UNIT-IV
BACKTRACKING

→ Backtracking: *Returning to a previous point.*

GENERAL METHOD: -

→ It is a technique used to solve many difficult combinatorial problems with a large search space, by systematically trying and eliminating different possibilities.

→ Any problem which deals with searching for a set of solutions or which asks for an optimal solution satisfying some constraints can be solved using the backtracking formulation.

→ In many applications of the backtrack method, the desired solution is expressed as an n-tuple or a vector (x_1, x_2, ..., x_n), where the values for x_i are chosen from some finite set S_i.

→ The solution vector is constructed by considering one component after another.

→ After considering the first choice of the next component, the partial solution vector is evaluated against a given criterion.

(i) If the criterion is satisfied, then the next component is included into the solution vector.

(ii) If the criterion is not satisfied, we will not include that component and there is no need of considering the remaining components also. In such a case, we backtrack by considering the next choice for the previous component of the partial solution vector.

→ In this way, the size of the solution search space of a problem can drastically reduce by using backtracking method when compared to exhaustive search.

State-Space Trees:

→ The process of obtaining a solution to a problem using backtracking can be illustrated with the help of a state space tree.

→ The nodes of the state space tree are generated in the depth first order beginning at the root node.

→ The nodes reflect specific choices made for components of a solution vector. The root node represents an initial state before the search for a solution begins.

→ The root node is at first level. The nodes at second level represent the choices made for the first component of a solution vector. The nodes at third level represent the choices made for the second component of a solution vector-, and so on.

→ The root node is considered both a live node and an E-node (i.e., expansion node). At any point of time, only one node is designated as an E-node. From the E-node, we try to move to (or, generate) a new node (i.e., child of E-node).

→ If it is possible to move to a child node from the current E-node (i.e., if there is any component yet to be included in the solution vector) then that child node will be generated by adding the first legitimate choice for the next component of a solution vector and the new node becomes a live node and also the new E-node. The old E-node remains as a live node.

→ At the newly generated node, we apply the constraint function (or, criterion) to determine whether this node can possibly lead to a solution. If it cannot lead to a solution (i.e., if it doesn't satisfy constraints) then there is no point in moving into any of its subtrees and so this node is immediately killed and we move back (i.e., backtrack) to the most recently seen live node (i.e., its parent) to consider the next possible choice for the previous component of a solution vector. If there is no such choice, we backtrack one more level up the tree, and so on. The final live node becomes new E-node.

→ Finally, if the algorithm reaches a complete solution, it either stops (if just one solution is required) or continues searching for other possible solutions.

<u>N-QUEENS PROBLEM: -</u>

→ We are given an *nxn* chessboard and we need to place *n* queens on the chess board such that they are non-attacking each another (i.e., no two queens should lie on the same row, or same column, or same diagonal).

→ Due to the first two restrictions, it is clear that each row and column of the board will have exactly one queen.

→ Let us number the rows and columns of the chessboard *1* through *n*. The queens can also be numbered *1* through *n*.

→ Since, each queen must be on a different row, we can assume that queen *i* will be placed on row *i*.

→ Each solution to the *n-queens* problem can therefore be represented as an n- tuple $(x_1, x_2, ..., x_n)$ where *xi* is the column number on which queen *i* is placed.

→ The x_i values should be distinct since no two queens can be placed on the same column.

<u>HOW TO TEST WHETHER TWO QUEENS ARE ON SAME DIAGONAL?</u>

→ We observe that all the elements on the same diagonal that runs from the upper-left to the lower-right have the same *row-column* value.

→ Also, all the elements on the same diagonal that runs from upper-right to lower-left have the same *row+column* value.

→ Suppose two queens are placed at positions *(i, j)* and *(k, l)*. Then they are on the same diagonal only if

$i-j = k-l$ **(or)** $i+j = k+l$

By rearranging the terms, we have

$j-l = i-k$ **(or)** $j-l = k-i$

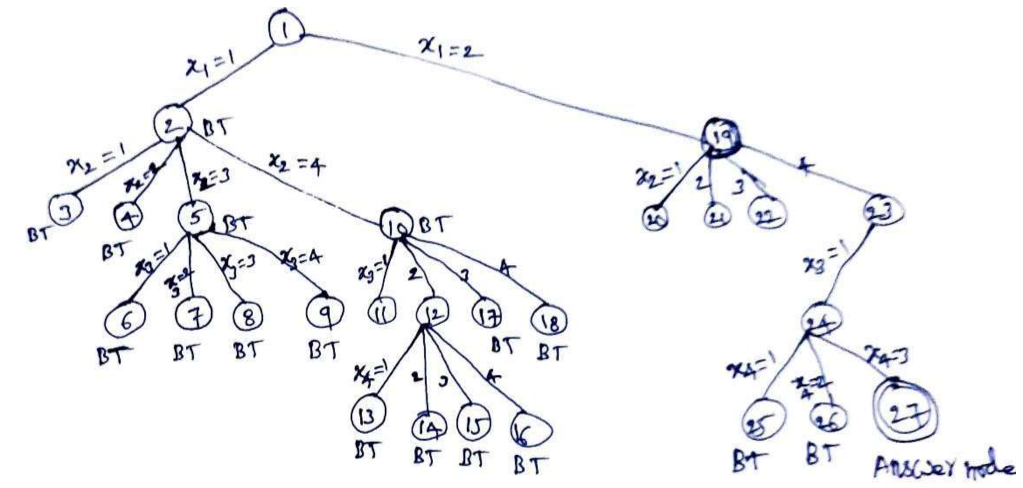
→ Therefore, two queens lie on the same diagonal if and only if $|j-l| = |i-k|$.

