Backtracking

- Problem which deals with searching for a set of Solutions or which ask for an Optimal Solution satisfying Some Constraints can be solved using Backtracking.
- The desired solution is expressible as an N-Tuple (X_1, X_2, _ _ _ _ _ _, X_n) where X_i are chosen from some Finite Set S_i. The problem to be solved calls for finding one vector that maximizes or minimizes a criterion function P(X_1, X_2, _ _ _ _ _ _, X_n) sometimes it seeks all vectors that satisfies P.
- Let m_i is the size of set S_i then there are $m = m_1 . m_2 . m_3 . \ldots\ldots\ldots m_n$ n – Tuples that are possible candidate sets.
- Brute force approach evaluates all possible m-Tuples
- Backtracking yields same problem with less than m trails
    - The basic idea of Backtracking is to build up the Solution vector one component at a time and
    - to use modified criterion function Pi(X_1, X_2, _ _ _ _ _ _, X_i)(Bounding Function) to test whether the vector being formed has any chance of success.
    - The major advantage of this method is, If it is realized that the partial vector (X_1, X_2, _ _ _ _ _ _, X_i) can in no way lead to an optimal solution, then M_{i+1}, _ _ _ _ M_n possible vectors can be ignored entirely.
- Many Problems solved using Backtracking require that all the solutions satisfy some complex set of Constraints
    - These are classified as
        - Explicit
        - Implicit
    - Explicit constraints are rules that restrict each Xi to take on values only from a given set
    - Implicit Constraints describe the way in which the Xi must relate to each other

    Examples

    In 8-Queens Problem the solution space contains 8^8 tuples 8-Tuples

    Explicit Constraints

    $S_i = \{1,2,3,4,5,6,7,8\}$

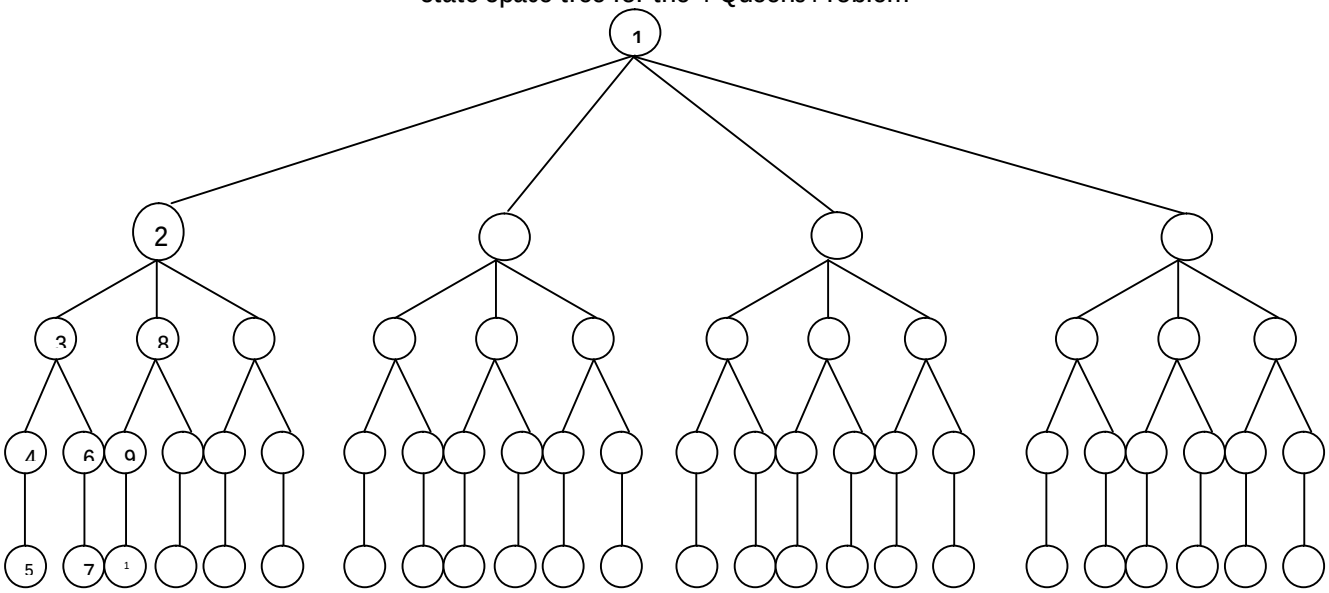
    Implicit Constraints

    No two X_i's can be the same (All Queens are in different Columns)

    No two queens can be on the same Diagonal

- Solution Space is facilitated by Tree Organization called State Space Tree

State Space tree for the 4-Queens Problem



Give Numbers to nodes in the above tree using DFS

State Space tree

Searching of the State Space tree for the solution (vector) can be performed as Begin with the root node and generate other nodes.

