UNIT IV Backtracking

Backtracking

Backtracking technique aims to find all solutions to a problem incrementally. Since a problem would have constraints, the solutions that fail to satisfy them will be removed.

It uses recursive calling to find a solution set by building a solution step by step, increasing levels with time. In order to find these solutions, a search tree named **state-space tree is used**. A space state tree is a tree that represents all of the possible states of the problem, from the root as an initial state to the leaf as a terminal state.

A backtracking algorithm follows the depth-first search method. When it starts exploring the solutions, **a bounding function is applied so that the algorithm can check if the so-far built solution satisfies the constraints**. If it does, it continues searching. If it doesn't, the branch would be eliminated, and the algorithm goes back to the level before(backtrack).

General method / backtracking algorithm

- Step 1: Return success if the current position is a feasible solution.
- Step 2: Otherwise, if all paths have been exhausted (i.e., the current position is an endpoint), return failure because there is no feasible solution.
- Step 3: If the current point is not an endpoint, backtrack and explore other points, then repeat the preceding steps.

Example of backtracking approach

N Queen Problem

Sum of subsets

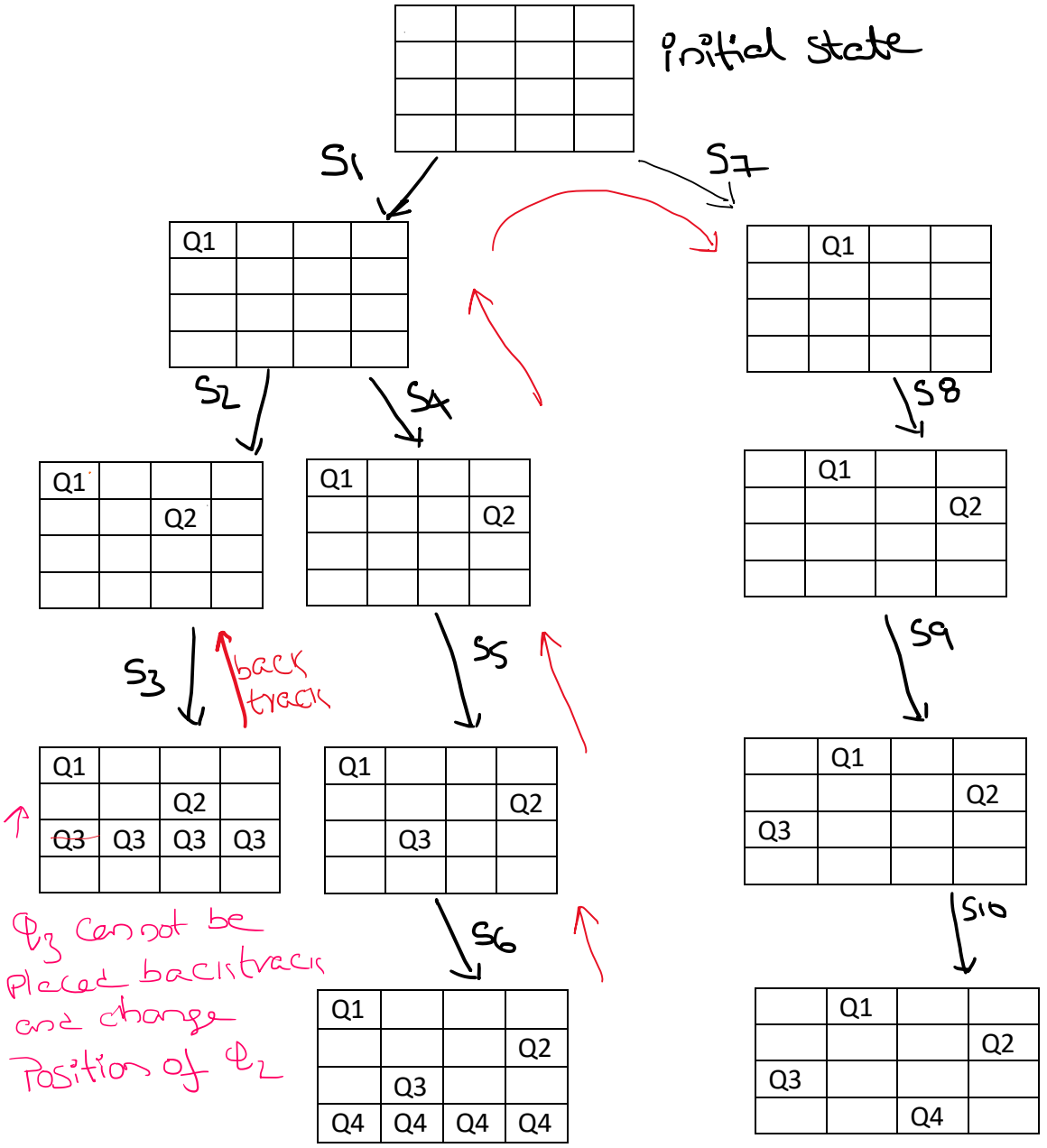
Graph coloring

Hamiltonian Cycle

N Queen Problem : An N Queen problem is example of backtracking, here we need to place n queens on a n*n chess board so that no two queens are in attack position , i.,e **no two queens are on the same row, column or diagonal**.

