RGPVNOTES.IN

Program : **B.Tech**

Subject Name: **Theory of Computation**

Subject Code: **CS-501**

Semester: **5th**

Subject Notes
CS-501- Theory of Computation
B. Tech V semester

Turing Machine: Techniques for construction. Universal Turing machine Multi-tape, multi-head and multidimensional Turing machine, N-P complete problems. Decidability and Recursively Enumerable Languages, decidability, decidable languages, undecidable languages, Halting problem of Turing machine & the post correspondence problem.

……

Objective: To develop an overview of how automata theory, languages and computation are applicable in engineering application.

……

Unit-V: Turing Machine

Introduction:

Turing machine is considered as a simple model of a real computer. Turing machines can be used to accept all context-free languages, but also languages such as L = {$a^m b^n c^{mn}$: m ≥ 0, n ≥ 0} which is not class of language comes under regular and context free. Every problem that can be solved on a real computer can also be solved by a Turing machine.

Description of Turing Machine:

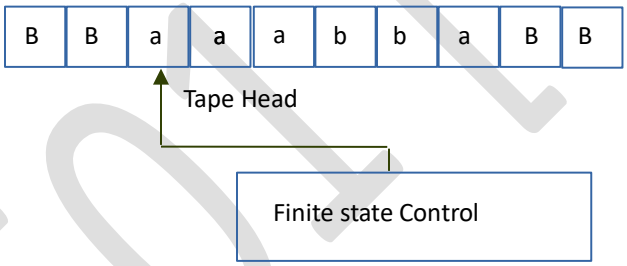


Figure 5.1: Turing Machine

- There are k tapes, for some fixed k ≥ 1 of infinite length. Each tape is divided into cells, each cell stores a symbol belonging to a finite set of tape symbols/ alphabets Γ. B iscalled blank symbol which also belongs to Γ.2. If a cell contains B, then this means that the cell is actually empty. (The given diagram is TM of single tape).
- Each tape has a tape head which can move along the tape, one cell per move. It can also read the cell it currently scans and replace the symbol in this cell by another tape symbol.
- There is a finite state control, which can be in any one of a finite number of states. The finite set of states from Q. The set Q contains three special states: a start (initial) state, an accept state, and a reject state.

Working of Turing Machine:

The Turing machine performs a sequence of computation steps. In one such step, it does the following:

- Immediately before the computation step, the Turing machine is in a state q of Q, and tape heads is on a certain cell.
- Depending on the current state q and the symbol that are read by the tape heads,

❖ The Turing machine switches to a state p of Q (which may be equal to p),
❖ Each tape head writes a symbol of Γ in the cell it is currently scanning (this symbol may be equal to the symbol

currently stored in the cell), and

❖ Each tape head either moves one cell to the left, moves one cell to the right, or stays at the current cell.

Formal Definition of Turing Machine: A Deterministic Turing machine is a 7-tuple
M = (Q,Σ, Γ, δ, q0, B, F) where

- Q is a finite set of states,
- Σ is a finite set of input alphabet; the blank symbol B is not contained in Σ,
- Γ is a finite set of tape alphabet; this alphabet also contains the blank symbol B, and Σ ⊆ Γ,
- δ is called the transition function, which maps: Q × Γ into Q × Γ × D.
- q0 is start state, element of Q
- B is Blank sym δ bol element of Γ,
- F is Set of final states which is subset of Q.

Transitions occurs in Turing Machine:

Transition function of TM is denoted as δ(q, X) = (p, Y, D) which specified transition in TM is function of two components:

- Present state of TM, q
- Tape Symbol X, being scanned by TM.

In every transition

- TM enters into new state p or remain into same state p = q.
- A new symbol Y is written on the scanning cell in place of symbol X.
- If Y = X then there is no change in symbol of scamming cell.
- If D = L or ←, tape head moves one cell left to cell being scanned.
- If D = R or →, tape head moves one cell right to cell being scanned.

Computation by Turing Machine:

Consider TM, T = ({q_1 q_2 q_3 q_4 q_5}, {0,1}, {0,1, b}, δ, q_1, b, {q_5})
Transition Function δ is given by following transition table

Present State	Tape Symbol		
	B	**0**	**1**
→ q_1	(q_2 ,1, L)	(q_1 ,0, R)	-
q_2	(q_3, b, R)	(q_2 ,0, L)	(q_2 ,1, L)
q_3	-	(q_0, b, R)	(q_5, b, R)
q_4	(q_5 ,0, R)	(q_4 ,0, R)	(q_4 ,1, R)
(q_5)	(q_2 ,0, L)	-	-

Table 5.1: Computation by Turing Machine

Computation sequence for input string w = 00

Initial ID: $q_1$00

$q_1$00 |- 0$q_1$0 |- 00q_1 |- 0$q_2$01 |- $q_2$001 |- q_2b001 |- $q_3$001 |- $q_4$01 |- 0$q_4$1 |- 01q_4 |- 010q_5 |- 01$q_2$00 |-*- $q_5$000

Transition Diagram of Turing Machine:

For transition function δ (q, β) = (p, Y, D) The transition diagram will have

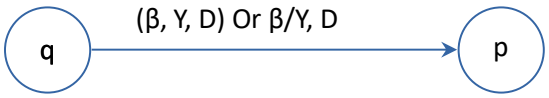


Figure 5.2: Transition Diagram

Acceptance of Language by Turing Machine:

A TM accepts a language if it enters into a final state for any input string w. A language is recursively enumerable (generated by Type-0 grammar) if it is accepted by a Turing machine.

