# UNIT-5

**Back tracking:** General method, N-Queens problem, Sum of subsets, Graph colouring problem.

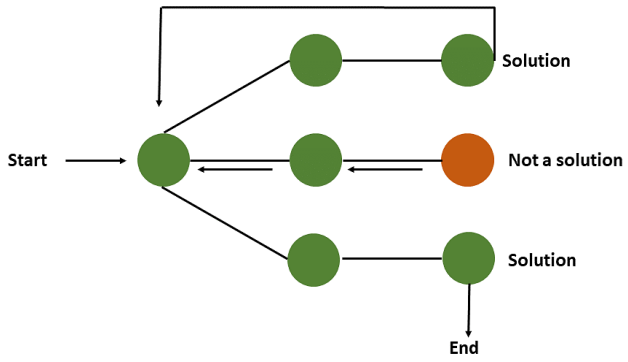
**Branch and bound:** General method, Least cost (LC) search, 0/1 Knapsack problem, Travelling salesperson problem.

## 1. BACKTRACKING –General Method

- Backtracking is an algorithmic technique whose goal is to use brute force to find all solutions to a problem.
- It entails gradually compiling a set of all possible solutions. Because a problem will have constraints, solutions that do not meet them will be removed.
- It finds a solution by building a solution step by step, increasing levels over time, using recursive calling.
- A search tree known as the state-space tree is used to find these solutions. Each branch in a state-space tree represents a variable, and each level represents a solution.
- A backtracking algorithm uses the depth-first search method.
- When the algorithm begins to explore the solutions, the abounding function is applied so that the algorithm can determine whether the proposed solution satisfies the constraints.
- If it does, it will keep looking. If it does not, the branch is removed, and the algorithm returns to the previous level.

## State-Space Tree

A space state tree is a tree that represents all of the possible states of the problem, from the root as an initial state to the leaf as a terminal state.



- Backtracking is used to solve problem in which a sequence of objects is chosen from a specified set so that the sequence satisfies some criterion.
- The desired solution is expressed as an n-tuple (x1, . . . . , xn) where each xi Є S, S being a finite set.
- The solution is based on finding one or more vectors that maximize, minimize, or satisfy a criterion function P (x1…. xn). Form a solution and check at every step if this has any chance of success. If the solution at any point seems not promising, ignore it.
- All solutions require a set of constraints divided into two categories: explicit and implicit constraints.
- **Definition 1**: Explicit constraints are rules that restrict each xi to take on values only from a given set. Explicit constraints depend on the particular instance I of problem being solved. All tuples that satisfy the explicit constraints define a possible solution space for I.
- **Definition 2:** Implicit constraints are rules that determine which of the tuples in the solution space of I satisfy the criterion function. Thus, implicit constraints describe the way in which the xi's must relate to each other.

```
1   Algorithm Backtrack(k)
2   // This schema describes the backtracking process using
3   // recursion. On entering, the first k − 1 values
4   // x[1], x[2], . . . , x[k − 1] of the solution vector
5   // x[1 : n] have been assigned. x[ ] and n are global.
6   {
7       for (each x[k] ∈ T(x[1], . . . , x[k − 1])) do
8       {
9           if (Bk(x[1], x[2], . . . , x[k]) ≠ 0) then
10          {
11              if (x[1], x[2], . . . , x[k] is a path to an answer node)
12                  then write (x[1 : k]);
13              if (k < n) then Backtrack(k + 1);
14          }
15      }
16  }
```

```
1   Algorithm IBacktrack(n)
2   // This schema describes the backtracking process.
3   // All solutions are generated in x[1 : n] and printed
4   // as soon as they are determined.
5   {
6       k := 1;
7       while (k ≠ 0) do
8       {
9           if (there remains an untried x[k] ∈ T(x[1], x[2], ...,
10              x[k − 1])  and B_k(x[1], ..., x[k]) is true) then
11          {
12              if (x[1], ..., x[k] is a path to an answer node)
13                  then write (x[1 : k]);
14              k := k + 1; // Consider the next set.
15          }
16          else k := k − 1; // Backtrack to the previous set.
17      }
18  }
```

## 2. N-Queens Problem

- The **N** Queen is the problem of placing **N** chess queens on an **N×N** chessboard so that no two queens attack each other.
- For example, the following is a solution for the 4 Queen problems.
- Let us consider, N = 8. Then 8-Queens Problem is to place eight queens on an 8 x 8 chessboard so that no two "attack", that is, no two of them are on the same row, column, or diagonal.
- All solutions to the 8-queens problem can be represented as 8-tuples (x1, . . . . , x8), where xi is the column of the ith row where the ith queen is placed.
- The explicit constraints using this formulation are Si = {1, 2, 3, 4, 5, 6, 7, 8}, 1 < i < 8. Therefore the solution space consists of 88 8-tuples. The implicit constraints for this problem are that no two xi's can be the same (i.e., all queens must be on different columns) and no two queens can be on the same diagonal.
- This realization reduces the size of the solution space from 88 tuples to 8! Tuples.

