Chapter 7

BACKTRACKING

General Method:

Backtracking is used to solve problem in which a sequence of objects is chosen from a specified set so that the sequence satisfies some criterion. The desired solution is expressed as an n-tuple (x1, ... , x_n) where each x_i Є S, S being a finite set.

The solution is based on finding one or more vectors that maximize, minimize, or satisfy a criterion function P (x_1, ... , x_n). Form a solution and check at every step if this has any chance of success. If the solution at any point seems not promising, ignore it. All solutions requires a set of constraints divided into two categories: explicit and implicit constraints.

Definition 1: Explicit constraints are rules that restrict each x_i to take on values only from a given set. Explicit constraints depend on the particular instance I of problem being solved. All tuples that satisfy the explicit constraints define a possible solution space for I.

Definition 2: 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's must relate to each other.

- For 8-queens problem:

  Explicit constraints using 8-tuple formation, for this problem are S= {1, 2, 3, 4, 5, 6, 7, 8}.

  The implicit constraints for this problem are that no two queens can be the same (i.e., all queens must be on different columns) and no two queens can be on the same diagonal.

Backtracking is a modified depth first search of a tree. Backtracking algorithms determine problem solutions by systematically searching the solution space for the given problem instance. This search is facilitated by using a tree organization for the solution space.

Backtracking is the procedure where by, after determining that a node can lead to nothing but dead end, we go back (backtrack) to the nodes parent and proceed with the search on the next child.

A backtracking algorithm need not actually create a tree. Rather, it only needs to keep track of the values in the current branch being investigated. This is the way we implement backtracking algorithm. We say that the state space tree exists implicitly in the algorithm because it is not actually constructed.

Terminology:

Problem state is each node in the depth first search tree.

Solution states are the problem states 'S' for which the path from the root node to 'S' defines a tuple in the solution space.

Answer states are those solution states for which the path from root node to s defines a tuple that is a member of the set of solutions.

State space is the set of paths from root node to other nodes. *State space* tree is the tree organization of the solution space. The state space trees are called static trees. This terminology follows from the observation that the tree organizations are independent of the problem instance being solved. For some problems it is advantageous to use different tree organizations for different problem instance. In this case the tree organization is determined dynamically as the solution space is being searched. Tree organizations that are problem instance dependent are called dynamic trees.

Live node is a node that has been generated but whose children have not yet been generated.

E-node is a live node whose children are currently being explored. In other words, an E-node is a node currently being expanded.

Dead node is a generated node that is not to be expanded or explored any further. All children of a dead node have already been expanded.

Branch and Bound refers to all state space search methods in which all children of an E-node are generated before any other live node can become the E-node.

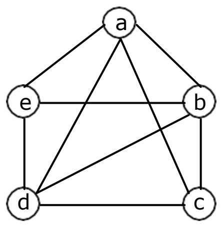
Depth first node generation with bounding functions is called ***backtracking***. State generation methods in which the E-node remains the E-node until it is dead, lead to branch and bound methods.

Planar Graphs:

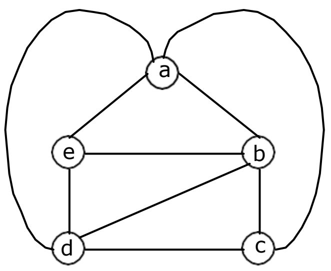
When drawing a graph on a piece of a paper, we often find it convenient to permit edges to intersect at points other than at vertices of the graph. These points of interactions are called crossovers.

A graph G is said to be planar if it can be drawn on a plane without any crossovers; otherwise G is said to be non-planar i.e., 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.

Example:



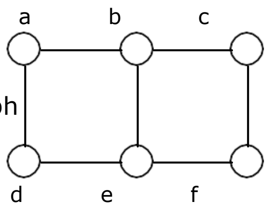
the following graph can be redrawn without crossovers as follows:

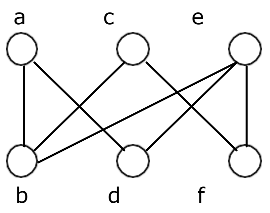


Bipartite Graph:

A bipartite graph is a non-directed graph whose set of vertices can be portioned into two sets V_1 and V_2 (i.e. $V_1 \cup V_2 = V$ and $V_1 \cap V_2 = ø$) so that every edge has one end in V_1 and the other in V_2. That is, vertices in V_1 are only adjacent to those in V_2 and vice- versa.