A TM decides a language if it accepts it and enters into a rejecting state for any input not in the language. A language is recursive if it is decided by a Turing machine. There may be some cases where a TM does not stop. Such TM accepts the language, but it does not decide it.

Example 1:

Design a TM to recognize all strings consisting of an odd number of α's.

Solution:

- The Turing machine M can be constructed by the following moves:
- Let q1 be the initial state.
- If M is in q1; on scanning α, it enters the state q2 and writes B (blank).
- If M is in q2; on scanning α, it enters the state q1 and writes B (blank).

From the above moves, we can see that M enters the state q1 if it scans an even number of α's, and it enters the state q2 if it scans an odd number of α's. Hence q2 is the only accepting state.

Hence,

M = {{q1, q2}, {1}, {1, B}, δ, q1, B, {q2}} where δ is given by:

Tape Alphabet Symbol	Present State "q1"	Present State "q2"
a	BRq1	BRq2

Table 5.2: Example of Turing Machine

Example 2:

Design a Turing Machine that reads a string representing a binary number and erases all leading 0's in the string. However, if the string comprises of only 0's, it keeps one 0.

Solution:

Let us assume that the input string is terminated by a blank symbol, B, at each end of the string.

The Turing Machine, M, can be constructed by the following moves:

- Let q0 be the initial state.
- If M is in q0, on reading 0, it moves right, enters the state q1 and erases 0. On reading 1, it enters the state q2 and moves right.
- If M is in q1, on reading 0, it moves right and erases 0, i.e., it replaces 0's by B's. On reaching the leftmost 1, it enters q2 and moves right. If it reaches B, i.e., the string comprises of only 0's, it moves left and enters the state q3.

- If M is in q2, on reading either 0 or 1, it moves right. On reaching B, it moves left and enters the state q4. This validates that the string comprises only of 0's and 1's.
- If M is in q3, it replaces B by 0, moves left and reaches the final state qi.
- If M is in q4, on reading either 0 or 1, it moves left. On reaching the beginning of the string, i.e., when it reads B, it reaches the final state qf.

Hence, M = {{q0, q1, q2, q3, q4, qf}, {0,1, B}, {1, B}, δ, q0, B, {qf}} where δ is given by:

Tape Alphabet Symbol	Present State "q0"	Present State "q1"	Present State "q2"	Present State "q3"	Present State "q4"
0	BRq1	BRq1	0Rq2	-	0Lq4
1	1Rq2	1Rq2	1Rq2	-	1Lq4
B	BRq1	BLq3	BLq4	0Lqf	BRqf

Table 5.3: Example of Turing Machine

Techniques of Construction:

Multi-tape Turing Machine:

Multi-tape Turing Machines have multiple tapes where each tape is accessed with a separate head. Each head can move independently of the other heads. Initially the input is on tape 1 and others are blank. At first, the first tape is occupied by the input and the other tapes are kept blank. Next, the machine reads consecutive symbols under its heads and the TM prints a symbol on each tape and moves its heads.

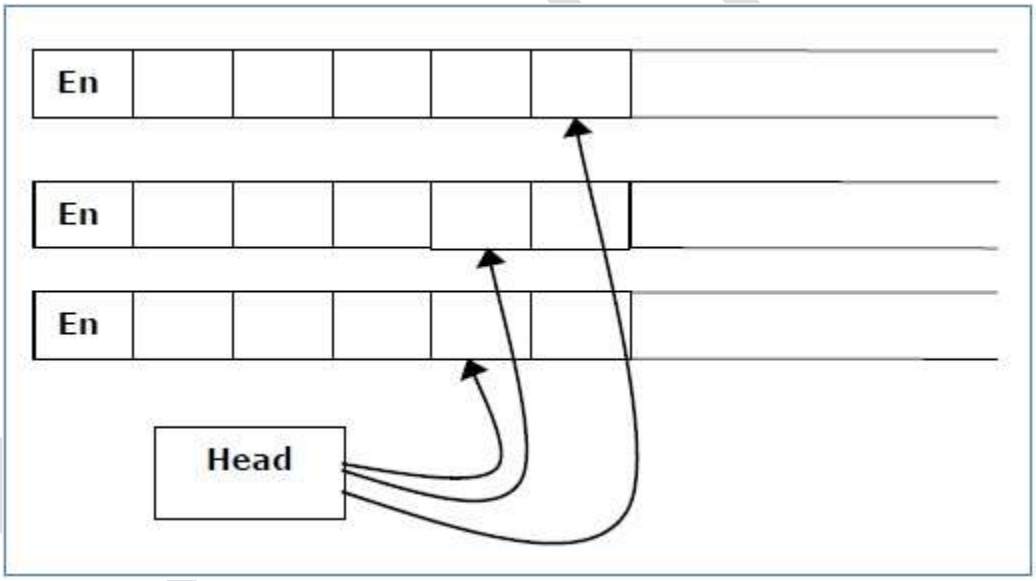


Figure 5.3: Multi-tape Turing Machine

A Multi-tape Turing machine can be formally described as a 7-tuple M = (Q,Σ, Γ, δ, q0, B, F) where −

- Q is a finite set of states
- Σ is set of input alphabet
- Γ is the tape alphabet
- B is the blank symbol
- δ is a relation on states and symbols where $\delta: Q \times X^k \rightarrow Q \times (X \times \{\text{Left shift, Right shift, OnShift}\})\ k$ where there is **k** number of tapes
- q0 is the initial state
- F is the set of final states

Note: Every Multi-tape Turing machine has an equivalent single-tape Turing machine.

