Theory of Computation (CT-502)

Course Instructor
ANUJ GHIMIRE

Chapter 5

Undecidability and Computational Complexity

Church Turing Thesis

- It states that every computation or algorithm can be carried out by a Turing Machine.
- This statement was first formulated by Alonzo Church.
- The thesis might be replaced on saying that the notation of effective or mathematical method in logic and mathematics is captured by TM.
- It is generally assumed that such methods must satisfy some of the requirements as:

Church Turing Thesis

- The method consists of a finite set of simple and precise instruction that are described with a finite no. of symbols.
- The method will always produce the result in a finite no. of steps.
- The method can in principle be carried out by human being with only paper and pencil.
- The execution of the method requires no intelligence of human being except that which is needed to understand and execute instruction.

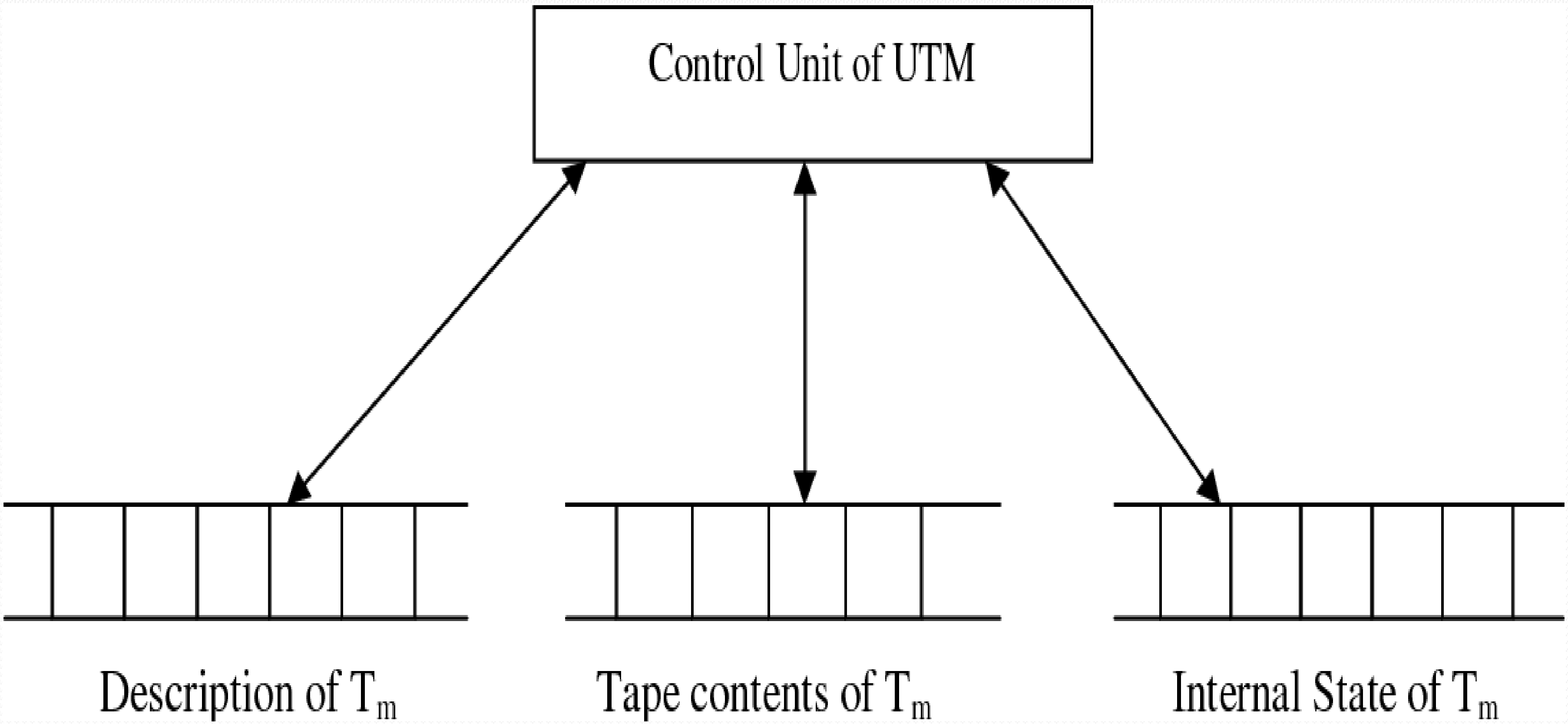
Church Turing Thesis

- Invention of Turing machine has accumulated enough evidence to adopt this hypothesis.
- It is believed that there is no function that can be defined by human, whose calculation can be described by any well defined mathematical algorithm that can not be computed by Turing machine.
- After adopting Church Turing Thesis, we can give precise meaning of term as
  - *"An algorithm is a procedure that can be executed on Turing Machine."*

Universal Turing Machine

- If a Turing Machine is a sound model of computation it should be possible to demonstrate that it can act as a stored program machine, where the program is regarded as an input rather than hardwired.
- We shall construct a Turing Machine M_U that takes as input a description of Turing Machine M and an input word X and simulates the computation of M on input X.
- A machine such as M_U that can simulate the behavior of an arbitrary Turing Machine is called Universal Turing Machine.

Universal Turing Machine



Universal Turing Machine

- Thus, we can describe universal Turing Machine T_U as a Turing Machine that on input <***M*** **,** ***w***> ; where ***M*** is a Turing Machine and ***w*** is a string, simulates computation of ***M*** on input ***w***.
