Backtracking, Branch & Bound

Module 2

Sem IV

AoA Even 2021-22

Introduction

- Backtracking and Branch and Bound are two graph based methods for design of algorithms.
- In both cases we explore a search tree.
- In Backtracking, we start with a node and explore the nodes in Depth First manner.
- All the nodes need not to be explored. Cut the branches of the tree based on the constraint of the problem. This reduces the time complexity of the algorithm.
- Branch and Bound explores the search tree in a Breadth First manner.

Backtracking : Introduction

- It is modified form of Depth First Search.

- Here solution vector is of form $x_1, x_2, x_3, \ldots, x_n$, n tuple $(x_1, x_2, x_3, \ldots, x_n)$, where x_i is chosen from finite set of S_i, such that constraint of the problem is satisfied.

- Backtracking algorithm solves the problem using two types of constraints:
  1. Explicit Constraint
  2. Implicit Constraint

Backtracking : Introduction

Terminologies Used:

Backtracking algorithms determine problem solutions by systematically searching for solution using tree structure.

1. **State space tree:** The solution space is organized as a tree called the state space tree.
2. **Explicit constraint**: These are the rules that restrict each component x_i of the solution vector to take values only from a given set S.
3. **Implicit constraint**: These are the rules that describe the way in which the x_i 's must relate to each other or which of the components of the solution vector satisfy the criteria function.
4. **Solution space**: It is the set of all tuples that satisfy the explicit constraints.
5. **Live node**: It is the node that has been generated, but none of its descendants are yet generated.
6. **Bounding function or criteria**: It is a function created that is used to kill live nodes without generating all its children.

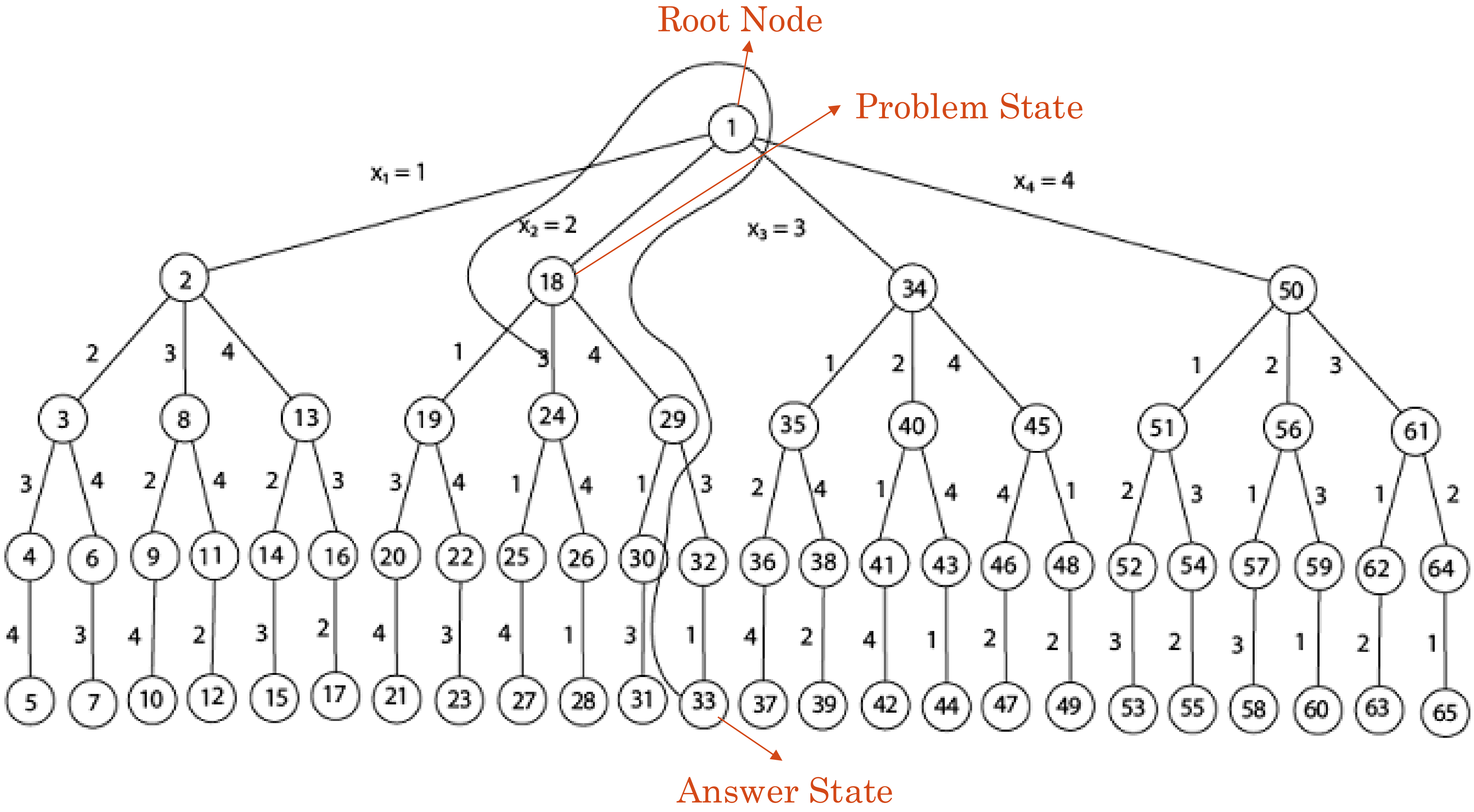
Backtracking : Introduction

Terminologies Used:

Backtracking algorithms determine problem solutions by systematically searching for solution using tree structure.

7. **Extended node or E-node**: It is the live node whose children are currently being generated.
8. **Dead node**: It is the node that is not to be extended further or all of whose children have already been generated.
9. **Answer node**: It is the node that represents the answer of the problem that means the node at which the criteria functions are maximized, minimized or satisfied.
10. **Solution node**: It is the node that has the possibility to become the answer node.

Backtracking : Introduction



Backtracking : N-Queen Problem

1. **2-Queen problem**
2. **3-Queen Problem**
3. **4-Queen Problem**
4. **8-Queen problem**

2-Queen problem:

Q	x
x	x

x	Q
x	x

Therefore, No Solution

Backtracking : N-Queen Problem

1. **2-Queen problem**
2. **3-Queen Problem**
3. **4-Queen Problem**
4. **8-Queen problem**

2-Queen problem:

Q	x
x	x

x	Q
x	x

Therefore, No Solution

Similarly, No Solution for 3-Queen problem

Backtracking : N-Queen Problem

4 Queen Problem:

State space Tree:

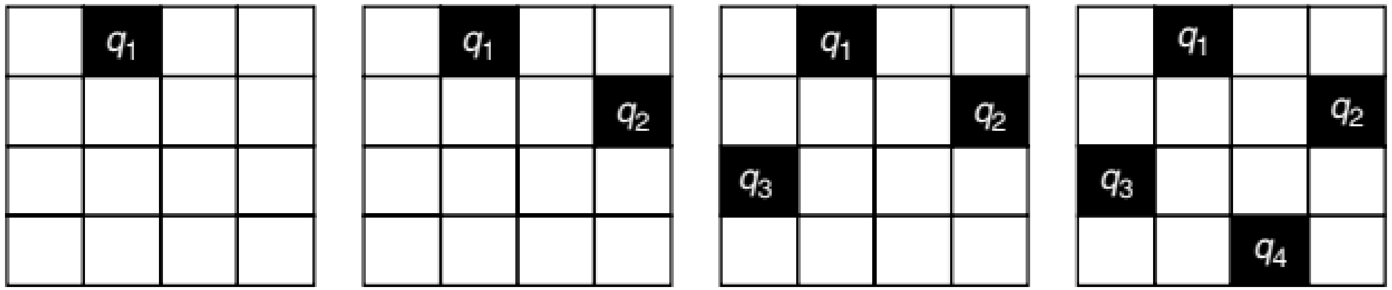
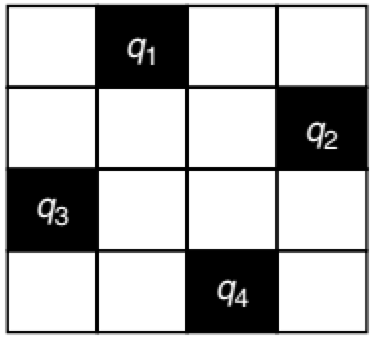


Figure 2 Solution of four-queens problem.

