0/1 Knapsack Problem Using FIFO Branch & Bound

Definition:.

Branch & Bound discovers branches within the complete search space by using estimated bounds to limit the number of possible solutions. The different types

1. FIFO
2. LIFO
3. LC

define different strategies to explore the search space and generate branches.

FIFO:-

FIFO (first in, first out) always the oldest node in the queue is used to extend the branch. This leads to a breadth-first search, where all nodes at depth d are visited first, before any nodes at depth d+1 are visited.

In FIFO branch and bound, as is visible by the name, the child nodes are explored in first in first out manner. We start exploring from starting child node.

Procedure:-

1. Draw a state space tree and set upper bound = ∞
2. compute $c^{\wedge}(x)$, $u(x)$ for each node.
3. $u(x) = -\sum p_i$

$$c^{\wedge}(x) = u(x) - [m - \text{current total weight}] * \frac{[\text{actual profit of remaining object}]}{[\text{actual weight of remaining object}]}$$

4. If $u(x)$ is minimum than upper will set to $u(x)$
5. If $c^{\wedge}(x) >$ upper, kill node x
6. Next live node becomes E-node and Generate children for E-node.
7. Repeat 2 to 6 until all nodes get covered
8. The minimum cost $c^{\wedge}(x)$ becomes the answer node. Trace the path in backward direction from x to root for solution subset.

Steps:

for node - 1:

We take the table

O_i	P_i	w_i
1	10	2
2	10	4
3	12	6
4	18	9

the bag capacity is $m = 15$.

i) node - 1 :

Let us consider $m = 15$ and upper bound $= \infty$

for node 1 $\hat{c}(1) = -38$

$u(1) = -32$

6	
~~18~~	$3/9 \times 18^2$
12	6
10	4
10	2

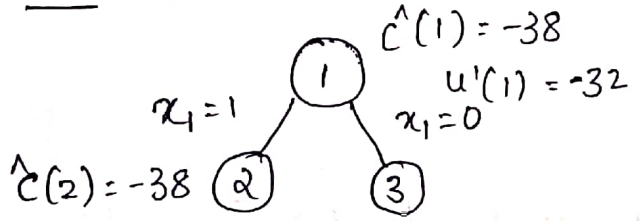
$\}u(1)\}\hat{c}(1)$

So now upper bound $= -32$

(1) $\hat{c}(1) = -38$

$u(1) = -32$

2) node - 2 :



The $x_1 = 1$ the node 1 is included so.

6	$3/9 \times 18$
12	6
10	4
10	2

$\}u(2)\}\hat{c}(2)$

3) node - 3 :

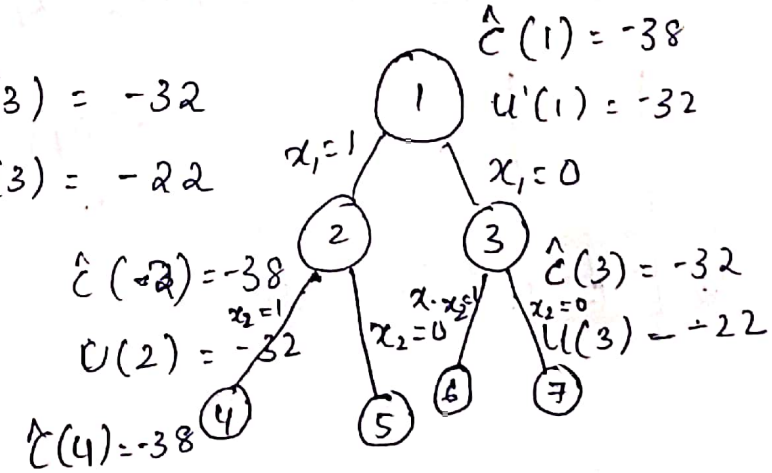
here for node - 3 $x_1 = 0$ x_1 is not included so

10	$5/9 \times 18$
12	6
10	4

$\}-22\}-32$

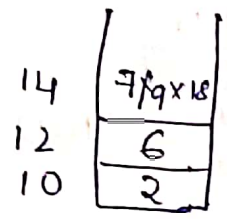
$\hat{c}(3) = -32$

$u(3) = -22$



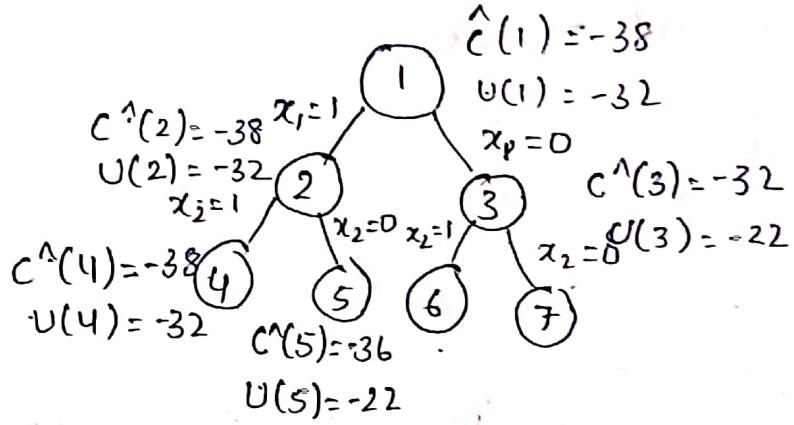
node - 5 :-

Here x_2 is not included



$\hat{c}(5) = -36$

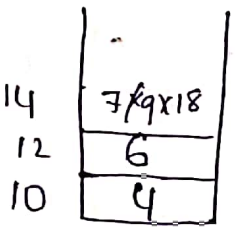
$u(5) = -22$



node - 6 :-

we have include x_2 and doesn't include x_1

so

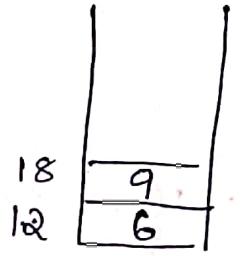


$\hat{c}(6) = -32$

$u(6) = -22$

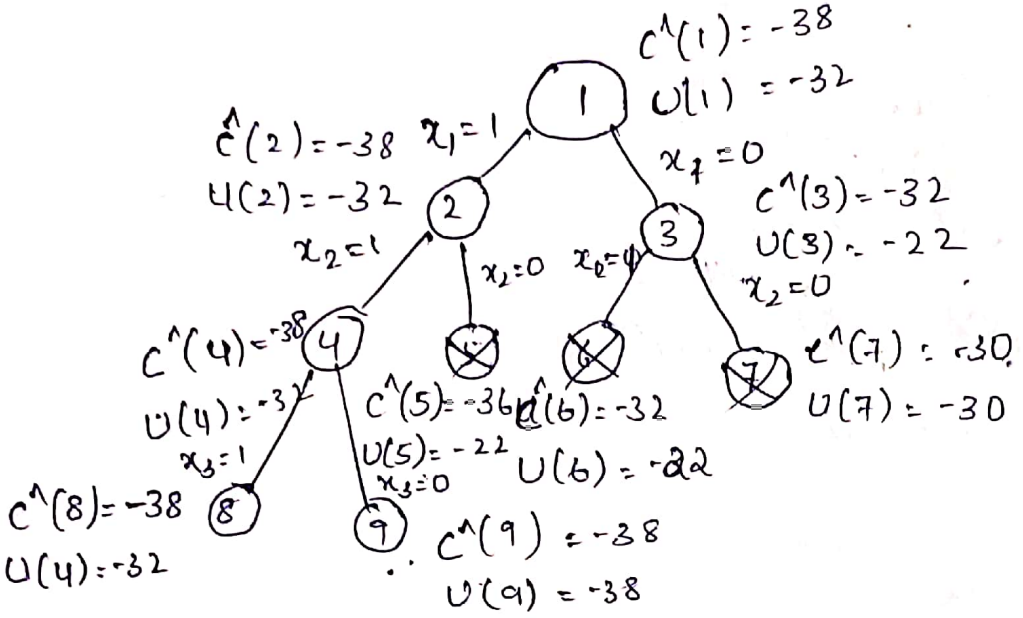
node - 7 :-

we have to exclude $x_1 = 0$ and $x_2 = 0$



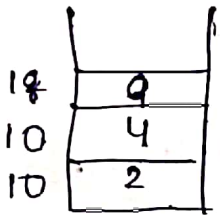
$\hat{c}(7) = -30$

$u(7) = -30$



Node - 9 :

x_3 must not be included



$c^(9) = -38$

$u^(9) = -38$

The upper bound node is replaced because -32 is larger than -38

upper bound = -38

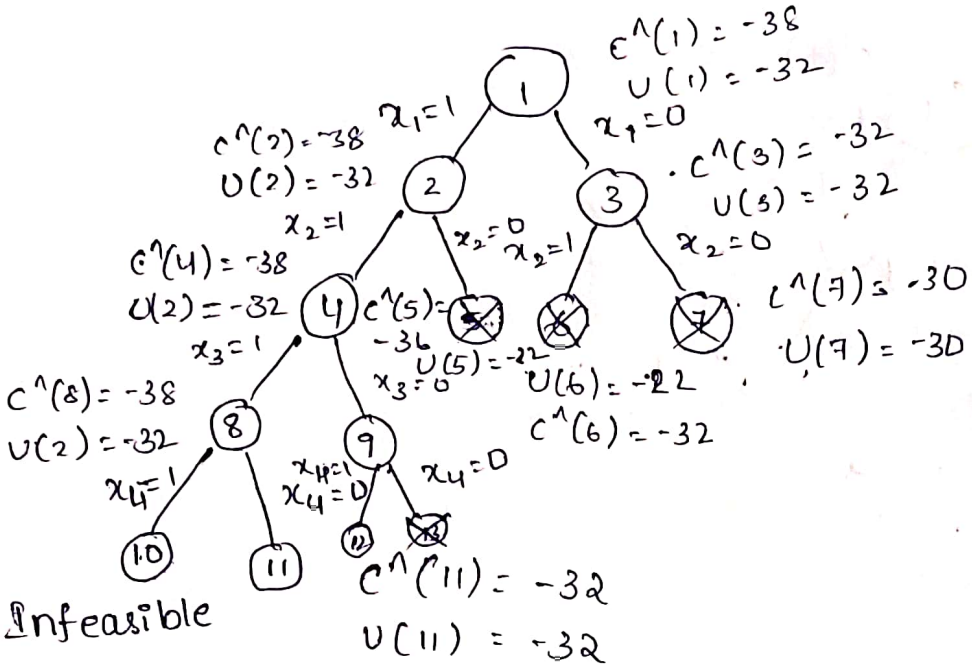
- $c^(x) >$ upper, Kill node x

5 - -36 > -38 so, Kill the node 5.

6 - -32 > -38, so Kill the node 6.

