

# Design and Analysis of Algorithms

## **Unit 4: Backtracking and Branch-n-Bound**

# Unit 4 -Syllabus

**Backtracking:** Principle, control abstraction, time analysis of control abstraction, 8-queen problem, graph coloring problem, sum of subsets problem.

**Branch-n-Bound:** Principle, control abstraction, time analysis of control abstraction, strategies FIFO, LIFO and LC approaches, TSP, knapsack problem.

**Case Study:** Airline Crew Scheduling

# Backtracking

1. Backtracking is an algorithmic technique whose goal is to use **brute force** to find all solutions to a problem
2. To represent solution it uses **State Space Tree**.

## Example Backtracking Approach

**Problem:** We want to find all the possible ways of arranging 2 boys and 1 girl on 3 benches.

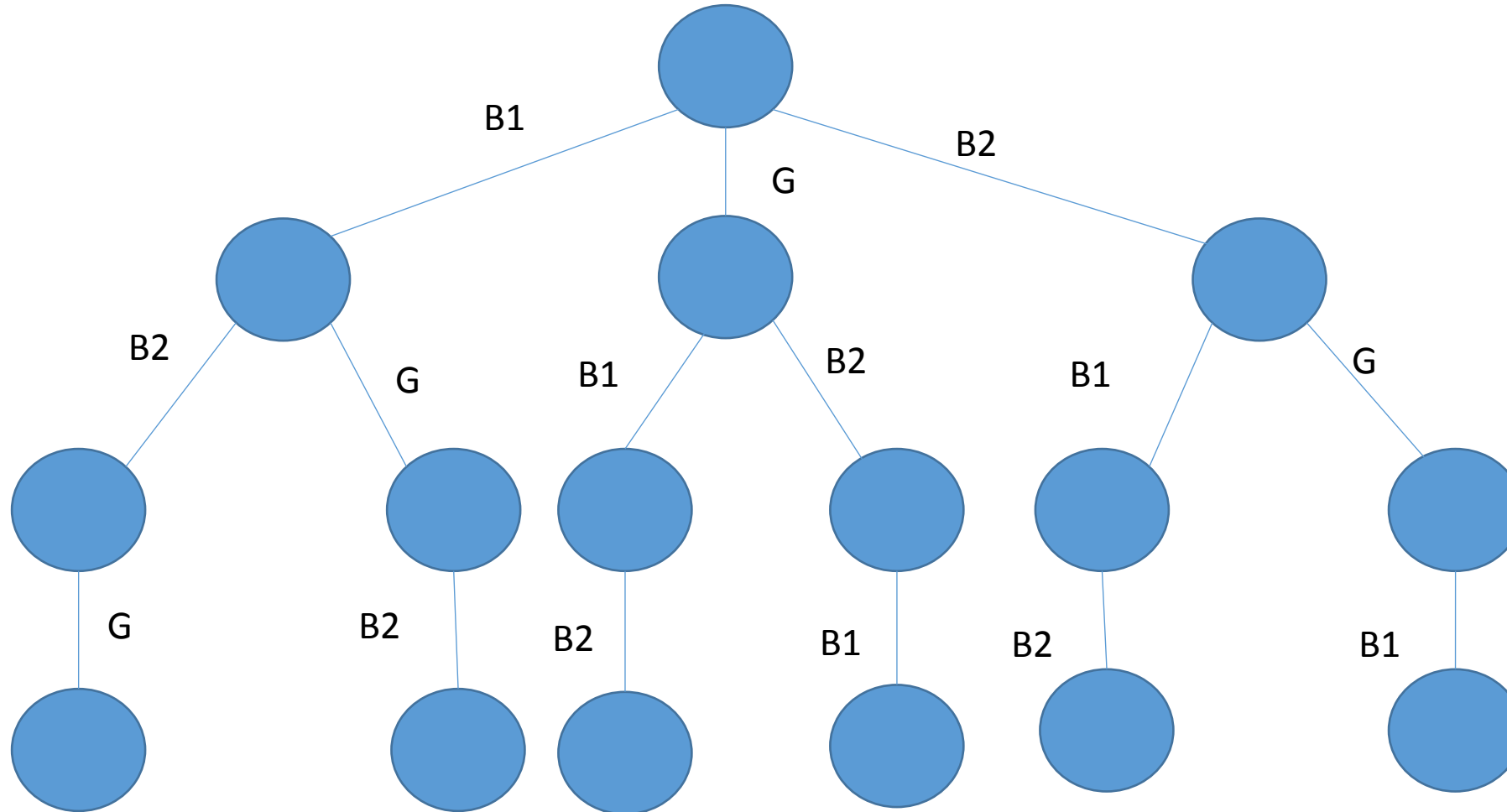
**Solution:** There are a total of  $3! = 6$  possibilities.

We will try all the possibilities and get the possible solutions.

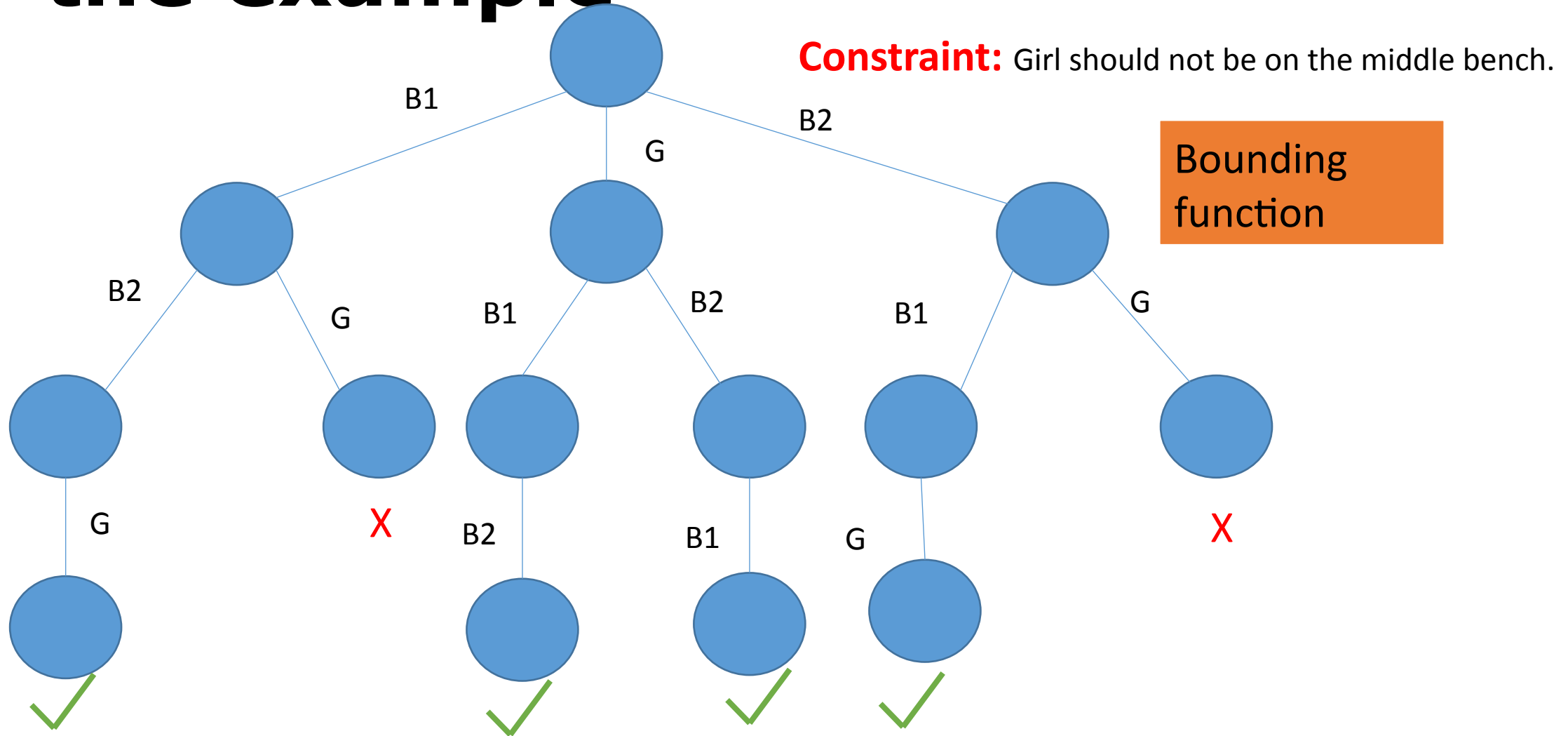
We recursively try all the possibilities.



