Tutorial - 5

Q.1) Explain general backtracking method.
Write note on "Backtracking".

Backtracking represents one of the most general techniques.

- Problems which deal with searching for a set of solution or which ask for an optimal soln satisfying some constraints can be solved using backtraing formulation.
- In many applications of backtrack method, desired soln is expressible as an n-tuple $(x_1, x_2, x_3, \cdots x_n)$ where x_i are shown chosen from some finite set S_i.
- Often the problem to be solved calls for finding one vector that, maximizes (or minimizes or satisfies) a criterion function $P(x_1, x_2, \cdots x_n)$
- Sometimes it seeks all vectors that satisfy P
- Eg: Sorting - Criteria function P is inequality
- Suppose m_i is the size of set S_i
- Then there are $m = m_1, m_2, m_3, \cdots m_n$ n-tuples tat are possible candidates for satisfying the fun 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 as its virtue 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 $P_i(x_1, x_2 \cdots x_i)$ (sometimes called bounding functions)

- To test whether the vector being formed has a chance of success.
- 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.
- Many of the problems we solve using backtracking require that all the solutions satisfy a complex set of constraints.
- For any problem these constraints can be divided into two categories: explicit and implicit.

Q.2) Define

a) Implicit constraints:

The implicit constraints are rules that determine which of the tuples in the solution space of I satisfy the criterion function. Thus implicit constraints describe the way in which the x_i must relate to each other.

b) Explicit constraints:

Explicit constraints are rules that restrict each x_i to take on values only from a given set.

Common examples of explicit constraints are:

$x_i \geq 0$ or $S_i = \{$all non-negative real numbers$\}$

$x_i = 0$ or 1 or $S_i = \{0, 1\}$

$l_i \leq x_i \leq u_i$ or $S_i = \{a : l_i \leq a \leq u_i\}$

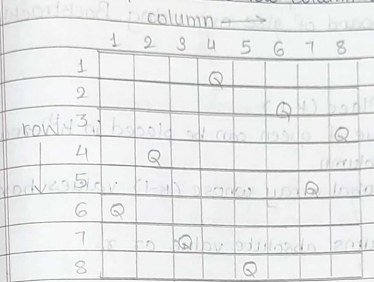
The explicit constraints depend on the particular instance I of the problem being solved. All tuples that satisfy the explicit constraints define a possible solution space for I.

Q.3) Explain solution to n-queens problem using Backtracking method.

Tackle the 8-queens problem via a backtracking solution. Generalize the problem and consides an nxn chessboard and try to find all ways to place n non-attacking queens.

- Let $(x_1, x_2, \ldots x_n)$ represents a solution in which x_i is the column of the i^{th} row where the i^{th} queen is placed.
- The 8-queens problem can be represented as 8-tuples $(x_1, x_2, \ldots, x_8)$
- $S_i = \{1, 2, 3, 4, 5, 6, 7, 8\}$; $1 \le i \le 8$
- The solution space consists of 8^8 8 tuples
- The implicit constraints for this problem are that no two x_i's can be the same (i.e. all queens must be on different columns) and no two queens can be on the same diagonal.
- The first of these two constraints implies that all solutions are permutations of the 8-tuple (1, 2, 3, 4, 5, 6, 7, 8)
- This realization reduces the size of the solution space from 8^8 tuples to 8! tuples.
- The x_i's will be all distinct since no two queens can be placed in the same column.
- Now how do we test whether two queens are on the same diagonal?
- Imagine the chessboard squares being numbered as the indices of the 2D Array a[1:n, 1:n].
- Then observe that every element on the same diagonal that runs from the upper left to the lower right has the same row-column value.

- For example, in figure consider the queen at a[4,2]
- The squares that are diagonal to this queen a[3,1], a[5,3], a[6,4], a[7,5] and a[8,6]
- All these squares have row-column value of 2.