  - T_U *accepts <M , w> iff M accepts w.*
  - T_U *rejects <M , w> iff M rejects w.*

Encoding of Turing Machine

- Formulate a notational system where we can encode both an arbitrary Turing Machine T_1 and an input string ***X*** over an arbitrary alphabet as string $e(T_1)$ and $e(X)$ over some fixed alphabet.
- This encoding must not destroy any information i.e. we must be able to reconstruct Turing Machine T_1 and string ***X***.
- For encoding of Turing Machine, we can only use alphabet {0 , 1}, although Turing Machine may have much larger alphabet.

Encoding of Turing Machine

- Here, we assume two fixed infinite sets

  $Q = \{q_1, q_2, \ldots\ldots\ldots\}$ and

  $S = \{a_1, a_2, \ldots\ldots\ldots\}$

so that for any Turing Machine

$T = \{Q_1, \Sigma, \Gamma, \delta, qo, B, F\}$; Q_1 is subset of Q and Γ is subset of S.

- *We can represent a state or a symbol by a string of 0's of appropriate length. Here 1's are used as separators.*
- *Once, we have established an integer to represent each state, symbol and direction, we can encoding the transition function δ.*

Encoding of Turing Machine

- *Let, one transition rule* $\mathbf{m_1}$ *as* $\boldsymbol{\delta(q_i, a_j) \rightarrow (q_k, a_l, D_m)}$ *for some integer i, j, k, l and m then, we can code this rule by string*

$$S(q_i)\ 1\ S(a_j)\ 1\ S(q_k)\ 1\ S(a_l)\ 1\ S(D_m)$$

where S is encoding function.

- *A code for entire Turing Machine* T_1 *consists of all codes for transitions in some order, separated by pair of 1's like* $m_1\ 11\ m_2\ 11 \ldots\ldots m_n$
- *Now, code for Turing Machine and input string* ***X*** *will be formed by separating them by three consecutive 1's. i.e.* $e(T_M)\ 111\ e(X)$

Encoding of Turing Machine

- First associate a string of 0's to each states, each tape symbols and each of directions.
- Let, function S is defined as :

$S(B) = 0$

$S(a_i) = 0^{i+1}$ for each $a_i \in S$

$S(q_i) = 0^{i+2}$ for each $q_i \in Q$

$S(L) = 00$

$S(R) = 000$

Encoding of Turing Machine

Example: Consider a Turing Machine T as

$T=(\{q_1, q_2, q_3\}, \{a , b\}, \{a, b ,B\}, \delta ,q ,B ,F)$ where δ is defined as :

- $\delta(q_1, b) = (q_3, a, R)$
- $\delta(q_3, a) = (q_1, b, R)$
- $\delta(q_3, b) = (q_2, a, R)$
- $\delta(q_3, B) = (q_3, b, L)$