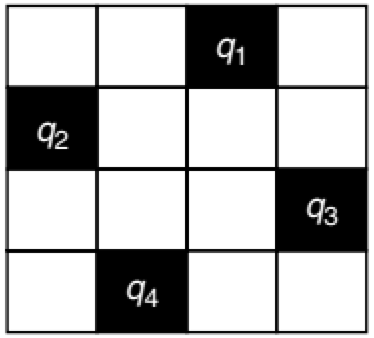
Backtracking : N-Queen Problem

4 Queen Problem:

Solution can be represented as four-tuple (x_1, x_2, x_3, x_4) where x_1 is column value in row 1 for placement of Q_1 and so on.



Solution 2, 4, 1, 3



Solution 3, 1, 4, 2

Figure 7 Two possible solutions to four-queens problem.

Backtracking : N-Queen Problem

N Queen Problem:

```
Algorithm NQueens(k, n)
// Using backtracking, this procedure prints all
// possible placements of n queens on an n × n
// chessboard so that they are nonattacking.
{
    for i := 1 to n do
    {
        if Place(k, i) then
        {
            x[k] := i;
            if (k = n) then write (x[1 : n]);
            else NQueens(k + 1, n);
        }
    }
}
```

Algorithm 7.5 All solutions to the n-queens problem

Backtracking : N-Queen Problem

N Queen Problem:

```
Algorithm Place(k, i)
// Returns true if a queen can be placed in kth row and
// ith column. Otherwise it returns false. x[ ] is a
// global array whose first (k − 1) values have been set.
// Abs(r) returns the absolute value of r.
{
    for j := 1 to k − 1 do
        if ((x[j] = i) // Two in the same column
            or (Abs(x[j] − i) = Abs(j − k)))
                // or in the same diagonal
            then return false;
    return true;
}
```

Algorithm 7.4 Can a new queen be placed?

Backtracking : N-Queen Problem

N Queen Problem:

- (4,1), (5,2), (6,3), (7,4), (8,5)

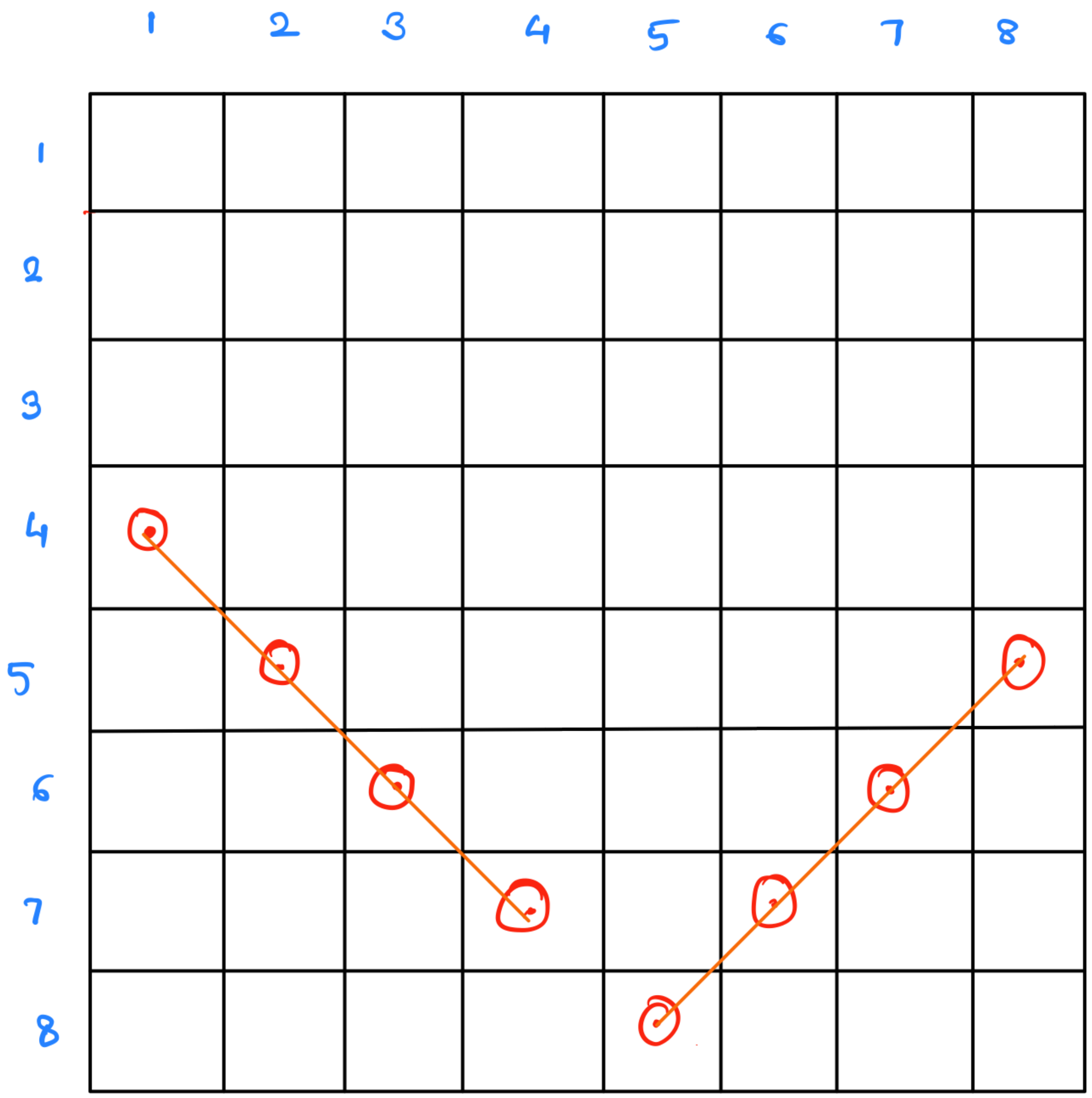
Let (i, j) and (k,l) be two cells in the chessboard.
If i-j = k-l
e.g. 4-1 = 6-3 = 3

Rearranging above equation, we have
i-k = j-l ⇒ $|i-k| = |j-l|$

- (5,8), (6,7), (7,6), (8,5)

Again let (i, j) and (k,l) be two cells in the chessboard.
If i+j = k+l
e.g. 5+8 = 6+7 = 13

Rearranging above equation, we have
i-k = l-j ⇒ $|i-k| = |j-l|$



Backtracking : Sum of Subsets

Given: 1. n distinct positive numbers (called weights w_i), where $1 \leq i \leq n$

2. Sum (m)

We need to find all possible subsets of given numbers (w_i)having sum equal to the target Sum (m).

Backtracking : Sum of Subsets

Example:

n=4; (w_1, w_2, w_3, w_4,) = (11, 13, 24,7) & m=31

Desired subsets are (11,13,7) & (24,7)

Solution vector (1, 2, 4) & (3, 4) → [Variable Length]

In general, all solution vectors are k-tuples, (x_1, x_2, x_3, x_4,) ; $1 \leq k \leq n$

Implicit Constraints:

1. **No two subsets should be same & sum of corresponding w_i's be m**
2. $x_i < x_{i+1}$ **such that** $1 \leq i \leq k$, to avoid generating multiple instances of same subset e.g. (1,2,4) & (1,4,2)

Backtracking : Sum of Subsets

Example:

n=4; (w_1, w_2, w_3, w_4,) = (11, 13, 24,7) & m=31

Another Approach: [Fixed Length]

Each solution set is represented by n-tuple (x_1, x_2, x_3, x_4) such that

$x_i \in \{0,1\}$ where $1 \leq i \leq n$

$x_i = 0 \rightarrow w_i$ *not selected*, and $x_i = 1 \rightarrow w_i$ *selected*

Therefore, Solution space of above instance are *(1,1,0,1)* & *(0,0,1,1)*

Backtracking : Sum of Subsets

Example:

n=4; (w_1, w_2, w_3, w_4,) = (11, 13, 24,7) & m=31

Another Approach: [Fixed Length]

Each solution set is represented by n-tuple (x_1, x_2, x_3, x_4) such that

$x_i \in \{0,1\}$ where $1 \leq i \leq n$

$x_i = 0 \rightarrow w_i$ *not selected*, and $x_i = 1 \rightarrow w_i$ *selected*

Therefore, Solution space of above instance are *(1,1,0,1)* & *(0,0,1,1)*

Backtracking : Sum of Subsets

Example:

S = {2, 3, 5, 6, 7, 9, 10} & M= 15

In state space tree of solution, node list values of sumSoFar, k & remWeight

Possible solutions:

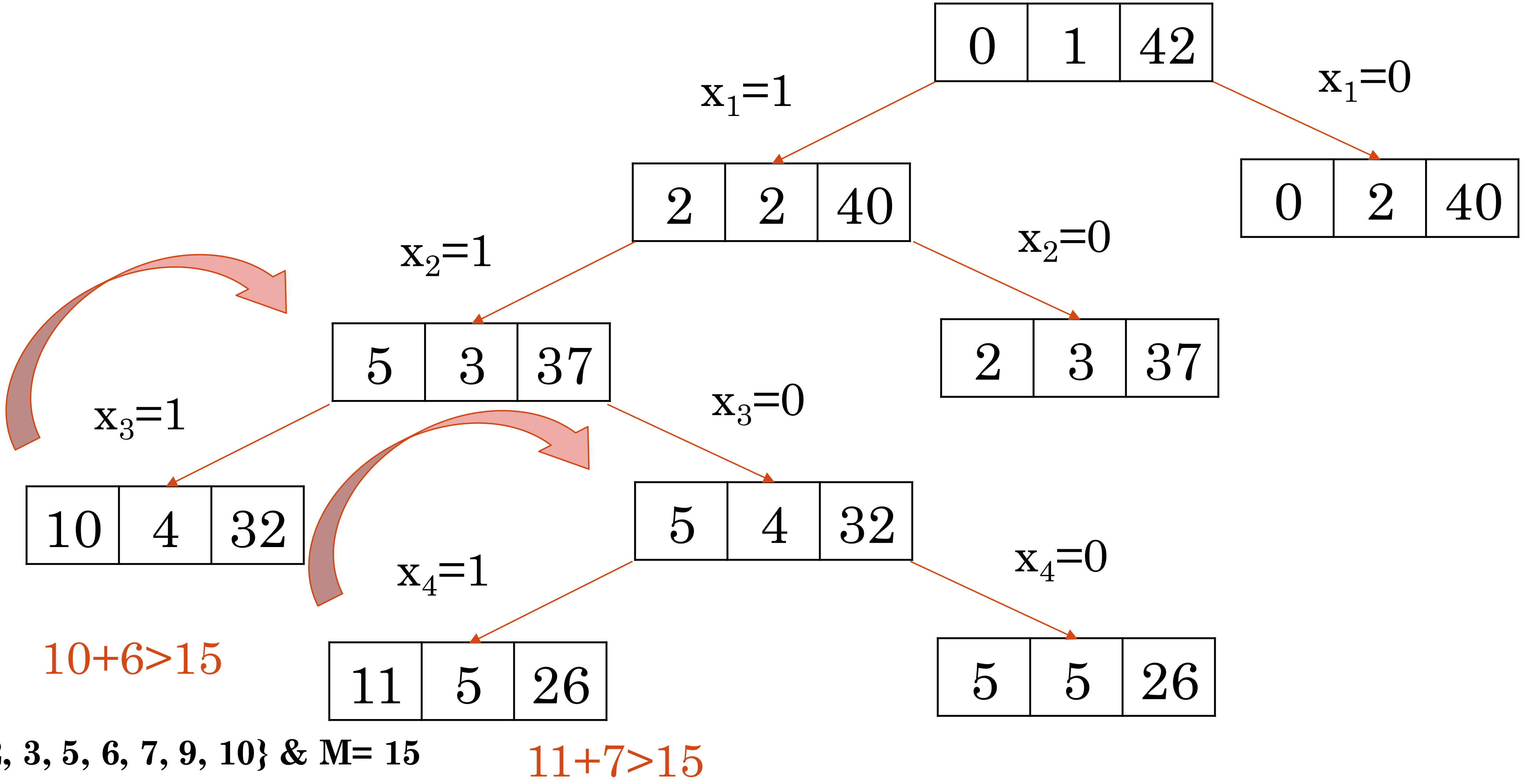
1. **{2,3,10}**
2. **{2,6,7}**
3. **{3,5,7}**
4. **{5, 10}**
5. **{6,9}**

Initialize root node with values, sumSoFar = 0, k= 1 & remWeight= 42

0	1	42
sumSoFar	**k**	**remWeight**

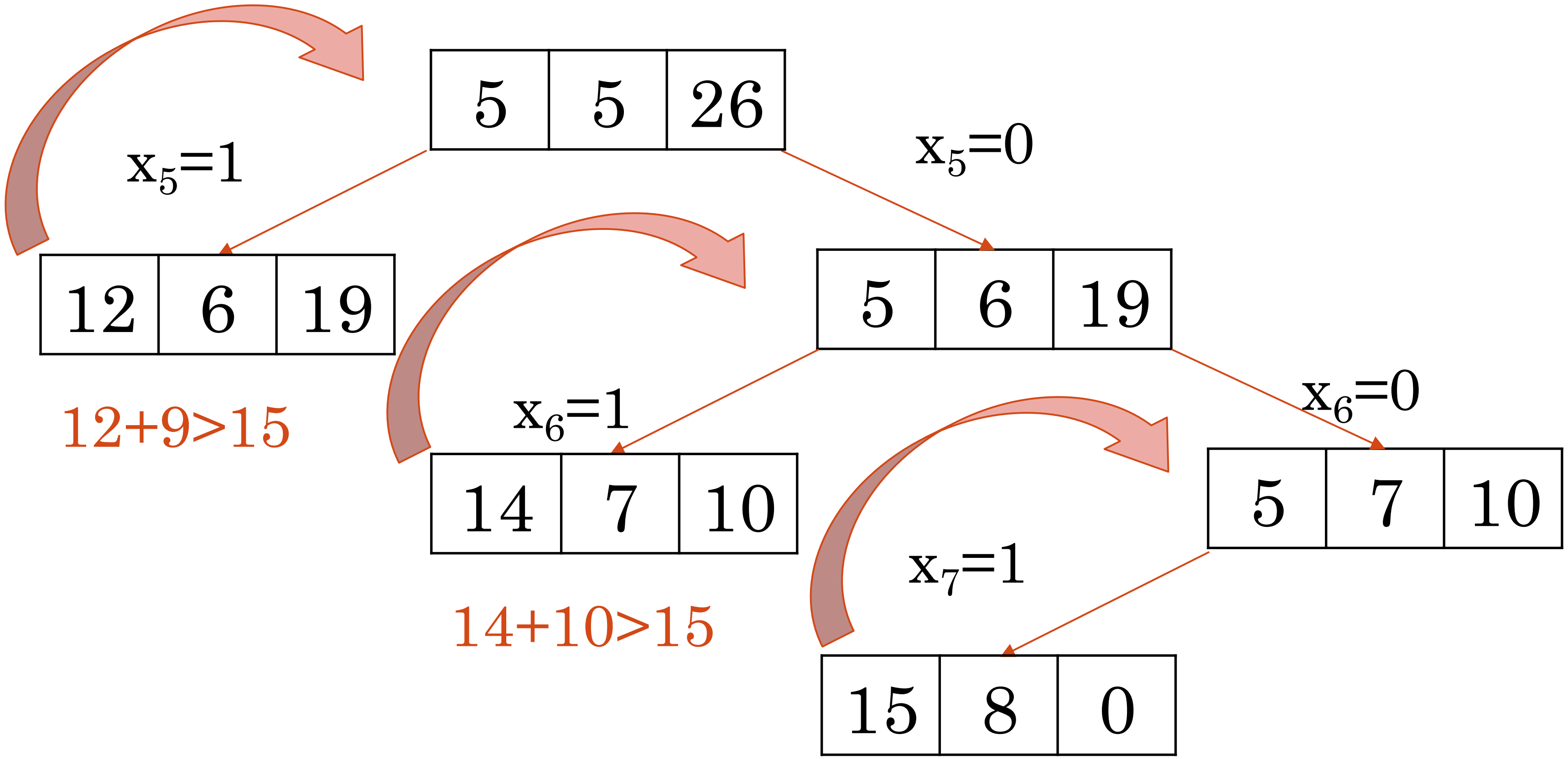
Backtracking : Sum of Subsets

Example:



S = {2, 3, 5, 6, 7, 9, 10} & M= 15

Backtracking : Sum of Subsets



S = {2, 3, 5, 6, 7, 9, 10} & M= 15

Backtracking : Sum of Subset

Algorithm 4 SUMOFSUBSETS (Sumsofar, k, remweight)

This algorithm is used to find all the solutions of the sum of subsets problem. The X[] is the solution vector.

```
1.  Set X[k]=1
2.  if (Sumsofar+w[k]=M) then
    print X[1..k]

    // solution is found
     else
            if(Sumsofar+w[k]+w[k+1]<=M) then
    // Generate Left child
            SUMOFSUBSETS(Sumsofar+w[k], k+1, remweight-w[k])
            Endif
3.  Endif
4.  if((Sumsofar+remweight-w[k]>=M) and (Sumsofar+w[k+1] ≥ M) then
    // Generate Right child
            {
             X[k]=0
            SUMOFSUBSETS(Sumsofar, k+1, remweight-w[k])
            }
5.  Stop
```

Backtracking : Graph Coloring

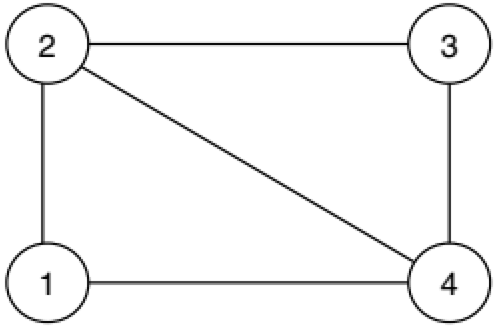
- It's a classic combinatorial Problem
- It's a problem of coloring N vertices of a given graph G in such a way that no two adjacent vertices share the same color and yet M colors are used.
- The problem is called as M coloring problem.
- <u>M coloring Decision problem</u>: M is given, whether graph can be colored using M colors
- <u>M coloring optimization problem</u>: smallest number of colors (M) required to color the graph.

