UNIT –V

Dynamic Programming: *General method, applications- 0/1 knapsack problem, All pairs shortest path problem, Travelling salesperson problem, Reliability design.*

Backtracking*: General method, applications-n-queen problem, sum of subsets problem, graph coloring, Hamiltonian cycles.*

Introduction to NP-Hard and NP-Complete problems*: Basic Concepts*

5.1 DYNAMIC PROGRAMING GENERAL METHOD:

1. The idea of dynamic programming is thus quit simple: avoid calculating the same thing twice, usually by keeping a table of known result that fills up a sub instances are solved.
2. Divide and conquer is a top-down method.
3. When a problem is solved by divide and conquer, we immediately attack the complete instance, which we then divide into smaller and smaller sub-instances as the algorithm progresses.
4. Dynamic programming on the other hand is a bottom-up technique.
5. We usually start with the smallest and hence the simplest sub- instances.
6. By combining their solutions, we obtain the answers to sub-instances of increasing size, until finally we arrive at the solution of the original instances.
7. The essential difference between the greedy method and dynamic programming is that the greedy method only one decision sequence is ever generated.
8. In dynamic programming, many decision sequences may be generated. However, sequences containing sub-optimal sub-sequences cannot be optimal and so will not be generated.

APPLICATIONS:

5.2. 0/1 KNAPSACK PROBLEM:

1. This problem is similar to ordinary knapsack problem but we may not take a fraction of an object.

2. We are given ' N ' object with weight W_i and profits P_i where I varies from l to N and also a knapsack with capacity ' M '.

3. The problem is, we have to fill the bag with the help of ' N ' objects and the resulting profit has to be maximum.

4. Formally, the problem can be started as, maximize $\sum_{i=l}^{n} X_i P_i$

$$\text{subject to } \sum_{i=1}^{n} X_i W_i \; L \; M$$

5. Where X_i are constraints on the solution $X_i \in \{0,1\}$. (u) X_i is required to be 0 or 1. if the object is selected then the unit in 1. if the object is rejected than the unit is 0. That is why it is called as 0/1, knapsack problem.

6. To solve the problem by dynamic programming we up a table T[1…N, 0…M] (ic) the size is N. where 'N' is the no. of objects and column starts with 'O' to capacity (ic) 'M'.

7. In the table T[i,j] will be the maximum valve of the objects i varies from 1 to n and j varies from O to M.

RULES TO FILL THE TABLE:-

1. If i=l and j < w(i) then T(i,j) =o, (ic) o pre is filled in the table.

2. If i=l and j ≥ w (i) then T (i,j) = p(i), the cell is filled with the profit p[i], since only one object can be selected to the maximum.

3. If i>l and j < w(i) then T(i,l) = T (i-l,j) the cell is filled the profit of previous object since it is not possible with the current object.

4. If i>l and j ≥ w(i) then T (i,j) = {f(i) +T(i-l,j-w(i)),. since only 'l' unit can be selected to the maximum. If is the current profit + profit of the previous object to fill the remaining capacity of the bag.

5. After the table is generated, it will give details the profit.

ES TO GET THE COMBINATION OF OBJECT:

- Start with the last position of i and j, T[i,j], if T[i,j] = T[i-l,j] then no object of 'i' is required so move up to T[i-l,j].
- After moved, we have to check if, T[i,j]=T[i-l,j-w(i)]+ p[I], if it is equal then one unit of object 'i' is selected and move up to the position T[i-l,j-w(i)]

- Repeat the same process until we reach T[i,o], then there will be nothing to fill the bag stop the process.
- Time is 0(nw) is necessary to construct the table T.
- Consider a Example,

M = 6,
N = 3
$W_1 = 2$, $W_2 = 3$, $W_3 = 4$
$P_1 = 1$, $P_2 = 2$, $P_3 = 5$

i ⟶ 1 to N
j ⟶ 0 to 6

i=l, j=o (ic) i=l & j < w(i)

o<2 ⟶ T_1,o =0

i=l, j=l (ic) i=l & j < w(i)

l<2 ⟶T_1,1 =0 (Here j is equal to w(i)⟶ P(i)

i=l, j=2
2 o,= T_1,2 = l.

i=l, j=3
3>2,= T_1,3 = l.

i=l, j=4
4>2,= T_1,4 = l.

i=l, j=5
5>2,= T_1,5 = l.

i=l, j=6
6>2,= T_1,6 = l.

=> i=2, j=o (ic) i>l,j<w(i)
o<3= T(2,0) = T(i-l,j) = T(2)
T 2,0 =0

i=2, j=1
l<3= T(2,1) = T(i-l)
T 2,1 =0

5.3. ALL PAIR SHORTEST PATH

Let G=<N,A> be a directed graph 'N' is a set of nodes and 'A' is the set of edges.

1.Each edge has an associated non-negative length.

2.We want to calculate the length of the shortest path between each pair of nodes.

3.Suppose the nodes of G are numbered from 1 to n, so N={1,2,...N},and suppose G matrix L gives the length of each edge, with L(i,j)=0 for i=1,2...n,L(i,j)>=for all i & j, and L(i,j)=infinity, if the edge (i,j) does not exist.

4.The principle of optimality applies: if k is the node on the shortest path from i to j then the part of the path from i to k and the part from k to j must also be optimal, that is shorter.

5.First, create a cost adjacency matrix for the given graph.

6.Copy the above matrix-to-matrix D, which will give the direct distance between nodes.

7.We have to perform N iteration after iteration k.the matrix D will give you the distance between nodes with only (1,2...,k)as intermediate nodes.

8.At the iteration k, we have to check for each pair of nodes (i,j) whether or not there exists a path from i to j passing through node k.

COST ADJACENCY MATRIX:

$$D0 = L = \begin{vmatrix} 0 & 5 & \propto & \propto \\ 50 & 0 & 15 & 5 \\ 30 & \propto & 0 & 15 \\ 15 & \propto & 5 & 0 \end{vmatrix}$$

$$\begin{vmatrix} 1 & 7\,5 \propto \propto \\ 2 & 7 \propto \propto 2 \\ 3 & \propto 3 \propto \propto \\ 4 & 4 \propto 1 \propto \end{vmatrix} \quad \begin{vmatrix} 11\,12 & - & - \\ 21 & - & -\,24 \\ - & 32 & - & - \\ 41 & - & 43 & - \end{vmatrix}$$

vertex 1:

$$\begin{vmatrix} 7 & 5 & \propto & \propto \\ 7 & \mathbf{12} & \propto & 2 \\ \propto & 3 & \propto & \propto \\ 4 & \mathbf{9} & 1 & \propto \end{vmatrix} \quad \begin{vmatrix} 11 & 12 & - & - \\ 21 & \mathbf{212} & - & 24 \\ - & 32 & - & - \\ 41 & \mathbf{412} & 43 & - \end{vmatrix}$$

vertex 2:

$$\begin{vmatrix} 7 & 5 & \propto & \mathbf{7} \\ 7 & 12 & \propto & 2 \\ \mathbf{10} & 3 & \propto & \mathbf{5} \\ 4 & 9 & 1 & \mathbf{11} \end{vmatrix} \quad \begin{vmatrix} 11 & 12 & - & \mathbf{124} \\ 21 & 212 & - & 24 \\ \mathbf{321} & 32 & - & \mathbf{324} \\ 41 & 412 & 43 & \mathbf{4124} \end{vmatrix}$$

vertex 3:

7	5	∝	7		11	12	-	124
7	12	∝	2		21	212	-	24
10	3	∝	5		321	32	-	324
4	**4**	1	**6**		41	**432**	43	**4324**

vertex 4:

7	5	**8**	7		11	12	**1243**	124
6	**6**	**3**	2		**241**	**2432**	**243**	24
9	3	**6**	5		**3241**	32	**3243**	324
4	4	1	6		41	432	43	4324

1. At 0^{th} iteration it nil give you the direct distances between any 2 nodes

$$D0 = \begin{vmatrix} 0 & 5 & \propto & \propto \\ 50 & 0 & 15 & 5 \\ 30 & \propto & 0 & 15 \\ 15 & \propto & 5 & 0 \end{vmatrix}$$

2. At 1^{st} iteration we have to check the each pair(i,j) whether there is a path through node 1.if so we have to check whether it is minimum than the previous value and if I is so than the distance through 1 is the value of d1(i,j).at the same time we have to solve the intermediate node in the matrix position p(i,j).

$$D1 = \begin{vmatrix} 0 & 5 & \propto & \propto \\ 50 & 0 & 15 & 5 \\ 30 & \mathbf{35} & 0 & 15 \\ 15 & \mathbf{20} & 5 & 0 \end{vmatrix} \quad \begin{matrix} \\ \\ p[3,2]= 1 \\ p[4,2]= 1 \\ \\ \end{matrix}$$

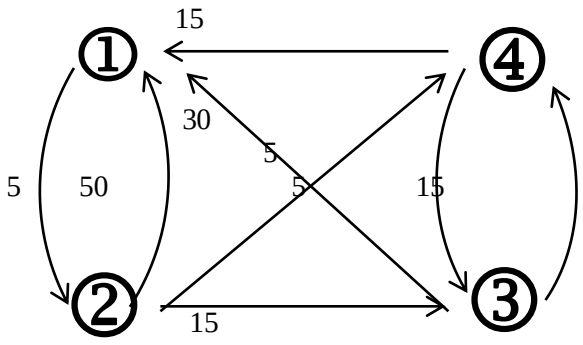


Fig: floyd's algorithm and work

3. likewise we have to find the value for N iteration (ie) for N nodes.