# State Space Tree for example

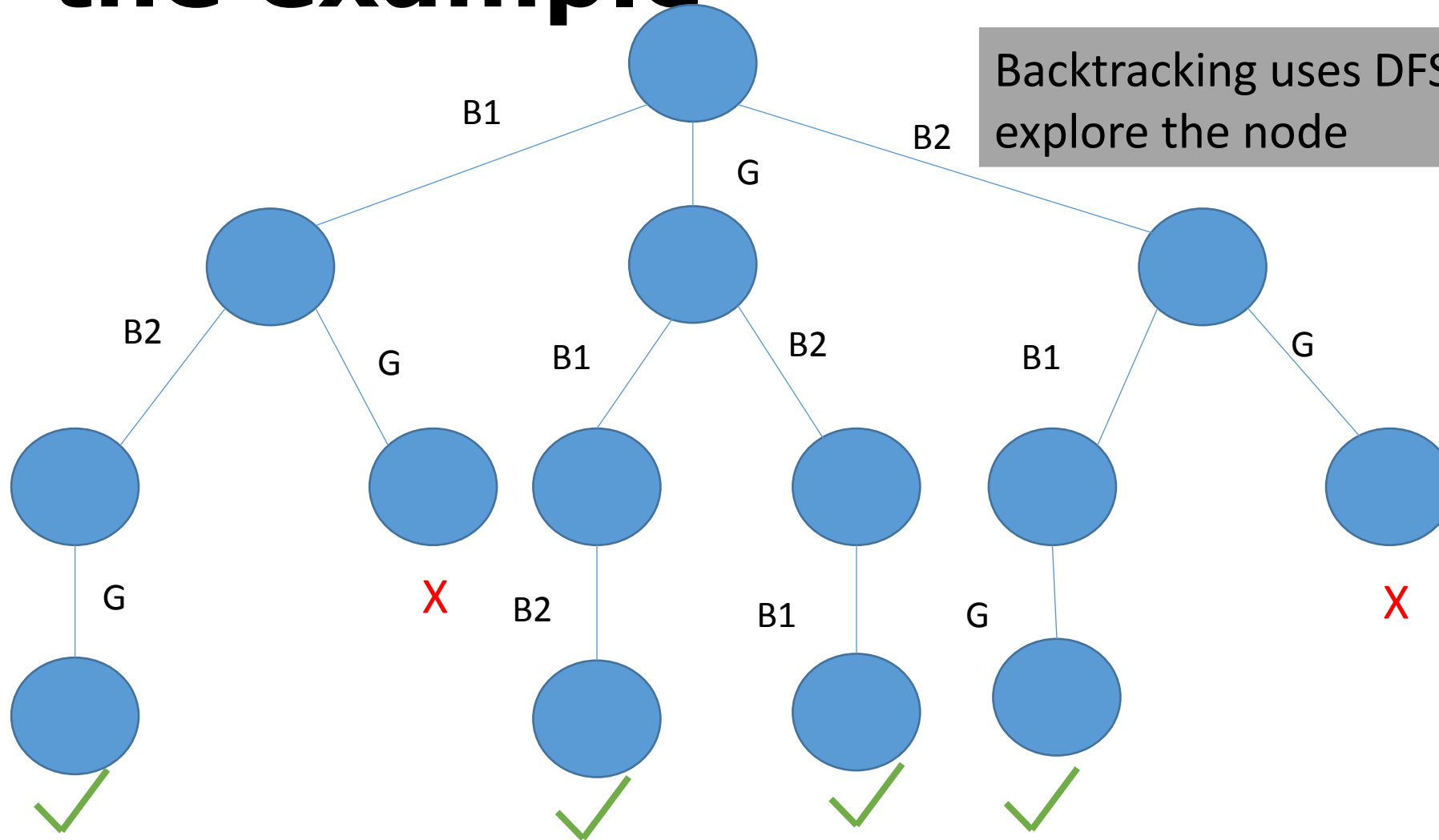


# Backtracking Approach to solve the example

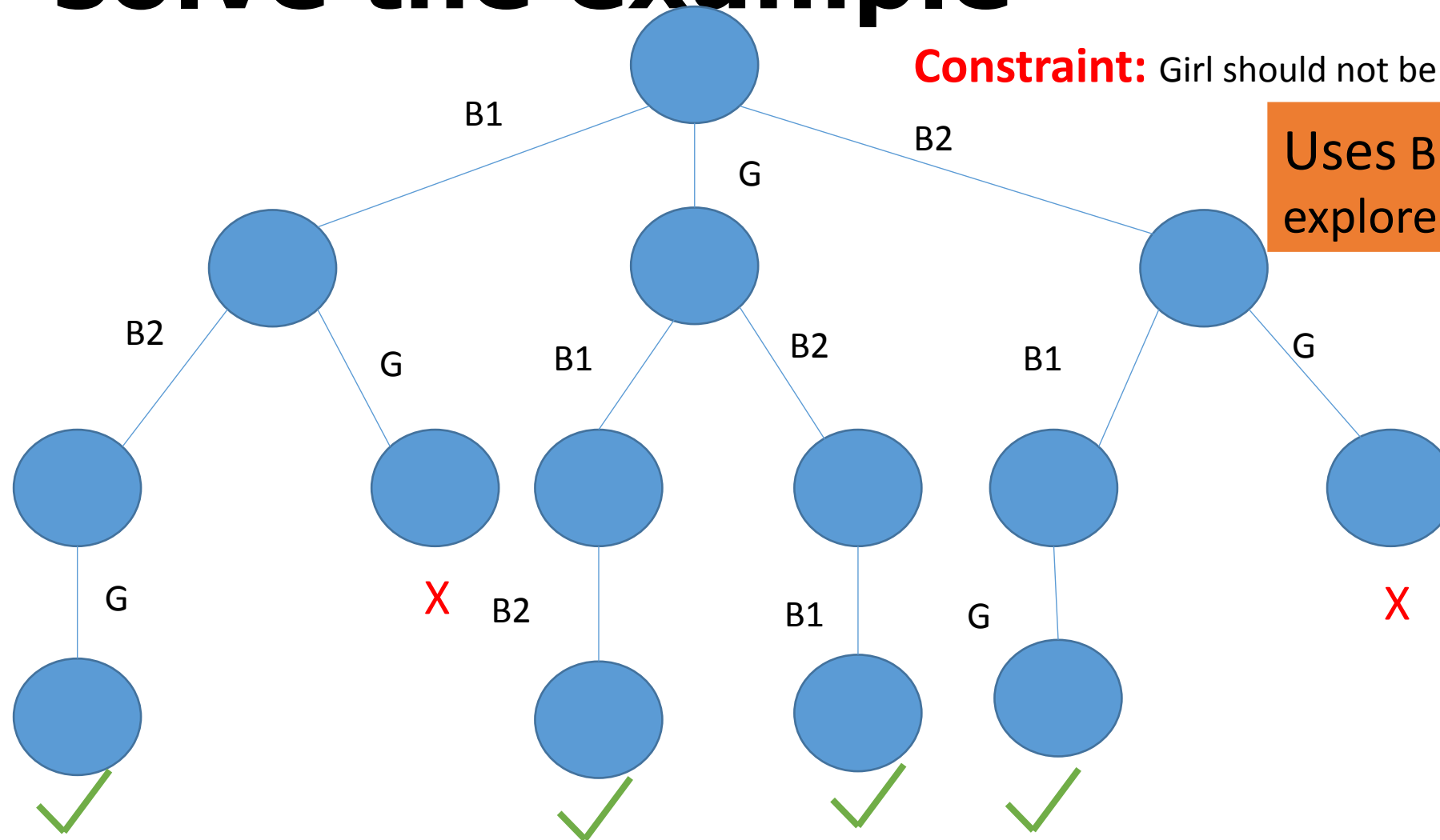


**Constraint:** Girl should not be on the middle bench.

Backtracking uses DFS approach to explore the node



# Branch and Bound Approach to solve the example

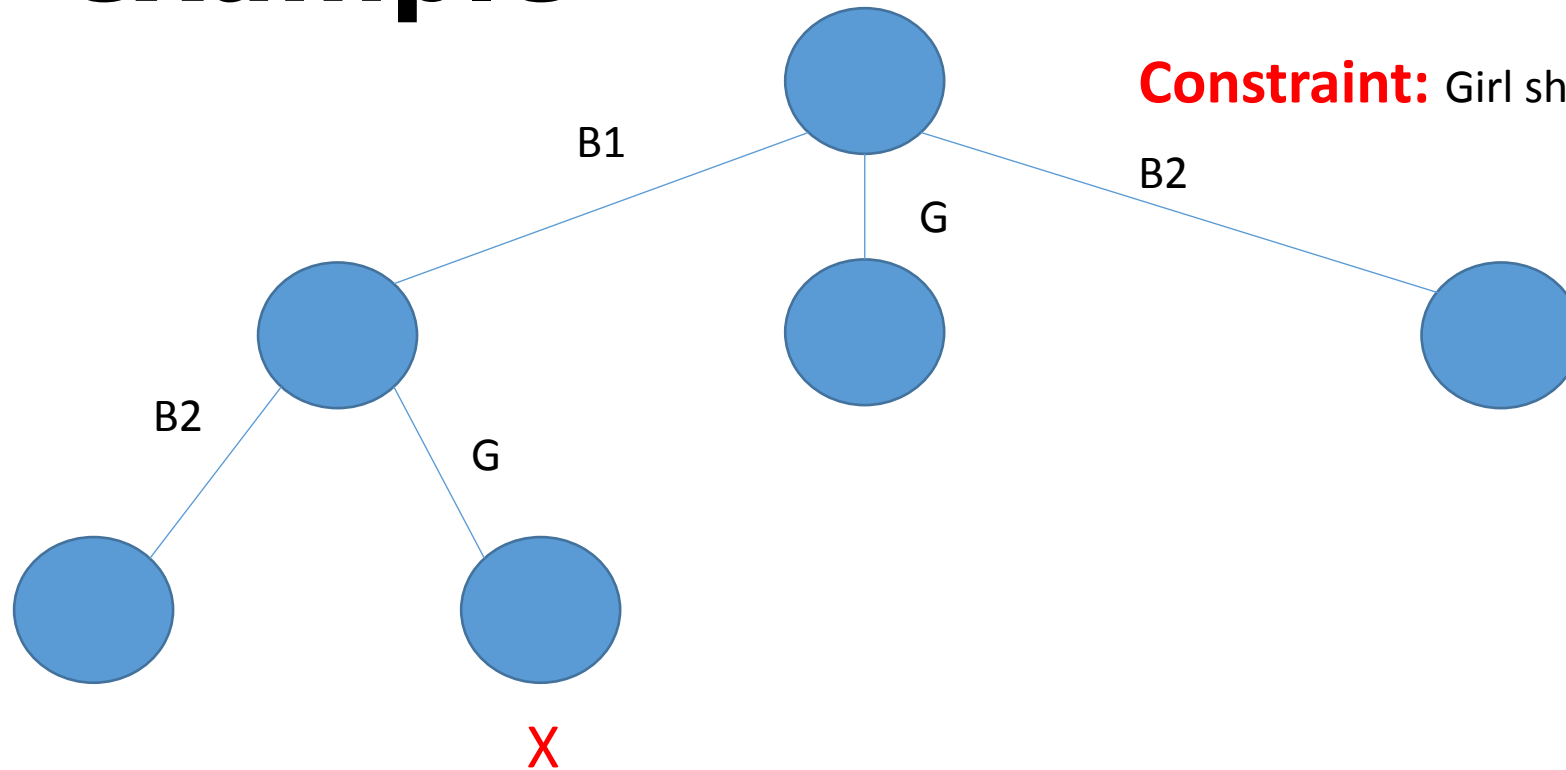


**Constraint:** Girl should not be on the middle bench.

Uses BFS approach to explore the node



# FIFO-BB Approach to solve the example

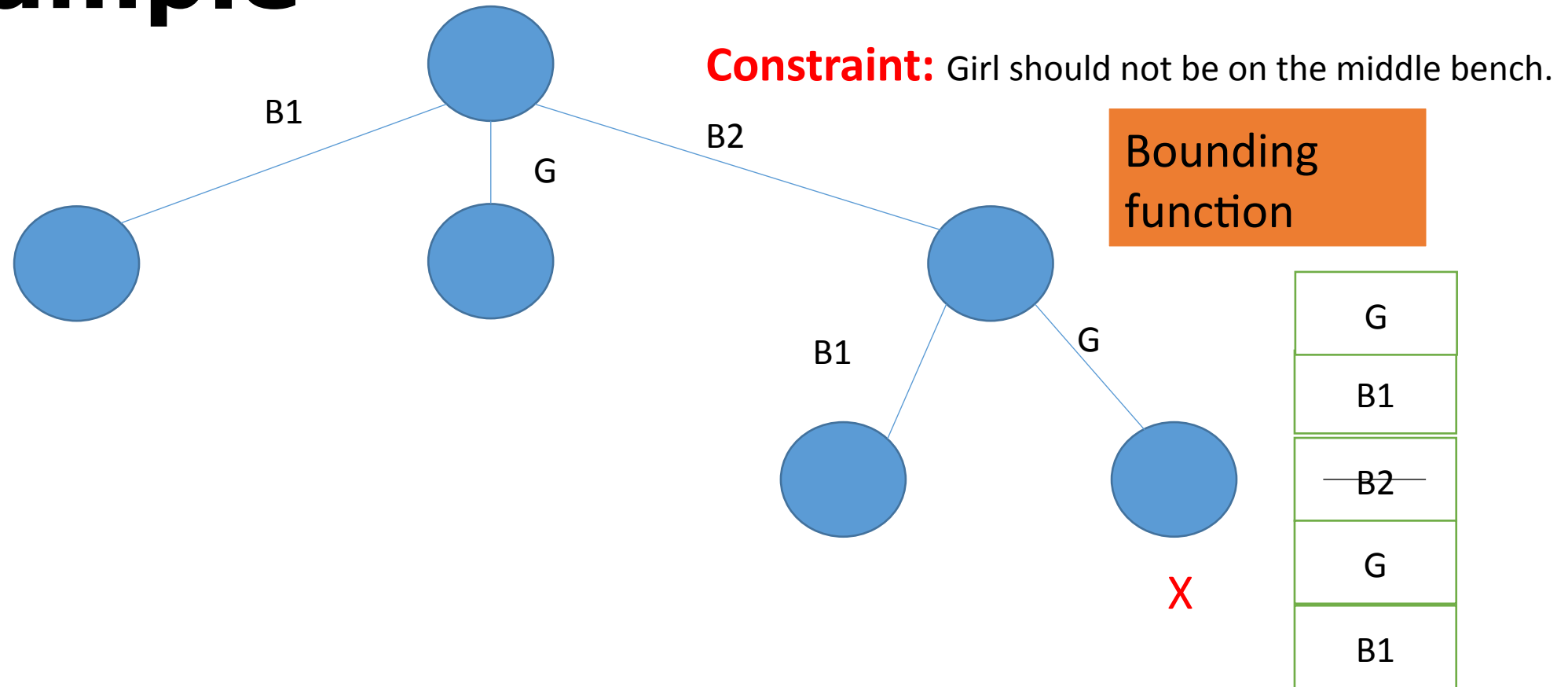


**Constraint:** Girl should not be on the middle bench.

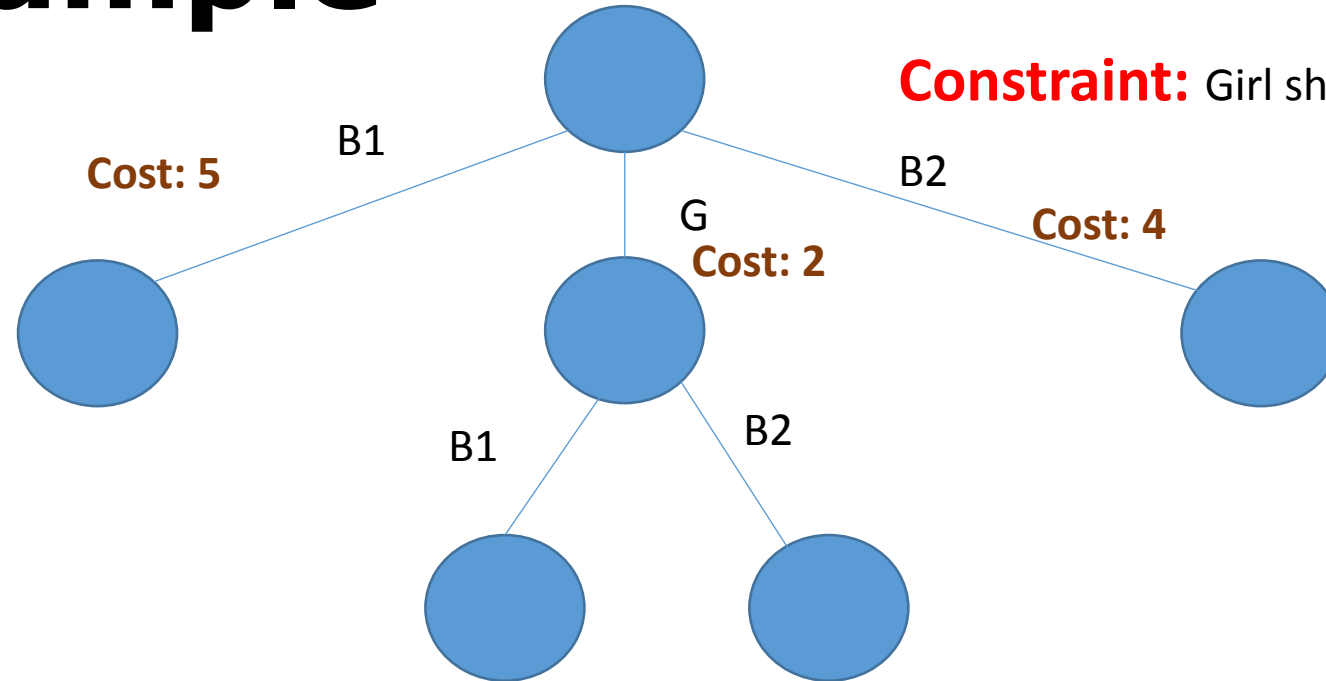
Bounding  
function

<del>B1</del>	G	B2	B2	G
---------------	---	----	----	---

# LIFO BB Approach to solve the example



# LC-BB Approach to solve the example



**Constraint:** Girl should not be on the middle bench.

Bounding  
function

# Backtracking principle

Backtracking is a general algorithm for solving some computational problems, most notably constraint satisfaction problems, that incrementally builds candidates to the solutions and abandons a candidate as soon as it determines that the candidate cannot be completed to a reasonable solution.

# Backtracking principle

- Backtracking find all answer nodes, not just one case.
- Let  $(x_1 \dots x_i)$  be a path from root to a node in the state space tree.
- $T(x_1 \dots x_i)$  be the set of all possible values for  $x_{i+1}$  such that  $(x_1, \dots, x_i, x_{i+1})$  is also a path to problem state.
- Let  $B_{i+1}$  be a bounding function such that if  $B_{i+1}(x_1, x_i, x_{i+1})$  is false for the path  $(x_1, \dots, x_i, x_{i+1})$  from the root to the problem state, then path cannot be extended to answer node.
- Then candidates for position  $i+1$  are those generated by the  $T$  satisfying  $B_{i+1}$ .

# Control Abstraction for backtracking

- Control abstraction for backtracking defines the flow of how it solves the given problem in an abstract way.
- In backtracking, solution is defined as n-tuple  $X = (x_1, x_2, \dots, x_n)$ , where  $x_i = 0$  or  $1$ .  $x_i$  is chosen from set of finite components  $S_i$ .  
