UNIT- IV

INTRODUCTION TO BACKTRACKING & BRANCH AND BOUND

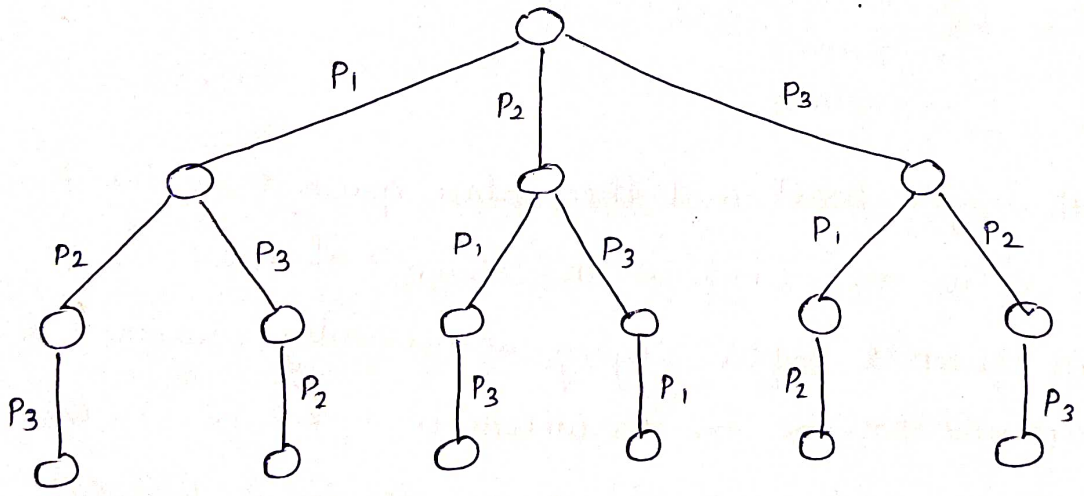
Backtracking:

* It is one of the problem solving approach to design algorithms to solve problems.

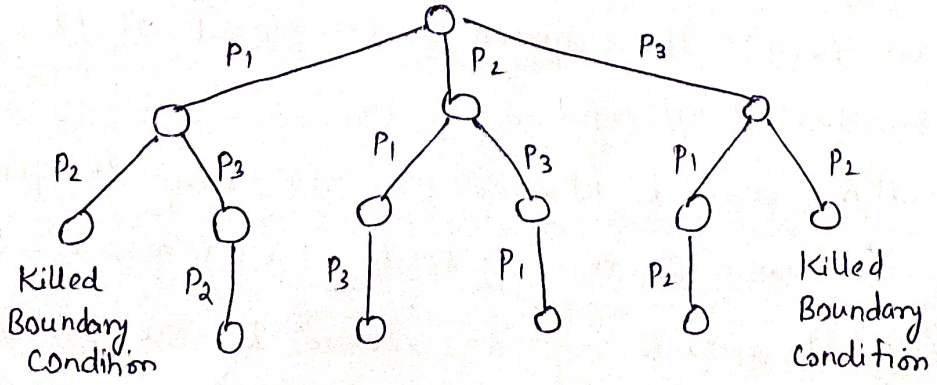
* It follows brute-force approach. It tries out all possible solutions for a problem and selects the desired solution The desired solution will be chosen based on the constraint given called as bounding function.

* Assume 3 persons P_1, P_2 and P_3 to be arranged in 3 chairs. Possible ways to arrange is $3! = 6$ ways.

* Solution is represented as a tree called as State space tree. It constructs state space tree following Depth First Search (DFS).



* Assume a constraint - P_2 should not be placed in the middle. This acts as a bounding function.



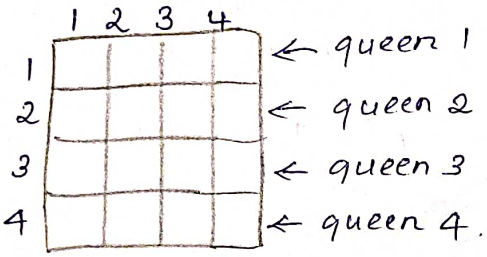
Branch and Bound:

* It is similar to backtracking. Only difference is that it constructs the state space tree using Breadth First Search (BFS).

N-Queens Problem:

* The problem is to place n queens on an n×n chessboard so that no two queens attack each other by being in the same row or in the same column or on the same diagonal.

* Since each of the four queens has to be placed in its own row, all we need to do is to assign a column for each queen.

