Unit-V

UNIT V: Backtracking: General method, applications-n-queen problem, sum of subsets problem, graph coloring, Hamiltonian cycles. Branch and Bound: General method, applications - Travelling sales person problem,0/1 knapsack problem- LC Branch and Bound solution, FIFO Branch and Bound solution.

Applications: Undo in MS-Word, Games

Difficult Problems

- Partition
  - Partition n positive integers $s_1, s_2, s_3, \ldots, s_n$ into two groups A and B such that the sum of the numbers in each group is the same.
  - [9, 4, 6, 3, 5, 1,8]
  - A = [9, 4, 5] and B = [6, 3, 1, 8]

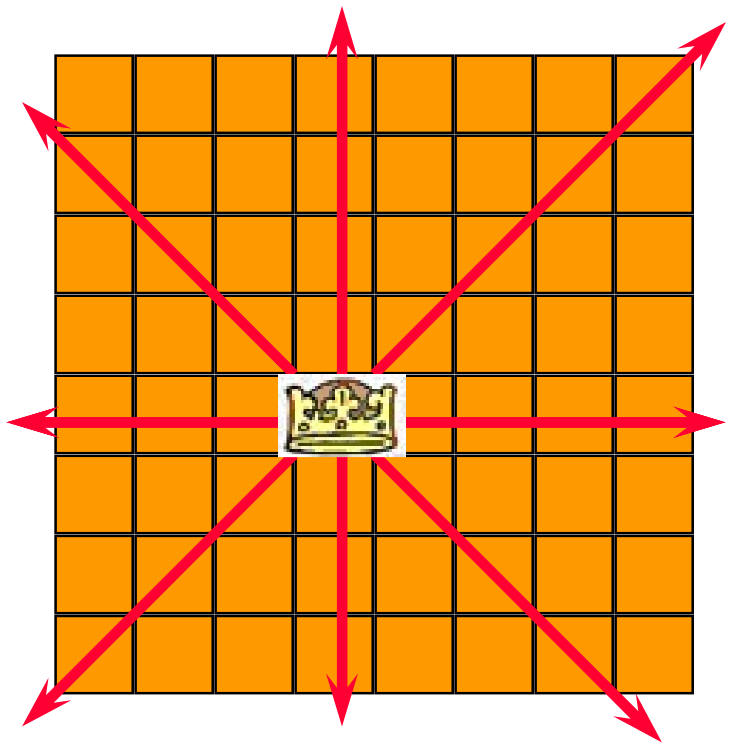
- Subset Sum Problem
  - Does any subset of n positive integers s_1, s_2, s_3, …, s_n have a sum exactly equal to c?
  - [9, 4, 6, 3, 5, 1,8] and $c = 18$
  - $A = [9, 4, 5]$

Traveling Salesperson Problem (TSP)

- Let G be a weighted directed graph.
- A tour in G is a cycle that includes every vertex of the graph.
- TSP => Find a tour of shortest length.

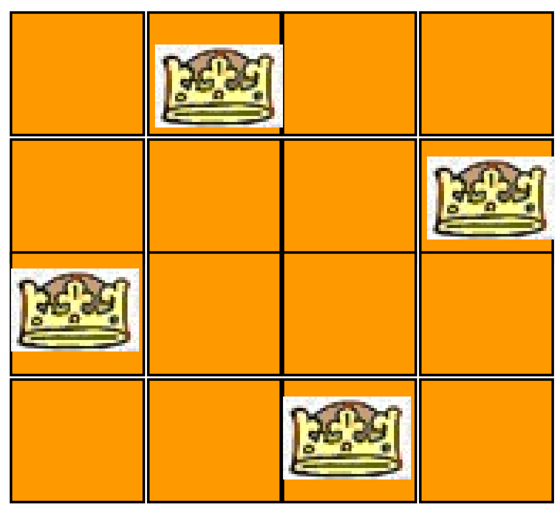
n-Queens Problem

A queen that is placed on an n x n chessboard, may attack with any other queen placed in the same column, row, or diagonal.

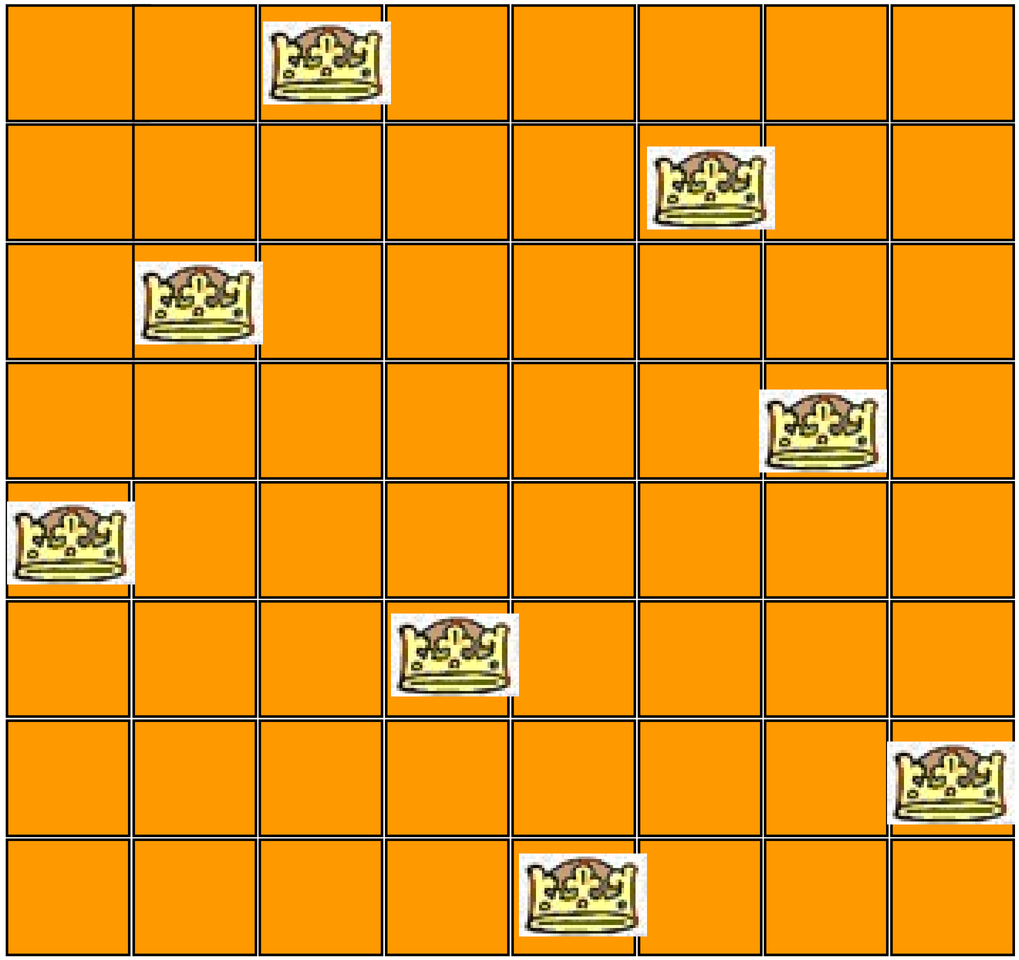


8x8 Chessboard

Can n queens be placed on an n x n chessboard so that no queen attacks another queen?



4x4



8x8

Difficult Problems

- Many difficult problems require you to find either a subset or permutation that satisfies some constraints and (possibly also) optimizes some objective function.
- These may be solved by organizing the solution space into a tree and systematically searching this tree for the answer.

Solution Space

- Solution Space is a set that includes at least one solution to the problem.
- Subset problem.
  - $n = 2$, {00, 01, 10, 11}
  - $n = 3$, {000, 001, 010, 100, 011, 101, 110, 111}
- Solution space for subset problem has 2^n members.
- Non systematic search of the space for the answer takes $O(2^n)$ time.

Solution Space

- Permutation problem.
  - n = 2, {12, 21}
  - n = 3, {123, 132, 213, 231, 312, 321}
- Solution space for a permutation problem has n! members.
- Non systematic search of the space for the answer takes O(n!) time.

Backtracking and Branch and Bound

- *Backtracking* and *branch and bound* perform a systematic search and take much less time than the time taken by a *non systematic* search.

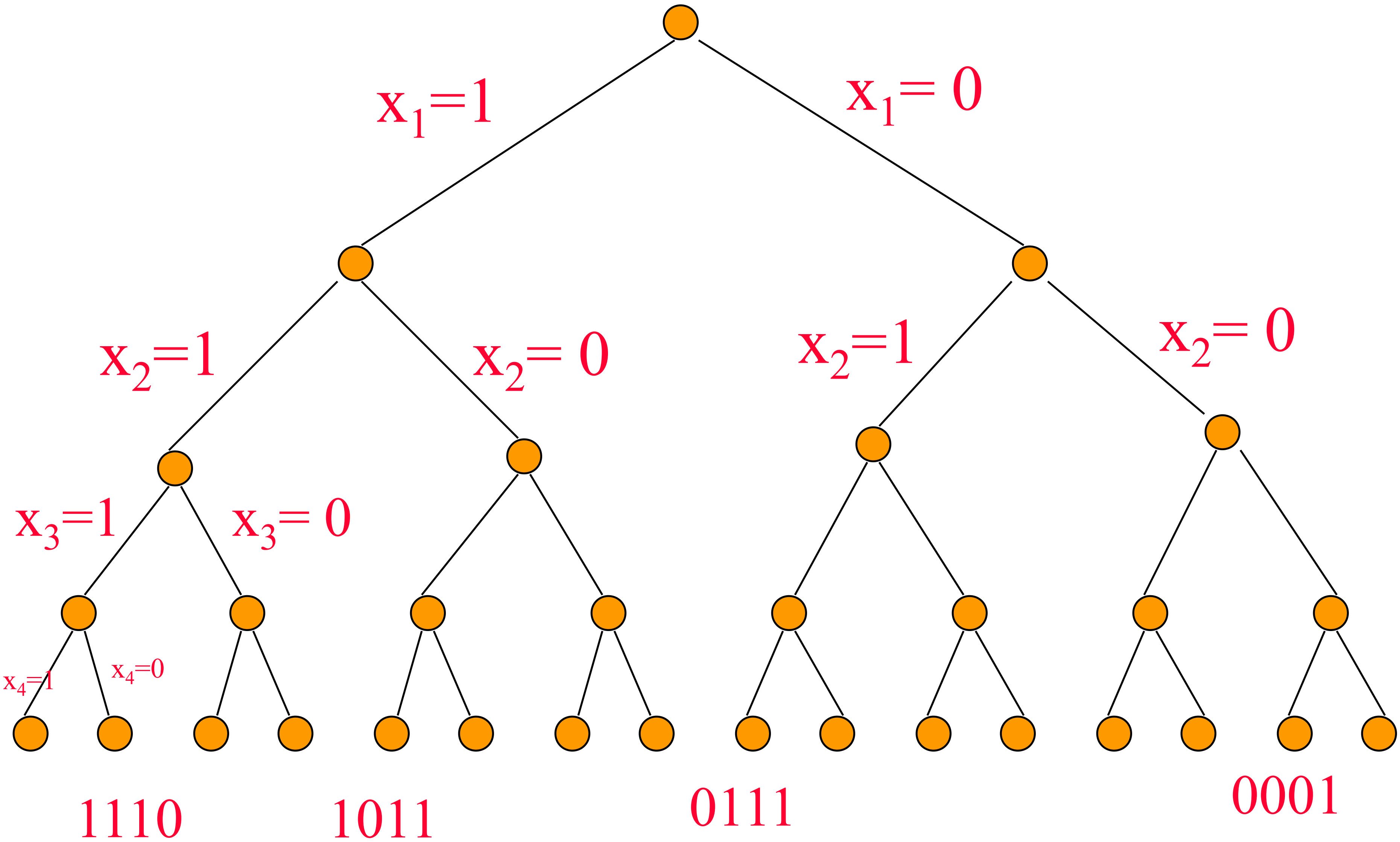
Tree Organization Of Solution Space

- Set up a tree structure such that the paths from root to leaves represent members of the solution space.
- For a size n subset problem, this tree structure has 2^n leaves.
- For a size n permutation problem, this tree structure has n! leaves.
- The tree structure is too big to store in memory; it also takes much time to create the tree structure.
- Portions of the tree structure are created by the *backtracking* and *branch and bound* algorithms as needed.

Subset Problem tree structure

- Use a full binary tree that has 2^n leaves.
- At level i the members of the solution space are partitioned by their x_i values.
- Members with $x_i = 1$ are in the left subtree.
- Members with $x_i = 0$ are in the right subtree.

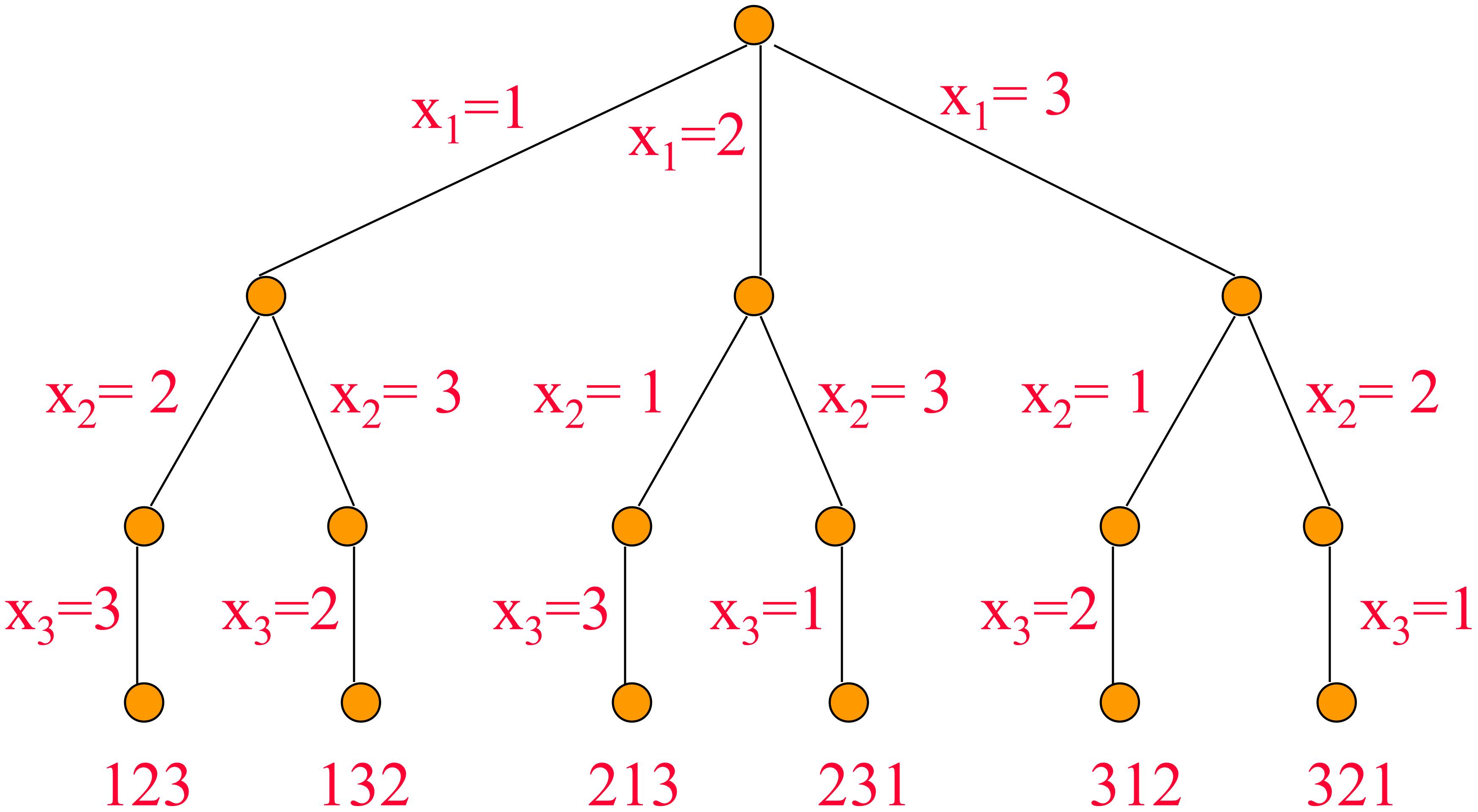
Subset Tree For n = 4
(fixed – sized tuple)



Permutation Problem tree structure

- Use a tree that has $n!$ leaves.
- At level i the members of the solution space are partitioned by their x_i values.
- Members (if any) with $x_i = 1$ are in the first subtree.
- Members (if any) with $x_i = 2$ are in the next subtree.
- And so on.

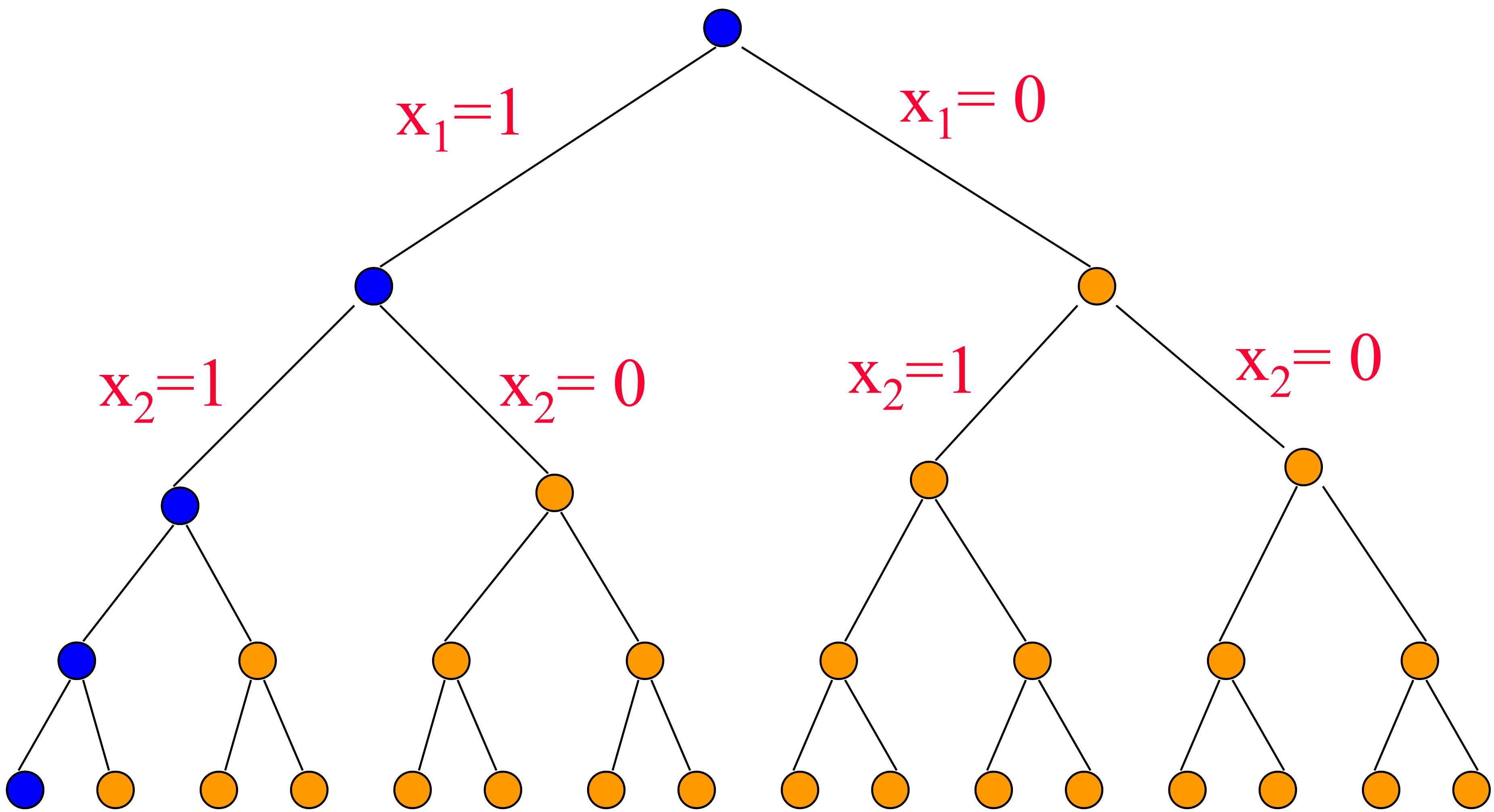
Permutation Tree For n = 3



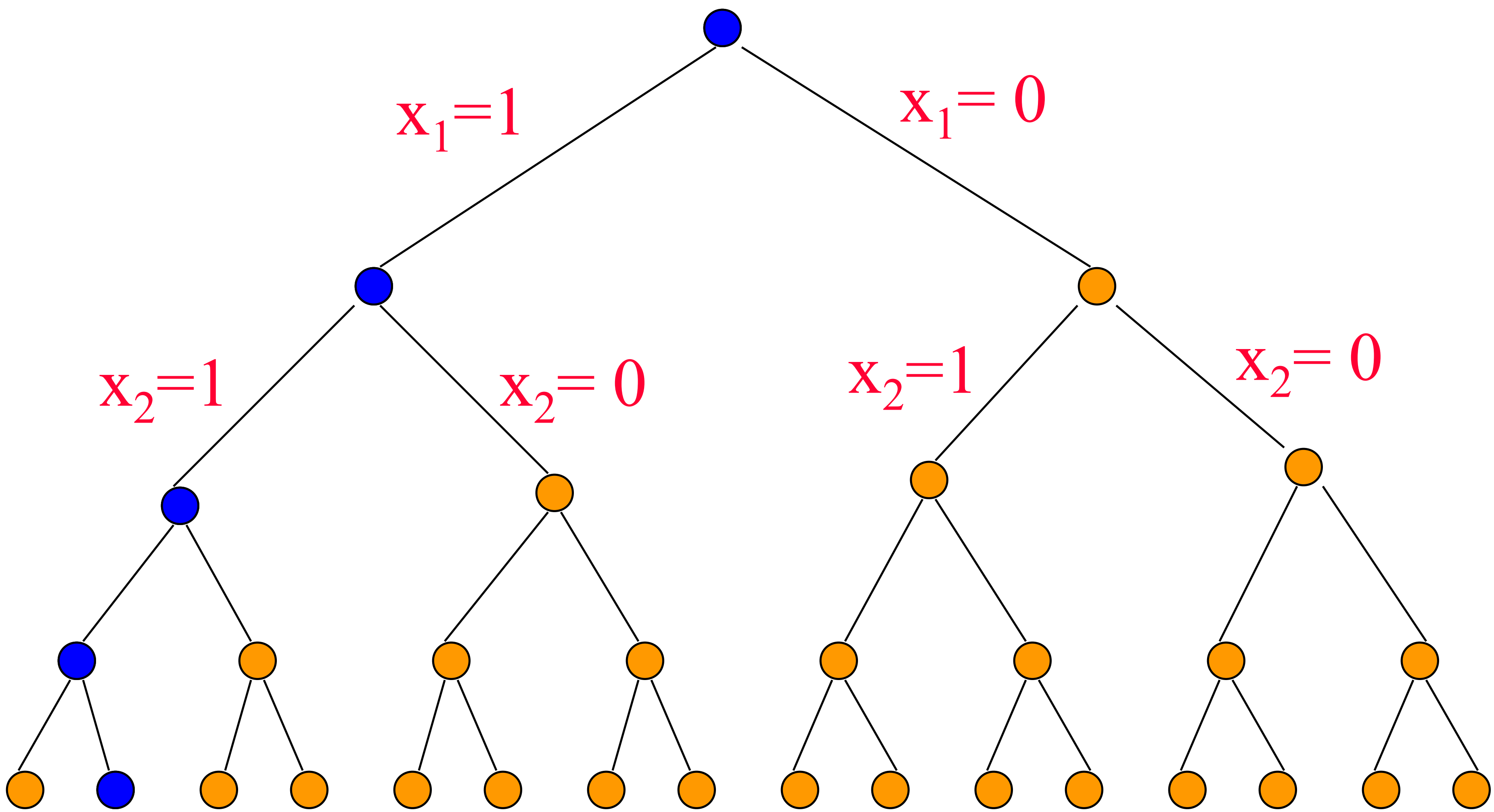
Backtracking

- Searches the solution space tree in a *depth-first* manner.
- Will be done *recursively.*
- The solution space tree exists only in your mind, not in the computer.

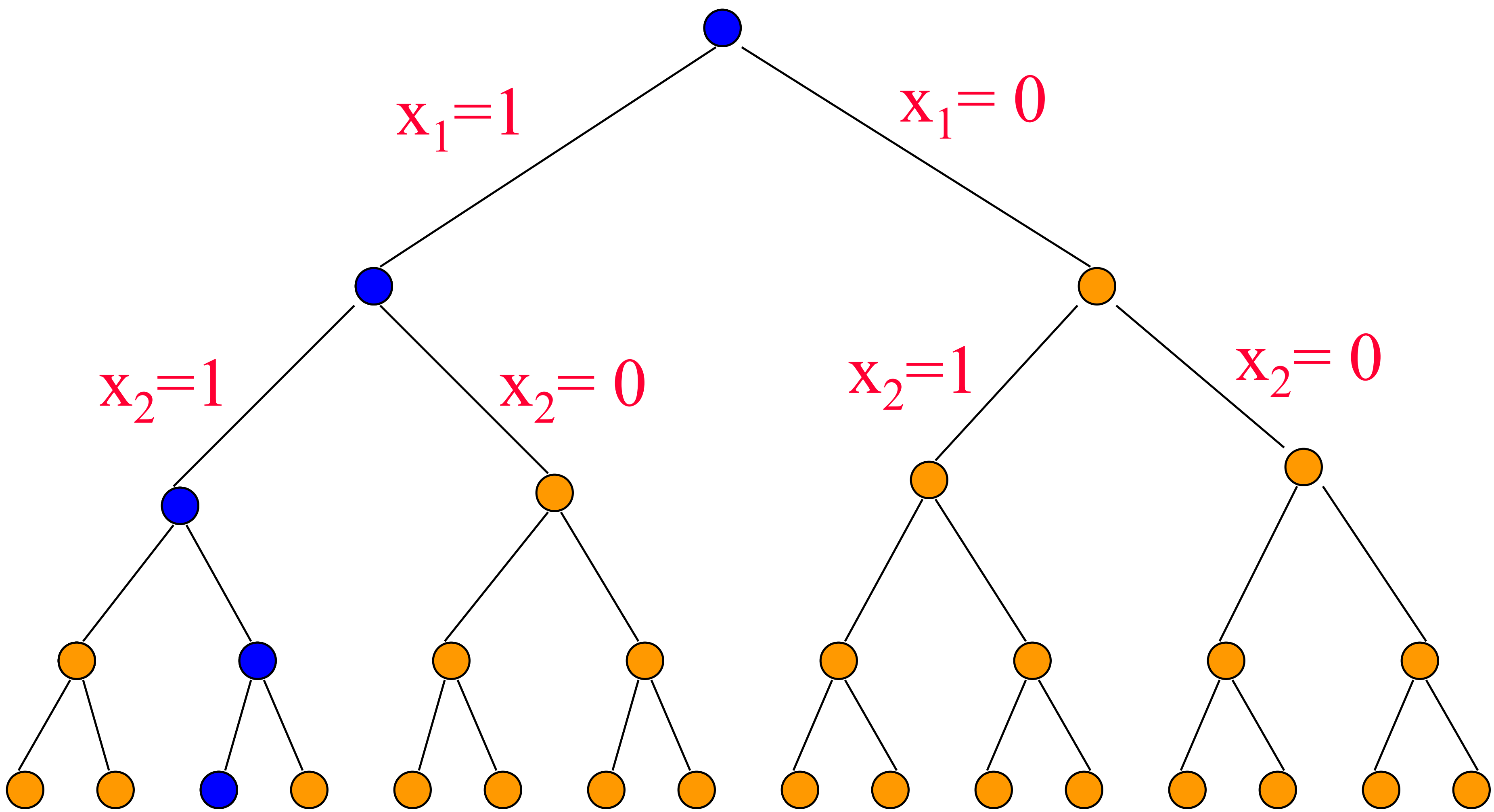
Backtracking Depth-First Search



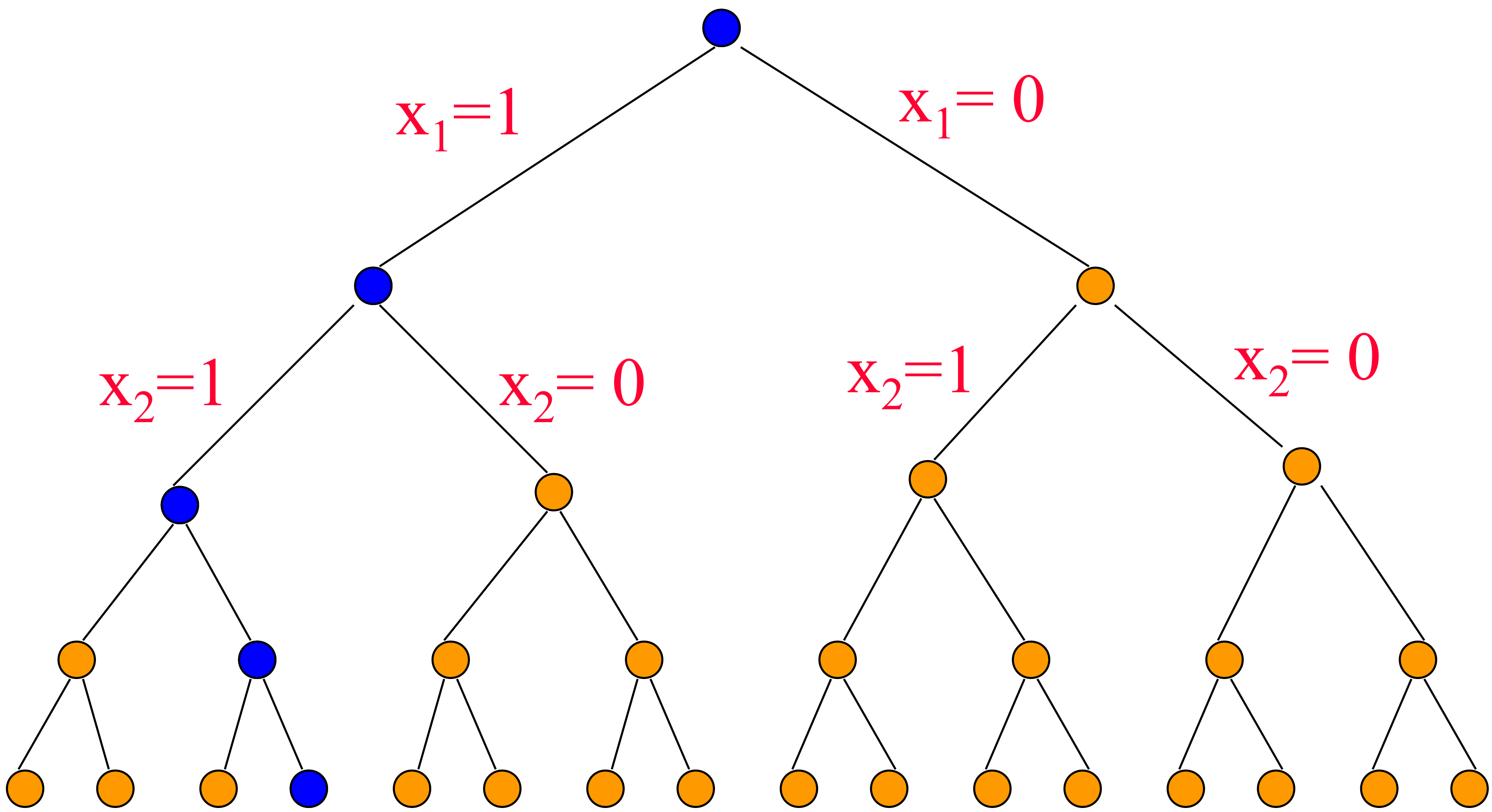
Backtracking Depth-First Search



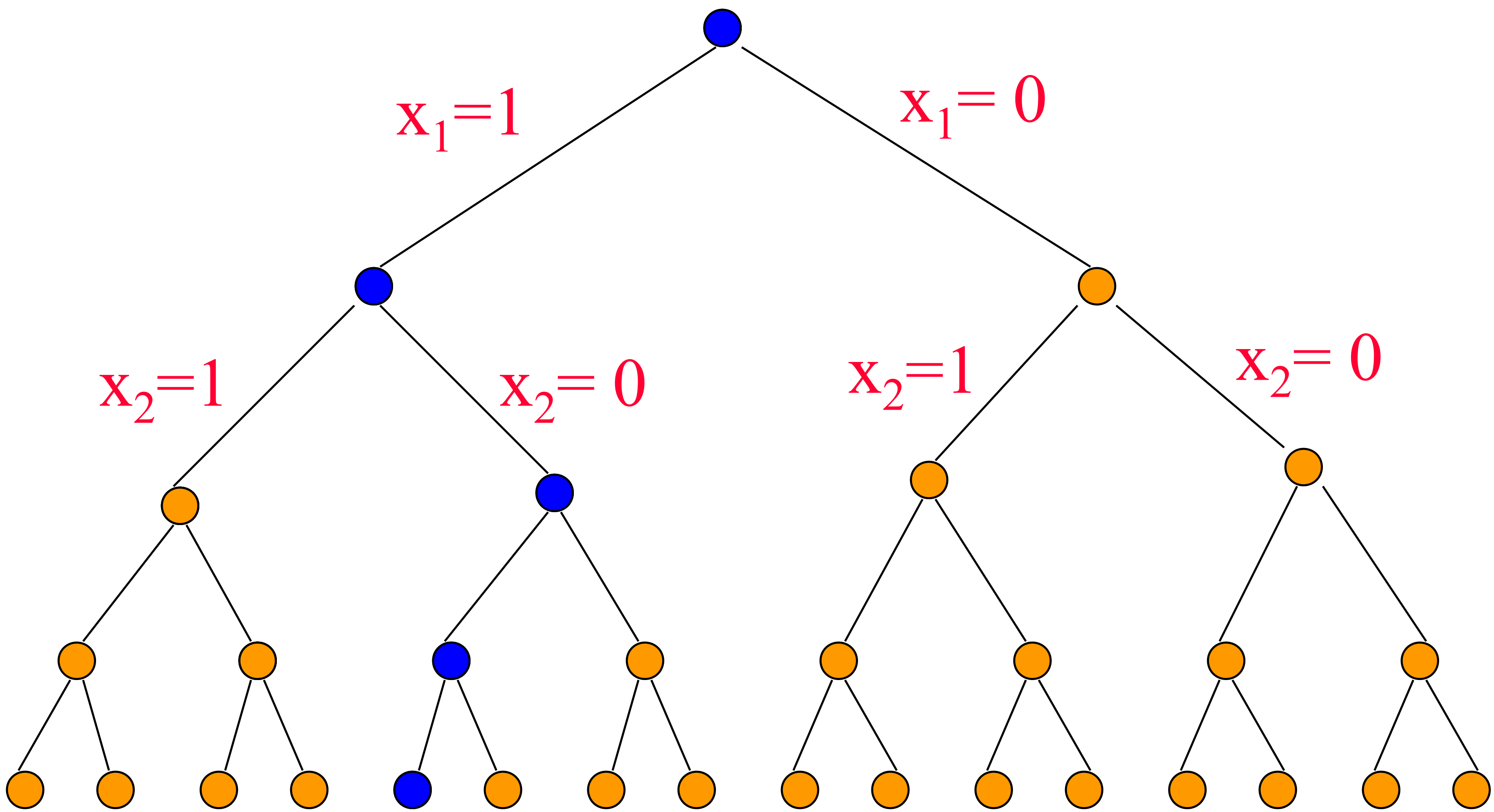
Backtracking Depth-First Search



Backtracking Depth-First Search



Backtracking Depth-First Search



n – queens problem:-

The problem is to place *n* queens on an $n \times n$ chessboard so that no two "attack" that is no two queens on the same *row, column, or diagonal.*

Defining the problem:-

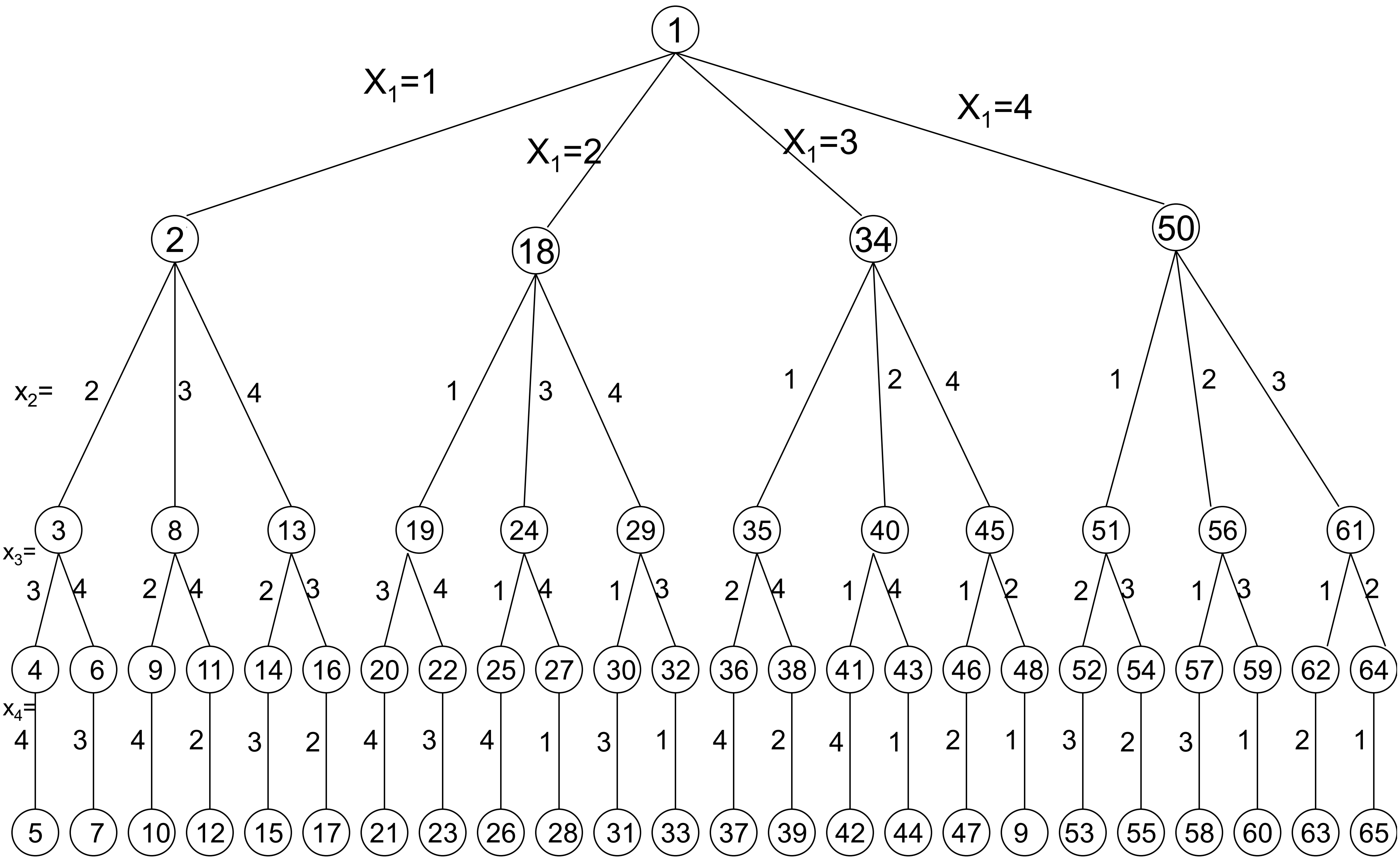
- Assume rows and columns of chessboard are numbered *1* through *n*.

- Queens also be numbered *1* through *n.*

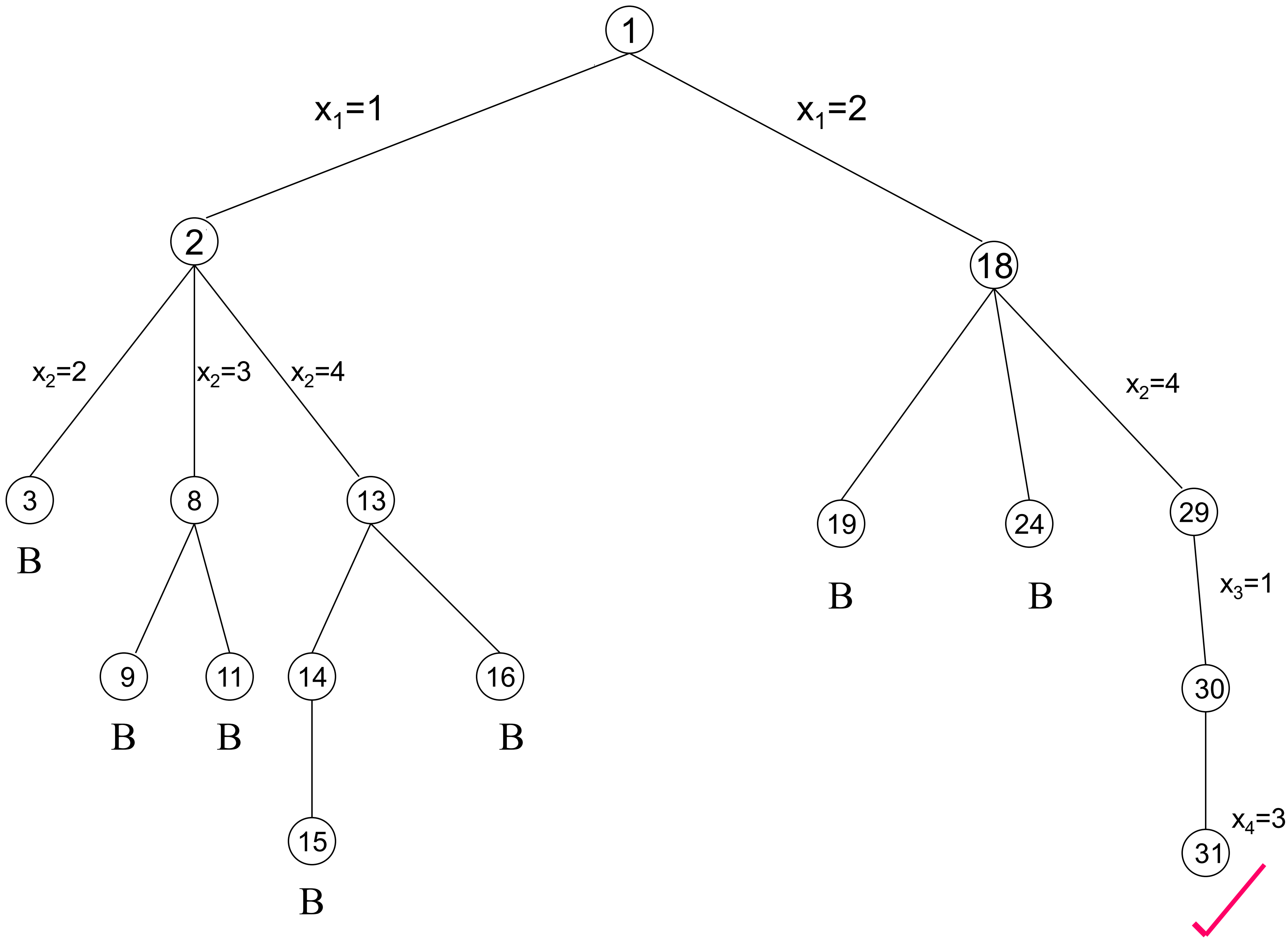
- Since each queen must be on a different row ,hence assume queen *i* is to be placed on *row i*.

- Therefore all solutions to the *n-queens* problem can be represented as *n-tuples* $(x_1, x_2, \ldots\ldots x_n)$, where x_i is the column on which *queen i* is placed.

Tree structure for the case n=4.



Tree organization of the 4-queens solution space. Nodes are numbered as in depth first search.



Portion of the tree that is generated during backtracking(n=4).

n - queens problem algorithm

- Every element on the same diagonal that runs from the upper left to the lower right has the same *row – column* value.

- Similarly, every element on the on the same diagonal that goes from the upper right to the lower left has the same *row + column* value.

- If two queens are placed at positions (i, j) and (k, l), then they are on the same diagonal only if

$$i - j = k - l \quad or \quad i + j = k + l$$

First equation implies

$$j - l = i - k$$

Second equation implies

$$j - l = k - i$$

- Therefore, two queens lie on the same diagonal if and only if $|j - l| = |i - k|$

```
Algorithm place( k,  l )
// It returns true if a queen can be placed in  column  l
// It tests both whether i is distinct from all previous values
// x [ 1 ], ….x [ k-1 ]  and there is other queen on the same
//diagonal.
// Abs(r) returns the absolute value of r.
{
          for j = 1 to k-1 do          //  for all previous queens
          {        // Two in the same column  or in the same diagonal
               if (  ( x [ j ] = l )   or   ( Abs( x[ j ] - l ) = Abs( j - k ) ) )  then
                    return false;
          }
    return true ;

}
```

Algorithm NQueen(k,n)

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

}
```

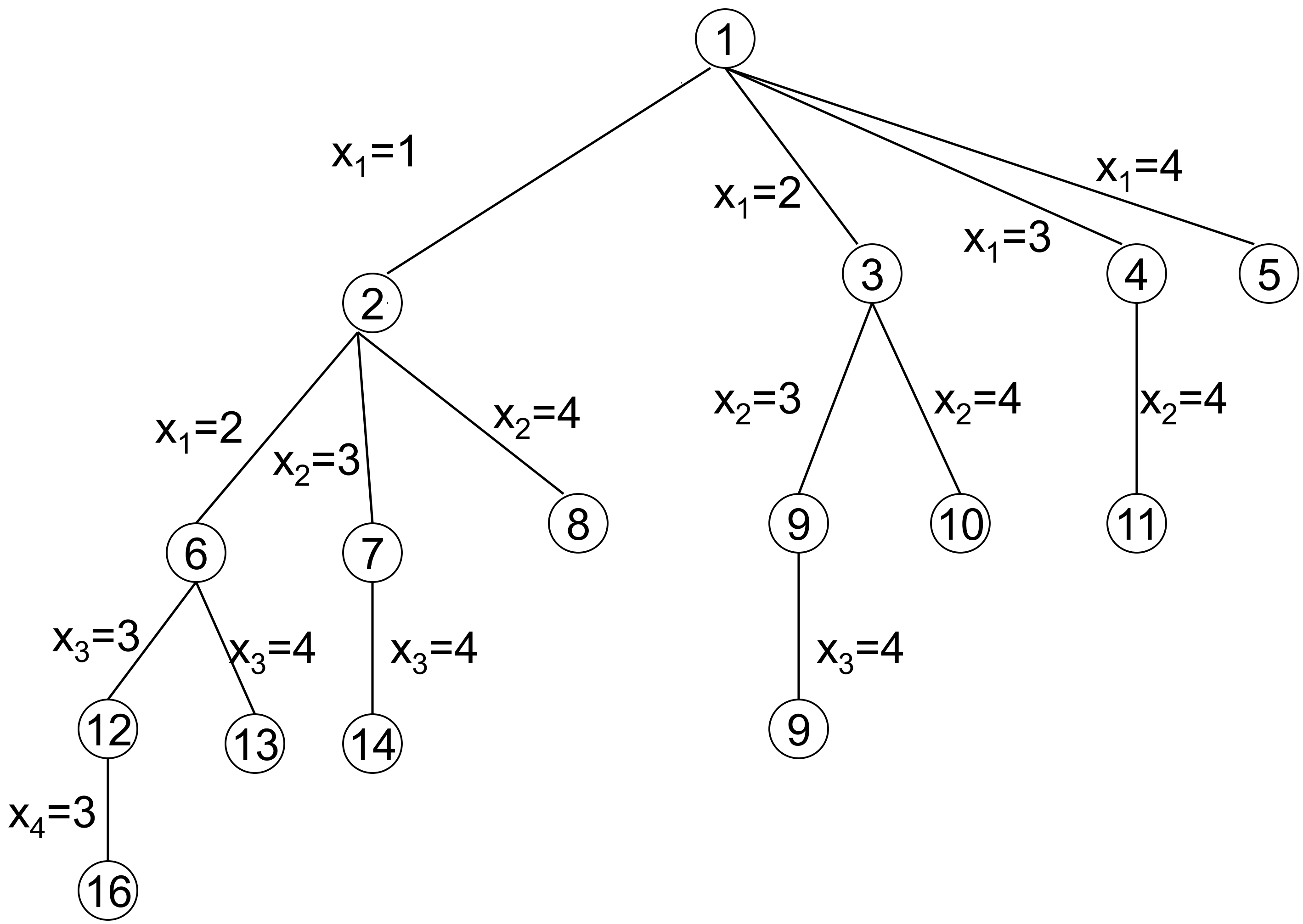
Sum of subsets

- Given *n* distinct positive numbers w_i, and *m*, find all subsets that sum to *m*.
- We can formulate this problem using either
  – *Fixed*- or *variable* – sized tuples.

Variable- sized tuple

- Ex:- n=4, (w1, w2, w3, w4)= (11,13, 24, 7), m=31.
  - Solutions are (11, 13, 7) and (24, 7)
- Rather than representing the solution by w_i *'s*, we can represent by giving the *indices* of these w_i
- Now the solutions are *(1, 2, 4)* and *(3, 4).*
- Different solutions may have *different-sized* tuples.
- We use the following condition to avoid generating multiple instances of the same subset (e.g., *(1,2,4)* and *(1,4,2)*)
  - $x_i < x_{i+1}$

Subset Tree for n=4 (variable- sized tuple)



Nodes are numbered as in Breadth first search.

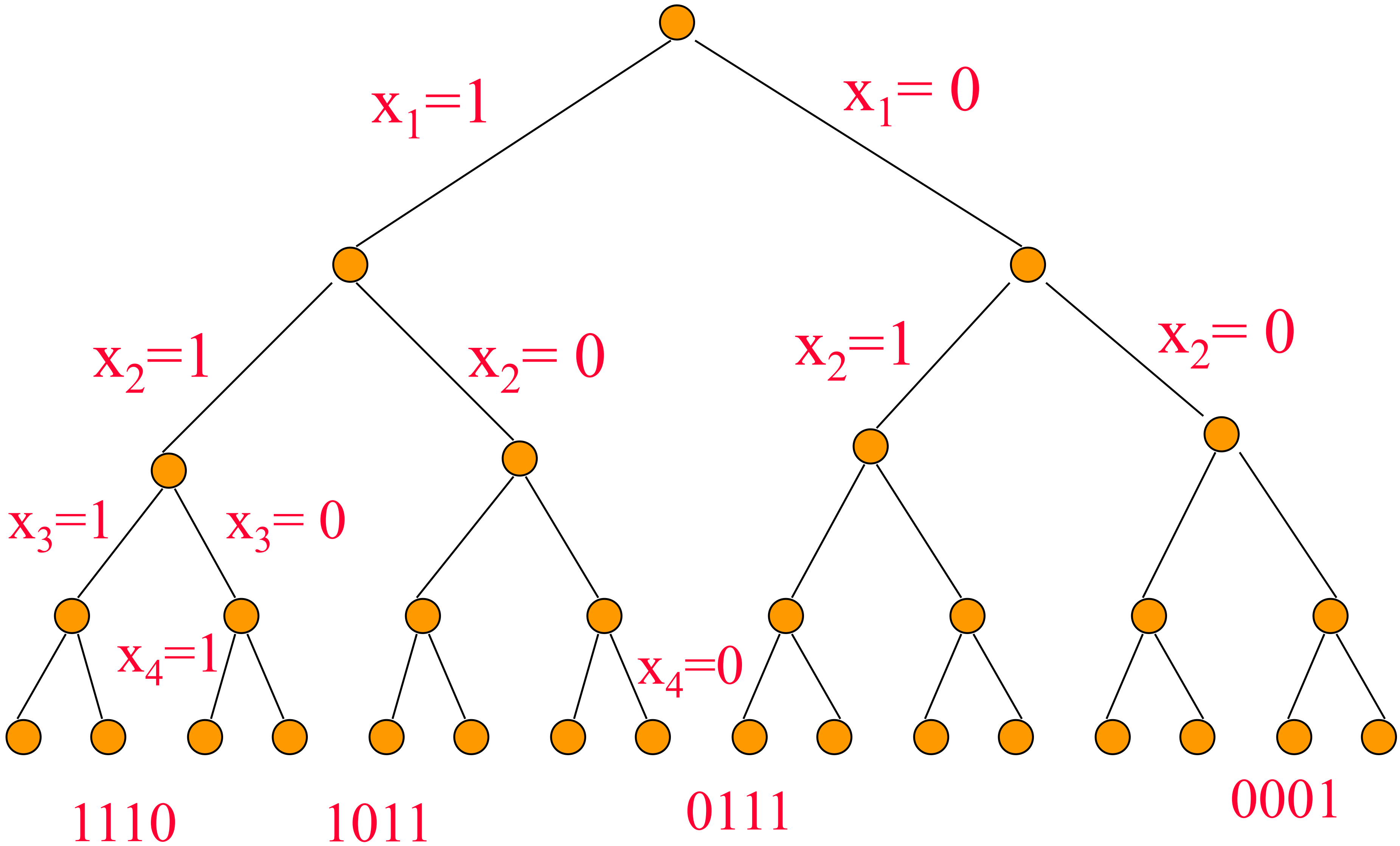
Fixed- sized tuple

- In this method, each solution subset is represented by an n-tuple $(x_1,x_2,\ldots\ldots x_n)$.
- $x_i=0$ if w_i is not chosen and $x_i=1$ if w_i is chosen

<u>Subset Tree for n=4</u> (Fixed- sized tuple)

- We have already discussed.
- Copy that tree.

Subset Tree For n = 4
(fixed – sized tuple)



Algorithm of sum of subsets

- Backtracking solution using *fixed- sized* tuple.
- A simple choice for bounding function is
  $B_k(x_1,x_2,\ldots\ldots,x_k) = true$ iff

$$\sum_{i=1}^{k} w_i x_i + \sum_{i=k+1}^{n} w_i \geq m$$

- Clearly $x_1,x_2,\ldots\ldots x_k$ cannot lead to an answer node if this condition is not satisfied.

- The bounding function *strength* further can be increased if we assume the *w_i's* in increasing order.
- In this case $x_1, x_2, \ldots\ldots x_k$ cannot lead to an answer node if

$$\sum_{i=1}^{k} w_i x_i + \sum w_{k+1} > m$$

- Therefore, the bounding functions we use are

$B_k(x_1, x_2, \ldots\ldots, x_k) = true$ iff

$$\sum_{i=1}^{k} w_i x_i + \sum_{i=k+1}^{n} w_i \geq m \quad \text{and} \quad \sum_{i=1}^{k} w_i x_i + \sum w_{k+1} \leq m$$

The initial call is SumOfSub(0, 1, $\sum_{i=1}^{n} w_i$)

Algorithm SumOfSub(s, k, r)

// Find all subsets of w[1:n] that sum to m

// It is assumed that $w[1] \leq m$ and $\sum_{i=1}^{n} w_i \geq m$

// The values of x[j] 1<=j<k, have already been determined.

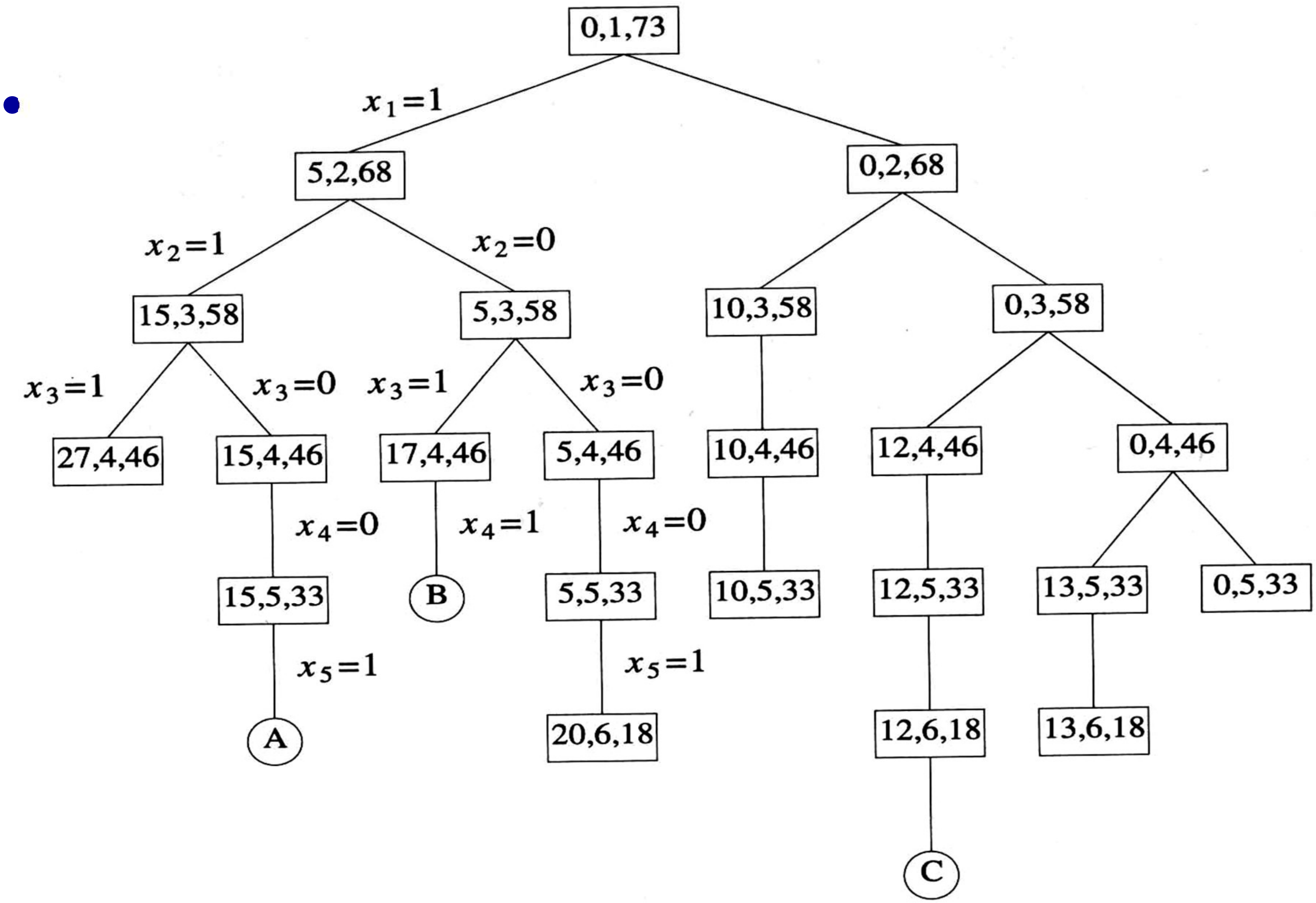
// $s = \sum_{j=1}^{K-1} w[j]*x[j]$ and $r = \sum_{j=k}^{n} w[j]$. W[j]'s in increasing order.

```
{
    x[ k ]=1;  // left child
    if( s + w[k] = m )  then write( x[ 1: k ] ) ; // Subset found
    else  if ( s + w [ k ] + w[ k+1 ] ≤ m )
        then  SumOfSub( s+ w[k], k+1, r- w[k] )
```

// Generate right child and evaluate B_k

```
if ( ( s + r – w[ k ] ≥ m ) and ( s + w[ k+1 ] ≤ m ) ) then
      {
                  x[ k ] = 0;
      }           SumOfSub( s, k+1, r- w[k] ) ;
}
```

Ex:- n=6, m=30, w [1:6]= { 5,10,12,13,15,18 }
Portion of state space tree generated by SumOfSub.
circular nodes indicate subsets with sums equal to m.

- 

General method of Backtracking

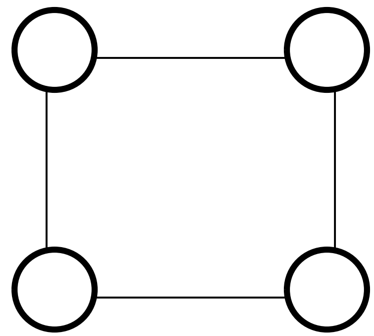
- Let T (x[1], x[2],…..x[k-1] be the partial solution

```
Algorithm   Backtrack( k )
{
        for( each x[k] € T ( x[1], x[2],…..x[ k-1 ] )    do
        {
                if( B_k (x[1], x[2],…..x[ k-1 ], x[k] is true )  then
                {
                        if ( x[1], x[2],…..x[ k-1 ], x[k]  ) is an answer )
                                then  write( x [ 1; k ] );
                        if ( k < n ) then Backtrack ( k+1 );
                }
        }
}
```

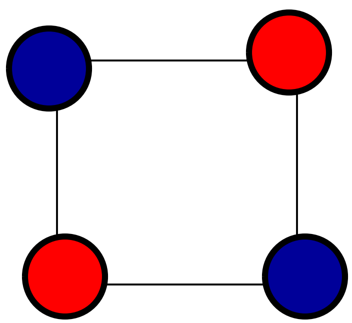
Graph Coloring Problem

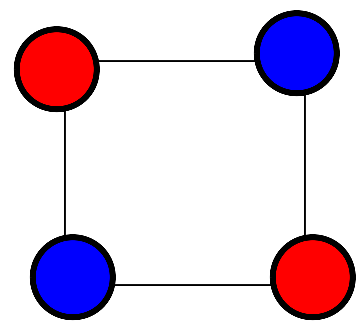
- Assign colors to the vertices of a graph so that no adjacent vertices share the same color.
  – Vertices i, j are adjacent if there is an edge from vertex *i* to vertex *j*.

Example



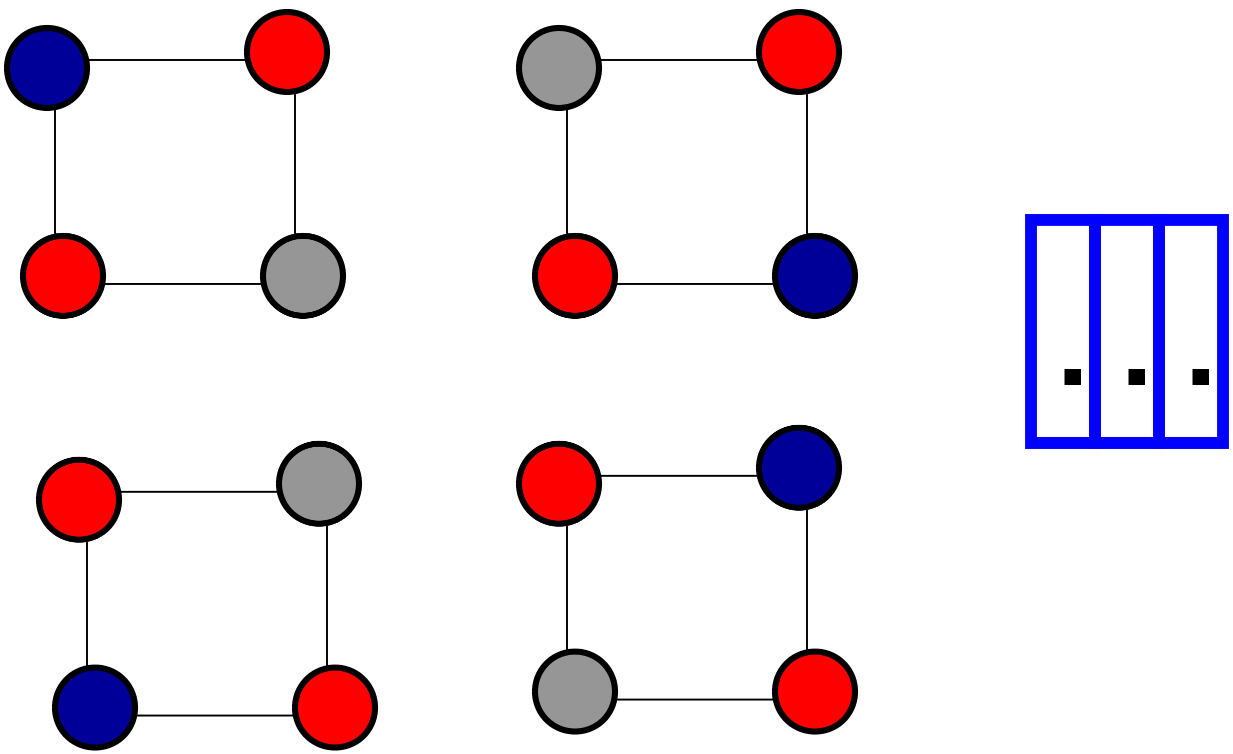
No.of Colors used: 2





Number of Possible ways : 2

No.of Colors used: 3



Some possible ways

- M-colorability optimization problem asks for the smallest integer for which the graph can be colored.
- This number is called chromatic number.

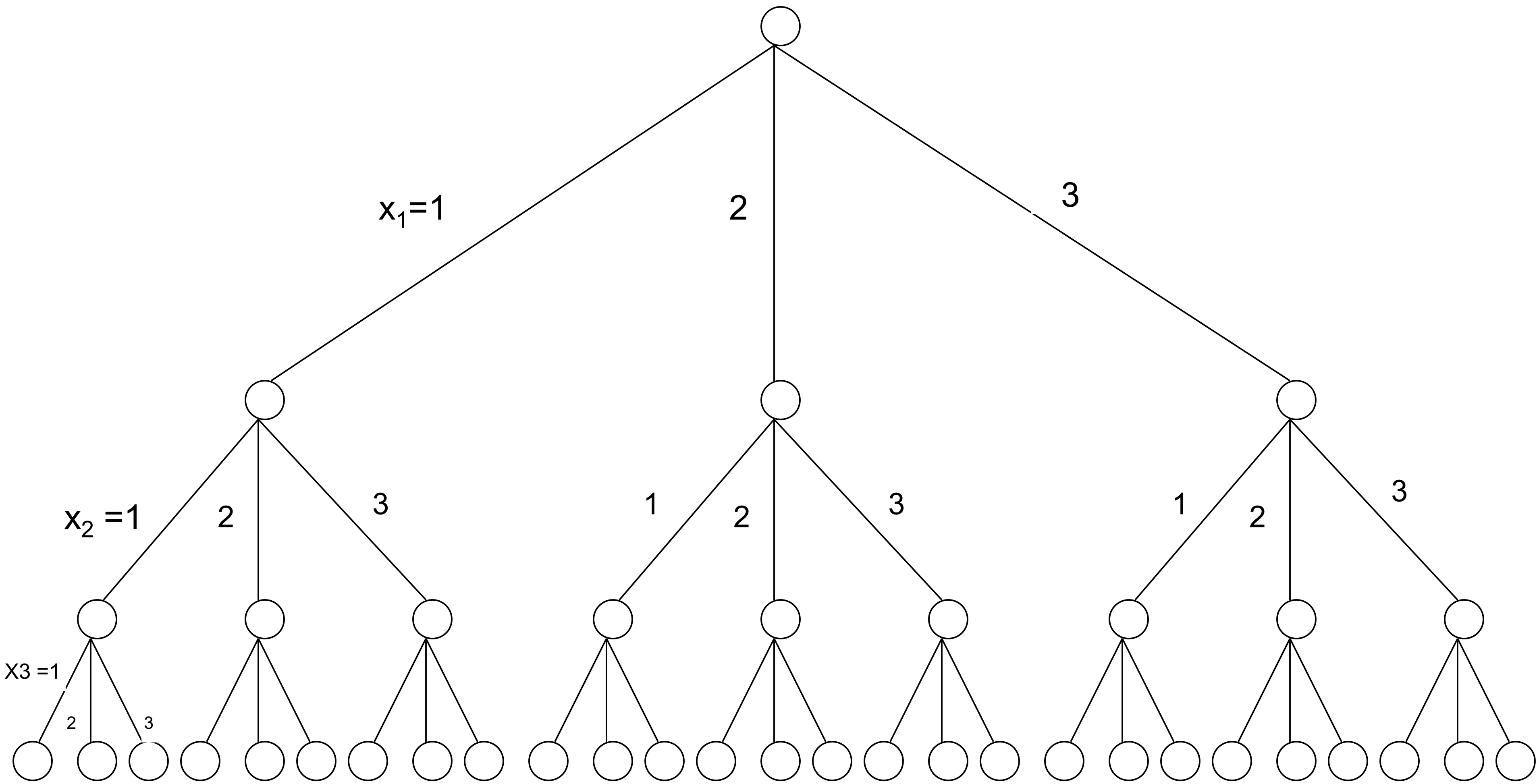
m-colorings problem:-

- Find all ways to color a graph with at *most m* colors.

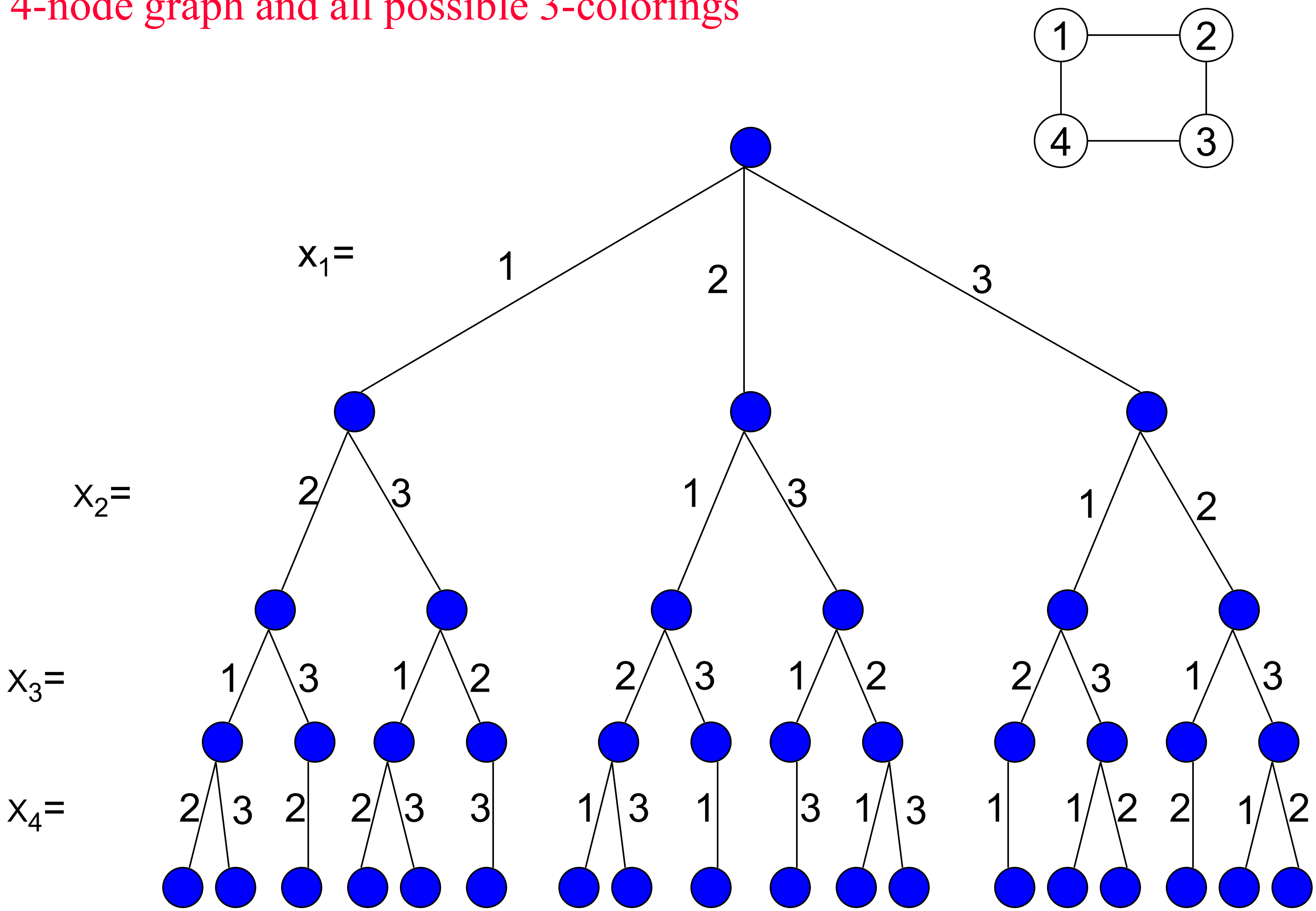
Problem formulation:-

- Represent the graph with adjacency matrix G[1:n,1:n].
- The colors are represented by integer numbers 1,2,….m.
- Solution is represented by n- tuple $(x_1,\ldots x_n)$, where x_i is the color of node i.

Solution space tree for mColorigng when n=3 and m=3



A 4-node graph and all possible 3-colorings



Algorithm :- finding all m- colorings of a graph.

Function mColoring is begun by first assiging the graph to its adjacency matrix, setting the array x[ ] to zero, and then invoking the statement mColoring(1);

```
Algorithm mColoring( k )
// 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 );
}
```

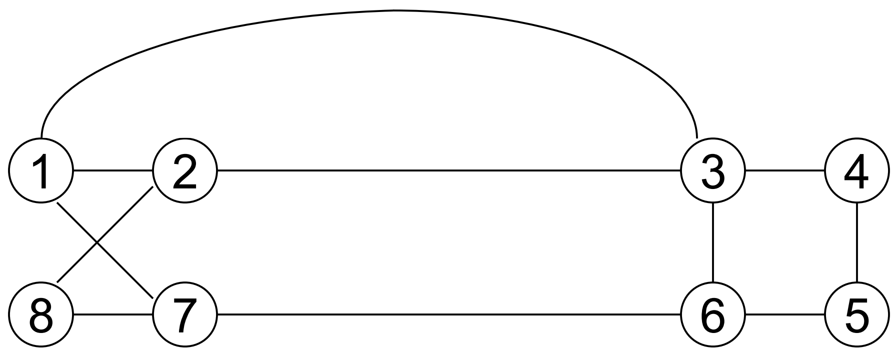
```
Algorithm NextValue( k )
// x[1],.....x[ k-1 ] have been assigned integer values in the range  [ 1 ,m ].
// A value for x[ k ] is determined in the range [ 0,m ]

{
    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
        {
            if (( G[ k, j ] ≠ 0 )  and ( x[k] = x[j] ))  then
                    break;
        }
        if( j = n+1 ) then return; // Color found
    } until ( false );
}
```

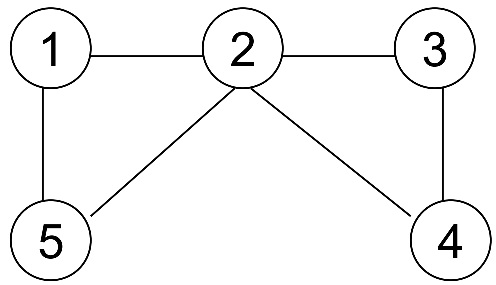
Hamiltonian cycles:-

A Hamiltonian cycle is a *round-trip path along n edges* of connected undirected graph G that visits every *vertex once* and returns to its *starting* position.



The above graph contains the Hamiltonian cycle

1,2,8,7,6,5,4,3,1



The above graph does not contain Hamiltonian cycles.

Find all possible Hamiltonian cycles

Problem formulation:-

- Represent the grapth with adjacency matrix *G[1:n,1:n]*.

- Solution is represented by *n- tuple* $(x_1, \ldots. x_n)$, where x_i represents the i^{th} visited vertex of the cycle.

- Start by setting *x[2: n]* to zero and *x[1]=1*, and then executing *Hamiltonian(2);*

```
Algorithm Hamiltonian( k )

{
   repeat
   {     // Generate values for x[k]

         NextValue( k ); // Assign a legal next value to x[ k ]

         if ( x[k]=0 ) then return;   // No new value possible


           if ( k=n ) then   write( x[1:n] );
         else Hamiltonian( k+1);
   } until ( false );
}
```

```
Algorithm NextValue( k )
{
    repeat
    {
        x[k] = ( x[k] +1) mod ( n+1 );          // Next vertex.
        if ( x[k]=0 ) then return;
        if( G[ x[ k-1], x[ k ]  ] ≠ 0 )
        {
            for j:= 1 to k-1 do  if ( x[ j ] = x [k ] ) then break;

            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 );
}
```

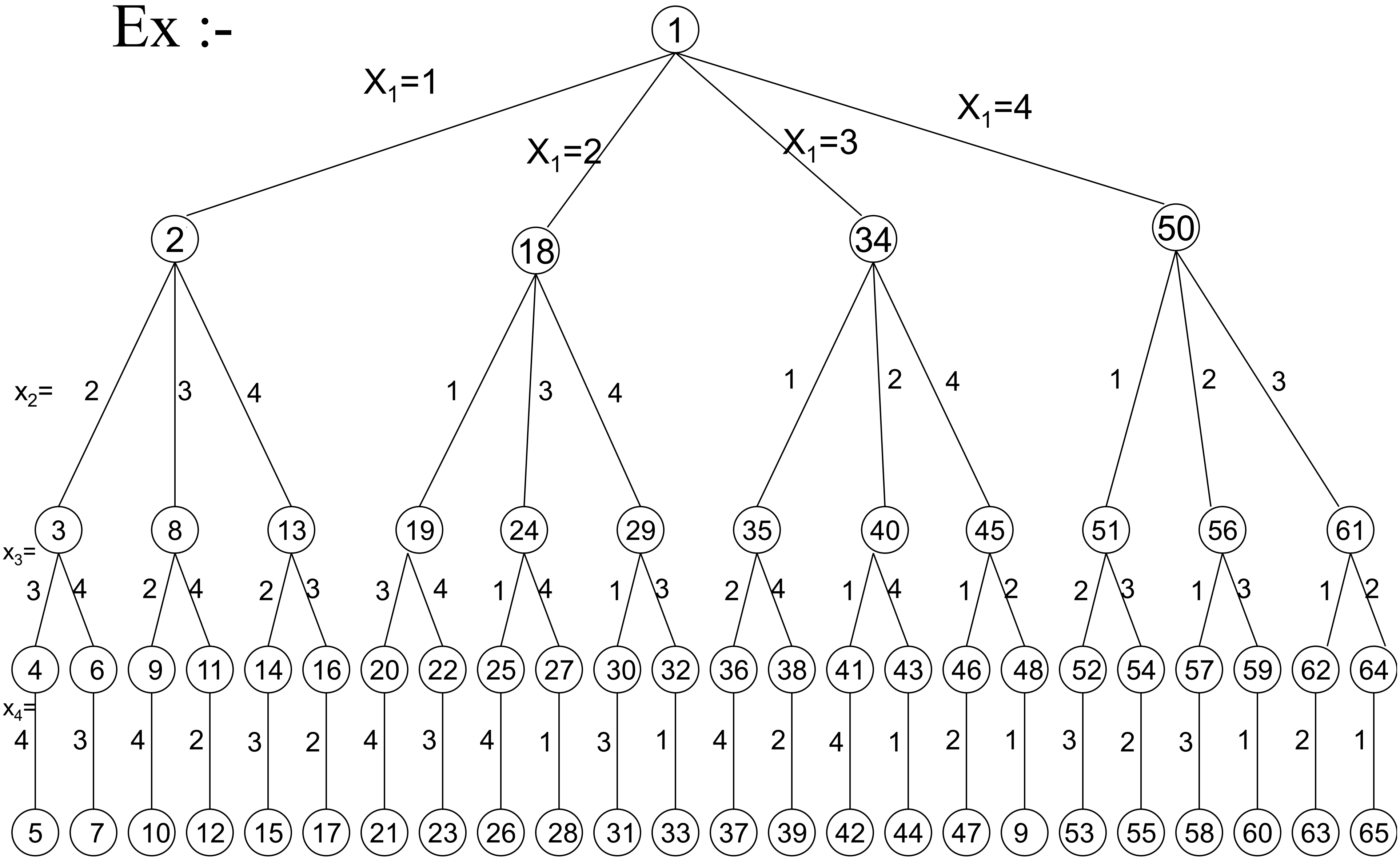
State Space :-

All paths from the root to other nodes define the *state space* of the problem.

Solution Space :-

All paths from the root to solution states define the *solution space* of the problem.

Ex :-



Tree organization of the 4-queens solution space. Nodes are numbered as in depth first search.

Problem state :-

Each node in the tree is a problem state.

Ex:-

1 2 18 and so on

Solution States :-

These are those problem states S for which the path from the root to S define a tuple in the solution space.

Ex:-

5 7 10 12 15 17 21 and so on

General Backtracking Algorithms :-

- Assume all answer nodes are to be found.

- Let (x1,x2,…..,xi) be a path from the root to a node xi in a state space tree .

- Let T(x1,x2,…..xi) be the set of all possible paths for x_{i+1} such that (x1,x2,…….., x_{i+1}) is also a path to a problem state.

- Assume bounding functionBi+1 (implicit conditions) such that if Bi+1(x1,x2,……,xi+1) is false for a path (x1,x2,…xi+1) , then the path cannot be expanded to reach an answer node.

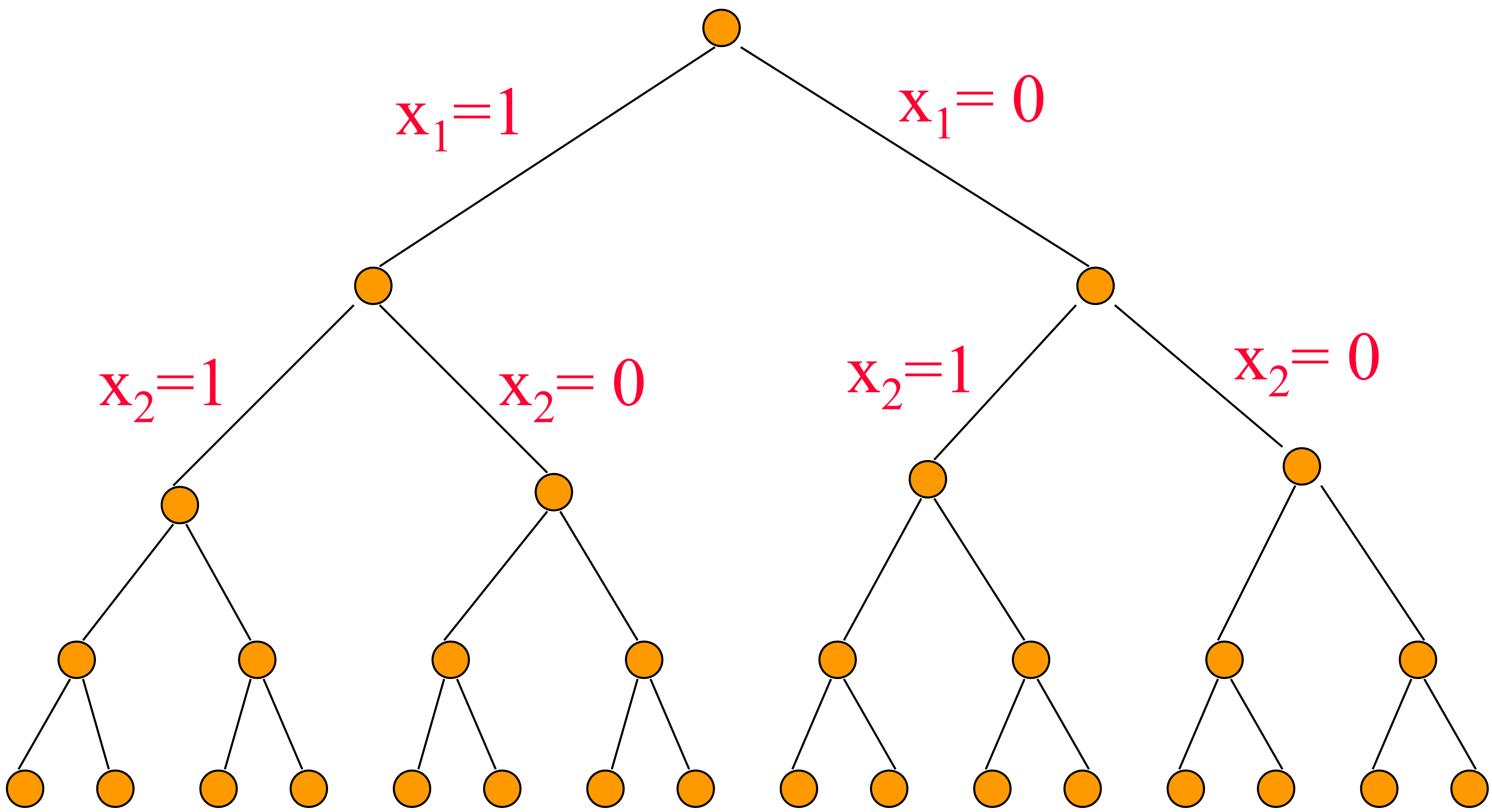
General Recursive backtracking Algorithm:-

Algorithm Backtrack(k)

```
// The first k-1 values have been assigned.
// x[ ] and n are global.
{
        for ( each x[k] € T(x[1],……x[k-1] )
        {
                if( Bk( x[1],x[2],…..x[k] ≠ 0)  then
                {
                        if(x[1],x[2],…..x[k] is a path to answer node )
                                then write( x[1:k] );
                        if ( k < n) then Backtrack( k+1 );
                }
        }
}
```

$O(2^n)$ Subet Sum & Bounding Functions

$\{10, 5, 2, 1\}$, $c = 14$



Each forward and backward move takes $O(1)$ time.

Bounding Functions

- When a node that represents a subset whose sum equals the desired sum c, terminate.
- When a node that represents a subset whose sum exceeds the desired sum c, backtrack. I.e., do not enter its subtrees, go back to parent node.
- Keep a variable r that gives you the sum of the numbers not yet considered. When you move to a right child, check if current subset sum + r >= c. If not, backtrack.

Backtracking

- Space required is O(tree height).
- With effective bounding functions, large instances can often be solved.
- For some problems (e.g., 0/1 knapsack), the answer (or a very good solution) may be found quickly but a lot of additional time is needed to complete the search of the tree.
- Run backtracking for as much time as is feasible and use best solution found up to that time.

Branch And Bound

- Search the tree using a breadth-first search (FIFO branch and bound).
- Search the tree as in a bfs, but replace the FIFO queue with a stack (LIFO branch and bound).
- Replace the FIFO queue with a priority queue (least-cost (or max priority) branch and bound). The priority of a node p in the queue is based on an estimate of the likelihood that the answer node is in the subtree whose root is p.

Branch And Bound

- Space required is O(number of leaves).
- For some problems, solutions are at different levels of the tree (e.g., 16 puzzle).

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

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

Branch And Bound

- FIFO branch and bound finds solution closest to root.
- Backtracking may never find a solution because tree depth is infinite (unless repeating configurations are eliminated).

- Least-cost branch and bound directs the search to parts of the space most likely to contain the answer. So it could perform better than backtracking.

Branch and Bound

(unit-vii)

- Backtracking is effective for *subset* or *permutation* problems.
- Backtracking is not good for *optimization* problems.
- This drawback is rectified by using *Branch And Bound* technique.
- *Branch And Bound is* applicable for only *optimization* problems.
- Branch and Bound also uses *bounding function* similar to backtracking.

Terminology of tree organization

Problem state :-

Each node in the tree is a problem state.

State Space :-

All paths from the root to problem states define the *state space* of the problem.

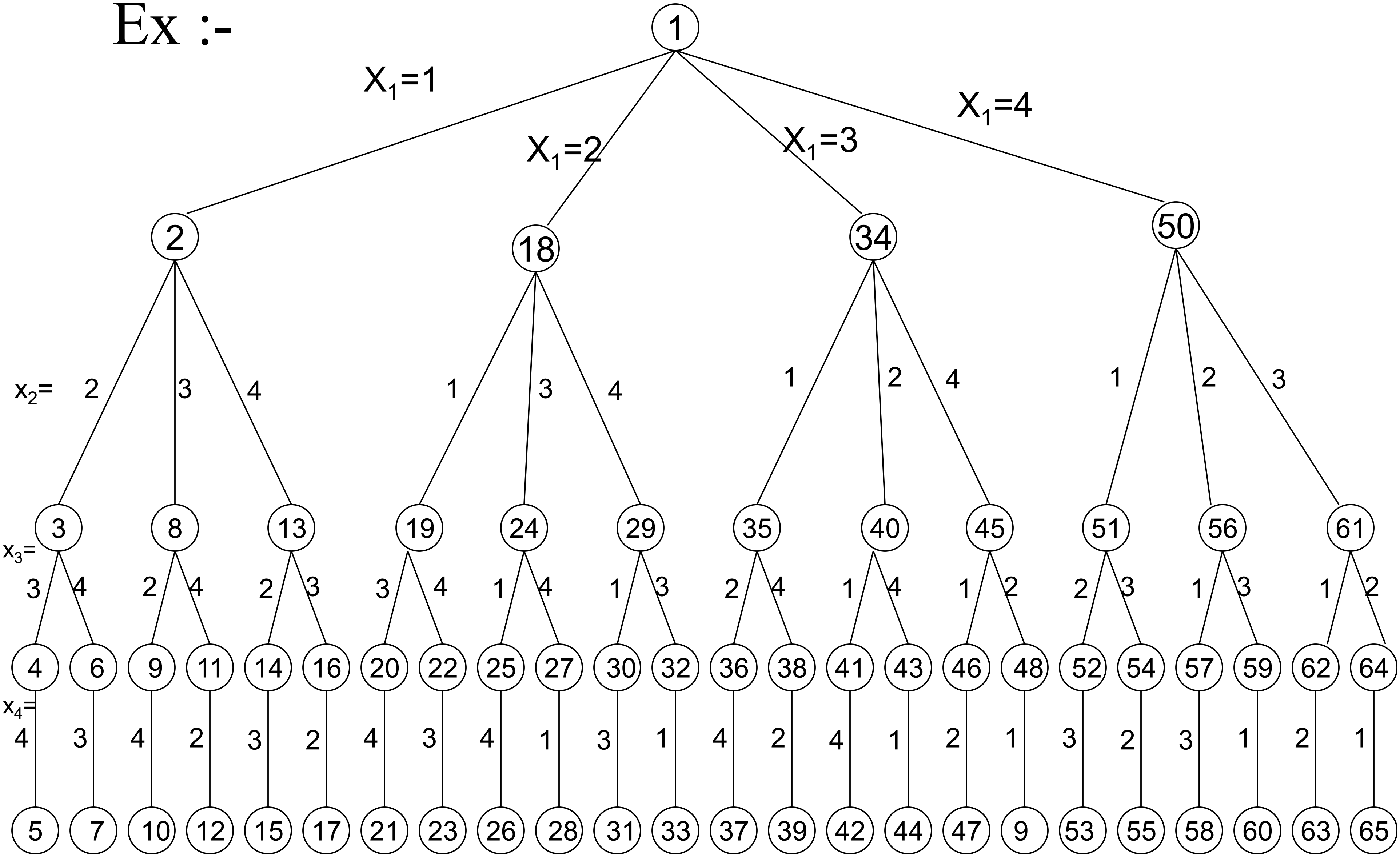
Solution States :-

These are those problem states S for which the path from the root to S define a tuple in the solution space.

Solution Space :-

All paths from the root to *solution states* define the *solution space* of the problem.

Ex :-



Tree organization of the 4-queens solution space. Nodes are numbered as in depth first search.

- A node which has been generated and all of whose children have not yet been generated is called a *live node.*

- The *live node* whose children are currently being generated is called *E-node (Expanding node)*.

- A *dead node* is a generated node which can not to be expanded further or all of whose children have been generated.

Branch And Bound

Nodes will be expanded in three ways.

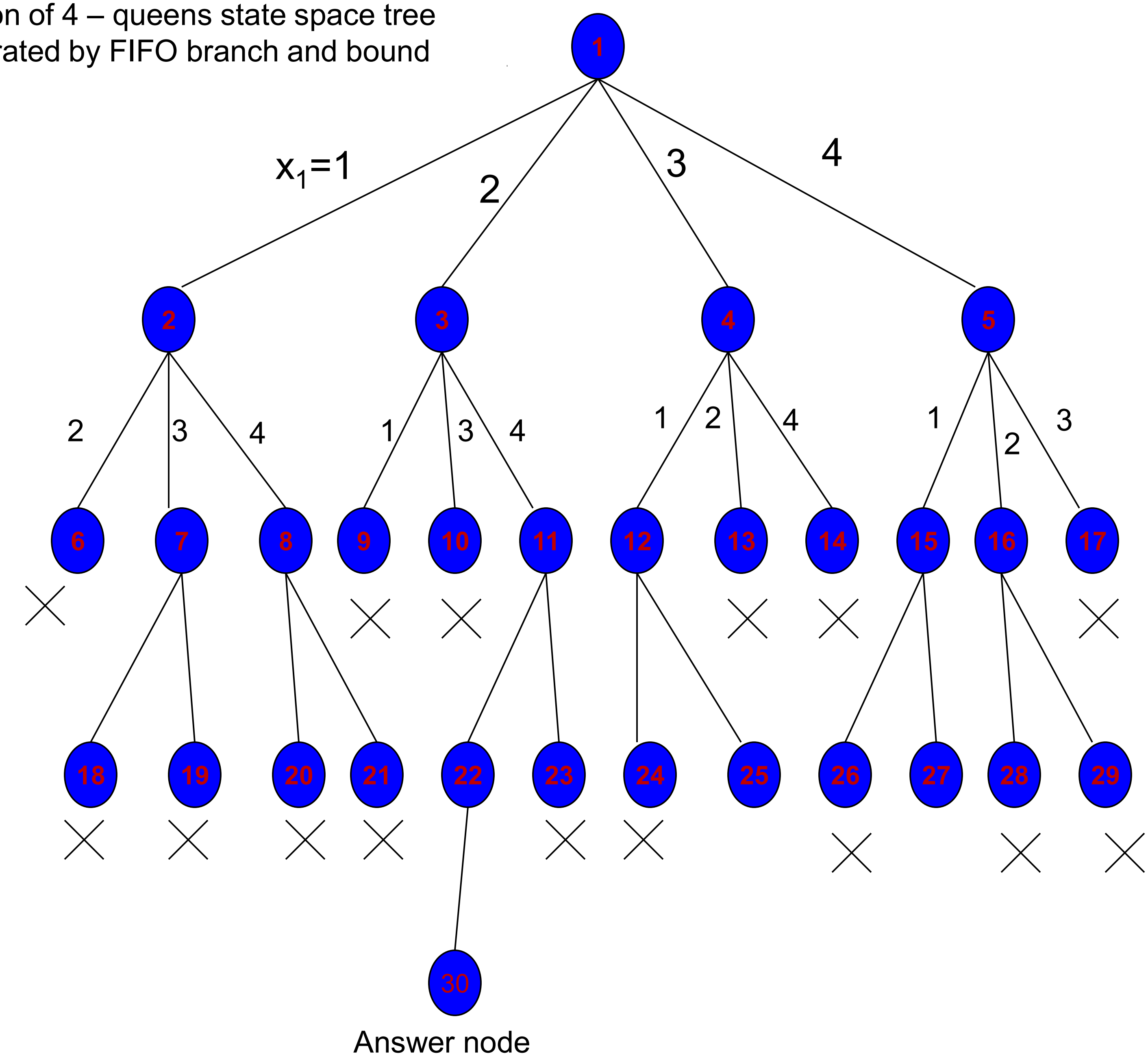
- FIFO branch and bound-- queue
- LIFO branch and bound-- stack
- *Least-Cost (or max priority) branch and bound)—priority queue.*

- *Least-cost* branch and bound directs the search to the parts which most likely to contain the answer.

Ex:- 4 – Queens Problem

FIFO Branch And Bound Algorithm

Portion of 4 – queens state space tree
generated by FIFO branch and bound



- In this case *backtracking* method is superior than *branch and bound* method.

Least Cost (LC) search :-

- In both FIFO and LIFO branch and bound the selection rule for the next E-node does not give any preference to a node that has a very good chance of getting an answer node quickly.

- In the above example when node 22 is generated, it should have become obvious that this node will lead to an answer node in one move.

- However , the FIFO rule first requires the expansion of all live nodes generated before node 22 was expanded.

- The search for an answer node can be speeded by using an *"intelligent "* ranking function *$\hat{c}$ (.)* for live nodes.

- The next *E-node* is selected on the basis of this ranking function.

- If we use ranking function that assigns node 22 a better rank than all other live nodes , then node 22 will become the E-node following node 29.

- The ideal way to assign ranks will be on the basis of the additional effort (cost) needed to reach an answer node from the live node.

For any node x, this could be

1) The number of nodes in the subtree with root x that need to be generated before an answer node is generated.

OR more simply

2) The number of levels in the subtree with root x that need to be generated to get an answer node.

Using cost measure 2,

- In the above fig. the cost of the root is 4 (node 30 is four levels from node 1) .

- The difficulty with cost functions is that computing the *cost* of the node usually involves a search of the subtree *x* for an answer node.

- Hence , by the time the cost of a node is determined , that subtree has been searched and there is no need to explore x again. For this reason , search algorithms usually rank nodes only on the basis of an estimated *cost* $\hat{g}(.)$.

- Let $\hat{g}(x)$ be an *estimate* of the additional *cost* needed to reach an answer node from *x*.
- Then , node *x* is assigned a rank using a function $\hat{c}(.)$ such that

$$\hat{c}(.) = h(x) + \hat{g}(x)$$

where

h(x) is the cost of reaching *x* from the root

Actual cost function *c(.)* is defined as follows:

- if *x* is an answer node , then *c(x)* is the cost of reaching *x* from the root .
- if *x* is not an answer node , then $c(x)=\infty$, it means that *subtree with root x* contains no answer node.
- otherwise *c(x)* equal to the cost of a *minimum cost* node in the *subtree with root x*.

- $\hat{c}(.)$ is an approximation to c(.)

- A FIFO or LIFO search always generates the state space tree by levels.

- What we need is more "intelligent" search method .
- We can associate a cost c(x) with each node x in the state space tree.
- The cost c(x) is the length of a path from the root to a nearest goal node(if any) in the subtree with root x.

  Thus in the above fig c(1)=c(4)=c(10)=c(23)=3.

- When such a cost function is available, a very efficient search can be carried out.

- In this search strategy , the only nodes to become E-nodes are nodes on the path from the root to a nearest goal node.

- Unfortunately, this is an impractical.

We can only compute estimate $\hat{c}(x)$ of $c(x)$. we can write $\hat{c}(x)=f(x)+\hat{g}(x)$, where $f(x)$ is the length of the path from the root to node x and $\hat{g}(x)$ is an estimate of the length of a shortest path from x to a goal node in the subtree with root x.

one possible choice for $\hat{g}(x)$ is

$\hat{g}(x)$ = number of nonblank tiles not in their goal position.

Control Abstraction of LC-Search:-

```
Algorithm LCSearch( t )
// 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
                }
```

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

- (x ->parent)=E is to print answer path.

Bounding:-

- The bounding functions are used to avoid the generation of sub trees that do not contain the answer nodes. In bounding *upper* and *lower* bounds are generated at each node.

- A cost function $\hat{c}(.)$ such that $\hat{c}(x) <= c(x)$ is used to provide the *lower* bounds on solutions obtainable from any node x.

- If *upper* is an upper bound on the cost of a minimum cost solution, then all live nodes x with $\hat{c}(x) > upper$ can be killed.

- *upper* is updated whenever a child is generated.

Job sequencing with deadlines

- The objective of this problem is to select a subset j of n jobs such that all jobs in j can be completed by their deadlines and the penalty incurred is minimum among all possible subsets j. such a j is optimal.

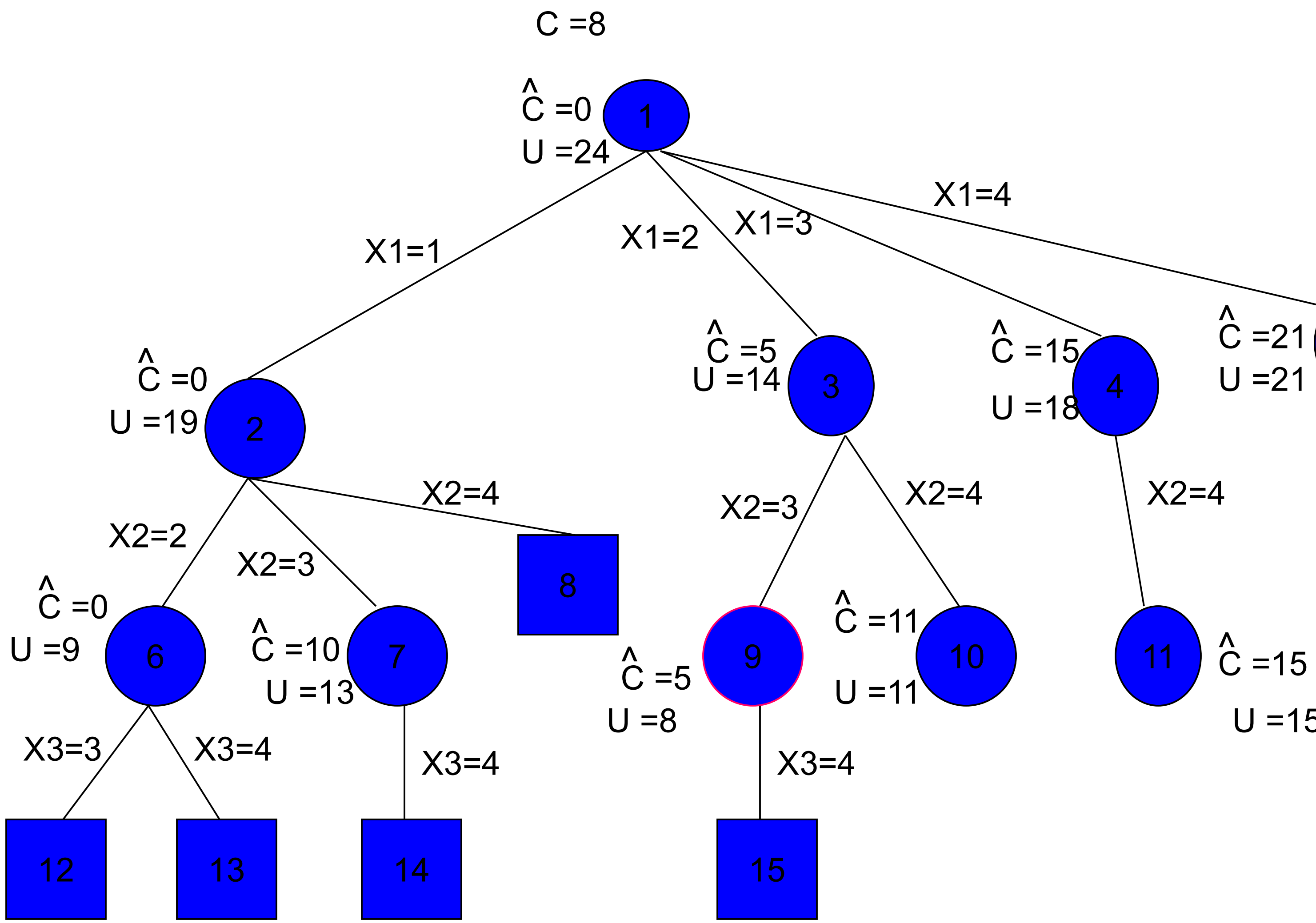
Ex:- let n=4

Job index	p_i	d_i	t_i
1	5	1	1
2	10	3	2
3	6	2	1
4	3	1	1

The solution space for this instance consists of all possible subsets of the job index set (1,2,3,4).

This space can be organized into a tree in two ways .

1. Using fixed size tuple formulation.
2. Using variable size tuple formulation.



C =8
Ĉ =0
U =24
1
X1=1
X1=2
X1=3
X1=4
Ĉ =0
U =19
2
Ĉ =5
U =14
3
Ĉ =15
U =18
4
Ĉ =21
U =21
5
X2=2
X2=3
X2=4
X2=3
X2=4
X2=4
Ĉ =0
U =9
6
Ĉ =10
U =13
7
8
Ĉ =5
U =8
9
Ĉ =11
U =11
10
11
Ĉ =15
U =15
X3=3
X3=4
X3=4
X3=4
12
13
14
15

- The above fig corresponds to the variable tuple size formulation.
- Square nodes represent infeasible subsets.
- All nonsquare nodes are answer nodes.
- Node 9 represents an optimal solution and is the only minimum-cost answer node. For this node j=(2,3) and the penalty (cost) is 8.

A cost function c() for the above solution space can be defined as

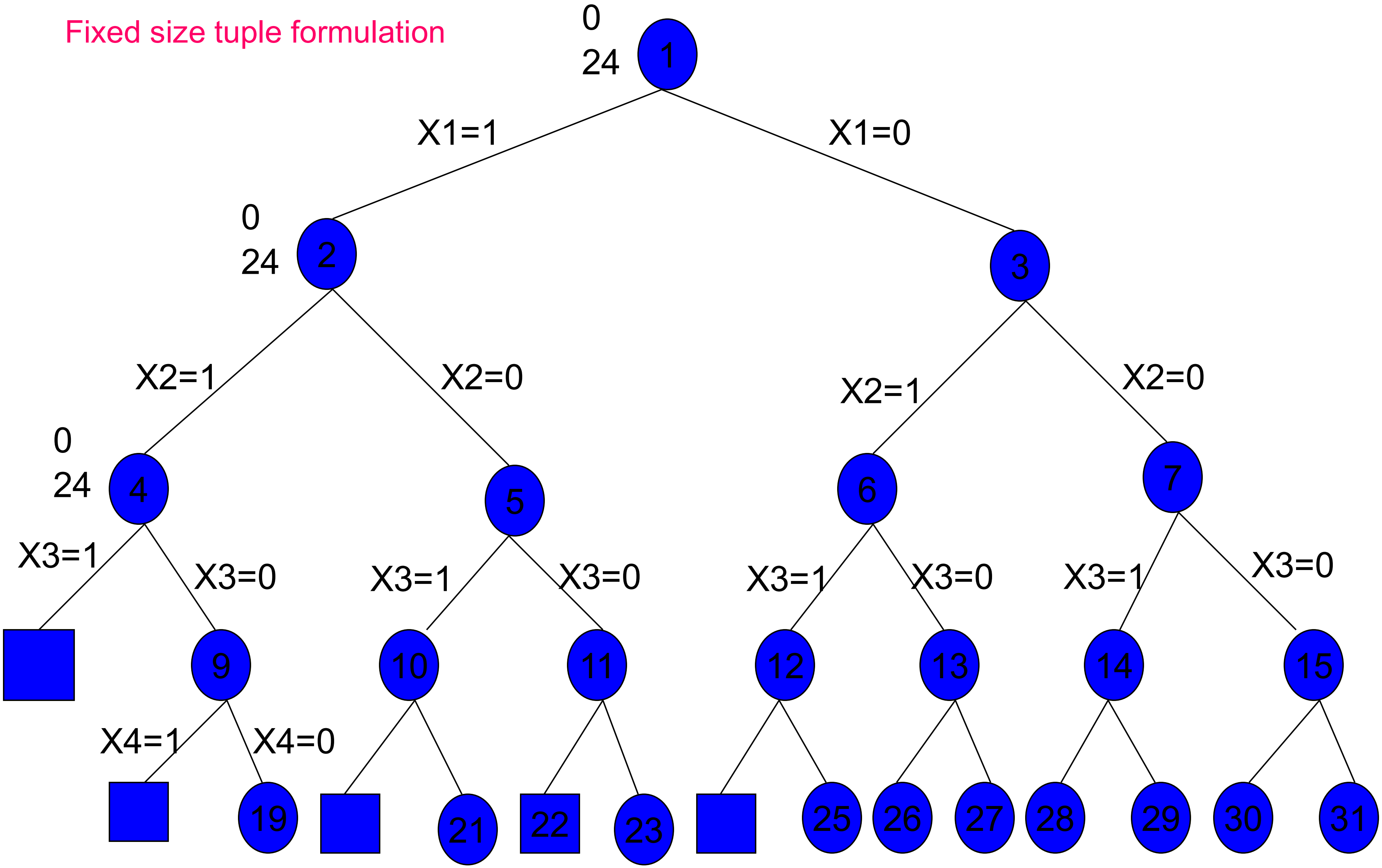
- For any circular node x, c(x) is the minimum penalty corresponding to any node in the subtree with root x.
- The value of $c(x)=\infty$ for a square node.

In the above fig c(3)=8 , c(2)=9 , and c(1)=8 etc.

Clearly c(1) is the penalty corresponding to an optimal selection j.

Job sequencing with deadlines

Fixed size tuple formulation



FIFO Branch and Bound

Refer page no.391

LIFO Branch and Bound

Do it yourself.

LC Branch and Bound

Refer page no.392

0/1 Knapsack Problem

- Generally Branch and Bound will minimize the objective function.

- The 0/1 knapsack problem is a maximization problem.

- This difficulty can be avoided by replacing the objective function $\sum p_i x_i$ by $-\sum p_i x_i$.

Note:

1. All live nodes with $\hat{c}(x)$ > upper can be killed when they are about to become E-nodes .

^

LC Branch and Bound Solution

Upper number = $\hat{c}$

Lower number = u

EX:- n=4, (p1,p2,p3,p4)= (10,10,12,18)

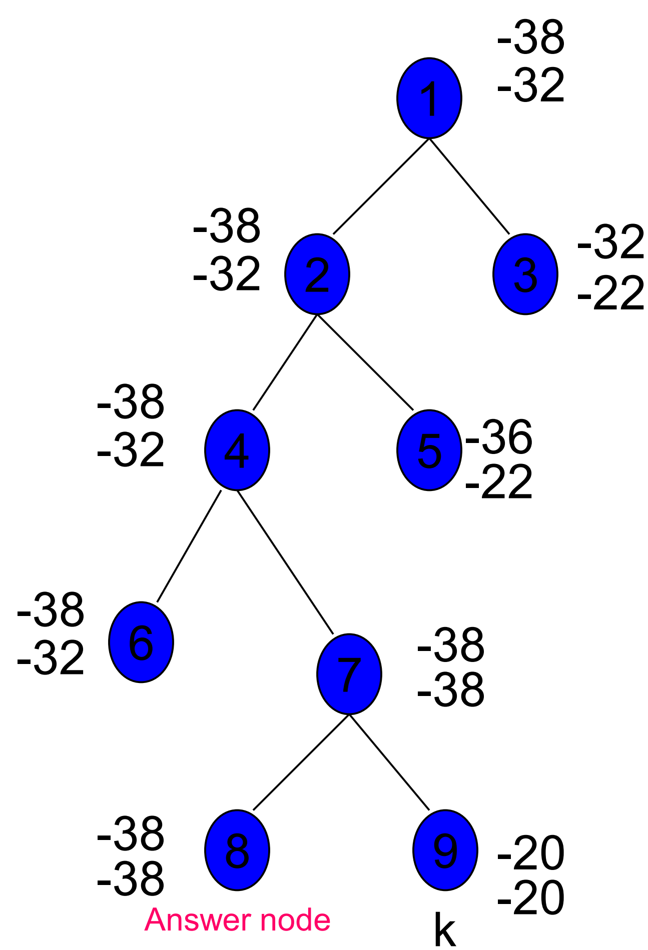
(w1,w2,w3,w4) = (2,4,6,9), m=15

Process: The calculation of U and $\hat{C}$ is as follows.

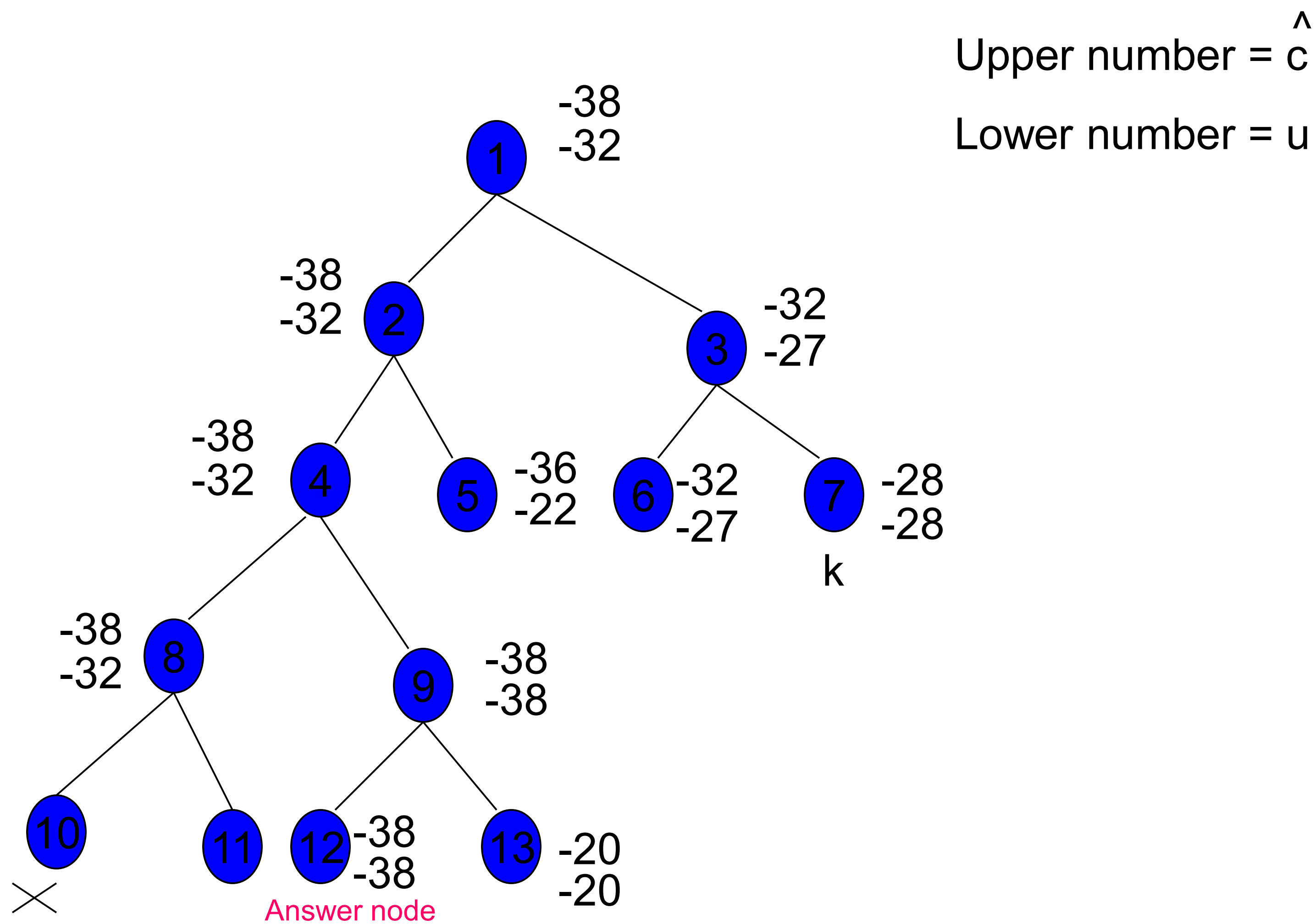
U(1) - Scan through the objects from left to right and put into the knapsack until the first object that does not fit is encontered.

$\hat{C}(1)$ – Similar to U(1) except that it also considers a fraction of the first object that does not fit the knapsack.

Continue this process until an answer node is found.



FIFO Branch and Bound



Home Work :-

1. Draw the portion of the state space tree generated by LCBB and FIFIBB for the following knapsack problem.

   a) $n=5$, $(p_1,p_2,p_3,p_4,p_5) = (10,15,6,8,4)$,
   $(w_1,w_2,w_3,w_4,w_5) = (4,6,3,4,2)$ and $m=12$

   b) $n=5$, $(p_1,p_2,p_3,p_4,p_5) = (w_1,w_2,w_3,w_4,w_5) = (4,4,5,8,9)$ and $m=15$.