7 - -30 > -38, so Kill the node 7

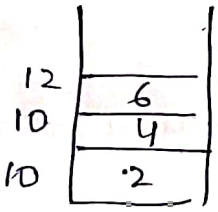
So the tree will be



For node-10 the bag exceeds the capacity so it is infeasible.

node-11 :-

we have to exceed x_4.



then

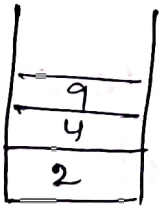
$c^(11) = -32$

$U(11) = -32$

Here $-32 > -38$; kill the node-11

node-12 :-

$x_3 = 0$ is excluded

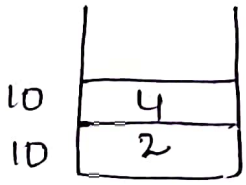


$c^(12) = -38$

$U(12) = -38$

node-13 :-

x_3, x_4 is excluded



$c^(13) = -20$

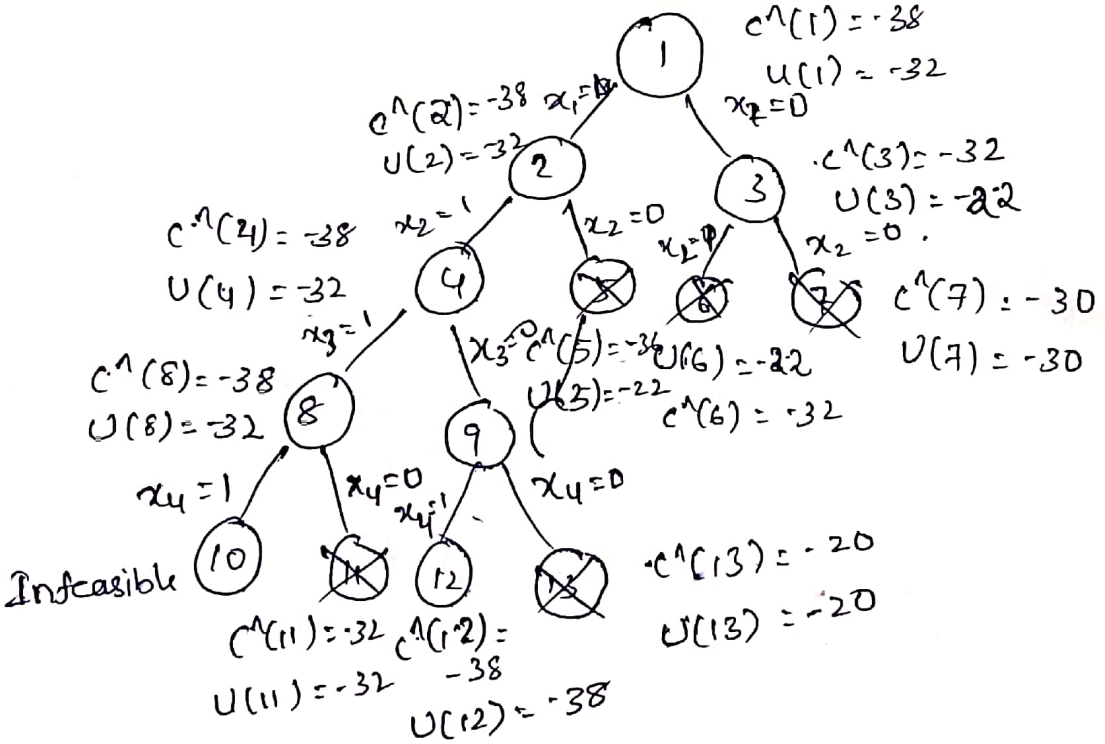
$U(13) = -20$

$-20 > -38$, node 13 is killed

Node - 12 is the answer

	P_i	w_i
$x_1 = 1$	10	2
$x_2 = 1$	10	4
$x_3 = 0$	-	
$x_4 = 1$	18	9
	+38	15

The profit is 38 and weight 15.



SUM OF SUBSETS

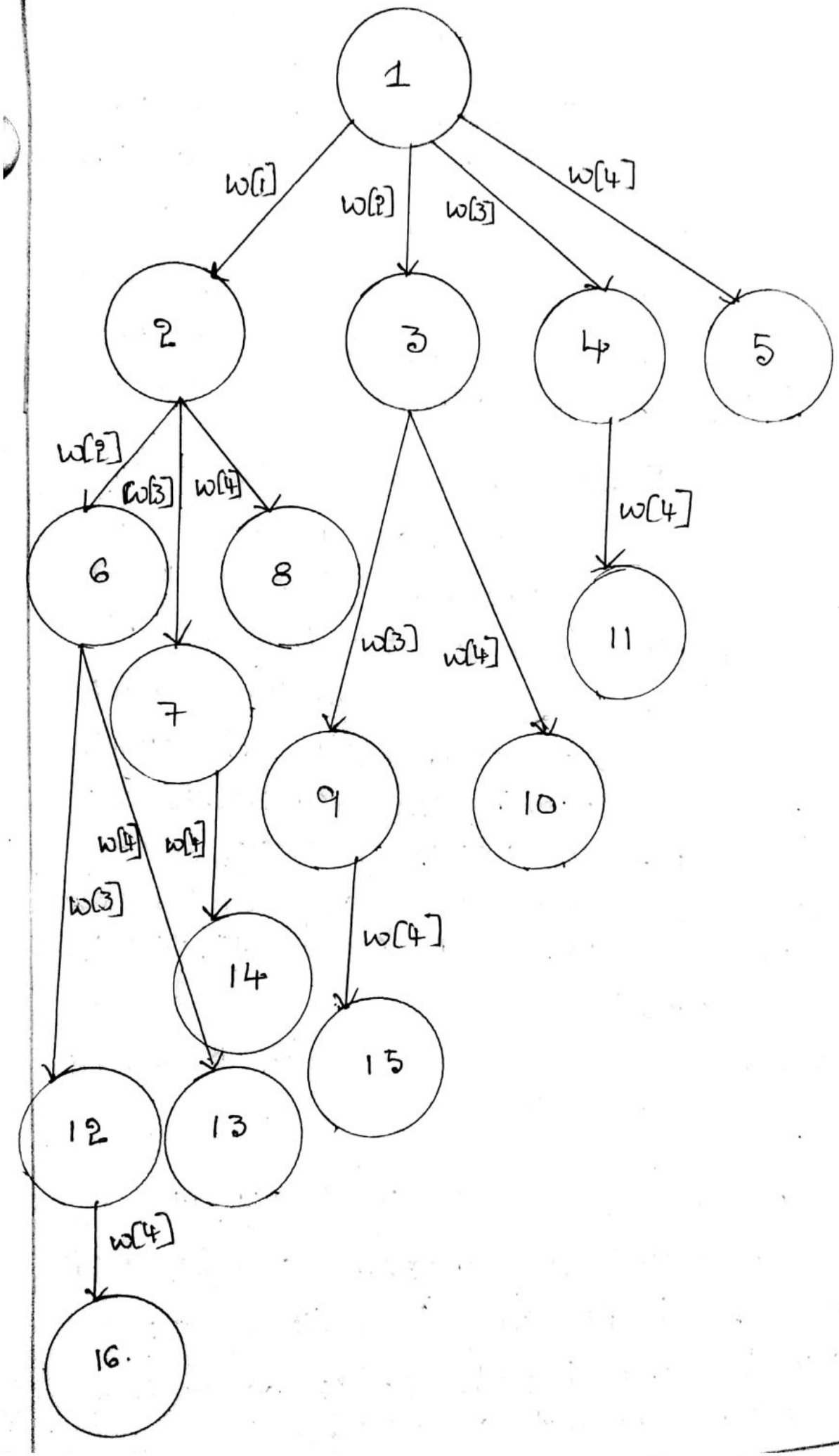
Subset sum problem is to find subset of elements that are selected from a given set whose sum adds up to a given number k. we are considering the set contains non-negative values. It is assumed that the input set is unique.

In a state space tree the rootnode represents a function called and the branch represents the candidate element as we go down the depth of the tree we add elements until we satisfy the explict constraints. we continue to add child's nodes for wherever the constraints are not satisfy the further generations of subtree that node are stop and backtra-ck to the previous node to explore the other nodes.

General state space tree with 4 elements [w[1] - - - w[4]).

Assume given set of 4 elements, say w[1] - - - w[4]. tree diagrams can be used to design backtracking algorithms. the following tree diagram depicts approach of generating variable sized tuple.

Scanned with CamScanner



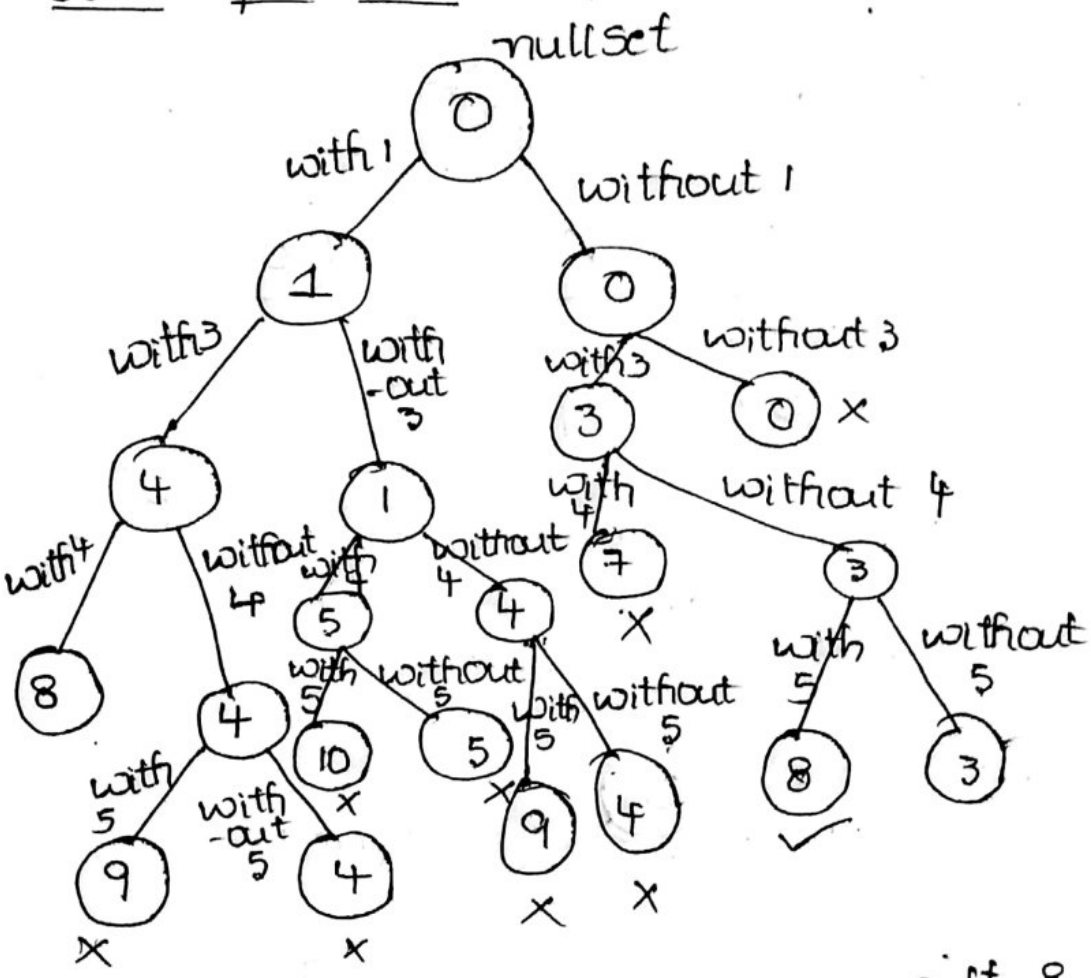
Scanned with CamScanner

Example-1:

consider Given set of elements $\{1,3,4,5\}$ where $K = 8$.

Sol:- $\{\{1,3,4\}\ \{3,5\}\}$

State space tree



∴ The subsets whose sum result 8 are $\{1,3,4\}$, $\{3,5\}$.

Here move from top to end until reach our K value.

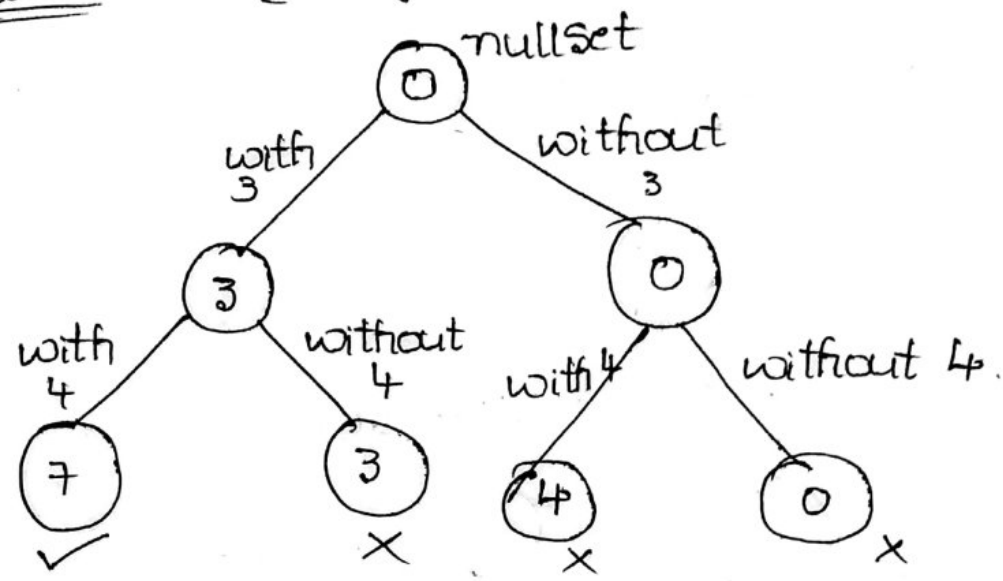
we are unable to get the solution then kill the tree.

