. Each time the edge of minimum weight has to be selected, from the graph. It is not necessary in this algorithm to have edges of minimum weights to be adjacent. Let us solve one example by Kruskal's algorithm.

Example 1 :



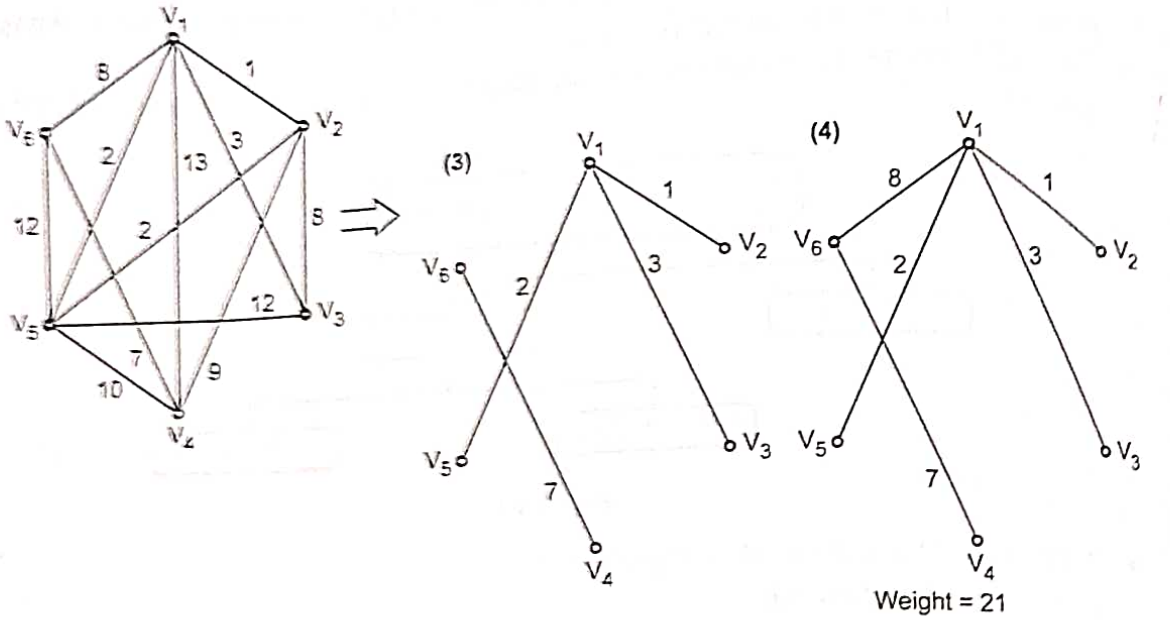


Fig. 6.6.2 Graph G

Example 2 : Find the minimum spanning tree for the following figure using Kruskal's algorithm.

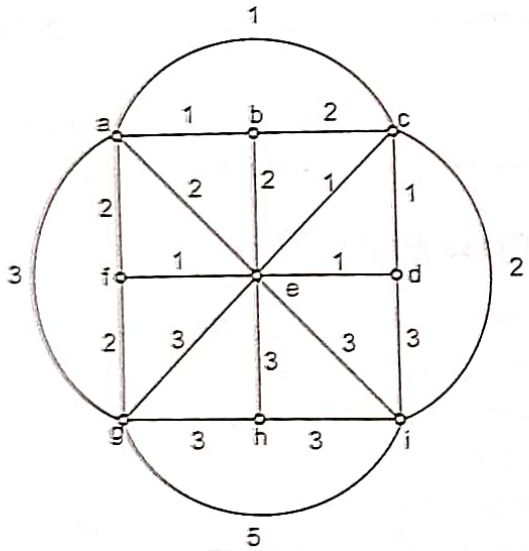


Fig. 6.6.3

In Kruskal's algorithm, we will start with some vertex and will cover all the vertices with minimum weight. The vertices need not be adjacent.

