Unit 5

Graph Structures

(1) Graph ADT:

Abstract Datatype Graph

Instance: A graph is a collection of two sets V & E where V is a set of vertices and E is set of edges.

Operations:

1. Graph(): Creates a new, empty graph
2. addVertex(Vertex Key): Add a vertex to the graph with the given vertex key.
3. AddEdge(fromVert, toVert): Add a new, directed edge to the graph that connect two vertices.
4. addEdge(fromVert, toVert, weight): Add a new, weighted, directed edge to the graph that connect two vertices.
5. display(): Display the graph either using adjacency matrix or adjacency list representation.

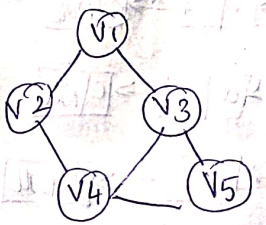
(2) Representation of graph:

The various representations of graph are

1. Adjacency Matrix representation
2. Adjacency list representation:

Adjacency Matrix:

Consider a graph G of n vertices and matrix M. If there is an edge present between vertices V_i & V_j then M[i][j]=1 else m[i][j]=0. Note that for an undirected graph if m[i][j]=1 then for m[j][i] is also 1. Here are some graphs shown by adjacency matrix.



	1	2	3	4	5
1	0	1	1	0	0
2	1	0	0	1	0
3	1	0	0	1	1
4	0	1	1	0	1
5	0	0	1	1	0

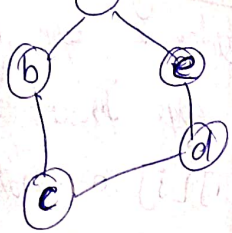
* Its a simple task. Adjacency matrix is a 2D array. The alg for creation of graph using adjacency matrix will be as follow:

1) Declare an array of M[size][size] which will store the graph
2) Enter how many nodes you want in a graph.
3) Enter the edge of the graph by two vertices each say V_i, V_j indicates some edge.
4) If the graph is directed set M[i][j] = 1. If graph is undirected set M[i][j]=1 & m[j][i]=1 as well.
5) When all the edges for the desired graph is entered print the graph m[i][j].

Adjacency list:

In this type of graph linked list is used so called as adjacency list representation.

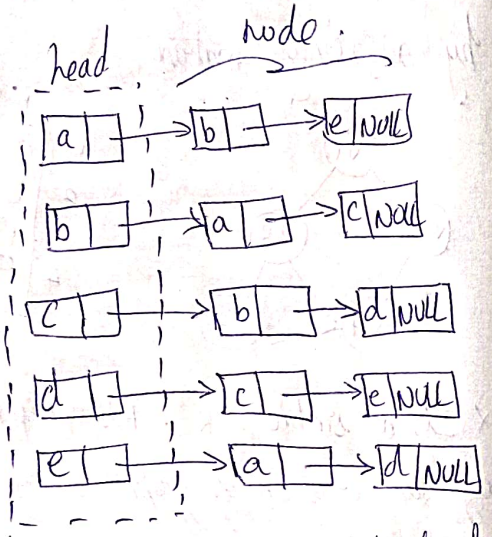
* For graph G1, the nodes a, b, c, d, e



maintain the linked list of these head nodes as well as the adjacent node.

* The C structure will be.

```
typedef struct head
{
  char data;
  struct head *down;
  struct head *next;
};
typedef struct node
{
  int ver;
  struct node *link;
};
```



* This is purely the adjacency list graph. The down pointer helps us to go to each node in the graph whereas the next node is for going to adjacent node of each of the head node.

(3) Graph Traversals:

The graph can be traversed using Breadth first search and depth first search method.

Breadth first Traversal:

For the graph to traverse it by BFS, a vertex V_i in the graph will be visited first, then all the vertices are $(V_2, V_3, V_4 \cdots V_n)$ ~~wise~~. So $V_2, V_3 \cdots V_n$ will be printed first. Then again from V_2 the adjacent vertices will be printed. This process will be continued for all the vertices to get encounted. To keep track of all vertices and their adjacent vertices we will make use of queue data structure. Also we will ~~also~~ make use of an array for visited nodes. The nodes which are ged visited are set to 1. In short BFS traversal follow the path in breadthwise.