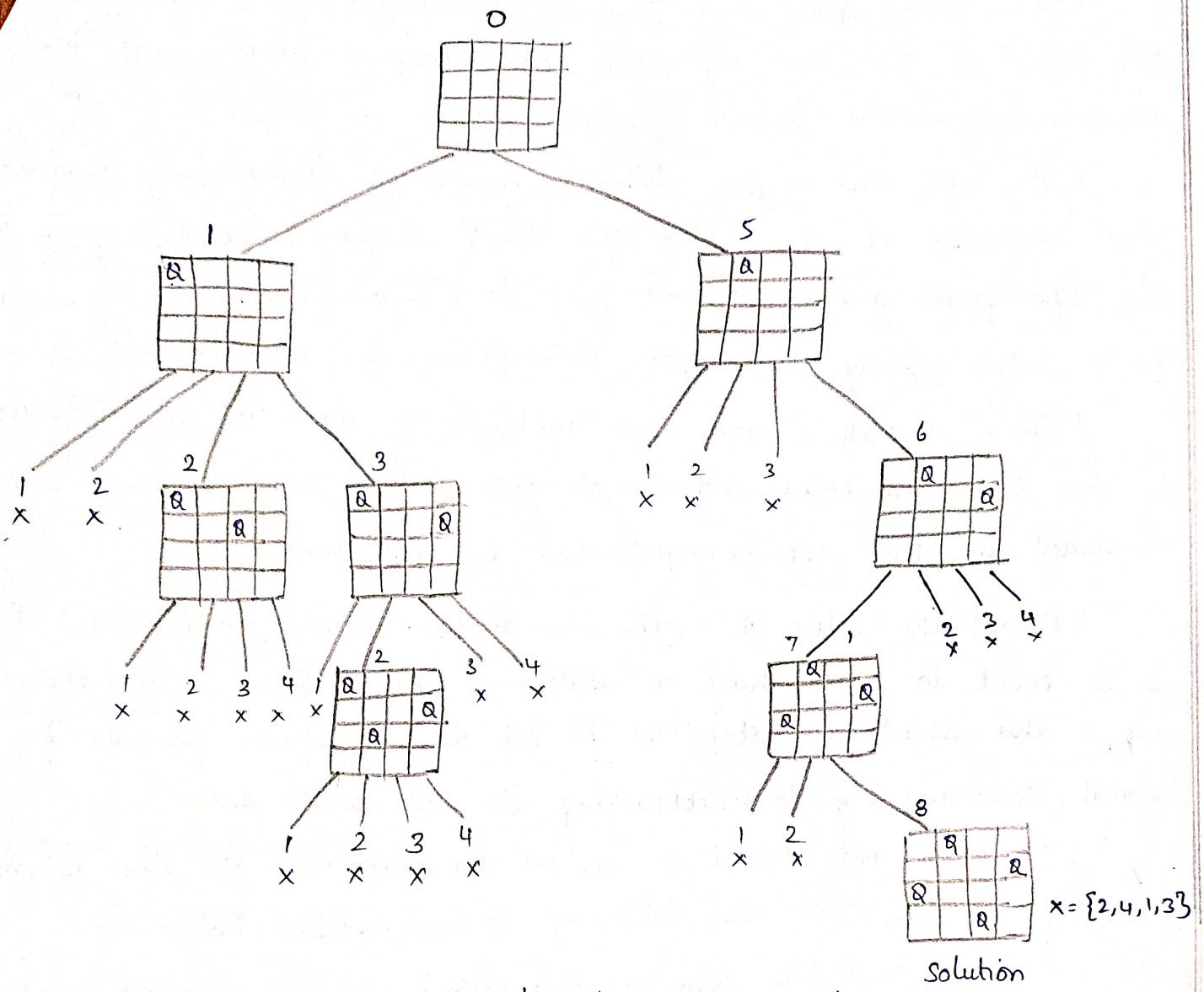


* Start with empty board and then place queen 1 in the first possible position of its row, which is in column 1 of row 1.

* Then place queen 2, after trying unsuccessfully columns 1 & 2, in the first accessible position for it, which is square (2,3), the square in row 2 and column 3. This proves to be a dead end because there is no acceptable position for queen 3.

* Hence, the algorithm backtracks and puts queen 2 in the next possible position at (2,4). Then queen 3 is placed at (3,2), which proves to be another dead end.

* The algorithm then backtracks all the way to queen 1 and moves it to (1,2). Queen 2 then goes to (2,4), queen 3 to (3,1) and queen 4 to (4,3), which is a solution to the problem.



* The diagram represents the state-space tree of solving the 4-queens problem by backtracking. x denotes an unsuccessful attempt to place a queen in the indicated column. The numbers above the nodes indicate the order in which the nodes are generated.

SUM OF SUBSET PROBLEM:

* The Sum of subset problem is to find a subset of a given set $A = \{a_1, \ldots a_n\}$ of n positive integers $a_1 < a_2 < \ldots < a_n$ whose sum is equal to a given positive integer d.