Now, using encoding function S as defined above

- $S(q_1) = 000$
- $S(q_2) = 0000$
- $S(q_3) = 00000$

Encoding of Turing Machine

- $S(a_1) = 00$
- $S(a_2) = 000$
- $S(B) = 0$
- $S(L) = 00$
- $S(R) = 000$

say, $a_1 = a$ and $a_2 = b$

Now for $\delta(q_1, b) = (q_3, a, R)$ say m_1

$$e(m_1) = S(q_1)1S(b)1S(q_3)1S(a)1S(R)$$
$$= 0001000100000100 1000$$

for $\delta(q_3, a) = (q_1, b, R)$ say m_2

$$e(m_2) = S(q_3)1S(a)1S(q_1)1S(b)1S(R)$$
$$= 00000100100010001000$$

Encoding of Turing Machine

for $\delta(q_3, b) = (q_2, a, R)$ say m_3

$e(m_3) = S(q_3)1S(b)1S(q_2)1S(a)1S(R)$

$= 0000010001000100100 0$

for $\delta(q_3, B) = (q_3, b, L)$ say m_4

$e(m_4) = S(q_3)1S(B)1S(q_3)1S(b)1S(L)$

$= 000001010000010001 00$

Encoding of Turing Machine

- Now, code for Turing Machine is ***e(T)*** as:

  $e(m_1)$ 11 $e(m_2)$ 11 $e(m_3)$ 11 $e(m_4)$

000100010000010010001100000100100010001000110000001000
100010010001100000101000001000100

- For input string x where x = ab, code will be

  e(x) = S(a) 1 S(b) = 001000

- Now, code for Turing Machine T and input string x is

  e(T) 111 e(x)

000100010000010010001100000100100010001000110000010001
00010010001100000101000001000100111001000

Properties of Recursive Language

- **The union of two recursive languages is also recursive.**
  - *Let, L_1 and L_2 be two recursive languages accepted by Turing machines T_1 and T_2.*
  - *Construct a Turing Machine T that first simulates T_1*
    - ***If T_1 accepts than T accepts.***
    - ***It T_1 rejects then T simulates T_2 and accepts if and only if T_2 accepts.***
  - *Here, T is guaranteed to halt because both T_1 and T_2 are algorithms i.e. T accepts $L_1 \cup L_2$*
  - *Hence, $L_1 \cup L_2$ is also recursive since there exists Turing Machine T for it.*