Non-Deterministic Turing Machine:

In a NDTM, for every state and symbol, there are a group of actions the TM can have. So, here the transitions are not deterministic. The computation of a non-deterministic Turing Machine is a tree of configurations that can be reached from the start configuration.

An input is accepted if there is at least one node of the tree which is an accept configuration, otherwise it is not accepted. If all branches of the computational tree halt on all inputs, the non-deterministic Turing Machine is called a **Decider** and if for some input, all branches are rejected, the input is also rejected.

A non-deterministic Turing machine can be formally defined as a 7-tuple (Q, ∑, Γ, δ, q0, B, F) where −

- Q is a finite set of states
- Γ is the tape alphabet
- ∑ is the input alphabet
- δ is a transition function; δ: Q × X → P (Q × X × {Left shift, Right shift}).
- q0 is the initial state
- B is the blank symbol
- F is the set of final states

Universal Turing Machine:

A universal Turing machine (UTM) is a Turing machine that can simulate an arbitrary Turing machine on arbitrary input. The universal machine essentially achieves this by reading both the description of the machine to be simulated as well as the input thereof from its own tape.

Linear Bounded Automata:

A linear bounded automaton is a multi-track non-deterministic Turing machine with a tape of some bounded finite length.

Length = function (Length of the initial input string, constant c)

Here,

Memory information ≤ c × Input information

The computation is restricted to the constant bounded area. The input alphabet contains two special symbols which serve as left end markers and right end markers which mean the transitions neither move to the left of the left end marker nor to the right of the right end marker of the tape.

A linear bounded automaton can be defined as an 8-tuple (Q, X, Σ, q0, ML, MR, δ, F) where:

- Q is a finite set of states
- X is the tape alphabet
- Σ is the input alphabet
- q0 is the initial state
- ML is the left end marker

- MR is the right end marker where MR≠ ML
- δ is a transition function which maps each pair (state, tape symbol) to (state, tape symbol, Constant 'c') where c can be 0 or +1 or -1
- F is the set of final states

A deterministic linear bounded automaton is always context-sensitive and the linear bounded automaton with empty language is undecidable.

Offline Turing Machine:

An Offline Turing Machine has two types:

- One tape is read only and contains the input.
- The other is read-write and is initially blank.

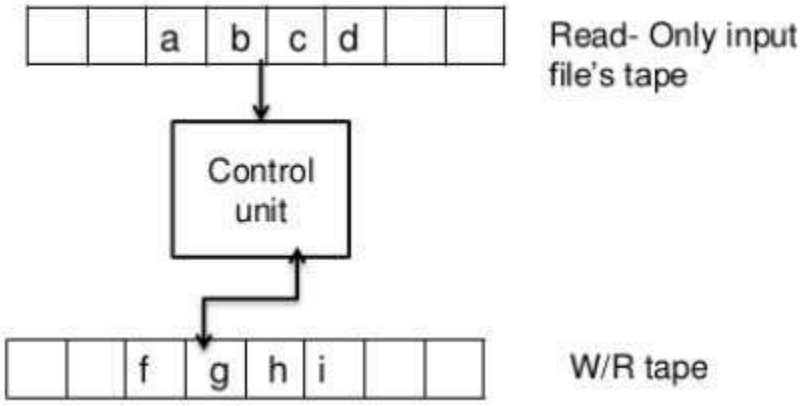


Figure 5.4: Offline Turing Machine

A standard Turing Machine is simulated by Offline Turing Machine and an Offline Turing Machine simulated by standard Turing Machine.

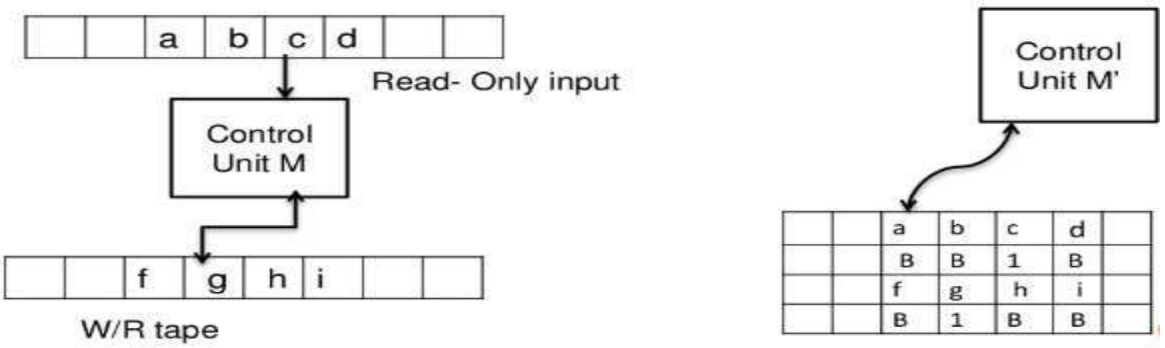


Figure 5.5: Offline Turing Machine

Equivalence of Single Tape & Multi Tape Turing Machine

In the classical framework k-tape Turing machines have the same computational power of Single-tape Turing machines and given a Multi-tape Turing Machine M_K it is always possible to define a Single-tape Turing Machine which is able to fully simulate its behavior and therefore to completely execute its computations. The Gross-one

methodology allows us to give a more accurate definition of the equivalence among different machines as it provides the possibility not only to separate different classes of infinite sets with respect to their cardinalities but also to measure the number of elements of some of them. With reference to Multi-tape Turing machines, the Single-tape Turing Machines adopted for their simulation use a particular kind of tape which is divided into tracks (multitrack tape). In this way, if the tape has m tracks, the head is able to access (for reading and/or writing) all the m characters on the tracks during a single operation. This tape organization leads to a straightforward definition of the behavior of a Single-tape Turing machine able to completely execute the computations of a given Multi-tape Turing machine

Recursive & Recursively Enumerable Language:

A Turing Machine may

- Halt and accept the input
- Halt and reject the input, or
- Never halt/loop

Recursive Enumerable or Type-0 Language: RE languages or type-0 languages are generated by type-0 grammars. A RE language can be accepted or recognized by Turing machine which means it will enter into final state for the strings of language and may or may not enter into rejecting state for the strings which are not part of the language. It means TM can loop forever for the strings which are not a part of the language. RE languages are also called as Turing recognizable languages.