Algorithm:

1) create a graph. Depending on the type of graph ie directed or undirected set the value of the flag as either 0 or 1.
2) Read the vertex from which you want to traverse the graph say V_i.
3) Initialize the visited array to 1 at the index of V_i.
4) Insert the visited vertex V_i in the queue.
5) Visit the vertex which is at the front of the queue. Delete it from the queue & place its adjacent node in the queue.
6) Repeat the step 5, till the queue is not empty.
7) Stop.

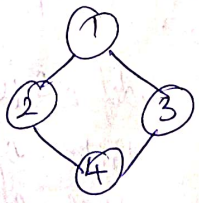
Program

```
def BFS (visited, graph, node):
    visited.append(node)
    queue.append(node)
    while queue:
        V1 = queue.pop(0)
```

```
print (v1, end = " ")
for v2 in graph [v1]:
if v2 not in visited:
visited . append (v2)
queue . append (v2)
#driver code
print ( program for displaying the graph using BFS")
graph = {'1' : ['2','3'] ,'2' : ['1', '4'], '3': ['1','4'],
         '4' : ['2','3'] }
visited = []
queue = []
print ("In The graph is as follows:")
n = len (graph)
i = 1
for i in range (n):
print [list (graph . keys ())[i] ,":" list (graph . values ())[i])
print ("BFS of graph is")
bfs (visited , graph , '1')
```

o/p Program for displaying the graph using BFS

The graph is as follows

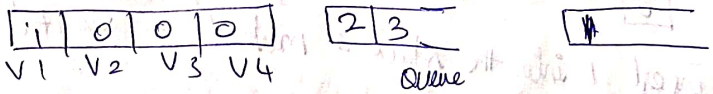


1: ['2', '3']
2: ['1', '4']
3: ['1', '4']
4: ['2', '3']
Bf of graph is 1 2 3 4

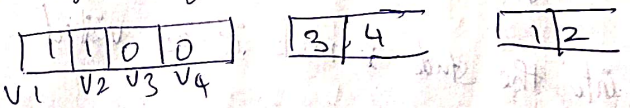
Breadth Explanation:

Step1: start with vertex 1, find adjacent of 1 & insert in queue.

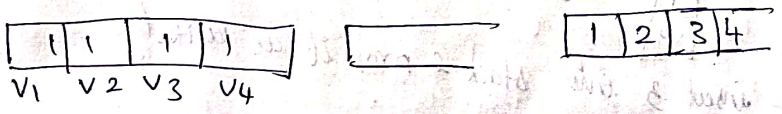
Step2



Step3 Find adjacent of 2 and insert in queue, mark 2 as visited. adjacent of 2 is 4.



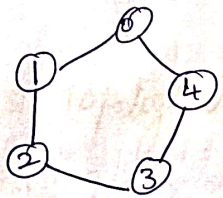
Step4 visit the element in queue & delete from queue



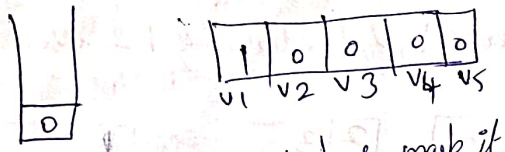
Depth first Traversal:

In depth first search traversal, we start from one vertex, and ~~trav~~ traverse the path as deeply as we go. when there is no vertex further, we traverse back & search for anyone unvisited vertex. An array is maintained for storing the visited vertex.

eg:-

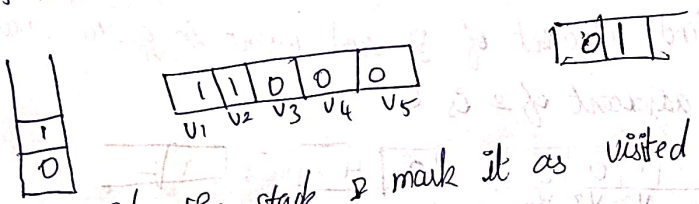


Start from 0 and adjacent vertex is 1, mark 0 as visited and push into the stack.