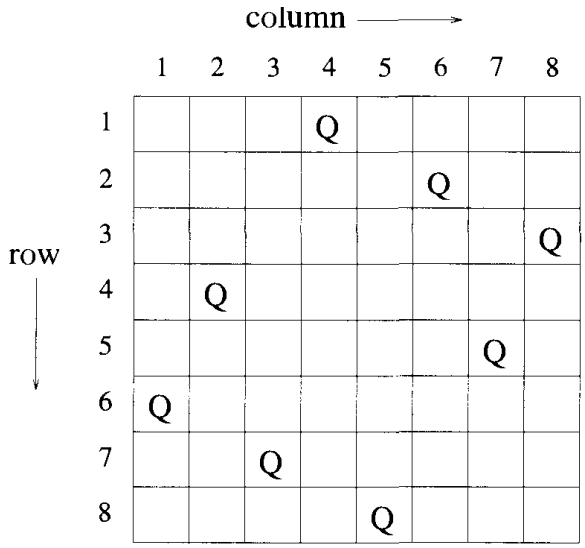



Figure: One solution of 8-queens problem

```
1   Algorithm Place(k, i)
2   // Returns true if a queen can be placed in kth row and
3   // ith column. Otherwise it returns false. x[ ] is a
4   // global array whose first (k − 1) values have been set.
5   // Abs(r) returns the absolute value of r.
6   {
7       for j := 1 to k − 1 do
8           if ((x[j] = i) // Two in the same column
9               or (Abs(x[j] − i) = Abs(j − k)))
10                  // or in the same diagonal
11              then return false;
12      return true;
13  }
```

Algorithm: Can a new queen be placed

```
1   Algorithm NQueens(k, n)
2   // Using backtracking, this procedure prints all
3   // possible placements of n queens on an n × n
4   // chessboard so that they are nonattacking.
5   {
6       for i := 1 to n do
7       {
8           if Place(k, i) then
9           {
10              x[k] := i;
11              if (k = n) then write (x[1 : n]);
12              else NQueens(k + 1, n);
13          }
14      }
15  }
```

Algorithm: All solutions to the n-queens problem

**4 – Queens Problem:**

- Let us see how backtracking works on the 4-queens problem. We start with the root node as the only live node.
- This becomes the E-node. We generate one child. Let us assume that the children are generated in ascending order.
- Let us assume that the children are generated in ascending order. Thus node number 2 of figure is generated and the path is now (1). This corresponds to placing queen 1 on column 1. Node 2 becomes the E-node.
- Node 3 is generated and immediately killed. The next node generated is node 8 and the path becomes (1, 3). Node 8 becomes the E-node.
- However, it gets killed as all its children represent board configurations that cannot lead to an answer node.

We backtrack to node 2 and generate another child, node 13. The path is now (1, 4). The board configurations as backtracking proceeds is as follows:

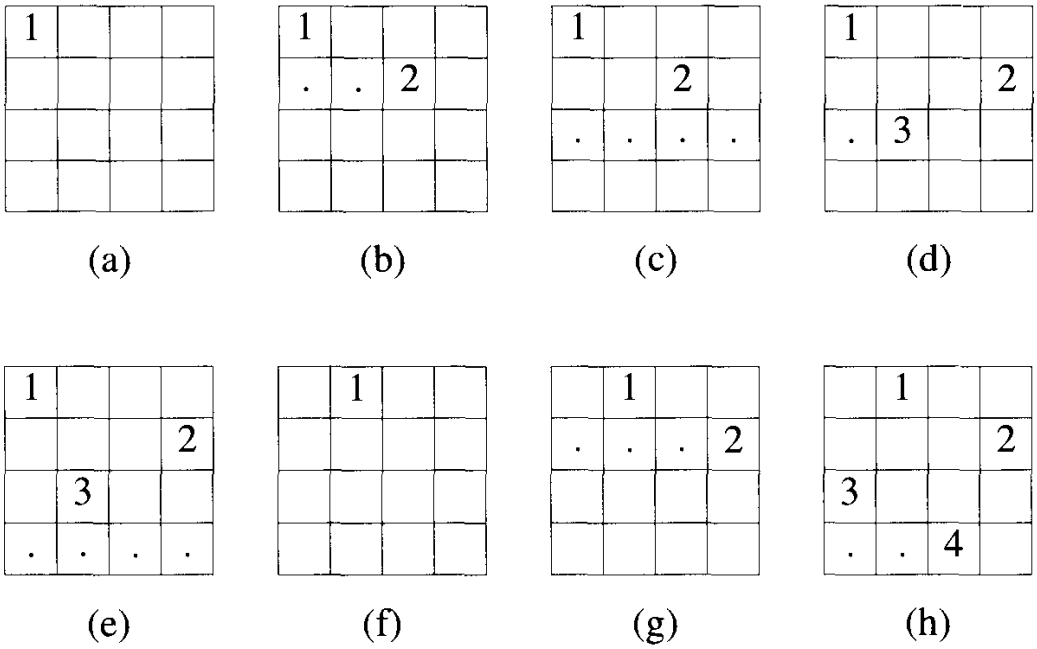



Figure: Example of a backtrack solution to the 4-queensproblem

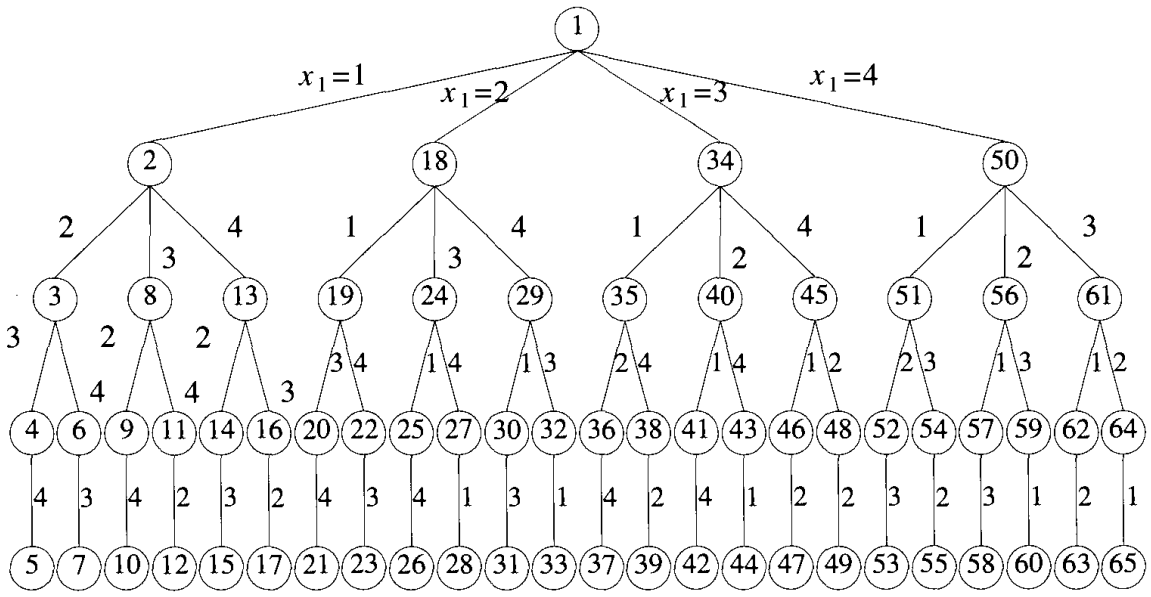



Figure: Tree organizationof the 4-queens solution space.Nodesare numbered as in depth first search.

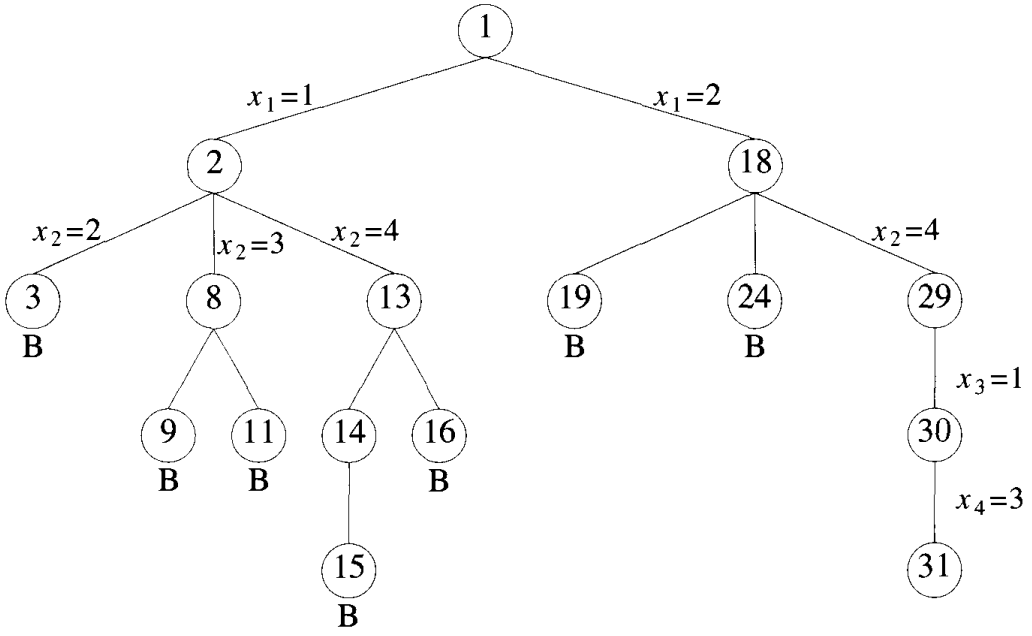



Figure: Portion of the state space tree for 4-queens problem during backtracking

## 3. Sum of Subsets

Given positive numbers wi, $1 \le i \le n$, and m, this problem requires finding all subsets of $w_i$ whose sums are 'm'.
All solutions are k-tuples, $1 \le k \le n$.