Scanned with CamScanner

Example-2:

Consider the subset {1,2,3,4} and K=7.

Solution: {3,4}.



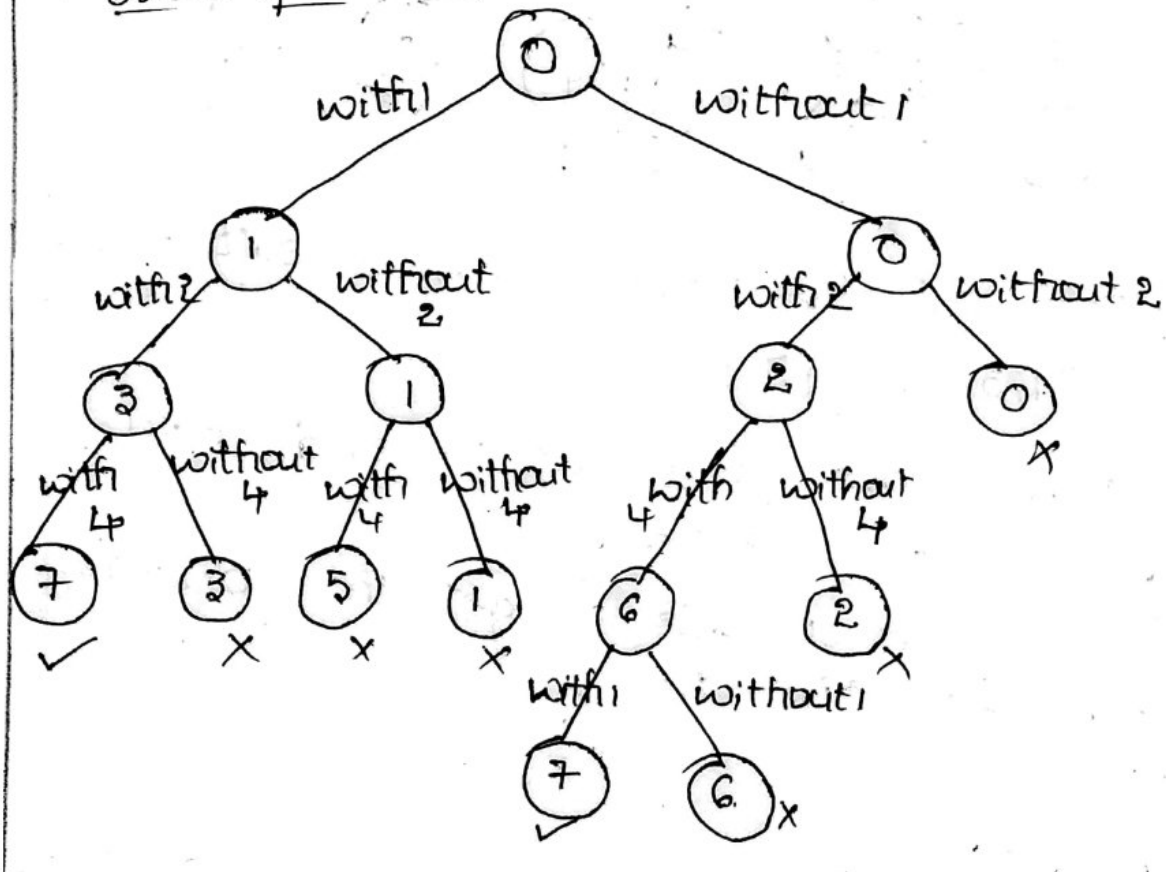
State space tree.

Example-3:

Consider the subset {1,2,4,6} K=7.

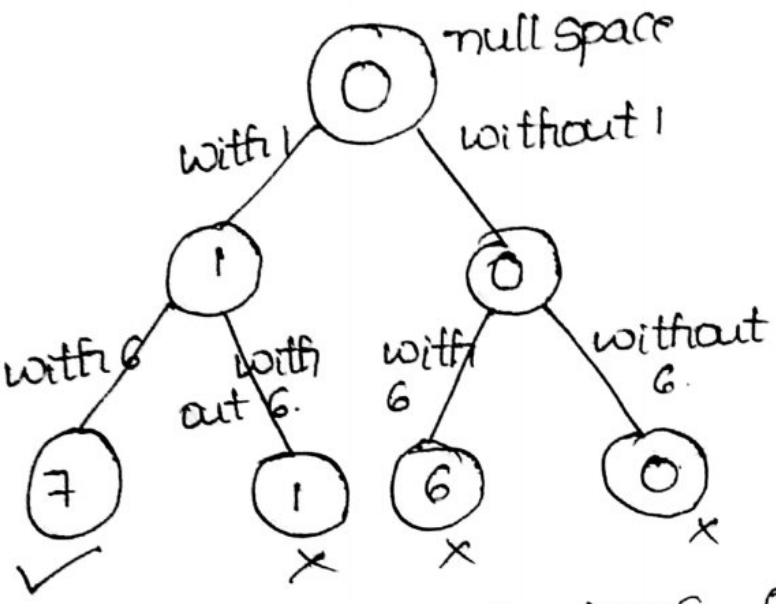
Ans: {1,2,4}, {1,6}.

State space tree



Scanned with CamScanner

State space tree.



Advantages and Applications of subset-sum problem.

* It is specially constructed problem which is known to be easy.
* There are security problems other than public-key codes for which subset-problems are useful.
* It is used in computer passwords.
* It is used in message verification.

Scanned with CamScanner

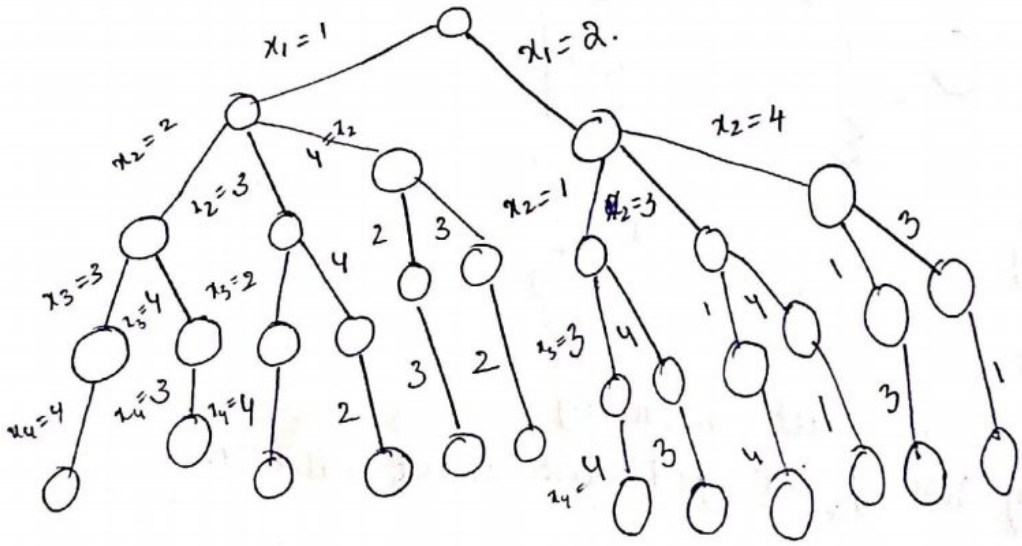
N-Queens Problem

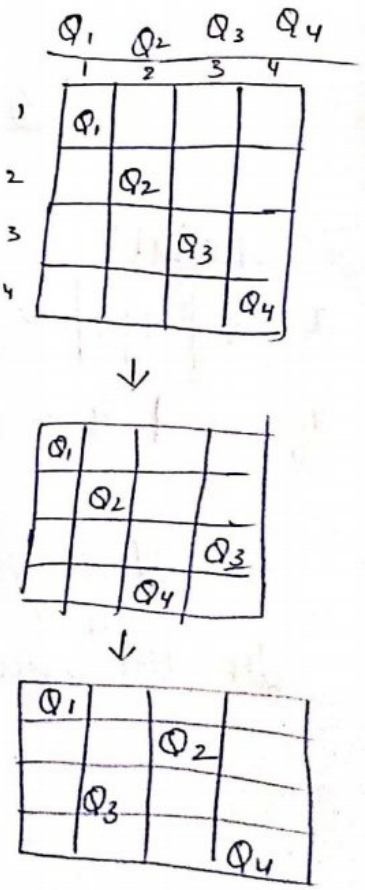
→ There will be a chess board given with n×n cells. n Queens will be given. In a chess board a Queen can move horizontally, vertically and diagonally. In this problem also we have to ~~keep~~ place N Queens such that no Queen attacks another.

→ for this problem, we may have many solutions and we need all the solutions which satisfies the condition. So, for these type of getting all solutions backtracking is used. (for getting optimal solution we have to choose dynamic Programming).

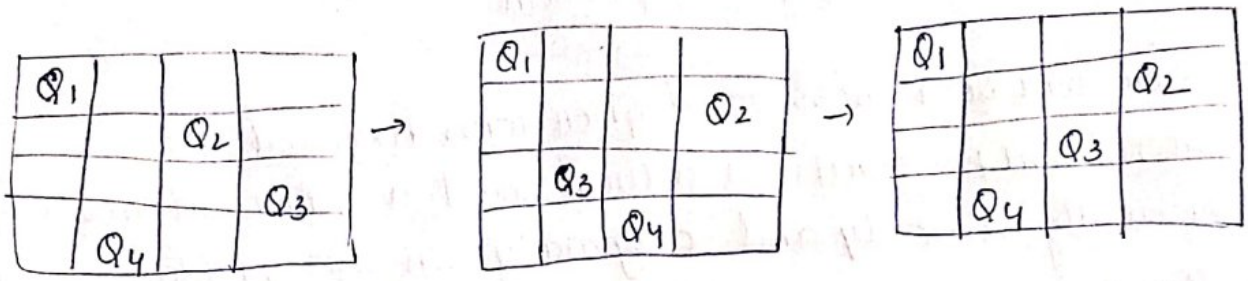
→ for example consider n=4 and for reducing the problem i^{th} Queen is placed in the i^{th} row where i ranges from 1 to 4. So we have to choose its appropriate column.