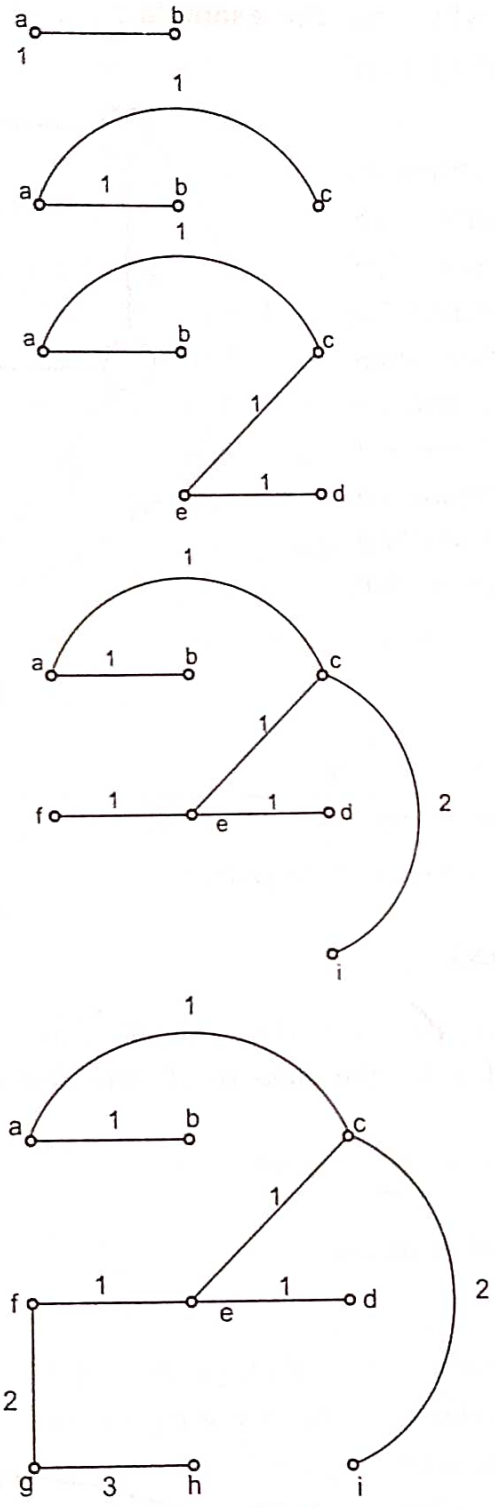


Fig. 6.6.4

6.6.2 Example of NP Class Problem

Travelling Salesman's Problem (TSP) : This problem can be stated as " Given a set of cities and cost to travel between each pair of cities, determine whether there is a path that visits every city once and returns to the first city. Such that the cost travelled is less".

The tour path will be **a-b-d-e-c-a** and total cost of tour will be **16.**

This problem is NP problem as there may exist some path with shortest distance between the cities. If you get the solution by applying certain algorithm then Travelling Salesman problem is **NPComplete Problem.** If we get no solution at all by applying an algorithm then the travelling salesman problem belongs to **NP hard class.**

For example :

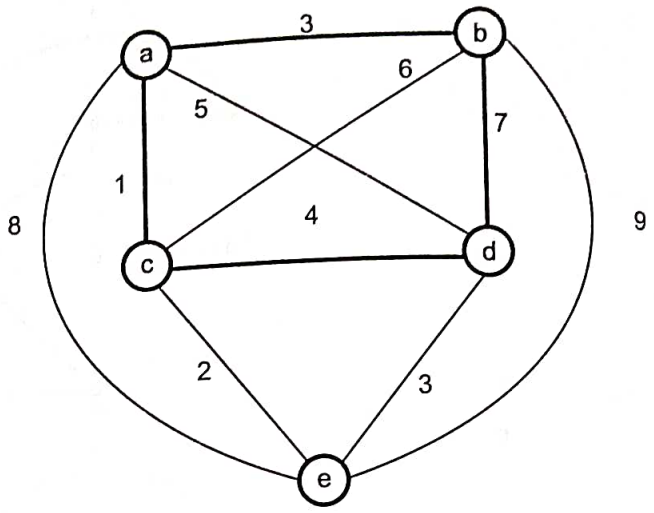


Fig 6.6.5

Review Question

1. *Write a note on NP propblems.* **AU : Dec.-12, Marks 6**
2. *Write the classes and definition of NP problems.* **AU : Dec.-13, Marks 6**