Backtracking : Graph Coloring Algorithm

- Suppose we have graph $G=(V,E)$ with N vertices and M is given number of colors.
- We represent Graph G by adjacency matrix $G[n,n]$ where,
  - $G[i,j]=1$ if (i,j) is an edge of G and
  - $G[i,j]=0$ otherwise.
- If d is degree of given graph, then it can be colored with $d+1$ colors [m is referred to as **chromatic number**].
- Here colors are represented as integers 1,2,3,…,M and coloring solution will be a vector $x[1 \ldots N]$.
- So, solutions are given by n-tuple $(x_1, x_2, x_3, \ldots, x_n)$ where, $1 \le x_i \le M$ and $1 \le i \le N$ and x_i is color of node i.

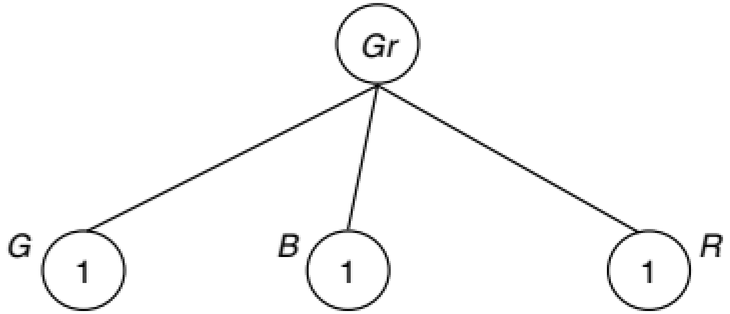
Backtracking : Graph Coloring Example

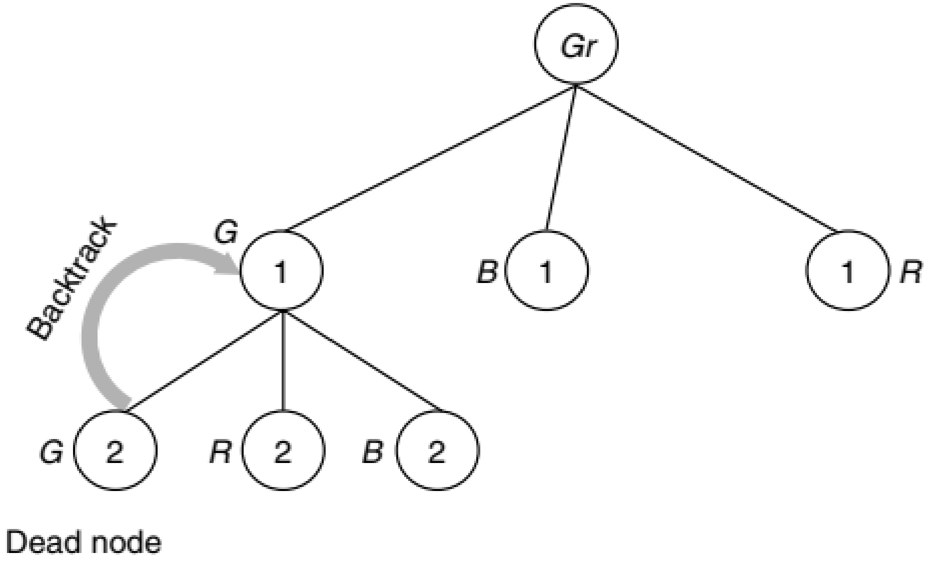
Example 1:



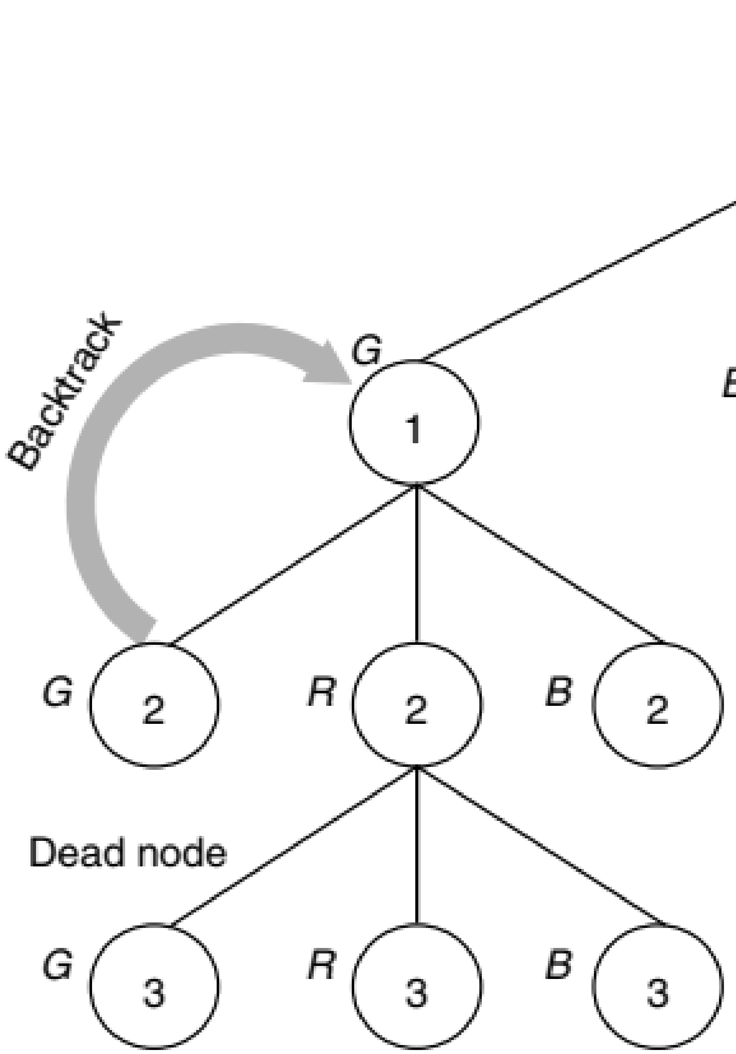
$$
\begin{array}{c c}
 & \begin{array}{cccc} \mathbf{1} & \mathbf{2} & \mathbf{3} & \mathbf{4} \end{array} \\
\begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} & \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}
\end{array}
$$

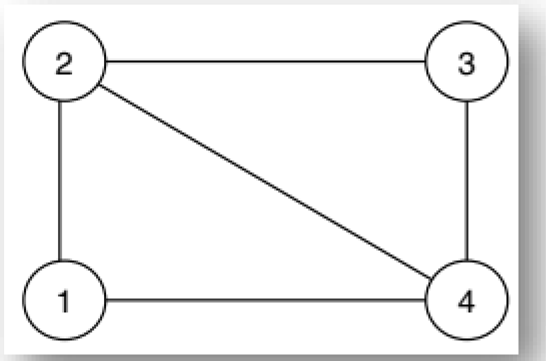
State Space Tree:



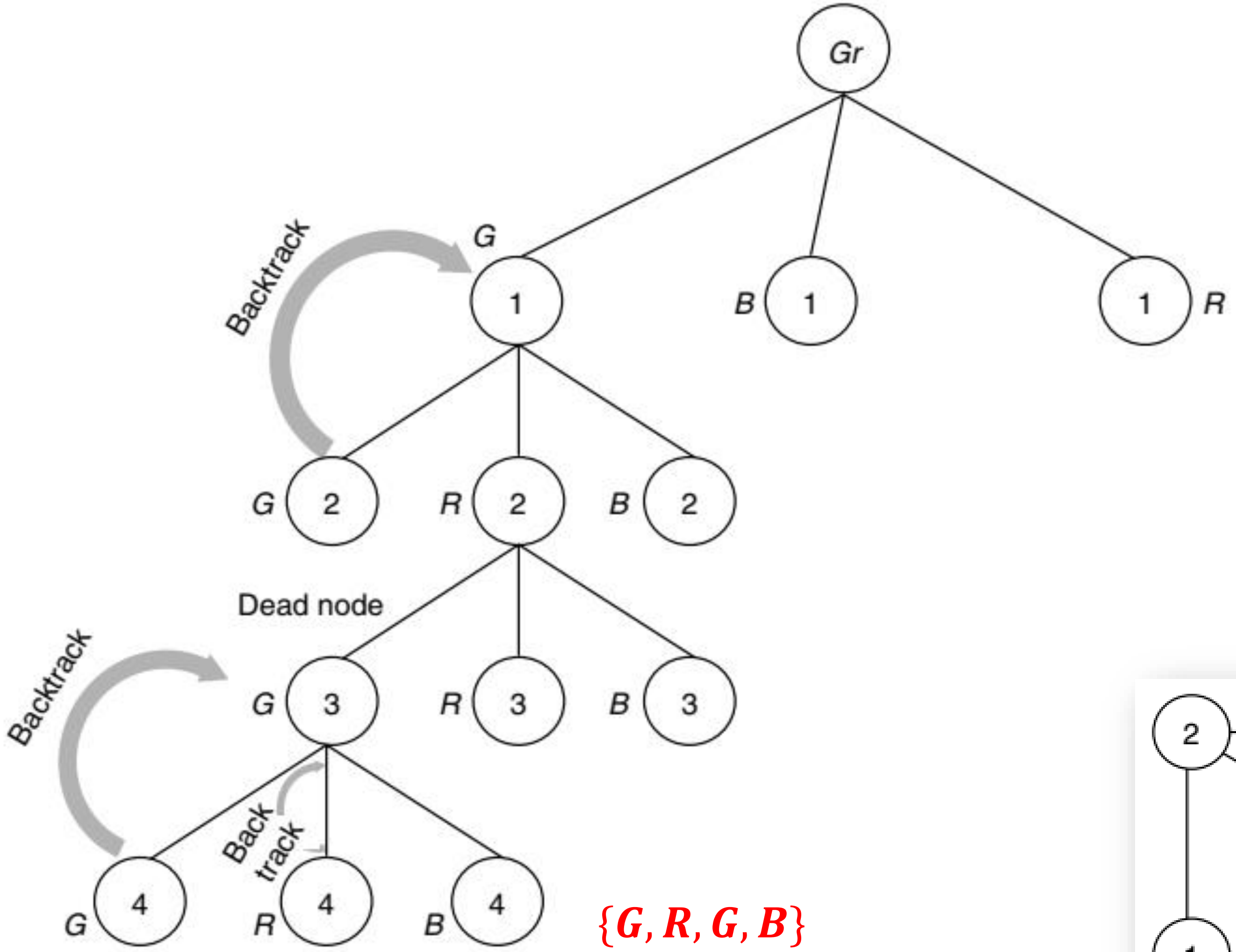


Backtracking : Graph Coloring Example

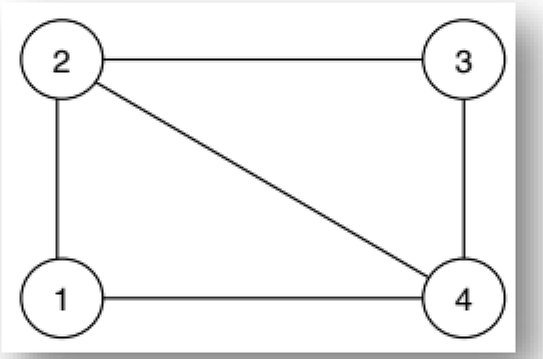




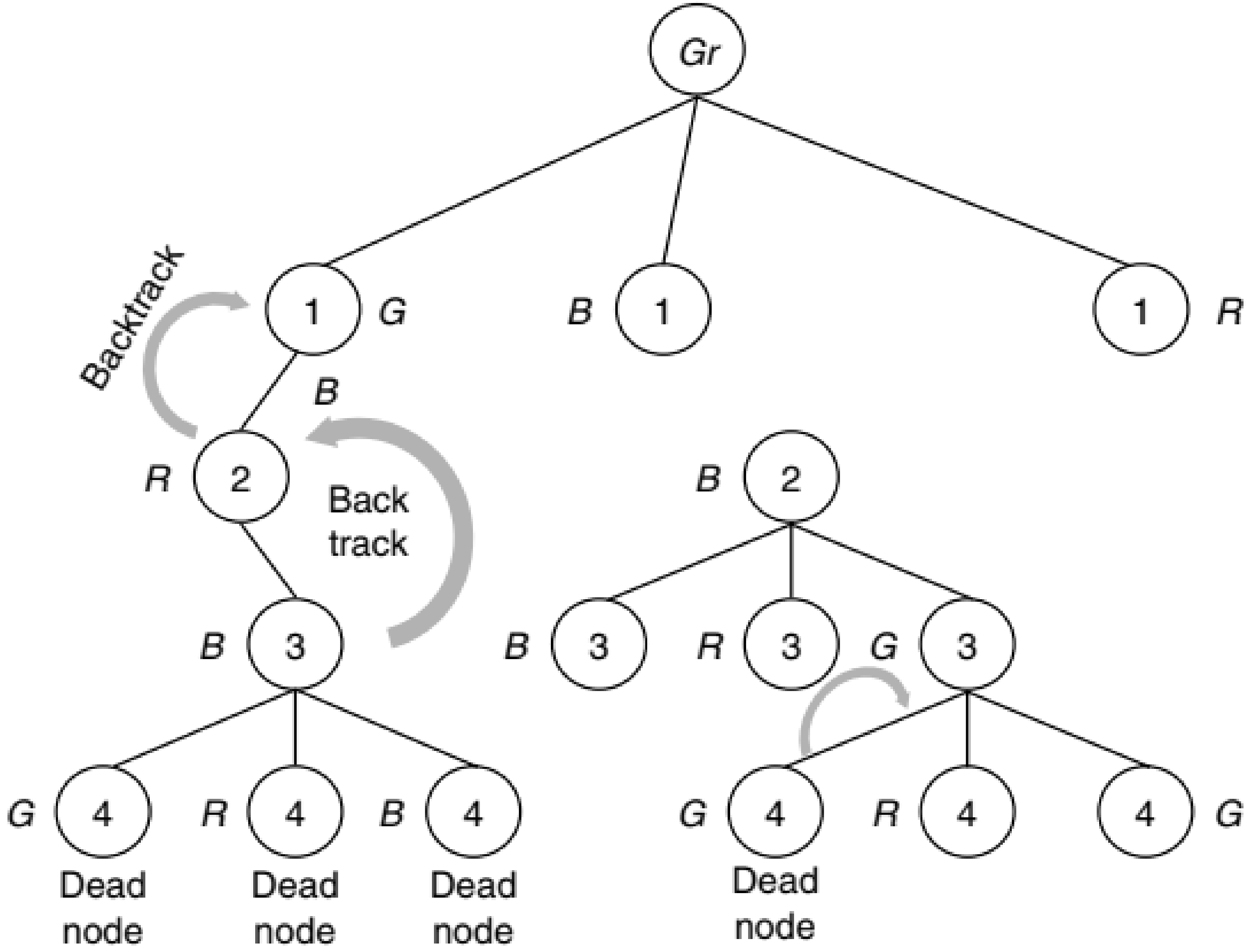
Backtracking : Graph Coloring Example

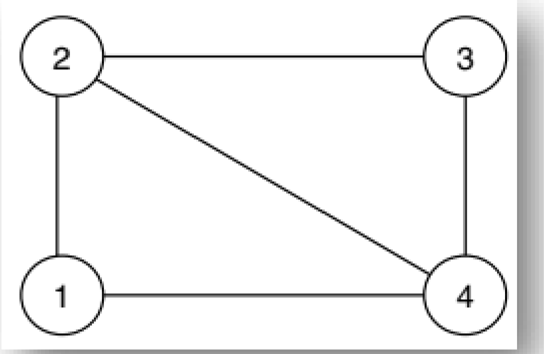


$\{G, R, G, B\}$



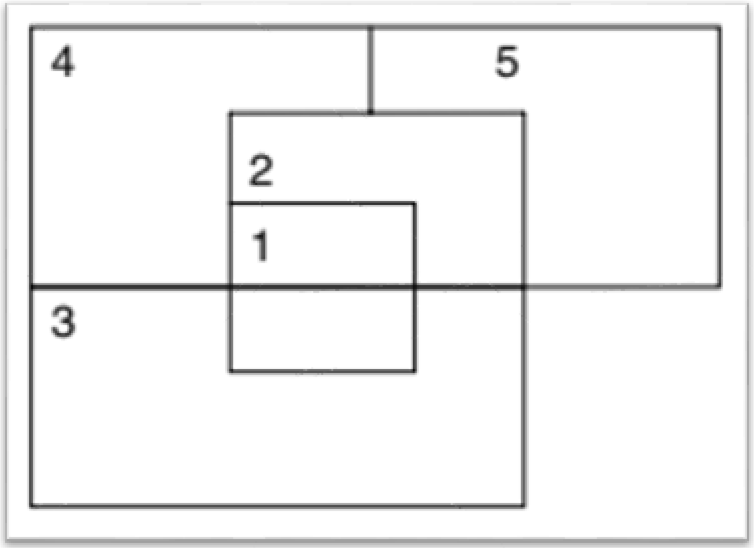
Backtracking : Graph Coloring Example



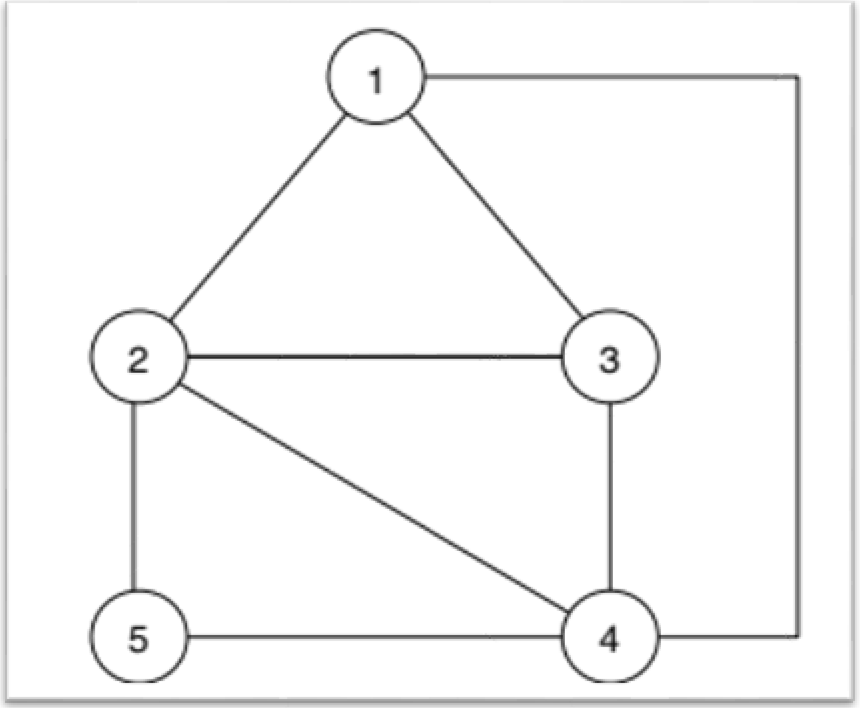


Backtracking : Graph Coloring Example

Example 2:



Map



Planer Graph

Branch & Bound : Introduction

- **Branch**: using State Space Tree (Similar to Backtracking)
- **Bound**: using Upper and Lower bounds
- Branch and Bound differs from backtracking in the sense that all the children of the E-Node are generated before any other live node becomes the E-Node.
- Branch and Bound is the generalization of both graph search strategies, BFS and DFS.
- The state space tree of the branch and bound method can be constructed using following three strategies:
  - FIFO (First In First Out) search (Implemented using QUEUE)
  - LIFO (Last In First Out) search (Implemented using STACK)
  - LC (Least Cost) search (Implemented using PRIORITY QUEUE)

Branch & Bound : Introduction

FIFO (First In First Out) Branch and Bound

- In FIFO search, queue data structure is used.
- Initially node 1 is taken as the E-node.