If component  $x_i$  is selected then set  $x_i = 1$  else set it to  $0$ .
- Backtracking approach tries to find the vector  $X$  such that it maximize or minimize certain criterion function  $P(x_1, x_2, \dots, x_n)$ .

# Control Abstraction for Recursive Backtracking

```
REC_BACKTRACK(k)
// X[1...k - 1] is the solution vector
// T(x[1], x[2], ..., x[K - 1]) is the state
space tree
// Bk( ) is the bounding function
for
each x[k] ∈ T[x[1], x[2], ..., x[k - 1]]
```

```
do
    if
        (Bk(x[1], x[2], ..., x[k])) == TRUE
    then
        // (Feasible solution)
        if
            (x[1], x[2], ..., x[k]) is path to answer node
        then
            print (x[1], x[2], ..., x[k])
        end
        if
            k < n
        then
            REC_BACKTRACK(k + 1)
        end
    end
end
end
```

	1	2	3	4
1				
2				
3				
4				

4x4 chessboard

# Time analysis for backtracking

The performance of backtracking algorithm depends on four parameters:

1. Time to compute the tuple  $x[k]$  i.e. next  $x[k]$
2. Number of  $x[k]$  which satisfy the explicit constraint
3. Time taken by bounding function  $B_k$  to generate a feasible sequence
4. A number of  $x[k]$  which satisfies the bounding function  $B_k$  for all  $k$ .



# Popular problems solved using backtracking

- N-Queen problem
- Sum of subset problem
- Graph coloring problem
- Knapsack problem
- Hamiltonian cycle problem

# N-Queen Problem

N - Queens problem is to place  $n$  - queens in such a manner on an  $n \times n$  chessboard that no queens attack each other by being in the same row, column or diagonal.

**Note:** It can be seen that for  $n = 1$ , the problem has a trivial solution, and no solution exists for  $n = 2$  and  $n = 3$ . So first we will consider the **4 queens problem** and then generate it to  $n$  - queens problem.

# 4 Queen Problem

Given a 4 x 4 chessboard and number the rows and column of the chessboard 1 through 4

	1	2	3	4
1				
2				
3				
4				

4x4 chessboard

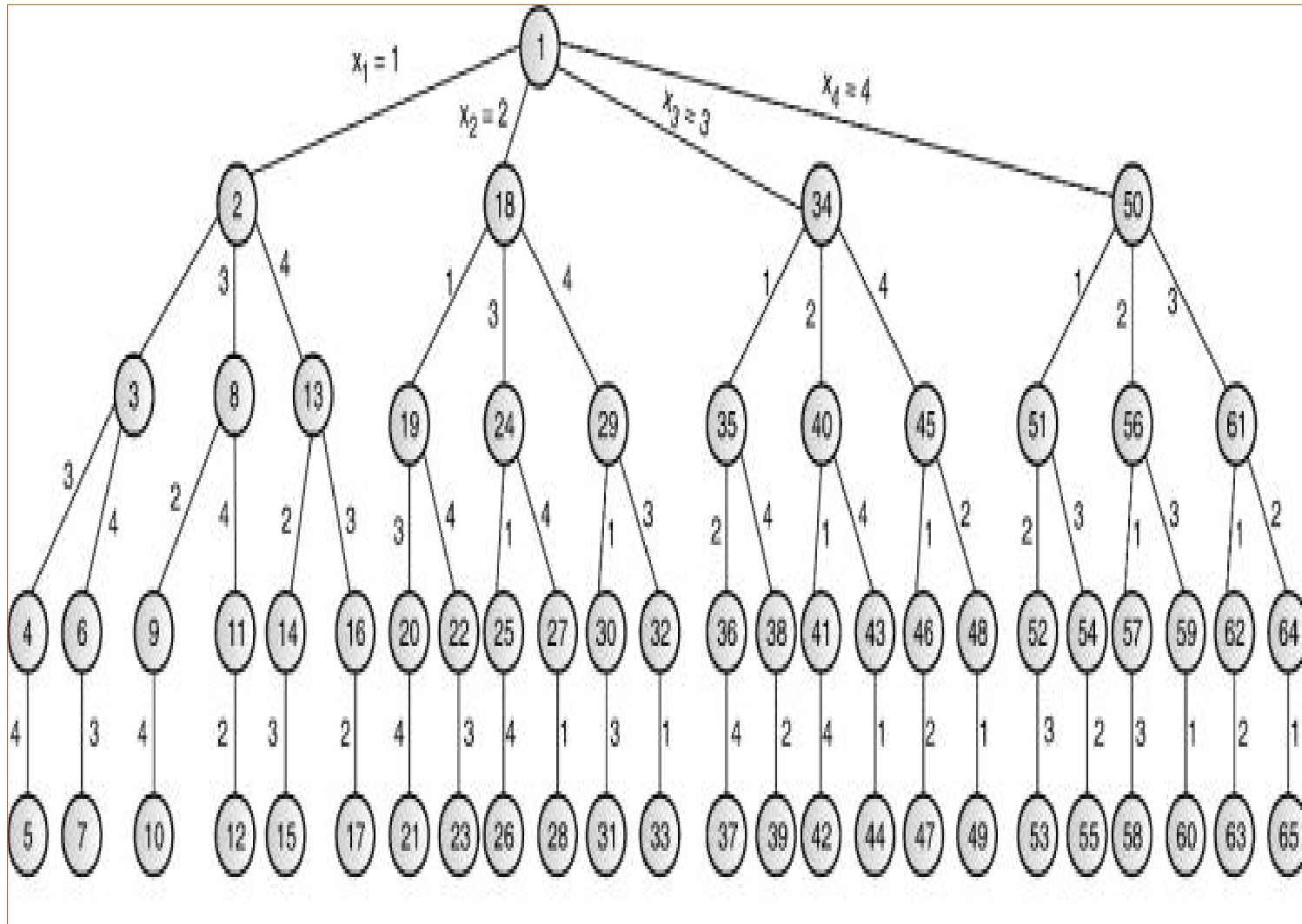
## Explicit Constraints:

Explicit constraints are the rules that allow/disallow selection of  $x_i$  to take value from the given set.