Recursive Language: A recursive language (subset of RE) can be decided by Turing machine which means it will enter into final state for the strings of language and rejecting state for the strings which are not part of the language. e.g.; L= $\{a^n b^n c^n | n>=1\}$ is recursive because we can construct a Turing machine which will move to final state if the string is of the form $a^n b^n c^n$ else move to non-final state. So, the TM will always halt in this case. REC languages are also called as Turing decidable languages.

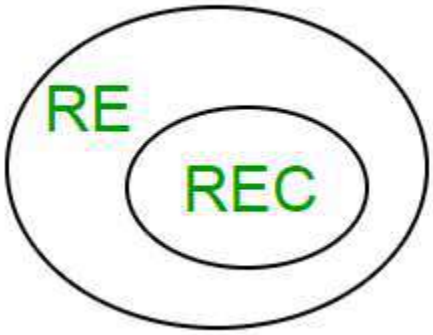


Figure 5.6: Relationship between RE & REC

Language Decidability:

A language is called Decidable or Recursive if there is a Turing machine which accepts and halts on every input string w. Every decidable language is Turing-Acceptable.

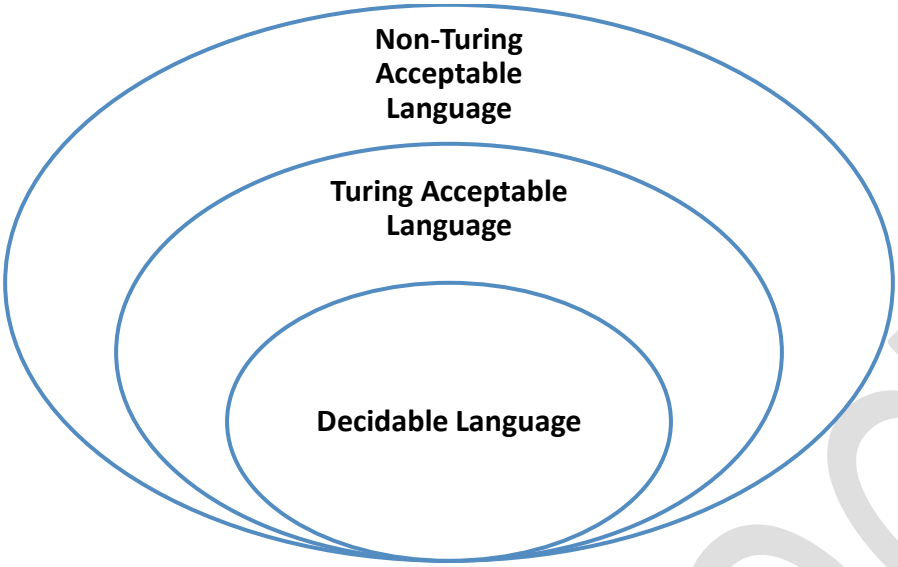


Figure 5.7: Language Decidability

A decision problem P is decidable if the language L of all yes instances to P is decidable.

For a decidable language, for each input string, the TM halts either at the accept or the reject state as depicted in the following diagram:

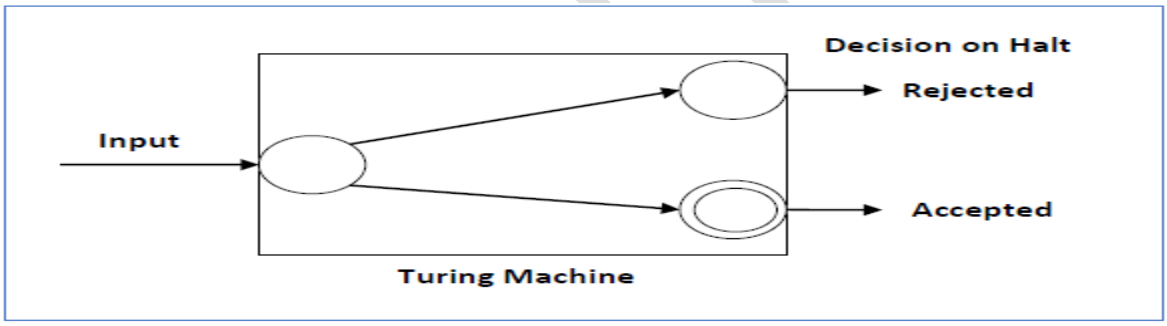


Figure 5.8: Decidable Language

Example 1: Find out whether the following problem is decidable or not: Is a number 'm' prime?

Solution:

Prime numbers = {2, 3, 5, 7, 11, 13, ……}

Divide the number 'm' by all the numbers between '2' and '√m' starting from '2'. If any of these numbers produce a remainder zero, then it goes to the "Rejected state", otherwise it goes to the "Accepted state". So, here the answer could be made by 'Yes' or 'No'.

Hence, it is a decidable problem.