State space tree

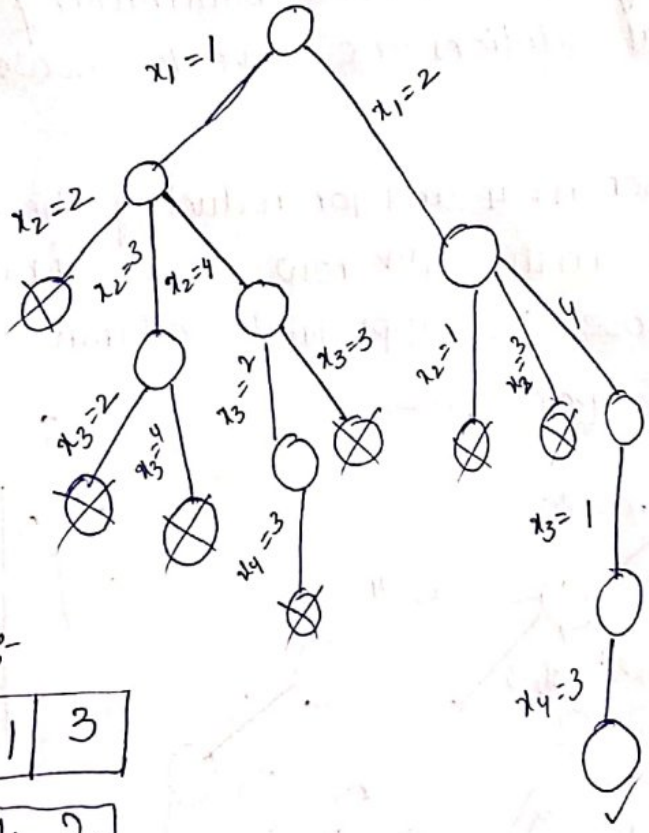




Scanned with CamScanner



Similarly we can draw for x_1 placed at 3rd and 4th columns. Now by using Bounding function we get the solutions for N-Queens problem.



Solution :-

x	2	4	1	3
x	3	1	4	2

Now by using, back tracking we get the above solution

In the state space tree there are

Scanned with CamScanner

$$1 + 4 + 4\times3 + 4\times3\times2 + 4\times3\times2\times1$$

$$= 1 + \sum_{i=0}^{3}\left[\prod_{j=0}^{i}(4-j)\right] \quad \text{for 4 Queens}$$

$$= 1 + \sum_{i=0}^{N-1}\left[\prod_{j=0}^{i}(N-j)\right] \quad \text{for N Queens the}$$

total nodes which are possible to place the Queens without checking all the conditions (diagonal check is not done) in state space tree are

$$1 + \sum_{i=0}^{N-1}\left[\prod_{j=0}^{i}(N-j)\right]$$

Scanned with CamScanner

Introduction to Branch-and-bound technique:

Branch-and-bound is a state-space search method that can be visualized as an improved form of backtracking. In the branch-and-bound technique, a set of feasible solutions are partitioned, and the subsets that do not have optimal solutions are deleted. Branch and bound is an algorithm design paradigm which is generally used for solving the optimization problems. Branch-and-bound approach has 2 major steps - branching and bounding. These 2 steps are repeated till the problem is solved.

Branching involves division of a given problem into two or more subproblems. These subproblems are similar to the original problem but smaller in size.

Let us assume that $f(x)$ is a objective function, S is the state-space tree that has all the solutions. Hence, S is a set of all solutions also called as feasible region. S is divided into k feasible subregions S_i. The union of these subregions will give S. i.e $S = \bigcup_{i=1}^{k} S_i$.

In the branch-and-bound technique, one has to search the state-space tree for finding optimal solution. Hence, it uses bounds for limiting the growth of the state-space tree exponentially.

Second step, bounding step aids in limiting the growth of the state space. In this step, the best solution of the subproblems is identified and used as lower bound.

For every node i of the state space, the lower bound needs to be calculated. Similarly, the upper bound needs to be computed. This includes the minimum amount of work required to compute the result, the best feasible solution.

In branching step, state space is divided into k feasible subregions, uses a lower bound for minimization problem and an upper bound for maximization problem.

The success of branch-and-bound algorithm depends on the quality of the lower and upper bounds. Implementation of the bounding step is more difficult than branching step.

Branch-and-bound technique terminates when:

1. The subproblem of given node gives an optimal solution.
2. The subproblem of given node yields a sub-optimal solution.
3. The subproblems turn out to be infeasible.

The objective of branch-and-bound is similar to that of backtracking method, difference is that the branch-and-bound technique calculates a bound of a possible solution at each and every stage.

This approach is used for a number of NP-hard problems like

Integer programming
Nonlinear programming
Travelling salesman problem (TSP)
0/1 knapsack problem etc.

Like the backtracking technique, branch-and-bound algorithms also construct a state-space tree and employ search techniques to search for optimal solution.

Differences between backtracking and branch-and-bound:

Backtracking	Branch-and-bound
uses only DFS technique	Uses BFS, DFS, least-cost search.
explores state-space trees partially, potential solutions may sometimes be ignored.	checks completely for an optimal solution, always potential solutions are obtained.
can be used for enumerative, decision and optimization problems.	can be used only for an optimization problem.

Backtracking:

Backtracking is a systematic method for searching or more solutions for a given problem. It is a redefined brute force technique used for solving problems. It was proposed by D.H. Lehmer and later refined by R.J. Walker.

Backtracking can effectively solve multi-decision problems, where the final solution is visualized as a set of decisions. The execution of a decision or choice leads to another set of decisions or choices. These decisions ~~gave be~~ encountered until an successful solution of a problem arrives. One can back-track and try other alternatives to achieve the aim. Thus the overall strategy may end in a successful or an unsuccessful outcome. A Backtracking design paradigm can solve following three types of problems:

Enumeration problems:

In an enumeration problem, all solutions are listed for a given problem.

Decision problems:

In an decision problem, a solution is given in terms of Yes/no.

Scanned with CamScanner

Optimization problems:

In optimization problems, optimal solutions are required, which maximize or minimize the given objective function as per constraints of given problem.

Example:

Consider a scenario of searching the torch light in a dark room. If one encounters a wall, it is a dead end. Hence, one has to then backtrack and continue the search process till the torch light is found if it is really present in the room. Other examples where backtracking can be used for puzzles such as mazes and sudoku. It can be observed that many trail and error processes are required to find solutions for these puzzles.

The brute force technique can solve a given problem by listing out all its possible solutions, from which the optimal solution can be picked. Brute force algorithm leads to exponential time complexity. The backtracking approach can solve most of these problems in polynomial time.

Scanned with CamScanner

A backtracking algorithm solves problems incrementally by adding candidate solutions till the final solution is obtained. If partial solution is not leading to a solution, then it is rejected along with its other partial solutions; this process is called domino principle. The backtracking process is continued till goal state is reached, otherwise the search is termed as unsuccessful. Backtracking is a depth first search with some bounding functions. Bounding functions represents constraints of given problem. First, the backtracking process defines a solution vector as n-tuple vector $(x_1, x_2 \ldots x_n)$ for the given problem. Hence n is number of components of solution vector ~~as A tuple vector $(x_1, x_2 \ldots$~~ for each x_i, where i ranges from 1 to n represented a partial solution. These partial solution components x_i are generated based on concept of constraints.

Scanned with CamScanner

Backtracking algorithm:

step 1: While there are many choices left out, perform step 2

step 2: Generate a state-space tree using DFS approach.

2a: check next configuration using bounding functions.

2b: If solution is promising then

if solution is obtained then

print the solution

else

backtrack to the parent of the node and try again.

step 3: end.

Complexity:

It is difficult to evaluate backtracking algorithm analytically. Donald E. knuth suggested a method where random path can be generated from root to a leaf of a statespace tree. if c_1 children are encountered for first compononent of solution vector and so on, Then no. of children encoured for solution of random vector is given by relation $T(n) = 1 + c_1 + c_1c_2 + \cdots + c_1c_2 \cdots c_n$.

Scanned with CamScanner

HAMITONIAN CYCLES

A Hamiltonian cycle, also called a Hamiltonian circuit, Hamilton cycle, or Hamilton circuit, is a graph cycle (i.e., closed loop) through a graph that visits each node exactly once . A graph possessing a Hamiltonian cycle is said to be a Hamiltonian graph. By convention, the singleton graph K_1 is considered to be Hamiltonian even though it does not possess a Hamiltonian cycle, while the connected graph on two nodes K_2 is not.

The Hamiltonian cycle is named after Sir William Rowan Hamilton, who devised a puzzle in which such a path along the polyhedron edges of an dodecahedron was sought (the Icosian game).

In general, the problem of finding a Hamiltonian cycle is NP-complete (Karp 1972; Garey and Johnson 1983, p. 199), so the only known way to determine whether a given general graph has a Hamiltonian cycle is to undertake an exhaustive search. Rubin (1974) describes an efficient search procedure that can find some or all Hamilton paths and circuits in a graph using deductions that greatly reduce backtracking and guesswork. A probabilistic algorithm due to Angluin and Valiant (1979), described by Wilf (1994), can also be useful to find Hamiltonian cycles and paths.

Determine whether a given graph contains Hamiltonian Cycle or not. If it contains, then prints the path. Following are the input and output of the required function.

Input:
A 2D array graph[V][V] where V is the number of vertices in graph and graph[V][V] is adjacency matrix representation of the graph. A value graph[i][j] is 1 if there is a direct edge from i to j, otherwise graph[i][j] is 0.
Output:
An array path[V] that should contain the Hamiltonian Path. path[i] should represent the ith vertex in the Hamiltonian Path. The code should also return false if there is no Hamiltonian Cycle in the graph.
For example, a Hamiltonian Cycle in the following graph is {0, 1, 2, 4, 3, 0}.

```
(0)--(1)--(2)
|  / \  |
```

```
| /  \ |
|/    \|
(3)-------(4)
```

And the following graph doesn't contain any Hamiltonian Cycle.

```
(0)--(1)--(2)
|  / \  |
| /  \ |
|/    \|
(3)     (4)
```

Naive Algorithm

Generate all possible configurations of vertices and print a configuration that satisfies the given constraints. There will be n! (n factorial) configurations.

```
while there are untried conflagrations
{
  generate the next configuration
  if ( there are edges between two consecutive vertices of this
     configuration and there is an edge from the last vertex to
     the first ).
  {
     print this configuration;
     break;
  }
}
```

Backtracking Algorithm

Create an empty path array and add vertex 0 to it. Add other vertices, starting from the vertex 1. Before adding a vertex, check for whether it is adjacent to the previously added vertex and not already added. If we find such a vertex, we add the vertex as part of the solution. If we do not find a vertex then we return false.

Implementation of Backtracking solution

Following are implementations of the Backtracking solution.

```
/* C program for solution of Hamiltonian Cycle problem
   using backtracking */
#include<stdio.h>
#define V 5

void printSolution(int path[]);

bool isSafe(int v, bool graph[V][V], int path[], int pos)
{

    if (graph [ path[pos-1] ][ v ] == 0)
        return false;

    for (int i = 0; i < pos; i++)
        if (path[i] == v)
            return false;

    return true;
}

bool hamCycleUtil(bool graph[V][V], int path[], int pos)
{
    if (pos == V){
        if ( graph[ path[pos-1] ][ path[0] ] == 1 )
            return true;
        else
            return false;
    }
    for (int v = 1; v < V; v++){
        if (isSafe(v, graph, path, pos)){
            path[pos] = v;
            if (hamCycleUtil (graph, path, pos+1) == true)
                return true
            path[pos] = -1;
        }
    }
    return false;
}
```

```
/* This function solves the Hamiltonian Cycle problem using
Backtracking.*/
bool hamCycle(bool graph[V][V])
{
   int *path = new int[V];
   for (int i = 0; i < V; i++)
      path[i] = -1;
   path[0] = 0;
   if ( hamCycleUtil(graph, path, 1) == false )
   {
      printf("\nSolution does not exist");
      return false;
   }

   printSolution(path);
   return true;
}
void printSolution(int path[])
{
   printf ("Solution Exists:"
         " Following is one Hamiltonian Cycle \n");
   for (int i = 0; i < V; i++)
      printf(" %d ", path[i]);
   printf(" %d ", path[0]);
   printf("\n");
}
int main()
{
  bool graph1[V][V] = {{0, 1, 0, 1, 0},
                       {1, 0, 1, 1, 1},
                       {0, 1, 0, 0, 1},
                       {1, 1, 0, 0, 1},
                       {0, 1, 1, 1, 0},
                      };

   hamCycle(graph1);
   bool graph2[V][V] = {{0, 1, 0, 1, 0},
                       {1, 0, 1, 1, 1},
                       {0, 1, 0, 0, 1},
```

```
                {1, 1, 0, 0, 0},
                {0, 1, 1, 0, 0},
               };
    hamCycle(graph2);

    return 0;
}
```

Output:

```
Solution Exists: Following is one Hamiltonian Cycle
0 1 2 4 3 0

Solution does not exist
```

The above code always prints cycle starting from 0. The starting point should not matter as the cycle can be started from any point. If you want to change the starting point, you should make two changes to the above code.
Change “path[0] = 0;” to “path[0] = s;” where s is your new starting point. Also change loop “for (int v = 1; v < V; v++)" in hamCycleUtil() to "for (int v = 0; v < V; v++)".

