Unit-7
Undecidability and Intractability:-

⊛ Computational Complexity:-

The complexity of computational problems can be discussed by choosing a specific abstract machine as a model of computation and considering how much resource machine of that type require for the solution of that problem. Complexity Measure is a means of measuring the resource used during a computation. In case of Turing Machines, during any computation, various resources will be used, such as space and time. When estimating these resources, we are always interested in growth rates than absolute values.

A problem is regarded as inherently difficult if solving the problem requires a large amount of resources, whatever the algorithm used for solving it. The time and storage are the main complexity measures to be used. Other complexity measures are also used, such as the amount of communication, the number of gates in a circuit and the number of processors.

⊛ Time and Space Complexity of a Turing Machine:

When a turing machine answers a specific instance of a decision problem we can measure time as number of moves and the space as number of tape squares, required by the computation. The most obvious measure of the size of any instance is the length of input string. The worst case is considered as the maximum time or space that might be required by any string of that length.

The time and space complexity of a TM can be defined as;

Let T be a TM. The time complexity of T is the function Tt defined on the natural numbers as; for $n \in N$, $Tt(n)$ is the maximum number of moves T can make on any input string of length n. If there is an input string x such that for $|x|=n$, T loops forever on input T, $Tt(n)$ is undefined.

The space complexity of T is the function St defined as $St(n)$ is the maximum number of the tape squares used by T for any input string of length n. If T is multi-tape TM, number of tape squares means maximum of the number of individual tapes. If for some input of length n, it causes T to loop forever, $St(n)$ is undefined.

⊛. Intractability:-

An algorithm for which the complexity measures $S_t(n)$ increases with n, no more rapidly than a polynomial in n is said to be polynomially bounded; & one in which it grows exponentially is said to be exponentially bounded. Intractability is a technique for solving problems not to be solvable in polynomial time. The problems that can be solved within reasonable time and space constraints are called tractable. The problems that cannot be solved in polynomial time but requires exponential time algorithm are called intractable or hard problems.

To introduce intractability theory, the class P and class NP of problems solvable in polynomial time by deterministic and non-deterministic TM's are essential. When the space and time required for implementing the steps of particular algorithm are reasonable, we can say that the problem is tractable. Problems are intractable if the time required for any of the algorithm is at least $f(n)$, where f is an exponential function of n.

⊛. Complexity Classes:-

In computational complexity theory, a <u>complexity class</u> is a set of problems of related resource-based complexity. A typical complexity class has a definition of the form: "The set of problems that can be solved by an abstract machine M using $O(f(n))$ of resource R, where n is the size of the input." The simpler complexity classes are defined by the following factors;

- <u>The type of computational problem:</u> The most commonly used problems are decision problems. However, complexity classes can be defined based on function problems, optimization problems etc.

<u>The model of computation:</u> The most common model of computation is the deterministic Turing machine, but many complexity classes are based on non-deterministic Turing machines, Boolean circuits etc.

<u>The resources that are being bounded and the bounds:</u> These two properties are usually stated together, such as "polynomial time", "logarithmic space", "constant depth" etc.

Class P: The class P is the set of problems that can be solved by deterministic TM in polynomial time. A language L is in class P if there is some polynomial time complexity T(n) such that L=L(M), for some deterministic Turing Machine M of time complexity T(n).

Class NP: The class NP is the set of problems that can be solved by non-deterministic TM in polynomial time. A language L is in the class NP if there is a non-deterministic TM, M, and a polynomial time complexity T(n), such that L=L(M) and when M is given an input of length n, there are no sequences of more than T(n) moves of M.

NP-Complete: The complexity class NP-complete (NP-C or NPC) is a class of problems having two properties,

→ It is the set of NP (nondeterministic polynomial time) problems: Any given solution to the problem can be verified quickly (in polynomial time).

→ It is also in the set of NP-hard problems: Any NP problem can be converted into this one by a transformation of the inputs in polynomial time.

Properties of NP-Complete problems:-

→ No polynomial time algorithms has been found for any of them.

→ It is not established that polynomial time algorithm for these problems do not exist.

→ If polynomial time algorithm is found for any of them, there will be polynomial time algorithm for all of them.

→ If it can be proved that no polynomial time algorithm exists for any of them, then it will not exist for every one of them.