Example:

The graph 

is bipartite. We can redraw it as 

The vertex set V = {a, b, c, d, e, f} has been partitioned into V_1 = {a, c, e} and V_2 = {b, d, f}. The complete bipartite graph for which $V_1 = n$ and $V_2 = m$ is denoted $K_{n,m}$.

N-Queens Problem:

Let us consider, N = 8. Then 8-Queens Problem is to place eight queens on an 8 x 8 chessboard so that no two "attack", that is, no two of them are on the same row, column, or diagonal.

All solutions to the 8-queens problem can be represented as 8-tuples $(x_1, ... , x_8)$, where x_i is the column of the ith row where the ith queen is placed.

The explicit constraints using this formulation are $S_i = \{1, 2, 3, 4, 5, 6, 7, 8\}$, $1 \leq i \leq 8$. Therefore 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.

This realization reduces the size of the solution space from 8^8 tuples to 8! Tuples.

The promising function must check whether two queens are in the same column or diagonal:

Suppose two queens are placed at positions (i, j) and (k, l) Then:

- Column Conflicts: Two queens conflict if their x_i values are identical.
- Diag 45 conflict: Two queens i and j are on the same 45^0 diagonal if:

  $$i – j = k – l.$$

  This implies, $j – l = i – k$
- Diag 135 conflict:

  $$i + j = k + l.$$

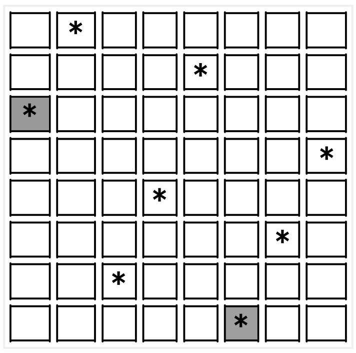
  This implies, $j – l = k – i$

Therefore, two queens lie on the same diagonal if and only if:

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

Where, j be the column of object in row i for the ith queen and l be the column of object in row 'k' for the kth queen.

To check the diagonal clashes, let us take the following tile configuration:



In this example, we have:

i	1	2	3	4	5	6	7	8
x_i	2	5	1	8	4	7	3	6

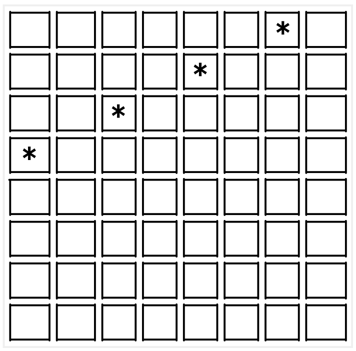
Let us consider for the case whether the queens on 3rd row and 8th row are conflicting or not. In this case (i, j) = (3, 1) and (k, l) = (8, 6). Therefore:

$$|j - l| = |i - k| \Rightarrow |1 - 6| = |3 - 8|$$
$$\Rightarrow 5 = 5$$

In the above example we have, $|j - l| = |i - k|$, so the two queens are attacking. This is not a solution.

Example:

Suppose we start with the feasible sequence 7, 5, 3, 1.



Step 1:
Add to the sequence the next number in the sequence 1, 2, . . . , 8 not yet used.

Step 2:
If this new sequence is feasible and has length 8 then STOP with a solution. If the new sequence is feasible and has length less then 8, repeat Step 1.

Step 3:
If the sequence is not feasible, then *backtrack* through the sequence until we find the *most recent* place at which we can exchange a value. Go back to Step 1.