TRAVELLING SALESMAN PROBLEM

The following graph shows a set of cities and distance between every pair of cities-If salesman starting city is A, then a TSP tour in the graph is-

A → B → D → C → A

Cost of the tour

= 10 + 25 + 30 + 15

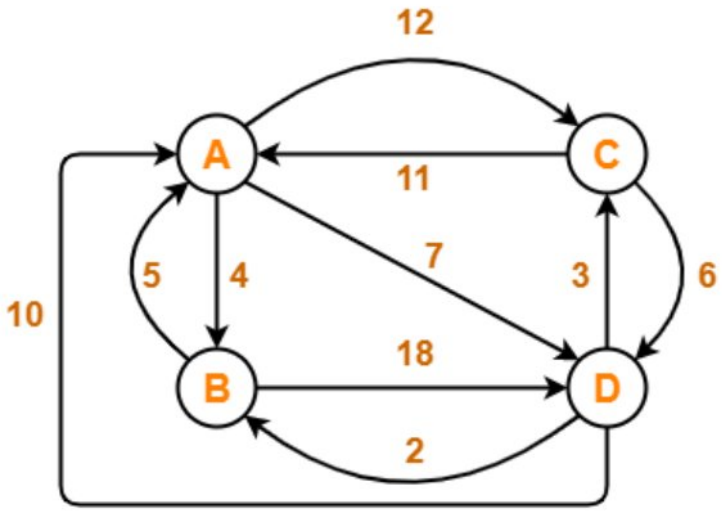
= **80 units**

In this article, we will discuss how to solve travelling salesman problem using branch and bound approach with example.

PRACTICE PROBLEM BASED ON TRAVELLING SALESMAN PROBLEM USING BRANCH AND BOUND APPROACH-

Problem-

Solve Travelling Salesman Problem using Branch and Bound Algorithm in the following graph-



Solution-

Step-01:

Write the initial cost matrix and reduce it-

	A	B	C	D
A	∞	4	12	7
B	5	∞	∞	18
C	11	∞	∞	6
D	10	2	3	∞

After row reduction Performing

this, we obtain the following row-reduced matrix-

	A	B	C	D
A	∞	0	8	3
B	0	∞	∞	13
C	5	∞	∞	0
D	8	0	1	∞

Column Reduction-

Performing this, we obtain the following column-reduced matrix-

	A	B	C	D
A	∞	0	7	3
B	0	∞	∞	13
C	5	∞	∞	0
D	8	0	0	∞

Finally, the initial distance matrix is completely reduced.

Now, we calculate the cost of node-1 by adding all the reduction elements.

Cost(1)

= Sum of all reduction elements

= 4 + 5 + 6 + 2 + 1

= 18

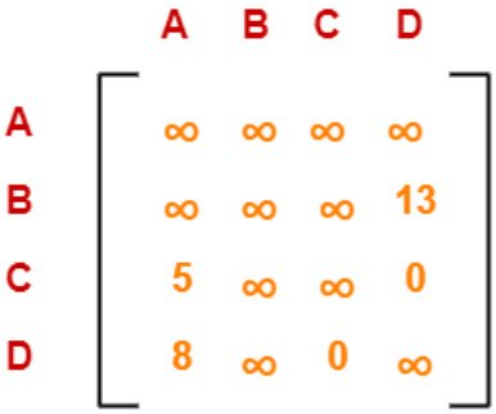
Step-02:

- We consider all other vertices one by one.
- We select the best vertex where we can land upon to minimize the tour cost.

Choosing To Go To Vertex-B: Node-2 (Path A → B)

- From the reduced matrix of step-01, M[A,B] = 0
- Set row-A and column-B to ∞
- Set M[B,A] = ∞

Now, resulting cost matrix is-



Row Reduction-

- We can not reduce row-1 as all its elements are ∞.
- Reduce all the elements of row-2 by 13.
- There is no need to reduce row-3.
- There is no need to reduce row -4

Column Reduction-

- Reduce the elements of column-1 by 5.
- We can not reduce column-2 as all its elements are ∞.
- There is no need to reduce column-3.
- There is no need to reduce column -4

Finally, the matrix is completely reduced.

Now, we calculate the cost of node-2.

Cost(2)

= Cost(1) + Sum of reduction elements + M[A,B]

= 18 + (13 + 5) + 0

= 36

Choosing To Go To Vertex-C: Node-3 (Path A → C)

- From the reduced matrix of step-01, M[A,C] = 7

- Set row-A and column-C to ∞
- Set M[C,A] = ∞

Now, resulting cost matrix is-

Now,

- We reduce this matrix.
- Then, we find out the cost of node-03.

Row Reduction-

- We can not reduce row-1 as all its elements are ∞.
- There is no need to reduce row-2.
- There is no need to reduce row-3.
- There is no need to reduce row-4.

Thus, the matrix is already row-reduced.

Column Reduction-

- There is no need to reduce column-1.
- There is no need to reduce column-2.
- We can not reduce column-3 as all its elements are ∞.
- There is no need to reduce column-4.

Thus, the matrix is already column reduced.

Finally, the matrix is completely reduced.

Now, we calculate the cost of node-3.

Cost(3)

= Cost(1) + Sum of reduction elements + M[A,C]

= 18 + 0 + 7

= 25

Choosing To Go To Vertex-D: Node-4 (Path A → D)

- From the reduced matrix of step-01, M[A,D] = 3
- Set row-A and column-D to ∞
- Set M[D,A] = ∞

Now, resulting cost matrix is-

	A	B	C	D
A	∞	∞	∞	∞
B	0	∞	∞	∞
C	5	∞	∞	∞
D	∞	0	0	∞

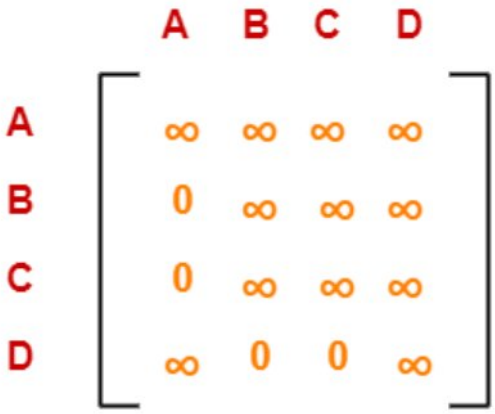
Now,

- We reduce this matrix.
- Then, we find out the cost of node-04.

Row Reduction-

- We can not reduce row-1 as all its elements are ∞.
- There is no need to reduce row-2.
- Reduce all the elements of row-3 by 5.
- There is no need to reduce row-4.

Performing this, we obtain the following row-reduced matrix-



Column Reduction-

- There is no need to reduce column-1.
- There is no need to reduce column-2.
- There is no need to reduce column-3.
- We can not reduce column-4 as all its elements are ∞.

Thus, the matrix is already column-reduced.

Finally, the matrix is completely reduced.

Now, we calculate the cost of node-4.

Cost(4)

= Cost(1) + Sum of reduction elements + M[A,D]

= 18 + 5 + 3

= 26

Thus, we have-

- Cost(2) = 36 (for Path A → B)
- Cost(3) = 25 (for Path A → C)
- Cost(4) = 26 (for Path A → D)

We choose the node with the lowest cost.

Since cost for node-3 is lowest, so we prefer to visit node-3.

Thus, we choose node-3 i.e. path **A → C**.

Step-03:

We explore the vertices B and D from node-3.

We now start from the cost matrix at node-3 which is- We now start from the cost matrix at node-3 which is-

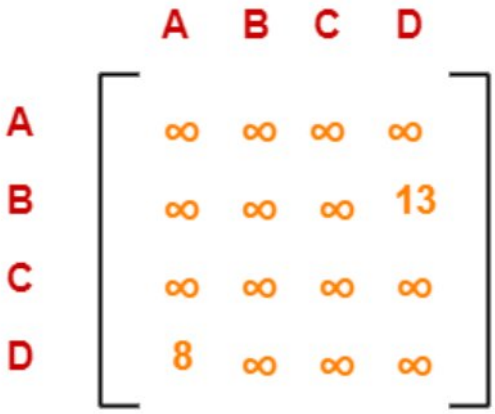
	A	B	C	D
A	∞	∞	∞	∞
B	0	∞	∞	13
C	∞	∞	∞	0
D	8	0	∞	∞

Cost(3) = 25

Choosing To Go To Vertex-B: Node-5 (Path A → C → B)

- From the reduced matrix of step-02, M[C,B] = ∞
- Set row-C and column-B to ∞
- Set M[B,A] = ∞

Now, resulting cost matrix is-



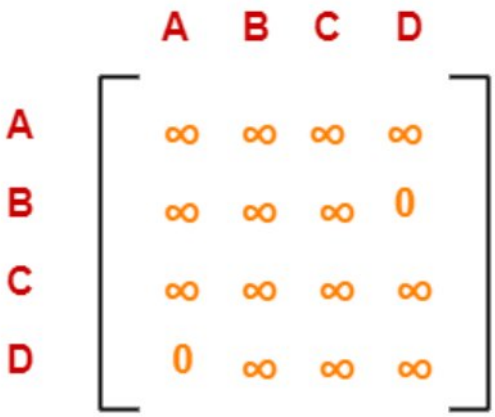
Now,

- We reduce this matrix.
- Then, we find out the cost of node-5.

Row Reduction-

- We can not reduce row-1 as all its elements are ∞.
- Reduce all the elements of row-2 by 13.
- We can not reduce row-3 as all its elements are ∞.
- Reduce all the elements of row-4 by 8.

Performing this, we obtain the following row-reduced matrix-



Column Reduction-

- There is no need to reduce column-1.
- We can not reduce column-2 as all its elements are ∞.
- We can not reduce column-3 as all its elements are ∞.
- There is no need to reduce column-4.

Thus, the matrix is already column reduced.

Finally, the matrix is completely reduced.

Now, we calculate the cost of node-5.

Cost(5)

= cost(3) + Sum of reduction elements + M[C,B]

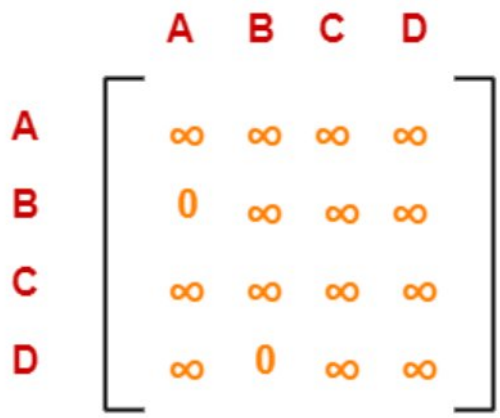
= 25 + (13 + 8) + ∞

= ∞

Choosing To Go To Vertex-D: Node-6 (Path A → C → D)

- From the reduced matrix of step-02, M[C,D] = ∞
- Set row-C and column-D to ∞
- Set M[D,A] = ∞

Now, resulting cost matrix is-



Now,

- We reduce this matrix.
- Then, we find out the cost of node-6.

Row Reduction-

- We can not reduce row-1 as all its elements are ∞.
- There is no need to reduce row-2.
- We can not reduce row-3 as all its elements are ∞.
- We can not reduce row-4 as all its elements are ∞.

Thus, the matrix is already row reduced.

Column Reduction-

- There is no need to reduce column-1.
- We can not reduce column-2 as all its elements are ∞.
- We can not reduce column-3 as all its elements are ∞.
- We can not reduce column-4 as all its elements are ∞.

Thus, the matrix is already column reduced.