Properties of Recursive Language

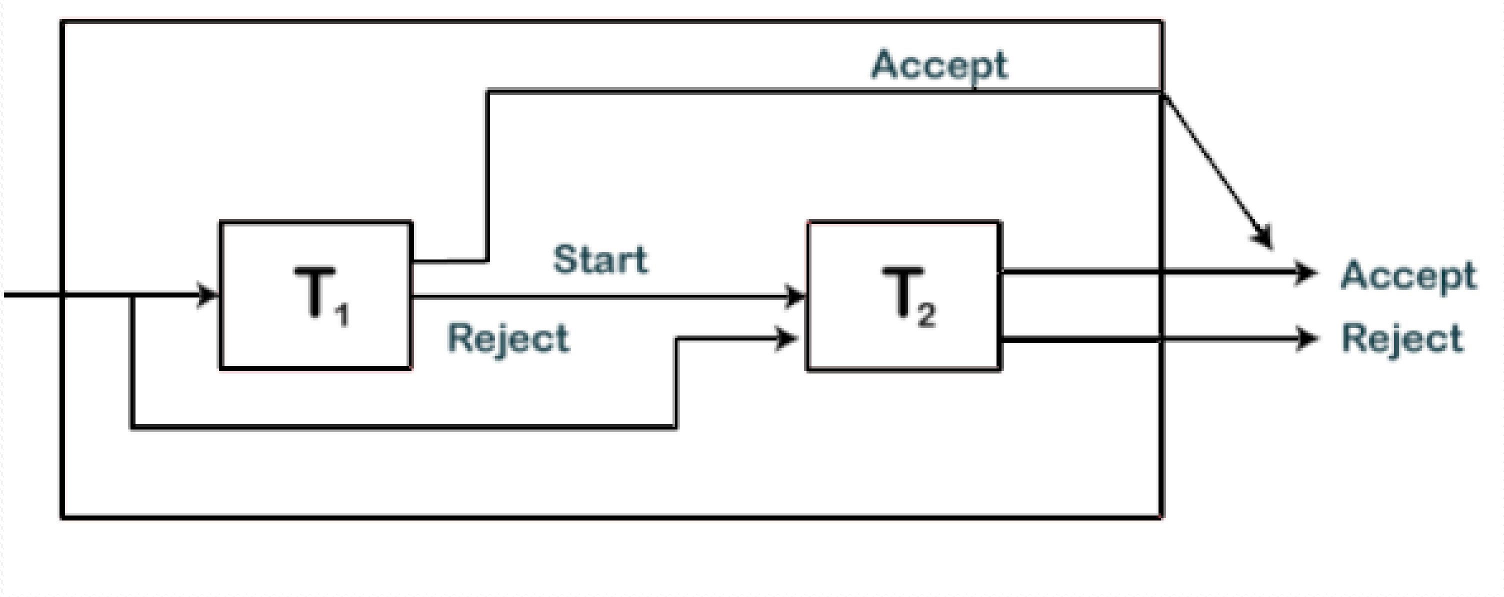


Fig: The union of two recursive languages

Properties of Recursive Language

- **The complement of recursive languages is also recursive.**
  - *Let, L be a recursive language.*
  - *T be a Turing Machine that halts on all inputs and accepts L.*
  - *Let us construct a T' from T so that if T enters a final state on input w, then T' halts without accepting.*
  - *If T halts without accepting, T' enters final state.*
  - *So, L(T') is the language accepted by T' is the complement of L.*
  - *Hence, the complement of recursive language is recursive*

Properties of Recursive Language

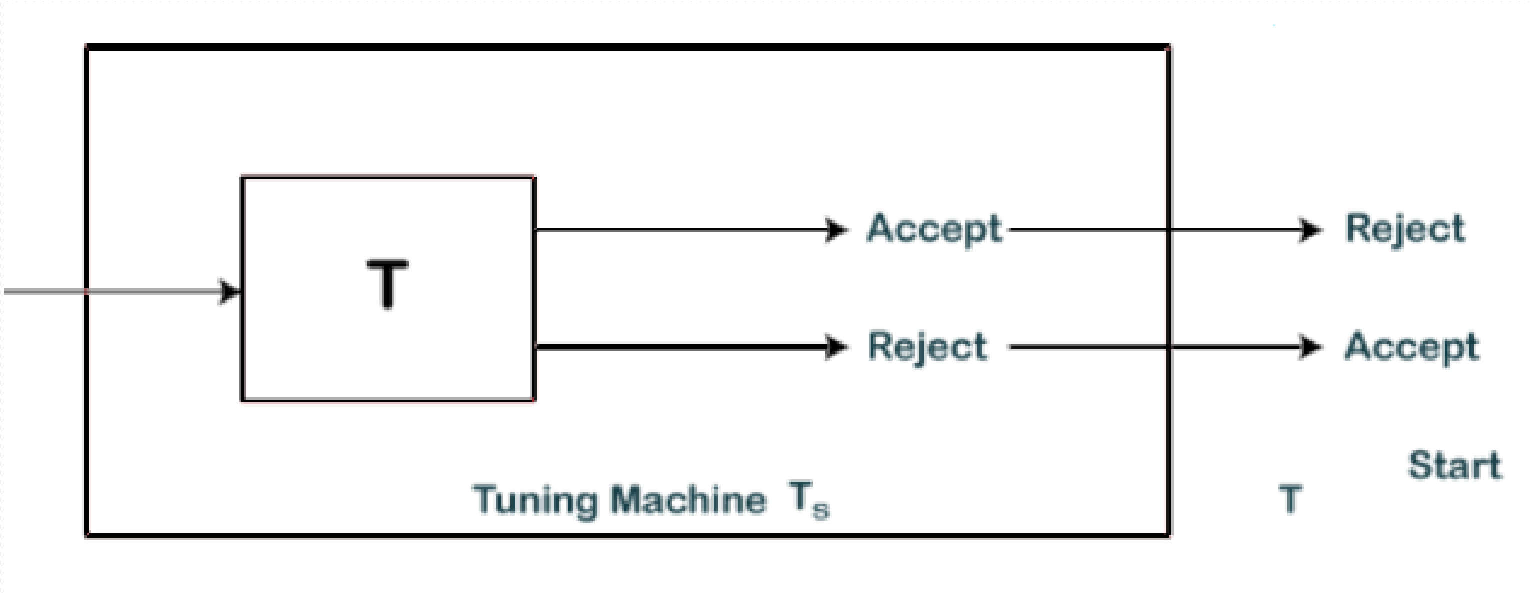
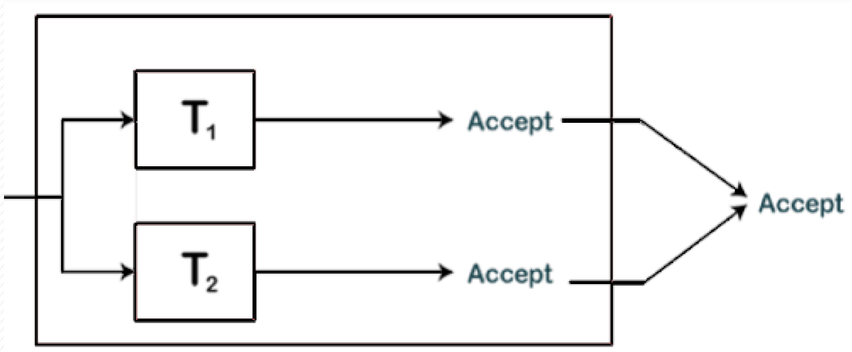