6.7 NP Completeness

AU : Dec.-11, Marks 8

- As we know, P denotes the class of all deterministic polynomial language problems and NP denotes the class of all non-deterministic polynomial language problems. Hence

$$P \subseteq NP.$$

- The question of whether or not

$$P = NP$$

holds, is the most famous outstanding problem in the computer science.

- Problems which are known to lie in P are often called as *tractable.* Problems which lie outside of P are often termed as *intractable.* Thus, the question of whether P = NP or P ≠ NP is the same as that of asking whether there exist problems in NP which are intractable or not.

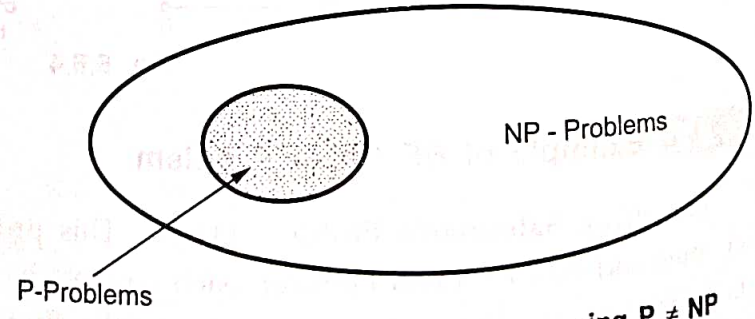


Fig. 6.7.1 P and NP problems assuming P ≠ NP

The relationship between P and NP is depicted by Fig. 6.7.1.

- We don't know if P = NP. However in 1971 S. A. Cook proved that a particular NP problem known as SAT(Satisfiability of sets of Boolean clauses) has the property that, if it is solvable in polynomial time, so are all NP problems. This is called a "NP-complete" problem.
- Let A and B are two problems then problem A reduces to B if and only if there is a way to solve A by deterministic polynomial time algorithm using a deterministic algorithm that solves B in polynomial time.

  A reduces to B can be denoted as $A \propto B$. In other words we can say that if there exists any polynomial time algorithm which solves B then we can solve A in polynomial time. We can also state that if $A \propto B$ and $B \propto C$ then $A \propto C$.
- A NP problem such that, if it is in P, then NP = P. If a (not necessarily NP) problem has this same property then it is called "NP-hard". Thus the class of NP-complete problem is the intersection of the NP and NP-hard classes.
- Normally the decision problems are NP-complete but optimization problems are NP-hard. However if problem A is a decision problem and B is optimization problem then it is possible that $A \propto B$. For instance the Knapsack decision problem can be Knapsack optimization problem.

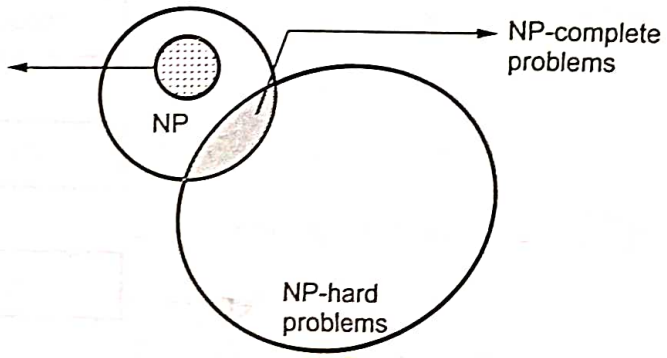


Fig. 6.7.2 Relationship between P, NP, NP-complete and NP-hard problems

- There are some NP-hard problems that are not NP-complete. For example *halting problem.* The halting problem states that : "Is it possible to determine whether an algorithm will ever halt or enter in a loop on certain input?"