Finally, the matrix is completely reduced.

Now, we calculate the cost of node-6.

Cost(6)

= cost(3) + Sum of reduction elements + M[C,D]

= 25 + 0 + 0

= 25

Thus, we have-

- Cost(5) = ∞ (for Path A → C → B)
- Cost(6) = 25 (for Path A → C → D)

We choose the node with the lowest cost.

Since cost for node-6 is lowest, so we prefer to visit node-6.

Thus, we choose node-6 i.e. path **C → D**.

Step-04:

We explore vertex B from node-6.

We start with the cost matrix at node-6 which is-

	A	B	C	D
A	∞	∞	∞	∞
B	0	∞	∞	∞
C	∞	∞	∞	∞
D	∞	0	∞	∞

Cost(6) = 25

Choosing To Go To Vertex-B: Node-7 (Path A → C → D → B)

- From the reduced matrix of step-03, M[D,B] = 0
- Set row-D and column-B to ∞
- Set M[B,A] = ∞

Now, resulting cost matrix is-

	A	B	C	D
A	∞	∞	∞	∞
B	∞	∞	∞	∞
C	∞	∞	∞	∞
D	∞	∞	∞	∞

Now,

- We reduce this matrix.
- Then, we find out the cost of node-7.

Row Reduction-

- We can not reduce row-1 as all its elements are ∞.
- We can not reduce row-2 as all its elements are ∞.
- We can not reduce row-3 as all its elements are ∞.
- We can not reduce row-4 as all its elements are ∞.

Column Reduction-

- We can not reduce column-1 as all its elements are ∞.
- We can not reduce column-2 as all its elements are ∞.
- We can not reduce column-3 as all its elements are ∞.

- We can not reduce column-4 as all its elements are ∞.

Thus, the matrix is already column reduced.

Finally, the matrix is completely reduced.

All the entries have become ∞.

Now, we calculate the cost of node-7.

Cost(7)

= cost(6) + Sum of reduction elements + M[D,B]

= 25 + 0 + 0

= 25

Thus,

- Optimal path is: **A → C → D → B → A**
- Cost of Optimal path = **25 units**

0/1 Knapsack using Branch and Bound

- Branch and bound is an algorithm design paradigm which is generally used for solving combinatorial optimization problems. These problems typically exponential in terms of time complexity and may require exploring all possible permutations in worst case. Branch and Bound solve these problems relatively quickly.
- The branch-and-bound technique can be applied to the Knapsack problem. The Knapsack problem is about filling a knapsack with 'n' items. Each item is associated with a profit v1 and weight w1.
- The procedure for solving the knapsack problem using the branch-and-bound technique is:

  Step 1: Arrange the items such that v1/w1>=v2/w2>=……>=vn/wn.

  Step 2: construct the state space tree as a binary tree. Take a node; branching to the left indicates that the item is included and branching to the right indicates that the item is excluded.

  Step 3: Compute the Lower bound as follows :

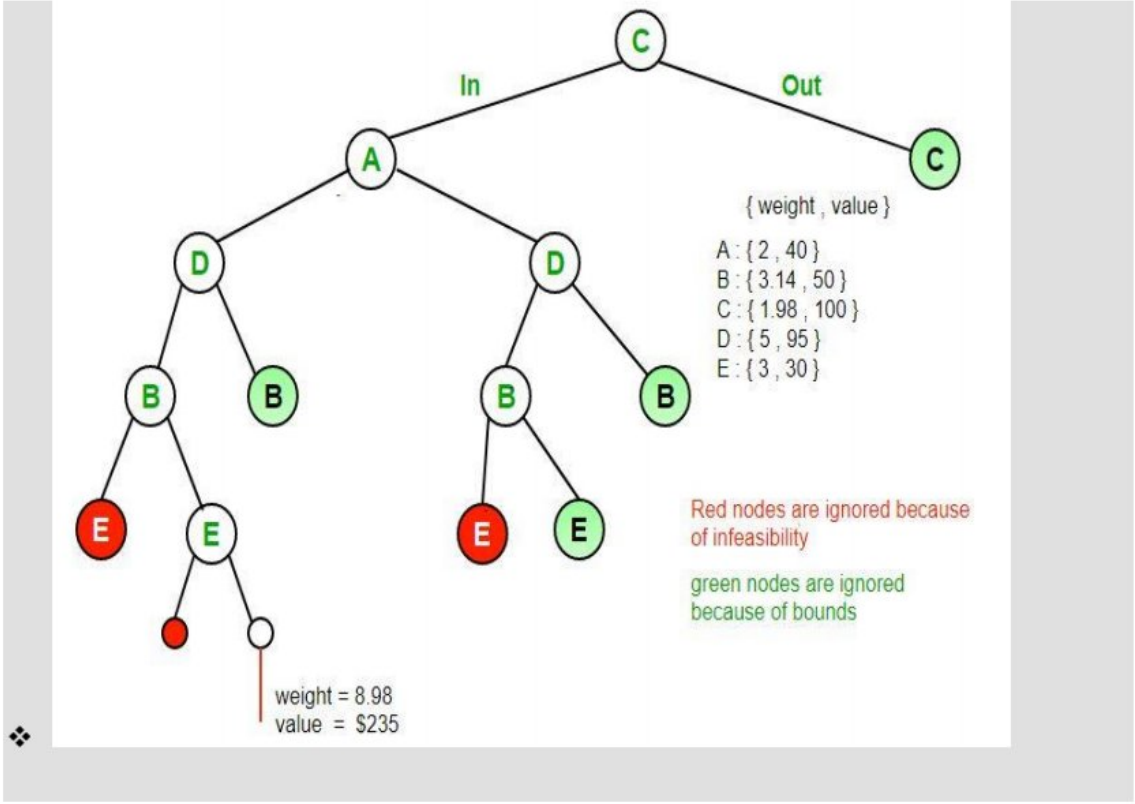
  ub=V+(W-w)*Vi+1/Wi+1.

  Here ,W is the capacity of the knapsack ,w is the weight of the item; And Vi+1 and Wi+1 are the value and weight of the next item,respectively.
- We can do better (than backtracking) if we know a bound on best possible solution subtree rooted with every node. If the best in subtree is worse than current best, we can simply ignore this node and its subtrees. So we compute bound (best solution) for every node and compare the bound with current best solution before exploring the node.

```
Input:
// First thing in every pair is weight of item
// and second thing is value of item
Item arr[] = {{2, 40}, {3.14, 50}, {1.98, 100},
              {5, 95}, {3, 30}};
Knapsack Capacity W = 10

Output:
The maximum possible profit = 235

Below diagram shows illustration. Items are
considered sorted by value/weight.
```

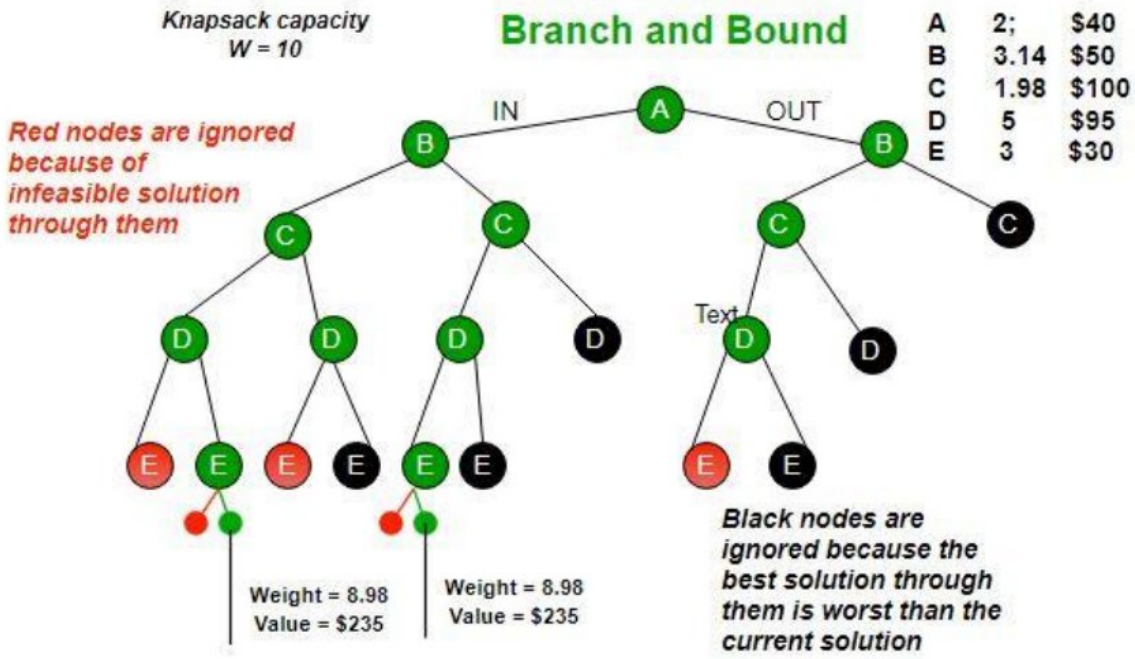


- **Example bounds used in below diagram are, A down can give $315, B down can $275, C down can $225, D down can $125 and E down can $30.**
- To check if a particular node can give us a better solution or not, we compute the optimal solution (through the node) using Greedy approach. If the solution computed by Greedy approach itself is more than the best so far, then we can't get a better solution through the node.

Complete Algorithm:

1. Sort all items in decreasing order of ratio of value per unit weight so that an upper bound can be computed using Greedy Approach.
2. Initialize maximum profit, maxProfit = 0
3. Create an empty queue, Q.
4. Create a dummy node of decision tree and enqueue it to Q. Profit and weight of dummy node are 0.
5. Do following while Q is not empty.
   - Extract an item from Q. Let the extracted item be u.
   - Compute profit of next level node. If the profit is more than maxProfit, then update maxProfit.
   - Compute bound of next level node. If bound is more than maxProfit, then add next level node to Q.

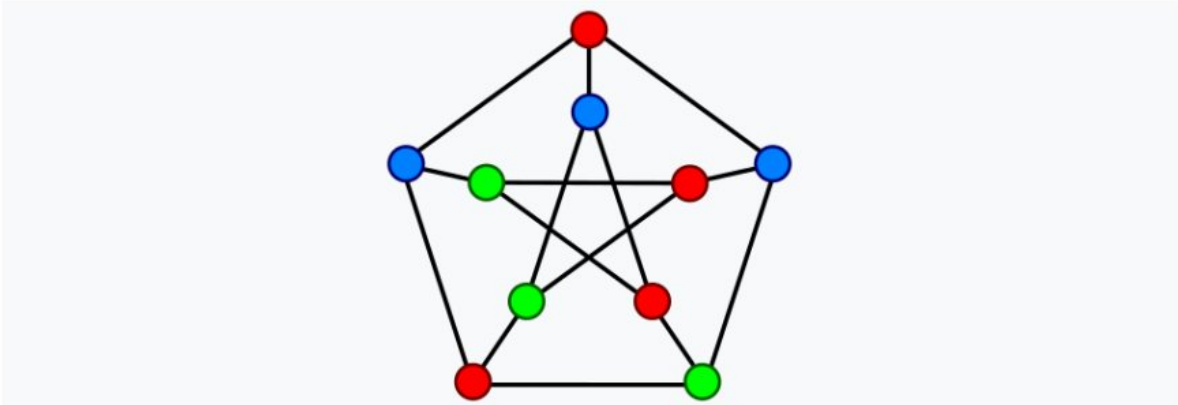
- Consider the case when next level node is not considered as part of solution and add a node to queue with level as next, but weight and profit without considering next level nodes.



GRAPH COLORING

In this problem, an undirected graph is given. There is also provided m colors. The problem is to find if it is possible to assign nodes with m different colors, such that no two adjacent vertices of the graph are of the same colors. If the solution exists, then display which color is assigned on which vertex.

Starting from vertex 0, we will try to assign colors one by one to different nodes. But before assigning, we have to check whether the color is safe or not. A color is not safe whether adjacent vertices are containing the same color.



