Unit III- Backtracking, Branch & Bound

Amrita Naik

BackTracking

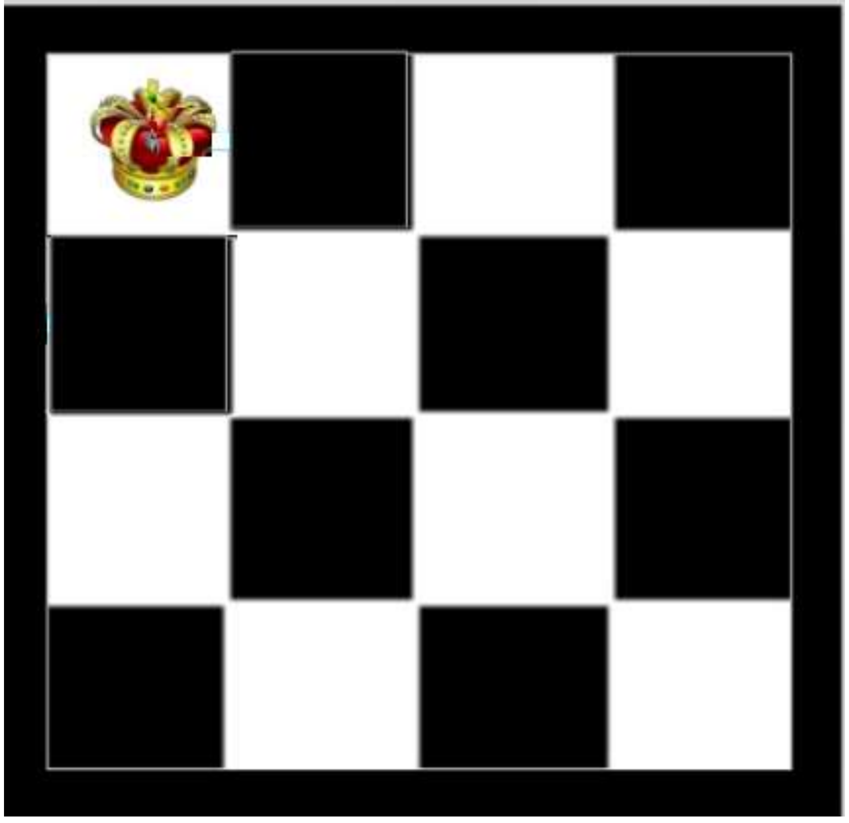
- Backtracking Algo has to satisfy the following set of rules:
- 1.Explicit constraint: Restrict elements to have values from a given set.
- 2.Implicit constraint: Rules specific to problem.

Algorithm

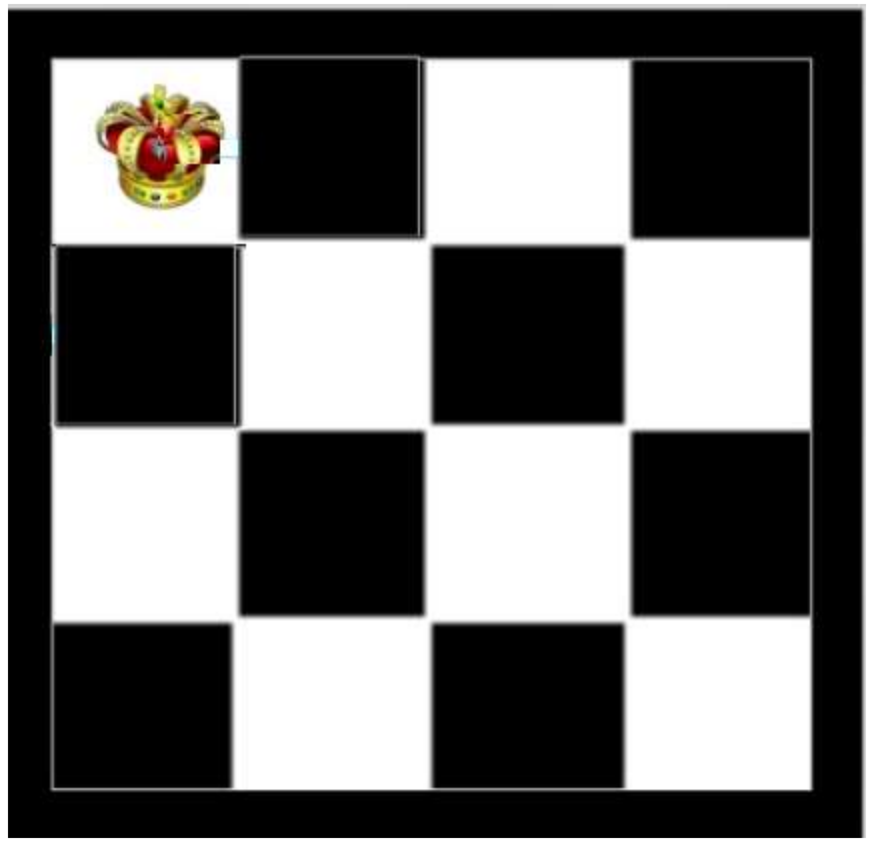
```
Algorithm Backtrack(k)
// This schema describes the backtracking process using
// recursion. On entering, the first k − 1 values
// x[1], x[2], ..., x[k − 1] of the solution vector
// x[1 : n] have been assigned. x[ ] and n are global.
{
    for (each x[k] ∈ T(x[1], ..., x[k − 1]) do
    {
        if (B_k(x[1], x[2], ..., x[k]) ≠ 0) then
        {
            if (x[1], x[2], ..., x[k] is a path to an answer node)
                then write (x[1 : k]);
            if (k < n) then Backtrack(k + 1);
        }
    }
}
```

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.

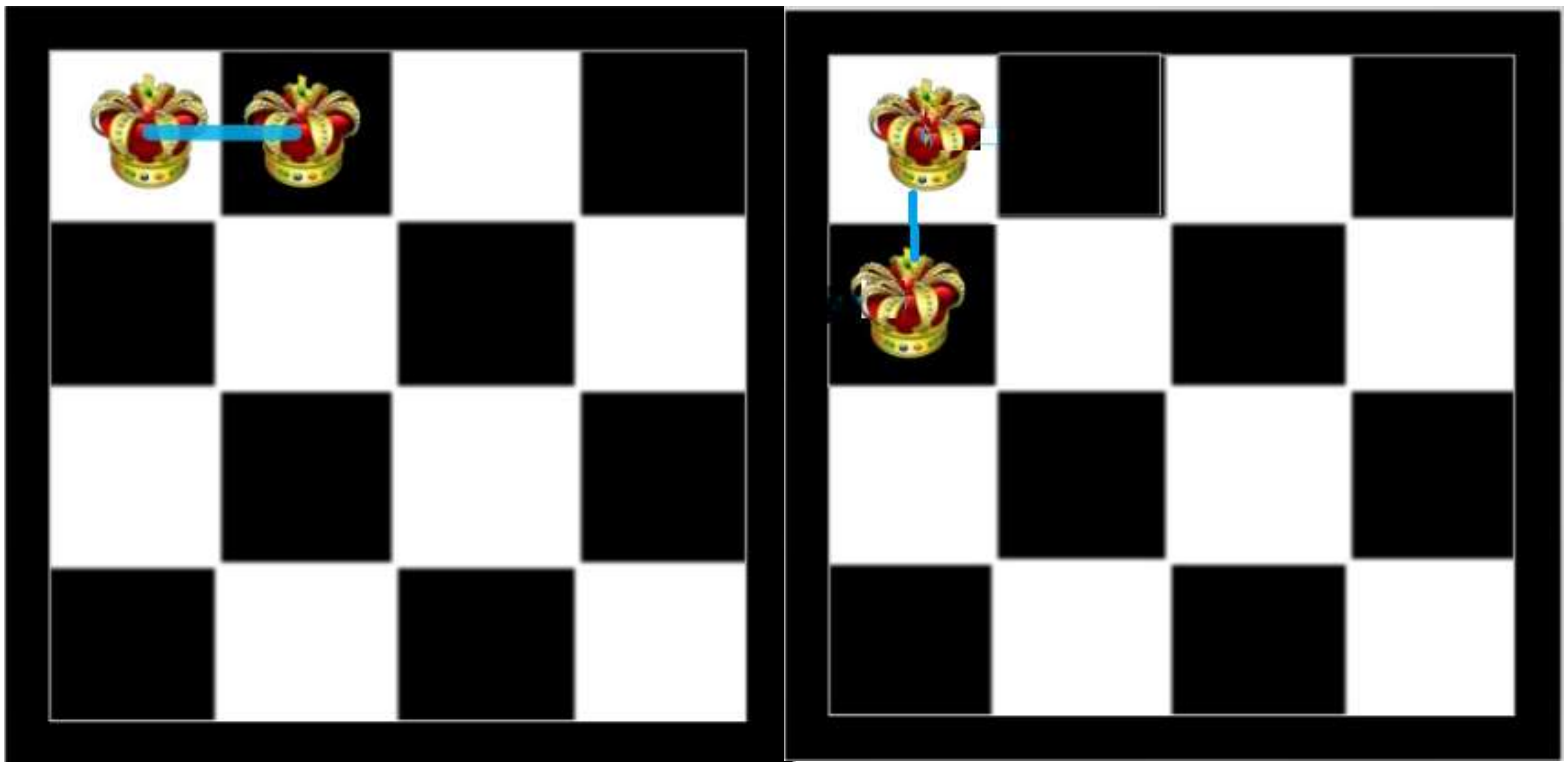


Explicit Constraint

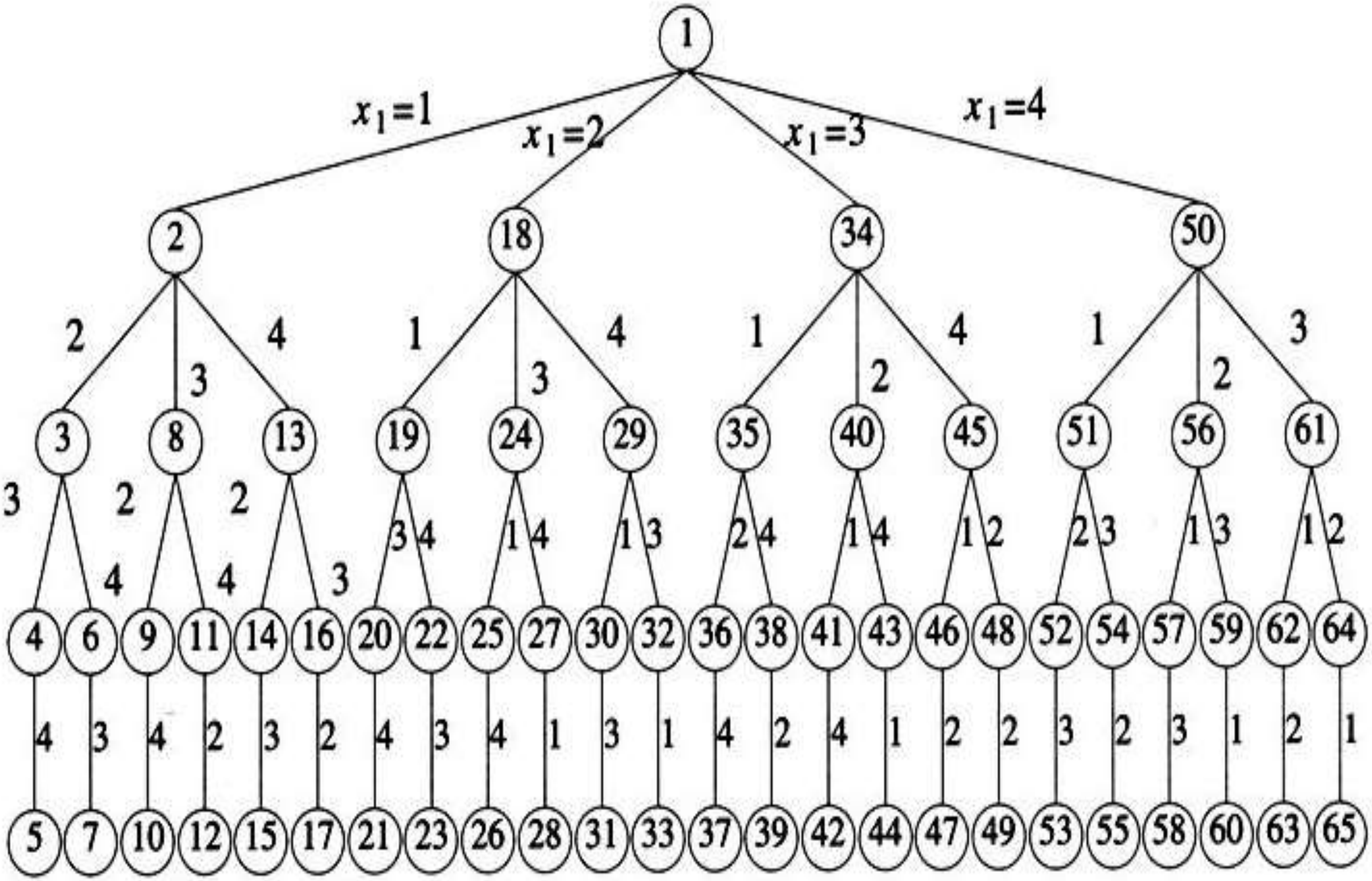


Xi can have values {1 2 3 4}

Implicit Constraint

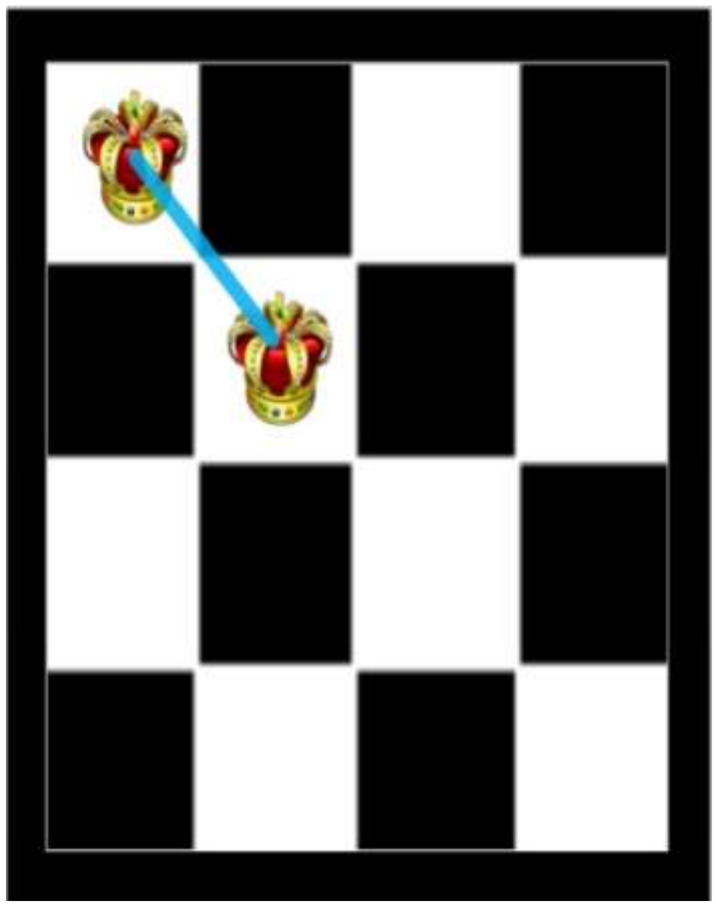


State Space Tree

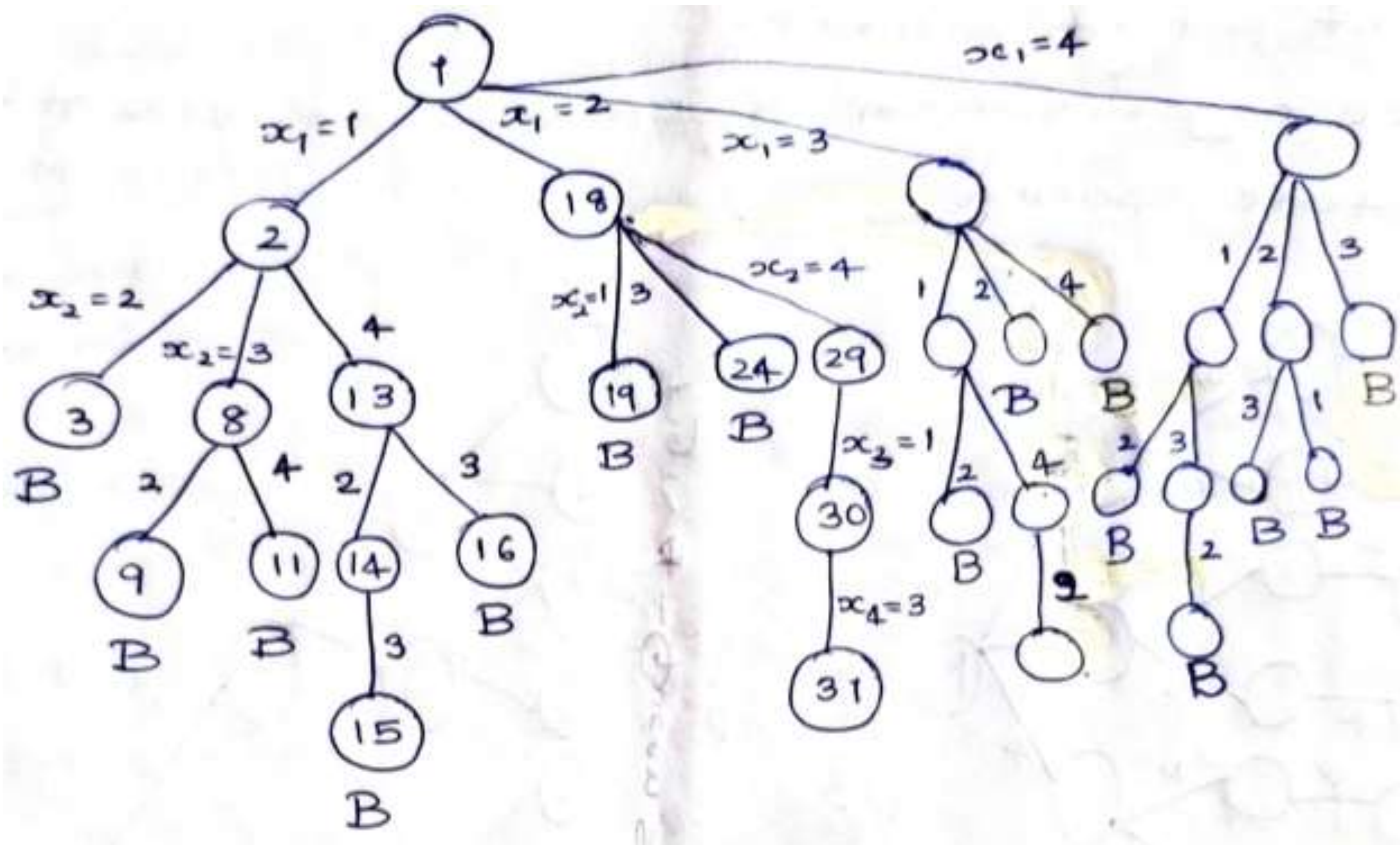


Bounding Function

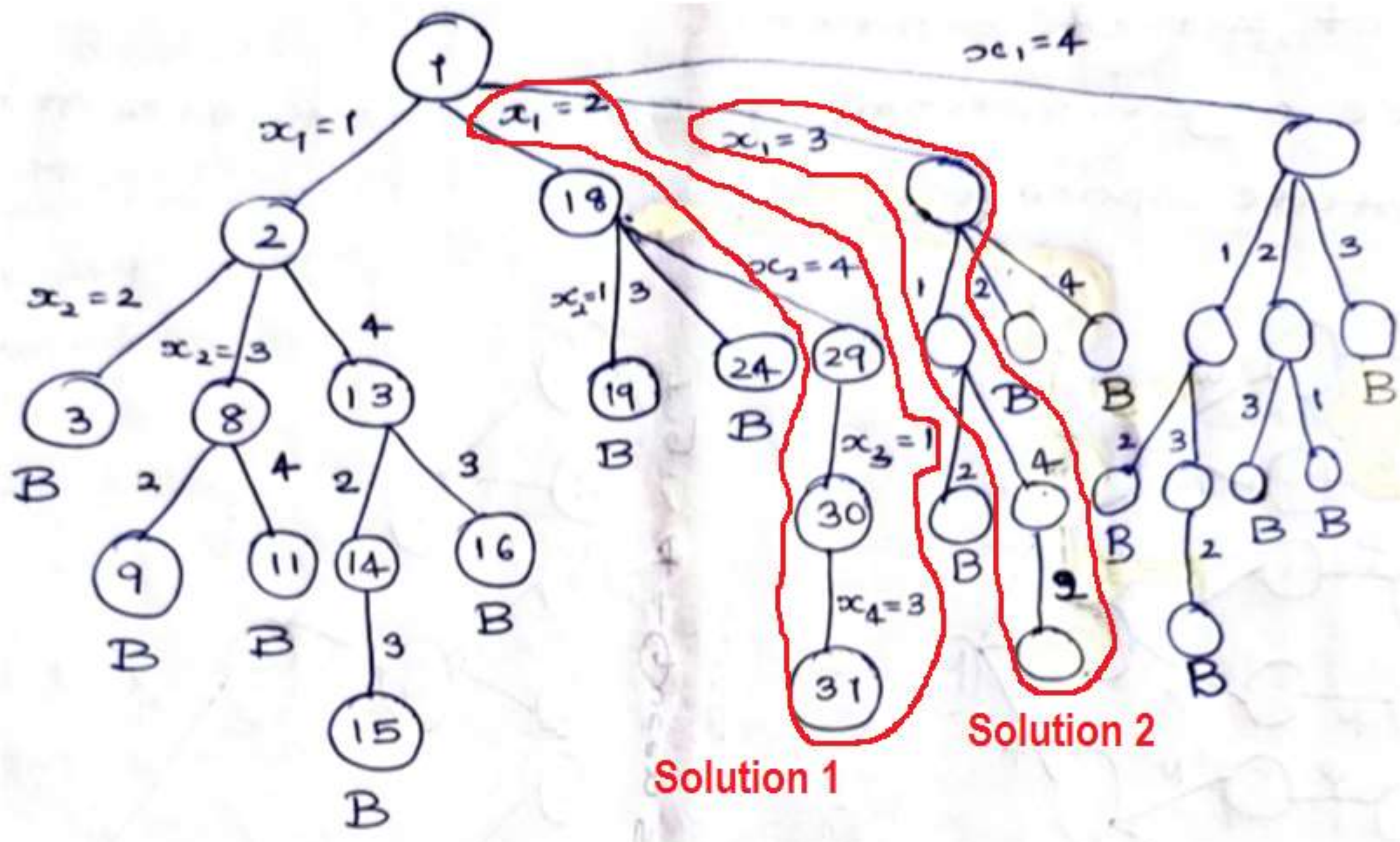
- Queen should not be diagonally opposite to each other.



4-Queen Solution



2 Solution to 4-Queens Problem



N-Queens Algorithm

```
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 is invoked by Nqueens(1,N)

N-Queens Algorithm

```
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;
}
```

Time Complexity of N-Queens

- Time Complexity=$T(n)=n*T(n-1)+n^2$
- $=O(n!)=O(n^n)$
- Brute force approach=$O({}^{t}C_{n})$
- Where n- is no of rows and cols
- t-no of cells on the board

Sum of Subset Problem

- X={11,13,24,7}   ,M=31

- Solution:
- (11,13,7)
- (24,7)

Sum of Subset Problem

1. For n=6, m=30, w[1:6]={5,10,12,13,15,18}. Find all possible subsets of w that sum to m.

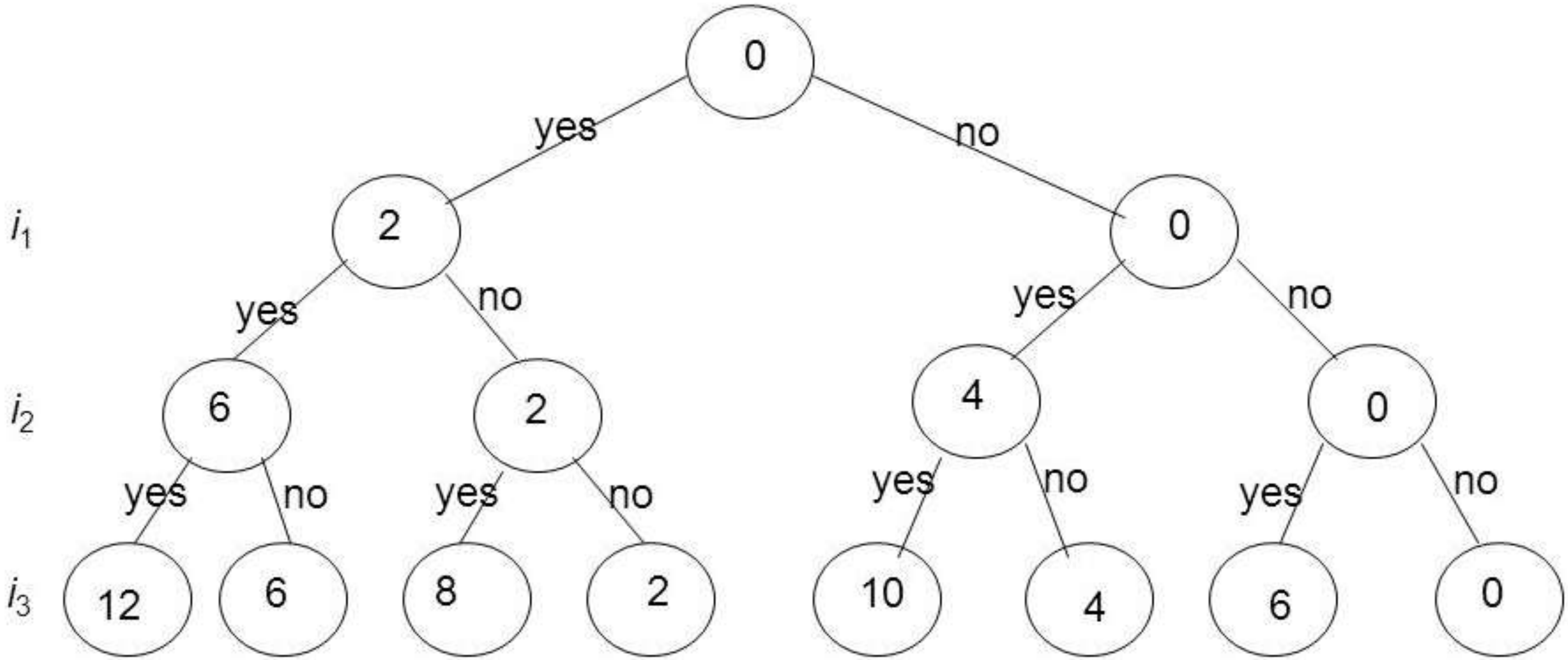
Sum Of Subset Problem

- **Explicit constraint** : If the set contains 'n' elements then Xi should be an integer and have values {from 1>=i<=n}
- **Implicit Constraint** :The sum of value[xi] should be equal to 'm'

X={2,4,6}

Sum of subset Problem:
State SpaceTree for 3 items

$w_1 = 2$, $w_2 = 4$, $w_3 = 6$ and m $= 6$



The sum of the included integers is stored at the node.

Bounding Function

Function Sumofsubset(s,k,r)

- 1.If (s+w[k])=m print all xi values in set
- 2.If (s+w[k]+w[k+1]<=m) ->Navigate to the Left ,

call sumofsubset(s+w[k],k+1,r-w[k])

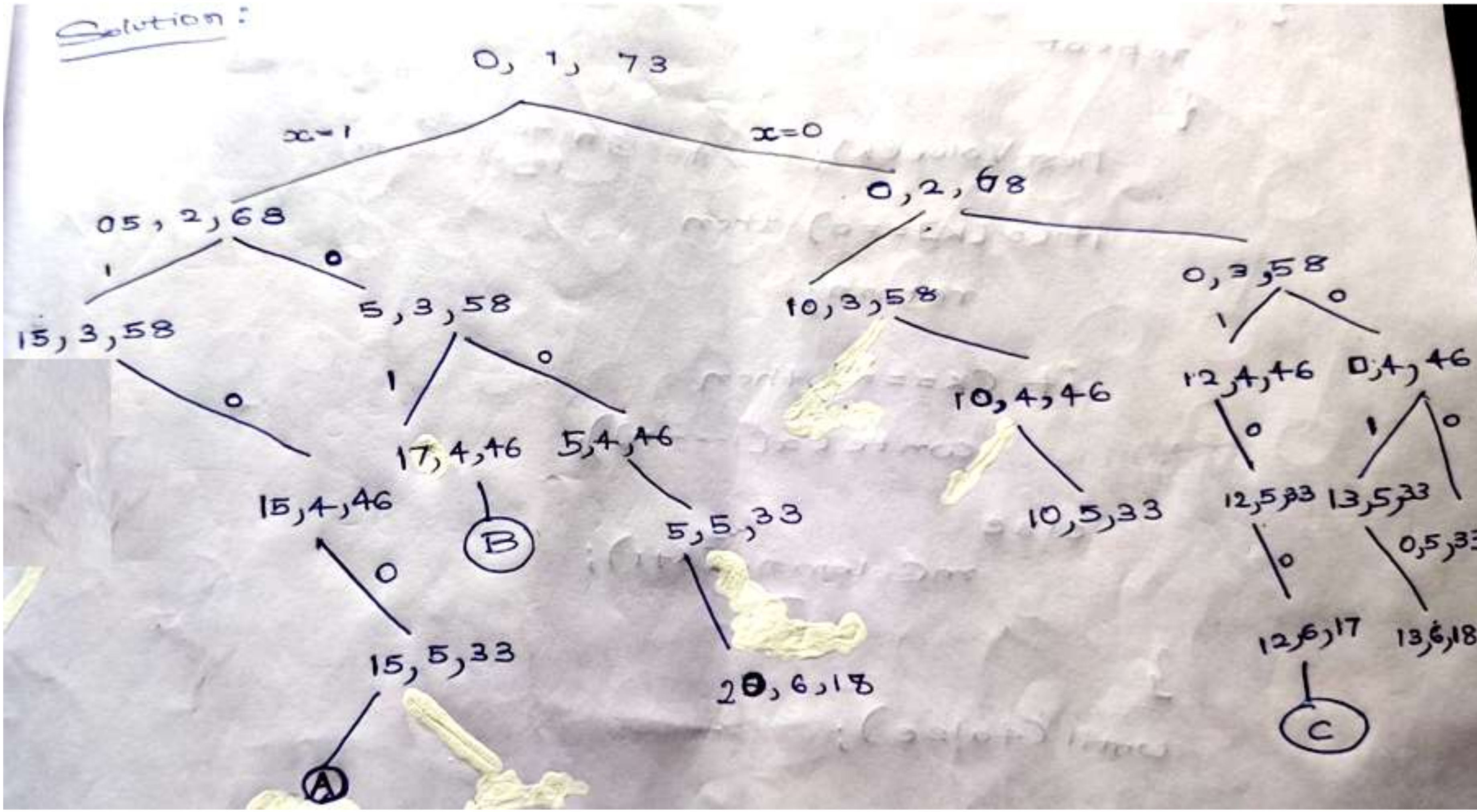
- 3.If (s+r-w[k]>=m) and (s+w[k+1]<=m)-> Navigate to the Right,

call sumofsubset(s,k+1,r-w[k])

W[1]	W[2]	W[3]	W[4]	W[5]	W[6]
5	10	12	13	15	18

m=30

(s,k,r) where
s=sum of values in subset
K-no of levels
r-capacity of set



SumofSubset Algorithm

Algorithm SumOfSub(s, k, r)
// Find all subsets of $w[1:n]$ that sum to m. The values of $x[j]$,
// $1 \le j < k$, have already been determined. $s = \sum_{j=1}^{k-1} w[j] * x[j]$
// and $r = \sum_{j=k}^{n} w[j]$. The $w[j]$'s are in nondecreasing order.
// It is assumed that $w[1] \le m$ and $\sum_{i=1}^{n} w[i] \ge m$.
{
    // Generate left child. Note: $s + w[k] \le m$ since B_{k-1} is true.
    $x[k] := 1;$
    if $(s + w[k] = m)$ **then write** $(x[1:k])$; // Subset found
        // There is no recursive call here as $w[j] > 0$, $1 \le j \le n$.
    else if $(s + w[k] + w[k+1] \le m)$
        then SumOfSub$(s + w[k], k+1, r - w[k])$;
    // Generate right child and evaluate B_k.
    if $((s + r - w[k] \ge m)$ **and** $(s + w[k+1] \le m))$ **then**
    {
        $x[k] := 0;$
        SumOfSub$(s, k+1, r - w[k])$;
    }
}

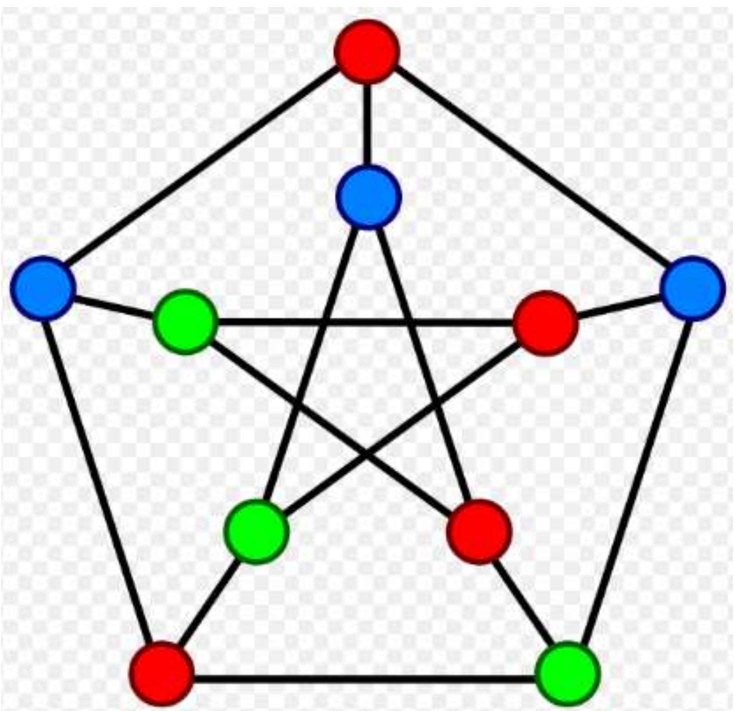
Algorithm starts with SumOfSub(0,1,r)

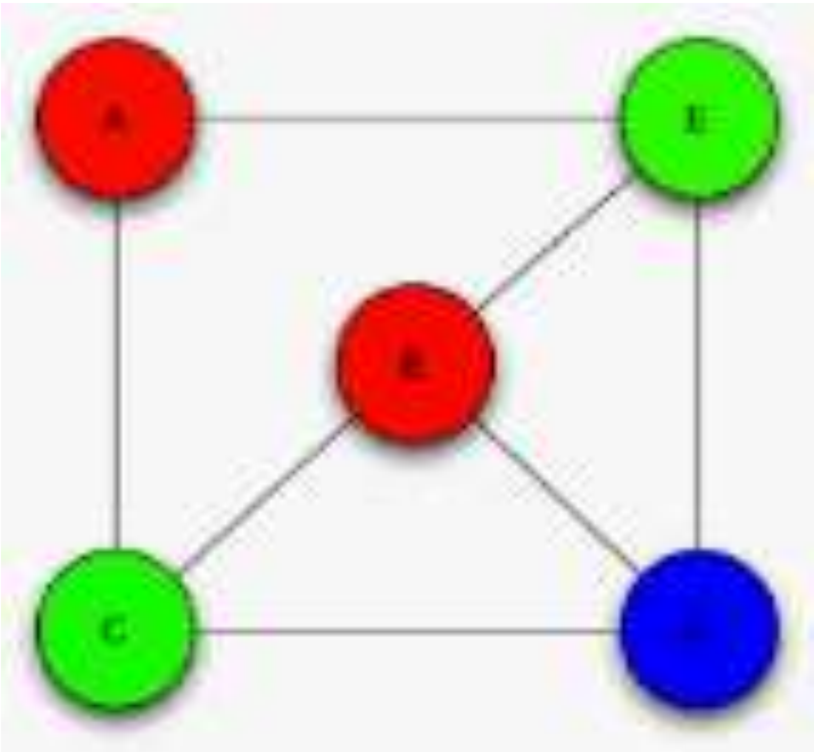
Time complexity of Sum of Subset

- $T(n)=O(2^{n})$

Graph Coloring Problem

Assignment of colors to the vertices or edges such that no two adjacent vertices are to be similarly colored.





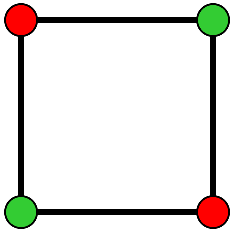
Graph Coloring Problem

- We want to minimize the number of colors used.
- Let G be undirected graph and let c be an integer that represent minimum number of colors used to color the vertices.
- The smallest c such that a c-coloring exists is called the graph's chromatic number and any such c-coloring is an optimal coloring.

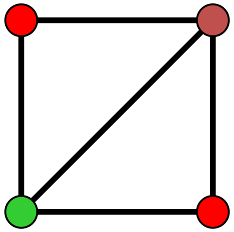
Coloring of Graph

1. The graph coloring optimization problem: find the minimum number of colors needed to color the vertices of a graph.

2. The graph coloring decision problem: determine if there exists a coloring for a given graph which uses at most m colors.



Two colors



No solution with two colors

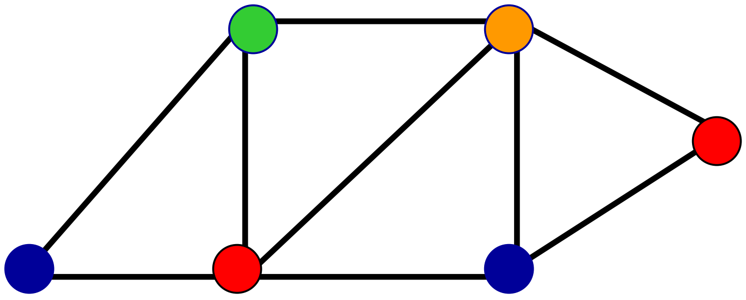
An Application-Map Coloring

Coloring of Graphs

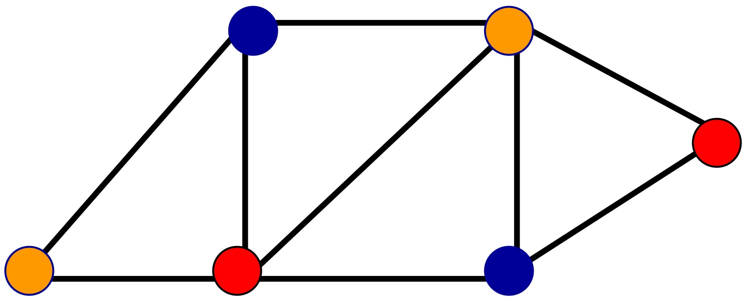
Practical applications: scheduling, time-tabling, register allocation for compilers, coloring of maps.

<u>A simple graph coloring algorithm</u> - choose a color and an arbitrary starting vertex and color all the vertices that can be colored with that color.

Choose next starting vertex and next color and repeat the coloring until all the vertices are colored.

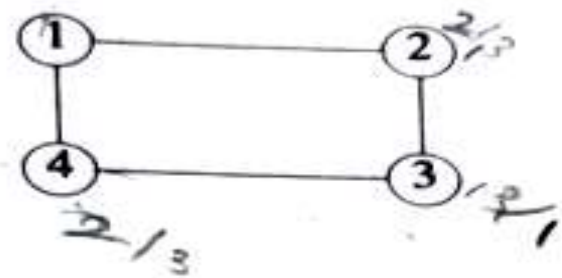


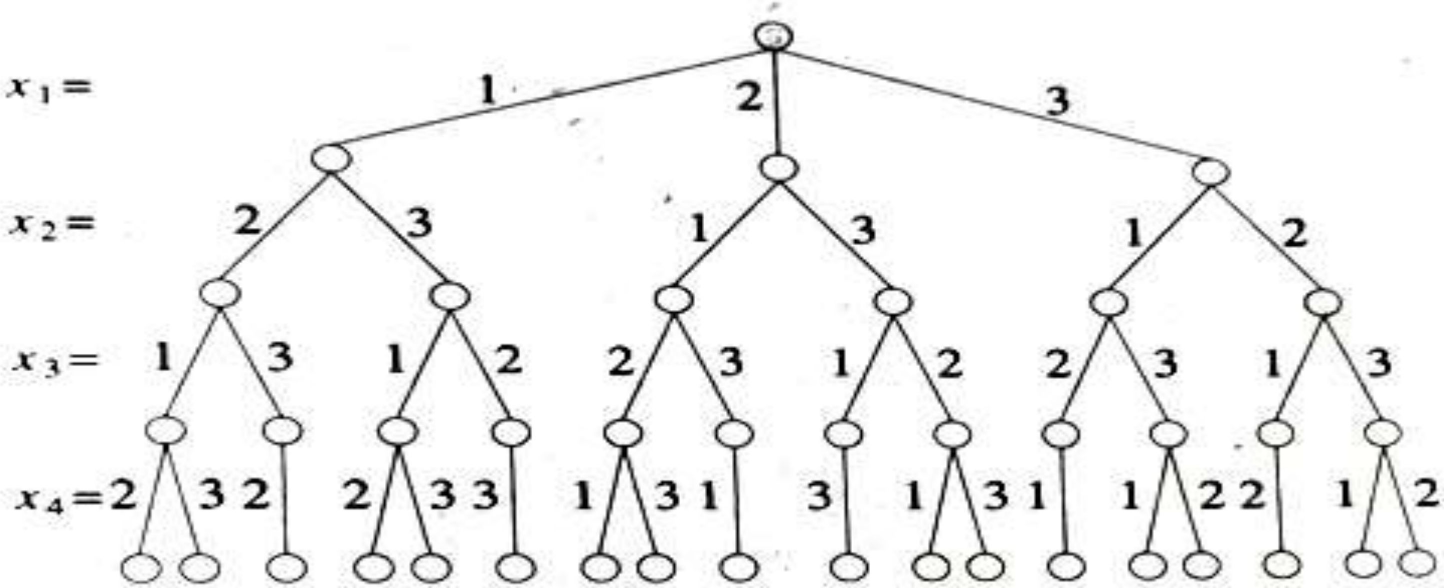
Four colors



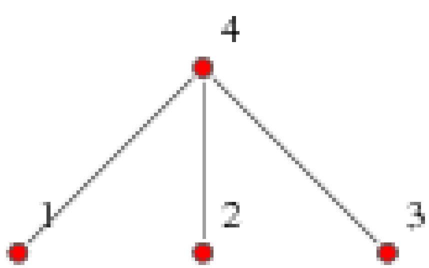
Three colors are enough

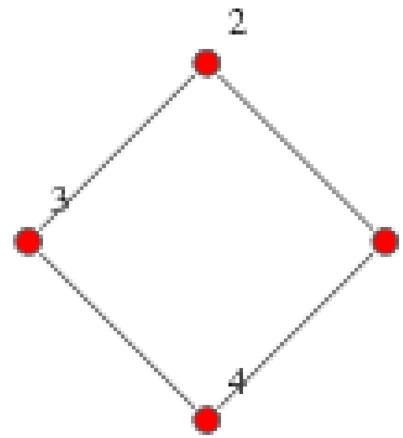
4 Node 3 coloring

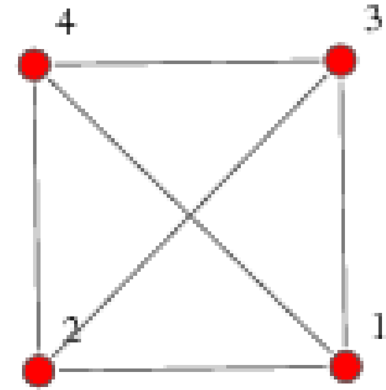




Adjacency Matrix







$$\begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix} \qquad \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix} \qquad \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

Backtracking Algorithm

```
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);
}
```

mColoring(1) is called initially

Backtracking Algorithm

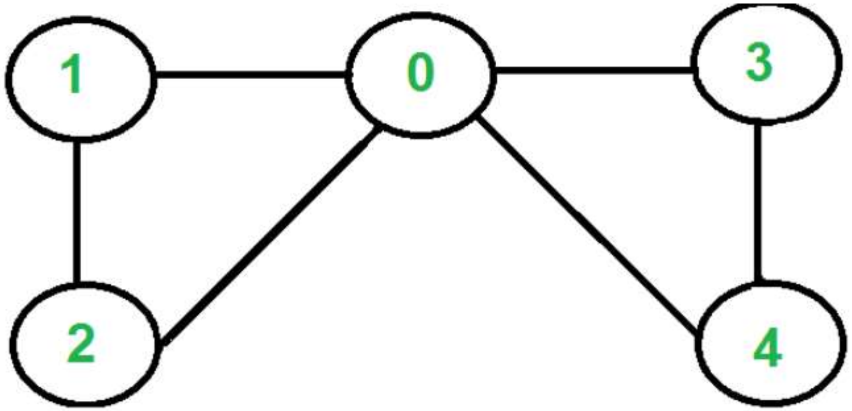
```
Algorithm NextValue(k)
// x[1],...,x[k-1] have been assigned integer values in
// the range [1,m] such that adjacent vertices have distinct
// integers. A value for x[k] is determined in the range
// [0,m]. x[k] is assigned the next highest numbered color
// while maintaining distinctness from the adjacent vertices
// of vertex k. If no such color exists, then x[k] is 0.
{
    repeat
    {
        x[k] := (++x[k]) mod (m + 1); // Next highest color.
        if (x[k] = 0) then return; // All colors have been use
        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.
}
```

Time complexity of graph coloring

- $T(n)=O(nm^{n})$
  - Where n is no of vertices
  - m is no of colors

Hamiltonian Path

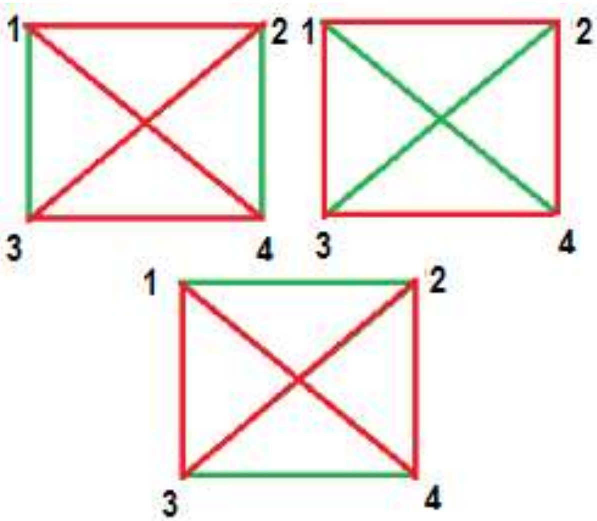
- A Hamiltonian path is a path in an undirected or directed graph that visits each vertex exactly once.



Hamiltonian Path => 2, 1, 0, 3, 4

Hamiltonian Cycle

- A Hamiltonian cycle (or Hamiltonian circuit) is a Hamiltonian path that is a cycle.
- A Hamiltonian cycle is a cycle that visits each vertex exactly once (except for the vertex that is both the start and end, which is visited twice).
- A graph that contains a Hamiltonian cycle is called a Hamiltonian graph.

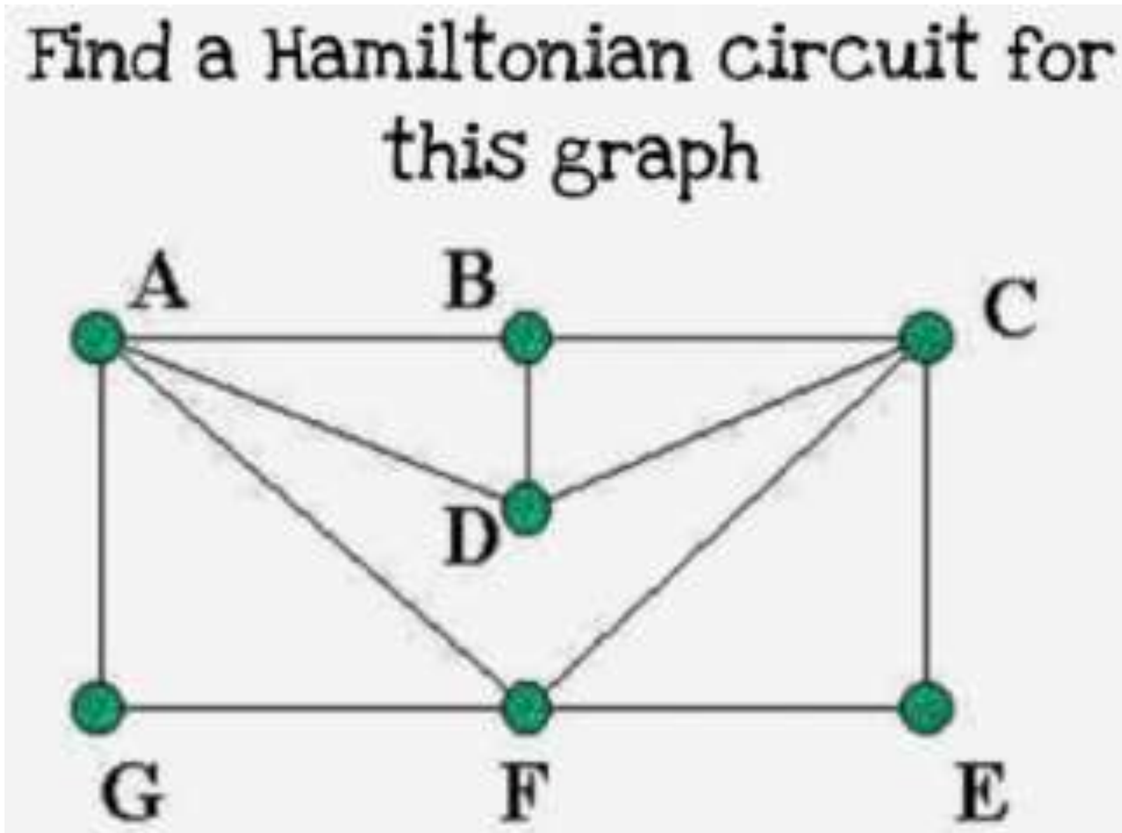


Hamiltonian Cycle=> 1,2,3,4,1

=>1,2,3,4,1

=>1,4,2,3,1

Hamiltonian Cycle: An Example



Hamiltonian Cycle Example

For example consider the given graph
and evaluate the mechanism:-

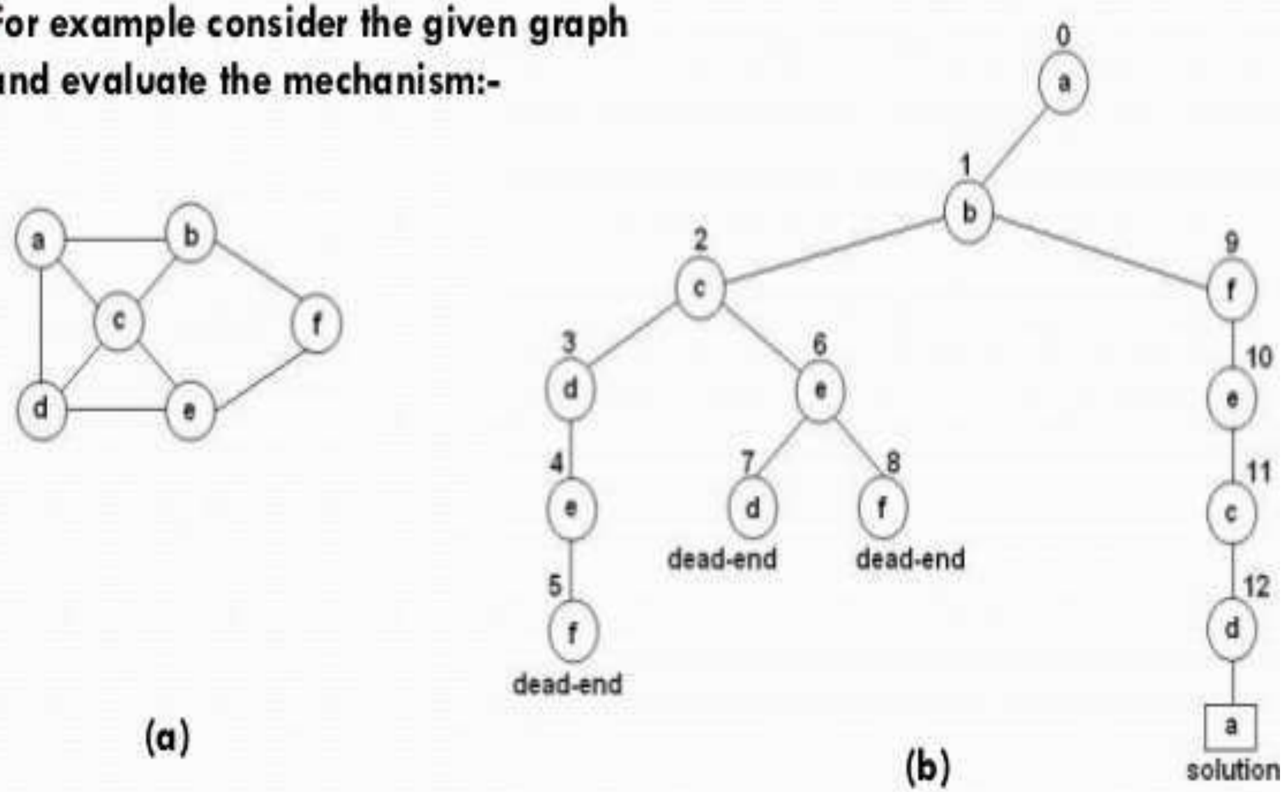
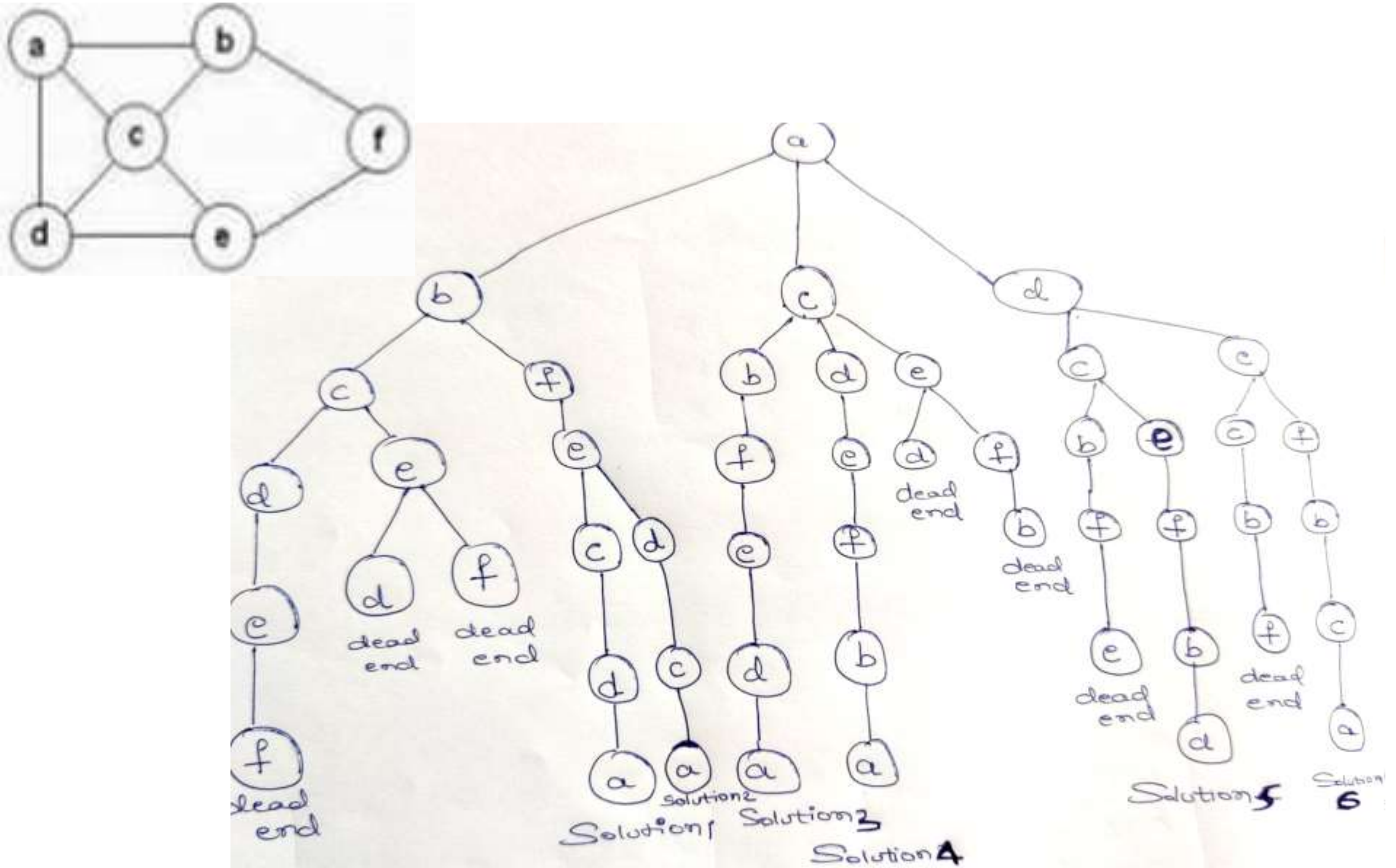


Figure: • (a) Graph.
• (b) State-space tree for finding a Hamiltonian circuit. The numbers above the nodes of the tree indicate the order the order in which nodes are generated.

Hamiltonian Cycle Example



Hamiltonian Cycle-Algorithm

```
Algorithm Hamiltonian(k)
// This algorithm uses the recursive formulation of
// backtracking to find all the Hamiltonian cycles
// of a graph. The graph is stored as an adjacency
// matrix G[1 : n, 1 : n]. All cycles begin at node 1.
{
    repeat
    { // Generate values for x[k].
        NextValue(k); // Assign a legal next value to x[k].
        if (x[k] = 0) then return;
        if (k = n) then write (x[1 : n]);
        else Hamiltonian(k + 1);
    } until (false);
}
```

Hamiltonian(1) is called initially

Hamiltonian Cycle-Algorithm

```
Algorithm NextValue(k)
// x[1 : k − 1] is a path of k − 1 distinct vertices. If x[k] = 0, then
// no vertex has as yet been assigned to x[k]. After execution,
// x[k] is assigned to the next highest numbered vertex which
// does not already appear in x[1 : k − 1] and is connected by
// an edge to x[k − 1]. Otherwise x[k] = 0. If k = n, then
// in addition x[k] is connected to x[1].
{
    repeat
    {
        x[k] := (++x[k]) mod (n + 1); // Next vertex.
        if (x[k] = 0) then return;
        if (G[x[k − 1], x[k]] ≠ 0) then
        { // Is there an edge?
            for j := 1 to k − 1 do if (x[j] = x[k]) then break;
                        // Check for distinctness.
            if (j = k) then // If true, then the vertex is distinct.
                if ((k < n) or ((k = n) and G[x[n], x[1]] ≠ 0))
                    then return;
        }
    } until (false);
}
```

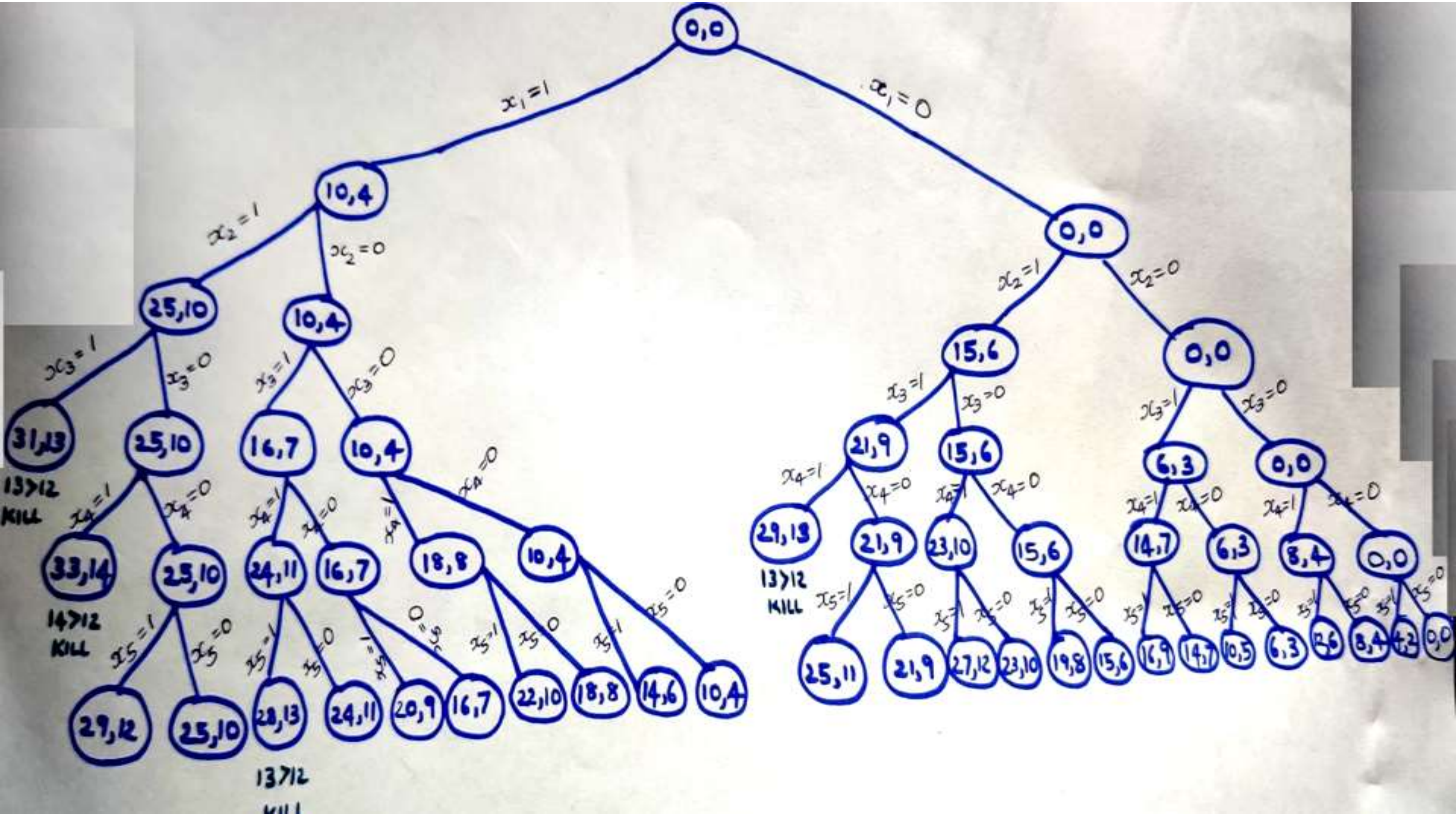
Time complexity of Hamiltonian Graph

- $T(n)=O(n!)=O(n^n)$

0/1 Knapsack Example

- 1.Solve 0/1 knapsack problem using backtracking where n=5 and p={10,15,6,8,4} and w={4,6,3,4,2} and capacity=12.
- Solution: Place elements in increasing order of p/w
  - X1=10/4=2.5
  - X2=15/6=2.5
  - X3=6/3=2
  - X4=8/4=2
  - X5=4/2=2

0/1 Knapsack Example



p={10,15,6,8,4} and w={4,6,3,4,2} and capacity=12

0/1 Knapsack using Backtracking

```
Algorithm BKnap(k, cp, cw)
// m is the size of the knapsack; n is the number of weights
// and profits. w[ ] and p[ ] are the weights and profits.
// p[i]/w[i] ≥ p[i + 1]/w[i + 1]. fw is the final weight of
// knapsack; fp is the final maximum profit. x[k] = 0 if w[k]
// is not in the knapsack; else x[k] = 1.
{
    // Generate left child.
    if (cw + w[k] ≤ m) then
    {
        y[k] := 1;
        if (k < n) then BKnap(k + 1, cp + p[k], cw + w[k]);
        if ((cp + p[k] > fp) and (k = n)) then
        {
            fp := cp + p[k]; fw := cw + w[k];
            for j := 1 to k do x[j] := y[j];
        }
    }
    // Generate right child.
    if (Bound(cp, cw, k) ≥ fp) then
    {
        y[k] := 0; if (k < n) then BKnap(k + 1, cp, cw);
        if ((cp > fp) and (k = n)) then
        {
            fp := cp; fw := cw;
            for j := 1 to k do x[j] := y[j];
        }
    }
}
```

Algorithm is called as Bknap(1,0,0)

0/1 Knapsack using Backtracking

```
Algorithm Bound(cp, cw, k)
// cp is the current profit total, cw is the current
// weight total; k is the index of the last removed
// item; and m is the knapsack size.
{
    b := cp; c := cw;
    for i := k + 1 to n do
    {
        c := c + w[i];
        if (c < m) then b := b + p[i];
        else return b + (1 - (c - m)/w[i]) * p[i];
    }
    return b;
}
```

Time Complexity for Knapsack

- $T(n)=O(2^{n})$