Fig: The complement of recursive languages

Properties of Recursive Language

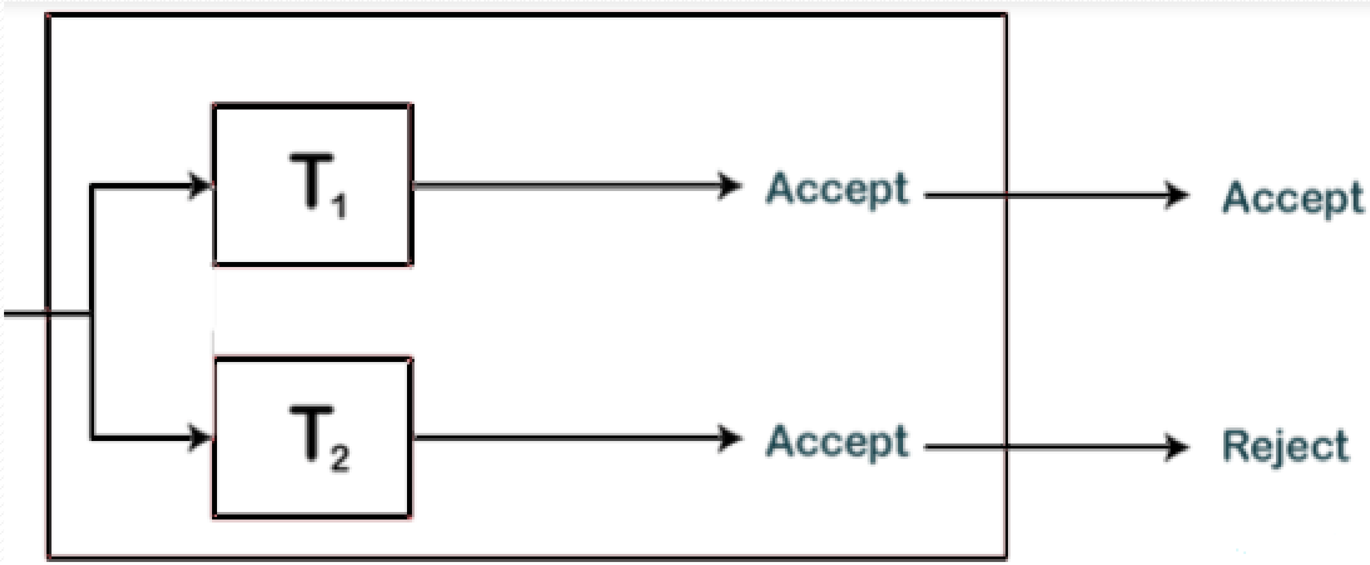
- **The union of two recursively enumerable languages is also recursively enumerable.**
  - *Let, L_1 and L_2 be two recursively enumerable languages accepted by Turing machines T_1 and T_2.*
  - *T be a Turing Machine that can simulate T_1 and T_2 simultaneously on separate tapes.*
  - *If either accepts then T accepts as follows*