A proper vertex coloring of the graph with 3 colors, the minimum number possible.

In graph theory, **graph coloring** is a special case of **graph labeling**; it is an assignment of labels traditionally called "colors" to elements of a graph subject to certain constraints. In its simplest form, it is a way of coloring the vertices of a graph such that no two adjacent vertices are of the same color; this is called a **vertex coloring**.

The problem can be solved as

• Assign colors to the vertices of a graph so that no adjacent vertices share the same color – Vertices i, j are adjacent if there is an edge from vertex i to vertex j.

• Find all m-colorings of a graph – Find all ways to color a graph with at most m colors. – m is called chromatic number

• Assign colors to the vertices of a graph so that no adjacent vertices share the same color – Vertices i, j are adjacent if there is an edge from vertex i to vertex j.

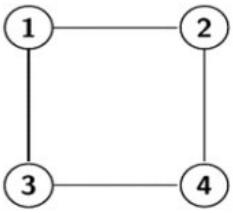
• Find all m-colorings of a graph – Find all ways to color a graph with at most m colors. – m is called chromatic number

Graph coloring Them-Coloring problem Finding all ways to color an undirected graph using at most m different colors, so that no two adjacent vertices are the same color.

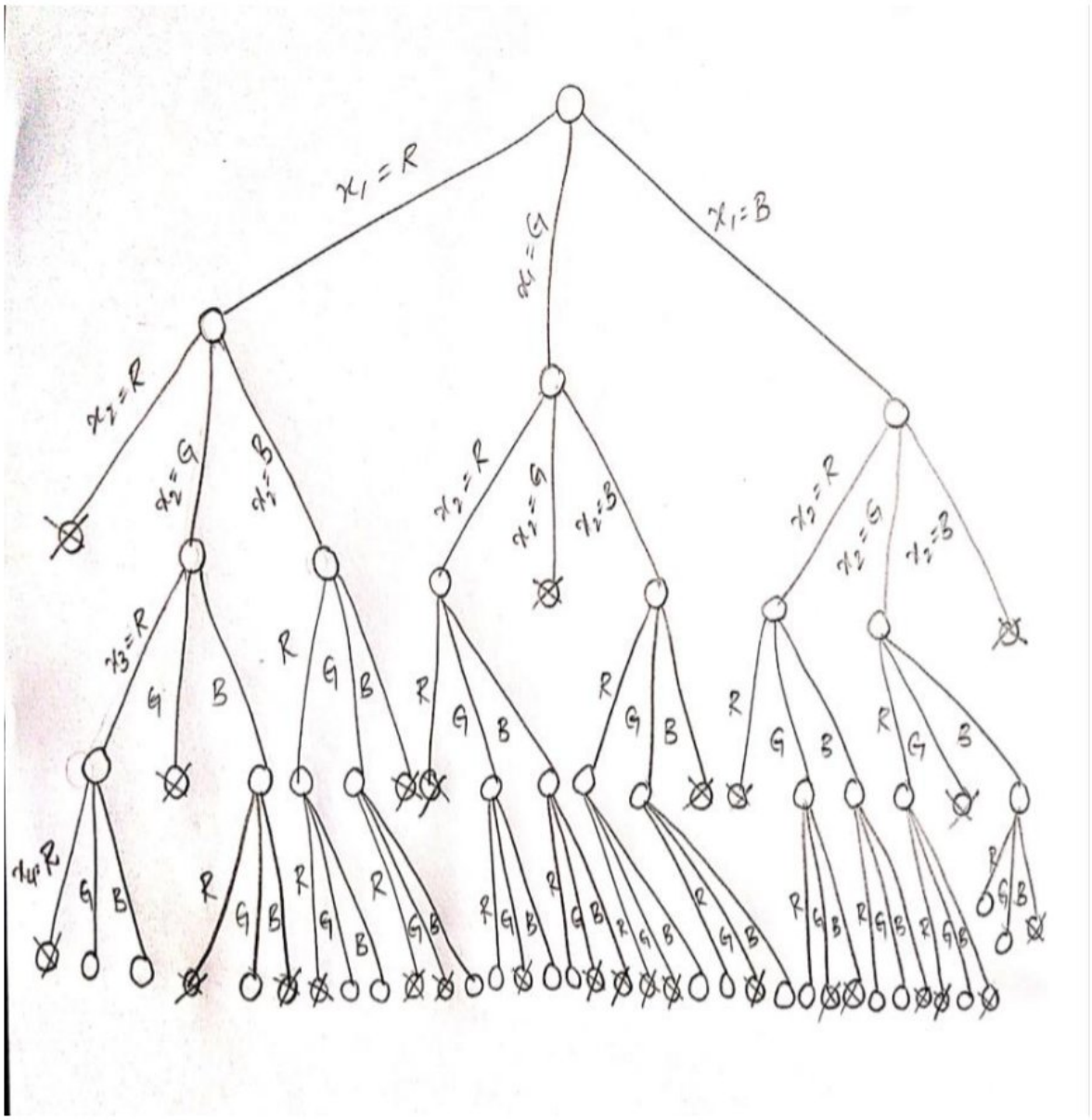
Usually the m-Coloring problem consider as a unique problem for each value of m.

GRAPH COLORING

Consider a graph having four vertices. Now we have to color this graph with given three colors red, green and blue.



m = 3 , that means number of colors



GRAPH COLORING

The possible combinations by using three colors are

RGRG	RGRB	RGBG
RBRG	RBRB	RBGB
GRGR	GRGB	GRBR
GBRB	GBGR	GBGB
BRGR	BRBR	BRBG
BGRG	BGBR	BGBG

Applications of graph coloring:

- Making schedule or time table.
- Mobile radio frequency assignment.
- Sudoku.
- Register allocation.
- Bipartite graph.
- Map coloring.s

GRAPH COLORING

0/1 Knapsack Problem using Branch and Bound

- In knapsack problem, consider 'n' no of objects each object having a profit 'p_i', weight 'w_i' and a knapsack capacity of 'm'.
- The objective is to place this objects 'n' into the knapsack 'm' to get maximum profit i.e., $\sum p_i x_i$ is maximum where x_i is to check whether the object is placed into knapsack or not.

i.e., $x_i = 1$ – object is placed or included.

$x_i = 0$ – object is not placed or not included.

Knapsack problem is maximization problem but whereas Branch and Bound is a minimization problem. By using minimization problem finding the solution to the maximization problem.

To solve this problem first the process is converted into negatives. So, whenever the positive profits are converted into negative profits, we are getting minimization problem.

Branch and Bound is solved in two ways:

1. Least Cost (LC)
2. First in First Out (FIFO)

0/1 Knapsack Problem using LC Branch and Bound:

Least Cost means at each stage having two decisions i.e., consider the objects or not.

Example:

Consider objects n = 4 and capacity m = 15

$(P_1, P_2, P_3, P_4) = (10,10,12,18)$

$(W_1, W_2, W_3, W_4) = (2,4,6,9)$

Step 1: Convert the profit into negatives

$(P_1, P_2, P_3, P_4) = (-10, -10, -12, -18)$

Step 2: Calculate Lower Bound and Upper Bound for each and every node.

- In Lower Bound fractions are allowed and it is represented by **ĉ**, where as in Upper Bound fractions are not allowed and represented by **û.**

Node 1:

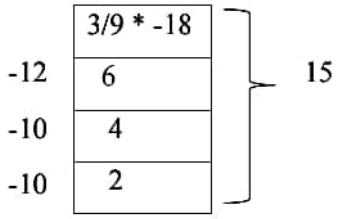
Given Knapsack having capacity 15, object 1 weight is 2 placed into knapsack getting profit

-10.

After placing first object remaining space is 13 so place object 2 into knapsack getting profit -10 and object 3 is placed into the knapsack getting profit -12.

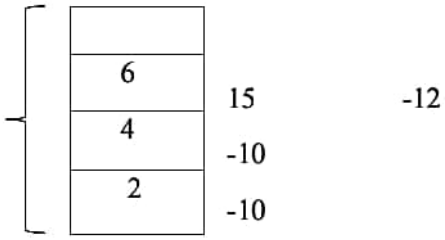
After placing object 3 remaining space is 3 but object 4 weight is 9 so complete object is not placed into the knapsack because given capacity is 15.

To calculate lower bound, fractions are allowed i.e., $\left\lceil \frac{\text{remaining space}}{\text{weight of the object}} * \text{profit} \right\rceil$



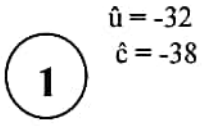
Lower Bound = -32-6 = -38

Similarly, Calculate Upper Bound for node 1 place object 1 getting profit -10, object 2 getting profit -10, object 3 getting profit -12 and remaining space is only 3 but object 4 weight is 9. So, object 4 is not placed into the knapsack.

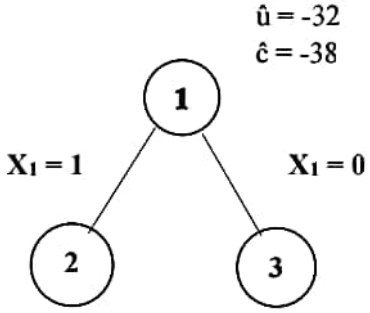


Upper Bound = -32

- In LB the space is not free, but in UB if there is a space, ignore the space.

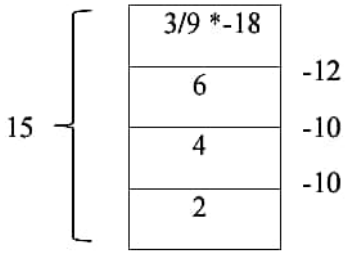


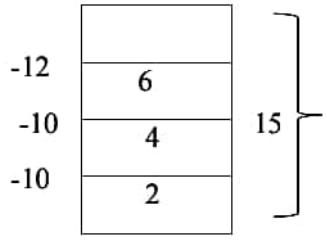
- From node 1, check whether object 1 is placed into the knapsack or not.



Node 2:

In node 2, $X_1 = 1$ i.e., object 1 should be placed 1st in the knapsack. After placing the 1st object, if any space is available place remaining objects into knapsack.





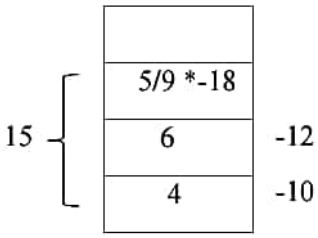
Lower Bound = -32-6 = -38

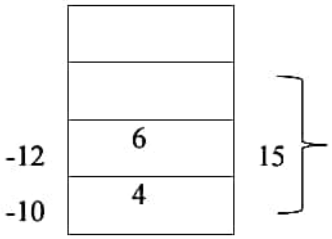
Upper Bound = -32

Node 3:

In node 3, $X_1 = 0$ i.e., first object is not placed into the knapsack. Place 2nd object getting profit -10, 3rd object getting profit -12 and remaining space is 5 so, (5/9) * -18 =

-10.

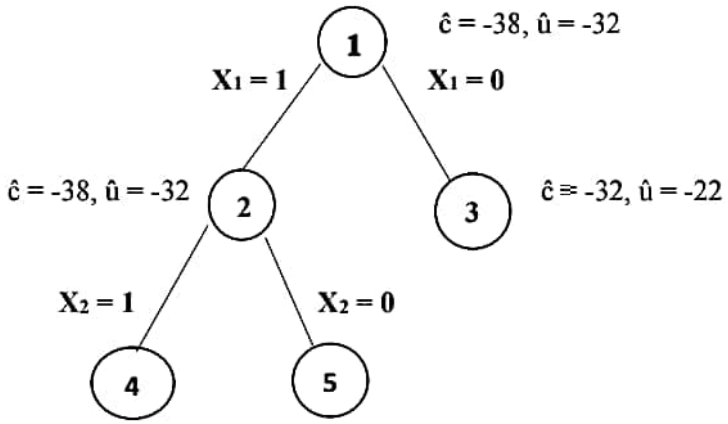




Lower Bound = -22-10 ≅ -32

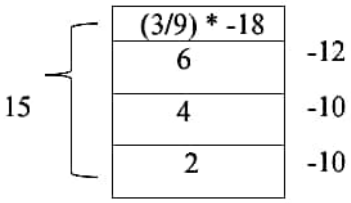
Upper Bound ≅ -22

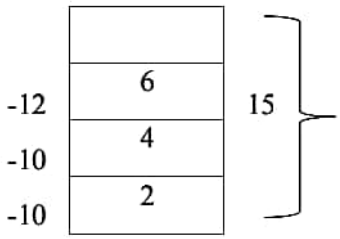
Consider minimum lower bound among node 2 and 3. Here node 2 has LB = -38 and node 3 has LB ≅ -32. consider node 2 because -38 is smaller than -32.



Node 4:

In node 4, $X_1 = 1$, $X_2 = 1$ i.e., compulsory 1st and 2nd object should be placed into the knapsack. If any space available then only place remaining objects.



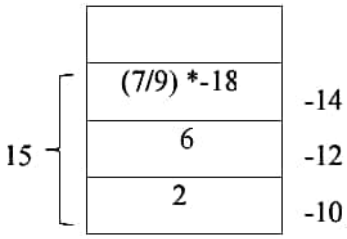


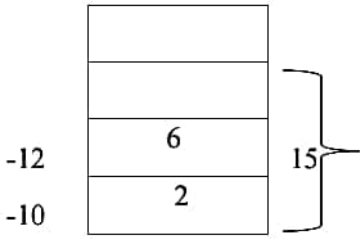
Lower Bound = -32-6 = -38

Upper Bound = -32

Node 5:

In node 5 $X_1 = 1$, $X_2 = 0$ i.e., 1st object is placed into knapsack and ignore the 2nd object and moves to next object.

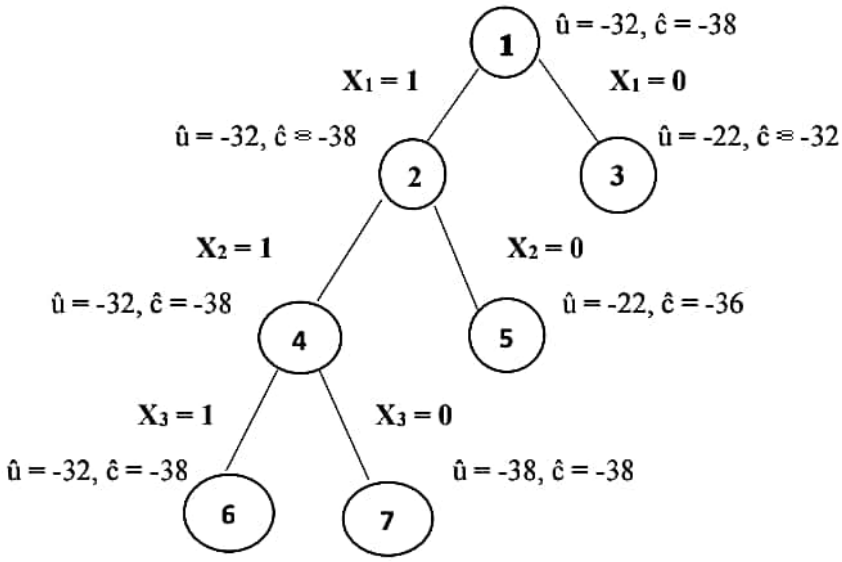




Lower Bound = -22 – 14 = -36

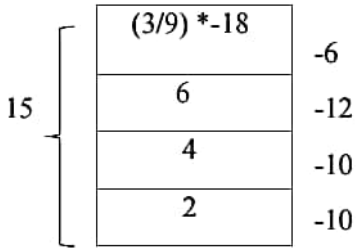
Upper Bound = -22

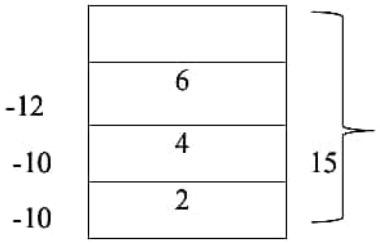
- Among nodes 4 and node 5, node 4 having minimum lower bound = -32



Node 6:

In node 6 $X_1 = 1$, $X_2 = 1$, $X_3 = 1$ i.e., 1st ,2nd and 3rd objects are placed into the knapsack.



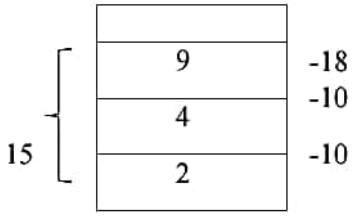


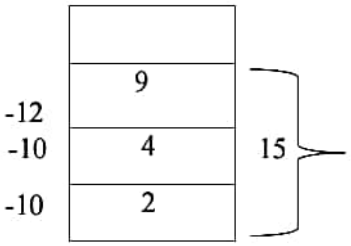
Lower Bound = -32 – 6 = -38

Upper Bound = -32

Node 7:

In node 7 $X_1 = 1$, $X_2 = 1$, $X_3 = 0$ i.e., 1st and 2nd object are placed into the knapsack. Ignore 3rd object, if any space is available place 4th object.

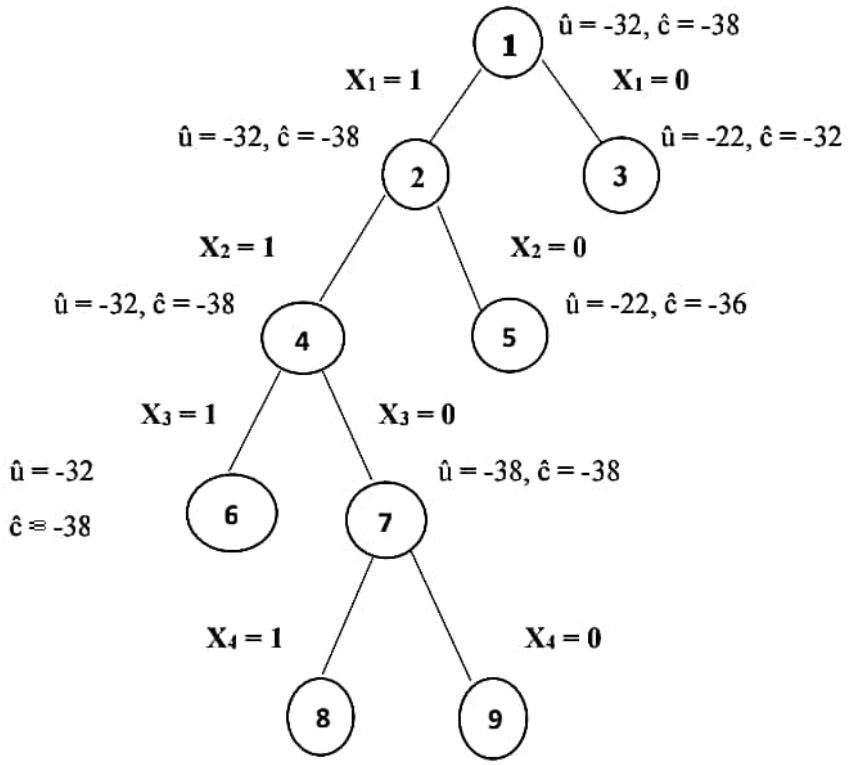




Lower Bound = -38

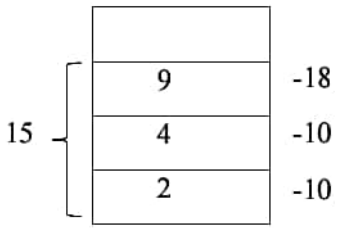
Upper Bound = -38

Among node 6 & 7, both node 6 & 7 lower bound are equal. If both lower bounds are equal then consider lower upper bound. Here. node 7 having lower upper bound = -38.

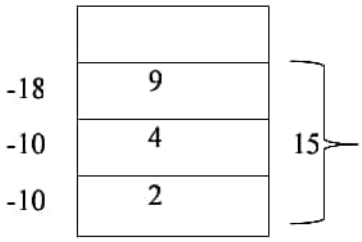


Node 8:

In node 8 $X_1 = 1$, $X_2 = 1$, $X_3 = 0$, $X_4 = 1$ i.e., 1st, 2nd and 4th objects are placed into the knapsack. Ignore the 3rd object.



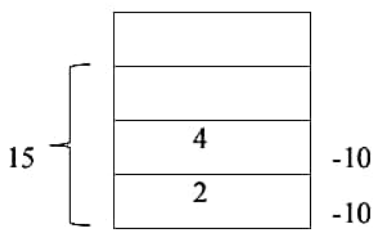
Lower Bound = -38

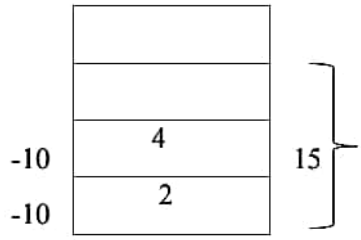


Upper Bound = -38

Node 9:

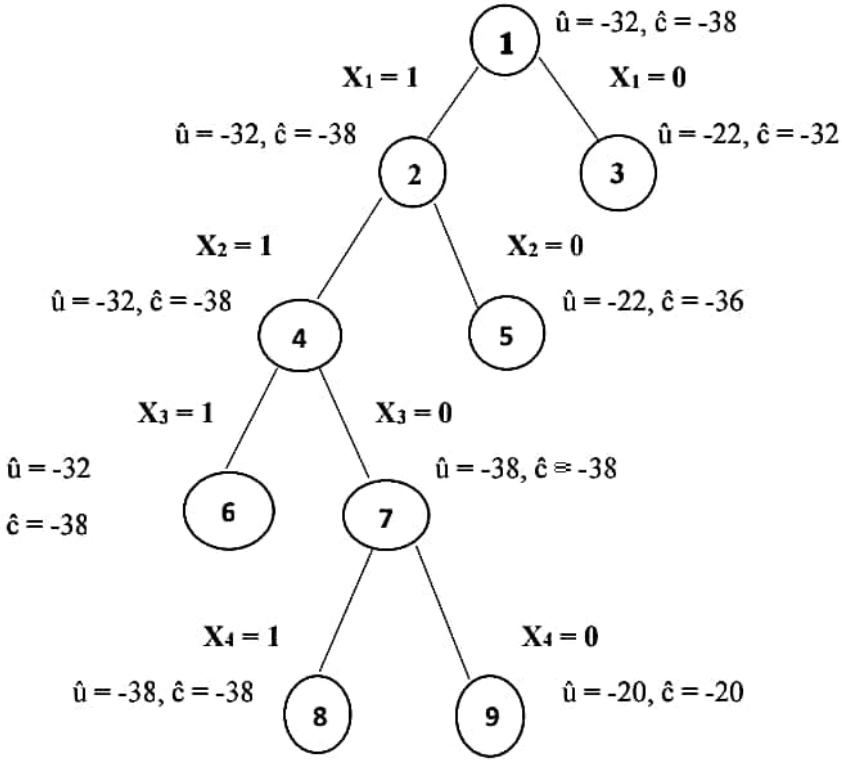
In node 9 $X_1 = 1$, $X_2 = 1$, $X_3 = 0$, $X_4 = 0$ i.e., 1st and 2nd objects are placed into the knapsack. Ignore the 3rd and 4th object and if place is available then simply ignore.





Lower Bound = -20

Upper Bound = -20



- Among nodes 8 and node 9, node 8 having minimum lower bound = -38. So, select node 8 and ignore node 9.
- Therefore, Knapsack instances are $(X_1, X_2, X_3, X_4) = (1, 1, 0, 1)$ and

  $(P_1, P_2, P_3, P_4) = (-10, -10, 0, -18)$
- Maximum profit = (-10) + (-10) + 0 + (-18) = -38. Since branch and bound is minimization problem so total profit = 38.