Lets solve the 4 Queen problem

We need to place 4 queens Q1, Q2, Q3, Q4 in 4*4 box


initial state
S1
S7
Q1
S2
S4
S8
Q2
S3
back track
S5
S9
Q3
S6
S10
Q4
Q3 cannot be placed backtrack and change position of Q2
→ Q4 cannot be placed backtrack and move Q3
→ Q3 cannot be moved further backtrack and move Q2
→ Q2 also cannot be moved further
→ backtrack and move Q1

```
Algorithm NQueen()
    Place (k, i)
    {
        For j ← 1 to k – 1
        {
                do
                        if (x [j] = i)  or (Abs x [j]) - i) = (Abs (j - k))
                                then return false;
        }
        return true;
    } // end of place(k, i)

     N - Queens (k, n)
     {
       For i ← 1 to n
           do if Place (k, i) then
           {
                   x [k] ← i;
                   if (k ==n) then
                           write (x [1....n));
                   else
                           N - Queens (k + 1, n);
           }
     } // end of N – Queens(k,n)
```

8 Queen Problem

8-queen Problem

Number of queens N =8 and Queens: Q1,Q2....Q8

			Q1				
					Q2		
							Q3
	Q4						
						Q5	
Q6							
		Q7					
				Q8			

Fig.: Solution space table for 8-queens

Hence solution vector for 8 queens is (4,6,8,2,7,1,3,5).

Sum Of Sub Sets

The problem is to find a subset from given elements, such that the sum of elements is equal to a given number 'M'. Here backtracking approach is used to find a valid subset. If adding an element is not feasible, we will backtrack to get the previous subset and add other elements to get the solution.

Algorithm Sum of Sub sets

1. Start with an empty set
2. Add the next element from the list to the set
3. If the subset is having sum M, then stop with that subset as solution.
4. If the subset is not feasible(subset sum > M) or if we have reached the end of the set, then backtrack through the subset until we find the most suitable value.
5. If the subset is feasible (sum of subset < M) then go to step 2.
6. If we have visited all the elements without finding a suitable subset and if no backtracking is possible then stop without solution.

Example

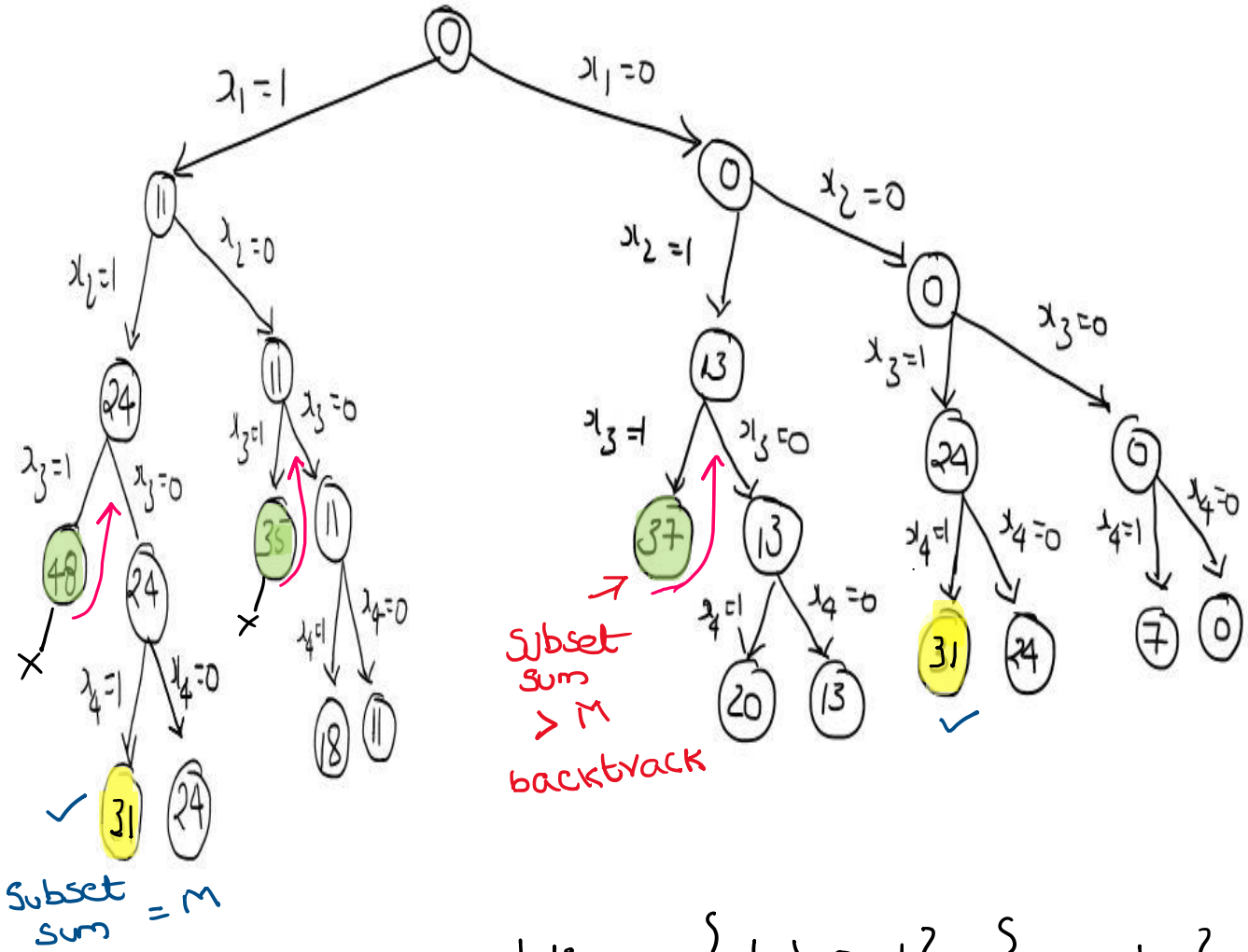
Consider the sum-of-subset problem, n = 4, Sum = 31, and w_1 = 11, w_2 = 13, w_3 = 24 and w_4 = 7. Find a solution to the problem using backtracking. Show the state-space tree leading to the solution.

Sol : Given Elements = { 11,13,24,7}

Required sum = 31

State space tree

The state space tree is drawn such At level i, the left branch corresponds to the inclusion of number w_i and the right branch corresponds to exclusion of number w_i.



Solution = $\{1,1,0,1\}, \{0,0,1,1\}$

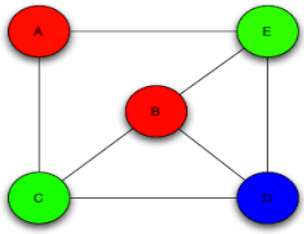
$= \{11,15,7\}, \{24,7\}$

Graph Coloring

The problem is, when given an undirected graph and a number m, determine if the graph can be coloured with at most m colours such that no two adjacent vertices of the graph are colored with the same color.

The least possible value of 'm' required to color the graph successfully is known as the chromatic number of the given graph.

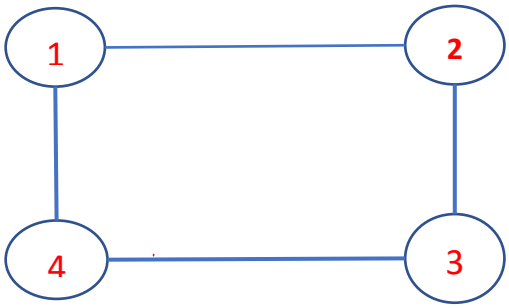
Example :

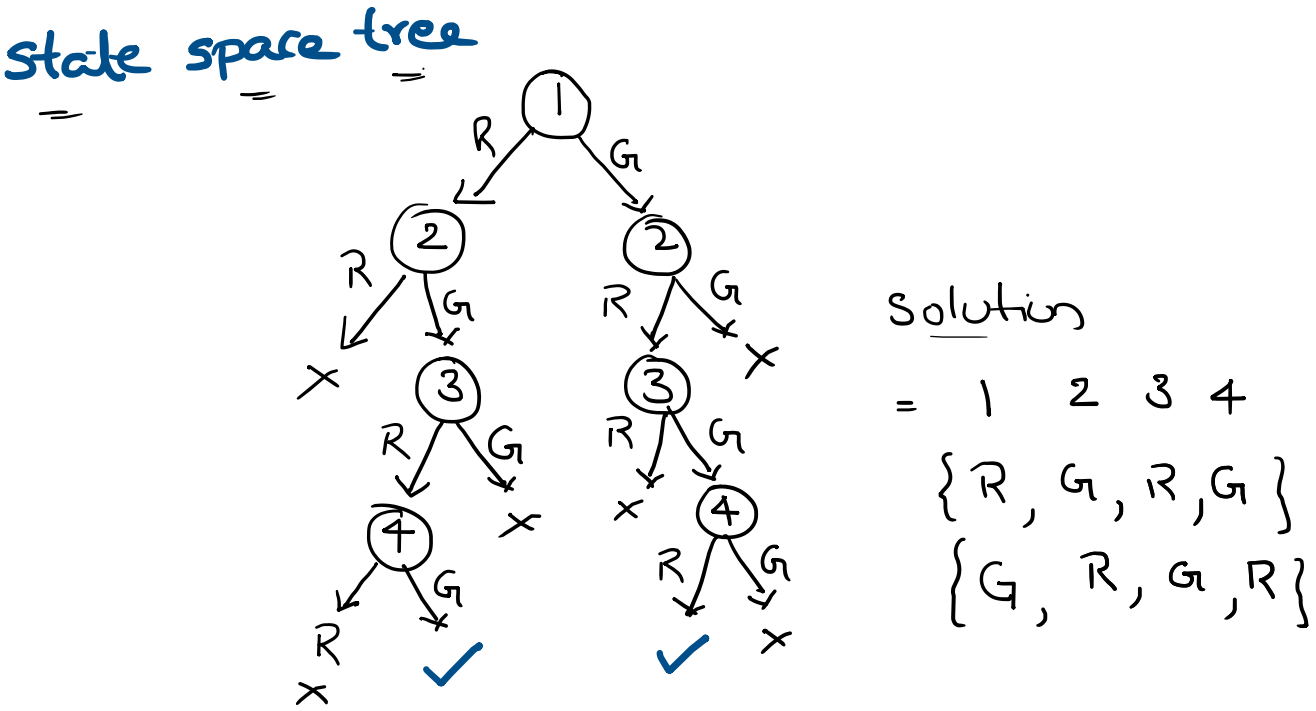



```
Algorithm mcoloring (k)
// This algorithm was formed using the recursive backtracking schema. The graph is
// represented by its Boolean adjacency matrix G [1: n, 1: n]. All assignments of
// 1, 2, . . . . . , m to the vertices of the graph such that adjacent vertices are assigned
// distinct integers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

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

Draw the State space tree for following graph, where colors ={Red, Green}





Hamiltonian Cycle

The **Hamiltonian cycle** is the path in a graph which visits all the vertices in graph exactly once and finally end at the starting node. It may not include all the edges.

The Hamiltonian cycle problem is the problem of finding a Hamiltonian cycle in a graph if there exists any such cycle or not.

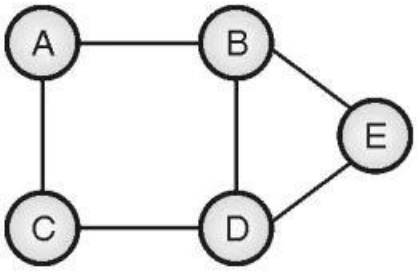
Hamiltonian Cycle Algorithm

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

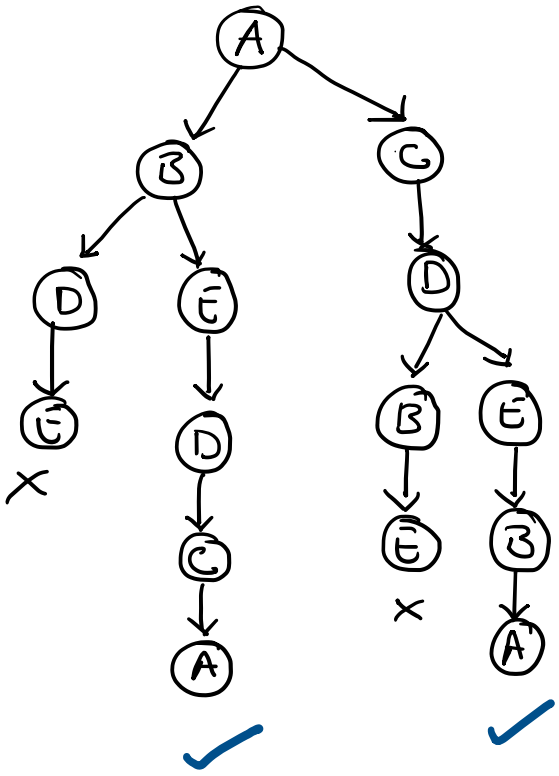
```
Algorithm Hamiltonian (k)
// This algorithm uses the recursive formulation of backtracking to find all the Hamiltonian
// cycles of a graph. The graph is stored as an adjacency matrix G [1: n, 1: n]. All cycles  begin
// at node 1.
{
        repeat
        {                                      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

Example :

Find the Hamiltonian cycle in following graph and draw the space tree



State space tree



Hamiltonian Cycles = $A \to B \to E \to D \to C \to A$

$A \to C \to D \to E \to B \to A$