⊛. Problems and it's types:

i) **Abstract Problems:** Abstract problem A is binary relation on set I of problem instances, and the set S of problem solutions.
For e.g. Minimum spanning tree of a graph G can be viewed as a pair of the given graph G and MST graph T.

ii) **Decision Problems:** Decision problem D is a problem that has an answer as either "true", "yes", "1" or "false", "no", "0". For e.g. if we have the abstract shortest path with instances of the problem and the solution set as {0,1}, then we can transform that abstract problem by reformulating the problem as "Is there a path from u to v with at most k edges." In this situation the answer is either yes or no.

iii) **Optimization Problems:** We encounter many problems where there are many feasible solutions and our aim is to find the feasible solution with the best value. This kind of problem is called optmization problem. For e.g. given the graph G, and the vertices u and v find the shortest path from u to v with minimum number of edges. The NP completeness does not directly deal with optimizations problems; however we can translate the optimization problem to the decision problem.

iv) **Function Problems:** A function problem is a computational problem where a single output is expected for every input, but the output is more complex than that of a decision problem, which isn't just Yes or No. For e.g, The travelling salesman problem, which asks for the route taken by the salesman, and the Inter Factorization problem, which asks for the list of factors.

⊛. Reducibility:-

Reducibility is a way of converging one problem into another in such a way that, a solution to the second problem can be used to solve the first one.

Many complexity classes are defined using the concept of a reduction. A reduction is a transformation of one problem into another problem. There are many different type of reductions based on the method of reduction, such as Cook reductions, Karp reductions and Levin reductions.

The most commonly used reduction is a polynomial-time reduction. This means that the reduction process takes polynomial time. For example, the problem of squaring an integer can be reduced to the problem of multiplying two integers. For complexity classes larger than P, polynomial-time reductions are commonly used.

⊛. Circuit Satisfiability:-

The circuit satisfiability problem (also known as CircutSAT, CSAT etc.) is the decision problem of determining whether a given Boolean circuit has an assignment of its inputs that makes the output true. In other words, it asks whether the inputs to a given Boolean circuit can be consistently set to 1 or 0 such that the circuit outputs 1. If that is the case, the circuit is called satisfiable. Otherwise, the circuit is called unsatisfiable.

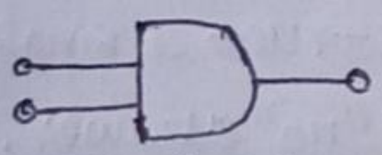


fig: satisfiable circuit.

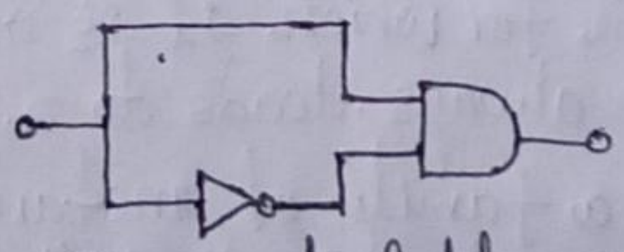


Fig: unsatisfiable circuit.

Cook's Theorem:

Lemma: SAT is NP-hard

Proof: Take a problem $V \in NP$, let A be the algorithm that verifies V in polynomial time (this must be true since $V \in NP$). We can program A on a computer and therefore there exists a (huge) logical circuit whose input wires correspond to the bits of the inputs x and y of A and which outputs 1 precisely when A(x,y) returns yes:

For any instance x of V let Ax be the circuit obtained from A by setting the x-input wire values according to the specific string x. The construction of Ax from x is our reduction function. If x is a yes instance of V, then the certificate y for x gives satisfying assignments for Ax. Conversely, if Ax outputs 1 for some assignments to its input wires, that assignment translates into a certificate for x.

Theorem: SAT is NP-complete

Proof: To show that SAT is NP-complete we have to show two properties as given by the definition of NP-complete problems. The first property i.e, SAT is NP & the second property i.e, SAT is NP-hard.

Circuit satisfiability problem (SAT) is the question "Given a Boolean combinational circuit, is it satisfiable?" Given the circuit satisfiability problem take a circuit x and a certificate y with the set of values that produce output 1, we can verify that whether the given certificate satisfies the circuit in polynomial time. So we can say that circuit satisfiability problem is NP.

This claims that SAT is NP. Now it is sufficient to show the second property holds for SAT. The proof for the second property i.e, SAT is NP-hard is from above lemma. This completes the proof.

⑦. Undecidability:-

In computational theory, an undecidable problem is a decision problem for which it is impossible to construct a single algorithm that always leads to a correct "yes" or "no" answer. It consists of a family of instances for which a particular yes/no answer is required, such that there is no computer program that, for any given problem instance as input, terminates and outputs the required answer after a finite number of steps. More formally, an undecidable problem is a problem whose language is not a recursive set or computable or decidable.