Example 2: Given a regular language L and string w, how can we check if w∈ L?

Solution:

Take the DFA that accepts L and check if w is accepted

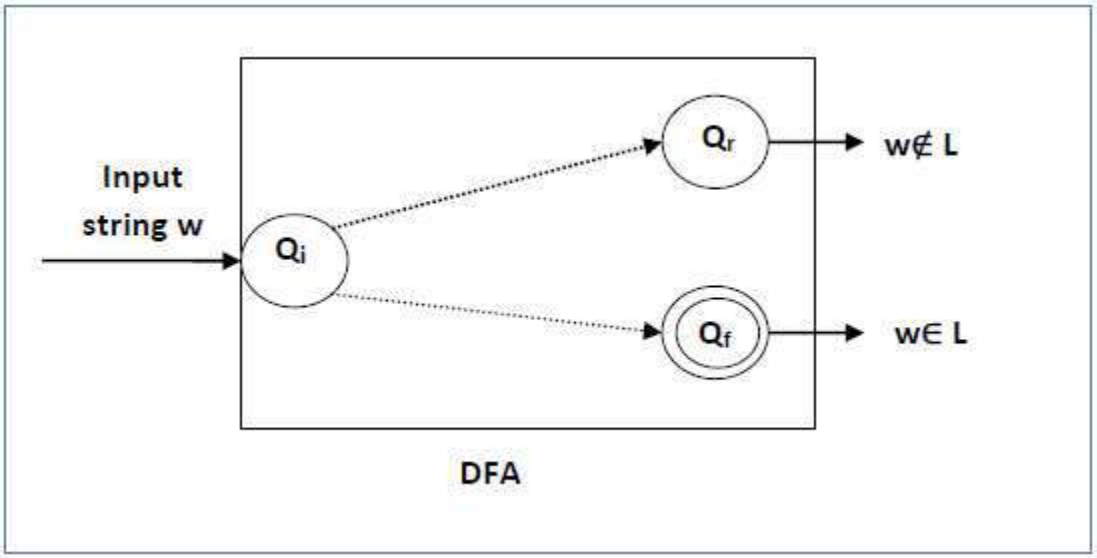


Figure 5.9: Example of Language Decidable

Note:

1. If a language L is decidable, then its complement L' is also decidable.
2. If a language is decidable, then there is an enumerator for it.

Undecidable Language:

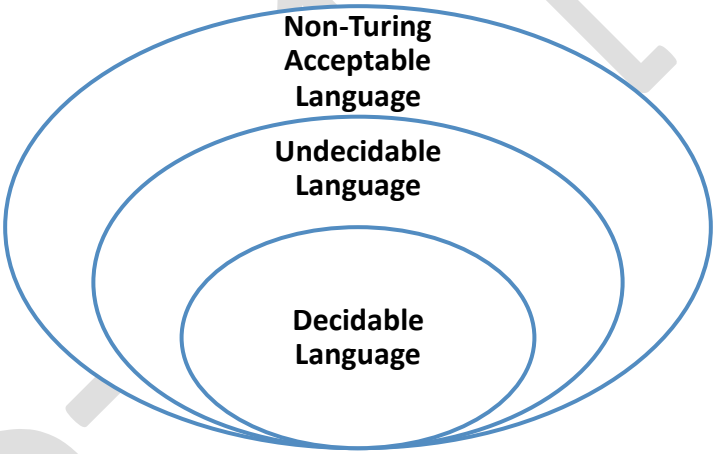


Figure 5.10: Undecidable Language

For an undecidable language, there is no Turing Machine which accepts the language and makes a decision for every input string w (TM can make decision for some input string though). A decision problem P is called "undecidable" if the language L of all yes instances to P is not decidable. Undecidable languages are not recursive languages, but sometimes, they may be recursively enumerable languages.

Example:

- The halting problem of Turing machine
- The mortality problem
- The mortal matrix problem
- The Post correspondence problem, etc.

Turing Machine Halting Problem:
Input: A Turing machine and an input string w.

Problem: Does the Turing machine finish computing of the string w in a finite number of steps? The answer must be either yes or no.

Proof: At first, we will assume that such a Turing machine exists to solve this problem and then we will show it is contradicting itself. We will call this Turing machine as a Halting machine that produces a 'yes' or 'no' in a finite amount of time. If the halting machine finishes in a finite amount of time, the output comes as 'yes', otherwise as 'no'. The following is the block diagram of a Halting machine:

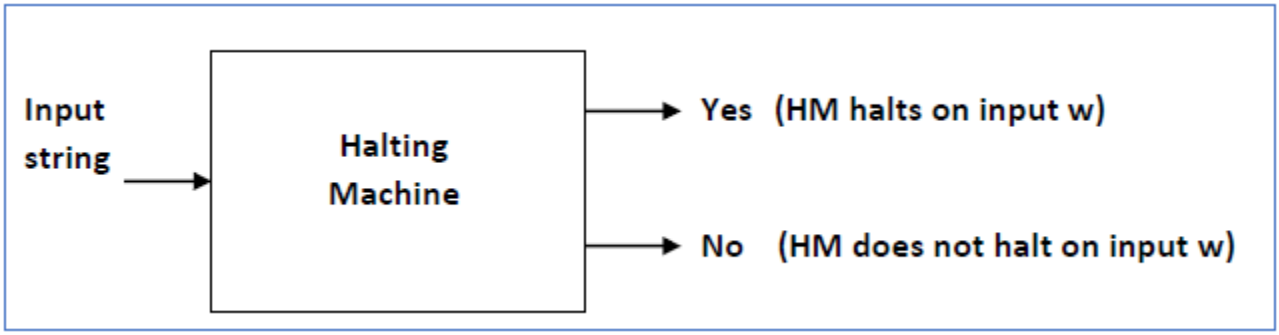


Figure 5.11: Turing Machine Halting Problem

Now we will design an inverted halting machine (HM)' as:

- If H returns YES, then loop forever.
- If H returns NO, then halt.