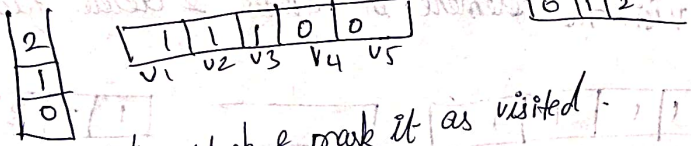


o/p 0

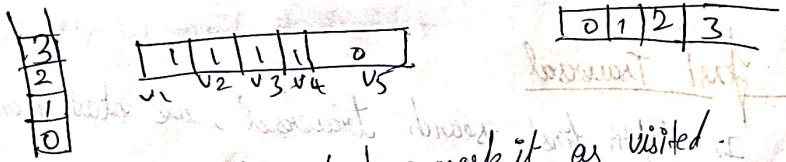
Step2 insert 1 into the stack & mark it as visited.



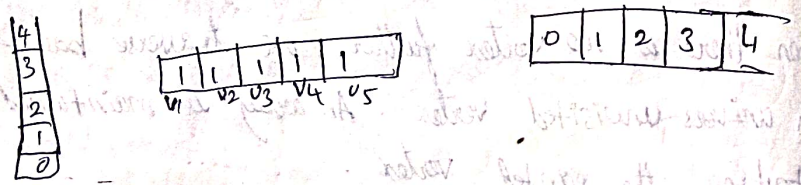
Step3 insert 2 into the stack & mark it as visited



Step4 insert 3 into stack & mark it as visited.

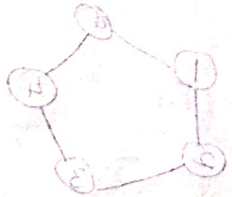


Step5 insert 4 into the stack & mark it as visited.



Program:

```
def dfs (visited, graph, v1):
    if v1 not in visited:
        print (v1)
        visited.add (v1)
        for v2 in graph [v1]:
            dfs (visited, graph, v2)
```



```
#driver code
print (" program for displaying the graph using DFS")
graph = {'1': ['2','3'], '2': ['1','4'], '3': ['1','4'], '4': ['2','3']}
visited = set ()
print ("\n The graph is as follows")
n = len (graph)
i = 1
for i in range (n):
    print (list (graph.keys ()) [i], ":", list (graph.values ()) [i])
print ("depth first search of graph is")
dfs (visited, graph, '1')
```

o/p Program for displaying the graph using DFS

The graph is as follows

```
1 : ['2', '3']
2 : ['1', '4']
3 : ['1', '4']
4 : ['2', '3']
```

Depth first search of graph is

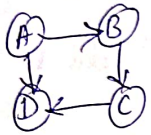
```
1
2
3
4.
```

(4) DAG & Topological ordering:

Definition: Topological sorting for directed acyclic graph (DAG) is a ord linear ordering of vertices such that every directed edge 'uv' vertex u comes befor v in ordering.

* Directed acyclic graph is a kind of graph which has no directed cycle.

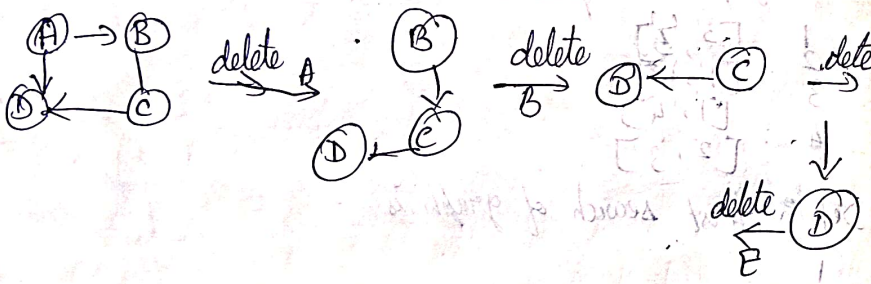
eg :-



Algorithm:

Following are the steps to followed in alg:

1. From a given graph find a vertex with no incoming edges. Delete it along with the edges outgoing from it. If there are more than one such vertices then break randomly.
2. Note the vertices that are deleted.
3. All these recorded vertices give topological sorted list.