Ex: For n-queen problem explicit constraint is  $S_i = \{1, 2, 3, \dots, n\}$   
 $1 \leq i \leq n$

## Implicit constraints:

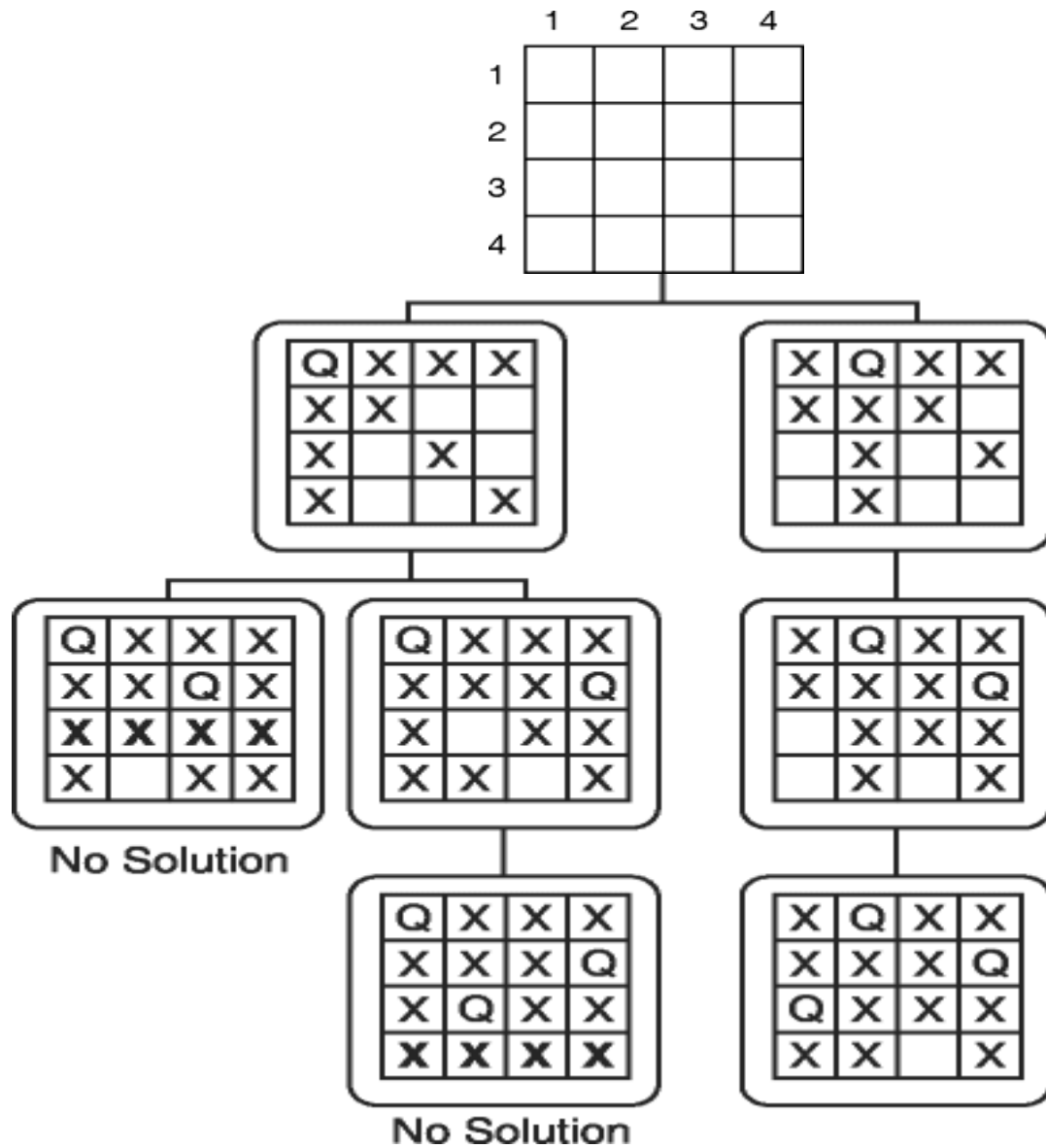
The implicit constraint is to determine which of the tuple of solution space satisfies the given criterion functions. The implicit constraint for n queen problem is that two queens must not appear in the same row, column or diagonal.



## State Space Tree for 4-Queen Problem

	1	2	3	4
1				
2				
3				
4				

4x4 chessboard

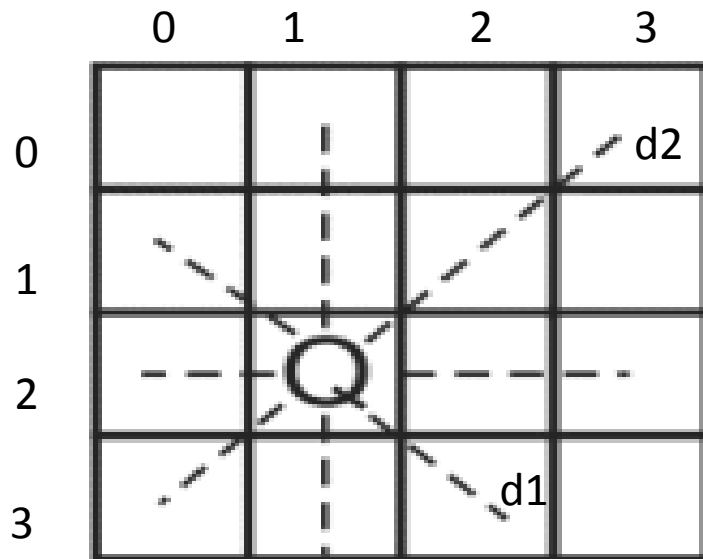


## Backtracking procedure of 4-Queen problem

**solution (2, 4, 1, 3)**

4 queens problem can be represented as **4 - tuples (x1, x2, x3, x4)** where **xi** represents the **column** on which queen "**qi**" is placed.

# Attacked cells by queen Q



Cells on diagonal 1(d1): (1,0),(3,2)

Cells on diagonal 2 (d2): (3,0),(1,2),(0,3)

Two elements (i,j) and (k,l) on diagonal **d1** have following relation

$$i-j=k-l$$

$$\text{i.e. } j-l = i-k \quad \text{i.e. } r1-r2=c1-c2....$$

1

Two elements (i,j) and (k,l) on diagonal **d2** have following relation

$$i+j=k+l$$

$$\text{i.e. } j-l = i-k \quad \text{i.e. } r1-r2=c1-c2....$$

2

From equation 1 and 2 we can write

$$|j-l| = |i-k| \quad \text{or} \quad |c1-c2| = |r1-r2|$$

# One possible solution for 8 queens problem

	1	2	3	4	5	6	7	8
1				q <sub>1</sub>				
2						q <sub>2</sub>		
3								q <sub>3</sub>
4		q <sub>4</sub>						
5							q <sub>5</sub>	
6	q <sub>6</sub>							
7			q <sub>7</sub>					
8					q <sub>8</sub>			

Solution tuple for the solution shown in fig is defined as **<4, 6, 8, 2, 7, 1, 3, 5>**

- 8 queen problem has  ${}^6P_8 = 4,42,61,65,368$  different arrangements.
- Of these, only 92 arrangements are valid solutions.
- Out of which, only 12 are the fundamental solutions.
- The remaining 80 solutions can be generated using reflection and rotation

# Algorithm for n-Queen

```
N - Queens (k, n)
{
  For i ← 1 to n
    do if Place (k, i) then
    {
      x [k] ← i;
      if (k ==n) then
        write (x [1....n]);
      else
        N - Queens (k + 1, n);
    }
}
```

```
Place (k, i)
{
  For j ← 1 to k - 1
    do if (x [j] = i)
      or (Abs x [j]) - i = (Abs (j - k))
    then return false;
  return true;
}
```



# Complexity Analysis

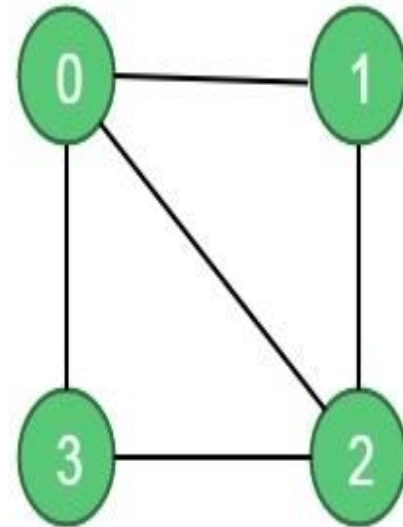
- In backtracking, at each level branching factor decreases by 1 and it creates a new problem of size  $(n - 1)$  .
- With  $n$  choices, it creates  $n$  different problems of size  $(n - 1)$  at level 1.
- PLACE function determines the position of the queen in  $O(n)$  time. This function is called  $n$  times.
- Thus, the recurrence of  $n$ -Queen problem is defined as,  $T(n)$   
 $= n * T(n - 1) + n^2$ . Solution to recurrence would be  $O(n!)$ .

# Graph Coloring problem

Input

- 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.

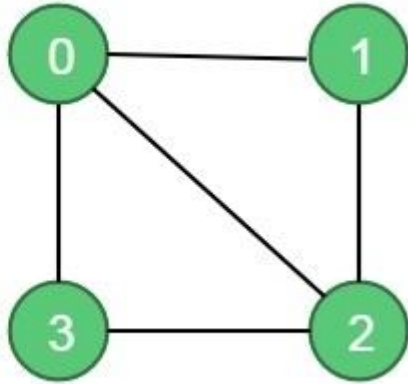
0	1	1	1
1	0	1	0
1	1	0	1
1	0	1	0



# Input and Output

Input

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



Let the maximum color  $m = 3$ .