- The child nodes of node 1 are generated. All these live nodes are placed in a queue.
- Next the first element in the queue is deleted, i.e. node 2, the child nodes of node 2 are generated and placed in the queue.
- This continues until the answer node is found.

Branch & Bound : FIFO

LIFO (Last In First Out) Branch and Bound

Example: Job sequencing with deadlines problem

- Jobs = {J1, J2, J3, J4}; P = {10, 5, 8, 3}; d = {1, 2, 1, 2}
- Node 1 is the E-node. Child nodes of node 1 are generated and placed in the queue.

2	3	4	5	

- First element in the queue is deleted, ie., 2 is deleted and its child nodes are generated.

3	4	5	6	7	8	

- Similarly, the next element is deleted, ie., 3 and its child nodes are generated and placed in the queue. This is continued until an answer node is reached.

Branch & Bound : FIFO

FIFO (First In First Out) Branch and Bound

Example: Job sequencing with deadlines problem

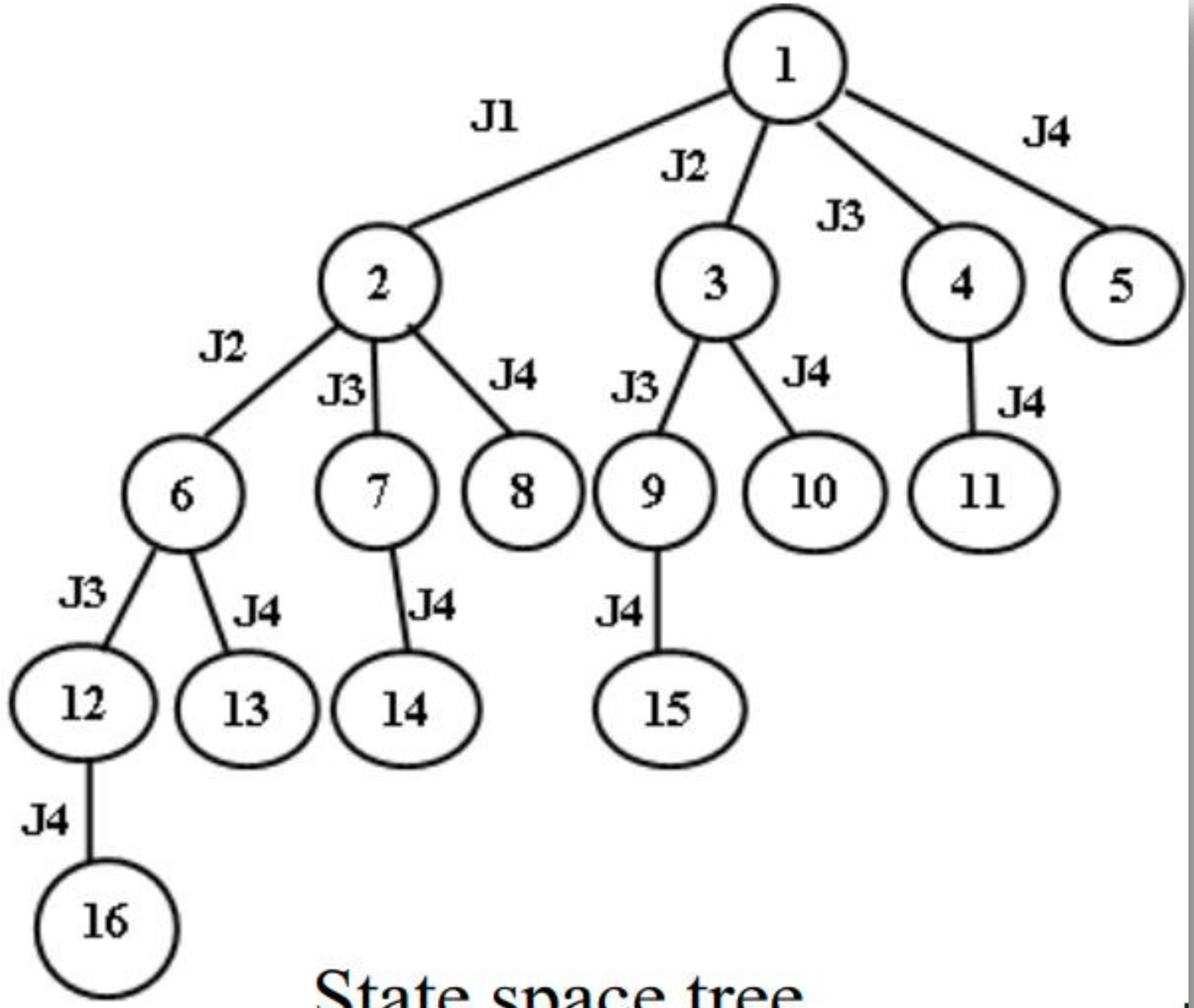
- Jobs = {J1, J2, J3, J4}; P = {10, 5, 8, 3}; d = {1, 2, 1, 2}
- Node 1 is the E-node. Child nodes of node 1 are generated and placed in the queue.

2	3	4	5	

- First element in the queue is deleted, ie., 2 is deleted and its child nodes are generated.

3	4	5	6	7	8	

- Similarly, the next element is deleted, ie., 3 and its child nodes are generated and placed in the queue. This is continued until an answer node is reached.