Hence the list after topological sorting will be A, B, C, D.

(5) Greedy algorithm:

In an alg strategy like greedy, the decision of soln is taken based on the info available. The greedy method is a straightforward method. Its popular for obtaining the optimized solutions. In greedy technique, the soln is constructed through seq of steps, each expanding a partially constructed soln obtained so far, until a complete soln to the problem is reached. At each step the choice made should be:

1. Feasible
2. Locally optimal
3. Irrevocable

In short, while making a choice there should be a greedy for the optimum solution.

* The greedy method uses the subset paradigm or ordering paradigm to obtain the soln. In subset paradigm, at each stage the decision is made based on whether a particular input is in optimal soln or not.
* For eg solving knapsack problem this approach is use.

Greedy method follows the activities:

1. First we select soln from input domain.
2. Then we check whether the soln is feasible or not.
3. From the set of feasible soln, particular soln that satisfies or nearly satisfies the objective of the function. Such soln is called optimal soln.
4. As greedy method works in stages. At each stage only one input is considered at each time. Based on the input it is decided whether particular input gives the optimal solution or not.

Applications of Greedy method:

i) Knapsack problem.
ii) Prims alg for minimum spanning tree.
iii) Kruskals alg for minimum spanning tree.
iv) Finding shortest path.
v) Job sequencing with deadline
vi) Optimal storage on tapes.

(5) Dynamic programming:

Dynamic programming is typically applied to optimization problem. This tech is invented by US mathematician Richard Bellman in 1950. In the world dynamic programming the word programming stands for planning and it does not mean by computer programming.

* Dynamic programming is a tech for solving problems with overlapping subproblems.
* In this method each subproblem is solved only once. The result of each subproblem is recorded in the table from which we can obtain a soln to the original problem.
* for a given problem we may get any number of soln. From all those solution we seek for optimum

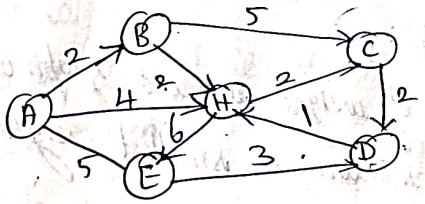
Solution:

(4) Dynamic

(4) Shortest paths:

Dijstras Algorithm is a popular alg for finding shortest path. This alg is called single source shortest path alg because in this alg, for a given vertex called source the shortest path to all other vertices is obtained.

eg:-



Sourcenode $S=\{A\}$. The target nodes $P=\{B, E, H, C, D\}$.

Step1 $S=\{A\}$
$d(A,B)=2$ (min)
$d(A,E)=5$
$d(A,H)=4$
$d(A,C)=\infty$
$d(A,D)=\infty$
minimum dist 2. so choose B

Step2 $S=\{A,B\}$
$P=\{E,H,C,D\}$
$d(A,E)=5$
$d(A,H)=4$ (min)
$d(A,C)=2+5=7$
$d(A,D)=\infty$
min dist 4. So choose H.

Step3 $S=\{A,B,H\}$
$P=\{E,C,D\}$
$d(A,E)=5$ (min)
$d(A,C)=4+2=6$
$d(A,D)=\infty$.
choose vertex E.

Step4 $S=\{A,B,H,E\}$
$P=\{C,D\}$
$d(A,C)=6$ (min)
$d(A,D)=5+3=8$.
choose vertex C.

Step5 $S=\{A,B,H,E,C\}$
$P=\{D\}$
$d(A,D)=8$.
Choose vertex D.