Algorithm for n-queens problem using backtracking:

```
Algorithm NQueens (k, n)
// using backtracking, this procedure prints all possible placements of n queens on
//an n×n chessboard so that they are non-attacking.
{
    for i := to n do   // for each column i from 1 to n
    {
        if (place(k, i)) then // check whether $k^{th}$ queen can be placed on column i
        {
                x[k] :=i;
                if(k=n) then write(x[1: n]);
                else NQueens(k+1, n);
        }
    }
}
Algorithm place(k, i)
//returns true if a queen can be placed in $k^{th}$ 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;
}
```

→ The **NQueens** algorithm is invoked as NQueens(1, n).

Portion of the State-space tree generated for solving 4-Queens problem: -



one of the solutions is:

$(x_1, x_2, x_3, x_4) = (2, 4, 1, 3)$

	1	2	3	4
1		Q1		
2				Q2
3	Q3			
4			Q4	

<u>Graph Coloring (or) m-coloring problem: -</u>

Given an undirected graph *G* and a positive integer *m*, determine if the graph can be colored with at most *m* colors such that no two adjacent vertices of the graph have the same color. This is called *m-coloring problem*. This is also known as *m-colorability decision problem.*

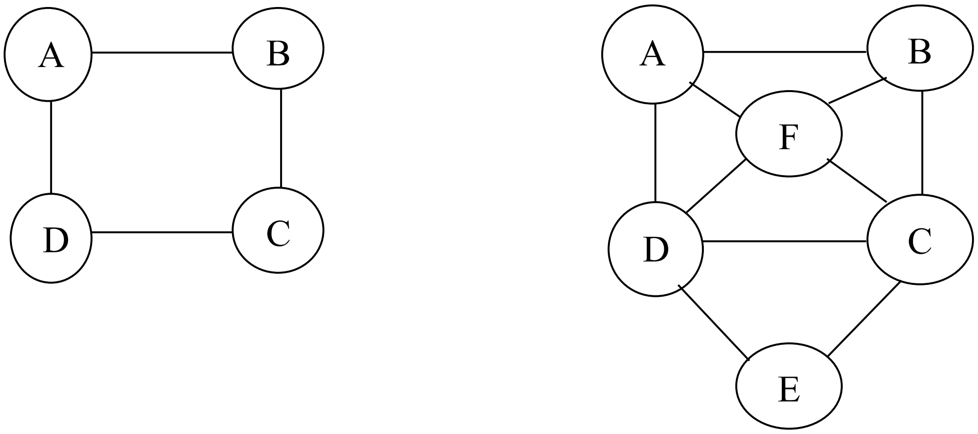
→ Here coloring of a graph means the assignment of colors to all vertices.

→ If the solution exists, then display which color is assigned to which vertex.

<u>Side points:</u>

1. <u>*m-colorability optimization problem:*</u> -
This problem asks for the smallest integer *m* for which the graph G can be colored. This integer is referred to as the chromatic number of the graph.
2. <u>*Chromatic number of a graph*</u>*:* - Minimum number of colors required to color a given graph such that no two adjacent vertices have same color.
3. If *d* is the maximum degree in a given graph, then its *chromatic number ≤ (d+1).*

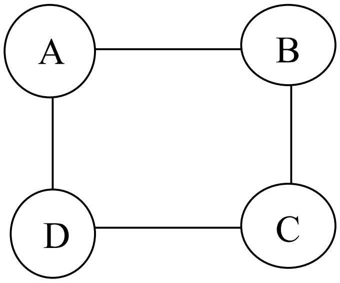
<u>Example:</u> Find the chromatic number of the following graphs.



→ Let us number the vertices of the graph *1* through *n* and the colors *1* through *m.*

→ So, a solution to the graph coloring problem can be represented as an *n*-tuple *(x1, x2, ..., xn)* where *xi* is the color of vertex *i*.

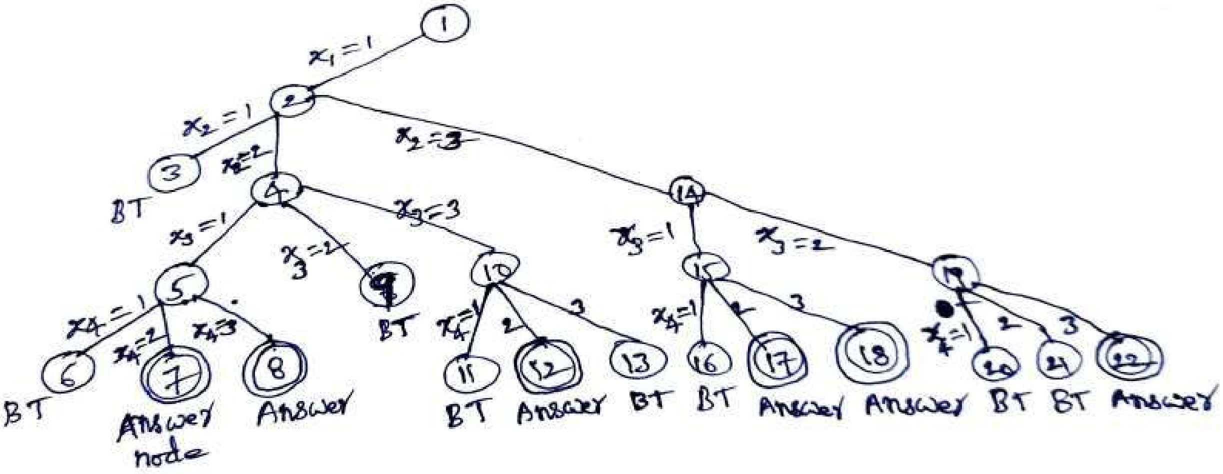
Problem: Find all possible ways of coloring the below graph with m=3.