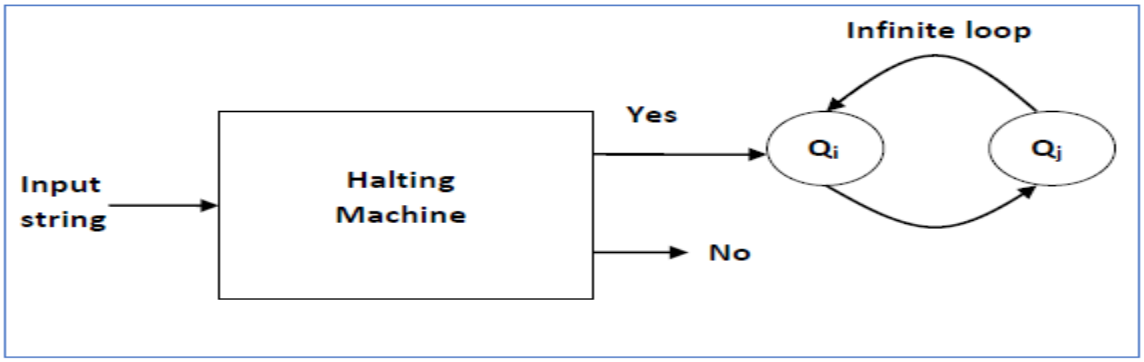


Figure 5.12: Inverted Halting Machine

Further, a machine (HM)2 which input itself is constructed as follows:

- If (HM)2 halts on input, loop forever.
- Else, halt.

Here, we have got a contradiction. Hence, the halting problem is undecidable

Introduction of P, NP, NP Complete & NP Hard Problem:

P & NP Problems:
If a problem can be solved in polynomial time, it is said to belong to the **P** class of problems. P-type problems

are tractable.
For some intractable problems, you can verify that the solution is correct using a P-type algorithm. For example, you can verify that a given solution to the TSP visits every city. These problems are referred to as Non-deterministic Polynomial problems or NP-type problems. The challenge for programmers is to find a P-type solution to NP-type problems.

P Problems:
As the name says these problems can be solved in polynomial time, i.e.; O(n), O(n^2) or O(nk) where k is a constant.

NP Problems:
Some people think NP as Non-Polynomial. But actually, it is Non-deterministic Polynomial time. i.e.; "yes" instances of these problems can be solved in polynomial time by a non-deterministic Turing machine and hence can take up to exponential time (some problems can be solved in sub-exponential but super polynomial time) by a deterministic Turing machine. In other words, these problems can be verified (if a solution is given, say if it is correct or wrong) in polynomial time. Examples include all P problems. One example of a problem not in P but in NP is Integer Factorization.

NP Complete Problems (NPC):
Over the years many problems in NP have been proved to be in P (like Primarily Testing). Still, there are many problems in NP not proved to be in P. i.e.; the question still remains whether P=NP (i.e.; whether all NP problems are actually P problems).
NP Complete Problems helps in solving the above question. They are a subset of NP problems with the property that all other NP problems can be reduced to any of them in polynomial time. So, they are the hardest problems in NP, in terms of running time. If it can be showed that any NPC Problem is in P, then all problems in NP will be in P (because of NPC definition), and hence P=NP=NPC.
All NPC problems are in NP (again, due to NPC definition). Examples of NPC problems

NP Hard Problems (NPH):
These problems need not have any bound on their running time. If any NPC Problem is polynomial time reducible to a problem X, that problem X belongs to NP Hard class. Hence, all NP Complete problems are also NPH. In other words, if a NPH problem is non-deterministic polynomial time solvable, it is a NPC problem. Example of a NP problem that is not NPC is Halting Problem.

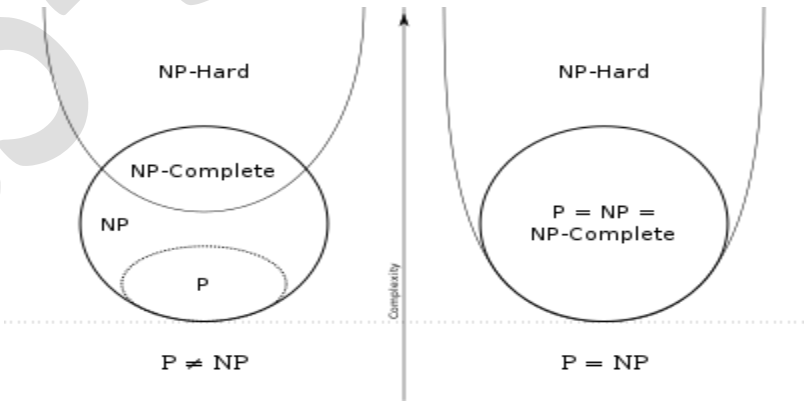


Figure 5.13: NP complete problem

From the figure 5.10, it's clear that NPC problems are the hardest problems in NP while being the simplest ones in NPH. i.e.; NP∩NPH=NPC

Given a general problem, we can say it's in NPC, if and only if we can reduce it to some NP problem (which shows it is in NP) and also some NPC problem can be reduced to it (which shows all NP problems can be reduced to this problem).
Also, if a NPH problem is in NP, then it is NPC

Examples of NP Problems:
Boolean Satisfiability Problem:
Boolean Satisfiability or simply SAT is the problem of determining if a Boolean formula is satisfiable or unsatisfiable.
Satisfiable: If the Boolean variables can be assigned values such that the formula turns out to be TRUE, then we say that the formula is satisfiable.
Unsatisfiable: If it is not possible to assign such values, then we say that the formula is unsatisfiable.