Undecidable Problems:

1. Posts Correspondance Problem (PCP):

The input of the problem consists of two finite lists $U = \{u_1, u_2, \ldots, u_n\}$ and $V = \{v_1, v_2, \ldots, v_n\}$ of words over the alphabet Σ having at least two symbols. A solution to this problem is a sequence of indices i_k; $1 <= k <= n$, for all k, such that

$$u_{i1} u_{i2} \ldots\ldots u_{ik} = v_{i1} v_{i2} \ldots\ldots v_{ik}$$

We say $i_1, i_2, \ldots\ldots, i_k$ is a solution to this instance of PCP.
Here, the decision problem is to decide whether such a solution exists or not.

Example: Consider the following two lists:

U	
u_1	a
u_2	ab
u_3	bba

V	
v_1	baa
v_2	aa
v_3	bb

A solution to this problem would be the sequence (3,2,3,1), because

$u_3 u_2 u_3 u_1 = bba + ab + bba + a = bbaabbbaa$

$v_3 v_2 v_3 v_1 = bb + aa + bb + baa = bbaabbbaa.$

Furthermore, since (3,2,3,1) is a solution, so all of it's "repetitions", such as (3,2,3,1,3,2,3,1).etc. are infinitely many solutions of this repetitive kind.
However, if the two lists had consisted of only u_2, u_3 and v_2, v_3 then, there would have been no solution.

2. Halting problem and its proof:

"Given a Turing Machine M and an input w, do M halts on w?"
Algorithms may contain loops which may be infinite or finite in length. The amount of work done in an algorithm usually depends on data input. Algorithms may consist of various number of loops nested or in sequence. Thus, the halting problem asks the question; "Given a program and an input to the program, determine if the program will eventually stop when it is given that input." The question is simply whether the given program will ever halt on a particular input.

Trial Solution: Just run the program with the given input. If the program stops we know that the program halts. But if the program does not stop in reasonable amount of time, we cannot conclude that it won't stop. May be we did not wait long enough!

The halting problem is famous because it was one of the first problems proven algorithmically undecidable. This means there is no algorithm which can be applied to any arbitrary program and input to decide whether the program stops when run with that input.

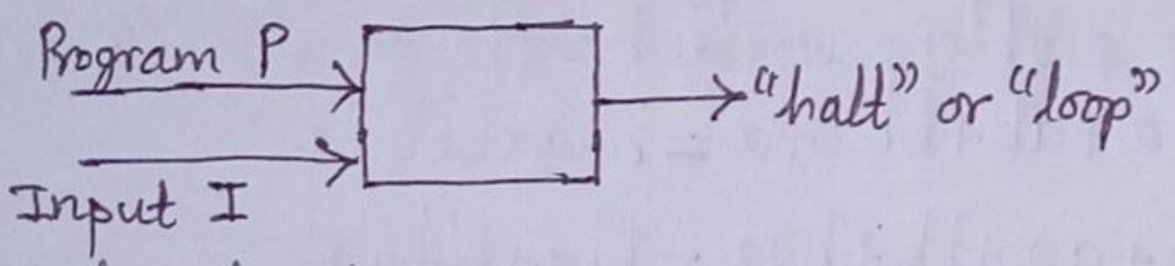
Sketch of a proof that the Halting Problem is undecidable:

Suppose we have a solution to the halting problem called H. Now H takes two inputs:

i) A program P.

ii) An input I for the program P.

H generates an output "halt" if H determines that P stops on input I or it outputs "loop" otherwise.

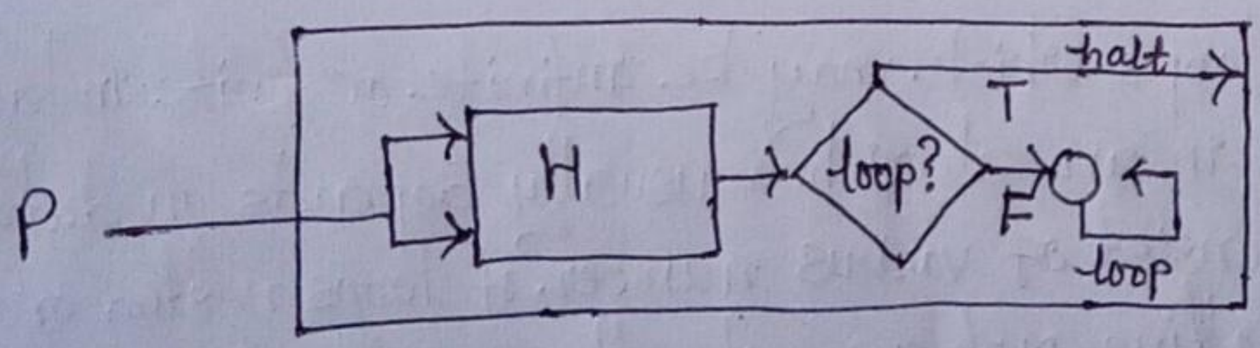


We can treat the program as data and therefore a program can be thought of as input. So, now H can be revised to take P as both inputs (the program and its input) and H should be able to determine if P will halt on P as its input.