D2= | 0 5 **20** **10** | P[1,3] = 2
50 0 15 5 | P[1,4] = 2
30 35 0 15 |
15 20 5 0 |

D3= | 0 5 20 10 |
45 0 15 5 | P[2,1]=3
30 35 0 15 |
15 20 5 0 |

D4= | 0 5 **15** 10 |
20 0 **10** 5 | P[1,3]=4
30 35 0 15 | P[2,3]=4
15 20 5 0 |

4. D4 will give the shortest distance between any pair of nodes.
5. If you want the exact path then we have to refer the matrix p.The matrix will be,

P= | 0 0 4 2 |
3 0 4 0 | 0 ⟶ direct path
0 1 0 0 |
0 1 0 0 |

- Since,p[1,3]=4,the shortest path from 1 to3 passes through 4.
- Looking now at p[1,4]&p[4,3] we discover that between 1 & 4, we have to go to node 2 but that from 4 to 3 we proceed directly.
- Finally we see the trips from 1 to 2, & from 2 to 4, are also direct.
- The shortest path from 1 to 3 is 1,2,4,3.

ALGORITHM :

```
Function Floyd (L[1..r,1..r]):array[1..n,1..n]
array D[1..n,1..n]
 D = L
For   k = 1 to n do
For   i  = 1 to n do
For   j  = 1 to n do
D [ i , j ] = min (D[ i, j ], D[ i, k ] + D[ k, j ]
```

Return D

ANALYSIS:
This algorithm takes a time of $\theta(n^3)$

2.8.MULTISTAGE GRAPH

1. A multistage graph G = (V,E) is a directed graph in which the vertices are portioned into K > = 2 disjoint sets Vi, 1 <= i<= k.
2. In addition, if < u,v > is an edge in E, then u < = Vi and V $\sum$ Vi+1 for some i, 1<= i < k.
   If there will be only one vertex, then the sets Vi and Vk are such that [Vi]=[Vk] = 1.
3. Let 's' and 't' be the source and destination respectively.
4. The cost of a path from source (s) to destination (t) is the sum of the costs of the edger on the path.
5. The *MULTISTAGE GRAPH* problem is to find a minimum cost path from 's' to 't'.
6. Each set Vi defines a stage in the graph. Every path from 's' to 't' starts in stage-1, goes to stage-2 then to stage-3, then to stage-4, and so on, and terminates in stage-k.

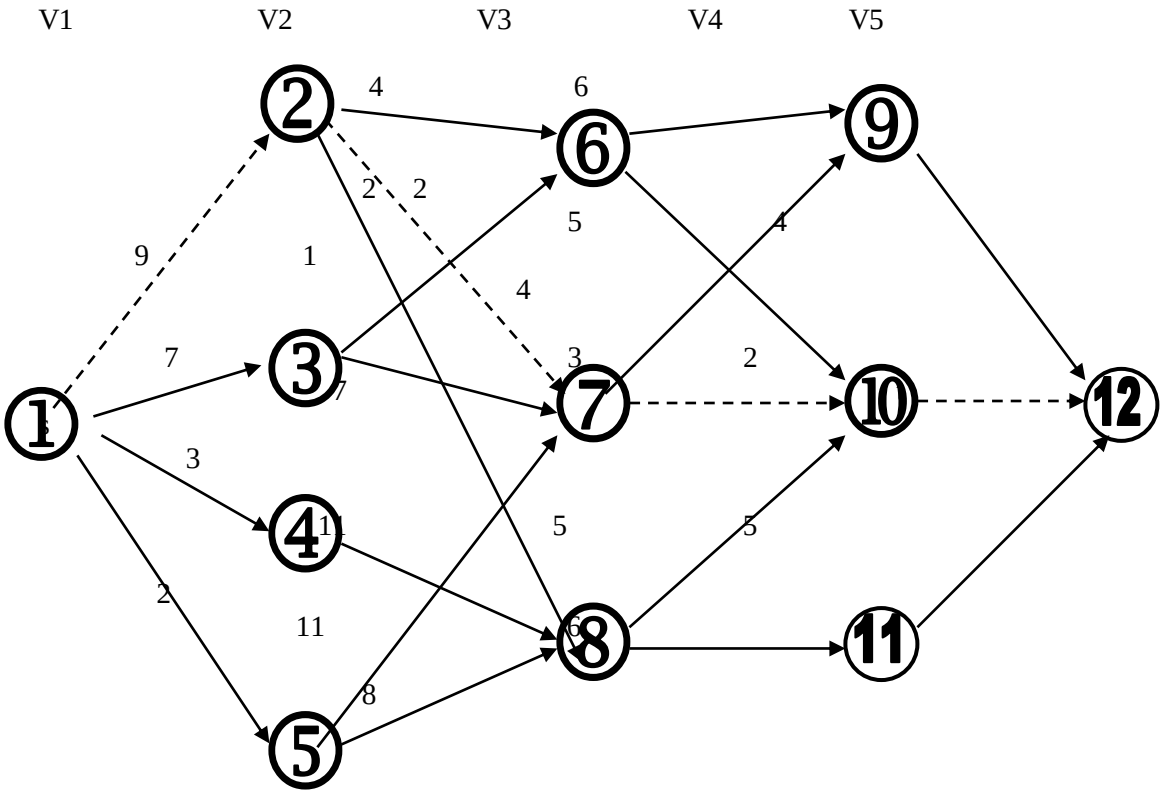
This *MULISTAGE GRAPH* problem can be solved in 2 ways.

- Forward Method.
- Backward Method.

2.8.1.FORWARD METHOD

Assume that there are 'k' stages in a graph.
In this *FORWARD* approach, we will find out the cost of each and every node starling from the 'k' th stage to the 1^{st} stage.
We will find out the path (i.e.) minimum cost path from source to the destination (ie) [Stage-1 to Stage-k].

PROCEDURE:



1. Maintain a cost matrix cost (n) which stores the distance from any vertex to the destination.
2. If a vertex is having more than one path, then we have to choose the minimum distance path and the intermediate vertex, which gives the minimum distance path, will be stored in the distance array 'D'.
3. In this way we will find out the minimum cost path from each and every vertex.
4. Finally cost(1) will give the shortest distance from source to destination.
5. For finding the path, start from vertex-1 then the distance array D(1) will give the minimum cost neighbour vertex which in turn give the next nearest vertex and proceed in this way till we reach the Destination.
6. For a 'k' stage graph, there will be 'k' vertex in the path.
7. In the above graph V1…V5 represent the stages. This 5 stage graph can be solved by using forward approach as follows,

STEPS: - **DESTINATION, D**

Cost (12)=0 ⟶ D (12)=0

Cost (11)=5 ⟶ D (11)=12

Cost (10)=2 ⟶D (10)=12

Cost (9)=4 ⟶D (9)=12

1. For forward approach,

$$\text{Cost}(i,j) = \min_{\substack{l \in Vi+1 \\ (j,l) \in E}} \{C(j,l) + \text{Cost}(i+1,l)\}$$

Cost(8) = min {C (8,10) + Cost (10), C (8,11) + Cost (11) }

= min (5 + 2, 6 + 5)

= min (7,11)

= 7

cost(8) =7 =>D(8)=10

cost(7) = min(c (7,9)+ cost(9),c (7,10)+ cost(10))

(4+4,3+2)

= min(8,5)

= 5

cost(7) = 5 =>D(7) = 10

cost(6) = min (c (6,9) + cost(9),c (6,10) +cost(10))

= min(6+4 , 5 +2)

= min(10,7)

= 7

cost(6) = 7 =>D(6) = 10

cost(5) = min (c (5,7) + cost(7),c (5,8) +cost(8))

= min(11+5 , 8 +7)

= min(16,15)

= 15

cost(5) = 15 =>D(5) = 18

cost(4) = min (c (4,8) + cost(8))

= min(11+7)

= 18

cost(4) = 18 =>D(4) = 8

cost(3) = min (c (3,6) + cost(6),c (3,7) +cost(7))

= min(2+7 , 7 +5)

= min(9,12)

= 9

cost(3) = 9 =>D(3) = 6

cost(2) = min (c (2,6) + cost(6),c (2,7) +cost(7) ,c (2,8) +cost(8))

= min(4+7 , 2+5 , 1+7)

= min(11,7,8)

= 7

cost(2) = 7 =>D(2) = 7

cost(1) = min (c (1,2)+cost(2) ,c (1,3)+cost(3) ,c (1,4)+cost(4) ,c(1,5)+cost(5))

= min(9+7 , 7 +9 , 3+18 , 2+15)

= min(16,16,21,17)

= 16

cost(1) = 16 =>D(1) = 2

⟶ The path through which you have to find the shortest distance.

(i.e.) ① ⟶ ② ⟶ ⑦ ⟶ ⑩ ⟶ ⑫

Start from vertex - 2

D (1) = 2

D (2) = 7

D (7) = 10

D (10) = 12

So, the minimum –cost path is,

① —9→ ② —2→ ⑦ —3→ ⑩ —2→ ⑫

∴ The cost is 9+2+3+2+=16

ALGORITHM: FORWARD METHOD

Algorithm FGraph (G,k,n,p)

// The I/p is a k-stage graph G=(V,E) with 'n' vertex.

// Indexed in order of stages E is a set of edges.

// and c[i,J] is the cost of<i,j>,p[1:k] is a minimum cost path.