Solution:

Let A=1, B=2, C=3, and D=4.

Portion of State-space tree generated for solving 3-coloring problem: -



Some of the solutions are: (x1, x2, x3, x4) = (1, 2, 1, 2)
= (1, 2, 1, 3)
= (1, 2, 3, 2)
= (1, 3, 1, 2)
= (1, 3, 1, 3)
= (1, 3, 2, 3)

Algorithm for m-coloring problem using backtracking:

→ Suppose we represent a graph by its adjacency matrix G[1:n, 1:n] where G[i, j]=1 if (i, j) is an edge and G[i, j]=0 otherwise.

→ Initially, the array x[] is set to zero.

```
Algorithm mColoring(k)
// This algorithm was formed using the recursive backtracking scheme.
// The graph is represented by its Boolean adjacency matrix G[1:n,1:n].
// All possible assignments of 1, 2, ..., m to the vertices of the graph are printed,
// such that adjacent vertices are assigned distinct integers.
// k is the index of the next vertex to color.
{
        while(TRUE)
        {
             // Generate all legal assignments for x[k].
             NextValue(k);   // Assign a legal color to x[k].
             if (x[k] = 0) then return;   //No new color is 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);
        }
}
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.
{
        while(TRUE)
        {
             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 an edge and if adjacent
                                                   // vertices have the same color.
                then break;
             }
             if (j=n+1) then return;   // New color found
        }   // Otherwise try to find another color.
}
```

→ **mColoring** algorithm is initially invoked as mColoring(1).

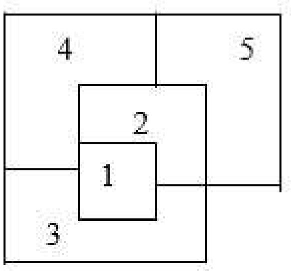
Planar graph:

→ A graph is said to be planar iff it can be drawn on a plane in such a way that no two edges cross each other.

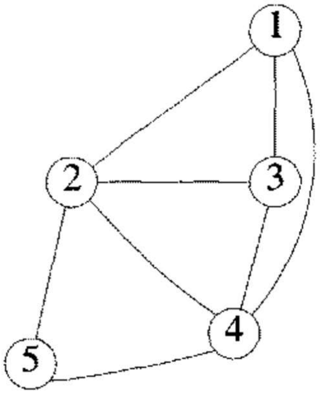
→ ***The chromatic number of a planar graph is not greater than 4.***

→ Suppose we are given a map, then it can be converted into planar graph as follows: Consider each region of the map as a node. If two regions are adjacent, then the corresponding nodes are joined by an edge.

→ Consider the following map with five regions:



The corresponding planar graph is:



HAMILTONIAN CYCLES: -

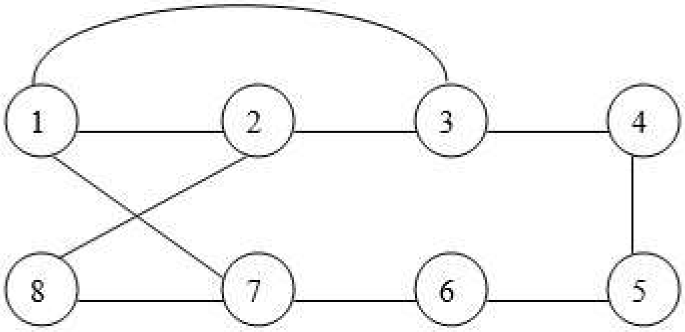
→ Let $G = (V, E)$ be a connected graph with n vertices.

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

→ In other words, if a Hamiltonian cycle begins at some vertex $v_1 \epsilon G$ and the vertices of G are visited in the order $v_1, v_2, \ldots, v_{n+1}$, then the edges (v_i, v_{i+1}) are in E, $1 \leq i \leq n$, and v_i are distinct except for v_1 and v_{n+1} which are equal.

Example:

Consider the following graph:

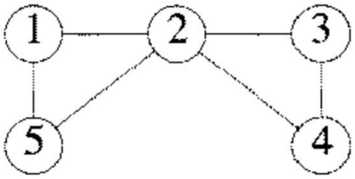


This graph has the following Hamiltonian cycle:

1—3—4—5—6—7—8—2—1

Note: If a graph has an articulation point, then there will be no Hamiltonian cycles.

Example: The following graph doesn't contain Hamiltonian cycle.



→ We can write a backtracking algorithm that finds all the Hamiltonian cycles in a graph. We will output only distinct cycles.

→ We assume that the vertices of the graph are numbered from 1 to n.

→ The backtracking solution vector *(x_1, x_2, ..., x_n)* is defined so that x_i represents the *i^{th}* visited vertex of the proposed cycle.

→ The graph is represented by its adjacency matrix *G[1:n, 1:n]*.

→ *x[2:n]* are initialized to zero. And, *x[1]* is initialized to 1 because we assume that cycles start from vertex 1.