$S = \{A, B, H, E, C, D\}$

So shortest path is

Path	Distance
A-B	2
A-H	4
A-E	5
A-H-C	6
A-E-D	8

(8) Minimum Spanning Tree:

Minimum spanning tree of weight connected graph G is a sequence spanning tree with minimum or smallest weight.

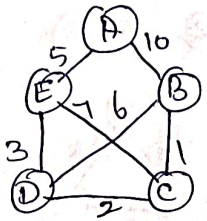
Spanning Tree:

A spanning tree of graph G is a subgraph which is basically a tree and it contains all the vertices of G containing no circuit.

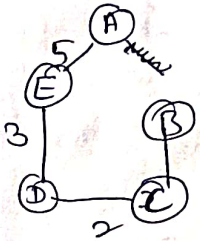
weight of the tree:

A weight of the tree is defined as the sum of weights of all its edges.

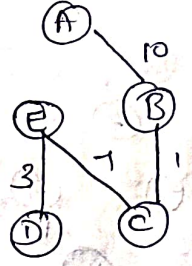
eg:- Consider a graph G as given below. This graph is called weighted connected graph because some weights are given along every edge and the graph is a connected graph.



Graph G



wt = 11



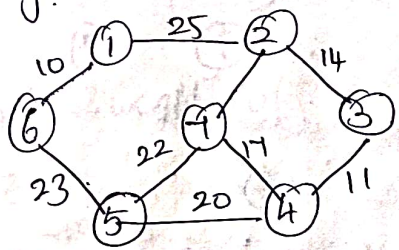
wt = 21

Appl of spanning trees:

1. Spanning trees are very important in designing efficient routing alg.
2. Spanning tree have wide appl in many areas such as network design.

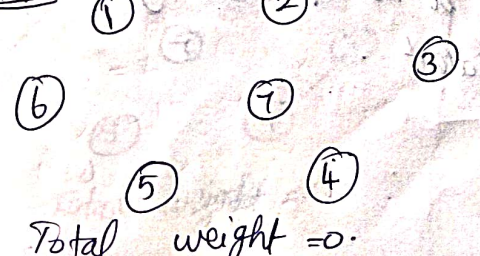
Prims alg:

eg:-



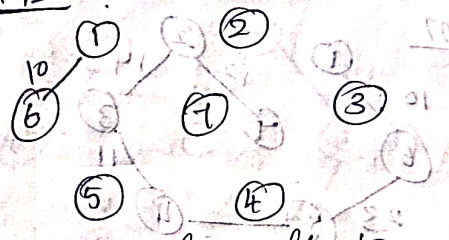
Select edge with minimum weight care should be taken for not forming circuit.

Step1



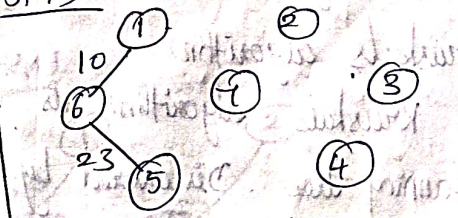
Total weight = 0.

Step2



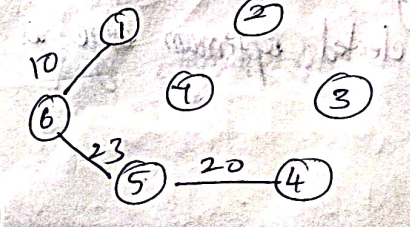
Total weight = 10.