1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**Remarks**
7	5	3	1					
7	5	3	1*	2*				$\lvert j - l\rvert = \lvert 1 - 2\rvert = 1$ $\lvert i - k\rvert = \lvert 4 - 5\rvert = 1$
7	5	3	1	4				
7*	5	3	1	4	2*			$\lvert j - l\rvert = \lvert 7 - 2\rvert = 5$ $\lvert i - k\rvert = \lvert 1 - 6\rvert = 5$
7	5	3*	1	4	6*			$\lvert j - l\rvert = \lvert 3 - 6\rvert = 3$ $\lvert i - k\rvert = \lvert 3 - 6\rvert = 3$
7	5	3	1	4	8			
7	5	3	1	4*	8	2*		$\lvert j - l\rvert = \lvert 4 - 2\rvert = 2$ $\lvert i - k\rvert = \lvert 5 - 7\rvert = 2$
7	5	3	1	4*	8	6*		$\lvert j - l\rvert = \lvert 4 - 6\rvert = 2$ $\lvert i - k\rvert = \lvert 5 - 7\rvert = 2$
7	5	3	1	4	8			*Backtrack*
7	5	3	1	4				*Backtrack*
7	5	3	1	6				
7*	5	3	1	6	2*			$\lvert j - l\rvert = \lvert 1 - 2\rvert = 1$ $\lvert i - k\rvert = \lvert 7 - 6\rvert = 1$
7	5	3	1	6	4			
7	5	3	1	6	4	2		
7	5	3*	1	6	4	2	8*	$\lvert j - l\rvert = \lvert 3 - 8\rvert = 5$ $\lvert i - k\rvert = \lvert 3 - 8\rvert = 5$
7	5	3	1	6	4	2		*Backtrack*
7	5	3	1	6	4			*Backtrack*
7	5	3	1	6	8			
7	5	3	1	6	8	2		
7	5	3	1	6	8	2	4	**SOLUTION**

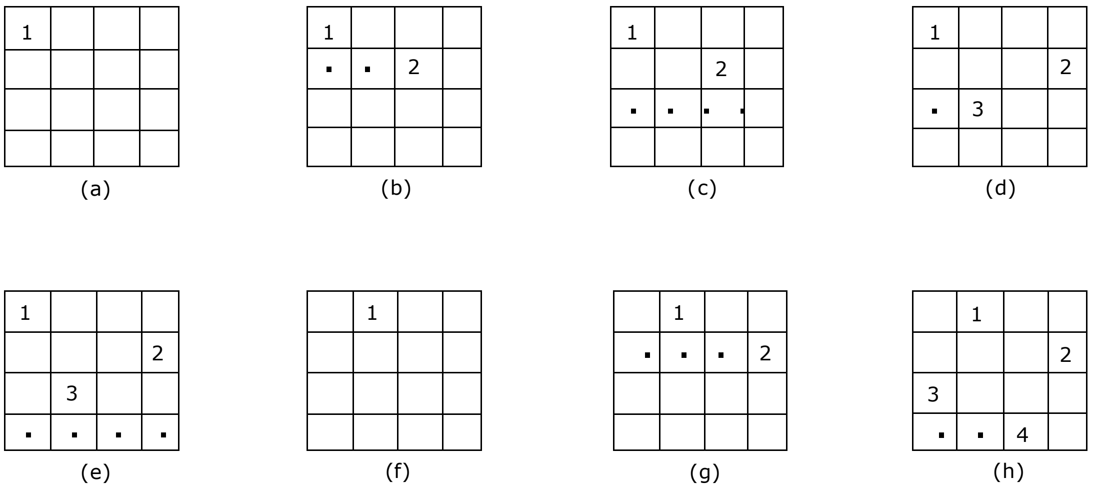
* indicates conflicting queens.

On a chessboard, the **solution** will look like:

						*	
				*			
		*					
*							
					*		
							*
	*						
			*				

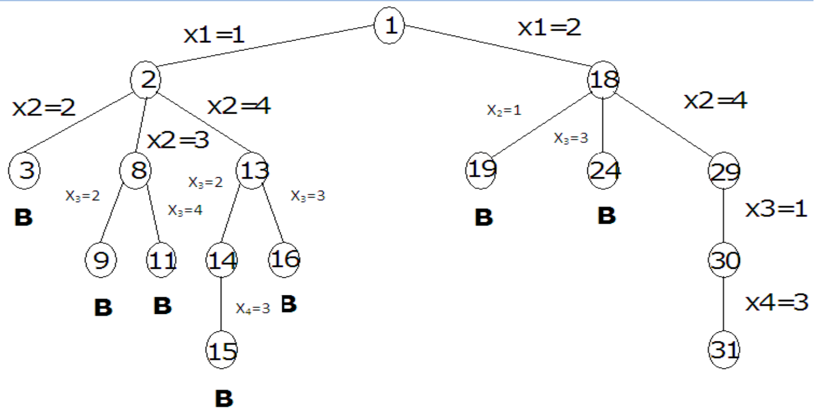
4 – Queens Problem:

Let us see how backtracking works on the 4-queens problem. We start with the root node as the only live node. This becomes the E-node. We generate one child. Let us assume that the children are generated in ascending order. Let us assume that the children are generated in ascending order. Thus node number 2 of figure is generated and the path is now (1). This corresponds to placing queen 1 on column 1. Node 2 becomes the E-node. Node 3 is generated and immediately killed. The next node generated is node 8 and the path becomes (1, 3). Node 8 becomes the E-node. However, it gets killed as all its children represent board configurations that cannot lead to an answer node. We backtrack to node 2 and generate another child, node 13. The path is now (1, 4). The board configurations as backtracking proceeds is as follows:



The above figure shows graphically the steps that the backtracking algorithm goes through as it tries to find a solution. The dots indicate placements of a queen, which were tried and rejected because another queen was attacking.

In Figure (b) the second queen is placed on columns 1 and 2 and finally settles on column 3. In figure (c) the algorithm tries all four columns and is unable to place the next queen on a square. Backtracking now takes place. In figure (d) the second queen is moved to the next possible column, column 4 and the third queen is placed on column 2. The boards in Figure (e), (f), (g), and (h) show the remaining steps that the algorithm goes through until a solution is found.



Portion of the tree generated during backtracking

Complexity Analysis:

$$1 + n + n^2 + n^3 + \ldots\ldots\ldots\ldots + n^n = \frac{n^{n+1} - 1}{n - 1}$$

For the instance in which n = 8, the state space tree contains:

$$\frac{8^{8+1} - 1}{8 - 1} = 19,\ 173,\ 961 \text{ nodes}$$

Program for N-Queens Problem:

```
# include <stdio.h>
# include <conio.h>
# include <stdlib.h>

int x[10] = {5, 5, 5, 5, 5, 5, 5, 5, 5, 5};

place (int k)
{
        int i;
        for (i=1; i < k; i++)
        {
                if ((x [i] == x [k]) || (abs (x [i] – x [k]) == abs (i - k)))
                return (0);
        }
        return (1);
}
nqueen (int n)
{
        int m, k, i = 0;
        x [1] = 0;
        k = 1;
        while (k > 0)
        {
                x [k] = x [k] + 1;
                while ((x [k] <= n) && (!place (k)))
                        x [k] = x [k] +1;
                if(x [k] <= n)
                {
                        if (k == n)
                        {
                                i++;
                                printf ("\ncombination; %d\n",i);
                                for (m=1;m<=n; m++)
                                printf("row = %3d\t column=%3d\n", m, x[m]);
                                getch();
                        }
                        else
                        {
                                k++;
                                x [k]=0;
                        }
                }

                else
                        k--;
        }
        return (0);
```

```
 }
main ()
{
        int n;
        clrscr ();
        printf ("enter value for N: ");
        scanf ("%d", &n);
        nqueen (n);
 }
```

Output:

```
Enter the value for N: 4

Combination: 1                      Combination: 2

Row  =  1     column = 2                  3
Row  =  2     column = 4                  1
Row  =  3     column = 1                  4
Row  =  4     column = 3                  2
```

For N = 8, there will be 92 combinations.

Sum of Subsets:

Given positive numbers wi, $1 \le i \le n$, and m, this problem requires finding all subsets of w_i whose sums are 'm'.

All solutions are k-tuples, $1 \le k \le n$.

Explicit constraints:

- $x_i \in \{j \mid j \text{ is an integer and } 1 \le j \le n\}$.

Implicit constraints:

- No two x_i can be the same.
- The sum of the corresponding w_i's be m.
- $x_i < x_{i+1}$, $1 \le i < k$ (total order in indices) to avoid generating multiple instances of the same subset (for example, (1, 2, 4) and (1, 4, 2) represent the same subset).