→ For $2 \leq k \leq n-1$, x_k can be assigned any vertex *v* in the range from *1* to *n* provided it is distinct from $x_1, x_2, ..., x_{k-1}$ and there exists an edge between *v* and x_{k-1}.

→ Now, x_n can be assigned the remaining vertex, provided there exists an edge to it from both x_1 and x_{n-1}.

Algorithm for Hamiltonian Cycles problem using backtracking:

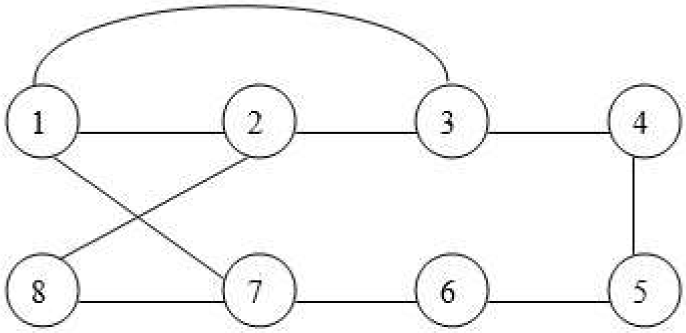
```
Algorithm Hamiltonian(k)
// This algorithm uses the recursive formulation of backtracking to find all the
// Hamiltonian cycles of a graph.
// The graph is stored as an adjacency matrix G[1:n, 1:n].
// All cycles begin at node 1.
{
    while(TRUE)
    {
        // Generate values for x[k].
        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);
    }
}
```

```
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].
{
    while(TRUE)
    {
```

```
                x[k] := (x[k]+1) 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        // Check for distinctness.
                        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;
                }
        }
}
```

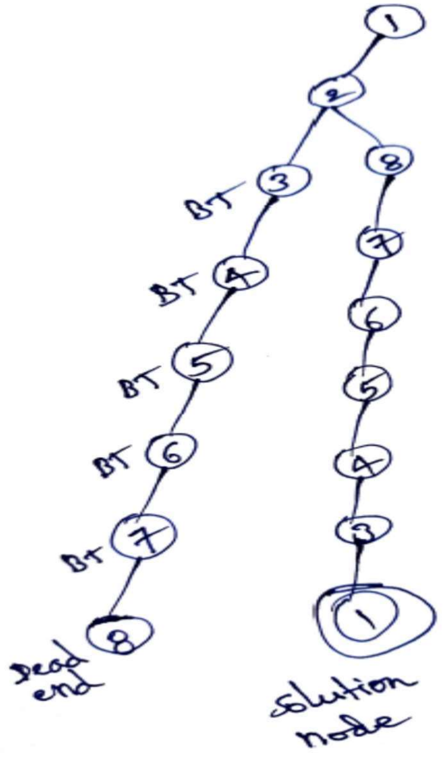
→ The algorithm **Hamiltonian** is initially invoked as Hamiltonian(2).

Example: **Find the Hamiltonian cycles for the following graph using backtracking.**

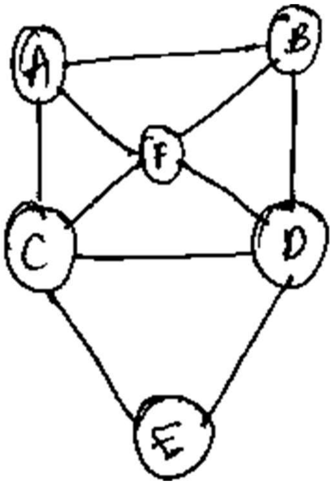


Solution:

Portion of State-space tree generated for solving Hamiltonian cycles problem: -

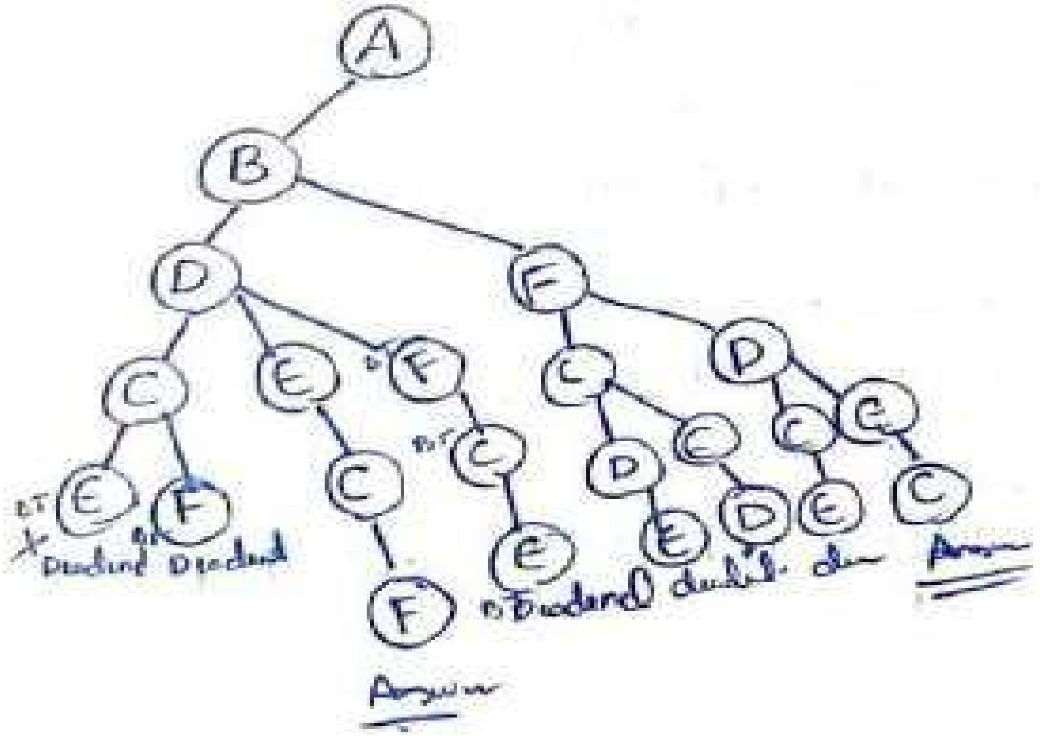


Example: **Find the Hamiltonian cycles for the following graph using backtracking.**