**Explicit constraints:**
xi **Є** {j | j is an integer and $1 \le j \le n$}.

**Implicit constraints:**
No two xi can be the same.

The sum of the corresponding wi's be m.

$x_i < x_{i+1}$ , $1 \le i < k$ (total order in indices) to avoid generating multiple instances of the same subset (for example, (1, 2, 4) and (1, 4, 2) represent the same subset).

A better formulation of the problem is where the solution subset is represented by an n-tuple (x1, . . . . . , xn) such that xi **Є** {0, 1}.
The above solutions are then represented by (1, 1, 0, 1) and (0, 0, 1, 1). For both the above formulations, the solution space is 2n distinct tuples.

```
1   Algorithm SumOfSub(s, k, r)
2   // Find all subsets of w[1 : n] that sum to m. The values of x[j],
3   // 1 ≤ j < k, have already been determined. s = Σ_{j=1}^{k-1} w[j] * x[j]
4   // and r = Σ_{j=k}^{n} w[j]. The w[j]'s are in nondecreasing order.
5   // It is assumed that w[1] ≤ m and Σ_{i=1}^{n} w[i] ≥ m.
6   {
7       // Generate left child. Note: s + w[k] ≤ m since B_{k-1} is true.
8       x[k] := 1;
9       if (s + w[k] = m) then write (x[1 : k]); // Subset found
10          // There is no recursive call here as w[j] > 0, 1 ≤ j ≤ n.
11      else if (s + w[k] + w[k + 1] ≤ m)
12          then SumOfSub(s + w[k], k + 1, r - w[k]);
13      // Generate right child and evaluate B_k.
14      if ((s + r - w[k] ≥ m) and (s + w[k + 1] ≤ m)) then
15      {
16          x[k] := 0;
17          SumOfSub(s, k + 1, r - w[k]);
18      }
19  }
```

Algorithm: Recursive Backtrack algorithm for sum of subsets problem

For example, n = 4, w = (11, 13, 24, 7) and m = 31, the desired subsets are (11, 13, 7) and (24, 7).

The following figure shows a possible tree organization for two possible formulations of the solution space for the case n = 4.

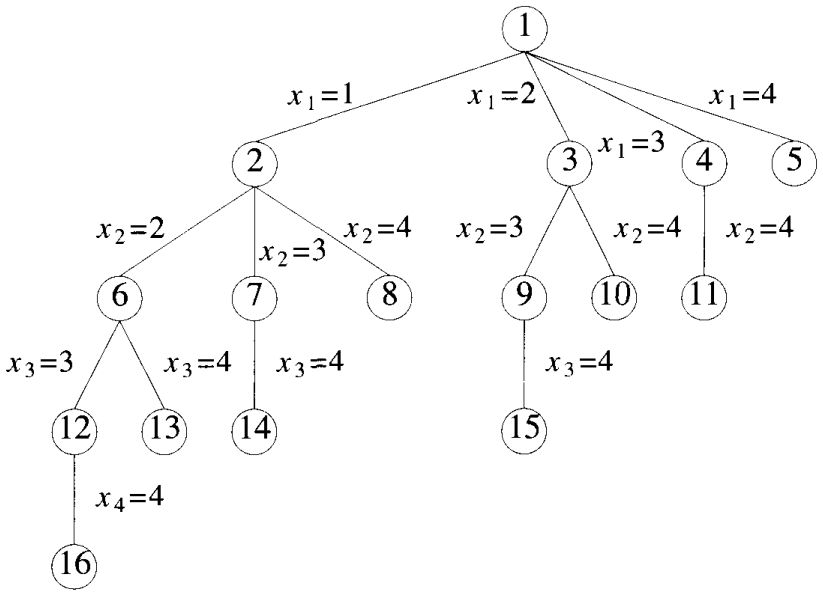



Figure: A possible solution space organization for the sum of subsets problem. Nodes are numbered as in breadth-first search