* Example - for $A = \{1, 2, 5, 6, 8\}$ and $d = 9$, there are 2 solutions : $\{1, 2, 6\}$ and $\{1, 8\}$.

* The state space tree can be constructed as a binary tree. The root of the tree represents the starting point, with no decisions about the given elements made as yet.

* Its left and right children represent, respectively, inclusion and exclusion of a_1 in a set being sought. Similarly, going to the left from a node of the first level corresponds to inclusion of a_2 while going to right corresponds to its exclusion and so on.

* Thus, a path from the root to a node on the i^{th} level of the tree indicates which of the first i numbers have been included in the subsets represented by that node.

* Record the value of s, the sum of the numbers, in the node. If s is equal to d, we have a solution to the problem. We can either report this result and stop (or) if all the solutions need to be found, continue by backtracking to the node's parent.

* If s is not equal to d, we can terminate the node as non promising if either of the following 2 inequalities hold:

$s + a_{i+1} > d$ (the sum s is too large).

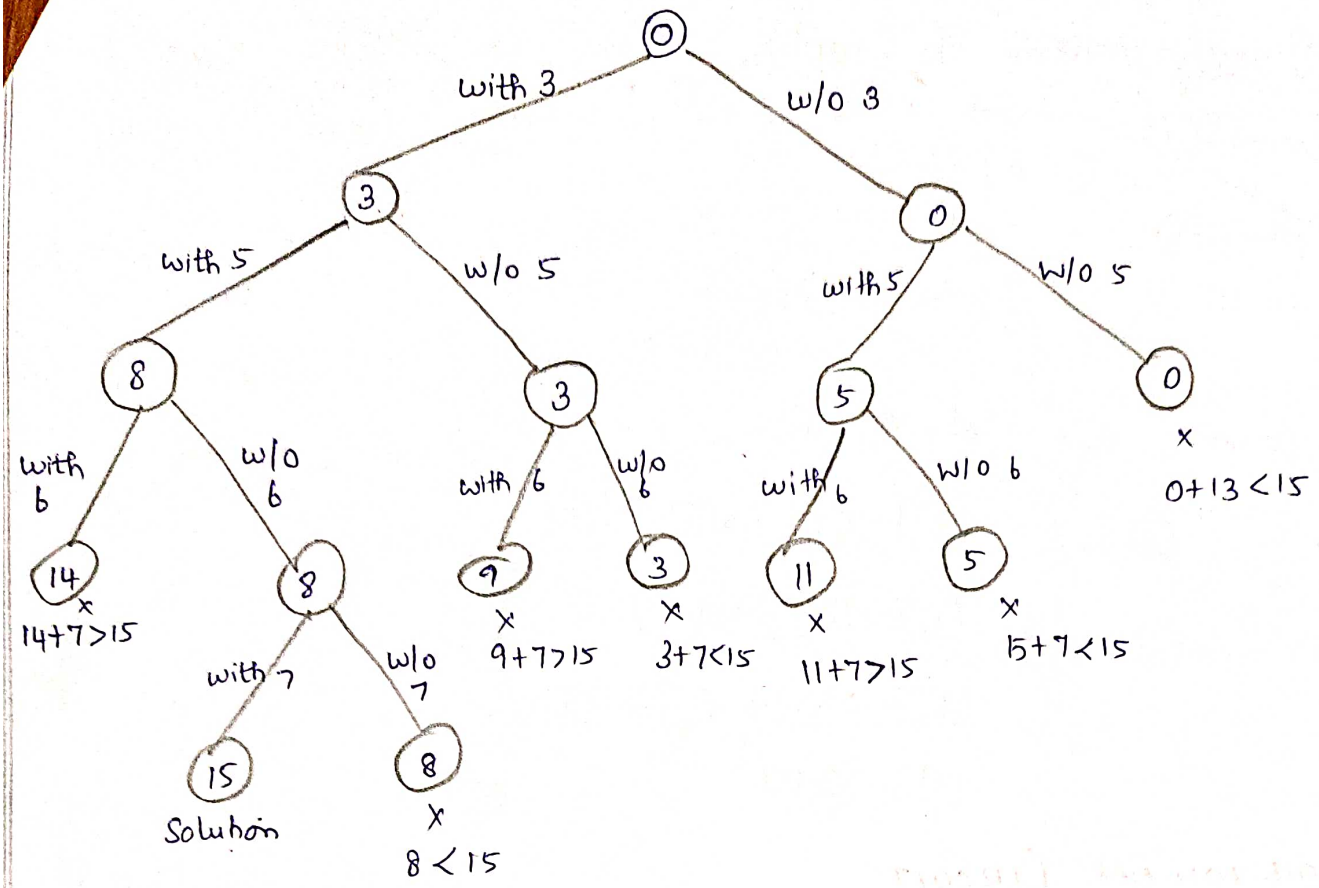
$s + \sum_{j=i+1}^{n} a_j < d$ (the sum s is too small)

* Time Complexity of Sum of subset is $O(2^N)$.