Solution:

<u>Portion of State-space tree generated for solving Hamiltonian cycles problem: -</u>



SUM OF SUBSETS: -

→ Suppose we are given *n* distinct positive numbers (usually called weights) w_i, $1 \leq i \leq n$ and we need to find all combination (subsets) of these numbers whose sums are equal to a given integer *m*.

→ Each solution subset is represented by an *n*-tuple $(x_1, x_2, \ldots, x_n)$ such that $x_i \in \{0,1\}$, $1 \leq i \leq n$.

If w_i is not included in subset, then $x_i = 0$.

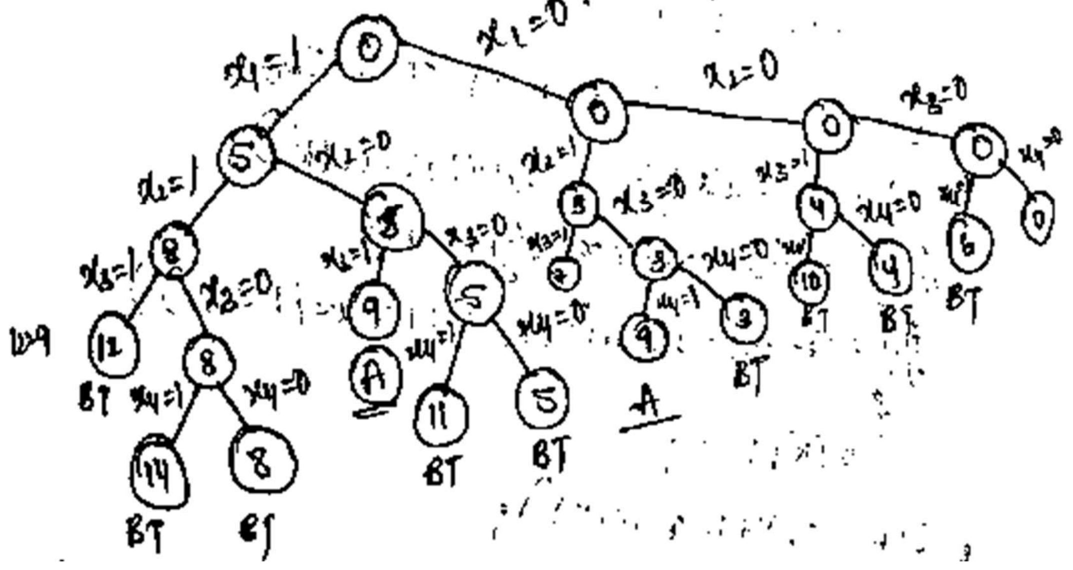
If w_i is included in subset, then $x_i = 1$.

<u>Example:</u> For $n=4$, $(w_1, w_2, w_3, w_4) = (5,3,4,6)$, *and* $m=9$,

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

Size of solution search space is $2 \times 2 \times 2 \times 2 \times \ldots$ *n* times $= 2^n$.

Inefficient backtracking solution:



Note: In the above figure, the number in the circle denotes the sum of the weights considered till now.

EFFICIENT BACKTRACKING SOLUTION FOR SUM OF SUBSETS: -

→ We assume that w_i*'s* are initially in non-decreasing order.

→ At each stage, we have two choices for x_i*'s,* i.e., *0* and *1*.

→ In the state space tree, for a node at level *i*, left child corresponds to $x_i=1$, right child corresponds to $x_i=0$.

→ Suppose we have fixed the values of $x_1, x_2, \ldots, x_{k-1}$. Now, if we choose $x_k=1$, then the following constraints have to be satisfied:

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

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

→ If the above constraints are not satisfied then we will not proceed further in state space tree and we will backtrack and make $x_k=0$.

→ We will use two variables *s* and *r* as follows:

$s=\sum_{i=1}^{k} w_i x_i$

$r=\sum_{i=k+1}^{n} w_i$

Algorithm for Sum of Subsets problem using backtracking:

→ The algorithm is initially invoked as SumOfSub(0, 1, $\sum_{i=1}^{n} w_i$).