Properties of Recursive Language

- **If a language L and its complement L' are both recursively enumerable then L (and L') is recursive**
  - *Let, L and L' be two recursively enumerable languages accepted by Turing machines T_1 and T_2 respectively.*
  - *Construct a Turing Machine T that simulates T_1 and T_2 simultaneously.*
  - *T accepts w if T_1 accepts and rejects w if T_2 will accept.*
  - *Thus T will say either yes or no but not both.*
  - *Since, T is algorithm that accepts L, L is recursive (hence L' also)*

Properties of Recursive Language



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 time and/or space machine of that type require for the solution of that problem.
  - *A given problem can be solved by using more than one computational model i.e. there may be more than one TM that solve the problem.*
  - *It is thus necessary to measure the qualities of alternative model to solve the same computational problem.*

Computational Complexity

- *The quality of a computational model is measured usually in terms of the resources needed by the algorithm for its execution.*
- *The two important resources used for executing a given algorithm are*
  - *Time required to execute that algorithm*
  - *Space required to execute that algorithm.*

Computational Complexity

- *When estimating execution time (Time complexity) we are interested in growth rate and not in absolute time.*
- *Similarly, we are interested in growth rate of memory need (space complexity) rather than the absolute value of space.*
- *So the boundary time and boundary space for executing an algorithm are usually expressed in terms of known mathematical functions.*

Time and Space Complexity of TM

- The model of computation we have chosen is the 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.

Time and Space Complexity of TM

- The time and space complexity of a TM can be defined as:
  - Let ***T*** be a TM and time complexity of ***T*** is the function $\boldsymbol{T_t}$ defined on the natural numbers as; for n ∈ N, $\boldsymbol{T_t(n)}$ 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***, $\boldsymbol{T_t(n)}$ is undefined.

Time and Space Complexity of TM

- The space complexity of T is the function S_t defined as $S_t(n)$ 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, $S_t(n)$ is undefined.

- *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**.*

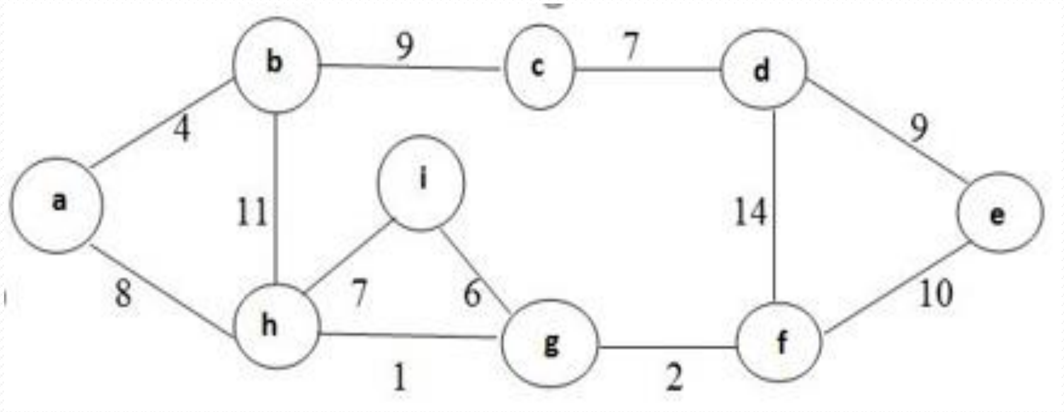
Complexity Classes: Class-P

- 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).
- Example of Class-P problem: ***Minimum Spanning Tree, Merge Sort, Binary Search.***
  - *This problem can be solved using Kruskal's Algorithm with complexity $O(n^2)$.*