{

```
    cost[n]=0.0;
     for j=n-1 to 1 step-1 do
  {
     //compute cost[j],
    //   let 'r' be the vertex such that <j,r> is an edge of 'G' &
   //   c[j,r]+cost[r] is minimum.

    cost[j] = c[j+r] + cost[r];
     d[j] =r;
   }
  // find a minimum cost path.

       P[1]=1;
       P[k]=n;
       For j=2 to k-1 do
       P[j]=d[p[j-1]];
}
```

ANALYSIS:

The time complexity of this forward method is O(V + E)

2.8.2.BACKWARD METHOD

if there one 'K' stages in a graph using back ward approach. we will find out the cost of each & every vertex starting from 1st
stage to the kth stage.

We will find out the minimum cost path from destination to source (ie)[from stage k to stage 1]

PROCEDURE:

- ❖ It is similar to forward approach, but differs only in two or three ways.
- ❖ Maintain a cost matrix to store the cost of every vertices and a distance matrix to store the minimum distance vertex.
- ❖ Find out the cost of each and every vertex starting from vertex 1 up to vertex k.
- ❖ To find out the path star from vertex 'k', then the distance array D (k) will give the minimum

cost neighbor vertex which in turn gives the next nearest neighbor vertex and proceed till we reach the destination.

STEP:

Cost(1) = 0 => D(1)=0

Cost(2) = 9 => D(2)=1

Cost(3) = 7 => D(3)=1

Cost(4) = 3 => D(4)=1

Cost(5) = 2 => D(5)=1

Cost(6) =min(c (2,6) + cost(2),c (3,6) + cost(3))

=min(13,9)

cost(6) = 9 =>D(6)=3

Cost(7) =min(c (3,7) + cost(3),c (5,7) + cost(5) ,c (2,7) + cost(2))

=min(14,13,11)

cost(7) = 11 =>D(7)=2

Cost(8) =min(c (2,8) + cost(2),c (4,8) + cost(4) ,c (5,8) +cost(5))

=min(10,14,10)

cost(8) = 10 =>D(8)=2

Cost(9) =min(c (6,9) + cost(6),c (7,9) + cost(7))

=min(15,15)

cost(9) = 15 =>D(9)=6

Cost(10)=min(c(6,10)+cost(6),c(7,10)+cost(7)),c (8,10)+cost(8)) =min(14,14,15)

cost(10)= 14 =>D(10)=6

Cost(11) =min(c (8,11) + cost(8))

cost(11) = 16 =>D(11)=8

cost(12)=min(c(9,12)+cost(9),c(10,12)+cost(10),c(11,12)+cost(11))

=min(19,16,21)

cost(12) = 16 =>D(12)=10

PATH:

Start from vertex-12

D(12) = 10

D(10) = 6

D(6) = 3

D(3) = 1

So the minimum cost path is,

1 →(7) 3 →(2) 6 →(5) 10 →(2) 12

The cost is 16.

ALGORITHM : BACKWARD METHOD

Algorithm BGraph (G,k,n,p)

```
//  The I/p is a k-stage graph G=(V,E) with 'n' vertex.
//  Indexed in order of stages E is a set of edges.
// and c[i,J] is the cost of<i,j>,p[1:k] is a minimum cost path.
{
     bcost[1]=0.0;
      for j=2 to n do
  {
     //compute bcost[j],
    //   let 'r' be the vertex such that <r,j> is an edge of 'G' &
   //   bcost[r]+c[r,j] is minimum.

   bcost[j] = bcost[r] + c[r,j];
    d[j] =r;
  }
 // find a minimum cost path.

       P[1]=1;
       P[k]=n;
       For j= k-1 to 2 do
       P[j]=d[p[j+1]];
}
```

5.4.TRAVELLING SALESMAN PROBLEM

❖ Let G(V,E) be a directed graph with edge cost c_{ij} is defined such that $c_{ij} > 0$ for all i and j and $c_{ij} = \infty$,if $<i,j> \notin E$.

Let $|V| = n$ and assume $n>1$.

❖ The traveling salesman problem is to find a tour of minimum cost.

❖ A tour of G is a directed cycle that include every vertex in V.

❖ The cost of the tour is the sum of cost of the edges on the tour.

❖ The tour is the shortest path that starts and ends at the same vertex (ie) 1.

APPLICATION :

➢ Suppose we have to route a postal van to pick up mail from the mail boxes located at 'n' different sites.

➢ An n+1 vertex graph can be used to represent the situation.

➢ One vertex represent the post office from which the postal van starts and return.

➢ Edge <i,j> is assigned a cost equal to the distance from site 'i' to site 'j'.

➢ the route taken by the postal van is a tour and we are finding a tour of minimum length.

➢ every tour consists of an edge <1,k> for some $k \in V-\{\}$ and a path from vertex k to vertex 1.

➢ the path from vertex k to vertex 1 goes through each vertex in V-{1,k} exactly once.

➢ the function which is used to find the path is

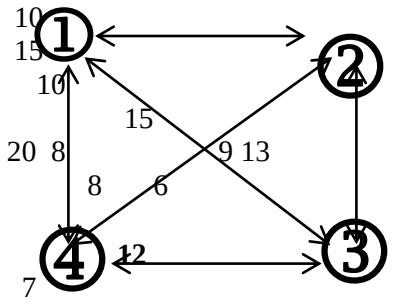
$$g(1,V-\{1\}) = \min\{ c_{ij} + g(j,s-\{j\})\}$$

➢ g(i,s) be the length of a shortest path starting at vertex i, going through all vertices in S,and terminating at vertex 1.

➢ the function g(1,v-{1}) is the length of an optimal tour.

STEPS TO FIND THE PATH:

1. Find $g(i,\Phi) = c_{i1}$, $1<=i<n$, hence we can use equation(2) to obtain g(i,s) for all s to size 1.
2. That we have to start with s=1,(ie) there will be only one vertex in set 's'.
3. Then s=2, and we have to proceed until $|s| <n-1$.
4. for example consider the graph.



Cost matrix

$$\begin{vmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{vmatrix}$$

g(i,s) → set of nodes/vertex have to visited.

↓ starting position

$g(i,s) = \min\{c_{ij} + g(j,s-\{j\})$

STEP 1:

$g(1,\{2,3,4\}) = \min\{c_{12}+g(2\{3,4\}), c_{13}+g(3,\{2,4\}), c_{14}+g(4,\{2,3\})\}$

$\min\{10+25, 15+25, 20+23\}$

$\min\{35,35,43\}$

$=35$

STEP 2:

$g(2,\{3,4\}) = \min\{c_{23}+g(3\{4\}), c_{24}+g(4,\{3\})\}$

$\min\{9+20, 10+15\}$

$\min\{29,25\}$

$=25$

$g(3,\{2,4\}) = \min\{c_{32}+g(2\{4\}), c_{34}+g(4,\{2\})\}$

$\min\{13+18, 12+13\}$

$\min\{31,25\}$

$=25$

$g(4,\{2,3\}) = \min\{c_{42}+g(2\{3\}), c_{43}+g(3,\{2\})\}$

$\min\{8+15, 9+18\}$

$\min\{23,27\}$

$=23$

STEP 3:

1. $g(3,\{4\}) = \min\{c_{34} + g\{4,\Phi\}\}$

   $12+8 = 20$

2. $g(4,\{3\}) = \min\{c_{43} + g\{3,\Phi\}\}$

   $9+6 = 15$

3. $g(2,\{4\}) = \min\{c_{24} + g\{4,\Phi\}\}$

   $10+8 = 18$

4. $g(4,\{2\}) = \min\{c_{42} + g\{2,\Phi\}\}$

   $8+5 = 13$

5. $g(2,\{3\}) = \min\{c_{23} + g\{3,\Phi\}\}$

   $9+6=15$

6. $g(3,\{2\}) = \min\{c_{32} + g\{2,\Phi\}\}$

   $13+5=18$

STEP 4:

$g\{4,\Phi\} = c_{41} = 8$

$g\{3,\Phi\} = c_{31} = 6$

$g\{2,\Phi\} = c_{21} = 5$

$\left| s \right| = 0.$

i =1 to n.

$g(1,\Phi) = c_{11} \Rightarrow 0$

$g(2,\Phi) = c_{21} \Rightarrow 5$

$g(3,\Phi) = c_{31} \Rightarrow 6$

$g(4,\Phi) = c_{41} \Rightarrow 8$

$\left| s \right| = 1$

i =2 to 4

$g(2,\{3\}) = c_{23} + g(3,\Phi)$

$= 9+6 = 15$

$g(2,\{4\}) = c_{24} + g(4,\Phi)$

$= 10+8 = 18$

$g(3,\{2\}) = c_{32} + g(2,\Phi)$

$= 13+5 = 18$

$g(3,\{4\}) = c_{34} + g(4,\Phi)$

$= 12+8 = 20$

$g(4,\{2\}) = c_{42} + g(2,\Phi)$

$= 8+5 = 13$

$g(4,\{3\}) = c_{43} + g(3,\Phi)$

$= 9+6 = 15$

$\left| s \right| = 2$

$i \neq 1,\ 1 \in s$ and $i \in s$.