→ The solution vector $(x_1, x_2, \ldots, x_n)$ is initialized to zero.

```
Algorithm SumOfSub(s, k, r)
// find all subsets of w[1: n] that sum to m.
// The value of x[j], 1≤ j≤ k-1 have already been determined.
// At this point of time, s=$\sum_{j=1}^{k-1} w[j]x[j]$ and r=$\sum_{j=k}^{n} w[j]$ .
// w[j]'s are in non-descending order.
// It's assumed that w[1]≤ m and $\sum_{i=1}^{n} w[i] \geq m$
{
        // generate left child
        x[k]:=1;
         if(s+w[k]=m) then        // subset found
                write(x[1: n]);
        else if(s+w[k]+w[k+1] ≤ m) then
                SumOfSub(s+w[k], k+1, r-w[k]);
        // otherwise generate right child.
        if((s+r-w[k] ≥ m) and (s+w[k+1] ≤ m)) then
        {
                x[k]:=0;
                SumOfSub(s, k+1, r-w[k]);
        }
}
```

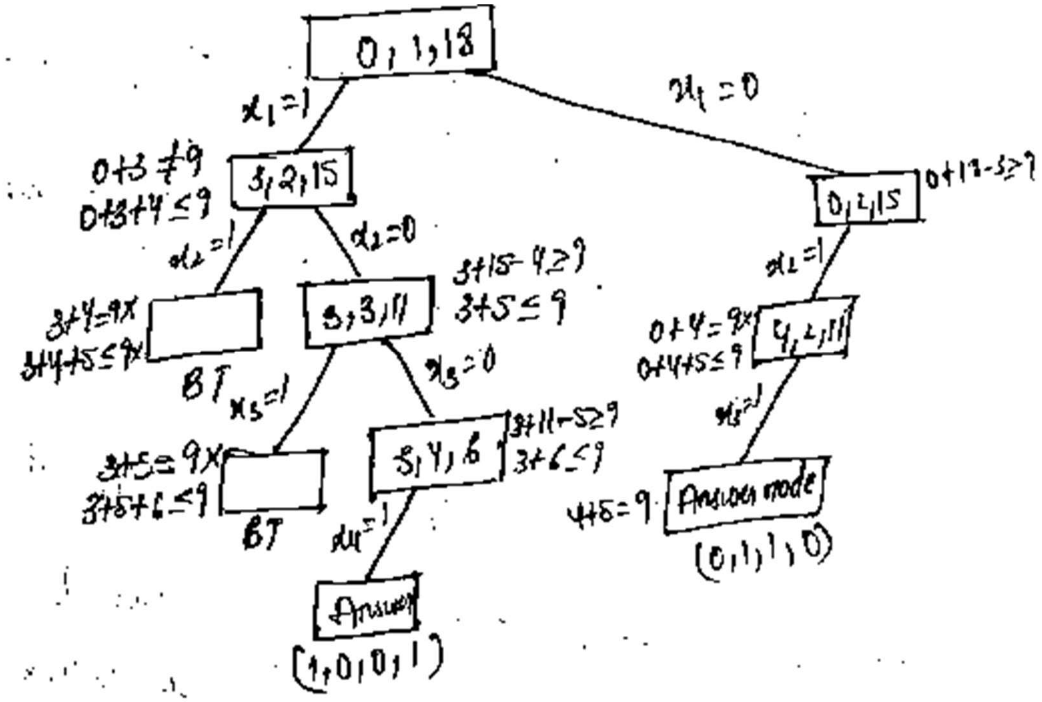
Example:

Let *w= {3,4,5,6}* and *m=9*. Find all possible subsets of *w* that sum to *m*. Draw the portion of the state space tree that is generated.

Solution:

Note: In the state space tree, the rectangular nodes list the values of *s, k,* and *r* on each call to SumOfSub. Initially *s=0, k=1, r=18.*

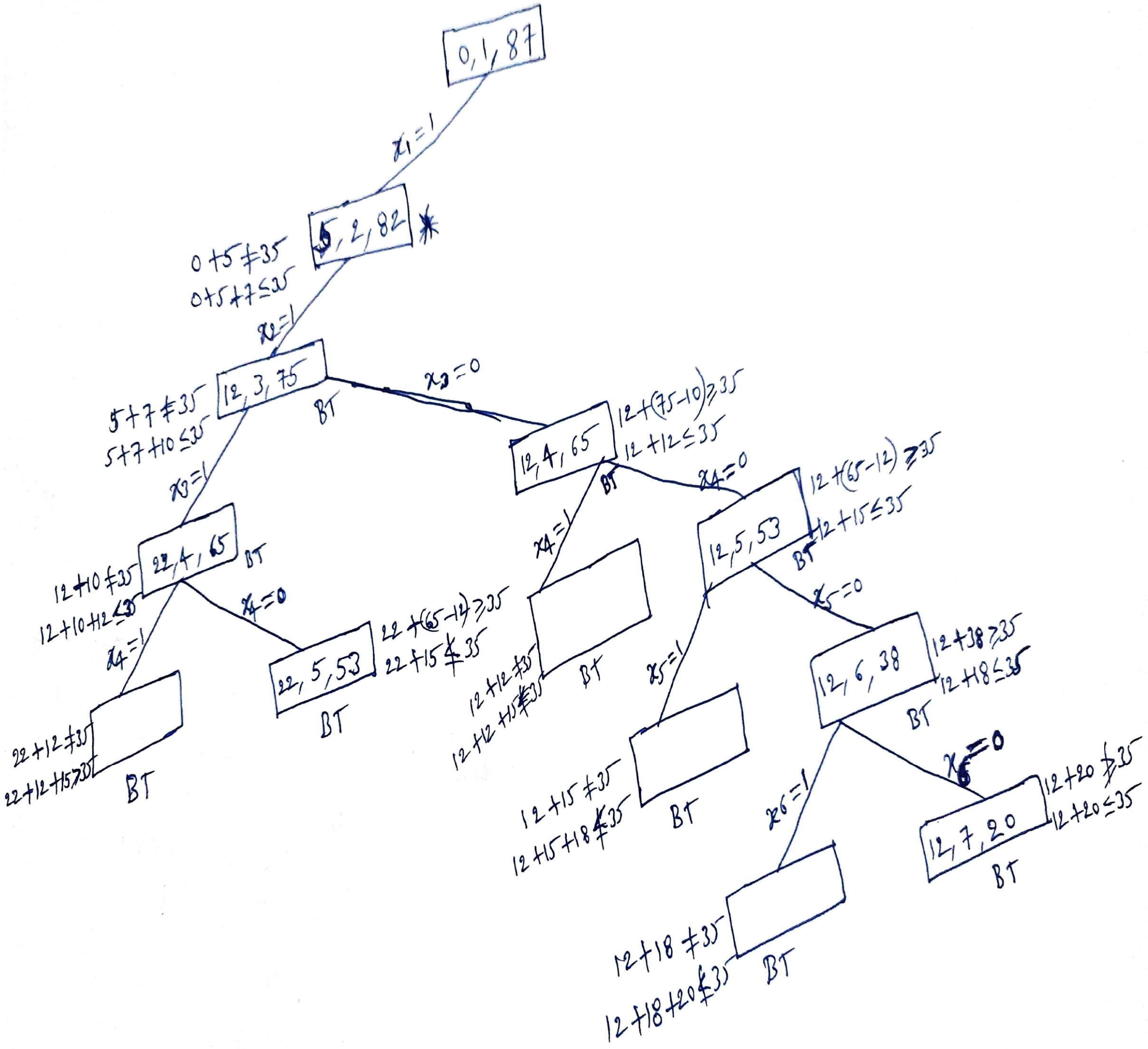
Portion of the state space tree that is generated:



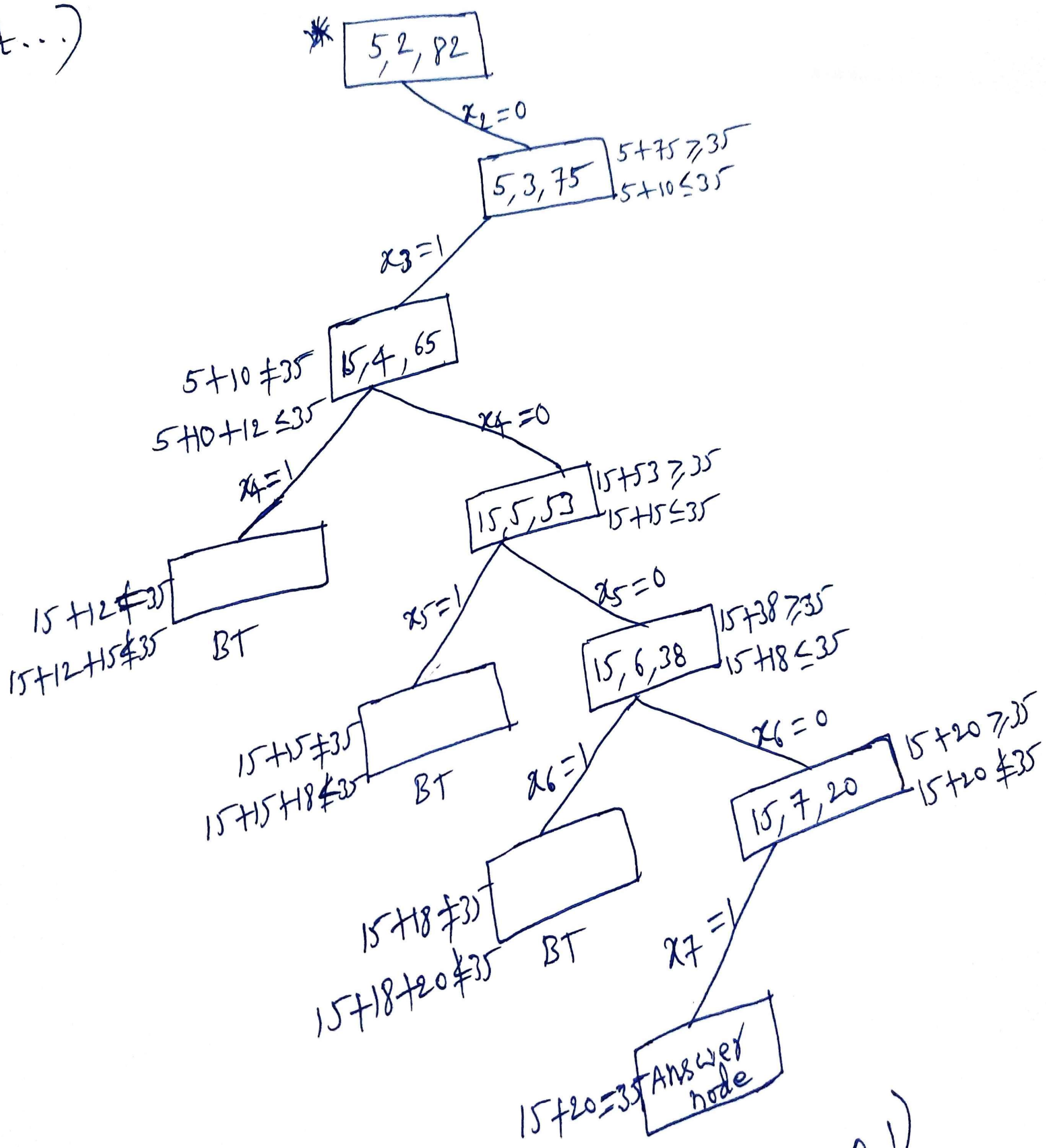
Example: **Let w = {5, 7, 10, 12, 15, 18, 20} and m=35. Find all possible subsets of w that sum to m. Draw the portion of the state space tree that is generated.**

Solution:

Initially, sum of weights, r=5+7+10+12+15+18+20=87.