Step3

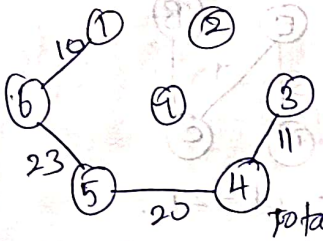


Total weight = 33

Step4

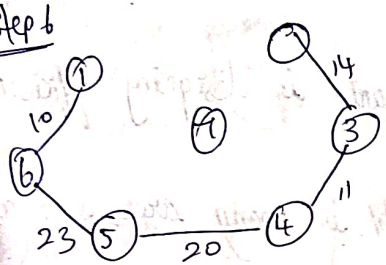


Step 5



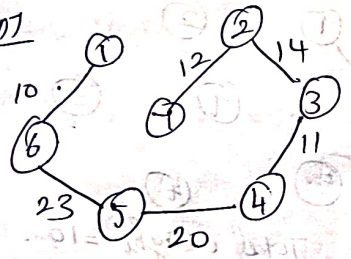
Total weight 64

Step 6



Total weight - 78

Step 7

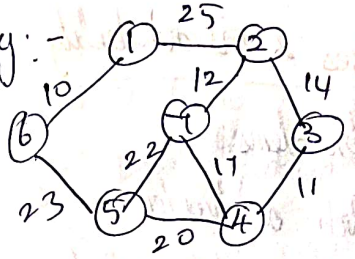


Total weight - 90.

Kruskals algorithm:

Kruskals algorithm is another alg of obtaining minimum spanning tree. Discovered by second year graduate student Joseph Kruskal. In this alg always the minimum cost edge has to be selected. But it is not necessarily that selected optimum edge is adjacent.

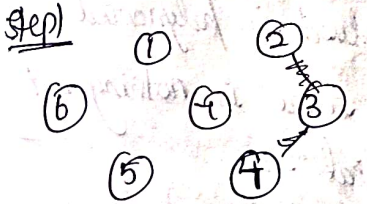
eg:-



First we will select all the vertices. Then an edge with optimum weight is selected from heap, even though it is not adjacent to previously selected edge.

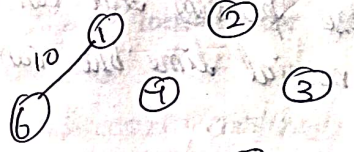
* circuit should not be formed.

Step 1



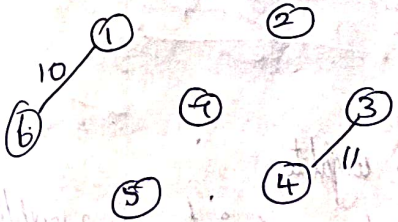
Total weight = 0

Step 2



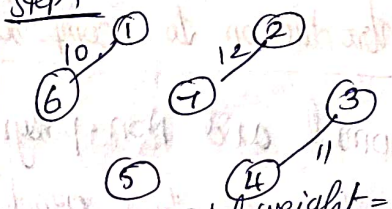
Total weight = 10

Step 3



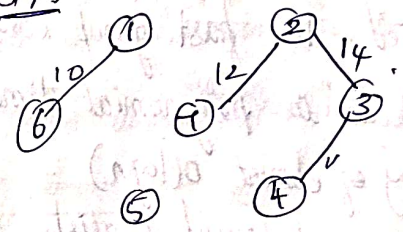
Total weight = 21

Step 4



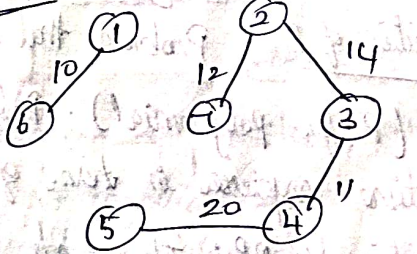
Total weight = 33

Step 5



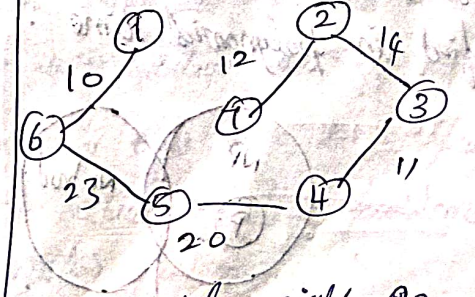
Total weight = 47

Step 6



Total weight = 67.