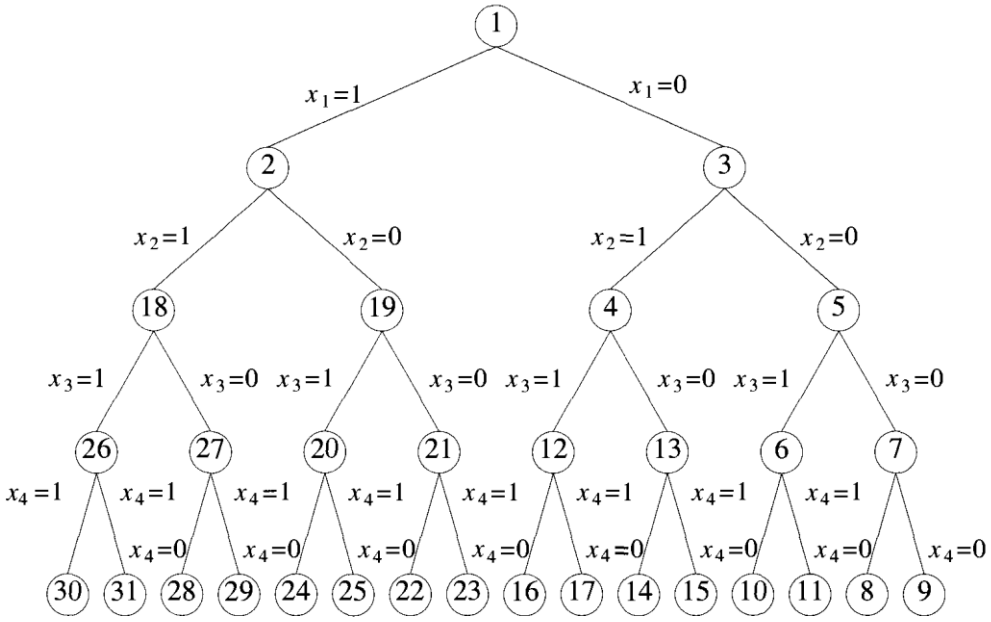



Figure: Another possible organization for the sum of subsets problems. Nodes are numbered as in D-search

## 4. Graph Coloring

- Let G be a graph and m be a given positive integer.
- We want to discover whether the nodes of G can be colored in such a way that no two adjacent nodes have the same color, yet only m colors are used.
- This is termed the m-colorabiltiy decision problem.
- The m-colorability optimization problem asks for the smallest integer m for which the graph G can be colored.
- Given any map, if the regions are to be colored in such a way that no two adjacent regions have the same color, only four colors are needed.
- For many years it was known that five colors were sufficient to color any map, but no map that required more than four colors had ever been found.
- After several hundred years, this problem was solved by a group of mathematicians with the help of a computer.
- They showed that in fact four colors are sufficient for planar graphs.
- The function m-coloring will begin by first assigning the graph to its adjacency matrix, setting the array x [] to zero.
- The colors are represented by the integers 1, 2, . . . , m and the solutions are given by the n-tuple (x1, x2, . . ., xn), where xi is the color of node i.
- A recursive backtracking algorithm for graph coloring is carried out by invoking the statement mcoloring(1);

**Algorithm mcoloring** (k)

```
// This algorithm was formed using the recursive backtracking schema. The graph is
// represented by its Boolean adjacency matrix G [1: n, 1: n]. All assignments of
// 1, 2, ..... , m to the vertices of the graph such that adjacent vertices are assigned
// distinct integers are printed. k is the index of the next vertex to color.
{
repeat
{ // Generate all legal assignments for x[k].
NextValue (k); // Assign to x [k] a legal color. If (x [k] = 0) then return; // No new color possible
If (k = n) then // at most m colors have been
// used to color the n vertices.
write (x [1: n]);
else mcoloring (k+1);
} until (false);
}
```

**Algorithm NextValue** (k)

```
// x [1] , ..... x [k-1] have been assigned integer values in the range [1, m] such that
// adjacent vertices have distinct integers. A value for x [k] is determined in the range
// [0, m].x[k] is assigned the next highest numbered color while maintaining distinctness
// from the adjacent vertices of vertex k. If no such color exists, then x [k] is 0.
{
repeat
{
x [k]: = (x [k] +1) mod (m+1) // Next highest color.
If (x [k] = 0) then return; // All colors have been used for j := 1 to n do
{ // check if this color is distinct from adjacent colors if ((G [k, j] □ 0) and (x [k] = x [j]))
// If (k, j) is and edge and if adj. vertices have the same color. then break;
}
if (j = n+1) then return; // New color found
} until (false); // Otherwise try to find another color.
}
```