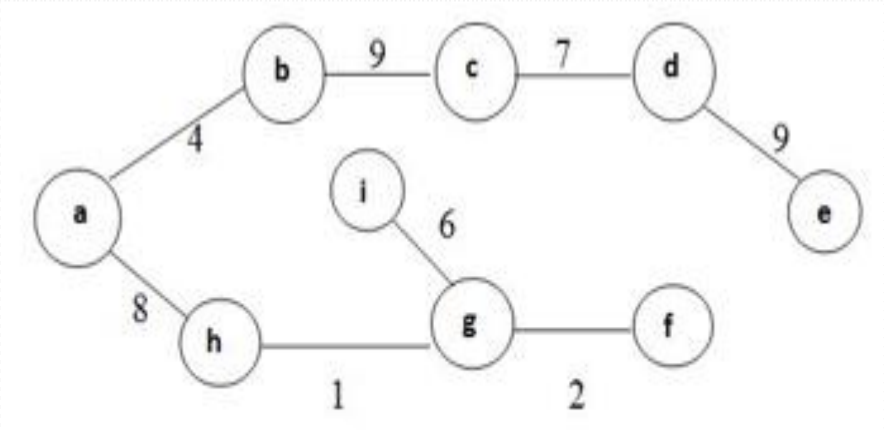
Complexity Classes: Class-P

- ***Minimum Spanning Tree (Algorithm)***
  - *We start with listing of all vertex pairs of the given graph according to their weight in ascending order.*
  - *After that the vertex pair with least weight from the list is selected and added to the tree.*
  - *Then the next vertex pair with minimum weight is selected and added to the tree.*
  - *During the process of adding vertex pair to the tree, if any vertex pair forms a cycle we discard it and move to vertex pair with next minimum weight from the list.*
  - *The process is continued until all the vertex pairs from the list are added to the tree.*
  - *When the list becomes empty the tree obtained is the required minimum spanning tree.*

Complexity Classes: Class-P



Minimum Spanning Tree is:



Complexity Classes: Class-NP

- The class NP is the set of problems that can be solved by a non-deterministic TM in polynomial time.
- Formally, we can say 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.
- Example of Class-NP problem: ***Travelling Salesman Problem, Hamiltonian Cycle Problem*** *and* ***Linear Programming***

Classification of Class-NP

- The class NP is classified into:
  - *NP-Complete Problem*
  - *NP-Hard Problem*
- ***NP-Complete:***
  - *Any NP problem can be considered as NP-complete problem if all the other NP problem are reducible to it in polynomial time complexity.*
  - *If a problem is NP and all other NP problems are polynomial time reducible to it, the problem is NP-complete.*

Classification of Class-NP

- *A decision problem* ***L*** *is* ***NP-complete*** *if it follow the below two properties:*
  - ***L*** *is in* ***NP***
  - *Every problem in* ***NP*** *is reducible to* ***L*** *in polynomial time*
  - *The most important property of NP complete is the so called polynomial time reducibility.*
  - *Any NP complete problem can be transformed into any other NP complete problem in polynomial time.*
  - *Thus, if it could be proved that any NP complete problem is formally intractable, all such problems would have proved intractable and vice-versa.*
  - *Eg. Travelling Salesman Problem, Vertex Cover Problem etc.*

Classification of Class-NP

- ***NP-Hard:***
  - *A problem is NP-hard if an algorithm for solving it can be translated into one for solving any other NP-problems (non deterministic polynomial time) problem.*
  - *NP-hard therefore means "at least as hard as any NP-problem" although it might, in fact, be harder.*
  - *This class is potentially harder to solve than NP-complete problems because although if any NP-complete problem is intractable then all NP-hard problems are intractable, the reverse is not true.*
  - *Eg. Halting Problem.*

Classification of Class-NP

	P	NP	NP-complete	NP-hard
Solvable in polynomial time	✓			
Solution verifiable in polynomial time	✓	✓	✓	
Reduces any NP problem in polynomial time			✓	✓

Undecidability

- In computability 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- ***the problem is not decidable***.
- In computability theory, **the halting problem** is a decision problem which can be stated as follows:
  - *Given a description of a program and a finite input, decide whether the program finishes running or will run forever.*

Undecidability

- Alan Turing proved in 1936 that a general algorithm running on a Turing machine that solves the halting problem for all possible program-input pairs necessarily cannot exist.
- Hence, ***the halting problem is undecidable*** for Turing Machines.