Step 7



Total weight = 90

⑨ Introduction to complexity classes

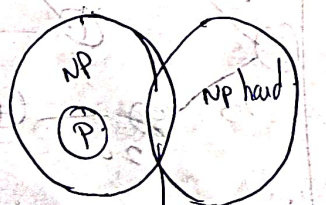
Polynomial and Non-polynomial problems:

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. Searching element $O(\log n)$ Sorting of element $O(\log n)$.

* The second group consists of problem that can be solved in non-deterministic polynomial time. eg. Knapsack problem $O(2^{n/2})$ & Travelling salesperson problem $(O(n^2 2^n))$.

Definition of P: Problems that can be solved in polynomial time (P for polynomial). Polynomial time is nothing but the time expressed in terms of polnomial.

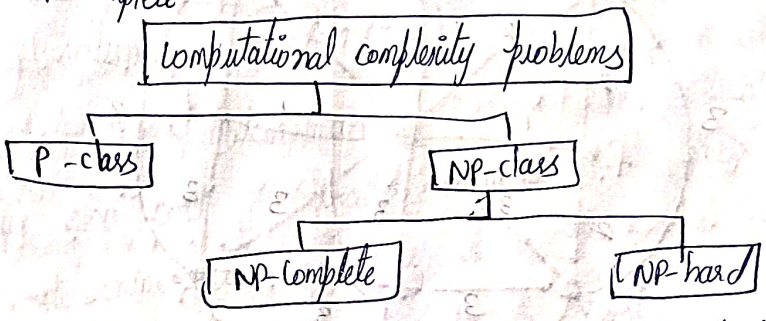
Definition of NP: It stands for non-deterministic polynomial time'. NP class problem are those problem that can be solved in non-deterministic polynomial time but can be verified in polynomid time.



NP complete

*NP class problems can be further categorized into NP-complete & NP hard problems.

* A problem D is called NP-complete if.
  1) It belong 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 problem are NP hard but all NP-hard problem cannot be NP complete.

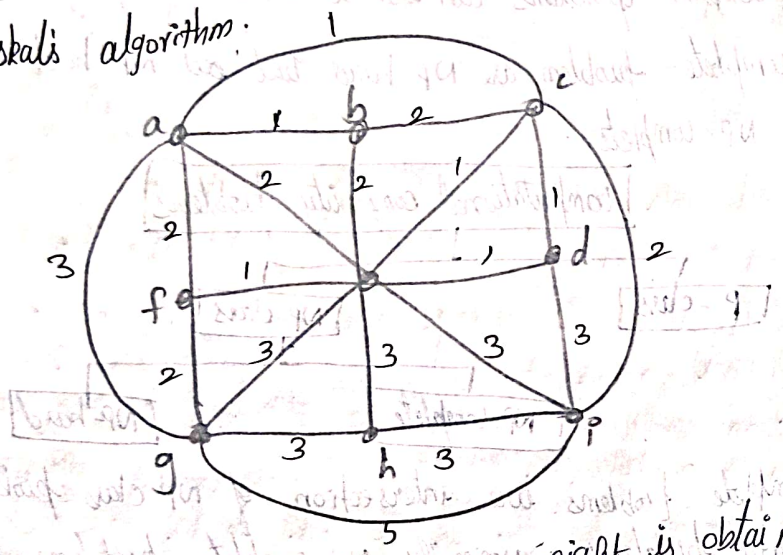


* NP complete problems are intersection of NP class problems
* NP hard problems: Normally NP complete problems are decision problems (eg is a graph Euler or not). NP hard problems are optimization problems (eg: Shortest path problem).
* Computational complexity: The compulation problems is an infinite collection of instance with solution for every instance.
* Complexity class: The complexity class is a set of problems of related complexity: It include function problems, pclasses, Np classes, optimization problem.
* Intractability: Problems that can be solved by taking a long time for their soln is known as intractable problems.