- Also, every element on the same diagonal that goes from the upper right to the lower left has the same row + column value.
- Suppose two queens are placed at positions (i,j) and (k,l).
- Then by above they are on the same diagonal only if: $i-j = k-l$
  or $i+j = k+l$
  The first equation implies $j-l = i-k$
  The second equation implies $j-l = k-i$
- Therefore two queens lie on the same diagonal if and only if $|j-l| = |i-k|$

$= +(i-k)$
$= +(i-k)$ $\quad i-k$
$= -(i-k)$ $\quad k-i$

Q.4) State n-queens problem and write an algorithm to test that no two queens are placed in same diagonal.
Write algorithm to place n non-attacking queens on a chessboard of size nxn using Backtracking Approach.

Algorithm Place(k,l)
// Return True if queen can be placed in k^{th} row and l^{th} column.
// x[] is global array whose (k-1) values have been set.
// Abs(r) returns absolute value of r.
{
For i=1 to k-1 do
if(x[i]=l) // Two queens in same column.
OR
Abs(x[i]-l)= Abs(i-k) // Two queens on same diagonal.
Then return false;
Return True;
}

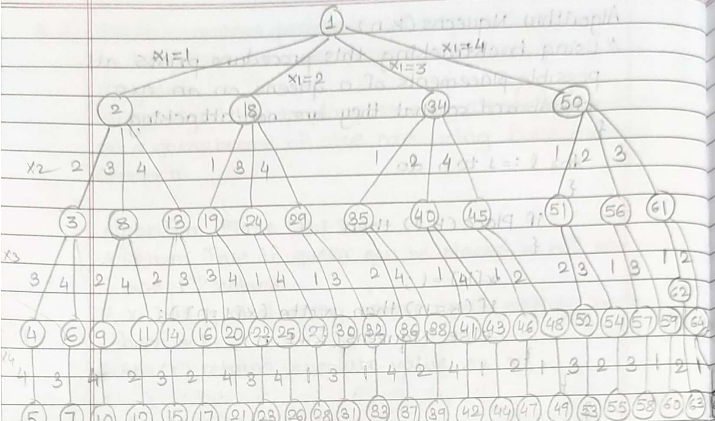
- Place(k,l) returns a boolean value that is true if the k^{th} queen can be placed in column l.
- It tests both whether l is distinct from all previous values x[1], ..., x[k-1] and whether there is no other queen on the same diagonal.
- It's computing time is O(k-1)

k = queen
i = row of previous column
l = column

Algorithm Nqueens(k,n)
// Using backtracking, this procedure prints all possible placements of n queens on an nxn chessboard so that they are non-attacking.
{
for l := 1 to n do
{
if Place(k,l) then
{
x[k] = l;
if(k=n) then write(x[1:n]);
else Nqueens(k+1, n);
}
}
}

Q.5) Draw and explain state space tree / Permutation tree / Tree organization for 4 queen problem.

Permutation Tree is used to show the solution space consisting of n! permutations of n-tuples (1,2,...,n)
- The edges are labeled by possible values of x_i.
- Edges from level 1 to level 2 nodes specify the values for x_1.
- Leftmost subtree contains all solutions with $x_1 = 1$, $x_2 = 2$ and so on.
- Edges from level i to i+1 are labeled with values of x_i.
- The solution space is defined by all paths from the root node to a leaf node.
- There are 4! = 24 leaf nodes.



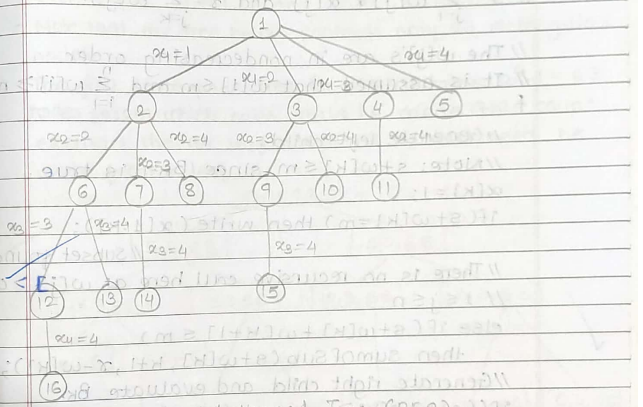
Q.6) With suitable example explain sum of subsets problem using Backtracking method.
Explain sum of subset problem and bounding functions used in solution to sum of subsets using backtracking.
Write algorithm to find subset of elements giving sum=m using Backtracking method.