$g(2,\{3,4\}) = \min\{c_{23}+g(3\{4\}),c_{24}+g(4,\{3\})\}$

$\min\{9+20,10+15\}$

$\min\{29,25\}$

$=25$

$g(3,\{2,4\}) = \min\{c_{32}+g(2\{4\}),c_{34}+g(4,\{2\})\}$

$\min\{13+18,12+13\}$

$\min\{31,25\}$

$=25$

$g(4,\{2,3\}) = \min\{c_{42}+g(2\{3\}),c_{43}+g(3,\{2\})\}$

$\min\{8+15,9+18\}$

$\min\{23,27\}$

$=23$

$\left| s \right| = 3$

$g(1,\{2,3,4\}) = \min\{c_{12}+g(2\{3,4\}),c_{13}+g(3,\{2,4\}),c_{14}+g(4,\{2,3\})\}$

$\min\{10+25,15+25,20+23\}$

$\min\{35,35,43\}$

$=35$

optimal cost is 35

the shortest path is,

g(1,{2,3,4}) = c_{12} + g(2,{3,4}) => 1->2

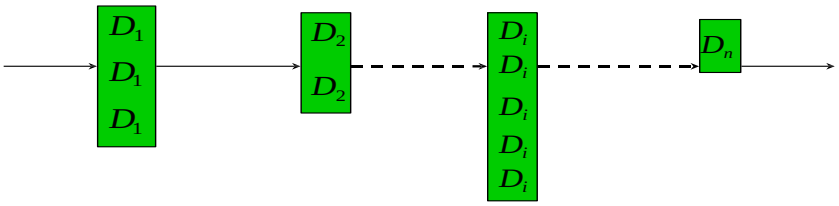
g(2,{3,4}) = c_{24} + g(4,{3}) => 1->2->4

g(4,{3}) = c_{43} + g(3{Φ}) => 1->2->4->3->1

so the optimal tour is **1 → 2 → 4 →3 → 1**

5.5 Reliability Design

- Input:
  - A system composed of several devices in serial
  - Each device (*D*) has a fixed reliability rating (*r*)
  - Multiple copies of the same device can be used in parallel to increase reliability
- Output:
  - A system with highest reliability rating subject to a cost constraint



m_i copies of devices D_i at stage i, r_i : *say*, 90%

with a reliability rating of $\Phi_i = 1-(1-r_i)^{m_i}$ ← Connected in parallel, At least one should work

$\max \prod_{1\le i\le n} \Phi_i(m_i)$ ← Connected in series, All of them have to work

subject to $\sum_{1\le i\le n} c_i m_i \le C$ and $m_i \ge 1, 1 \le i \le n$

- **Greedy method may not be applicable**
  - **Strategy to maximize reliability: Buy more less reliable units (Costs may be high)**
  - **Strategy to minimize cost: Buy more less expensive units (Reliability may not improve significantly)**
- **Divide-and-Conquer may fail**

Comparison

- A knapsack of capacity C
- Objects of size *ci* and profit *pi*
- Fill up the knapsack with 0 or 1 copy of *i*
- Maximize profit

- Total expenditure of C
- Stages of cost *ci* and reliability *ri*
- Construct a system with 1 or more copies of *i*
- Maximize reliability

$$u_i = 1 + \left\lfloor \frac{C - \sum_{j=1}^{n} c_j}{c_i} \right\rfloor$$

- How to build solutions recursively?
  - One stage and one device at a time
- How does principle of optimality apply?

$m_1 m_2 \cdots m_i \cdots m_n$

$(1, x \cdots x \cdots x) : (x \cdots x \cdots x)$ must be optimal for $C - c_1$

$(2, y \cdots y \cdots y) : (y \cdots y \cdots y)$ must be optimal for $C - 2 \times c_1$

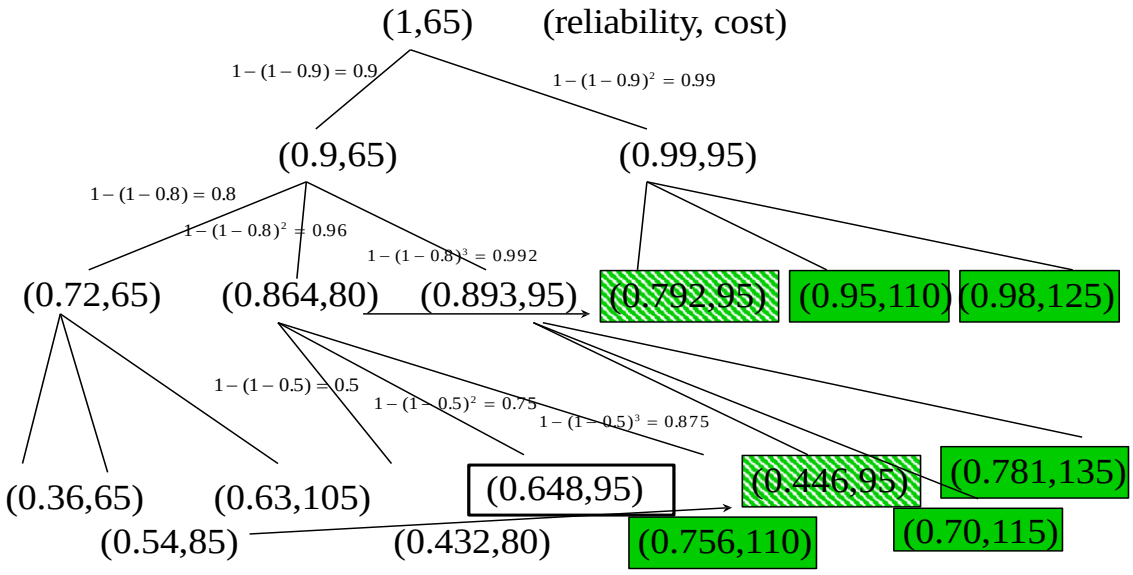
$\cdots$

$(u_1, y \cdots y \cdots y) : (y \cdots y \cdots y)$ must be optimal for $C - u_1 \times c_1$

- How to identify sub-optimal solutions?
  - if two solutions: (*m1*,...,*mi*,x,...,x), and (*n1*,...,*ni*,x,..., x) are such that one achieves higher reliability with a smaller cost, then the other cannot be optimal

- How to build table?

r=(0.9,0.8,0.5), c=(30,15,20),C=105



5.6 BACKTRACKING

- It is one of the most general algorithm design techniques.
- Many problems which deal with searching for a set of solutions or for a optimal solution satisfying some constraints can be solved using the backtracking formulation.
- To apply backtracking method, tne desired solution must be expressible as an n-tuple (x1…xn) where xi is chosen from some finite set Si.
- The problem is to find a vector, which maximizes or minimizes a criterion function P(x1….xn).
- The major advantage of this method is, once we know that a partial vector (x1,…xi) will not lead to an optimal solution that (m_{i+1}…….m_n) possible test vectors may be ignored entirely.
- Many problems solved using backtracking require that all the solutions satisfy a complex set of constraints.
- These constraints are classified as:

  i) Explicit constraints.

  ii) Implicit constraints.

- Explicit constraints:

  Explicit constraints are rules that restrict each Xi to take values only from a given set.

  Some examples are,

$Xi \geq 0$ or Si = {all non-negative real nos.}

Xi =0 or 1 or Si={0,1}.

$Li \leq Xi \leq Ui$ or Si= {a: $Li \leq a \leq Ui$}

- All tupules that satisfy the explicit constraint define a possible solution space for I.

- Implicit constraints:

  The implicit constraint determines which of the tuples in the solution space I can actually satisfy the criterion functions.

Algorithm:

Algorithm IBacktracking (n)

// This schema describes the backtracking procedure .All solutions are generated in X[1:n]

//and printed as soon as they are determined.

```
{
  k=1;
  While (k ≠ 0) do
```

```
    {
      if (there remains all untried
      X[k] ∈ T (X[1],[2],…..X[k-1]) and Bk (X[1],…..X[k])) is true ) then
      {
        if(X[1],……X[k] )is the path to the answer node)
       Then write(X[1:k]);
       k=k+1;              //consider the next step.
     }
  else k=k-1;                  //consider backtracking to the previous set.
 }
}
```

- All solutions are generated in X[1:n] and printed as soon as they are determined.
- T(X[1]…..X[k-1]) is all possible values of X[k] gives that X[1],……..X[k-1] have already been chosen.
- B_k(X[1]……X[k]) is a boundary function which determines the elements of X[k] which satisfies the implicit constraint.

Applications:

- N-Queens problem.
- Sum of subsets.
- Graph coloring.
- Hamiltonian cycle.

5.6.1 N-Queens problem:

This 8 queens problem is to place n-queens in an 'N*N' matrix in such a way that no two queens attack each otherwise no two queens should be in the same row, column, diagonal.

Solution:

- The solution vector X (X1…Xn) represents a solution in which Xi is the column of the th row where I th queen is placed.
- First, we have to check no two queens are in same row.
- Second, we have to check no two queens are in same column.
- The function, which is used to check these two conditions, is [I, X (j)], which gives position of the I th queen, where I represents the row and X (j) represents the column position.
- Third, we have to check no two queens are in it diagonal.

- Consider two dimensional array A[1:n,1:n] in which we observe that every element on the same diagonal that runs from upper left to lower right has the same value.
- Also, every element on the same diagonal that runs from lower right to upper left has the same value.
- Suppose two queens are in same position (i,j) and (k,l) then two queens lie on the same diagonal , if and only if |j-l|=|I-k|.

STEPS TO GENERATE THE SOLUTION:

- ❖ Initialize x array to zero and start by placing the first queen in k=1 in the first row.
- ❖ To find the column position start from value 1 to n, where 'n' is the no. Of columns or no. Of queens.
- ❖ If k=1 then x (k)=1.so (k,x(k)) will give the position of the k th queen. Here we have to check whether there is any queen in the same column or diagonal.
- ❖ For this considers the previous position, which had already, been found out. Check whether X (I)=X(k) for column |X(i)-X(k)|=(I-k) for the same diagonal.
- ❖ If any one of the conditions is true then return false indicating that k th queen can't be placed in position X (k).
- ❖ For not possible condition increment X (k) value by one and precede d until the position is found.
- ❖ If the position X (k) $\leq$ n and k=n then the solution is generated completely.
- ❖ If k<n, then increment the 'k' value and find position of the next queen.
- ❖ If the position X (k)>n then k th queen cannot be placed as the size of the matrix is 'N*N'.
- ❖ So decrement the 'k' value by one i.e. we have to back track and after the position of the previous queen.

Algorithm:

```
Algorithm place (k,I)
//return true if a queen can be placed in k th row and I th column. otherwise it returns //
//false .X[] is a global array whose first k-1 values have been set. Abs® returns the //absolute value
of r.
{
  For j=1 to k-1 do
    If ((X [j]=I)            //two in same column.
```

```
    Or (abs (X [j]-I)=Abs (j-k)))
Then return false;
Return true;
}
```

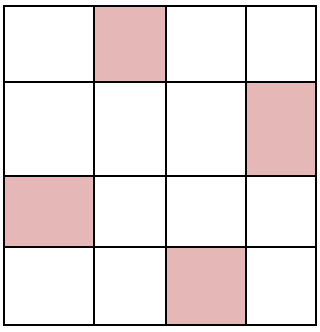
Algorithm Nqueen (k,n)

```
//using backtracking it prints all possible positions of n queens in 'n*n' chessboard. So
 //that they are non-tracking.
{
    For I=1 to n do
 {
   If place (k,I) then
     {
       X [k]=I;
        If (k=n) then write (X [1:n]);
         Else nquenns(k+1,n)    ;
      }
  }
}
```

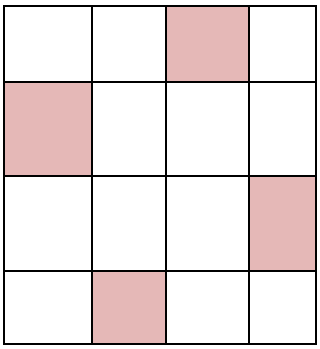
Example: 4 queens.

Two possible solutions are

Solutin-1
(2 4 1 3)



Solution 2
(3 1 4 2)



5.6.2 SUM OF SUBSETS:

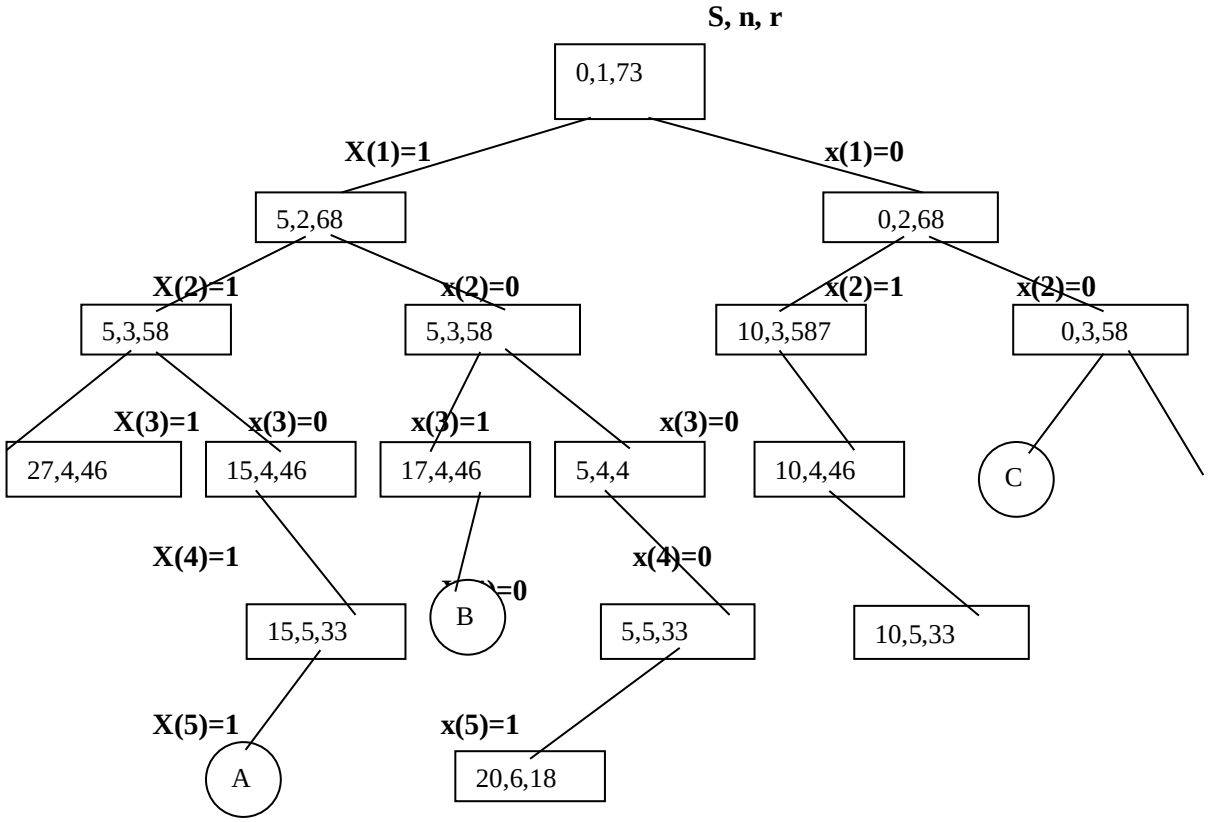
1) We are given 'n' positive numbers called weights and we have to find all combinations of these numbers whose sum is M. this is called sum of subsets problem.
2) If we consider backtracking procedure using fixed tuple strategy , the elements X(i) of the solution vector is either 1 or 0 depending on if the weight W(i) is included or not.
3) If the state space tree of the solution, for a node at level I, the left child corresponds to X(i)=1 and
4) right to X(i)=0.

Example:

a. Given n=6,M=30 and W(1…6)=(5,10,12,13,15,18).We have to generate all possible combinations of subsets whose sum is equal to the given value M=30.

b. In state space tree of the solution the rectangular node lists the values of s, k, r, where s is the sum of subsets,'k' is the iteration and 'r' is the sum of elements after 'k' in the original set.

c. The state space tree for the given problem is,



Ist solution is A -> 1 1 0 0 1 0

IInd solution is B -> 1 0 1 1 0 0

III rd solution is C -> 0 0 1 0 0 1

In the state space tree, edges from level 'i' nodes to 'i+1' nodes are labeled with the values of Xi, which is either 0 or 1.

The left sub tree of the root defines all subsets containing Wi.

The right subtree of the root defines all subsets, which does not include Wi.

3.5.GENERATION OF STATE SPACE TREE:

Maintain an array X to represent all elements in the set.

The value of Xi indicates whether the weight Wi is included or not.

Sum is initialized to 0 i.e., s=0.

We have to check starting from the first node.

Assign X(k)<- 1.

- If S+X(k)=M then we print the subset b'coz the sum is the required output.
- If the above condition is not satisfied then we have to check S+X(k)+W(k+1)<=M. If so, we have to generate the left sub tree. It means W(t) can be included so the sum will be incremented and we have to check for the next k.
- After generating the left sub tree we have to generate the right sub tree, for this we have to check S+W(k+1)<=M.B'coz W(k) is omitted and W(k+1) has to be selected.
- Repeat the process and find all the possible combinations of the subset.

Algorithm:

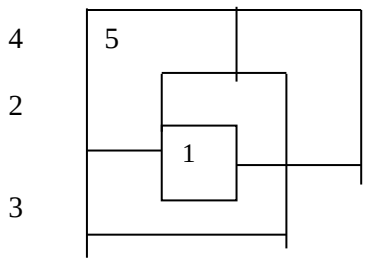
```
Algorithm sumofsubset(s,k,r)
{
//generate the left child. note s+w(k)<=M since Bk-1 is true.
X{k]=1;
If (S+W[k]=m) then write(X[1:k]); // there is no recursive call here as W[j]>0,1<=j<=n.
Else if (S+W[k]+W[k+1]<=m) then sum of sub (S+W[k], k+1,r- W[k]);
//generate right child and evaluate Bk.
```

```
If ((S+ r- W[k]>=m)and(S+ W[k+1]<=m)) then
{
  X{k]=0;
  sum of sub (S, k+1, r- W[k]);
}
}
```

5.6.3 GRAPH COLORING:

- Let 'G' be a graph and 'm' be a given positive integer. If the nodes of 'G' can be colored in such a way that no two adjacent nodes have the same color. Yet only 'M' colors are used. So it's called M-color ability decision problem.
- The graph G can be colored using the smallest integer 'm'. This integer is referred to as chromatic number of the graph.
- A graph is said to be planar iff it can be drawn on plane in such a way that no two edges cross each other.
- Suppose we are given a map then, we have to convert it into planar. Consider each and every region as a node. If two regions are adjacent then the corresponding nodes are joined by an edge.

Consider a map with five regions and its graph.



1 is adjacent to 2, 3, 4.

2 is adjacent to 1, 3, 4, 5

3 is adjacent to 1, 2, 4

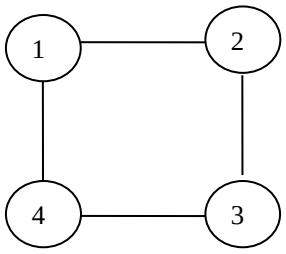
4 is adjacent to 1, 2, 3, 55 is adjacent to 2, 4

Steps to color the Graph:

1. First create the adjacency matrix graph(1:m,1:n) for a graph, if there is an edge between i,j then C(i,j) = 1 otherwise C(i,j) =0.
2. The Colors will be represented by the integers 1,2,…..m and the solutions will be stored in the array X(1),X(2),……..,X(n) ,X(index) is the color, index is the node.
3. He formula is used to set the color is,
   X(k) = (X(k)+1) % (m+1)
4. First one chromatic number is assigned ,after assigning a number for 'k' node, we have to check whether the adjacent nodes has got the same values if so then we have to assign the next value.
5. Repeat the procedure until all possible combinations of colors are found.
6. The function which is used to check the adjacent nodes and same color is,

If((Graph (k,j) == 1) and X(k) = X(j))

Example:



N= 4

M= 3

Adjacency Matrix:

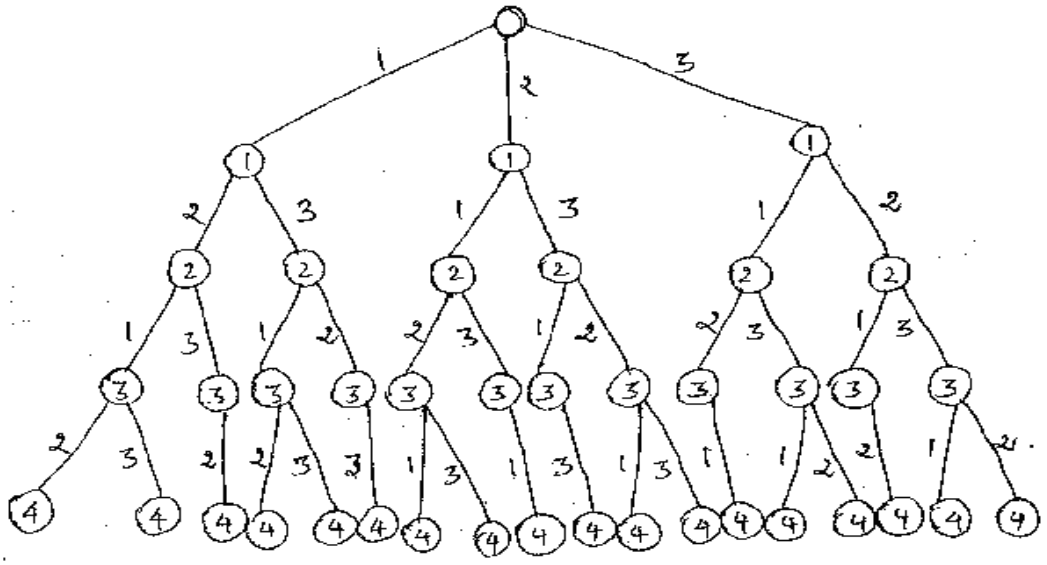
0	1	0	1
1	0	1	0
0	1	0	1
1	0	1	0

→ Problem is to color the given graph of 4 nodes using 3 colors.

→Node-1 can take the given graph of 4 nodes using 3 colors.

→ The state space tree will give all possible colors in that ,the numbers which are inside the circles are nodes ,and the branch with a number is the colors of the nodes.

State Space Tree:



Algorithm:

Algorithm mColoring(k)

// the graph is represented by its Boolean adjacency matrix G[1:n,1:n] .All assignments //of 1,2,…….,m to the vertices of the graph such that adjacent vertices are assigned //distinct integers are printed. 'k' is the index of the next vertex to color.

```
{
repeat
{
   // generate all legal assignment for X[k].
  Nextvalue(k);    // Assign to X[k] a legal color.
       If (X[k]=0) then return;          // No new color possible.
      If (k=n) then                  // Almost 'm' colors have been used to color the 'n' vertices
              Write(x[1:n]);
    Else mcoloring(k+1);
}until(false);
}
```

Algorithm Nextvalue(k)

// X[1],……X[k-1] have been assigned integer values in the range[1,m] such that //adjacent values have distinct integers. A value for X[k] is determined in the //range[0,m].X[k] is assigned the next

highest numbers color while maintaining //distinctness form the adjacent vertices of vertex K. If no such color exists, then X[k] is 0.

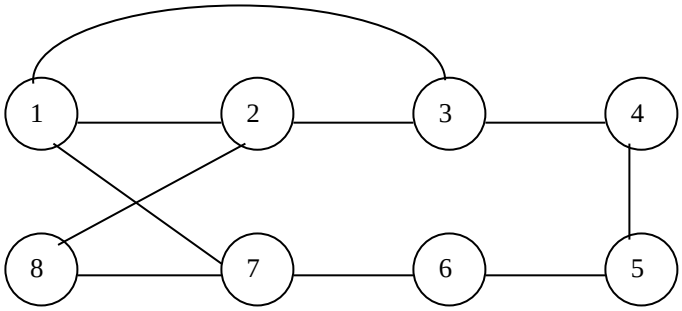
```
{
  repeat
   {
        X[k] = (X[k]+1)mod(m+1);    // next highest color.
       If(X[k]=0) then return;          //All colors have been used.
         For j=1 to n do
         {
           // Check if this color is distinct from adjacent color.
         If((G[k,j] ≠ 0)and(X[k] = X[j]))
           // If (k,j) is an edge and if adjacent vertices have the same color.
         Then break;
         }
      if(j=n+1) then return;     //new color found.
   } until(false);    //otherwise try to find another color.
}
```

→ The time spent by Nextvalue to determine the children is θ(mn)

→Total time is = θ(m^{n} n**).**

5.6.4 HAMILTONIAN CYCLES:

- Let G=(V,E) be a connected graph with 'n' vertices. A HAMILTONIAN CYCLE is a round trip path along 'n' edges of G which every vertex once and returns to its starting position.
- If the Hamiltonian cycle begins at some vertex V1 belongs to G and the vertex are visited in the order of V1,V2…….Vn+1,then the edges are in E,1<=I<=n and the Vi are distinct except V1 and Vn+1 which are equal.
- Consider an example graph G1.



The graph G1 has Hamiltonian cycles:

->1,3,4,5,6,7,8,2,1 and

->1,2,8,7,6,5,4,3,1.

- The backtracking algorithm helps to find Hamiltonian cycle for any type of graph.

Procedure:

- ❖ Define a solution vector X(Xi……..Xn) where Xi represents the I th visited vertex of the proposed cycle.
- ❖ Create a cost adjacency matrix for the given graph.
- ❖ The solution array initialized to all zeros except X(1)=1,b'coz the cycle should start at vertex '1'.
- ❖ Now we have to find the second vertex to be visited in the cycle.
- ❖ The vertex from 1 to n are included in the cycle one by one by checking 2 conditions,

1.There should be a path from previous visited vertex to current vertex.

2.The current vertex must be distinct and should not have been visited earlier.

6. When these two conditions are satisfied the current vertex is included in the cycle, else the next vertex is tried.
7. When the nth vertex is visited we have to check, is there any path from nth vertex to first vertex. if no path, the go back one step and after the previous visited node.
8. Repeat the above steps to generate possible Hamiltonian cycle.

Algorithm:(Finding all Hamiltonian cycle)

```
Algorithm Hamiltonian (k)
{
 Loop
        Next value (k)
```

```
If (x (k)=0) then return;
{
  If k=n then
     Print (x)
Else
Hamiltonian (k+1);
End if
}
Repeat
}
```

Algorithm Nextvalue (k)

```
{
 Repeat
{
  X [k]=(X [k]+1) mod (n+1); //next vertex
  If (X [k]=0) then return;
  If (G [X [k-1], X [k]] ≠ 0) then
{
  For j=1 to k-1 do if (X [j]=X [k]) then break;
  // Check for distinction.
  If (j=k) then          //if true then the vertex is distinct.
    If ((k<n) or ((k=n) and G [X [n], X [1]] ≠ 0)) then return;
}
} Until (false);
}
```

5.7.NP-HARD AND NP-COMPLETE PROBLEMS:

5.7.1Basic concepts:

→ **Tractability:** Some problems are *tractable*: that is the problems are solvable in reasonable amount of time called polynomial *time.* Some problems are *intractable*: that is as problem grow large, we are unable to solve them in reasonable amount of time called polynomial *time.*

→**Polynomial Time Complexity:** An algorithm is of **Polynomial Complexity,** if there exists a polynomial p() such that the computing time is O(p(n)) for every input size of 'n'. Polynomial time is the worst-case running time required to an algorithm to process an input of size *n* the is O(n^k) for some constant *k*

→ Polynomial time: O(n^2), O(n^3), O(n log n)

→ Not in polynomial time: O(2^n), O(n^n), O(*n*!) →Exponential Time

→Most problems that do not yield polynomial-time algorithms are either optimization or decision problems.

Decision Problems: Computational problem with produces output of "yes" or "no", 1 or 0 are decision problems.

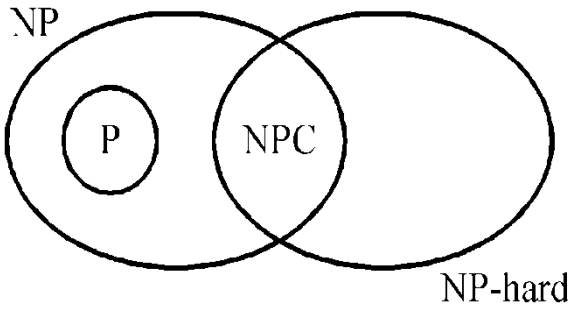
Examples: 1. Path in a graph

2. Minimum Spanning Tree whose cost is less than some value w.

Optimization Problems: Computational problem where we try to maximize or minimize some value that is identifying optimal solution to problem

Examples: 1. Shortest-path in a graph.

2. Minimum Spanning Tree



5.7.2.CLASS P PROBLEMS:

- Class P problems are the set of decision problems solvable by deterministic algorithms in polynomial-time.
- A deterministic algorithm is (essentially) one that always computes the correct answer
- **Examples:** Fractional Knapsack, MST, Single-source shortest path

5.7.3.CLASS NP PROBLEMS:

- NP problems are set of decision problems solvable by non-deterministic algorithms in polynomial-time.
- A nondeterministic algorithm is one that can "guess" the right answer or solution
- **Examples:** Hamiltonian Cycle (Traveling Sales Person), Conjunctive Normal Form (CNF)

5.8. NP-Complete Problems:

- A problem 'x' is a NP class problem and also NP-Complete if and only if every other problem

in NP can be reducible (solvable) using non-deterministic algorithm in polynomial time.

- The class of problems which are NP-hard and belong to NP.
- The NP-Complete problems are always decision problems only.
- **Example :** TSP, Vertex covering problem

Examples of NP-complete problems:

- Packing problems: SET-PACKING, INDEPENDENT-SET.
- Covering problems: SET-COVER, VERTEX-COVER.
- Sequencing problems: HAMILTONIAN-CYCLE, TSP.
- Partitioning problems: 3-COLOR, CLIQUE.
- Constraint satisfaction problems: SAT, 3-SAT.

Numerical problems: SUBSET-SUM, PARTITION, KNAPSACK

5.9.NP-Hard Problems:

- A problem 'x' is a NP class problem and also NP-Hard if and only if every other problem in NP can be reducible (solvable) using non-deterministic algorithm in exponential time.
- The class of problems to which every NP problem reduces.
- The NP-Hard problems are decision problems and sometimes may be optimization problems.
- **Example :** Integer Linear Programming.

Nondeterministic Algorithms:

Deterministic Algorithms:

- Let A be an algorithm to solve problem P. A is called deterministic if it has only one choice in each step throughout its execution. Even if we run algorithm A again and again, there is no change in output.
- Deterministic algorithms are identified with uniquely defined results in terms of output for a certain input.

Nondeterministic Algorithms:

- Let A be a nondeterministic algorithm for a problem P. We say that algorithm A accepts an instance of P if and only if, there exists a guess that leads to a yes answer.
- In non deterministic algorithms, there is no uniquely defined result in terms of output for a

certain input.

- Nondeterministic algorithms are allowed to contain operations whose outcomes are limited to a given set of instances of P, instead of being uniquely defined results.
- A Non-deterministic algorithm A on input x consists of two phases:

– **Guessing:** An arbitrary "string of characters" is generated in polynomial time. It may

→Correspond to a solution →Not correspond to a solution

→Not be in proper format of a solution →Differ from one run to another

– **Verification:** A deterministic algorithm verifies

→The generated "string of characters" is in proper format

→Whether it is a solution in polynomial time

- The Nondeterministic algorithm uses three basic procedures, whose time complexity is O(1).

1. CHOICE(1,*n*) or CHOICE(*S*) : This procedure chooses and returns an arbitrary element, in favor of the algorithm, from the closed interval [1,*n*] or from the set *S*.

2. SUCCESS : This procedure declares a successful completion of the algorithm.

3. FAILURE : This procedure declares an unsuccessful termination of the algorithm.

- Non deterministic algorithm terminates unsuccessfully if and only if there is no set of choices leading to successful completion of algorithm
- Non deterministic algorithm terminates successfully if and only if there exists set of choices leading to successful completion of algorithm

Nondeterministic Search Algorithm: The following algorithm enables nondeterministic search of x in an unordered array A with n elements. It determines an index j such that A[j] = x or j = −1 if x does not belongs to A.

```
Algorithm nd_search ( A, n, x )
{
int j = choice ( 0, n-1 );
if ( A[j] == x )
{
cout << j;
success();
}
cout << -1;
failure();
}
```

By the definition of nondeterministic algorithm, the output is -1 iff there is no j such that A[j] = x Since A is not ordered, every deterministic search algorithm is of complexity O(n), whereas the nondeterministic algorithm has the complexity as O(1).

Nondeterministic Sort Algorithm: The following algorithm sorts 'n' positive integers in non-decreasing order and produces output in sorted order. The array B[] is an auxiliary array initialized to 0 and is used for convenience.

```
Algorithm nd_sort ( A, n )
{
for ( i = 0; i < n; B[i++] = 0; );
for ( i = 0; i < n; i++ )
{
j = choice ( 0, n - 1 );
if ( B[j] != 0 ) failure();
B[j] = A[i];
}
// Verify order
for ( i = 0; i < n-1; i++ )
if ( B[i] > B[i+1] ) failure();
write ( B );
success();
}
```

The time complexity of nd_sort is O(n). Best-known deterministic sorting algorithm like binary search has a complexity of (n log n).

5.10 Satisfiability: (SAT Problem)

→ Let x1, x2 . . . denote a set of Boolean variables and xi denote the complement of $\bar{x}_i$.

→A variable or its complement is called a literal

→A formula in propositional calculus is an expression that is constructed by connecting literals using the operations and ($\wedge$) & or ($\vee$)

→Examples of formulas in propositional calculus

$(x_1 \wedge x_2) \vee (x_3 \wedge \bar{x}_4)$

$(x_3 \vee \bar{x}_4) \wedge (x_1 \vee \bar{x}_2)$

→Conjunctive normal form (CNF): A Boolean formula is said to be in conjunctive normal form (CNF) if it is the conjunction of formulas.

Example: $(x_1 \vee \bar{x}_2) \wedge (\bar{x}_1 \vee x_5)$

→Disjunctive normal form (DNF) : A Boolean formula is said to be in disjunctive normal form (CNF) if it is the disjunction of formulas.

Example: $(x_1 \wedge \bar{x}_2) \vee (x_1 \wedge \bar{x}_5)$

→ **Satisfiability problem** is to determine whether a formula is true for some assignment of truth values to the variables

→ CNF--satisfiability is the satisfiability problem for CNF formulas

→ DNF--satisfiability is the satisfiability problem for DNF formulas

→ Polynomial time nondeterministic algorithm that terminates successfully iff a given propositional formula E(x1, . . . , xn) is satisfiable

→ Non deterministically choose one of the 2^n possible assignments of truth values to (x1, . . . , xn) and verify that E(x1, . . . , xn) is true for that assignment

Algorithm eval (E, n)

```
{
// Determine whether the propositional formula E is satisfiable.
Here variable are x1, x2, ..., xn
 for ( i = 1; i <= n; i++ )
x(i) = choice ( true, false );
if ( E ( x1, ..., xn ) )
success();
else
failure();
}
```

→The nondeterministic time to choose the truth value is O(n)

→The deterministic evaluation of the assignment is also done in O(n) time

Decision Problem Vs Optimization Problem:

Decision Problem and Algorithm : Any problem for which the answer is either zero or one is called a decision problem. An algorithm for a decision problem is termed a decision algorithm.

- A decision algorithm will output 0 or 1
- Implicit in the signals success() and failure()
- Output from a decision algorithm is uniquely defined by input parameters and algorithm specification.

Optimization Problem and Algorithm: Any problem that involves the identification of an optimal (either minimum or maximum) value of a given cost function is known as an optimization problem. An optimization algorithm is used to solve an optimization problem.

- An optimization problem may have many feasible solutions
- The problem is to find out the feasible solution with the best associated value
- NP-completeness applies directly not to optimization problems but to decision problems.

Casting (Conversion) of Optimization Problem into Decision Problem:

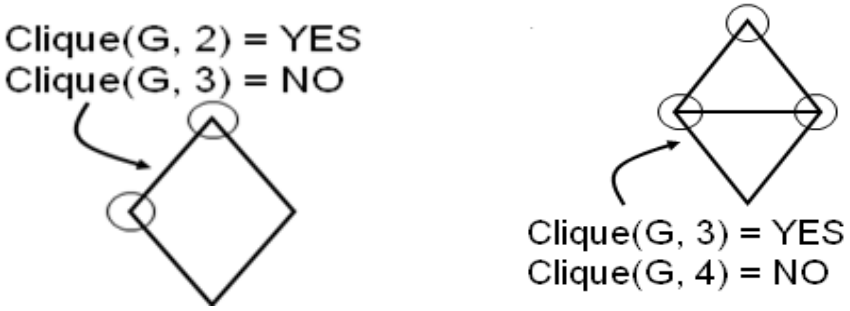
Optimization problems can be cast into decision problems by imposing a bound on output or solution. Decision problem is assumed to be easier (or no harder) to solve compared to the optimization problem. Decision problem can be solved in polynomial time if and only if the corresponding optimization problem can be solved in polynomial time. If the decision problem cannot be solved in polynomial time, the optimization problem cannot be solved in polynomial time either.

For example consider, shortest path problem. Optimization problem is to find a shortest path between two vertices in an undirected weighted graph, so that shortest path consists least number of edges. Whereas the decision problem is to determine that given an integer k, whether a path exists between two specified nodes consisting of at most k edges.

Maximal Clique:

Clique is a maximal complete sub-graph of a graph G = (V,E), that is a subset of vertices in V all connected to each other by edges in E (i.e., forming a complete graph).

Example:



The **Size of a clique** is the number of vertices in it. The Maximal clique problem is an optimization problem that has to determine the size of a largest clique in G. A decision problem is to determine whether G has a clique of size at least 'k'.

Input for Maximal clique problem: Input can be provided as a sequence of edges. Each edge in E(G) is a pair of vertices (i, j) .The size of input for each edge (i, j) in binary representation is

$$\lfloor \log_2 i \rfloor + \lfloor \log_2 j \rfloor + 2$$

And input size of any instance is given by

$$n = \sum_{\substack{(i,j) \in E(G) \\ i < j}} (\lfloor \log_2 i \rfloor + \lfloor \log_2 j \rfloor + 2) + \lfloor \log_2 k \rfloor + 1$$

Where 'k' is the number to indicate the clique size

Maximal clique problem as Decision Problem:

Let us denote the deterministic decision algorithm for the clique decision problem as dclique(G, k). If |V | = n, then the size of a maximal clique can be found by

for (k = n; dclique (G, k) != 1; k--);

If time complexity of dclique is f(n), size of maximal clique can be found in time

g(n) <= n.f(n). Therefore, the decision problem can be solved in time g(n)

Note that Maximal clique problem can be solved in polynomial time if and only if the clique decision problem can be solved in polynomial time.

Non deterministic Clique Algorithm:

```
    Algorithm DCK(G, n, k)
{
    S=0;  // Empty set.
    for  i=1 to k  do
    {
            t = Choice(1,n);
            if t ε S then Failure();
            S= SU{ t };
    }
    for all pairs (i,j) such that i ε S, j ε S and i!=j do
            if (i, j) is not an edge of G then Failure();
    Success();
}
```

N**on-deterministic knapsack problem:**

- It is a non-deterministic polynomial time complexity algorithm.
- The for loop selects or discards each of the n items It also re-computes the total weight and profit corresponding to the selection
- The if statement checks to see the feasibility of assignment and whether the profit is above a lower bound r
- The time complexity of the algorithm is O(n) . If the input length is q in binary, then O(q).

Algorithm nd_knapsack (p, w, n, m, r, x)

```
{
W = 0; P = 0;
for ( i = 1; i <= n; i++ )
{
x[i] = choice ( 0, 1 );
W += x[i] * w[i];
```

```
P += x[i] * p[i];
}
if ( ( W > m ) || ( P < r ) )
failure();
else
success();
}
```

5.11.SUM OF SUBSETS PROBLEM:

- Bits are numbered from 0 to m from right to left
- Bit i will be 0 if and only if no subsets of A[j], 1 _ j _ n sums to i
- Bit 0 is always 1 and bits are numbered 0, 1, 2, . . . ,m right to left
- Number of steps for this algorithm is O(n)
- Each step moves m + 1 bits of data and would take O(m) time on a conventional computer
- Assuming one unit of time for each basic operation for a fixed word size, the complexity of deterministic algorithm is O(nm)
- Consider the deterministic decision algorithm to get sum of subsets

Algorithm Sum_of_subsets (A, n, m)

```
{
// A[n] is an array of integers
s = 1 // s is an m+1 bit word
// bit 0 is always 1
for i = 1 to n
s |= ( s << A[i] ) // shift s left by A[i] bits
if bit m in s is 1
write ( "A subset sums to m" );
else
write ( "No subset sums to m" );
}
```

5.12.COOK'S THEOREM:

- We know that, Class **P** problems are the set of all decision problems solvable by deterministic algorithms in polynomial time. Similarly **Class NP** problems are set of all decision problems solvable by nondeterministic algorithms in polynomial time.
- Since deterministic algorithms are a special case of nondeterministic algorithms, P NP
- Cook formulated the following question: Is there any single problem in NP such that if we showed it to be in P, then that would imply that P = NP? This led to Cook's theorem as :

Satisfiability is in P if and only if P = NP.

Cook's Theorem Proof:

Consider Z - denotes a deterministic polynomial algorithm

A - denotes a non- deterministic polynomial algorithm

I - denotes input instance of algorithm

n - denotes length of input instance

Q - denotes a formulae

m - denotes length of formulae

→ Now, the formula 'Q' is satisfiable if and only if the non-deterministic algorithm 'A' has a successful termination with input 'I'.

→ If the time complexity of 'A' is p(n) for some polynomial p(), then the time needed to construct the formula 'Q' by algorithm 'A' is given by $O(p^3(n)\log n)$.

Therefore complexity of non-deterministic algorithm 'A' is $O(p^3(n)\log n)$. (NP)

→ Similarly, the formula 'Q' is satisfiable if and only if the deterministic algorithm 'Z' has a successful termination with input 'I'.

→ If the time complexity of 'Z' is q(m) for some polynomial q(), then the time needed to construct the formula 'Q' by algorithm 'Z' is given by $O(p^3(n)\log n + q(p^3(n)\log n))$.

Therefore complexity of deterministic algorithm 'Z' is $O(p^3(n)\log n + q(p^3(n)\log n))$. (P)

→ If satisfiability is in P, then q(m) is a polynomial function and the complexity of 'Z' becomes O(r(n)) for some polynomial r(n).

→ Hence, P is satisfiable, then for every non-deterministic algorithm 'A' in NP can obtain a deterministic algorithm 'Z' in P.

→ So, the above construction shows that "if satisfiability is in P, then P=NP"

PART - A(2 Marks)

1. Write the general procedure of dynamic programming.

Ans: The development of dynamic programming algorithm can be broken into a sequence of 4 steps.

- Characterize the structure of an optimal solution.
- Recursively define the value of the optimal solution.
- Compute the value of an optimal solution in the bottom-up fashion.
- Construct an optimal solution from the computed information.

2. Define principle of optimality.

Ans: It states that an optimal sequence of decisions has the property that whenever the initial stage or decisions must constitute an optimal sequence with regard to stage resulting from the first decision.

3. Write the difference between the Greedy method and Dynamic programming.

Greedy method

1. Only one sequence of decision is generated.
2. It does not guarantee to give an optimal solution always.

Dynamic programming

1. Many number of decisions are generated.
2. It definitely gives an optimal solution always.

4. What are the drawbacks of dynamic programming?

Ans: Time and space requirements are high, since storage is needed for all level.

Optimality should be checked at all levels.

5. What are the features of dynamic programming?

Optimal solutions to sub problems are retained so as to avoid recomputing their values.

Decision sequences containing subsequences that are sub optimal are not considered.

It definitely gives the optimal solution always.

6. What is meant by n-queen Problem?

Ans: The problem is to place n queens on an n-by-n chessboard so that no two queens attack each

other by being in the same row or in the same column or in the same diagonal.

7. Define Backtracking

Ans: Backtracking is used to solve problems with tree structures. Even problems seemingly remote to trees such as a walking a maze are actually trees when the decision \'back-left-straight-right\' is considered a node in a tree. The principle idea is to construct solutions one component at a time and evaluate such partially constructed candidates

8. What is the Aim of Backtracking?

Ans:Backtracking is the approach to find a path in a tree. There are several different aims to be achieved : • just a path • all paths • the shortest path.

9. Define the Implementation considerations of Backtracking?

Ans: The implementation bases on recursion. Each step has to be reversible; hence the state has to be saved somehow. There are two approaches to save the state: • As full state on the stack • As reversible action on the stack

10. List out the implementation procedure of Backtracking

Ans: As usual in a recursion, the recursive function has to contain all the knowledge. The standard implementaion is :

1. check if the goal is achieved REPEAT
2. check if the next step is possible at all
3. check if the next step leads to a known position - prevent circles
4. do this next step UNTIL (the goal is achieved) or (this position failed) .

11. Define Subset-Sum Problem?

Ans:This problem find a subset of a given set S={s1,s2,……,sn} of n positive integers whose sum is equal to a given positive integer d.

12. Define Traveling Salesman Problem?

Ans: Given a complete undirected graph G=(V, E) that has nonnegative integer cost c(u, v) associated with each edge (u, v) in E, the problem is to find a hamiltonian cycle (tour) of G with minimum cost.

13. Define Knapsack Problem

Ans: Given n items of known weight wi and values vi=1,2,..,n and a knapsack of capacity w, find the most valuable subset of the items that fit in the knapsack.

14. . What is a state space tree?

Ans:The processing of backtracking is implemented by constructing a tree of choices being made. This is called the state-space tree. Its root represents a initial state before the search for a solution begins. The nodes of the first level in the tree represent the choices made for the first component of the solution, the nodes in the second level represent the choices for the second component and so on.

15.Define nondeterministic Polynomial

Ans: Class NP is the class of decision problems that can be solved by nondeterministic Polynomial algorithms. This class of problems is called nondeterministic Polynomial.

16. Define NP-Complete

Ans: An NP-Complete problem is a problem in NP that is as difficult as any other problem in this class because any other problem in NP can be reduced to it in Polynomial time.

17.Define Polynomial reducible

Ans: A Decision problem D1 is said to be polynomial reducible to a decision problem D2 if there exists a function t that transforms instances of D2 such that o T maps all yes instances of D1 to yes instances of D2 and all noninstances of D1 to no instance of D2 o T is computable by a Polynomial-time algorithm

18.What is the difference between tractable and intractable?

Problems that can be solved in polynomial time are called tractable and the problems that cannot be solved in Polynomial time are called intractable.

19.Define undecidable Problem

Some decision problem that cannot be solved at all by any algorithm is called undecidable algorithm.

20 .Define Heuristic Generally speaking, a heuristic is a "rule of thumb," or a good guide to follow when making decisions.

In computer science, a heuristic has a similar meaning, but refers specifically to algorithms.

PART - B (10 Marks)

1) Write an algorithm to estimate the efficiency of backtracking.
2) Explain 4-queen problem using backtracking.
3) How many solutions are there to the eight queens problem? How many distinct solutions are there if we do not distinguish solution that can be transformed into one another by rotations and reflections?
4) Explain about graph coloring and Hamiltonian cycles with examples.
5) Explain Sum of subsets problem with an example.
6) Explain traveling sales person problem?
7) Explain about optimal binary search tress?
8) Take an example problem and solve All-Pairs shortest path?
9) Explain and give an example for single source shortest path?
10) Prove that any two NP complete problems are polynomial turning equivalent.
11) Give brief description about the cook's theorem and prove with example.
12) Give out the relation between NP hard and NP completeness problems.
13) Discuss NP hard and NP complete problems.
14) Discuss in detail the different classes in NP hard and NP complete.