Algorithm:

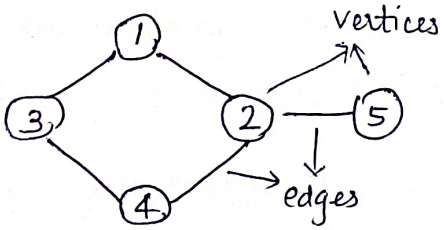
```
SumofSubsets (s, k, r)
{
  X[k] = 1;
  if (s+W[k] == m) then
     write (x[1:k]);
  else if (s + W[k] + W[k+1] <= m) then
     Sumofsubsets (s + W[k], k+1, r - W[k]);
  if ((s+r-W[k] >= m) and (s + W[k+1] <= m)) then
  {  X[k] = 0;
     Sumofsubsets (s, k+1, r - W[k]);
  }
}
```

Example A = {3,5,6,7} and d = 15

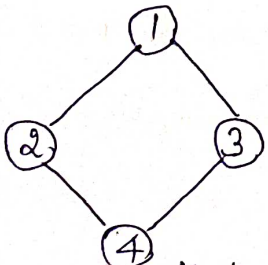


GRAPH - INTRODUCTION:

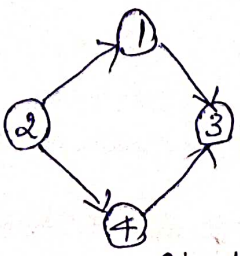
* A graph is composed of set of vertices and set of edges denoted as G = (V, E).



* Graph whose edges are not directed are called undirected graph. Graph whose edges are directed are called directed graph.



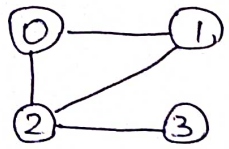
Undirected graph



Directed graph

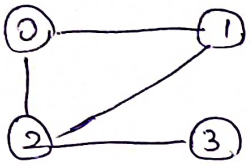
* Graph Representation.

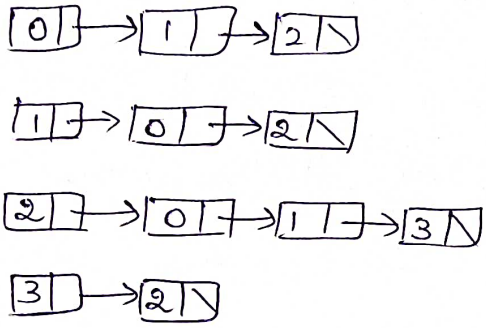
→ Adjacency matrix of Graph.



	0	1	2	3
0	0	1	1	0
1	1	0	1	0
2	1	1	0	1
3	0	0	1	0

→ Adjacency list of Graph



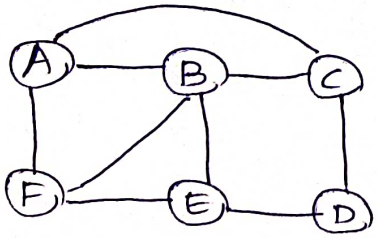


HAMILTONIAN CIRCUIT:

* Hamiltonian circuit (or) Hamiltonian cycle - Given a graph, start a vertex, visit all other vertices in the graph exactly once and return back to the start vertex.

* Given graph, can be either directed or undirected but it must be connected. This is NP hard problem. It takes exponential time to find all hamiltonian cycle in graph.

Example:



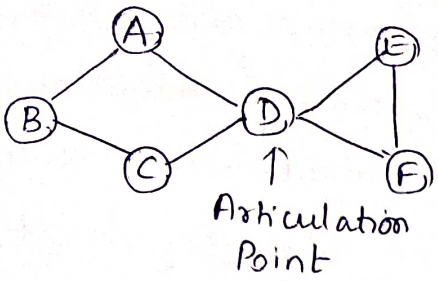
Cycle 1 – A, B, C, D, E, F, A

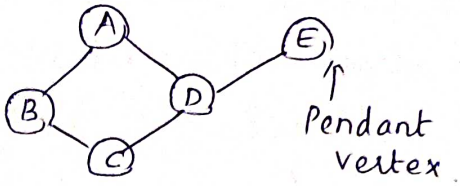
Cycle 2 - A, B, F, E, D, C, A

Cycle 3 - A, F, B, E, D, C, A

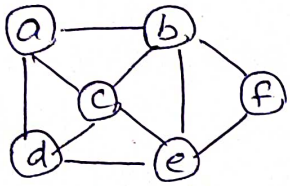
B, C, D, E, F, A, B - Similar to cycle 1