Output

This algorithm will return which node will be assigned with which color. **If the solution is not possible, it will return false.**

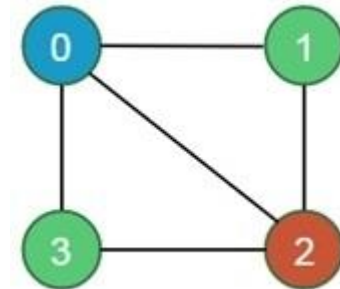
For this input the assigned colors are:

Node 0 -> color 1

Node 1 -> color 2

Node 2 -> color 3

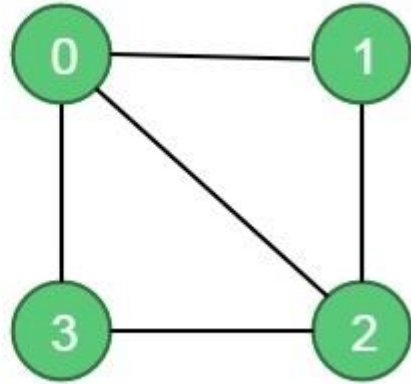
Node 3 -> color 2



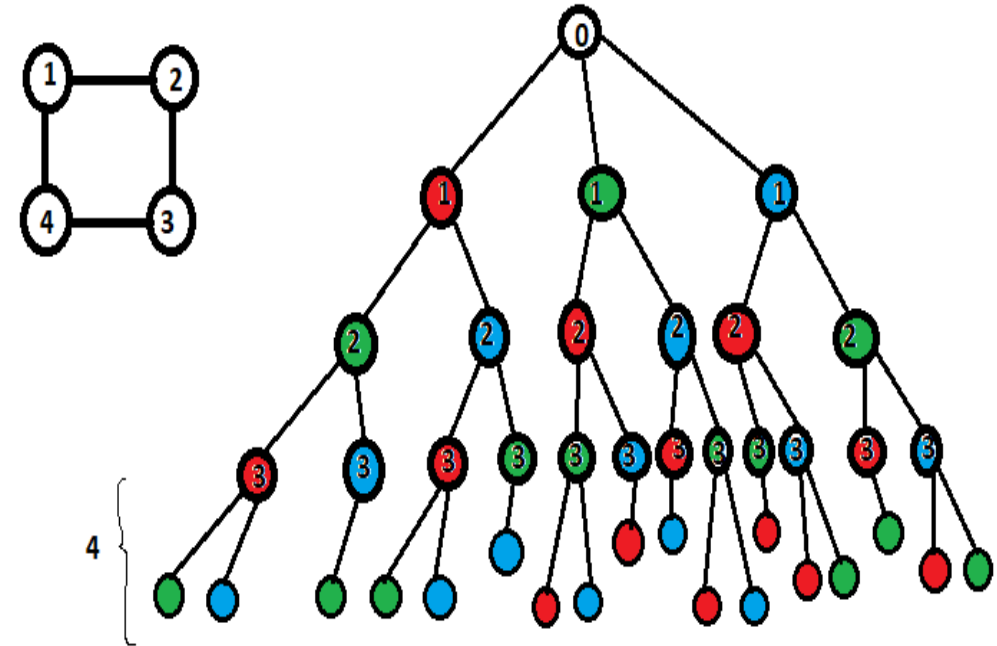
# Input and Output

## Input

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



Let the maximum color  $m = 3$ .



# Algorithm 1

isValid(vertex, colorList, col)

**Input** – Vertex, colorList to check, and color, which is trying to assign.

**Output** – True if the color assigning is valid, otherwise false.

**Begin**

**for** all vertices v of the graph, do

**if** there is an edge between v and i, and col = colorList[i], then

**return** false

**done**

**return** true

**End**

# Algorithm 2

## graphColoring(colors, colorList, vertex)

**Input** – Most possible colors, the list for which vertices are colored with which color, and the starting vertex.

**Output** – True, when colors are assigned, otherwise false.

**Begin**

**if** all vertices are checked, then

**return** true

**for** all colors col from available colors, do

**if** isValid(vertex, color, col), then

            add col to the colorList for vertex

**if** graphColoring(colors, colorList, vertex+1) = true, then

**return** true

        remove color for vertex

**done**

**return** false

**End**

# Time Complexity

- Since backtracking is also a kind of brute force approach, there would be total  $O(m^V)$  possible color combinations.
- It is to be noted that the upperbound time complexity remains the same but the average time taken will be less due to the refined approach.

# Subset Sum Problem

In this problem, there is a given set with some integer elements. And another sum value is also provided, we have to find a subset of the given set whose sum is the same as the given sum value.

Here **backtracking** approach is used for trying to select a valid subset when an item is not valid, we will backtrack to get the previous subset and add another element to get the solution.



# Input and Output

## Input:

The Set: {10, 7, 5, 18, 12, 20, 15}

The sum Value: 35

## Output:

All **possible subsets** of the given set, where sum of each element for every subsets is same as the given sum value.

{10, 7, 18}

{10, 5, 20}

{5, 18, 12}

{20, 15}

# **Write an algorithm for sum of subsets. Solve the following problem. $M=30$ $W=\{5, 10, 12, 13, 15, 18\}$**

## **Problem statement:**

- Let,  $S = \{S_1 \dots S_n\}$  be a set of  $n$  positive integers, then we have to find a subset whose sum is equal to given positive integer  $d$ . It is always convenient to sort the set's elements in ascending order. That is,  $S_1 \leq S_2 \leq \dots \leq S_n$

## **Algorithm:**

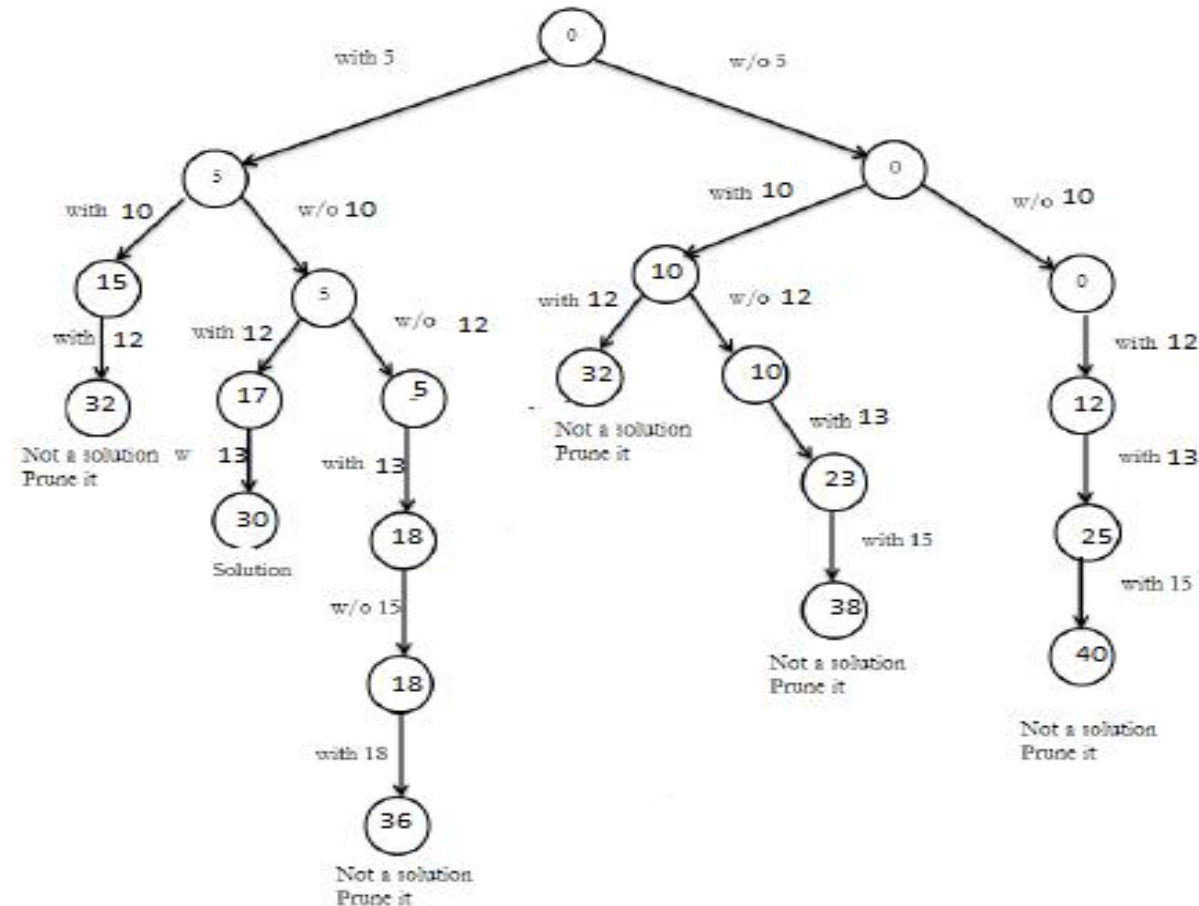
- Let,  $S$  is a set of elements and  $m$  is the expected sum of subsets. Then:
- Start with an empty set.
- Add to the subset, the next element from the list.
- If the subset is having sum  $m$  then stop with that subset as solution.
- If the subset is not feasible or if we have reached the end of the set then backtrack through the subset until we find the most suitable value.
- If the subset is feasible then repeat step 2.
- If we have visited all the elements without finding a suitable subset and if no backtracking is possible then stop without solution.

**Write an algorithm for sum of subsets.  
Solve the following problem.  $M=30$   
 $W=\{5, 10, 12, 13, 15, 18\}$**

Initially subset = {}	Sum = 0	Description
5	5	Then add next element.
5, 10	15 i.e. $15 < 30$	Add next element.
5, 10, 12	27 i.e. $27 < 30$	Add next element.
5, 10, 12, 13	40 i.e. $40 < 30$	Sum exceeds $M = 30$ . Hence backtrack.
5, 10, 12, 15	42	Sum exceeds $M = 30$ . Hence backtrack.
5, 10, 12, 18	45	Sum exceeds $M = 30$ . Hence backtrack.
5, 12, 13	30	Solution obtained as $M = 30$

# Write an algorithm for sum of subsets. Solve the following problem. $M=30$ $W=\{5, 10, 12, 13, 15, 18\}$

- The state space tree is shown as below in figure.  $\{5, 10, 12, 13, 15, 18\}$



# Algorithm

subsetSum(set, subset, n, subSize, total, node, sum)

**Input** – The given set and subset, size of set and subset, a total of the subset, number of elements in the subset and the given sum.