Examples (as shown in table 5.4):
F= A^B', is satisfiable, because A = TRUE and B = FALSE makes F = TRUE.
G=A^A', is unsatisfiable, because:

A	A'	G
True	false	False
False	True	False

Table 5.4: Example of SAT

Boolean satisfiability problem is NP-complete (This was proved by Cook's Theorem).

2-SAT Problem:

- 2-SAT is a special case of Boolean Satisfiability Problem and can be solved in polynomial time.
- To understand this better, first let us see what is Conjunctive Normal Form (CNF) or also known as Product of Sums (POS).

CNF: CNF is a conjunction (AND) of clauses, where every clause is a disjunction (OR).
$F = (A_1 \vee B_1) \wedge (A_2 \vee B_2) \wedge (A_3 \vee B_3) \ldots\ldots\ldots\ldots\ldots\ldots (A_m \vee B_m)$

- Now, 2-SAT limits the problem of SAT to only those Boolean formula which are expressed as a CNF with every clause having only **2 terms** (also called **2-CNF**)
- For 2-SAT problem the CNF value is TRUE, if value of every clause is TRUE. Let one of the clauses be (A V B) so we can say (A V B) = TRUE in following two conditions
  - If A = 0, B must be 1 i.e. $(A' \Rightarrow B)$
  - If B = 0, A must be 1 i.e. $(B' \Rightarrow A)$

  Thus (A V B) is true equivalent to (A' => B) ^ (B' => A)

Vertex Cover Problem:
A vertex cover of an undirected graph is a subset of its vertices such that for every edge (u, v) of the graph, either 'u' or 'v' is in vertex cover. Although the name is Vertex Cover, the set covers all edges of the given graph. Given an undirected graph, the vertex cover problem is to find minimum size vertex cover.
Following are some examples (as shown in fig 5.11): -

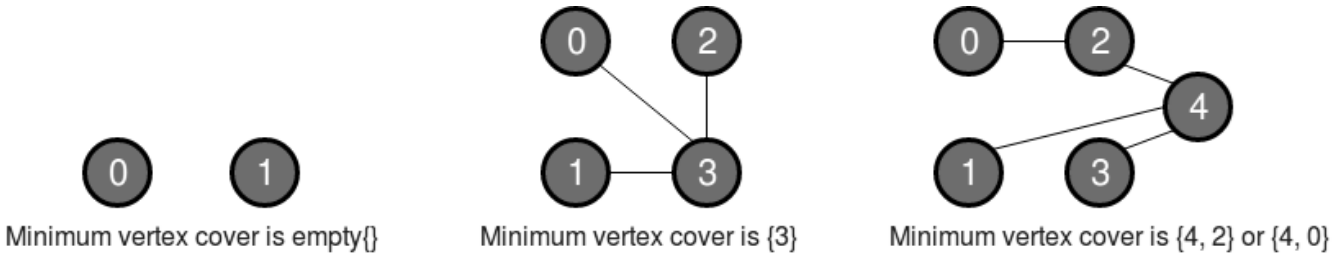


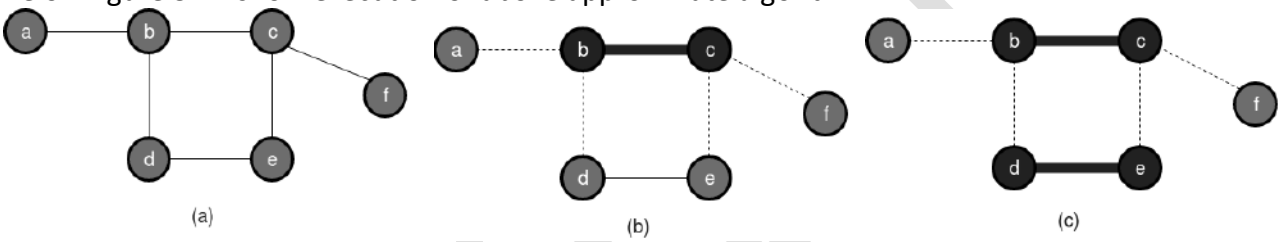
Figure 5.14: Examples for vertex cover problem

Vertex Cover Problem is a known NP Complete problem, i.e., there is no polynomial time solution for this unless P = NP. Although There can be an approximate polynomial time algorithm to solve the problem. Following is a simple approximate algorithm.

Approximate Algorithm for Vertex Cover:

1) Initialize the result as {}
2) consider a set of all edges in given graph. Let the set be E.
3) Do following while E is not empty
    a) Pick an arbitrary edge (u, v) from set E and add 'u' and 'v' to result
    b) Remove all edges from E which are either incident on u or v.
4) Return result

Below figure 5.12 show execution of above approximate algorithm:



Minimum vertex cover is {b,c,d} or {b,c,e}

Figure 5.15: Execution steps of vertex cover problem

Hamiltonian Cycle Problem:

A Hamiltonian cycle is a cycle in a graph that visits each vertex exactly once. To show Hamiltonian Cycle Problem is NP-complete, we first need to show that it actually belongs to the class NP, and then use a known NP-complete problem to Hamiltonian Cycle.