0,1,87
x1=1
5,2,82
*
0+5≠35
0+5+7≤35
x2=1
12,3,75
BT
x3=0
5+7≠35
5+7+10≤35
12+(75-10)≥35
12+12≤35
12,4,65
BT
x4=0
12+(65-12)≥35
12+15≤35
x3=1
22,4,65
BT
12+10≠35
12+10+12≤35
x4=0
22,5,53
22+(65-12)≥35
22+15≰35
BT
x4=1
22+12≠35
22+12+15≥35
BT
x4=1
12+12≠35
12+12+15≰35
BT
12,5,53
BT
x5=0
x5=1
12,6,38
12+38≥35
12+18≤35
BT
12+15≠35
12+15+18≰35
BT
x6=1
x6=0
12,7,20
12+20≱35
12+20≤35
BT
12+18≠35
12+18+20≰35
BT

(Cont...)



Answer = (1,0,1,0,0,0,1)

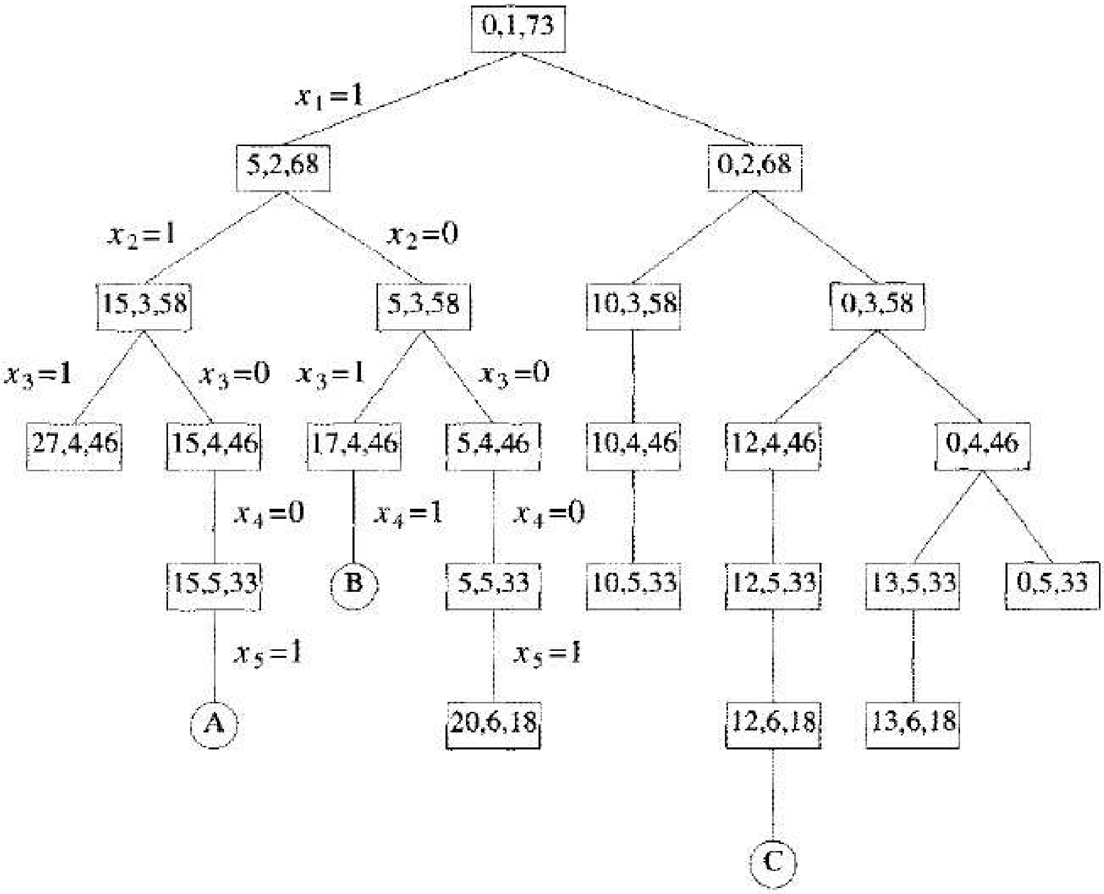
One solution is: (x1, x2, x3, x4, x5, x6, x7) = (1,0,1,0,0,0,1).

<u>Example:</u>

Let *w= {5,10,12,13,15}* and *m=30.* Find all possible subsets of w that sum to m. Draw the portion of the state space tree that is generated.

<u>Solution:</u>

<u>Note:</u> In the state space tree, the rectangular nodes list the values of *s, k,* and *r* on each call to SumOfSub. Answer nodes are represented in circles. Initially *s=0, k=1, r=73.*



0,1,73
$x_1=1$
5,2,68
0,2,68
$x_2=1$
$x_2=0$
15,3,58
5,3,58
10,3,58
0,3,58
$x_3=1$
$x_3=0$
$x_3=1$
$x_3=0$
27,4,46
15,4,46
17,4,46
5,4,46
10,4,46
12,4,46
0,4,46
$x_4=0$
$x_4=1$
$x_4=0$
15,5,33
B
5,5,33
10,5,33
12,5,33
13,5,33
0,5,33
$x_5=1$
$x_5=1$
A
20,6,18
12,6,18
13,6,18
C