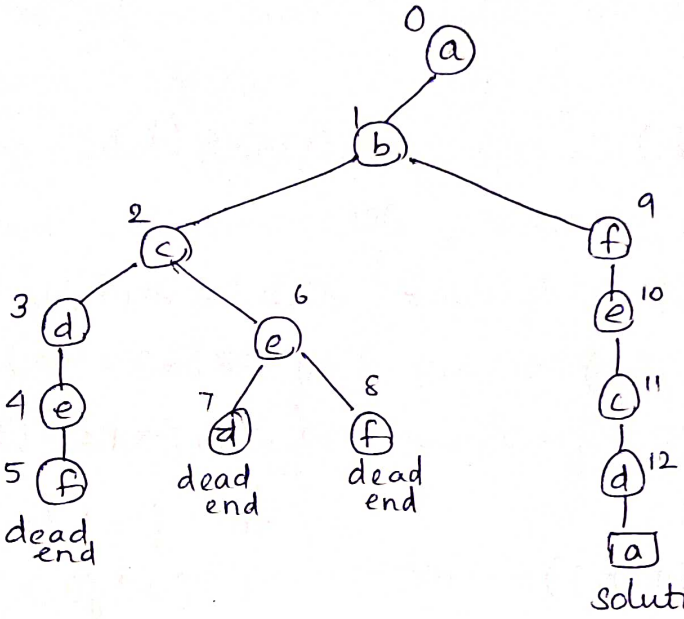
* Graphs with articulation points and pendant vertices will not contain Hamiltonian cycle.





Example:





* Consider the problem of finding a Hamiltonian circuit in the given graph. Without loss of generality, we can assume that if a Hamiltonian circuit exists, it starts at vertex a. Accordingly, vertex a is made as the root of the State-Space tree.

* Using the alphabet order to break the three-way tie among the vertices adjacent to a, we select vertex b. From b, the algorithm proceeds to c, then to d, then to e and finally to f, which proves to be a dead end.

* So, the algorithm backtracks from f to e, then to d, and then to c, which provides the first alternative for the algorithm to pursue.

* Going from c to e eventually proves useless, and the algorithm has to backtrack from e to c and then to b.

* From there, it goes to the vertices f, e, c and d, from which it can legitimately return to a, yielding Hamiltonian circuit a, b, f, e, c, d, a. To find other Hamiltonian circuit, continue this process of by backtracking from the leaf to the solution found.

Algorithm:

```
Hamiltonian (k)
do
 {
  Nextvertex (k);
  if (x[k] == 0)
    return;
  if (k == n)
    print (x[1..n]);
  else
    Hamiltonian (k+1);
 } while (true);
```

```
Nextvertex (k)
do
 {
  x[k] = x[k+1] mod (n+1);
  if (x[k] == 0) return;
  if (G[x[k-1], x[k]] != 0)
  {
   for j = 1 to k-1
   if (x[j] == x[k]) then break;
   if (j == k)
   if (k < n) or (k == n) and
                G[x[n], x[1]] != 0)
    return;
  }
```

BRANCH AND BOUND:

* Branch and bound is a systematic method for solving optimization problems. Branch and bound is a rather general optimization technique that applies where the greedy method and dynamic programming fails.

* However, it is much slower. Indeed, it often leads to exponential time complexities in the worst case. On the other hand, if applied carefully, it can lead to algorithms that run reasonably fast on average.

* The general idea of branch and bound is a BFS-like search for the optimal solution, but not all nodes get expanded.

* Rather, a carefully selected criterion determines which node to expand and when, and another criterion tells the algorithm when an optimal solution has been found.

KNAPSACK PROBLEM:

* Given n items of known weights w_i and values v_i, $i = 1, 2, \ldots, n$ and a knapsack of capacity W, find the most valuable subset of the items that fit in the knapsack.

* It is convenient to order the items of a given instance in descending order by their value-to-weight ratios. Then the first item gives the best payoff per weight unit and the last one gives the worst payoff per weight unit, with ties resolved arbitrarily:

$$v_1/w_1 \geq v_2/w_2 \geq \ldots \geq v_n/w_n$$

* Each node on the ith level of this tree, $0 \leq i \leq n$, represents all the subsets of n items that include a particular selection made from the first i ordered items.

* This particular selection is uniquely determined by the path from the root to the node: a branch going to the left indicates the inclusion of the next item, and a branch going to the right indicates its exclusion.

* We record the total weight w and the total value v of this selection in the node, along with some upper bound ub on the value of any subset that can be obtained by adding zero or more items to this selection.