Live Node: A node which has been generated and all of whose children have not yet been generated is called Live Node

E-Node: The Live node whose children are currently being generated is called E-node

Dead Node: Dead node is a generated which is not to be expanded further or all of whose children have been generated

Method 1:

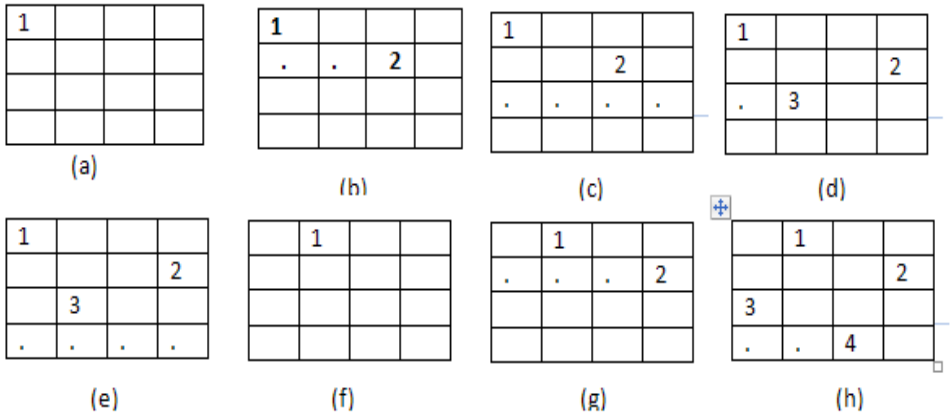
As soon as a new child "C" of the current E Node "R" is generated, this child will become the new E-Node. Then R will become E-node again when the sub-tree C has been Fully Explored. (This is the DFS Strategy)

Method 2:

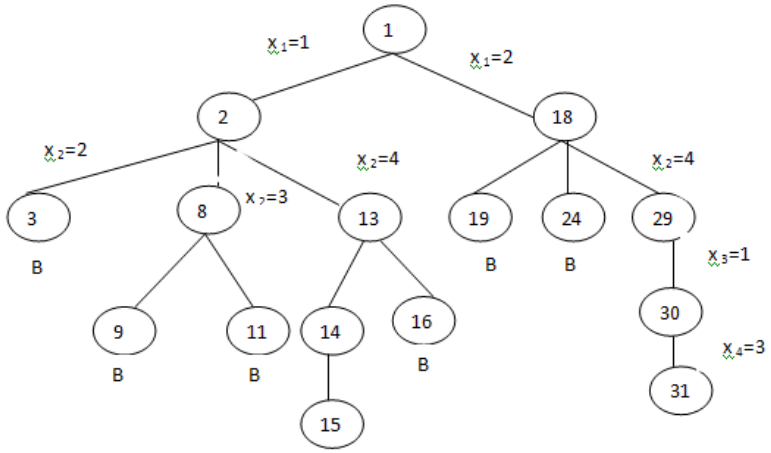
The E-Node will remain E-node until it is Dead.

In the both of the methods, Bounding Functions are used to kill Live Nodes with out generating all of their Children.

Depth First Generation (Method 1) with Bounding Function is called Backtracking.



Example of a backtrack solution to the 4-queens problem



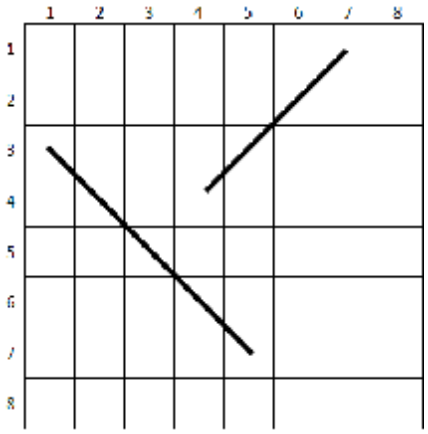
Portion of the tree of State Space Tree for 4-Queens problem that is generated during back-tracking

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

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

8- Queens of N-queens problem:

- We try to find all ways to place "n" not attacking queens in a chess board.
- We represent solution as N-Tuple (X_1, X_2, _ _ _ _ _ _, X_n), where X_i is the column of i^{th} row where the i^{th} queen is placed.
- All Xi's are distinct (not in the Same Column) So our Solution Space is reduced from n^n to n! n-Tuples.
- How to Place not in the Same Diagonal is crucial.
- Dsfsd



In the above board the Diagonal (3,1) and (7,5) is normal Diagonal
Where as (1,7) and (4,4) is opposite Diagonal

In normal Diagonals Difference Between Row and Column is Same
In Opposite Diagonals Sum and Row and Columns is Same
By Keeping these things in mind we formulate a rule that
If (I,j) and (k,l) are two cells then
In Normal Diagonals i-j = k-l

In Opposite Diagonals i+j = k+l

Which implies |j-l| = |i-k|

- With the help of this equation we Frame the Function *Place()*

```
1.  Algorithm place(k,i)
2.  // Returns true if a queen can be placed in kth row and
3.  //ith column. Otherwise it returns false. x[]is a
4.  //global array whose first (k-1) values have been set.
5.  //Abs(r) returns the absolute value of r.
6.  {
7.       for j :=1 to k-1 do
8.        if((x[j] =i) // Two in the same column
9.           or (Abs(x[j]-i) = Abs(j-k)))
10.               // or in the same diagonal
11.          then return false;
12.   return true;
13.  }
```

Algortihm N-Queen is developed using the concept of Backtracking

```
1.  Algorithm N Queen(k,n)
2.  // Using backtracking, this procedure prints all
3.  //possible placement of n queens on an n x n
4.  //chessboard so that they are nonattacking.
5.  {
6.           for i:=1 to n do
7.          {
8.                  if place(k, i) then
9.                  {
10.                         x[k] := i;
11.                         if (k = n] then write (x[1 ; n]);
12.                         else NQueens (k+1,n);
13.                 }
14.         }
15.  }
```

Sum of subsets problem:

- Suppose we are given *n* Distinct Positive numbers (Called Weights) and we desire to find all the combinations of these numbers whose sums are *m*. This is called Sum of Sub Sets Problem.
    - Ex: n=4, (w1, w2, w3, w4) = (11, 13, 24, 7) & m = 31 then the possible subsets are (11, 13, 7) & (24, 7)
- We could formulate this problem using either Fixed or variable sized Tuple
- In Fixed Sized Tuple representation the solution subset is represented by n-Tuple where the elements in this solution vector are either 0 or 1. For Above example in fixed size Tuple is (1,1,0,1) or (0,0,1,1)
- In Variable Size representation we represent our solution set with the index of the element in the actual set
    - For our above example our solution subsets are (1,2,4) or (3,4)
- We Consider the Backtrack solution using fixed Size representation.

A simple choice for the bounding functions is $B_k(x_1,\ldots,x_k)$ = true iff

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

Clearly $x_1,\ldots,x_1$ cannot lead to an answer node if this condition is not satisfied. The bounding functions can be strengthened if we assume the w_i's are initially in non decreasing order .In this case $x_1,\ldots,x_k$ cannot lead to an answer node if

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

The bounding functions we use are therefore

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

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

and

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

Since our algorithm will not make use of B_n, we need not be concerned by the appearance of W_{n+1} in this function. Although we have now specified that is needed to directly use either of the backtracking schemas, a simpler algorithm results if we tailor either of these schemas to the problem at hand This simplification results from the realization that if $x_k = 1$, then

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

- Consider a node at depth i
- weightSoFar (i.e. s) = weight of node, i.e., sum of numbers included in partial solution node represents
- totalPossibleLeft (i.e. r) = weight of the remaining items i+1 to n (for a node at depth i)
- A node at depth i is non-promising
  if (weightSoFar + totalPossibleLeft < S)
  or (weightSoFar + w[i+1] > S)
- To be able to use this "promising function" the wi must be sorted in non-decreasing order

```
1.    Algorithm SumOfSub(s, k ,r)
2.    //Find all subsets of w[1 :n]that sum to m. The values of x[j],
3.    //1≤j<k, have already  been determined. s = ∑ (j=1 to k-1) w[j]* x[j]
4.    //and r=∑ (j=k to n) w[j]'s are in nondecreasing order. n
5.    //It is assumed that w[1]≤ m and=∑ (i=1 to n) w[i] ≥m
6.    {
7.          // Generate left child. Note: s+w[k]≤m since Bk-1 is true.
8.          x[k]:=1
9.          if (s+w[k]=m) then write (1:k); // Subset found
10.         // There is no recursive call here as w[j] > 0,1≤j≤n.
11.         else if (s+w[k]+w[k+1]≤m)
12.               then Sum Of Sub(s+w[k],k+1,r-w[k]);
13.         //Generate right child and evaluate Bk
14.         if((s+r-w[k]≥m) and (s+w[k+1]≤m)) then
15.         {
16.               x[k]:=0;
17.               Sum Of Sub(s, k+1,r-w[k]);
18.         }
19.   }
```

Recursive backtracking algorithm for sum of subset problem

Graph Coloring

- Let G be a Graph and m be a +ve integer
- Nodes of G must be colored with 'm' Colors such that no two adjacent nodes have the same color yet only 'm' colors are used.
- This is termed "m-color ability Decision" Problem
- 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
- Ex:
- For a Planar Graph the maximum number for the chromatic number is 4.

```
1.    Algorithm  mColoring(k)
2.     // This algorithm was formed using the recursive backtracking
3.     // schema. The graph is represented by its Boolean adjacency
4.     // matrix G[1 :n, 1:n]. All assignments of 1,2,…, m to the
5.     // vertices of the graph such that adjacent vertices are
6.     // assigned distinct integers are printed. k is the index
7.     // of the next vertex to color.
8.     {
9.          repeat
10.         {// generate all legal assignments for x[k].
11.               Next value (k);// Assign to x(k) a legal color.
12.               if (x[k]=0) then return; // No new color possible
13.               if(k=n) then          // At most m colors have been
14.                     // used to color the n vertices.
15.                         write (x[k+1];
16.               else mColoring(k+1)
17.         } until ( false);
18.   }
```

Finding all – m coloring of a graph

- The NextValue(k) will assign the next possible color to the vertex k. This function will checks the newly assigned color is not as the color of the adjacent vertex(which were already assigned)
- mColoring(k) calls NextValue(k) and checks whether all vertices were colored(k=n or not), if not it will calls mColoring(k+1)

```
1.  Algorithm Next Value(k)
2.    // x[1],.....,x[k-1] have been assigned integer values in
3.    // the range [1,m] such that adjacent vertices have distinct
4.    //integers  A value for x[k] is determined in the range
5.    //[o,m].x[k] is assigned the next highest. numbered color
6.    //while maintaining distinctness from the adjacent vertices
7.    //of vertex k. If no such color exists, then x[k] is 0
8.  {
9.        repeat
10.       {
11.           x[k]:=(x[k]+1) mod (m+1); // Next highest color.
12.           if ((x[k]=0) then return; // All colors have been used.
13.           for j:= 1 to n do
14.           {       // Check if this color is
15.                   // distinct from adjacent colors.
16.                   if((G[k,j]≠0) and (x[k]=x[j]))
17.                   //if (k,j) is and edge and if adj.
18.                   //vertices have the same color.
19.                   then break
20.           }
21.           if (j=n+1) then return; // New color found
22.       } until (false); //Otherwise try to find another color.
23. }
```

Generating a next color

Hamiltonian Cycle

- Let graph G=(V,E) be a connected graph with n vertices. Hamiltonian Cycle is a round trip path along n edges of G that visits every vertex once and returns to its starting position.
- 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. Now all we need is determine how to compute the set of possible vertices for xk if x_1,x_2 ..... x_{k-1} have already been chosen. If k =1 then x1 can be any of the n vertices.
- To avoid printing the same cycle n times, we require that x_1 =1.
- If 1<k<n, then x_k can be any vertex v that is distinct from x_1, x_2 ..... x_{k-1} and *v* is connected by an edge to x_{k-1}. The vertex X_n can only be the one remaining vertex and it must be connected to both x_{n-1} and x_1.

```
1.  Algorithm  HamiItonian(k)
2.  // This algorithm uses the recursive formulation of
3.  // backtracking to find all the HamiItonian cycles
4.  //of a graph. The graph is stored as an adjacency
5.  //matrix G[1: n,1:n]. All cycles begin at node 1.
6.  {
7.          repeat
8.          { // Generate values for x[k]
9.                  Next value(k);// Assign a legal next value to x[k]
10.                 if(x[k]=0) then return;
11.                 if (k=n) then write (x[1:n]);
12.                 else HamiItonian(k+1);
13.         } until (false);
14. }
```

```
1.  Algorithm Next values(k)
2.   // x[1:k-1]is a path of k-1 distinct vertices. If[k]=0,then
3.  // no vertex has as yet been assigned to x[k]. After execution
4.  //x[k]is assigned to the next highest numbered vertex which
5.  //does not already appear in x[1:k-1]and is connected by
6.  // an edge to x[k-1].Otherwise x[k]=0.If k=n, then
7.  // in addition x[k] is connected to x[1]
8.  {
9.          repeat
10.         {
11.                 x[k]:=(x[k]+1)mod (n+1); //Next vertex
12.                 if(x[k]=0) then return;
13.                 if(G[x[k-1],x[k]≠0) Then
14.                 {// Is there an edge?
15.                         for j:=1 to k-1 do if (x[j]-x[k] then break;
16.                         //check for distinctness.
17.                         if (j=k) then // if true, then the vertex is distinct.
18.                         if((k<n) or((k=n) and G[x[n],x[1]]≠0))
19.                         then return;
20.                 }
21.          } until (false)
22.  }
```

Generating a next vertex