Undecidability

- The problems for which no algorithms exists are called undecidable or unsolvable. e.g. halting problem
- Following problems about Turing Machine are undecidable:
  - *Given a Turing Machine M and input string w, does M halts on input w?*
  - *Given a Turing Machine M, does M halt on the empty tape?*
  - *Given a Turing Machine M, is there any string at all on which M halts?*
  - *Given a Turing Machine M, does M halts on every input string?*
  - *Given two Turing Machines M_1 and M_2, do they halt on same input string?*

For proof go through Text Book "Element of the TOC" by Lewis-Page-255 (2nd Edition).

Halting Problem

- There are limits to the power of Turing Machine.
- A Turing Machine continues until it reaches accept state or reject state where it will halt.
- If it never reaches one , then it continues computing forever.
- There exists problems that Turing Machine cannot solve.
- The best known problem i.e. unsolvable by a Turing Machine is the halting problem.

Halting Problem

- ***Halting Problem is:***
  - *"Given an arbitrary Turing Machine T as input and equally arbitrary tape t, decide whether T halts on t "*
  - *"To determine for any arbitrary given Turing machine T_M and input w, whether T_M will eventually halts on input w."*
- *So can we have an algorithm that will tell that the given program will halt or not?*
- *In terms of Turing machine, will it terminate when run on some machine with some particular given input string?*

Halting Problem

- The answer is no we cannot design a generalized algorithm which can appropriately say that given a program will ever halt or not?
- Given a program written in some programming language(c/c++/java) will it ever get into an infinite loop(loop never stops) or will it always terminate(halt)?
- *The only way is to run the program and check whether it halts or not.*

Halting Problem

- Alan Turing has proved that a general algorithm running on a Turing machine that solves the halting problem for all possible program-input pairs necessarily cannot exist.
- He uses proof by contradiction to make a proof.
- To make a proof, we assume the fact that we have a program that always correctly determines whether another program halts and that program be ***H***.
- ***H*** takes in another program as input and after scanning through the program it tells us if the program will halt or if the program run forever.

Halting Problem

- Now let’s create a bigger machine called ***D*** that encompasses ***H*.**
- ***D*** is designed such that for every input it gets it gives it to ***H*** and whatever the ***H*** says, it does the opposite.
- So if ***H*** says the program runs forever, then ***D*** will halt and if ***H*** says that the program halts then ***D*** never halts.
- The contradiction arises when we give machine ***D*** its own program.

Halting Problem

- So when ***D*** takes in its own program as input.
- It passes it to ***H*** and ***H*** decides whether ***D*** will halt.
- Let's say ***H*** determines that ***D*** will halt but because ***D*** is designed to do the opposite of what ***H*** says, then ***D*** ends up running forever even though ***H*** said ***D*** will halt.
- This concludes ***H*** is wrong but we assumed in the beginning that ***H*** is always right.

Halting Problem

- So let's try again ***D*** takes in its own program and process it to ***H*.**
- This time ***H*** says ***D*** does not halt and runs forever and again, because ***D*** is designed to do the opposite of what ***H*** says, ***D*** halts i.e. ***H*** is wrong again.
- But initially we assume that this the program ***H*** exists that always correctly tells us if a program will halt or not.
- But our experiment with ***D*** just showed that if ***H*** existed and we built ***D*** using ***H*,** then ***H*** can be wrong.

Halting Problem

- This contradicts our initial assumption that ***H*** is always right.
- Therefore a program such as ***H*** that correctly determines if another program will halt cannot exist.

End of Chapter 5

Thank You !!!!