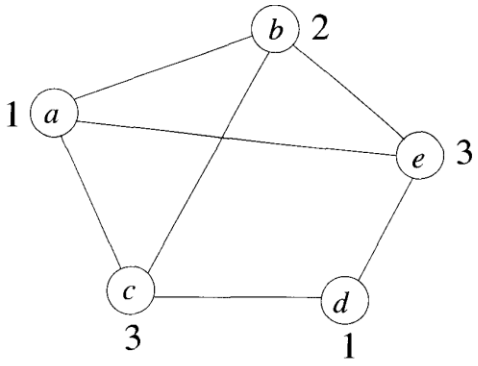



Figure: Example of Graph coloring Problem

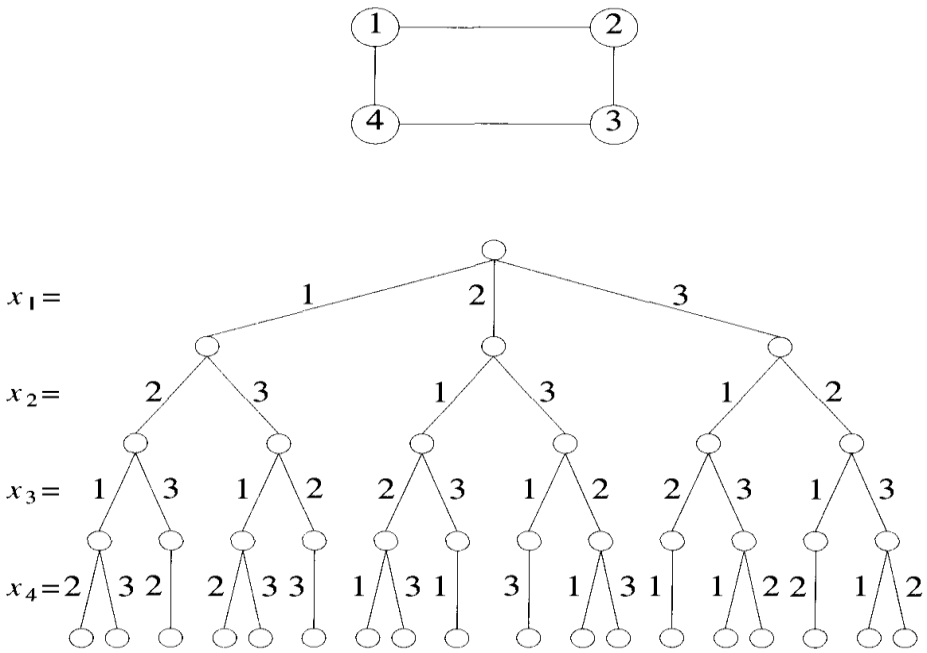



Figure: A node graph and all possible 3-colours

## 5. Branch and bound – General Method

- Branch and Bound is another method to systematically search a solution space.
- Just like backtracking, we will use bounding functions to avoid generating sub trees that do not contain an answer node.

However branch and Bound differs from backtracking in two important manners:

1. It has a branching function, which can be a depth first search, breadth first search or based on bounding function.

2. It has a bounding function, which goes far beyond the feasibility test as a mean to prune efficiently the search tree.

- 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 is the generalization of both graph search strategies, BFS and Dsearch.

- A BFS like state space search is called as FIFO (First in first out) search as the list of live nodes in a first in first out list (or queue).

- A D search like state space search is called as LIFO (Last in first out)• search as the list of live nodes in a last in first out (or stack).
  - ✓ Definition 1: Live node is a node that has been generated but whose children have not yet been generated.
  - ✓ Definition 2: E-node is a live node whose children are currently being explored. In other words, an E-node is a node currently being expanded.
  - ✓ Definition 3: Dead node is a generated node that is not to be expanded or explored any further. All children of a dead node have already been expanded.
  - ✓ Definition 4: Branch-an-bound refers to all state space search methods in which all children of an E-node are generated before any other live node can become the E-node.

## 6. Lease-Cost Search (LC Search)

- In both LIFO and FIFO 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 preference to a node that has a very good chance of getting the search to an answer node quickly.
- The search for an answer node can be speeded by using an "intelligent" ranking ) for live nodes. The next E-node is selected on the basis of this ranking·function c( function. The node x is assigned a rank using:

$$c( x ) = f(h(x)) + g( x )$$

where, c( x ) is the cost of x.

h(x) is the cost of reaching x from the root and f(.) is any non-decreasing function.

g ( x ) is an estimate of the additional effort needed to reach an answer node from x

- A search strategy that uses a cost function c( x ) = f(h(x) + g( x ) to select the next E-node would always choose for its next E-node a live node with least LC–search (Least Cost search).
- BFS and D-search are special cases of LC-search. If g( x ) = 0 and f(h(x)) = level of node x, then an LC search generates nodes by levels. This is eventually the same as a BFS. If f(h(x)) = 0 and essentially a D-search.
- An LC-search coupled with bounding functions is called an LC-branch and bound search We associate a cost c(x) with each node x in the state space tree. It is not possible to easily compute the function c(x). So we compute a estimate c( x ) of c(x).

```
    listnode = record {
        listnode * next, * parent; float cost;
    }

1   Algorithm LCSearch(t)
2   // Search t for an answer node.
3   {
4       if *t is an answer node then output *t and return;
5       E := t; // E-node.
6       Initialize the list of live nodes to be empty;
7       repeat
8       {
9           for each child x of E do
10          {
11              if x is an answer node then output the path
12                  from x to t and return;
13              Add(x); // x is a new live node.
14              (x → parent) := E; // Pointer for path to root.
15          }
16          if there are no more live nodes then
17          {
18              write ("No answer node"); return;
19          }
20          E := Least();
21      } until (false);
22  }
```