Suppose we are given n distinct positive numbers (usually called weights), we desire to find all combinations of these numbers whose sums are m. This is called the sum of subsets problem.
Consides a backtracking solution using the fixed tuple size strategy.

- 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 $n=4$, $(W_1, W_2, W_3, W_4) = (7, 11, 13, 24)$ 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. Now the two solutions are described by the vectors (1, 2, 3) and (1, 4)



Unbalanced Tree (BFS)

- In this case the element x_i of the solution vector is either one or zero depending on whether the weight w_i is included or not.
  S = sum upto k-1 element
  K = current element
  R = sum from k to n.

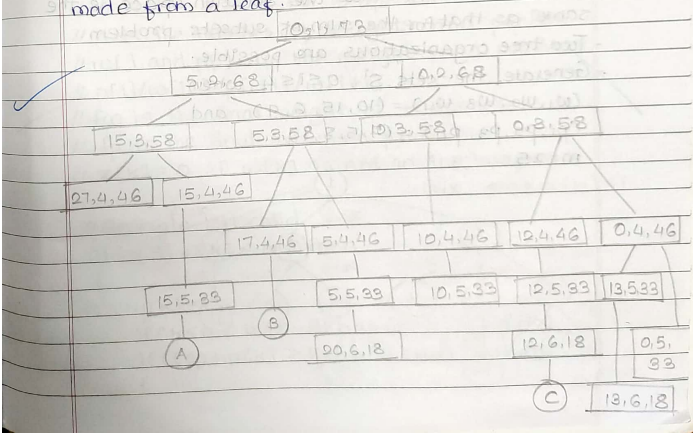
- Algorithm SumOfSub (s, k, r)
// Find all subsets of w[1:n] that sum to m.
// The values of x[j], $1 \le j < k$, have already been determined.
// $s = \sum_{j=1}^{k-1} w[j] * x[j]$ and $r = \sum_{j=k}^{n} w[j]$.

// The w[j]'s are in nondecreasing order.
// It is assumed that $w[1] \le m$ and $\sum_{i=1}^{n} w[i] \ge m$
{
  // Generate left child.
  // Note: $s + w[k] \le m$ 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,
  // $1 \le j \le n$
  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]);
  }
}

8.7) Draw possible solution space organization diagram for sum of subset problem.

Diagram below shows the portion of the state space tree generated by function SumofSub while working on the instance n=6, m=30 and w[1:6] = {5, 10, 12, 13, 15, 18}. The rectangular node list the values of s, k and r on each of the calls to SumOfSub. Circular nodes represent points at which subsets with sums m are printed out.

- At nodes A, B and C the output is respectively (1,1,0,0,1), (1,0,1,1) and (0,0,1,0,0,1).
- Note that the tree below contains only 23 rectangular nodes.
- The full state space tree for n=6 contains $2^6 - 1 = 63$ nodes from which calls could be made (this count excludes the 64 leaf nodes as no call need be made from a leaf.

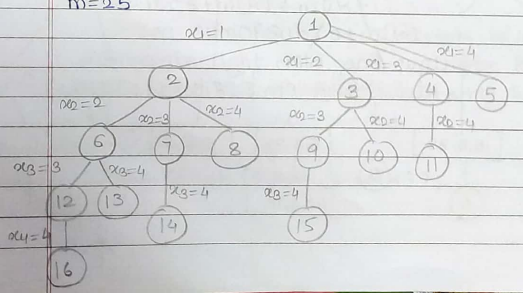


Q.8) Explain solution to 0/1 Knapsack problem using Backtracking method.
Write algorithm to solve 0/1 knapsack problem using Backtracking Approach.

Reconsider 0/1 knapsack problem solved using dynamic programming approach.
- Given n positive weights wi, n positive profits pi, and positive number m that is a knapsack capacity.
- This problem calls for choosing a subset of weights such that:

$$\sum_{1 \le i \le n} w_i x_i \le m \text{ and } \sum_{1 \le i \le n} p_i x_i \text{ is maximized}$$

- The x_i's constitute a zero-one-valued vector.
- The solution space for this problem consists of the 2^n distinct ways to assign zero or one values to the x_i's. Thus the solution space is the same as that for the sum of subsets problem.
- Two tree organizations are possible.
- Generate the sets S^i, $0 \le i \le 4$, when $(w_1, w_2, w_3, w_4) = (10, 15, 6, 9)$ and $(p_1, p_2, p_3, p_4) = (2, 5, 8, 1)$
$m = 25$



- Algorithm Bound (cp, cw, k)

```
// cp is the current profit total;
// cw is the current weight total;
// k is the index of the last removed item
// m is the knapsack size.
{
    b = cp; c = cw;
    for i = k+1 to n do
    {
        c = c + w[i];
        if (c < m) then b = b + p[i];
        else return b + (1 - (c-m)/w[i]) * p[i];
    }
    return b;
}
```

- Algorithm BKnap(k, cp, cw)

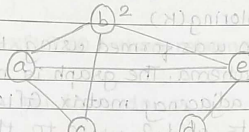
```
// m is the size of the knapsack;
// n is the number of weights and profits.
// w[] and p[] are the weights and profits.
// p[i]/w[i] ≥ p[i+1]/w[i+1].
// fw is the final weight of knapsack.
// fp is the final maximum profit.
// x[k] = 0 if w[k] is not in the knapsack;
                                    else x[k] = 1

// Generate left child
if (cw + w[k] ≤ m) then
{
    y[k] = 1;
    if (k < n) then BKnap(k+1, cp + p[k], cw + w[k]);
    if ((cp + p[k] > fp) and (k = n)) then
    {
```

```
            fp = cp + p[k];
            fw = cw + w[k];
            for j = 1 to k do x[j] = y[j];
        }
    }
    // Generate right child
    if(Bound(cp, cw, k) >= fp) then
    {
        y[k] = 0;
        if(k<n) then BKnap(k+1, cp, cw);
        if((cp>=fp) and (k=n)) then
        {
            fp = cp;
            fw = cw;
            for j=1 to k do x[j] = y[j];
        }
    }
}
```

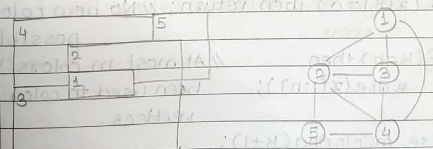
Q.9) Write note on Graph coloring problem.
Give backtracking solution to Graph coloring problem.
Write algorithm to color vertices of a graph with unique colors from adjacent vertices using Backtracking approach.

Let G be a graph and m be a given positive integes.
We want to discover whether the nodes of G can be coloured in such a way that no 2 adjacent nodes have same color. Yet only m colors are used. This is termed the m-colorability decision problem.

- If d is degree of given graph, then it can be colored with d+1 colors.
- The m-colorability optimization 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.
- For eg: the graph of figure below can be colored with 3 colors 1, 2 & 3.
- The color of each node is indicated next to it.
- It can also be seen that three colors are needed to color this graph and hence this graph's chromatic number is 3.



- A graph is said to be planar iff it can be drawn in a plane in such a way that no two edges cross each other.
- A famous special case of m-colorability decision problem is 4-color problem for planar graphs.
- Figure shows a map with 5 regions and its corresponding graph. This map requires 4 colors.



- We are interested in determining all the different ways in which a given graph can be colored using at most m colors.
- 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 of G,
and G[i,j] = 0 otherwise.
- The colors are represented by the integers 1,2,...,m & the soln are given by n-tuple $(x_1, x_2 \cdots x_n)$ where x_i is the color of node i. Using the recursive backtracking formulation the resulting algorithm is mColoring:

```
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
         write(x[1:n]);         been used to color n
                                vertices.
      else mColoring(k+1);
   } until(false)
}
```

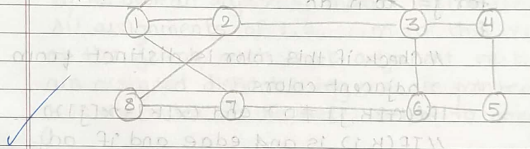
- Algorithm NextValue(k)
// x[1], ... , x[k-1] have been assigned integer values in the range [1,m] such that adjacent vertices have distinct integers. A value for x[k] is determined in the range [0,m]. x[k] is assigned the next highest numbered color while maintaining distinctness from the adjacent 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 an edge and if adj. vertices have same color.
        then break;
    }
    if(j=n+1) then return; // New color found.
  }until(false); // Otherwise try to find another color;
}

Q.10) Write note on Hamiltonian cycle.
Give backtracking solution to Hamiltonian cycle.
Write algorithm to find Hamiltonian cycle using Backtracking approach.

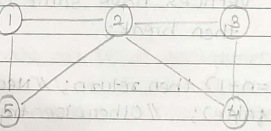
Let G=(V,E) be a connected graph with n vertices.
- A Hamiltonian cycle is a round trip path along n edges of G that visits every vertex once and returns to its starting position.
- In other words,

If a hamiltonian cycle begins at some vertex $v_1 \in G$ and the vertices of G are visited in the ordes $v_1, v_2, \ldots, v_{n+1}$ then the edges (v_i, v_{i+1}) are in E, $1 \le i \le n$ and the v_i are distinct except for v_1 & v_{n+1} which are equal.

G1:



G2:



- The graph G1 contains the Hamiltonian cycle 1,2,8,7,6,5,4,3,1
- The graph G2 contains no Hamiltonian cycle.

- There is no known easy way to determine whether a given graph contains a Hamiltonian cycle.
- Backtracking algorithm can find all the Hamiltonian cycles in a graph. The graph may be directed or undirected
- Only distinct cycles are output.
- The backtracking soln vector $(x_1, \dots x_n)$ is defined as so that x_i represent the i^{th} visited vertex of the proposed cycle.
- Now all we need to do is determine how to compute the set of possible vertices.
- For x_k if $x_1, x_2, \dots x_{k-1}$ have already been choosen. 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 $x_1, x_2, \dots x_{k-1}$ and v is connected by edges 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
- Function NextValue(k) determines a possible next vertex for the proposed cycle.
- Using NextValue particularize the recursive backtracking schema to find all Hamiltonian cycles.
- This algorithm is started by first initializing the adjacency matrix G[1:n, 1:n], then setting x[2:n] to zero and x[1] to 1, and then executing Hamiltonian(2)

```
Algorithm NextValue(k)
{
    repeat
    {
        x[k] = (x[k]+1) mod (n+1);
        if(x[k] = 0)
            then return;
```

```
if (G[x[k-1], x[k]] ≠ 0) then
{
    for j=1 to k-1 do
        if (x[j] = x[k])
            then break;
    if(j=k) then
        if((k≤n) or (k=n) and G[x[n], x[1]] ≠0))
            then return;
}
}until (false)
}

Algorithm Hamiltonian(k)
{
    repeat
    {
        NextValue(k);
        if(x[k] =0) then return;
        if(k=n) then write(x[1:n]);
        else Hamiltonian (k+1);
    }until (false);
}
```

Q.11) Explain the following terms with suitable example.

a) Live node:

Live node is a generated node for which all of the children have not been generated yet.

b) E-node:

E-node is live node whose children are currently being generated or explored.

c) Dead node:

Dead node is a generated node that is not to be expanded any further.

- All the children of dead node are aldready generated.
- Live nodes are killed using a bounding function to make them dead nodes.

d) Bounding function:

Modified criteria function.

- Bounding fun is needed to help kill some live nodes without expanding them.

Eg: Sorting - Comparison.