**Output** – All possible subsets whose sum is the same as the given sum.

**Begin**

```
    if total = sum, then
        display the subset
        //go for finding next subset
        subsetSum(set, subset, , subSize-1, total-
set[node], node+1, sum)
    return
else
    for all element i in the set, do
        subset[subSize] := set[i]
        subSetSum(set, subset, n, subSize+1,
total+set[i], i+1, sum)
    done
End
```

# Time Complexity

It is intuitive to derive the complexity of sum of the subset problem. In the state-space tree, at level  $i$ , the tree has  $2^i$  nodes. So, given  $n$  items, the total number of nodes in the tree would be  $1 + 2 + 2^2 + 2^3 + \dots 2^n$ .

$$T(n) = 1 + 2 + 2^2 + 2^3 + \dots 2^n = 2^{n+1} - 1 = O(2^n)$$

# Branch and Bound

- Branch and bound is one of the techniques used for problem solving.
- It is similar to the backtracking since it also **uses the state space tree**.
- It is used for solving the **optimization problems and minimization problems**.
- If we have given a maximization problem then we can convert it using the Branch and bound technique by simply converting the problem into a minimization problem.
- Branch and Bound refers to all state space search methods in which all children of the E-Node are generated before any other live node becomes the E-Node.

# Branch and Bound

- **Types of Branch and Bound Solutions:**

The solution of the Branch and the bound problem can be represented in two ways:

1. **Variable size solution:** Using this solution, we can find the subset of the given set that gives the optimized solution to the given problem. For example, if we have to select a combination of elements from {A, B, C, D} that optimizes the given problem, and it is found that A and B together give the best solution, then the solution will be {A, B}.
2. **Fixed-size solution:** There are 0s and 1s in this solution, with the digit at the  $i$ th position indicating whether the  $i$ th element should be included, for the above example, the solution will be given by {1, 1, 0, 0}, here 1 represent that we have select the element which at  $i$ th position and 0 represent we don't select the element at  $i$ th position.

- **Classification of Branch and Bound Problems:**

The Branch and Bound method can be classified into three types based on the order in which the state space tree is searched.

1. FIFO Branch and Bound
2. LIFO Branch and Bound
3. Least Cost-Branch and Bound



# Principle of Branch and Bound

It is an algorithm design paradigm which is generally used for solving combinatorial optimization problems using

## 2 mechanisms:

- A mechanism to generate branches when searching the solution space
- A mechanism to generate a bound so that many branches can be terminated

# Control Abstraction for LC Branch and Bound

```
// search tree t for answer node
{
  if *t is an answer node then output *t and return;
  E=t // E-node
  repeat
  {
    for each child x of E do
    {
      if x is an answer node then output the path
      from x to t and return;
      Add( x ); // x is a new live node
      ( x ->parent )=E // pointer for path to root
      //( x ->parent )=E is to print answer
      path.
    }
  }
```

```
if there are no more live nodes then
{
  write(“ No answer node “);
  return;
}
E=Least();
} until ( false )
}
```

Min-heap data structure is used to implement a list of live nodes so that the node with the lowest cost is always at the front.

So that *Least( )* function can retrieve the minimum cost node in constant time

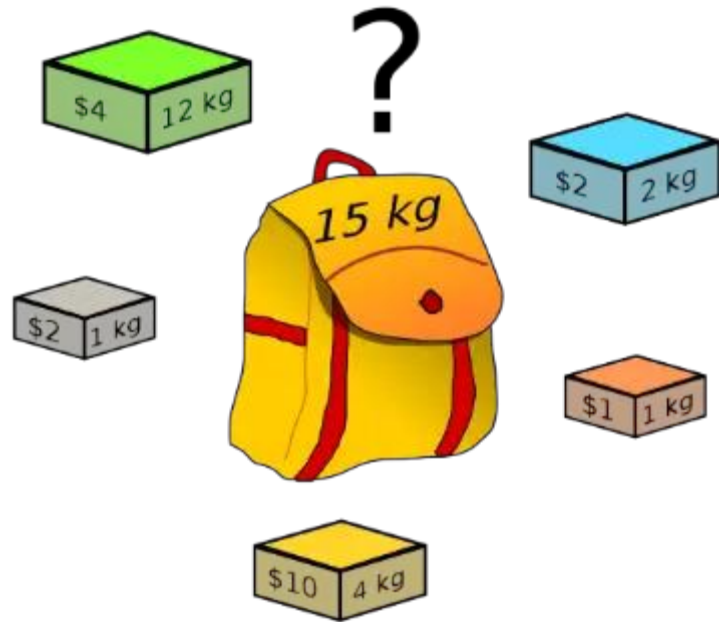
# Time Analysis of Branch and Bound

- Branch and bound is an algorithm design paradigm which is generally used for solving combinatorial optimization problems.
- These problems are typically exponential in terms of time complexity and may require exploring all possible permutations in worst case.

# Popular problems solved using Branch and Bound

- Knapsack problem
- Travelling salesman problem
- Assignment problem
- Job scheduling problem

# 0/1 Knapsack problem using Branch and Bound



**Knapsack Problem**

## **Knapsack Problem-**

You are given the following-

- A knapsack (kind of shoulder bag) with limited weight capacity.
- Few items each having some weight and value.

## **The problem states-**

Which items should be placed into the knapsack such that-

- The value or profit obtained by putting the items into the knapsack is maximum.
- And the weight limit of the knapsack does not exceed.

# 0/1 Knapsack problem using Branch and Bound

## Approach:

The implementation of **Branch and Bound method using Least cost(LC)** for 0/1 Knapsack Problem is as follows:

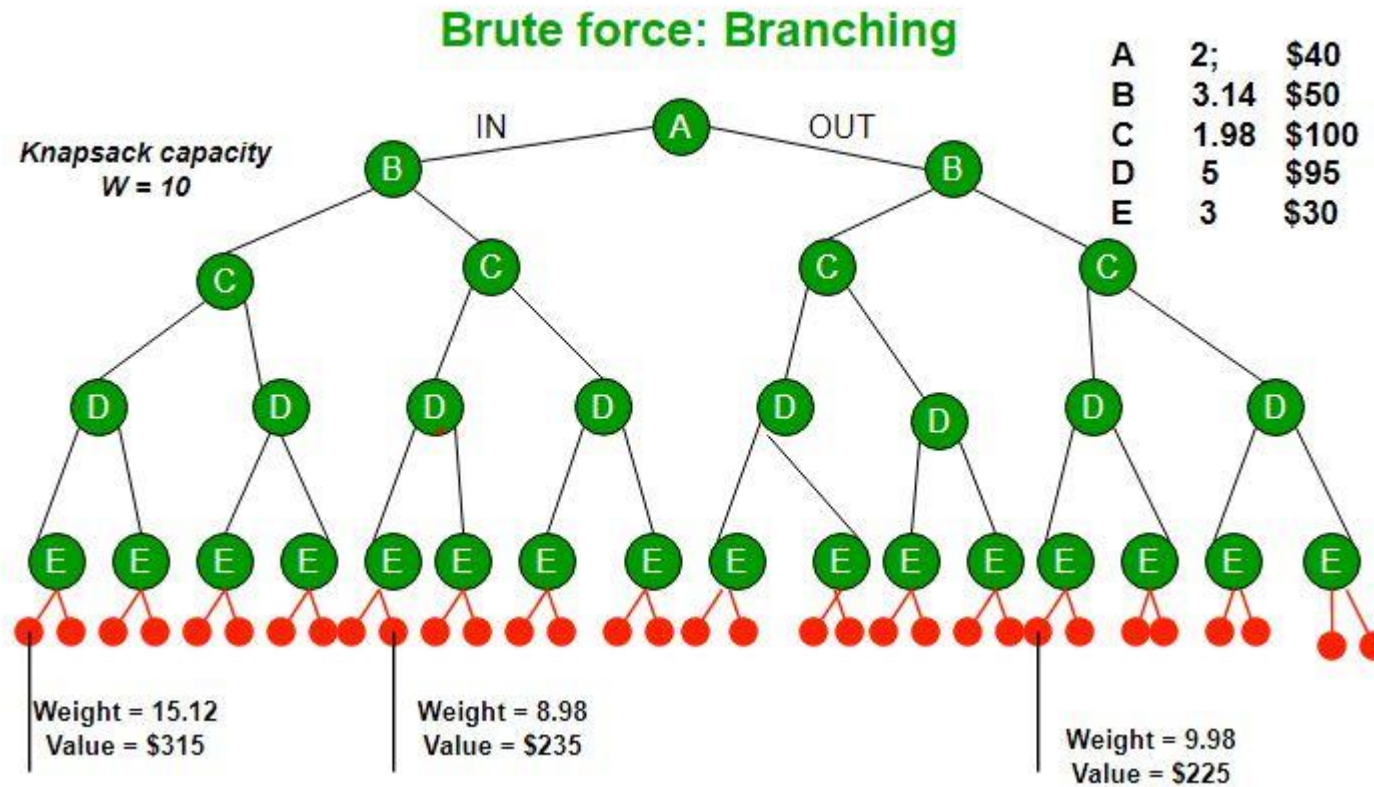
- Branch and Bound can be solved using **FIFO**, **LIFO** and **LC** strategies.
- The **least cost(LC)** is considered the most intelligent as it selects the next node based on a **Heuristic Cost Function**. It picks the one with the least cost.
- As **0/1 Knapsack** is about maximizing the total value, we cannot directly use the **LC Branch and Bound** technique to solve this.
- Instead, we convert this into a minimization problem by taking negative of the given values.

# 0/1 Knapsack problem using Branch and Bound

Follow the steps below to solve the problem:

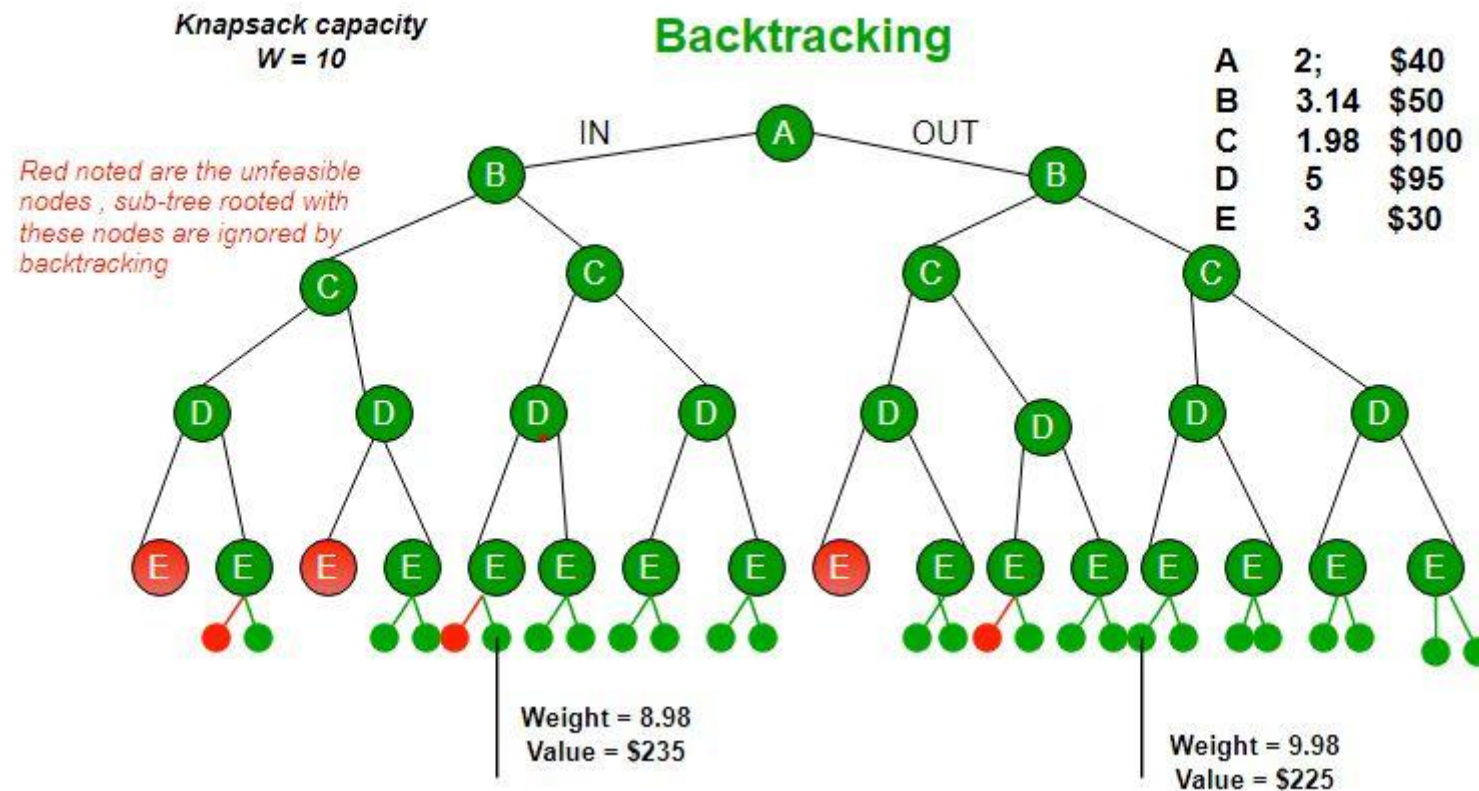
1. Sort the items based on their **value/weight(V/W)** ratio.
2. Insert a dummy node into the priority queue.
3. Repeat the following steps until the priority queue is empty:
  - Extract the peek element from the priority queue and assign it to the **current node**.
  - If the upper bound of the current node is less than **minLB**, the minimum lower bound of all the nodes explored, then there is no point of exploration. So, continue with the next element. The reason for not considering the nodes whose upper bound is greater than **minLB** is that, the upper bound stores the best value that might be achieved. If the best value itself is not optimal than **minLB**, then exploring that path is of no use.
  - Update the **path array**.
  - If the current node's level is **N**, then check whether the lower bound of the current node is less than **finalLB**, minimum lower bound of all the paths that reached the final level. If it is true, update the **finalPath** and **finalLB**. Otherwise, continue with the next element.
  - Calculate the lower and upper bounds of the right child of the current node.
  - If the current item can be inserted into the knapsack, then calculate the lower and upper bound of the left child of the current node.
  - Update the **minLB** and insert the children if their upper bound is less than **minLB**.

# 0/1 Knapsack problem using Branch and Bound



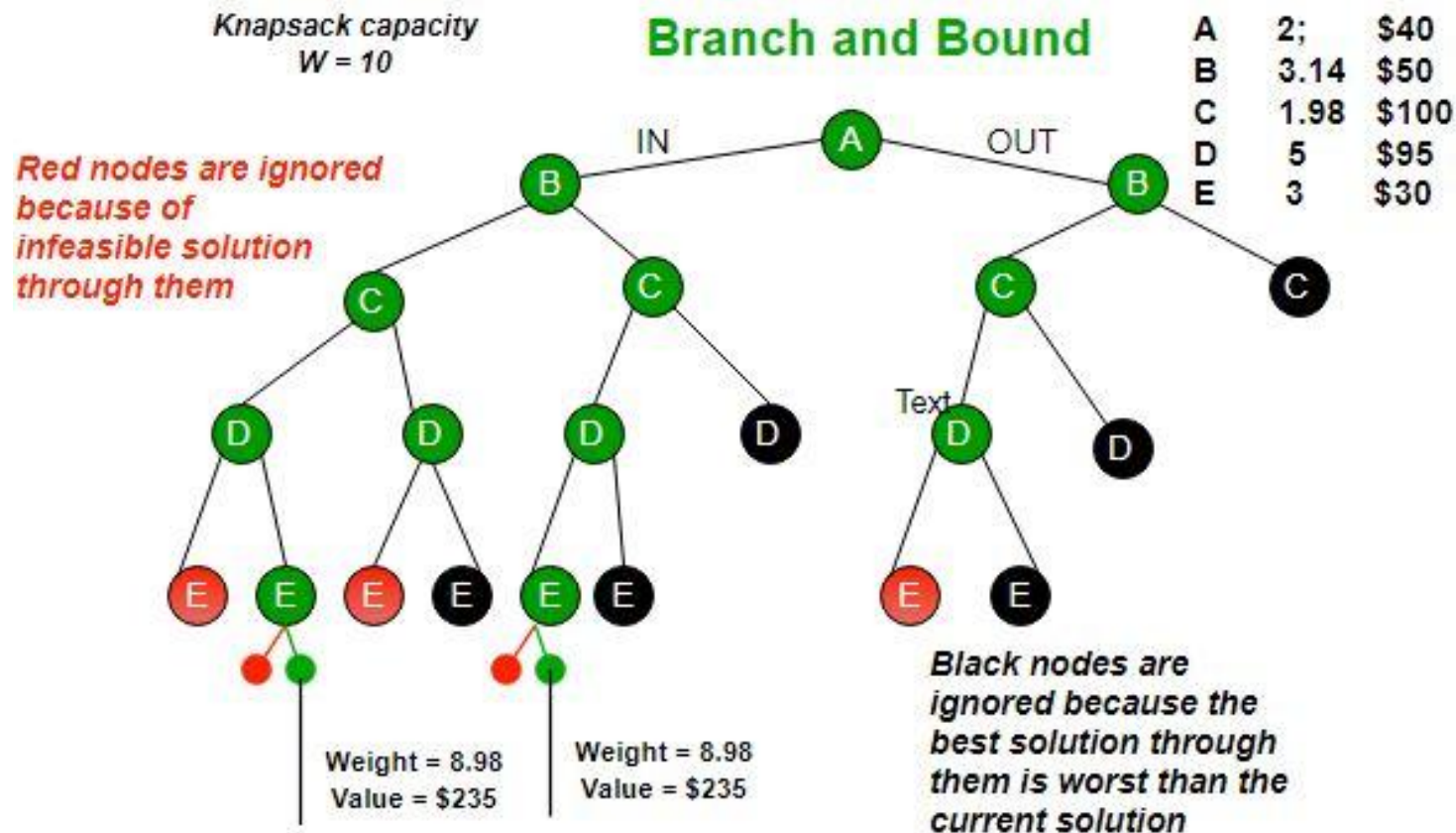


# 0/1 Knapsack problem using Branch and Bound



We can use **Backtracking** to optimize the Brute Force solution. In the tree representation, we can do DFS of tree. If we reach a point where a solution no longer is feasible, there is no need to continue exploring. In the given example, backtracking would be much more effective if we had even more items or a smaller knapsack capacity.

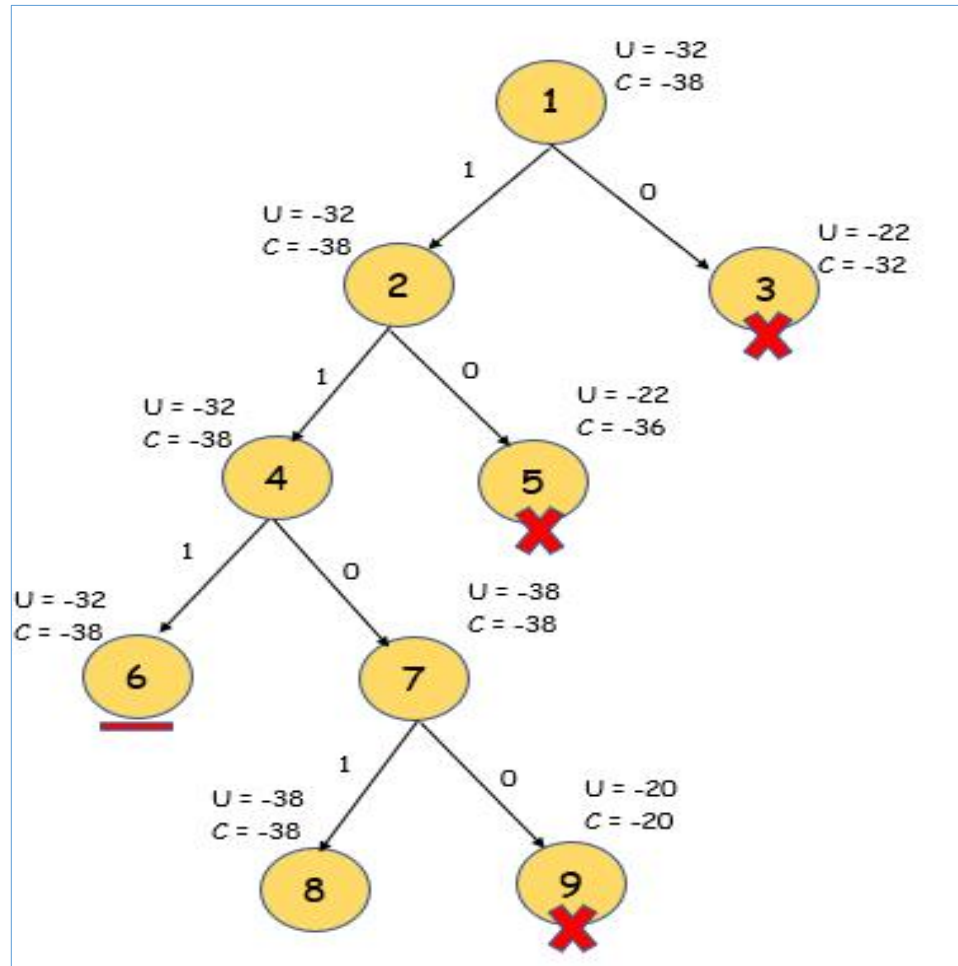
# 0/1 Knapsack problem using Branch and Bound



## Branch and Bound

The backtracking based solution works better than brute force by ignoring infeasible solutions. 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.