Algorithm: LC Search

## 7. 0/1 Knapsack Problem

To use the branch-and-bound technique to solve any problem, it is first necessary to conceive of a state space tree for the problem. We have already seen two possible state space tree organizations for the knapsack problem. Knapsack problem is a maximization problem.

$$\text{minimize } -\sum_{i=1}^{n} p_i x_i$$

$$\text{subject to } \sum_{i=1}^{n} w_i x_i \leq m$$

$$x_i = 0 \text{ or } 1, \quad 1 \leq i \leq n$$

```
1   Algorithm UBound(cp, cw, k, m)
2   // cp, cw, k, and m have the same meanings as in
3   // Algorithm 7.11. w[i] and p[i] are respectively
4   // the weight and profit of the ith object.
5   {
6       b := cp; c := cw;
7       for i := k + 1 to n do
8       {
9           if (c + w[i] ≤ m) then
10          {
11              c := c + w[i]; b := b − p[i];
12          }
13      }
14      return b;
15  }
```

Algorithm: Function u(.) for knapsack problem

**LC Branch-and-Bound Solution**

- Consider the instance: M = 15, n = 4, (P1, P2, P3, P4) = (10, 10, 12, 18) and (w1, w2, w3, w4) = ( 2, 4, 6, 9).
- 0/1 knapsack problem can be solved by using branch and bound technique.
- In this problem we will calculate lower bound and upper bound for each node. Place first item in knapsack. Remaining weight of knapsack is 15 – 2 = 13.
- Place next item w2 in knapsack and the remaining weight of knapsack is 13 – 4 = 9.
- Place next item w3 in knapsack then the remaining weight of knapsack is 9 – 6 = 3.
- No fractions are allowed in calculation of upper bound so w4 cannot be placed in knapsack. Profit = P1 + P2 + P3 = 10 + 10 + 12 So,
  Upper bound = 32.
  To calculate lower bound we can place w4 in knapsack since fractions are allowed in calculation of lower bound. Lower bound = 10 + 10 + 12 + ( 3/9 X 18) = 32 + 6 = 38.
- Knapsack problem is maximization problem but branch and bound technique is applicable for only minimization problems. In order to convert maximization problem into minimization problem we have to take negative sign for upper bound and lower bound.

  Therefore, Upper bound (U) = -32
  Lower bound (L) = -38

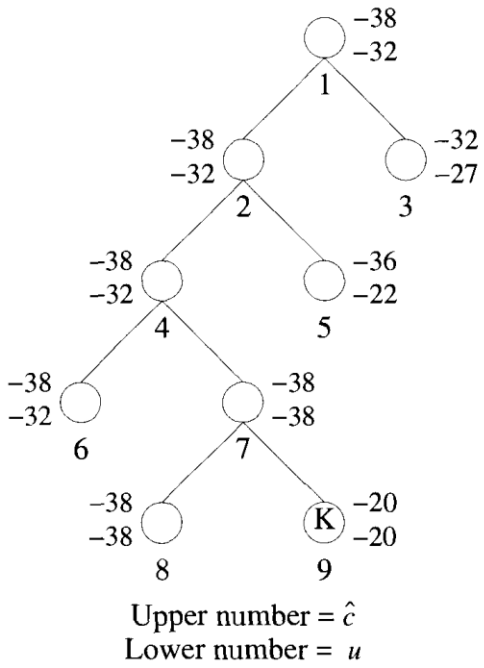



Figure: LC Branch-and-Bound Tree

## 8. Travelling Sales Person Problem

- By using dynamic programming algorithm we can solve the problem with time complexity of $O(n^2 2^n)$ for worst case.
- This can be solved by branch and bound technique using efficient bounding function.
- The time complexity of traveling sale person problem using LC branch and bound is $O(n^2 2^n)$ which shows that there is no change or reduction of complexity than previous method.
- We start at a particular node and visit all nodes exactly once and come back to initial node with minimum cost.