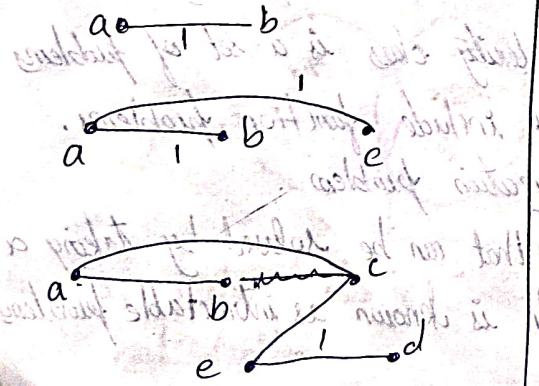
*A Problem D is called NP complete if
i. It belongs to class NP
ii) Every problem in NP can also be solved in Polynomial time

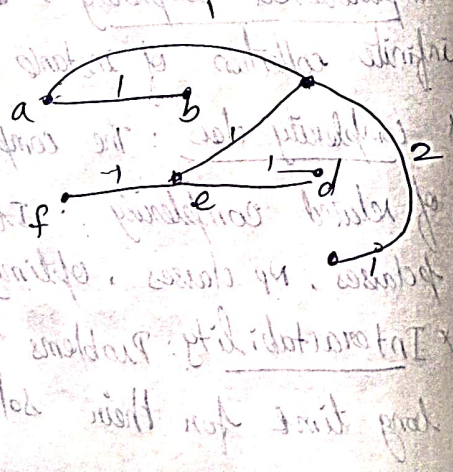
eg of P class Problem

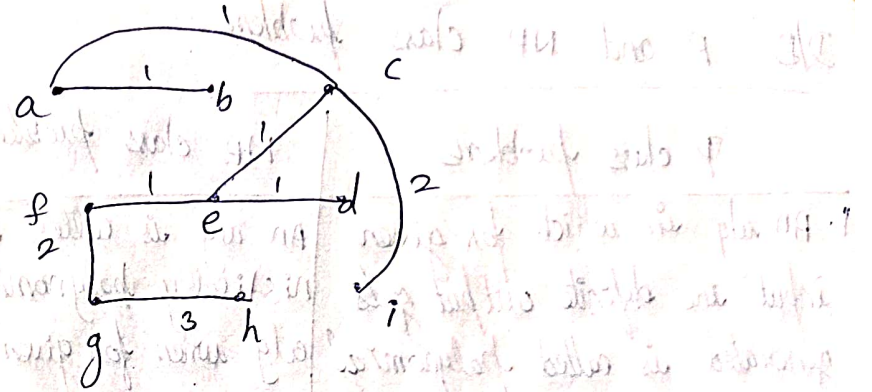
Kruskals algorithm:



In Kruskals algorithm the minimum weight is obtained & Circuit should not be formed. In this alg it is necessary edge of minimum weight to be adjacent. Start vertex is a.



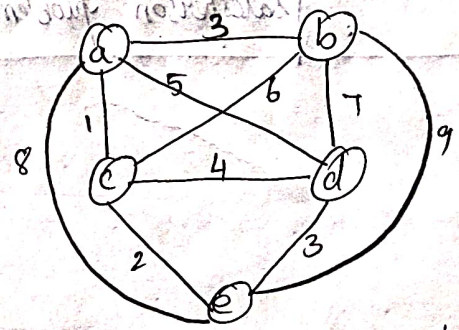




eg of NP class Problem:

Travelling salesmans 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. If we get no soln at all by applying an alg then the travelling salesman problem belongs to NP hard class.

D/B P and NP class problems:

P class problems	NP class problems.
1. An alg in which for given input the definite output gets generated is called polynomial time alg (P class)	An alg is called non-deterministically polynomial time alg when for given input. there are more than one paths that the alg can follow.
2) All the P class problems are basically deterministic.	All the NP class problems are basically non-deterministic.
3. Every problem which is a P class is also in NP class	Every problem which is in NP is not the P class problem.
4. P class problems can be solved efficiently.	NP class problems can not be solved efficiently as efficiently as P class problems.
5. eg:- Binary search, bubble sort.	eg: Knapsack problem, traveling salesperson problem.