State space tree

Branch & Bound : LIFO

LIFO (Last In First Out) Branch and Bound

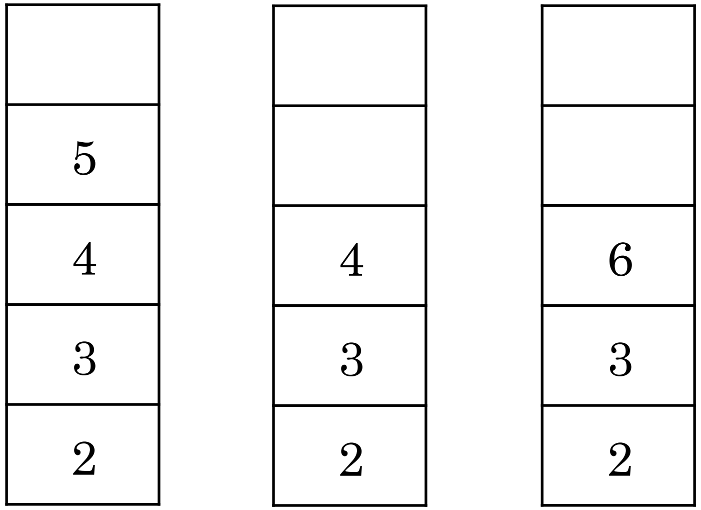
- In LIFO search, stack data structure is used.
- Initially node 1 is taken as the E-node.
- The child nodes of node 1 are generated. All these live nodes are placed in a stack.
- Next the first element in the stack is deleted, i.e. node 5, the child nodes of node 5 are generated and placed in the stack.
- This continues until the answer node is found.

Branch & Bound : LIFO

FIFO (First In First Out) Branch and Bound

Example: Job sequencing with deadlines problem

- Jobs = {J1, J2, J3, J4}; P = {10, 5, 8, 3}; d = {1, 2, 1, 2}
- Node 1 is the E-node. Child nodes of node 1 are generated and placed in the stack.
- First element in the stack is deleted, i.e., 5 is deleted and its child nodes are generated.
- Similarly, the next element is deleted, i.e., 4 and its child nodes are generated and placed in the queue. This is continued until an answer node is reached.

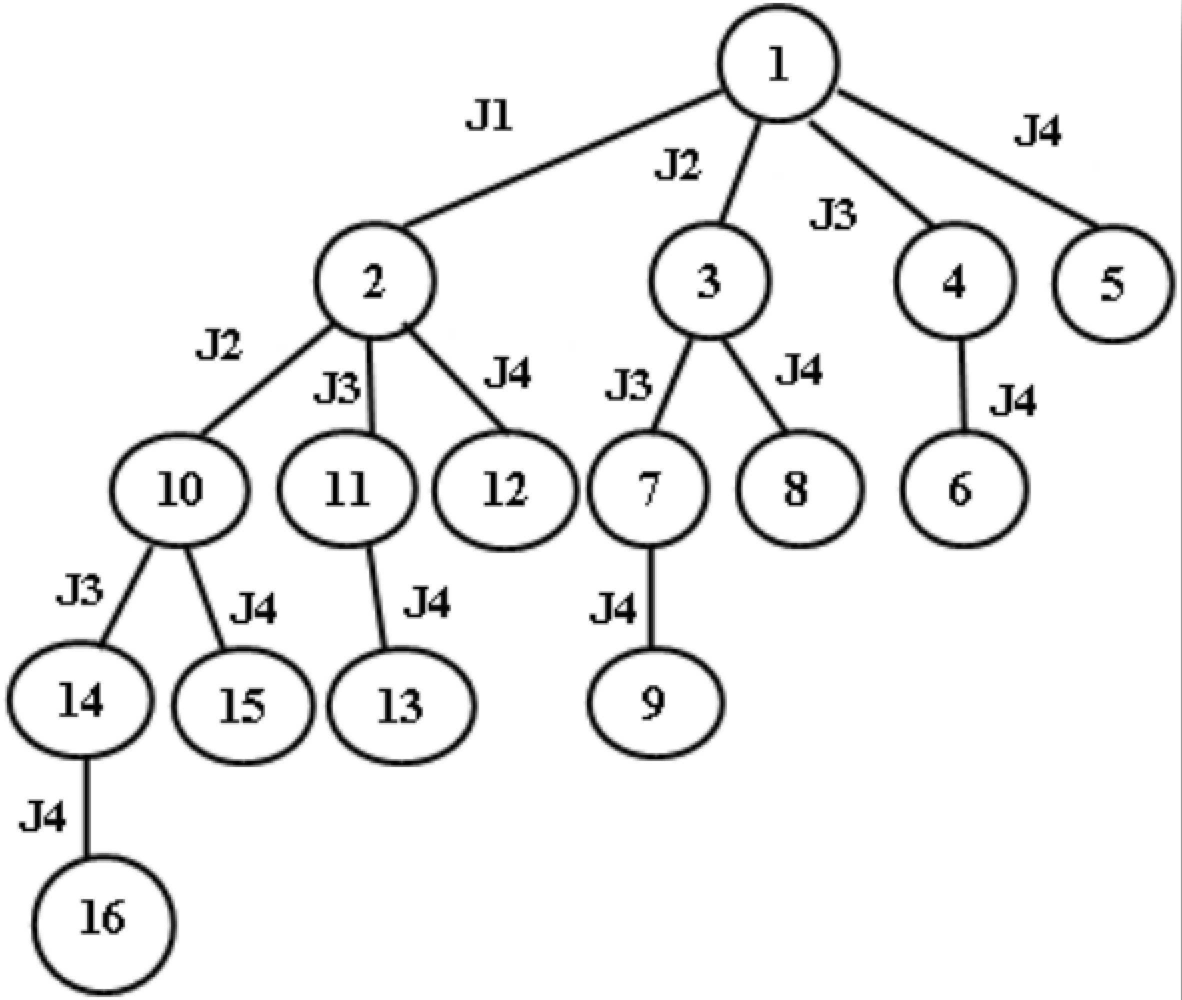


Branch & Bound : LIFO

FIFO (First In First Out) Branch and Bound

Example: Job sequencing with deadlines problem

- Jobs = {J1, J2, J3, J4}; P = {10, 5, 8, 3}; d = {1, 2, 1, 2}
- Node 1 is the E-node. Child nodes of node 1 are generated and placed in the stack.
- First element in the stack is deleted, i.e., 5 is deleted and its child nodes are generated.
- Similarly, the next element is deleted, i.e., 4 and its child nodes are generated and placed in the queue. This is continued until an answer node is reached.



State space tree

Branch & Bound : LCBB

LC (Least Count) Branch and Bound:

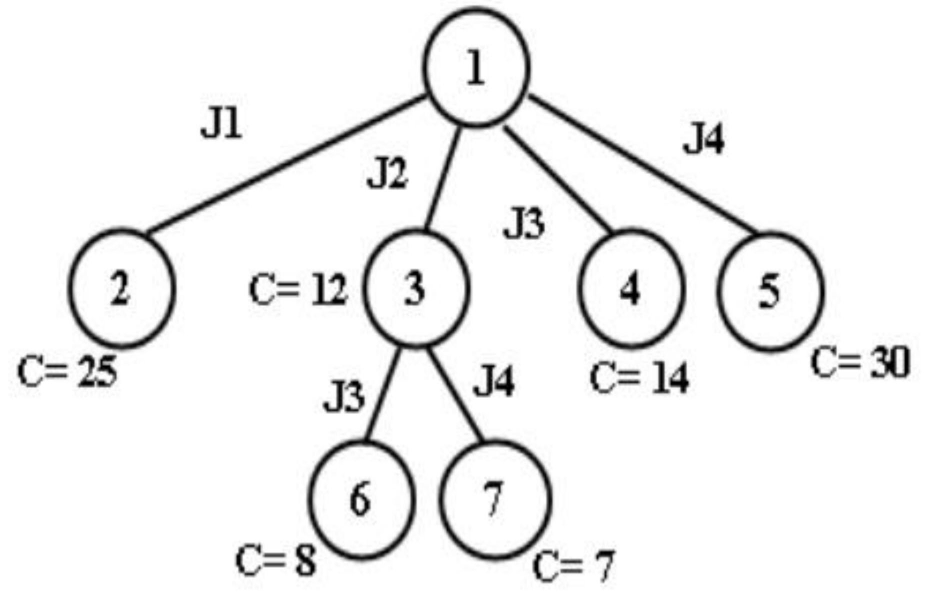
- In both FIFO and LIFO Branch and Bound the selection rules for the next E-node in rigid and blind.
- The selection rule for the next E-node does not give any preferences to a node that has a very good chance of getting the search to an answer node quickly.
- In this method ranking function or cost function is used.
- The child nodes of the E-node are generated, among these live nodes; a node which has minimum cost is selected. By using ranking function, the cost of each node is calculated.