So does Hamiltonian Cycle Problem ∈ NP?

Given: $Graph\ G = (V, E)$

Certificate: List of vertices on Hamiltonian Cycle

To check if this list is actually a solution to the Hamiltonian cycle problem, one counts the vertices to make sure they are all there, and then checks that each is connected to the next by an edge, and that the last is connected to the first. It takes time proportional to n, because there are n vertices to count and n edges to check. n is a polynomial, so the check runs in polynomial time.

Therefore, Hamiltonian Cycle ∈ NP.

Prove Hamiltonian Cycle Problem ∈ NP-Complete

Reduction: Vertex Cover to Hamiltonian Cycle

Definition: Vertex cover is set of vertices that touch all edges in the graph.

Given a $graph\ G$ and integer k, construct a $graph\ G'$ such that G has a vertex cover of size k if G' has a $Hamiltonian\ cycle$.

Idea: To construct widget for each edge in the graph.
$i.e \forall uv$ in the $Graph\ G$, create a widget shown below: -

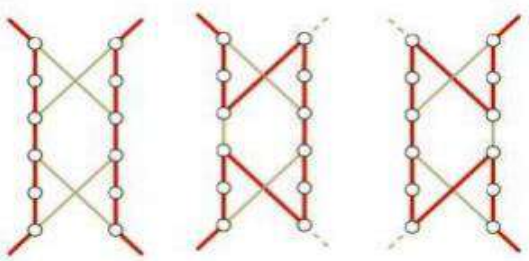


Figure 5.16: An Example for Hamiltonian Cycle Graph

As shown in figure 5.13, there are three ways to traverse a widget; 1. Enter from u, go somewhere else in the graph, and then come back through the other side i.e. v 2. Enter and Exit through u 3. Enter and Exit through v Construct G' for G (Vertex cover) of size k = 2 with the construction, any graph with a vertex cover, can be used to make a graph with a Hamiltonian Cycle graph. Since creating such a graph can be done under polynomial time, simply replace edges with widgets and make proper connections, we have a reduction from Vertex Cover to Hamiltonian Cycle. This means that finding whether a graph has a Hamiltonian Cycle or not is NP Hard. As we have seen earlier it's also in NP, therefore, Hamiltonian Cycle is an NP Complete Problem.

Traveling Salesman Problem:

The traveling salesman problem consists of a salesman and a set of cities. The salesman has to visit each one of the cities starting from a certain one (e.g. the hometown) and returning to the same city. The challenge of the problem is that the traveling salesman wants to minimize the total length of the trip.
The traveling salesman problem can be described as follows:
TSP = {(G, f, t): G = (V, E) a complete graph that contains a traveling salesman tour with cost that does not exceed t.}
f is a function V×V → Z, t ∈ Z,
Example: Consider the following set of cities as shown in figure 5.14:

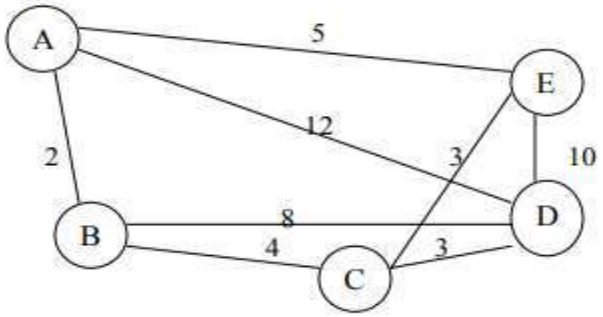


Figure 5.17: An example of TSP

The problem lies in finding a minimal path passing from all vertices once. For example, the path Path1 {A, B, C, D, E, A} and the path Path2 {A, B, C, E, D, A} pass all the vertices but Path1 has a total length of 24 and Path2 has a total length of 31.

Theorem: The traveling salesman problem is NP-complete.

Proof:

First, we have to prove that TSP belongs to NP. If we want to check a tour for credibility, we check that the tour contains each vertex once. Then we sum the total cost of the edges and finally we check if the cost is minimum. This can be completed in polynomial time thus TSP belongs to NP.

Secondly, we prove that TSP is NP-hard. One way to prove this is to show that Hamiltonian cycle <= TSP (given that the Hamiltonian cycle problem is NP-complete). Assume G = (V, E) to be an instance of Hamiltonian cycle. An instance of TSP is then constructed. We create the complete graph = (V, ≤ P G' E') where E' = {(I, j): i, j ∈ V and i ≠ j}. Thus, the cost function is defined as:

$$t(i,j) = \begin{cases} 0 \text{ if } (i, j) \in E, \\ 1 \text{ if } (i, j) \notin E. \end{cases}$$

Now suppose that a Hamiltonian cycle h exists in G. It is clear that the cost of each edge in h is 0 in G' as each edge belongs to E. Therefore, h has a cost of 0 in G'. Thus, if graph G has a Hamiltonian cycle then graph G' has a tour of 0 costs.

Conversely, we assume that G' has a tour h' of cost at most 0. The cost of edges in E' are 0 and 1 by definition. So, each edge must have a cost of 0 as the cost of h' is 0. We conclude that h' contains only edges in E.

So, we have proven that G has a Hamiltonian cycle if and only if G' has a tour of cost at most 0. Thus, TSP is NP-complete.

RGPVNOTES.IN

We hope you find these notes useful.

You can get previous year question papers at https://qp.rgpvnotes.in .

If you have any queries or you want to submit your study notes please write us at rgpvnotes.in@gmail.com

LIKE & FOLLOW US ON FACEBOOK
facebook.com/rgpvnotes.in