Let us construct a new, simple algorithm K that takes H's output as its input and does the following:

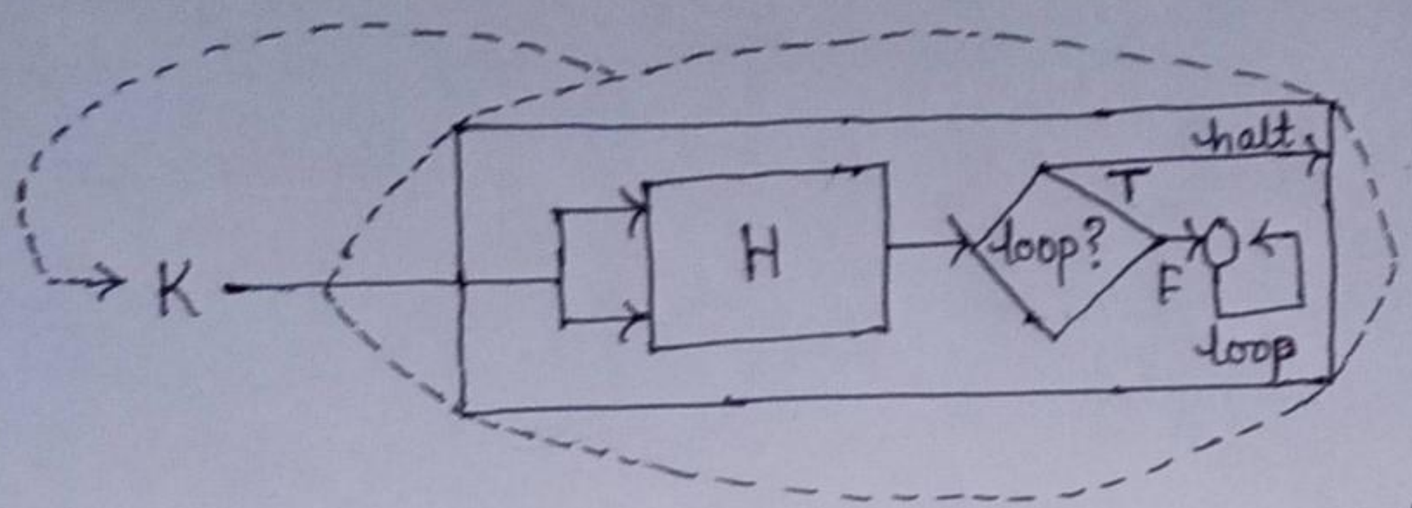
i) If H outputs "loop" then K halts.

ii) Otherwise H's output of "halt" causes K to loop forever.




```
function K(){ if(H()=="loop"){
                    return;
              }
        else{
              while(true); // loop forever
            }
}
```

Since K is a program, let us use K as the input to K.



If H says that K halts then K itself would loop (that's how we constructed it). If H says that K loops then K will halt.
In either case H gives the wrong answer for K. Thus, H cannot work in all cases. We've shown that it is possible to construct an input that causes any solution H to fail. Hence Proved!

⑦. Undecidable Problems about Turing Machines:

A problem is undecidable if there is no Turing machine which will always halt in finite amount of time to give answer as 'yes' or 'no'. An undecidable problem has no algorithm to determine the answer for a given input.

Examples:-

i) Ambiguity of context-free languages: Given a context-free language, there is no Turing Machine which will always halt in finite amount of time and give answer whether language is ambiguous or not.

ii) Equivalence of two context-free languages: Given two context-free languages, there is no Turing Machine which will always halt in finite amount of time and give answer whether two context free languages are equal or not.

iii) Completeness of CFG: Given a CFG and input alphabet, whether CFG will generate all possible strings of input alphabet is undecidable.

iv) Regularity of CFL:- Given a CFL, determining whether this language is regular is undecidable.

If my notes really helped you,then you can support me on esewa for my hardwork.

Esewa ID: 9806470952