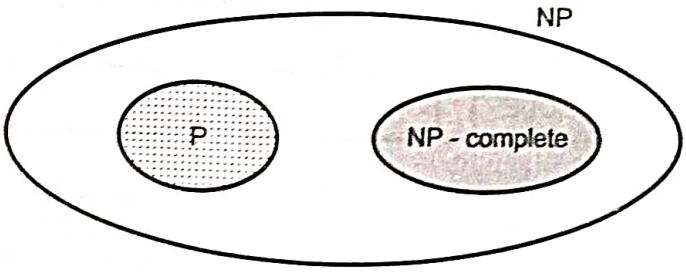


Fig. 6.7.3

- Two problems P and Q are said to be polynomially equivalent if and only if $P \propto Q$ and $Q \propto P$.

Review Questions

1. *Write short note on NP hard and NP complete problems.* **AU : Dec.-11, Marks 8**
2. *Discuss the difference between NP-complete and NP-hard problems.*

6.8 Polynomial Time Reductions

AU : Dec.-10,11, May-12, Marks 8

To prove whether particular problem is NP complete or not we use polynomial time reducibility. That means if

$A \xrightarrow{Poly} B$ and $B \xrightarrow{Poly} C$ then $A \xrightarrow{Poly} C$.

The **reduction** is an important task in NP completeness proofs. This can be illustrated by Fig. 6.8.1.