# 0/1 Knapsack problem using Branch and Bound(LC approach)



Consider the problem with  $n = 4$ ,  $V = \{10, 10, 12, 18\}$ ,  $w = \{2, 4, 6, 9\}$  and  $W = 15$

Here, we calculate the initial upper bound to be

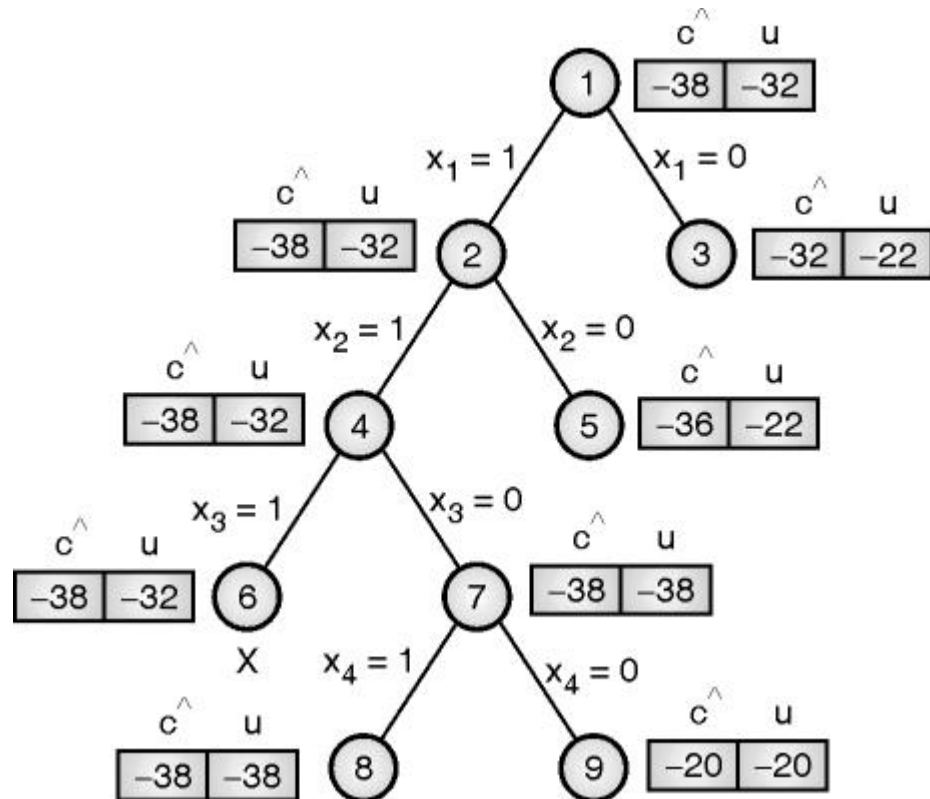
$$U = 10 + 10 + 12 = 32.$$

Note that the 4th object cannot be included here, since that would exceed  $W$ . For the cost, we add  $3/9^{\text{th}}$  of the final value, and hence the cost function is 38.

**Remember to negate the values after calculation before comparison.**

# 0/1 Knapsack problem using Branch and Bound(LC approach) (Contd..)

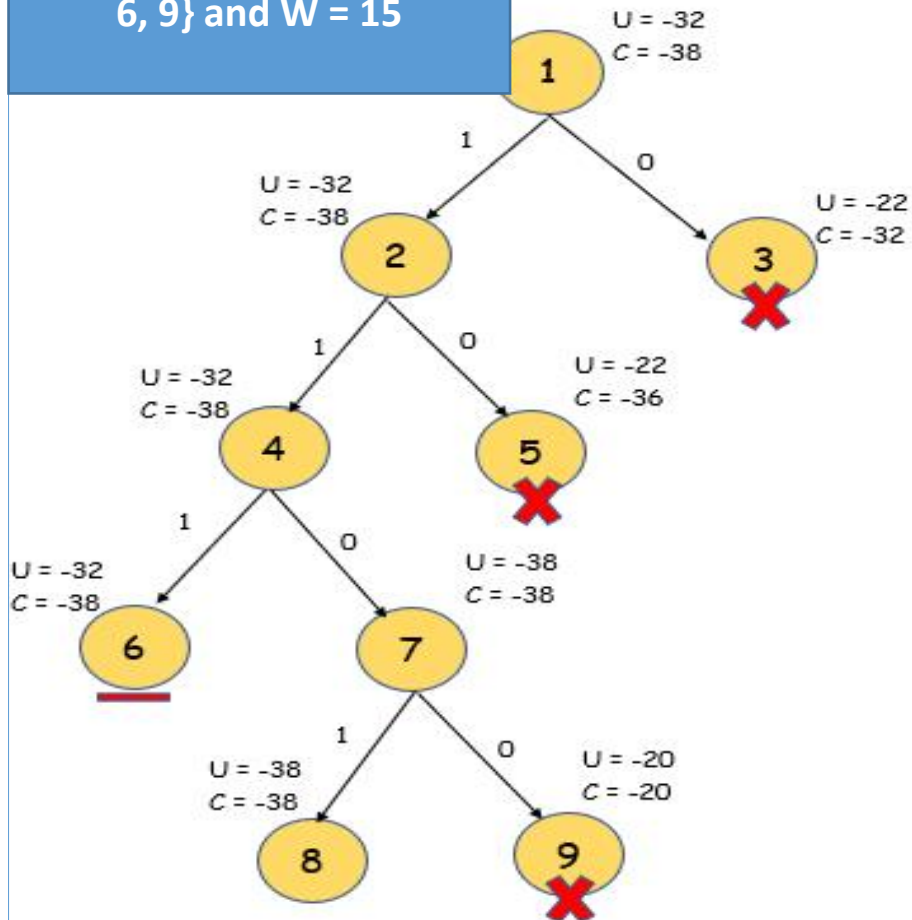
problem with  $n = 4$ ,  $V = \{10, 10, 12, 18\}$ ,  $w = \{2, 4, 6, 9\}$  and  $W = 15$



- Note here that node 3 and node 5 have been killed after updating U at node 7.
- Also, node 6 is not explored further, since adding any more weight exceeds the threshold.
- At the end, only nodes 6 and 8 remain.
- Since the value of U is less for node 8, we select this node. Hence the solution is  $\{1, 1, 0, 1\}$ , and we can see here that the total weight is exactly equal to the threshold value in this case.

# 0/1 Knapsack problem using Branch and Bound(LC approach) (Contd..)

problem with  $n=4$ ,  $V = \{10, 10, 12, 18\}$ ,  $w = \{2, 4, 6, 9\}$  and  $W = 15$



- Note here that node 3 and node 5 have been killed after updating U at node 7.
- Also, node 6 is not explored further, since adding any more weight exceeds the threshold.
- At the end, only nodes 6 and 8 remain.
- Since the value of U is less for node 8, we select this node. Hence the solution is  $\{1, 1, 0, 1\}$ , and we can see here that the total weight is exactly equal to the threshold value in this case.

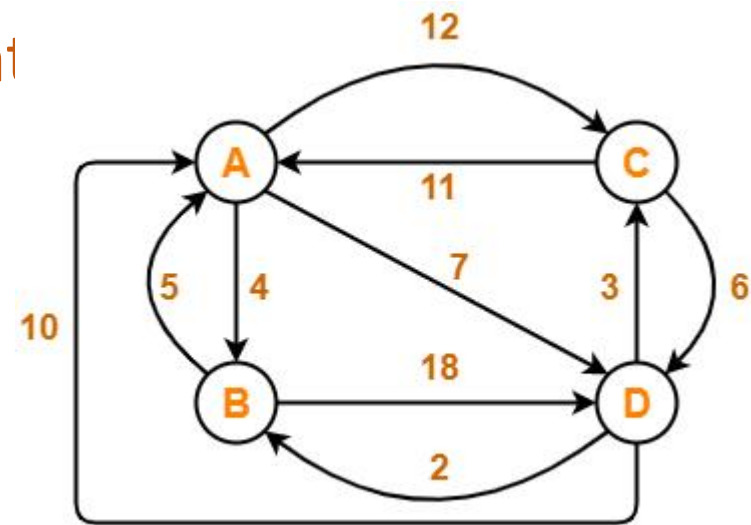
# Time Complexity

- Even though this method is more efficient than the other solutions to this problem, its worst case time complexity is still given by  $O(2^n)$ , in cases where the entire tree has to be explored.
- However, in its best case, only one path through the tree will have to be explored, and hence its best case time complexity is given by  $O(n)$

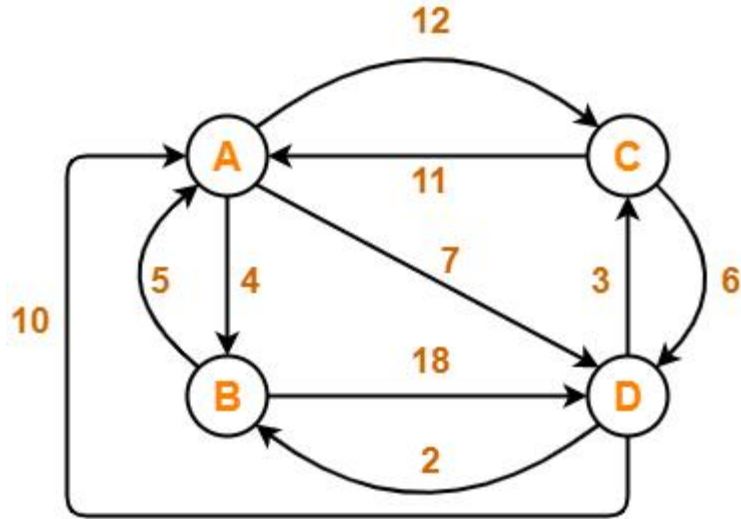


# Travelling Salesman Problem using Branch and Bound

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



# TSP (Contd..)



	A	B	C	D
A	$\infty$	4	12	7
B	5	$\infty$	$\infty$	18
C	11	$\infty$	$\infty$	6
D	10	2	3	$\infty$

## Rules

- To reduce a matrix, perform the row reduction and column reduction of the matrix separately.
- A row or a column is said to be reduced if it contains at least one entry '0' in it.



# Time Complexity

- The worst case complexity of Branch and Bound remains same as that of the Brute Force clearly because in worst case, we may never get a chance to prune a node.
- Whereas, in practice it performs very well depending on the different instance of the TSP.
- The complexity also depends on the choice of the bounding function as they are the ones deciding how many nodes to be pruned.

# Case Study

## Airline Crew Scheduling

### Imp Links:

1. <https://www.engineeringenotes.com/project-management-2/operations-research/assignment-problem-meaning-methods-and-variations-operations-research/15652>
2. <https://www.geeksforgeeks.org/job-assignment-problem-using-branch-and-bound/>

An airline, that operates seven days a week, has a time table s, crews must have a minimum layover of 6 hours between flights. Obtain the pairing of flights that minimizes layover time away from home. For any given pairing the crew will 1 be based at the city that results in the smaller layover

Flight	Mumbai (Depart)	Chennai (Arrive)	Flight	Chennai (Depart)	Mumbai (Arrive)
1	7:00 AM	9:00 AM	101	9:00AM	11:00 AM
2	9:00 AM	11:00 AM	102	10:00 AM	12:00 PM
3	1:30 PM	3:30 PM	103	3:30 PM	5:30 PM
4	7:30 PM	9:30 PM	104	8:00 PM	10:00 PM

# Thank You!!