Branch & Bound : LCBB

LC (Least Count) Branch and Bound:

Example: Job sequencing with deadlines problem

- Jobs = {J1, J2, J3, J4}; P = {10, 5, 8, 3}; d = {1, 2, 1, 2}
- Initially we will take node 1 as E-node. Generate children of node 1, the children are 2, 3, 4, 5. By using ranking function we will calculate the cost of 2, 3, 4, 5 nodes is $\hat{c}$ =25, $\hat{c}$ =12, $\hat{c}$ =14, $\hat{c}$ =30 respectively.
- Now we will select a node which has minimum cost i.e., node 3. For node 3, the children are 6, 7.



State space tree

Branch & Bound : LCBB

LC (Least Count) Branch and Bound:

- All the live nodes are stored in a PRIORITY QUEUE or HEAP.
- The live nodes are not selected according to the order in which they have been queued or stacked but according to their heuristic value.
- The heuristic value is calculated for each live node and then the node with the highest heuristic value is chosen as the E-node.

Branch & Bound : 0/1 Knapsack Problem

LC (Least Count) Branch and Bound:

- The 0/1 knapsack problem is to

$$\text{Maximize} \sum_{i=1}^{n} p_i x_i \quad \text{subject to} \sum_{i=1}^{n} w_i x_i \le M$$

- objective of this problem is to fill the knapsack in order to maximize the profit subject to its capacity.
- But Branch & Bound is used for **minimization problem**.

Branch & Bound : 0/1 Knapsack Problem

LC (Least Count) Branch and Bound:

- This modified knapsack problem is stated as,
- The 0/1 knapsack problem is the maximization problem where the value of the objective function $\hat{c}(x) = \sum p_i \, x_i$ is maximized subjected to $\sum w_i x_i \leq M$,
- Now our aim is minimization, so we take the objective function $\hat{c}(x) = -\sum pi\ xi$ subjected to $\sum w_i x_i \leq M$ in order to convert the 0/1 knapsack problem as the minimization problem where $x_i = 0$ or 1, $1 \leq i \leq n$
- The two functions $\hat{c}(x)$ and $U(x)$ are defined using two algorithms Bound and UBound .

Branch & Bound : 0/1 Knapsack Problem

LC (Least Count) Branch and Bound:

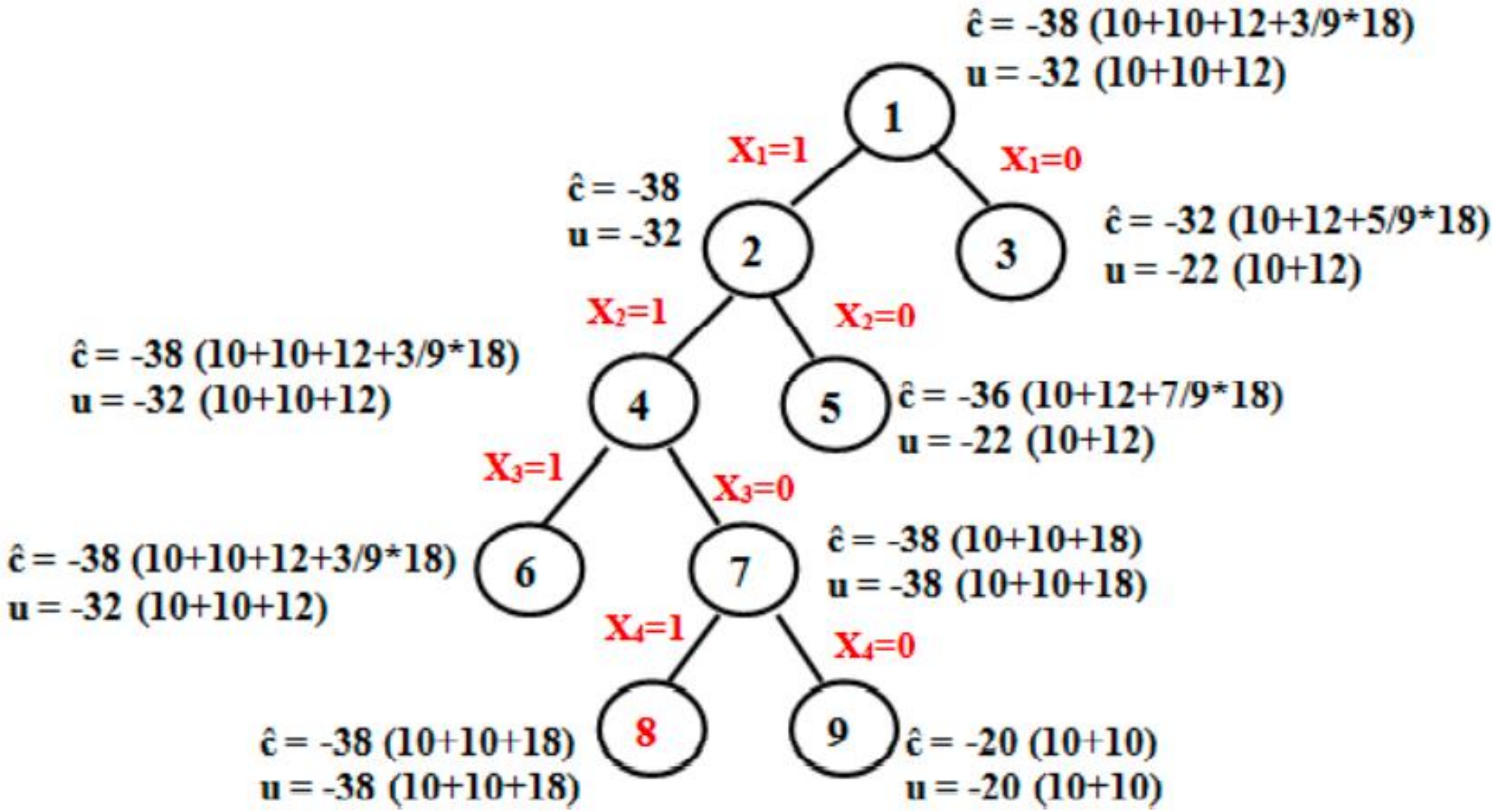
- UBound computes the weights of the list of objects placed in the knapsack as a whole and their sum $\leq m$, and the profit is correspondingly decremented from initial profit and returned.
- Bound is similar to UBound but it considers fractional objects to use the entire capacity of the sack $\Sigma w_i x_i = m$.

Branch & Bound : 0/1 Knapsack Problem

$n = 4;\ m = 15;$

$(p_1, p_2, p_3, p_4) = \{10, 10, 12, 18\};\ (w_1, w_2, w_3, w_4) = \{2, 4, 6, 9\}$

$x_1 = 1,$
$x_2 = 1,$
$x_3 = 0,$
$x_4 = 1$



Branch & Bound : 15 Puzzle Problem

- The 15 Puzzle problem is invented by Sam Loyd in 1878.
- The problem consist of 15 numbered (0-15) tiles on a square box with 16 tiles(one tile is blank or empty).
- The objective of this problem is to change the arrangement of initial node to goal node by using series of legal moves.
- The Initial and Goal node arrangement is shown by following figure

Initial state

1	2	3	4
5	6		8
9	10	7	11
13	14	15	12

Goal state

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

Figure 19 Initial and goal states for 15-puzzle problem.

Branch & Bound : 15 Puzzle Problem

- In initial node four moves are possible. User can move any one of the tile like 2,or 3, or 5, or 6 to the empty tile. From this we have four possibilities to move from initial node.
- The legal moves are for adjacent tile number is left, right, up, down, ones at a time.
- Each and every move creates a new arrangement, and this arrangement is called state of puzzle problem.
- By using different states, a state space tree diagram is created, in which edges are labeled according to the direction in which the empty space moves.
- The LCBB method is the general method used to solve the 15-puzzle problem so that the goal state can be achieved in minimum number of tile movement.

Branch & Bound : 15 Puzzle Problem

- In state space tree, nodes are numbered as per the level. In each level we must calculate the value or cost of each node by using given formula:

$$C(x)=f(x)+g(x),$$

- f(x) is length of path from root or initial node to node x,
- g(x) is estimated length of path from x downward to the goal node. Number of non-blank tile not in their correct position.
- C(x)< Infinity.(initially set bound).
- Each time node with smallest cost is selected for further expansion towards goal node. This node become the e-node.

Branch & Bound : 15 Puzzle Problem

- Example:

Solve the given15-puzzle problem using LCBB.

Initial state

1	2	3	4
5	6		8
9	10	7	11
13	14	15	12

Goal state

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

Branch & Bound : 15 Puzzle Problem

- Example: