BACKTRACKING

The General Method

- Problems searching for a set of solutions or which require an optimal solution satisfying some constraints can be solved using the backtracking method.
- To apply the backtrack method, the solution must be expressible as an n-tuple $(x_1 \ldots\ldots, x_n)$, where the x_i are chosen from some finite set S_i
- The problem to be solved calls for finding one vector that maximizes(or minimizes or satisfies) a criterion function $P(x_1 \ldots\ldots, x_n)$.
- The backtracking procedure works by adding the elements to the solution set, one element at a time.
- The selection of choice depends on the implicit criteria function that must be satisfied by the selected element.
- It not only reduces the computational efforts but also saves time.

- Suppose m_i is the size of set S_i. Then there are $m = m_1\ m_2\ m_3 \ldots\ m_n$ n-tuples that are possible candidates for satisfying the function P.
- The *brute force* approach would be to form all these n-tuples, evaluate each one with P, and save those which yield the optimum.
- The **Backtrack** algorithm has the ability to yield the same answer with far fewer than m trials.
- Its basic idea is to build up the solution vector one component at a time and to use modified criterion functions $P_i(x_1, x_2, \ldots x_i)$ to test whether the vector being formed has any chance of success. – sometimes called **bounding functions**.

- The major advantage of this method is this: if it is realized that the partial vector ($x_1, x_2, \ldots x_i$) can in no way lead to an optimal solution, then $m_{i+1} \ldots m_n$ possible test vectors can be ignored entirely.
- Backtracking is a methodical way of trying out various sequences of decisions, until you find one that "works".
- The problem we solve using backtracking requires that all solutions satisfy a complex set of constraints.
- The constraints can be divided into two categories
  - ❖Explicit
  - ❖Implicit

Explicit Constraints

- Explicit constraints are rules that restrict each x_i to take one value only from a given set.
- Common Examples of explicit constraints are
  - $x_i \geq 0$ or S_i={all nonnegative real numbers}
  - $x_i = 0$ or 1 or S_i={0,1}
  - $l_i \leq x_i \leq u_i$ or S_i={a: $l_i \leq a \leq u_i$}
- **Example 8-queens**
  - The explicit constraints S_i={ 1,2,3,4,5,6,7,8}
- The explicit constrains depends on the particular instance **I** of the problem being solved. All tuples that satisfy the explicit constraints define a possible ***solution space*** for **I**.

Implicit constraints

- The implicit constraints are rules that determine which of the tuples in the *solution space* of an instance **I** of a problem ***satisfy the criterion function.***
- The implicit constraints describe the way in which the x_i must relate to each other.
- **Example 8-queens**
  - Implicit constraints : no two x_i's can be the same column(ie, all queens must be on different columns) and no two queens can be on the same diagonal.

Terms Used in Backtracking

- **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.
  - For example, x_i = 0 or 1 means set S = {0,1}
- **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.
- **Solution Space**: It is the set of all tuples that satisfy the explicit constraints.

- **State space tree:** The solution space is organized as a tree called the state space tree.
- **Bounding function or criteria :** It is a function created that is used to kill live nodes without generating all its children.
- **Live node:** It is the node that has been generated, but none of its descendants are yet generated.
- **Extended node or E-node:** It is the live node whose children are currently being generated.
- **Dead node:** It is the node that is not to be extended further or all of whose children have already been generated.
- **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.
- **Solution node:** it is the node that has the possibility to become the answer node.

- A **state-space tree** is the **tree** of the construction of the solution from partial solution starting with the root with no component solution (...).
- Let B be a function from the set of vertices of the state space tree to the positive integers. Suppose that for any partial solution X

$$B(X) \geq P(X)$$

then we say B is a bounding function

BACKTRACKING (Contd..)

- The problem is to place eight queens on an 8 x 8 chess board so that no two queens attack i.e. , no two of them are on the same row, column or diagonal.
- Strategy : The rows and columns are numbered through 1 to 8.

  The queens are also numbered through 1 to 8.
- Since each queen must be on a different row without loss of generality, we assume queen i is to be placed on row i .
- The solution to the 8-queens problem be represented as 8-tuples(x_1,x_2,.....,x_8) where x_i is the column on which queen i is placed.

BACKTRACKING (Contd..)

- The explicit constraints are :

  $S_i = \{1,2,3,4,5,6,7,8\}$ $1 \le i \le n$ or $1 \le x_i \le 8$

  $i = 1,.....8$

- The solution space consists of 8^8 8- tuples.
- The implicit constraints are :

I. no two x_i's can be the same that is, all queens must be on different columns.

II. no two queens can be on the same diagonal.

III. reduces the size of solution space from 8^8 to 8! 8 – tuples.

Two solutions are (4,6,8,2,7,1,3,5) and (3,8,4,7,1,6,2,5)

BACKTRACKING (Contd..)

A[4,2]

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

Terminology

- Backtracking algorithms determine problem solution by *systematically searching* the solution space for the given problem instance.
- This is done by using a *tree organization* for the solution space.
- For a given solution space many tree organization may be possible.
- Each node in the tree defines *a problem state.*

Terminology

- All paths from the root to other node defines the *state space* of the problem.
- *Solution state* of the problem state s is the path from the root to s defines a tuple in the solution space.
- The tree organization of the solution space is referred as the *state space tree.*
- *Answer states* are those solution states s for which the path from the root to s defines a tuple that is a member of the set of solutions(ie, it satisfies the implicit constraints).

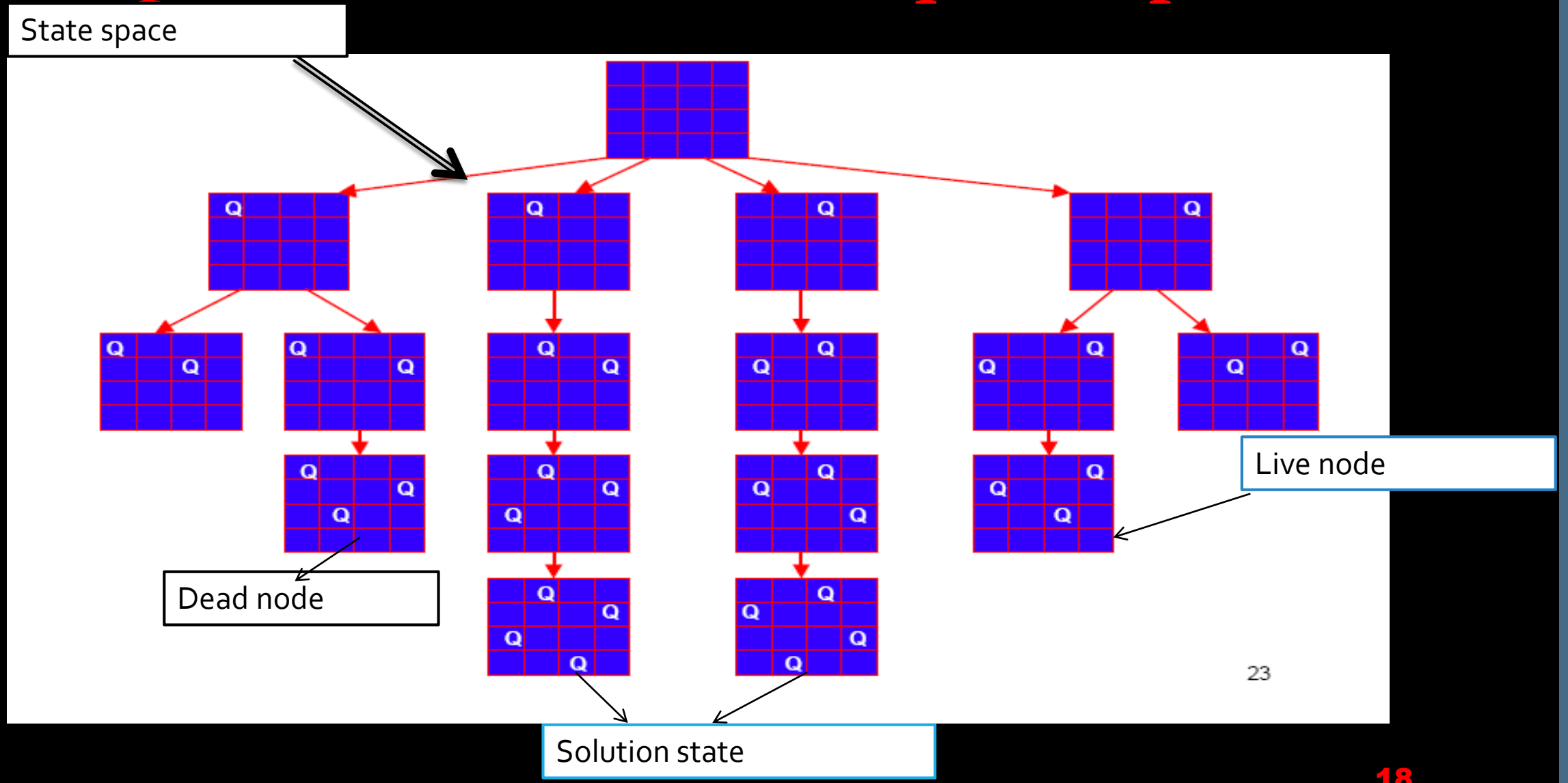
Terminology

- **Static tree** : independent of the problem instance being solved
- **Dynamic tree**: Tree organization that are problem instance is dependent
- **Live node**: a node which has been generated and all of whose children have not yet been generated
- **Dead node**: is a generated node which is not to expanded further or all of whose children have been generated.
- **E-NODE** (Node being expanded) - The live node whose children are currently being generated .

Terminology

- Bounding function –used to bound the searching in a tree. It will be used to kill live nodes without generating all their children if it does not lead to a feasible solution.
- Backtracking - is depth first node generation with bounding functions.
- Branch-and-bound is state generation method in which e-node remains e-node until it is dead.

State Space Tree of the Four-queens problem



Backtracking

- Backtracking consists of doing a DFS of the state space tree, checking whether each node is promising and if the node is non promising backtracking to the its parent.

Two methods

- Recursive
- Iterative

General Recursive Backtracking Algorithm

```
1. Algorithm Backtrack(k)
2. //This schema describes the backtracking process using recursion.
3. //On entering, the  first k-1 values x[1],x[2]……..x[k-1] of
4. //the solution vector x[1:n] have been assigned.
5. //X[] and n are global.
6. {
7.   for (each  x[k] € T(x[1],…….,x[k-1])do
8.   {
9.         if(B_k( 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.}
```

Explanation Of Recursive Algorithm

- Line 7 --- check the x_k is an element in the tree.
- B_k is a bounding function.
- Line 9 ---check whether possible elements satisfies B_k
- Line 11-12--- check it leads to answer node then adjoined to the current position
- Line 13--- algorithm recursively invoked

General Iterative Backtracking Method Algorithm

```
1.  Algorithm IBacktrack(n)
2.  //This schema describes the backtracking process.
3.  //All solutions are generated in x[1:n] and printed as soon as they are
4.  // determined.
5.  {
6.        k=1;
7.        while(k ≠ 0) do
8.        {
9.            if(there remains an untried x[k] € T(x[1],x[2],…….x[k-1] )
10.            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. }
```

Explanation for iterative backtracking method

- Line 9-10--- x[k] that satisfy the bounding function
- Line 12-14---- check x[k] leads to answer node then adjoined to the current position and goes to next set
- Line 15--- backtrack to the previous set

Efficiency of Backtracking Algorithm

- The time required by a backtracking algorithm or the efficiency depends on four factors

  (i) The time to generate the next x(k);

  (ii) The no. of x(k) satisfying the explicit constraints

  (iii) The time for bounding functions B_k

  (iv) The no. of x(k) satisfying the B_k for all k.

Efficiency of Backtracking Algorithm (Bt) (Contd..)

- The first three are relatively independent of the problem instance being solved.
- The complexity for the first three is of polynomial complexity .
- If the number of nodes generated is 2^n or n!, then the worst case complexity for a backtracking algorithm is **$O(p(n)2^n)$** or **$O(q(n)n!)$** where p(n) and q(n) is a polynomials in n .

Estimation of Nodes generated in a BT Algorithm

- We can estimate the number of nodes that will be generated by a backtracking algorithm working on a certain instance by using Monte Carlo Methods
- The general idea in the estimation method is to generate a random path in the state space tree.
- Let X be a node at level i on this path.
- The bounding functions are used at node X to determine the number m_i be the children of X (at level i+1) that do not get bounded. (i.e. m_i are the nodes which can be considered for getting an answer node).
- Choose randomly one of the m_i.
- Continue this until this node is either is a leaf or all its children are bounded.

Estimation of Nodes generated in a BT Algorithm (Contd..)

- The number of unbounded nodes on level one is 1.
- The number of unbounded nodes on level 2 is m_1
- The total no. of nodes generated till level 2 is 1 + m1

$\therefore$ The total number of nodes generated till level i+1 is $1+m_1+\ldots+m_1\ldots m_i$.

- The above procedure can be written as an algorithm.

Estimation of Nodes generated in a BT Algorithm (Contd..)

- Let m be the no. of unbounded nodes to be generated.
- Let us assume that the bounding functions are static, i.e., the BT algorithm does not change its bounding functions.
- The number of estimated number of unbounded nodes

$=1+m_1+m_1m_2+\ldots. +m_1m_2m_3..m_i$ where m_i is to be estimated no. of nodes at level i+ 1.

Algorithm Estimate()
//The algorithm follows a random path in a state space tree and produces an estimate of the number of nodes in the tree.
{
m:=1; r:=1; k:=1 ;
repeat
{
T_k ={x[k] | x[k] ∈T (x[1] ,x[2],….x[k-1]) and B_k (x[1], …., x[k]is true};
if (Size (T_k) = 0 then return m // SIZE returns the //
r := r * Size(T_k) ; m:=m+r; // size of the set T_k //
x [k]:= Choose (T_k); k:=k+1; // Choose makes a random choice of an element in T_k //
}until (false);
}

The n-Queen problem

- Place n queens on an n by n chess board so that no two of them are on the same row, column, or diagonal

- NOTES: A queen can attack horizontally, vertically, and on both diagonals, so it is pretty hard to place several queens on one board so that they don’t attack each other

The n-queens problem and solution

- In implementing the n – queens problem we imagine the chessboard as a two-dimensional array A (1 : n, 1 : n), then we observe that every element on the same diagonal that runs from the upper left to the lower right has the same row-column value.
- The condition to test whether two queens, at positions (i, j) and (k, l) are on the same row or column is simply to check i = k or j = l
- The conditions to test whether two queens are on the same diagonal or not are to be found

The n-queens problem and solution contd..

Observe that

i. For the elements in the the upper left to lower Right diagonal, the row -column values are same or row - column = 0,

e.g. 1-1=2-2=3-3=4-4=0

(1,1)	(1,2)	(1,3)	(1,4)
(2,1)	(2,2)	(2,3)	(2,4)
(3,1)	(3,2)	(3,3)	(3,4)
(4,1)	(4,2)	(4,3)	(4,4)

ii) For the elements in the upper right to the lower left diagonal, row + column value is the same e.g. 1+4=2+3=3+2=4+1=5

The n-queens problem and solution contd..

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

$$i - j = k - l \quad \text{or} \quad i + j = k + l$$

$$\text{or} \quad j - l = i - k \quad \text{or} \quad j - l = k - i$$

- Therefore two queens lie on the same diagonal if and only if

$$|j - l| = |i - k|$$

N Queen problem- Algorithm

Algorithm NQueens(k,n)

//using backtracking , this procedure prints all possible placements of n queens on an n x n *chessboard so that they are non attacking.*

```
{
        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);
                }
        }
}
```

N Queen problem- Algorithm

Algorithm Place (k , i)

*//returns **true** if a queen can be placed in the k^{th} row and i^{th} column. Otherwise it returns **false**. X[ ] is a global array whose first (k-1) values have been set. Abs(r) returns absolute value of r.*

```
    {
          for j := 1 to k - 1 do
                if (( [ X[j] = i )  or (Abs ( X[j] – i ) = Abs (j - k)))
                      then return false;
        return true;
 }
```

two are in the same column

in the same diagonal

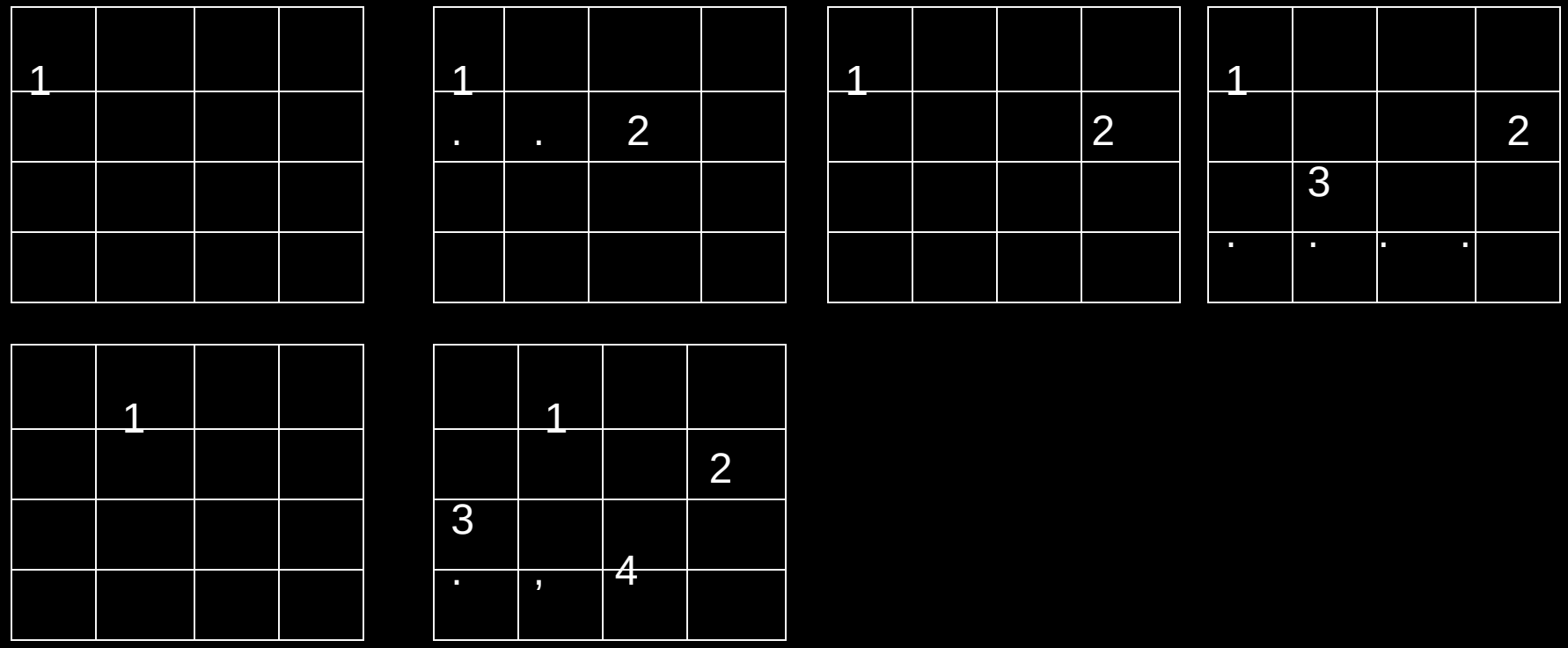
- Place(k,i) returns a boolean value that is **true** if the k^{th} queen can be placed in column i. It tests both whether i is distinct from all previous values x[1],….x[k-1] and whether there is no other queen on the same diagonal. Its computing time is O(k-1).
- Using Place we can refine the general backtracking method and gives a precise solution to the n-queens problem.
- The array x[] is global.
- The algorithm is invoked by Nqueens(1,n);

4-queen solution

- NQ(1,4)
- i=1 k=1
- Place(1,1)=true
- X[1]=1
- NQ(2,4)
- i=1,k=2,x[1]=1
- Place(2,1)=false
- Place(2,2)=false
- Place(2,3)=true
- X[2]=3
- NQ(3,4)
- Place(3,1)=false
- Place(3,2)=false
- Place(3,3)=false
- Place(3,4)=false // backtrack
- NQ(2,4)
- Place(2,4)=true
- X[2]=4
- NQ(3,4)
- Place(3,1)=false
- Place(3,2)=true
- X[3]=2;
- NQ(4,4)
- Place(41)=false
- Place(42)=false
- Place(43)=false
- Place(44)=false//backtrack
- NQ(3,4)
- Place(3,3)=false
- Place(3,4)=false//backtrack
- NQ(2,4) //backtrack
- NO(1,4)
- Place(1,2)==true
- X[2]=1
- NQ(2,4)
- Place(2,4)=true
- X[2]=4
- NQ(3,4)
- Place(3,1)=true
- X[3]=1
- NQ(4,4)
- Place(4,3)=true
- X[4]=3

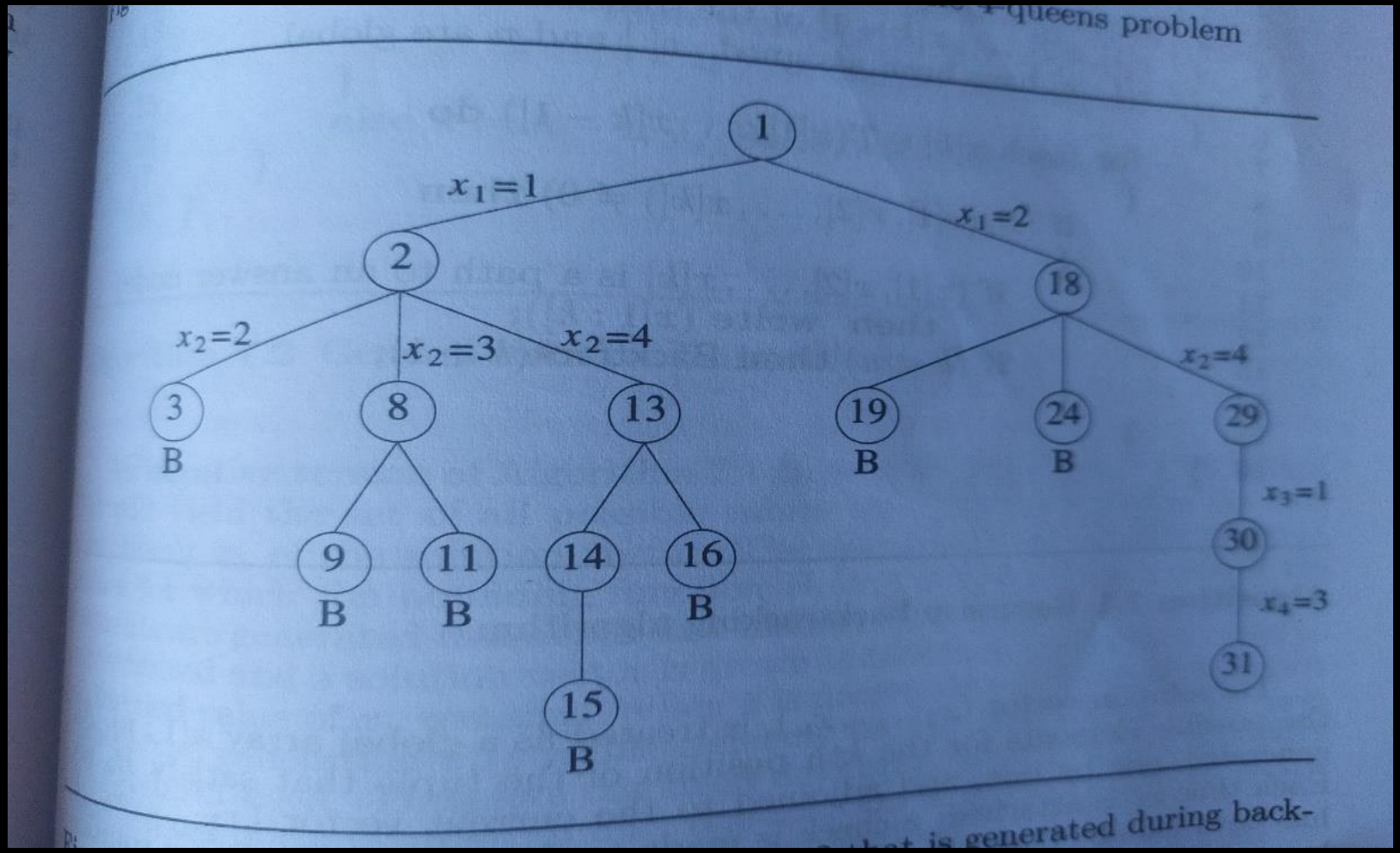
BACKTRACKING (Contd..)

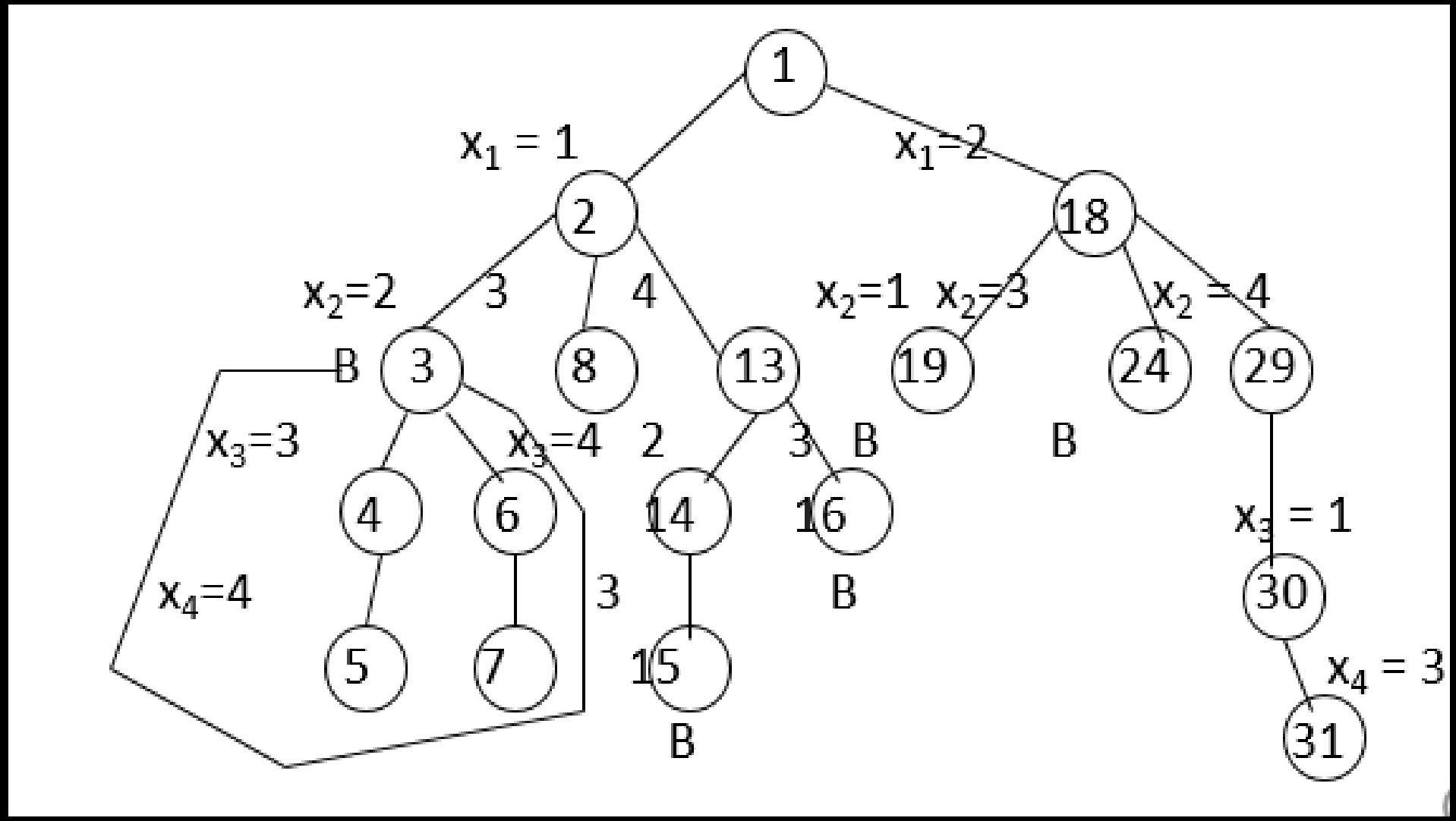
Example : 4 Queens problem



BACKTRACKING (Contd..)

Example : 4 Queens state space tree




1
x1 = 1
x1=2
2
18
x2=2
3
4
x2=1
x2=3
x2 = 4
B
3
8
13
19
24
29
x3=3
x3=4
2
3
B
B
4
6
14
16
x3 = 1
x4=4
3
B
30
5
7
15
x4 = 3
B
31

Sum of Subsets

- Suppose we are given n distinct positive numbers (usually called weights) and we desire to find all combinations of these numbers whose sums are m. This is called sum of subsets problem.

Sum of subsets

- Problem: Given positive numbers $w_i, 1 \le i \le n$, *and m, this problem calls for finding all subsets of the* w_i *whose sums are m* .
- For example: if n=4, (w_1, w_2, w_3 ,w_4)= (11,13,24,7) and m=31, then the desired subsets are (11,13,7) and (24,7)
- Rather than represent the solution vector by the w_i which sum to m, we could represent the solution vector by giving the indices of these w_i .The two solutions are described by the vectors(1,2,4) and (3,4).

Sum of subsets

- Problem: **Given *n* positive integers $w_1, \ldots w_n$ and a positive integer m. Find all subsets of $w_1, \ldots w_n$ that sum to m.**
- Example:
  n=3, m=6, and w_1=2, w_2=4, w_3=6 (2, 4, 6)
- Solutions:
  {2,4} and {6}

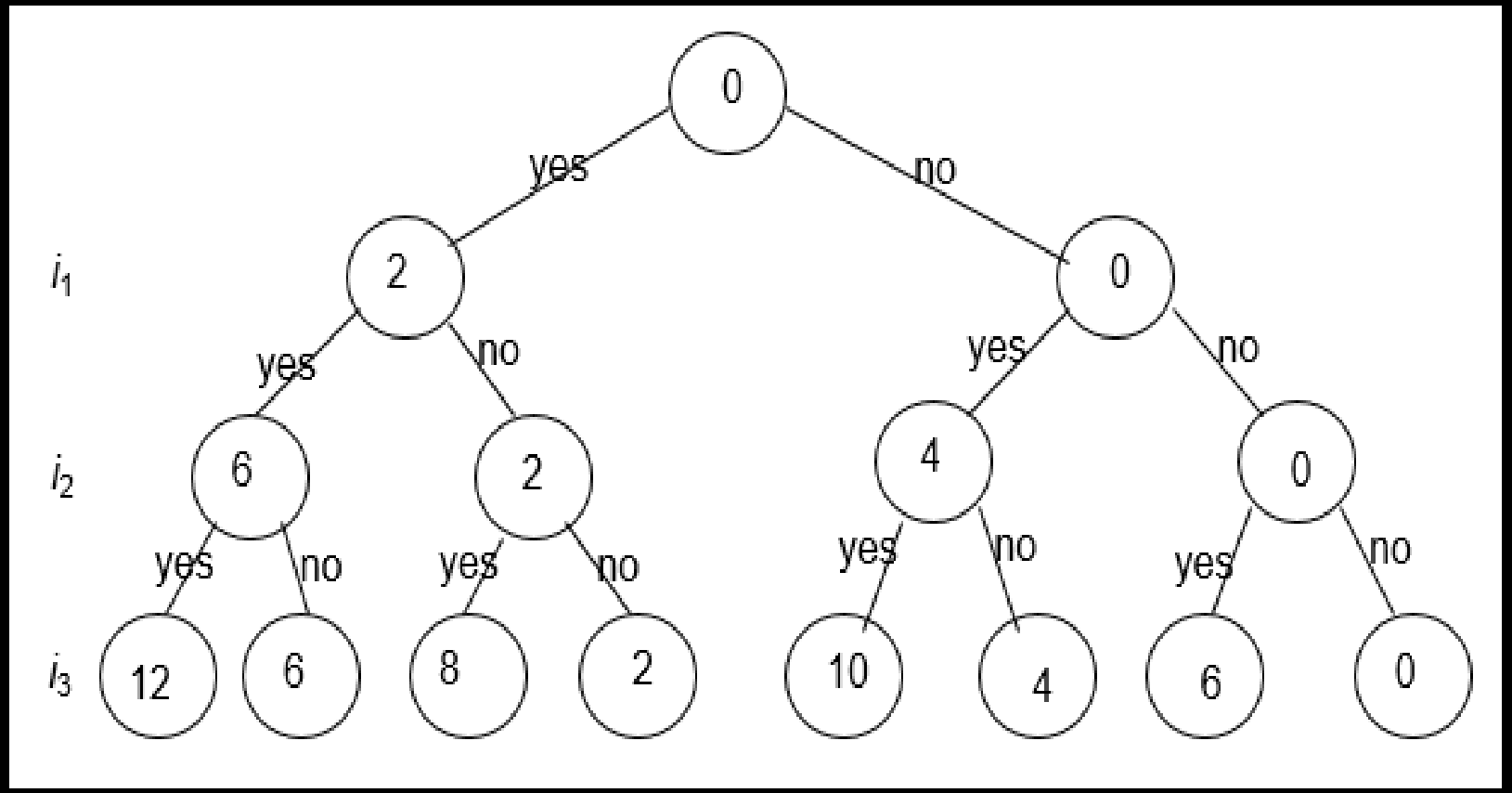
Sum of subsets

- We will assume a binary state space tree.
- The element x_i of the solution vector is either **one or zero** depending on whether the weight w_i is included or not.
- For a node at level i the left child corresponds to $x_i = 1$ and the right to $x_i = 0$
- The nodes at depth 1 are for including (yes, no) item 1, the nodes at depth 2 are for item 2, etc.
- The left branch includes w_i, and the right branch excludes w_i.
- The nodes contain the sum of the weights included so far

Sum of subsets

State Space Tree for 3 items (2,4,6)

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



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

Sum of subsets

- Bounding functions is $B_k(x_1,\ldots,x_k)$ = true iff

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

and

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

- To be able to use this "promising function" the w_i must be sorted in non-decreasing order

Sum of subsets

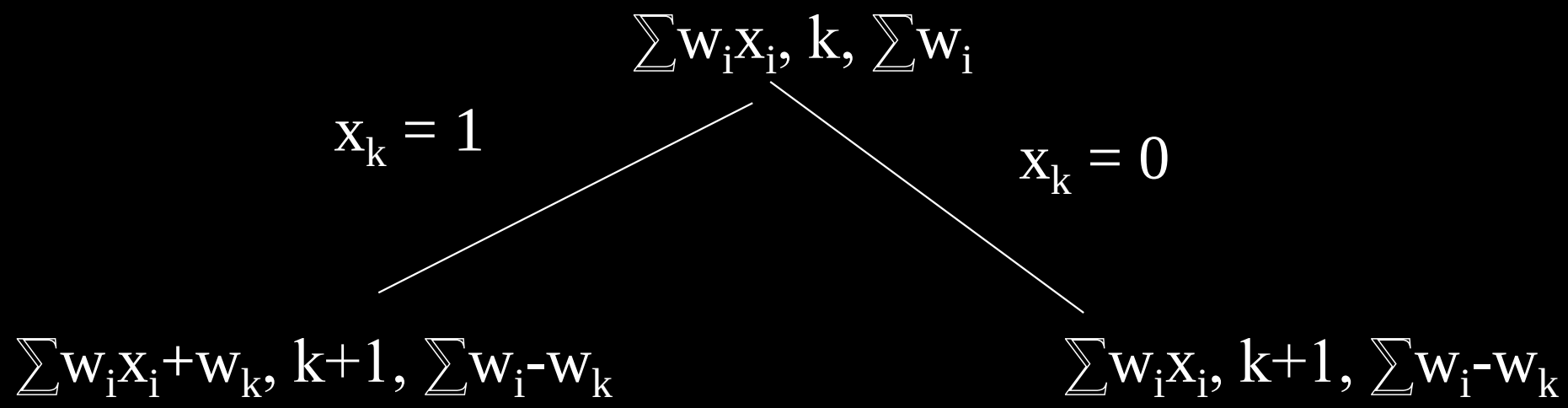
Algorithm SumOfSub(s,k,r)

// find all subsets of w[1:n] that sum to m. The values of x[j] , 1 ≤ j < k , have already been determined.

$s=\sum_{J=1}^{K-1} w[j]*x[j]$ and $r=\sum_{J=k}^{n} w[j]$. The w[j]'s are in nondecreasing order. It is assumed that w[1] ≤ m and $\sum_{I=1}^{n} w[i] \geq m$.

```
{
    // Generate left child. Note: s+ w[k] ≤ 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 ≤ j ≤ n.
    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]);
    }
}
```

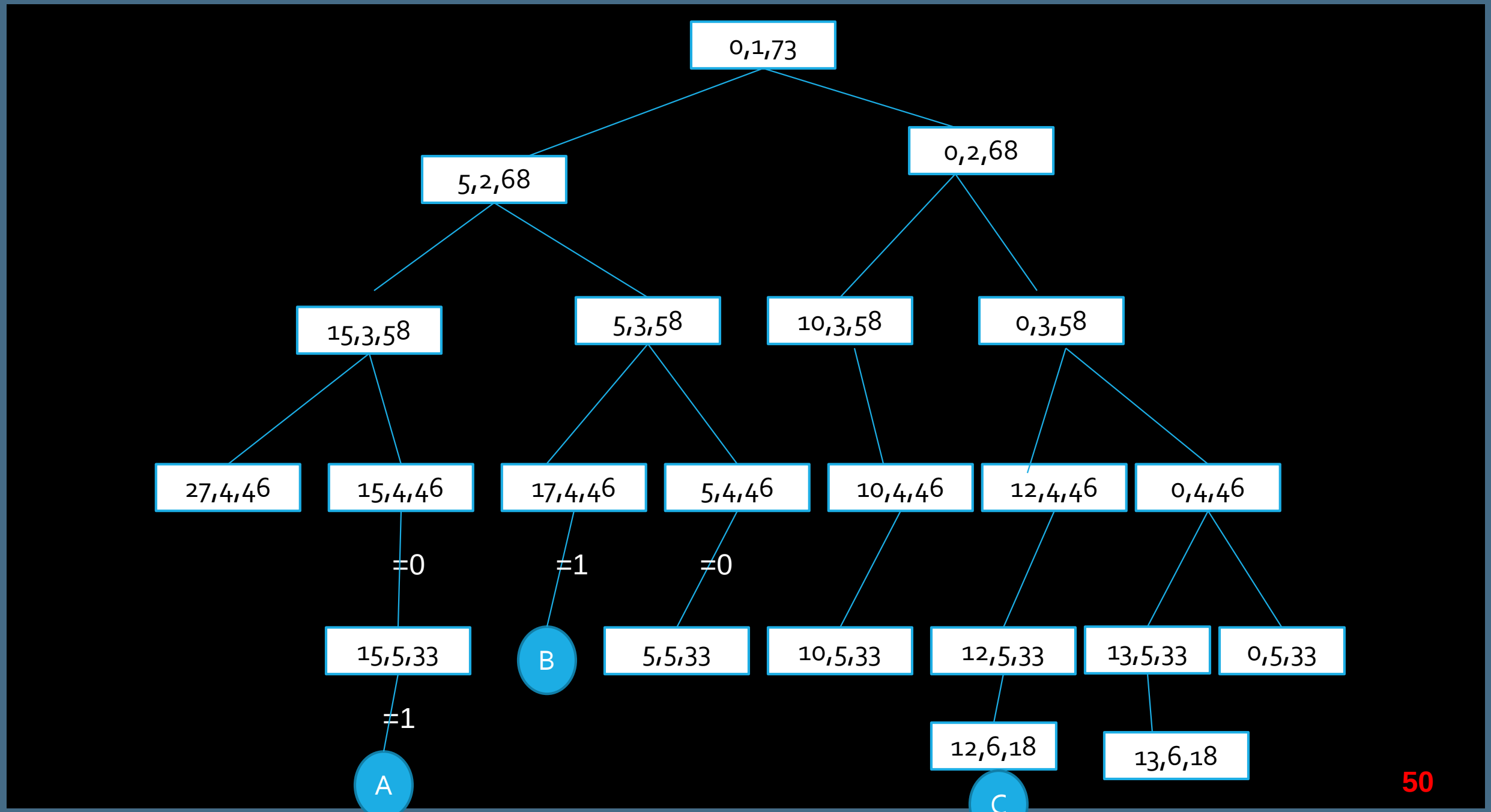
Rules followed here is:



Sum of subsets

Example 1:

Let n=6 , m=30,w[1:6]= { 5,10,12,13,15,18}. find all possible subsets of w that sum to m. do this using SumOfSub. Draw the portion of the state space tree that is generated


0,1,73
5,2,68
0,2,68
15,3,58
5,3,58
10,3,58
0,3,58
27,4,46
15,4,46
17,4,46
5,4,46
10,4,46
12,4,46
0,4,46
=0
=1
=0
15,5,33
B
5,5,33
10,5,33
12,5,33
13,5,33
0,5,33
=1
12,6,18
13,6,18
A
C

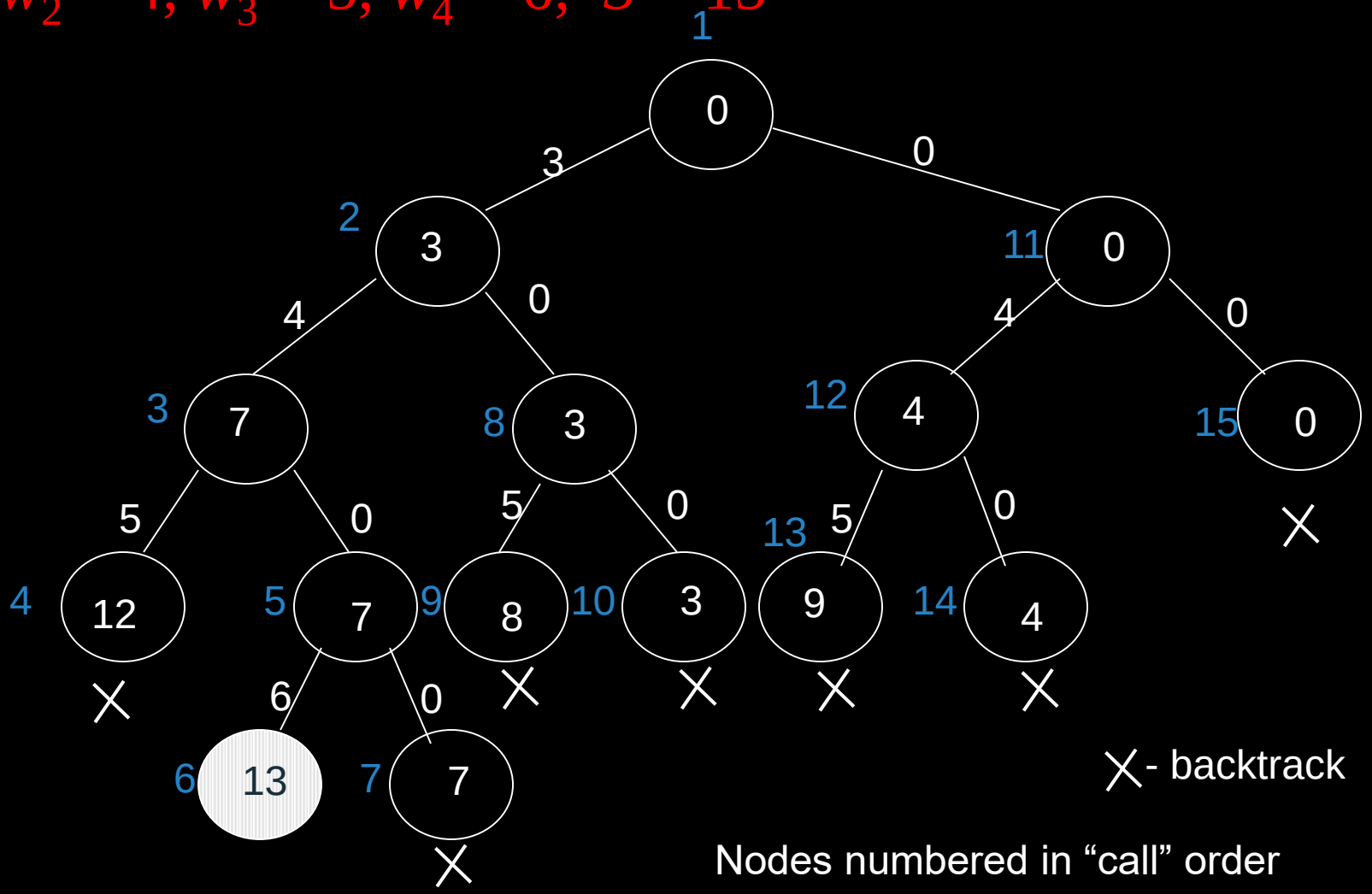
Sum of subsets

Example

Let w= { 3,4,5} and m=13. find all possible subsets of w that sum to m. do this using SumOfSub. Draw the portion of the state space tree that is generated.

A Pruned State Space Tree

$w_1 = 3, w_2 = 4, w_3 = 5, w_4 = 6;\ S = 13$



Branch And Bound

- Branch and bound is a state space search method in which all children of the E-node are generated before any other live node can become the E-node.
- Composed of two basic actions: **Branching**, which is done by using the concept of state space tree, **Bounding** which is done by using the concept of upper bound and lower bound.
- Used to compute an optimal solution of the discrete and combination optimization problems.
- Similar to backtracking that B&B also uses the concept of state-space tree to find the solution, but differs from backtracking that all children of E-node are generated before any other live node can become an E-node.
- Selection of next E-node depends on satisfying the condition of the bounding function

Branch and Bound method is first proposed by **A. H. Land** and **A. G. Doig** in 1960

Branch and Bound requires two tools

- Branching: Way of covering the feasible solution by covering several smaller feasible solution
- Bounding: Way of finding upper and lower bound for the optimal solution with-in a feasible sub-solution.
- If the lower bound of a sub problem A from the search tree is greater than the upper bound for any other previously examined sub-problem B, than A may be safely discarded from the search. This is called pruning.

- Depth-first branch-and-bound search is a way to combine the space saving of depth-first search with heuristic information.
- A breadth-first search version of branch and bound is almost identical in concept to backtracking. Except instead of a depth-first search we perform a breadth-first search. We eliminate non promising branches using the same rules as in a backtracking algorithm.
- A best-first branch and bound algorithm can be obtained by using a priority queue that sorts nodes on their lower bound.

Branch and Bound

- 3 methods
- LIFO B&B(Last in first out)
  - Also known as DFS(Depth First Search) B&B
  - Implemented using stack
  - Eg: sum of subset
- FIFO(First In First Out)
  - Also known as BFS(Breadth First Search) B&B
  - Implemented using queue
  - Eg: : 4 queens
- LC(Least Cost)
  - Implemented using priority queue
  - Eg: 15 puzzle problem

- Branch and Bound strategy can be used to solve optimization problems using the following two mechanisms:
  - A mechanism to generate branches when searching the state-space tree. This is implemented by using any of the following methods: LIFO, FIFO or LC.
  - A mechanism to generate a bound so that many branches can be terminated by killing the non promising nodes. This is implemented by using the concept of lower bound and upper bound.

Terms used in branch and Bound:

1. Live node: the node that has been generated but none of its descendants are yet generated.
2. Extended node or E-node: the live node whose children are currently being explored.
3. Dead node: the node that is not to be expanded further or all of whose children have already been generated.
4. Bounding Function: this function is used to kill live nodes without generating all its children so that it avoids generating subtrees that do not contain answer node.
5. Answer node: The node that represents the answer of the problem, which means the node at which the criteria functions are maximized, minimized or satisfied.
6. Fixed- tuple size formulation: When the elements are considered individually as whether they have to be included or not, this results in fixed-tuple formulation.

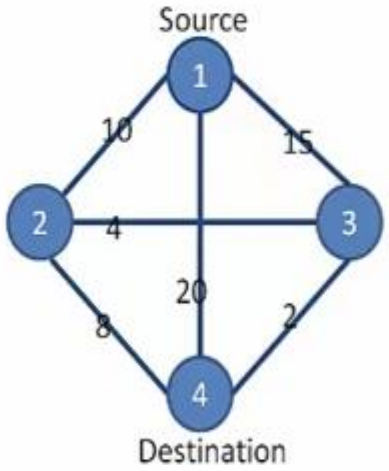
Types of Branch and Bound Techniques

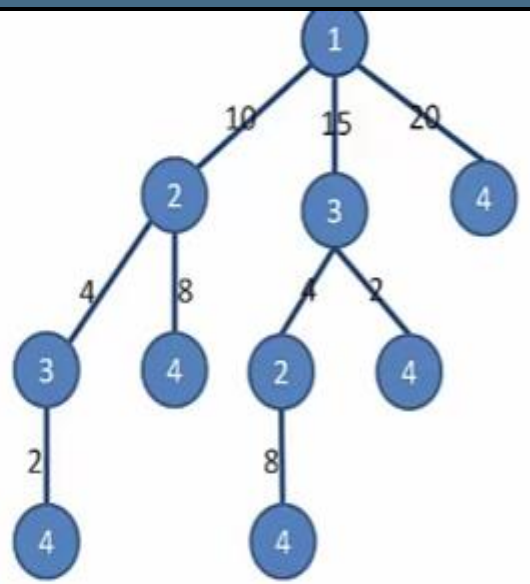
1. FIFOBB

- Branch and bound technique in which all the live nodes are stored in a QUEUE.
- The insertion is done from “rear” and the deletion is done from the “front” end of the queue.
- The state space tree is constructed using the FIFO strategies.
- The FIFO search state space is also known as the BFS state space and the corresponding algorithm as FIFOBB.

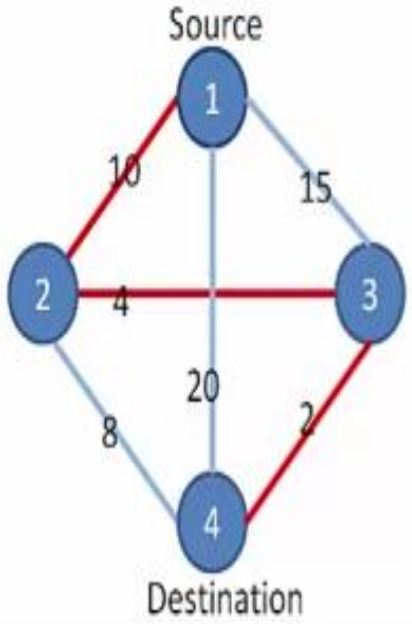
• FIFO (First-In-First-Out) Search

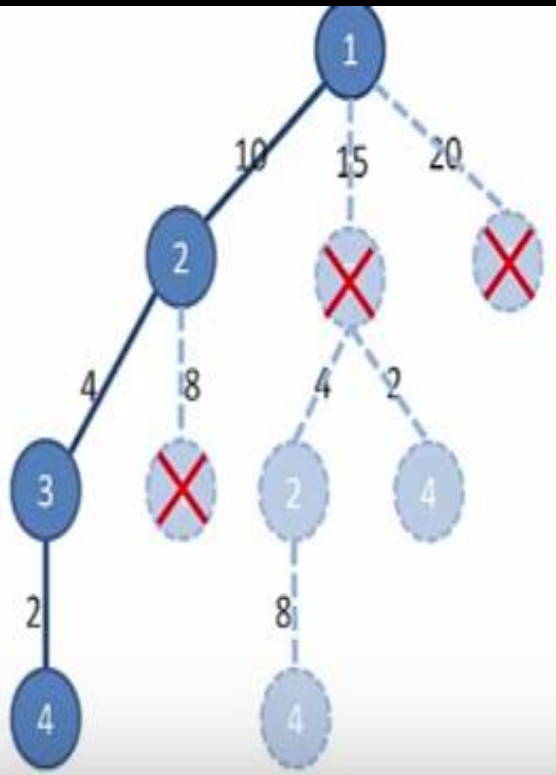
- BFS with QUEUE based Branch & Bound is called FIFO Branch & Bound.
- In FIFO search the children of the root node are generated in the first iteration.
- In the next step, children of the first child is generated.
- If the children not already killed by bounding function then put it in the queue. Then the children of second child is explored.
- Put all the children in the queue except for those children which are killed.


Source
1
10
15
2
4
3
20
8
2
4
Destination


1
10
15
20
2
3
4
4
8
4
2
3
4
2
4
2
8
4
4

~~1234342444~~


Source
1
10
15
2
4
3
20
8
2
4
Destination


1
10
15
20
2
4
8
4
2
3
2
4
2
8
4

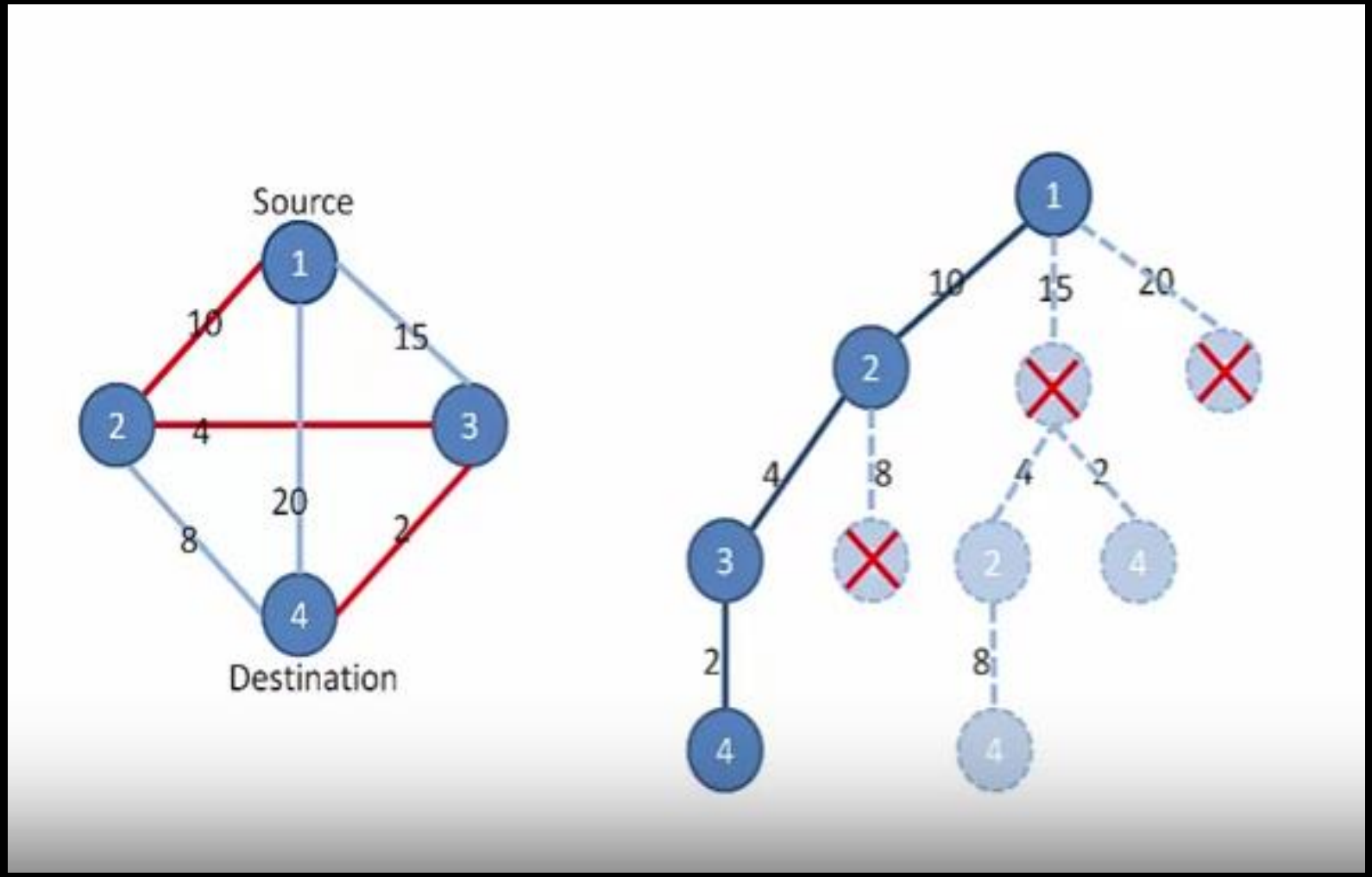
```
Initialize queue Q
Let v be the root of T
Let the current value of v be 'best'
Insert v into Q
While (Q is not empty) do
    remove node v
    for each child u of v do
        if bound(u) is better than best then
            insert u onto Q
            update the best value as value(u)
```

2.LIFOBB

- Branch and bound technique in which all the live nodes are stored in a stack.
- The element is inserted and deleted from one end only, which is known as the Top of the stack.
- The state-space tree is constructed using the LIFO strategies.
- The LIFO search state space is also known as DFS state space and the corresponding algorithm as LIFOBB.

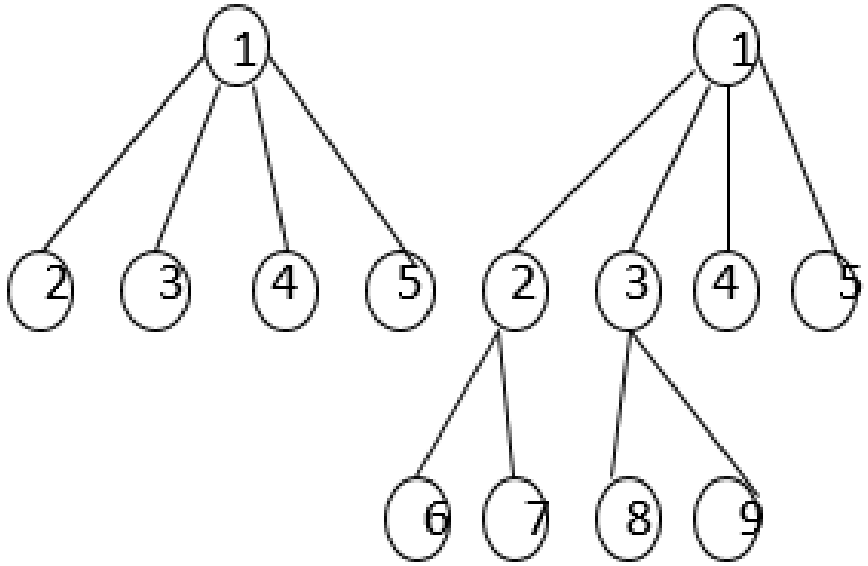
• LIFO(Last-In-Last-Out) Search

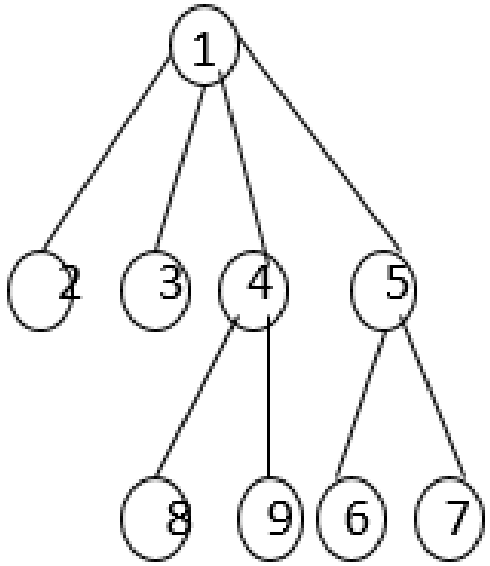
- DFS with stack based Branch & Bound is called LIFO Branch & Bound.
- In LIFO search the children of the root node are generated in the first iteration.
- In the next step, children of the first child is generated.
- If the children not already killed by bounding function then put it in the stack. Then the children of second child is explored.
- Put all the children in the stack except for those children which are killed.


Source
1
10
15
2
4
3
20
8
2
4
Destination
1
10
15
20
2
4
8
4
2
3
2
4
2
8
4

3. LCBB

- Branch and bound technique in which all the live nodes are stored in a **priority queue or heap**.
- The least cost(LC) is considered the most intelligent as it selects the next node based on a Heuristic Cost Function. It picks the one with the least cost.





Live Nodes 2,3,4,5 FIFO Branch & Bound (Breadth First Search)

LIFO Branch & Bound (D-Search)

- **Example: 4-queens(– FIFO branch-and-bound algorithm)**
- Initially, there is only one live node; no queen has been placed on the chessboard
- The only live node becomes E-node
- Expand and generate all its children; children being a queen in column 1, 2, 3, and 4 of row 1 (only live nodes left)
- Next E-node is the node with queen in row 1 and column 1
- Expand this node, and add the possible nodes to the queue of live nodes
- Bound the nodes that become dead nodes
- **– Compare with backtracking algorithm**
- Backtracking is superior method for this search problem

LC- SEARCH(Least Cost Search)

- In both FIFO and LIFO Branch & Bound, the selection rule for the next E-node is rigid/fixed.
- The rigid selection rule does not give any preference to a node which best and can search the answer node quickly.
- The expansion of all live-nodes is required before the node leading to an answer node.
- The search for an answer node can be speeded up by using a ranking function ĉ(.) for live nodes.
- The next E-node is selected on the basis of this ranking function.

LC- SEARCH

- The ideal way to assign ranks would be on the basis of additional computational effort or the additional cost needed to reach an answer node from the live node.

- For any node x, the cost could be :
  - (i) the number of nodes in the subtree x that needed to be generated before an answer node is generated.
  - (ii) The number of levels the nearest answer node is from x.

LC- SEARCH

- If cost measure (i) is used then the search would always generate the minimum number of nodes every branch and bound algorithm must generate.

- If cost measure (ii) is used, then the only nodes to become E-nodes are the nodes on the path from the root to the nearest answer node.

- The ranking function C (x) is defined as below

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

LC- SEARCH

- h(x) is the cost of reaching x from the root and f(.) is any non decreasing function.
- $\hat{g}(x)$ is the estimated additional effort or cost needed to reach an answer node from X.

- A search strategy that uses a cost function $\hat{c}(x) = f(h(x)) + \hat{g}(x)$ to select the next E-node, would always choose for its next E-node, a live node with least $\hat{c}(.)$.
- An LC-Search coupled with bounding function is called Least Cost Branch and Bound Search.

LC- SEARCH

- BFS and DFS are special cases of LC-Search
- LC search with $\hat{g}(x)=0$ and $f(h(x))$ = level of node x, then a LC-search generates nodes by levels is BFS
- LC search with $f(h(x))=0$ and $\hat{g}(x) >= \hat{g}(y)$ whenever y is a child of x is DFS.
- LC-Search with bounding functions is LC branch and bound search.

LC- SEARCH

- **Cost function**
- If x is an answer node, c(x) is the cost of reaching x from the root of state space tree
- If x is not an answer node, c(x) = 1, provided the subtree x contains no answer node
- If subtree x contains an answer node, c(x) is the cost of a minimum cost answer node in subtree x
- ˆc(·) with f(h(x)) = h(x) is an approximation to c(·)

Control abstractions for LC-Search

- Search space tree t
  - – Cost function for the nodes in t: c(·)
  - – Let x be any node in t
- c(x) is the minimum cost of any answer node in the sub tree with root x
- c(t) is the cost of the minimum cost answer node in t
  - – Since it is not easy to compute c(), we'll substitute it by a heuristic estimate as ˆc()
  - – Algorithm LC-Search uses ˆc() to find an answer node

– FIFO search
  - Implement list of live nodes as a queue
  - Least() removes the head of the queue
  - Add() adds the node to the end of the queue

– LIFO search
  – The only difference in LC, FIFO, and LIFO is in the implementation of list of live nodes

LC- SEARCH - Algorithm

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

```
1.  Algorithm LCSearch(t)
2.  // search t for an answer node
3.  {
4.      if  *t is the 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 node then
17. {
18.     write("No answer node"); return;
19. }
20. E:= Least();
21.       } until (false);
22. }
```

- Least()--- find a live node with least c(). This node is deleted from the list of live nodes and returned.
- Add(x) adds the new live node x to the list of live nodes.

Explanation

- The algorithm always keeps the list of live nodes in a list
- Line 5: root node is the E node
- Line 6 : initially the list should be empty
- Line 9: for loop examines all the children of the E-node
- Line 13: If a child of E is not an answer node, then live node and add to the list of live nodes
- Line 14: parent field set to E
- When all the children of E have been generated, E becomes a dead node
- Happens only if none of E's children is an answer node
- If there are no live nodes left, the algorithm terminates; otherwise, Least() correctly chooses the next Enode and the search continues

N^2-1 puzzle problem

The 15-puzzle

- The 15 puzzle is a well known sliding puzzle which was introduced by Sam Loyd in 1878.
- It uses a 4*4 board.
- This problem consists of 15 numbered tiles on a square frame with a capacity of 16 tiles in random order with one tile missing.
- These tiles are placed on a 4*4 chessboard
- The objective of this problem is to transform the arrangement of tiles from initial arrangement of the tiles in to the goal arrangement through a series of legal movements.
- The blank tile can move up, down, left and right and a tile can be moved only if there is an empty space next to it.

- There is always an empty slot in the initial arrangement.
- Legal moves are the moves in which the tiles adjacent to empty slot are moved to either left, right, up, down
- Each moves create a new arrangement, these arrangements are called as states of the puzzle
- The initial arrangement is called as initial state and goal arrangement is called as goal state.
- The state space tree for 15 puzzle problem is very large because there can be 16!=20.9x10^{12} different arrangements. A partial state space tree can be shown
- In the state space tree, the nodes are numbered as per the level.
- Each next move is generated based on empty slot positions.
- Edges are labelled according to the directions in which empty slot moves.

- The root node becomes the E-node
- We can decide which node to become an e-node based on estimation formula
- $\hat{c}(x)=f(x)+\hat{g}(x)$
- Where f(x) is the length of the path from the root node to node ‘x’
- $\hat{g}(x)$ is the number of non-blank tiles which are not in their goal position for node ‘x’.
- $\hat{c}(x)$ is the lower bound on the value of c(x)

The 15-puzzle

- If the board size is 2*3 , then it is called the 5-puzzle problem.
- If the size of the board is 3*3, then it is called the 8 puzzle problem.
- 35 puzzle problem involves 6*6 chess board
- The object of the puzzle is to place the tiles in order by making sliding moves that use the empty space.
- The family of puzzle is called the N^2-1 puzzle, where N^2 corresponding to the no of block on the board

The 15-puzzle

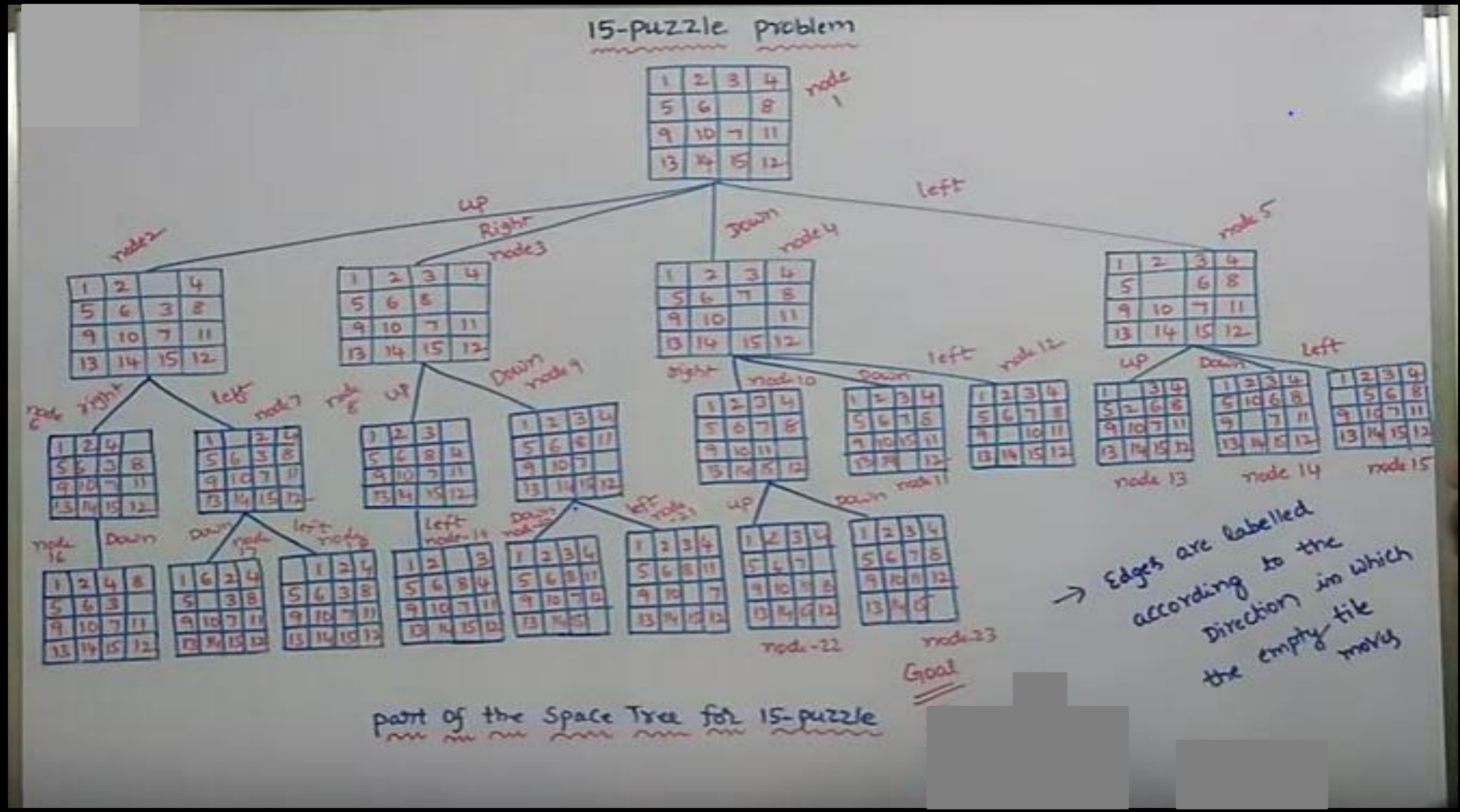
- 15 numbered tiles on a square frame with a capacity for 16 tiles
  - – Given an initial arrangement, transform it to the goal arrangement through a series of legal moves

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

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

The 15-puzzle

- Legal move involves moving a tile adjacent to the empty spot E to E
  - – Four possible moves in the initial state above: tiles 2, 3, 5, 6
  - – Each move creates a new arrangement of tiles, called state of the puzzle
  - – Initial and goal states
  - – A state is reachable from the initial state iff there is a sequence of legal moves from initial state to this state
  - – The state space of an initial state is all the states that can be reached from initial state
  - – Search the state space for the goal state and use the path from initial state to goal state as the answer
- Number of possible arrangements for tiles: $16! \approx 20.9 \times 10^{12}$



part of the Space Tree for 15-puzzle

Since node 8 has the minimum cost value, so it becomes the next E-node. Now node 8 is expanded to nodes 9 and 10 according to the up and down movement of the black tile. The cost value of node 9 is calculated as 5 and node 10 is the goal node, so the solution is found. The search space tree for this problem is shown in Fig. 20. The misplaced tiles are shown in gray shades.

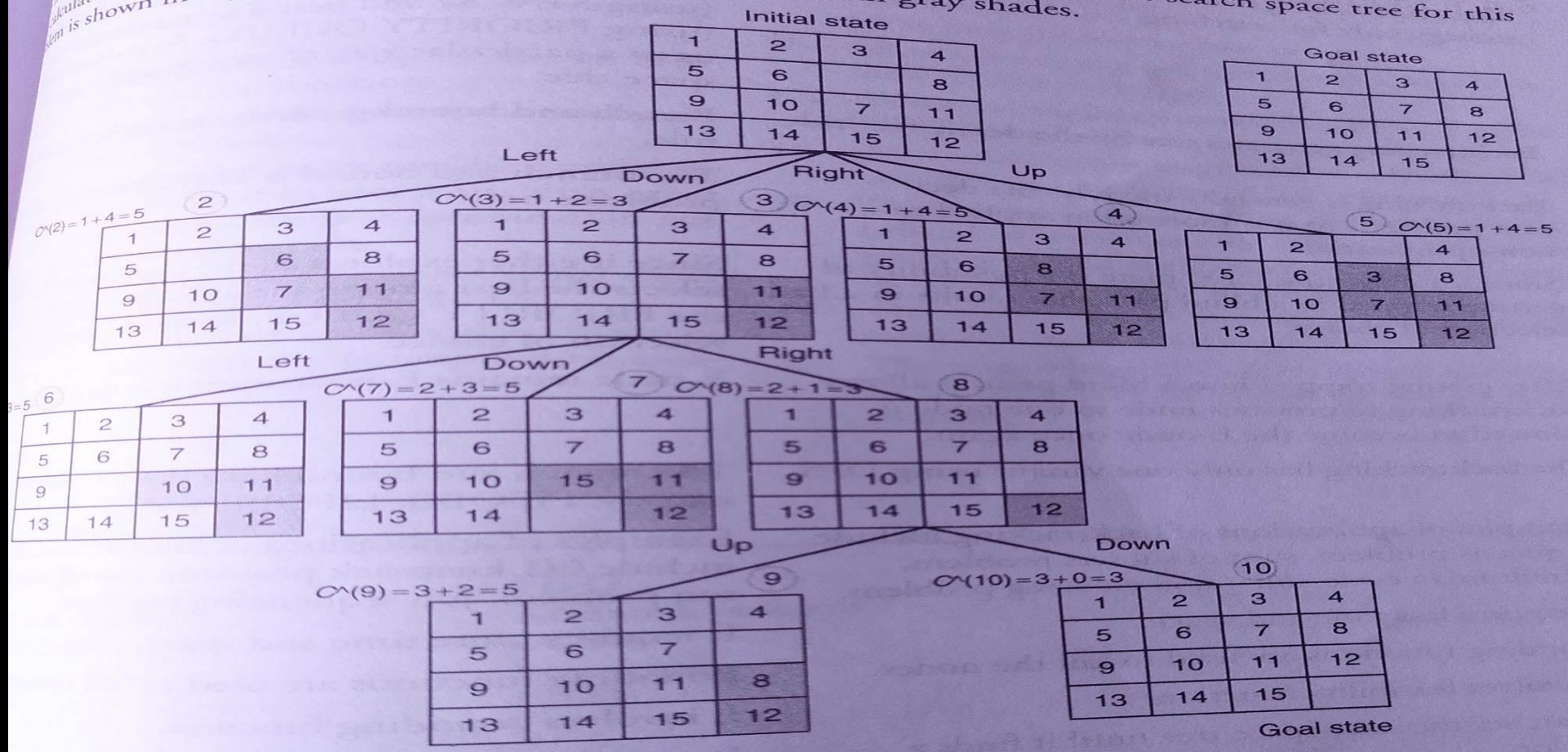
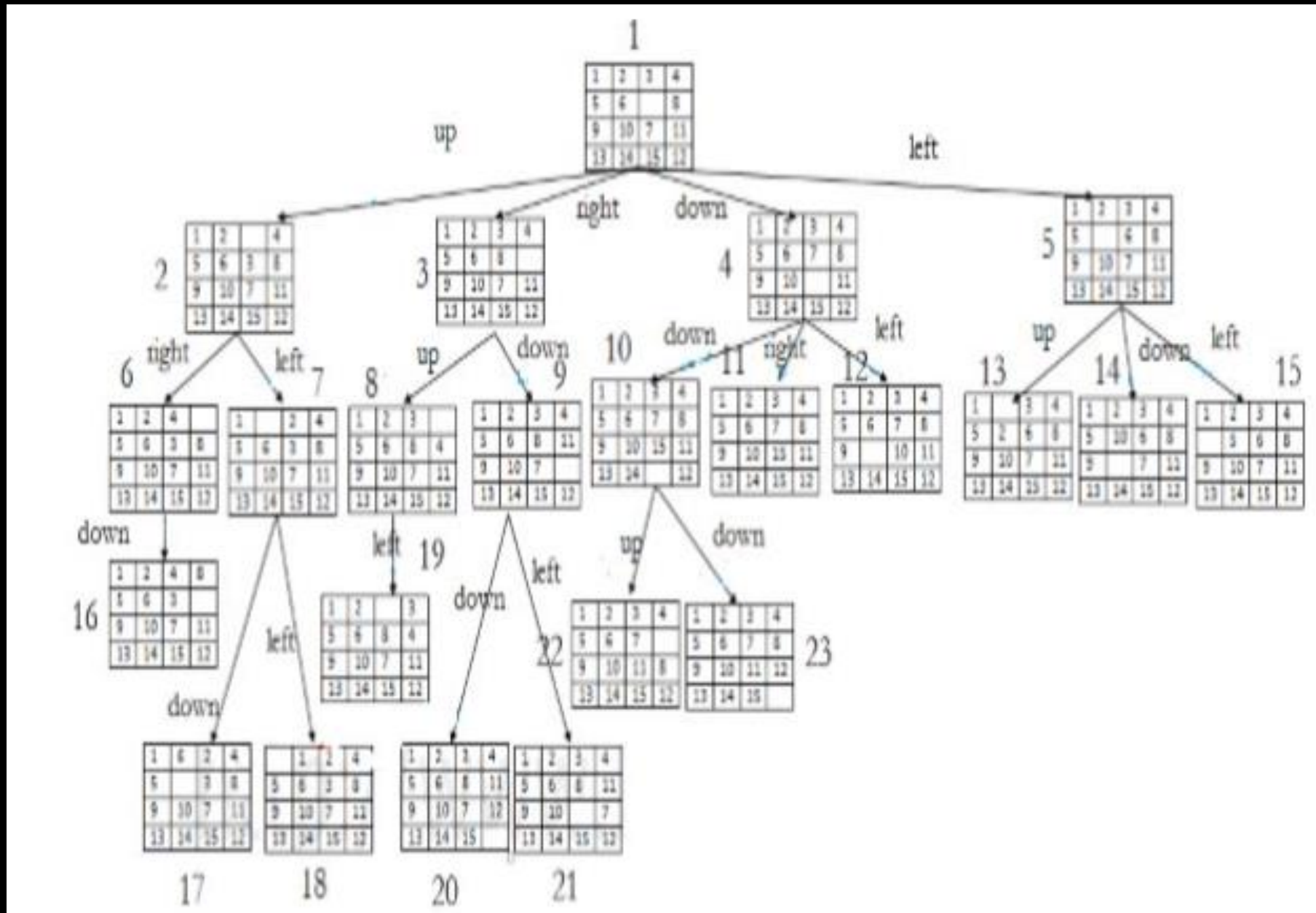


Figure 20 Search space tree for 15-puzzle problem.


1
up
left
2
3
4
5
right
down
6
right
left
7
8
up
down
9
10
down
11
right
12
left
13
up
14
down
15
left
down
16
left
19
up
down
left
down
22
23
left
down
17
18
20
21

Application of Branch and Bound

- Branch and bound algorithms can be applied to optimization problems.

- We only deal with minimization problems, because a maximization problem is easily converted into a minimization problem by changing the sign of the objective function.

Application of Branch and Bound contd..

- Searching for an optimal solution is equivalent to searching for a least cost answer node.
- The ĉ(.) function will estimate the objective function value and not the cost of reaching an answer node.
- Any node representing a feasible solution will be an answer node.
- Answer nodes and solution nodes are indistinguishable.