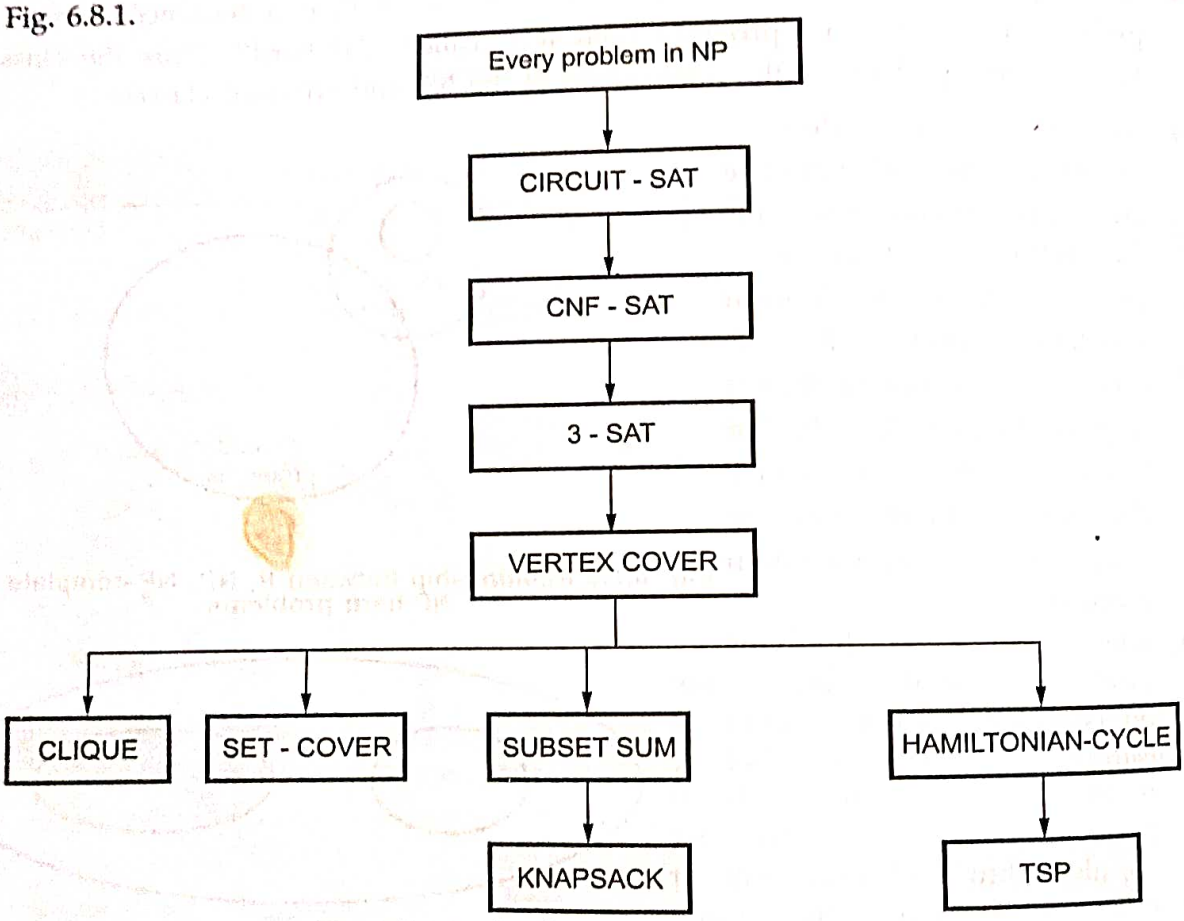


Fig. 6.8.1 Reduction in NP completeness

Various types of reductions are

Local replacement - In this reduction A→B by dividing input to A in the form of components and then these components can be converted to components of B.

Component design - In this reduction A→B by building special component for input B that enforce properties required by A.

6.9 Proving NP Complete Problems

6.9.1 The Satisfiability Problem

1. CNF - SAT problem

This problem is based on Boolean formulas. The Boolean formula has various Boolean operations such as OR(+), AND (·) and NOT. There are some notations such as $\rightarrow$ (means implies) and $\leftrightarrow$ (means if and only if).

A Boolean formula is in **Conjuctive Normal Form (CNF)** if it is formed as collection of subexpressions. These subexpressions are called **clauses.**

For example

$(\bar{a} + b + d + \bar{g})\ (c + \bar{e})\ (\bar{b} + d + \bar{f} + h)\ (a + c + e + \bar{h})$

This formula evaluates to 1 if b, c, d are 1.

The CNF-SAT is a problem which takes Boolean formula in CNF form and checks whether any assignment is there to Boolean values so that formula evaluates to 1.

Theorem : CNF-SAT is in NP complete.

Proof : Let S be the Boolean formula for which we can construct a simple non-deterministic algorithm which can guess the values of variables in Boolean formula and then evaluates each clause of S. If all the clauses evaluate S to 1 then S is satisfied. Thus CNF-SAT is in NP-complete.

2. A 3SAT Problem

A 3SAT problem is a problem which takes a Boolean formula S in CNF form with each clause having **exactly three literals** and check whether S is satisfied or not. [Note that CNF means each literal is ORed to form a clause, and each clause is ANDed to form Boolean formula S].

Following formula is an instance of 3SAT problem :

$(\bar{a} + b + \bar{g})\ (c + \bar{e} + f)\ (\bar{b} + d + \bar{f})\ (a + e + \bar{h})$

Theorem : 3SAT is in NP complete.

Proof : Let S be the Boolean formula having 3 literals in each clause for which we can construct a simple non-deterministic algorithm which can guess an assignment of Boolean values to S. If the S is evaluated as 1 then S is satisfied. Thus we can prove that 3SAT is in NP-complete.

Some other NP complete problems

1. **The 0/1 Knapsack Problem** - It can be proved as NP complete by reduction from SUM OF SUBSET problem.
2. **Hamiltonian Cycle** - It can be proved as NP complete, by reduction from vertex cover.
3. **Travelling Salesperson Problem** - It can be proved as NP complete, by reduction from Hamiltonian cycle.

1. HAMILTONIAN CYCLE : This is a problem in which graph G is accepted as input and it is asked to find a simple cycle in G that visits each vertex of G exactly once and returns to its starting vertex. Such a cycle is called an **Hamiltonian cycle.**

Theorem : HAMILTONIAN CYCLE is in NP.

Proof : Let A be some non-deterministic algorithm to which graph G is given as input. The vertices of graph are numbered from 1 to N. We have to call the algorithm recursively in order to get the sequence S. This sequence will have all the vertices without getting repeated. The vertex from which the sequence starts must be ended at the end. This check on the sequence S must be made in polynomial time n. Now if there is a Hamiltonian cycle in the graph then algorithm will output "yes". Similarly if we get output of algorithm as "yes" then we could guess the cycle in G with every vertex appearing exactly once and the first visited vertex getting visited at the last. That means A non-deterministically accepts the language HAMILTONIAN CYCLE. It is therefore proved that HAMILTONIAN CYCLE is in NP.

2. CIRCUIT-SAT : This is a problem in which a Boolean circuit is taken as input with single output node. And then finds whether there is an assignment of values to the circuit's input so that we get its output value as "1". This assignment of values is called **satisfying assignment.**

Theorem : CIRCUIT-SAT is in NP (See Fig. 6.9.1 on next page).

Proof : The logic gates that are used are

Let us construct a non-deterministic algorithm. A which works in polynomial time. Then we should have **choose ( )** method which can guess the values of input node as well as the output, then the algorithm says "no" if we get output of Boolean circuit as 0. Similarly the algorithm says "yes" if we get output of Boolean circuit as 1.

Now from the output of algorithm we can guess the inputs to logic gates in the circuit. If the algorithm has output "yes" then we can say that Boolean circuit has satisfying input values. Thus we can say that CIRCUIT-SAT is in NP.

Fig. 6.9.2

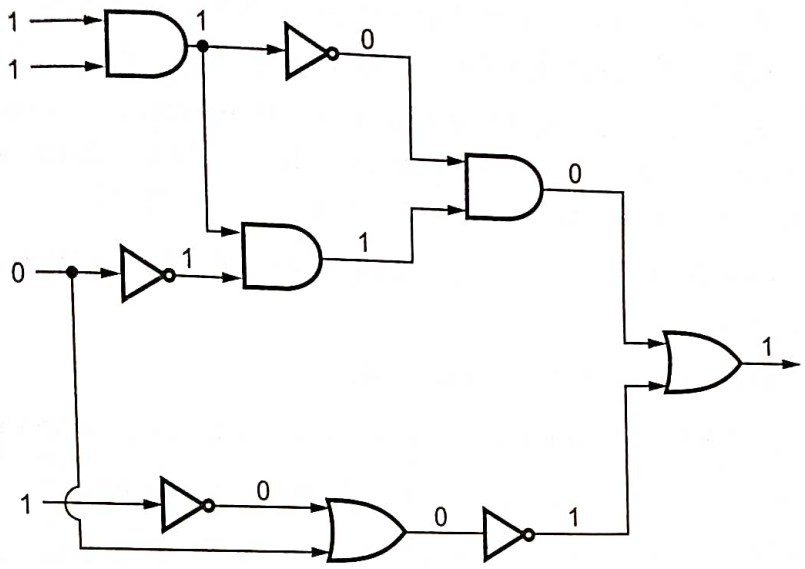


Fig. 6.9.1

6.10 Post's Correspondence Problem

AU : Dec.-07, 10, 11, 12, 14, Marks 8, May-13, Marks 6

In this section, we will discuss the undecidability of strings and not of Turing machines. The undecidability of strings is determined with the help of Post's Correspondence Problem (PCP). Let us define the PCP.

"The post's correspondence problem consists of two lists of strings that are of equal length over the input Σ. The two lists are A = $w_1, w_2, w_3, \ldots, w_n$ and B = $x_1, x_2, x_3 \ldots x_n$ then there exists a non empty set of integers $i_1, i_2, i_3, \ldots i_n$ such that $w_1, w_2, w_3, \ldots w_n = x_1, x_2, x_3, \ldots x_n$"

To solve the post correspondence problem we try all the combinations of $i_1, i_2, i_3 \ldots, i_n$ to find the $w_i = x_i$ then we say that PCP has a solution.

Example 6.10.1 *Consider the correspondence system as given below -*

A = (1; 0; 010; 11) and B = (10; 10; 01; 1). The input set is Σ = {0, 1}. find the solution.

Solution : A solution is 1; 2; 1; 3; 3; 4. That means $w_1w_2w_1w_3w_3w_4 = x_1x_2x_1x_3x_3x_4$.

The constructed string from both lists is 10101001011.

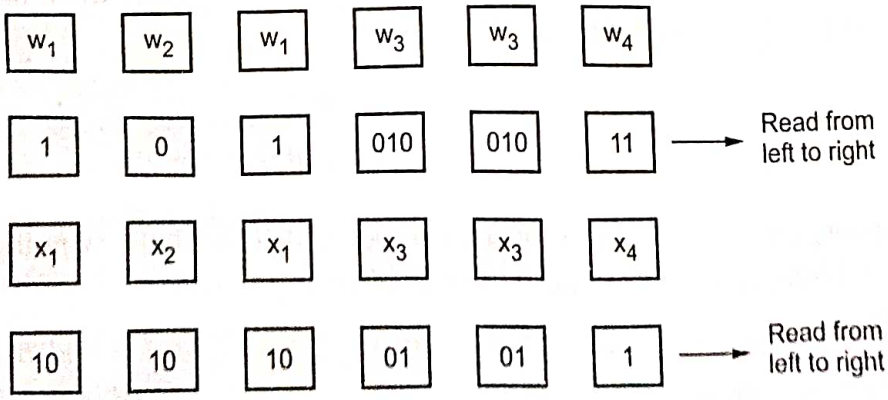


Fig. 6.10.1

Example 6.10.2 *Obtain the solution for the following correspondence system.*

A = {ba, ab, a, baa, b}, B = {bab, baa, ba, a, aba}. The input set {a, b}.

Solution : To obtain the corresponding system the one sequence can be chosen. Hence we get $w_1w_5w_2w_3w_4w_4w_3w_4 = x_1x_5x_2x_3x_4x_4w_3w_4$. This solution gives a unique string bababababaaabaaaabaa = bababababaaabaaaabaa. Hence solution is 15234434.

Example 6.10.3 *Obtain the solution for the following system of posts correspondence problem.*

A = {100, 0, 1}, B = {1, 100, 00}

Solution : The solution is 1 3 1 1 3 2 2. The string is

A1A3A1A1A1A3A2A2 = 100 + 1 + 100 + 100 + 1 + 0 + 0 = 1001100100100

B1B3B1B1B3B2B2 = 1 + 00 + 1 + 1 + 00 + 100 + 100 = 1001100100100

Example 6.10.4 *Obtain the solution for the following system of posts correspondence problem*

A = {ba, abb, bab} B = {bab, bb, abb}.

Solution : Now to consider 1, 3, 2 the string bababab from set A and bababbbb from set B thus the two strings obtained are not equal. As we can try various combinations from both the sets to find the unique sequence but we could not get such a sequence. Hence there is no solution for this system.

Example 6.10.5 *Does PCP with two lists x = (b, bab^3, ba) and y = (b^3, ba, a) have a solution ?*

AU : May-07, Marks 5

Solution : Now we have to find out such a sequence that strings formed by x and y are identical. Such a sequence is 2, 1, 1, 3. Hence from x and y list

2	1	1	3		2	1	1	3
bab^3	b	b	ba	=	ba	b^3	b^3	a

which forms bab^3b^3a. Thus PCP has a solution.

Example 6.10.6 *Explain how post correspondence problem can be treated as a game of dominoes.*

Solution : As in the game of dominoes, the upper half corresponds to some strings and lower half corresponds to some strings.

A_i	Upper half
B_i	Lower half

To win a game, the same string appear in upper and lower half. Winning of a game is equivalent to getting solution for post correspondence problem.

Example 6.10.7 *Define post correspondence problem. Let $\Sigma = \{0, 1\}$. Let A and B be the lists of three strings each, defined as,*

AU : May-14, Marks 6