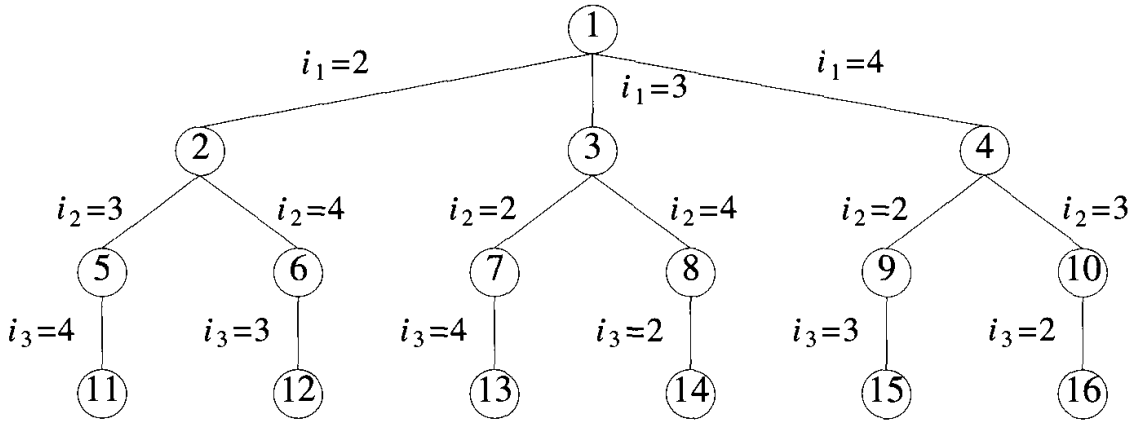



Figure: State space tree for travelling sales person problem

Example:

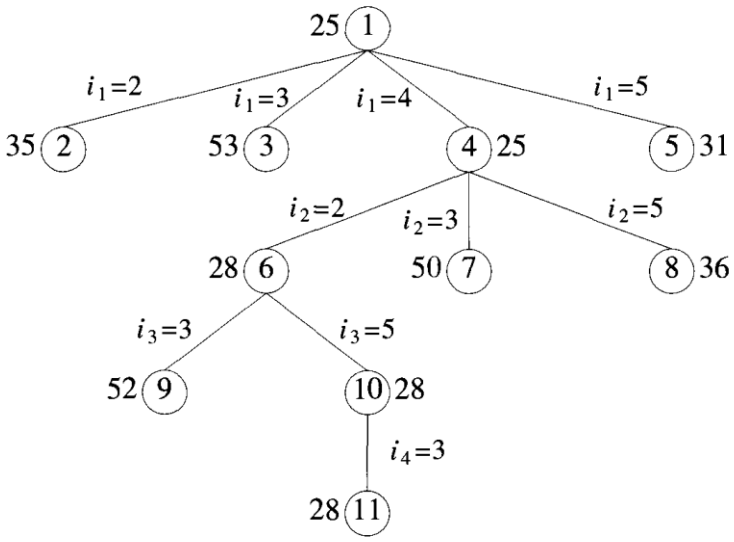
$$\begin{bmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{bmatrix}$$

(a) Cost matrix

$$\begin{bmatrix} \infty & 10 & 17 & 0 & 1 \\ 12 & \infty & 11 & 2 & 0 \\ 0 & 3 & \infty & 0 & 2 \\ 15 & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & 12 & \infty \end{bmatrix}$$

(b) Reduced cost matrix

L = 25




Numbers outside the node are $\hat{c}$ values

Figure: State space tree generated by procedure LCBB

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & 2 & 0 \\ 0 & \infty & \infty & 0 & 2 \\ 15 & \infty & 12 & \infty & 0 \\ 11 & \infty & 0 & 12 & \infty \end{bmatrix}$$

(a) Path 1,2; node 2

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & 2 & 0 \\ \infty & 3 & \infty & 0 & 2 \\ 4 & 3 & \infty & \infty & 0 \\ 0 & 0 & \infty & 12 & \infty \end{bmatrix}$$

(b) Path 1,3; node 3

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ \infty & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & \infty & \infty \end{bmatrix}$$

(c) Path 1,4; node 4

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 10 & \infty & 9 & 0 & \infty \\ 0 & 3 & \infty & 0 & \infty \\ 12 & 0 & 9 & \infty & \infty \\ \infty & 0 & 0 & 12 & \infty \end{bmatrix}$$

(d) Path 1,5; node 5

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & \infty & 0 \\ 0 & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \end{bmatrix}$$

(e) Path 1,4,2; node 6

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & \infty & 0 \\ \infty & 1 & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & 0 & \infty & \infty & \infty \end{bmatrix}$$

(f) Path 1,4,3; node 7

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & 0 & \infty & \infty \\ 0 & 3 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & 0 & \infty & \infty \end{bmatrix}$$

(g) Path 1,4,5; node 8

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \end{bmatrix}$$

(h) Path 1,4,2,3; node 9

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \end{bmatrix}$$

(i) Path 1,4,2,5; node 10

Figure: Reduced cost matrices corresponding to nodes