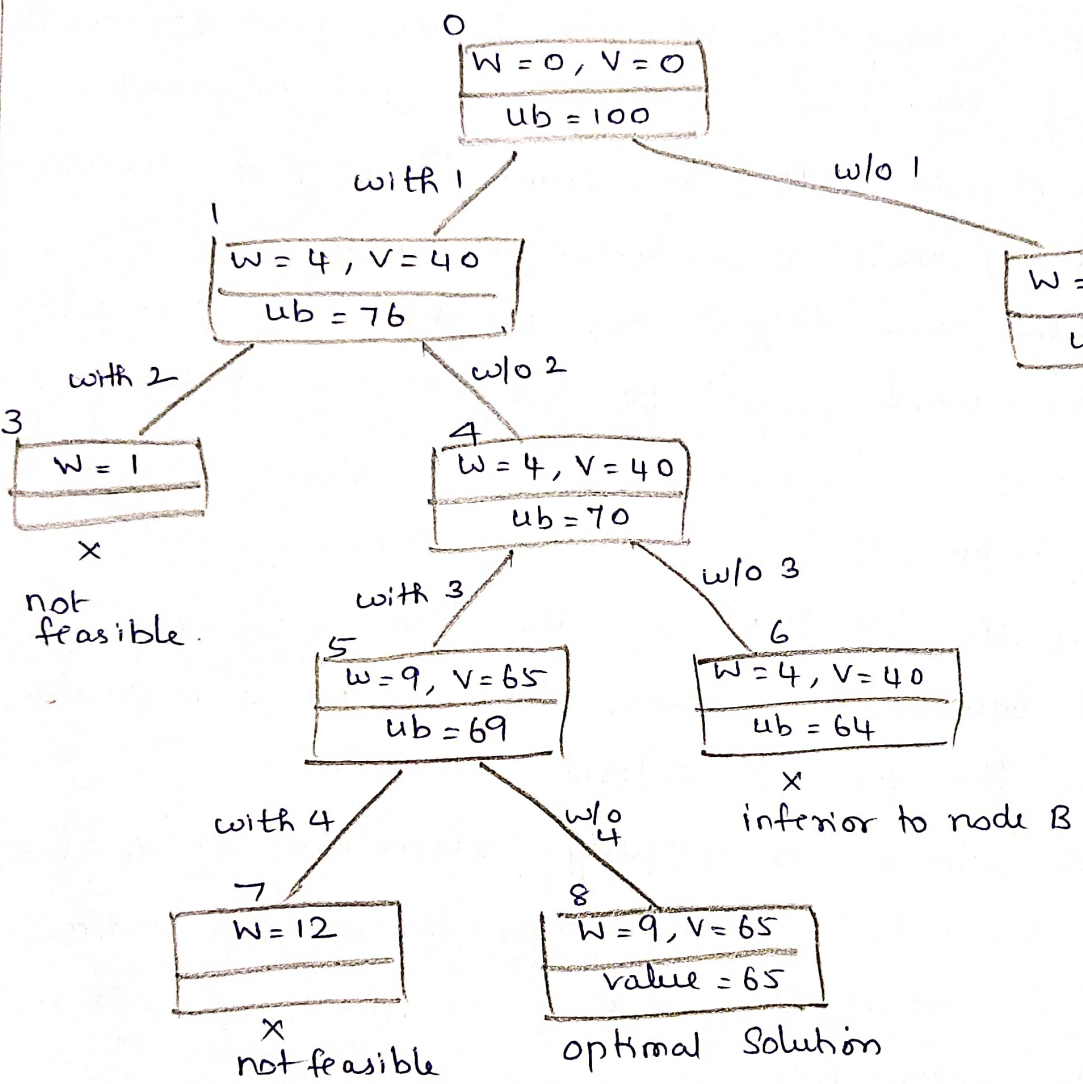
* To compute the upper bound ub, add to v, the total value of the items already selected, the product of the remaining capacity of knapsack $W - w$ and the best per unit payoff among the remaining items, v_{i+1}/w_{i+1}:

$$ub = v + (W - w)(v_{i+1}/w_{i+1}).$$

Example:

Item	Weight	value	value/weight
1	4	$40	10
2	7	$42	6
3	5	$25	5
4	3	$12	4

The knapsack capacity W is 10.



* At the root of the state-space tree, no items have been selected as yet. Hence, both the total weight of the items already selected w and their total value v are equal to 0. The value of the upper bound computed by formula is $100.

* Node 1, the left child of the root, represents the subsets that include item 1. The total weight and value of the items already included are 4 and $40, respectively; the value of the upper bound is $40 + (10 - 4) * 6 = \$76$.

* Node 2 represents the subsets that do not include item 1. Accordingly, $w = 0$, $v = \$0$, and $ub = 0 + (10-0) * 6 = \$60$.

* Since node 1 has a larger upper bound than the upper bound of node 2, it is more promising for this maximization problem, and we branch from node 1 first. Its children - nodes 3 and 4 represent subsets with item 1 and without item 2, respectively.

* Since the total weight w of every subset represented by node 3 exceeds the knapsack's capacity, node 3 can be terminated immediately. Node 4 has the same values of w and v as its parent; the upper bound ub is equal to $40 + (10-4) * 5 = \$70$

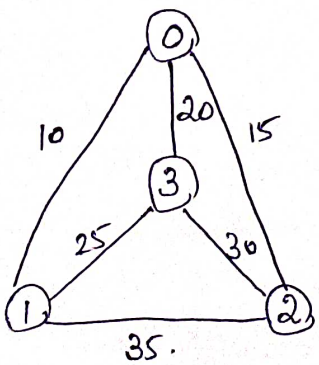
Difference between Greedy and Dynamic Programming

* A greedy approach is to pick the items in decreasing order of value per unit weight. The greedy approach works only for fractional knapsack problem and may not produce correct result for 0/1 knapsack.

* Dynamic Programming is used to solve 0/1 knapsack. In Dynamic Programming, 2 dimensional table of size $n \times w$ is used. This solution doesn't work if item weights are not integers.

TRAVELLING SALESMAN PROBLEM (TSP)

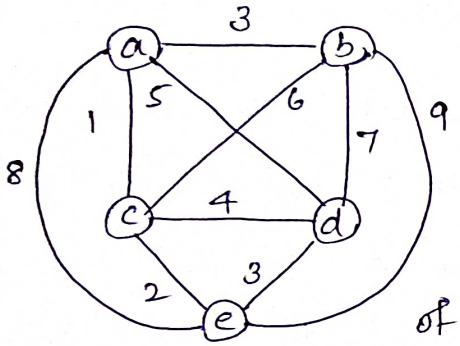
* Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible tour that visits every city exactly once and returns to the starting point.



* For example, consider the graph shown in figure. A TSP tour in the graph is 0-1-3-2-0. The cost of the tour is 10 + 25 + 30 + 15 which is 80.

* Problem statement - For each city i, $1 \le i \le n$, find the su
s_i of the distances from city i to the two nearest cities; compute the sum s of these n numbers, divide the result by 2, and, if all the distances are integers, round up the result to the nearest integer: $lb = s/2$.

Example:



* For eg, for the instance in figure, formula yields

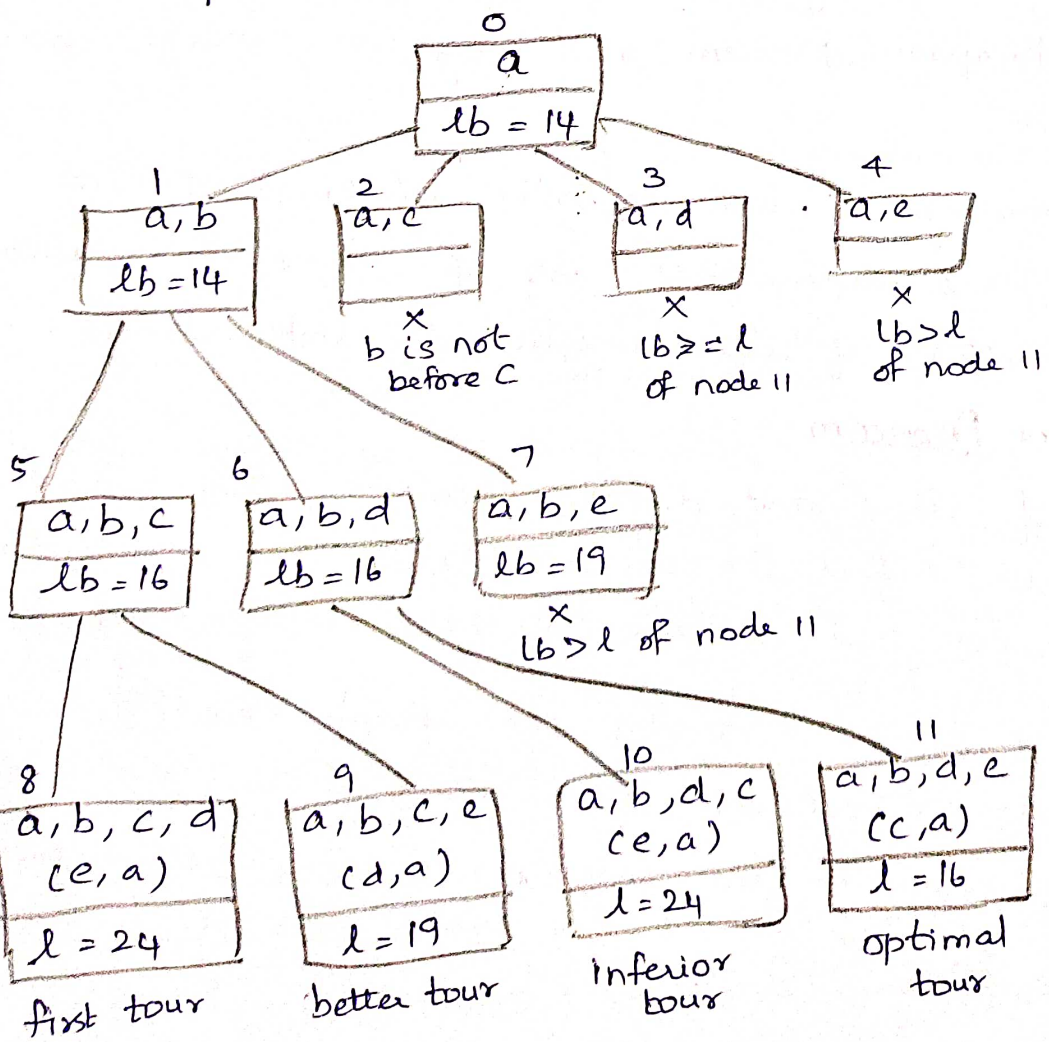
$$lb = [(1+3)+(3+6)+(1+2)+(3+4)+(2+3)]/2$$
$$= 14.$$

* To include particular edge on the subset of tour, the lower bound can be modified.

* eg, on including edge (a, d) the lower bound is

$$[(1+5)+(3+6)+(1+2)+(3+5)+(2+3)]/2 = 16.$$

State Space Tree.



FLOYD-WARSHALL'S ALGORITHM:

* It is used in All-pairs shortest path. Given a weighted connected graph (undirected or directed), the all-pairs shortest paths problem asks to find the distances - ie, the lengths of the shortest paths - from each vertex to all other vertices.

* It is convenient to record the lengths of shortest paths in an nxn matrix D called the distance matrix. The element dij in the ith row and the jth column of this matrix indicates the length of the shortest path from the ith vertex to j^{th} vertex.

* The floyd-warshall's algorithm computes the distance matrix of a weighted graph with n vertices through a series of nxn matrices: D(0), ... D(k-1), D(k), ..., D(n).

* In particular, the series starts with D(0), which does not allow any intermediate vertices in its paths; hence, D(0) is simply the weight matrix of the graph.

* The last matrix in the series, D(n), contains the lengths of the shortest paths among all paths that can use all n vertices as intermediate and hence is nothing other than the distance matrix being sought.

* The length of the shortest path is

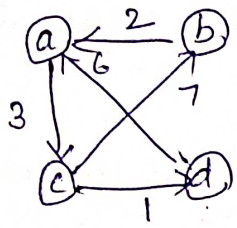
$$d_{ij}^{(k)} = \min \{ d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \} \quad \text{for } k \geq 1, \; d_{ij}^{(0)} = w_{ij}$$

ALGORITHM:

```
Floyd ( W[1...n, 1...n])
// Implements floyds algorithm for the all-pairs shortest-paths problem
// Input : The weight matrix W of a graph with no negative-length
                                                        cycle
// Output : The distance matrix of the shortest paths lengths
D ← W // is not necessary if W can be overwritten.
```

```
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.
```

Example:



$D^{(0)}$ =

	a	b	c	d
a	0	∞	3	∞
b	2	0	∞	∞
c	∞	7	0	1
d	6	∞	∞	0

Lengths of the shortest paths with no intermediate vertices

$D^{(1)}$ =

	a	b	c	d
a	0	∞	3	∞
b	2	0	5	∞
c	∞	7	0	1
d	6	∞	9	0

Lengths of the shortest paths with intermediate vertices numbered not higher than 1. ie, just a

$D^{(2)}$ =

	a	b	c	d
a	0	∞	3	∞
b	2	0	5	∞
c	9	7	0	1
d	6	∞	9	0

Lengths of the shortest paths with intermediate vertices numbered not higher than 2. ie. just a and b.

$D^{(3)}$ =

	a	b	c	d
a	0	10	3	4
b	2	0	5	6
c	9	7	0	1
d	6	16	9	0

Lengths of the shortest paths with intermediate vertices numbered not higher than 3. (ie) a, b and c

$D^{(4)}$ =

	a	b	c	d
a	0	10	3	4
b	2	0	5	6
c	7	7	0	1
d	6	16	9	0

Lengths of the shortest paths with intermediate vertices numbered not higher than 4, (ie) a, b, c and d.