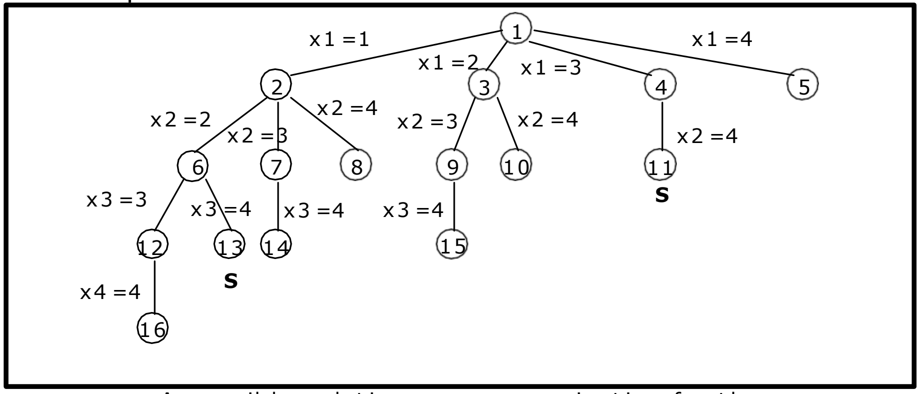
A better formulation of the problem is where the solution subset is represented by an n-tuple $(x_1, \ldots, x_n)$ such that $x_i \in \{0, 1\}$.

The above solutions are then represented by (1, 1, 0, 1) and (0, 0, 1, 1).

For both the above formulations, the solution space is 2^n distinct tuples.

For example, n = 4, w = (11, 13, 24, 7) and m = 31, the desired subsets are (11, 13, 7) and (24, 7).

The following figure shows a possible tree organization for two possible formulations of the solution space for the case n = 4.



A possible solution space organisation for the sum of the subsets problem.

The tree corresponds to the variable tuple size formulation. The edges are labeled such that an edge from a level i node to a level i+1 node represents a value for x_i. At each node, the solution space is partitioned into sub - solution spaces. All paths from the root node to any node in the tree define the solution space, since any such path corresponds to a subset satisfying the explicit constraints.

The possible paths are (1), (1, 2), (1, 2, 3), (1, 2, 3, 4), (1, 2, 4), (1, 3, 4), (2), (2, 3), and so on. Thus, the left mot sub-tree defines all subsets containing w_1, the next sub-tree defines all subsets containing w_2 but not w_1, and so on.

Graph Coloring (for planar graphs):

Let G be a graph and m be a given positive integer. We want to discover whether the nodes of G can be colored in such a way that no two adjacent nodes have the same color, yet only m colors are used. This is termed the m-colorabiltiy decision problem. The m-colorability optimization problem asks for the smallest integer m for which the graph G can be colored.

Given any map, if the regions are to be colored in such a way that no two adjacent regions have the same color, only four colors are needed.

For many years it was known that five colors were sufficient to color any map, but no map that required more than four colors had ever been found. After several hundred years, this problem was solved by a group of mathematicians with the help of a computer. They showed that in fact four colors are sufficient for planar graphs.

The function m-coloring will begin by first assigning the graph to its adjacency matrix, setting the array x [] to zero. The colors are represented by the integers 1, 2, . . . , m and the solutions are given by the n-tuple (x_1, x_2, . . ., x_n), where x_i is the color of node i.

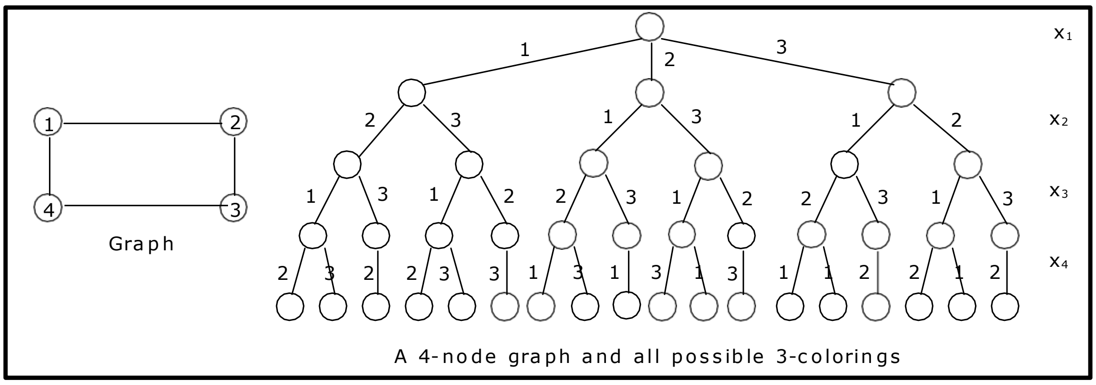
A recursive backtracking algorithm for graph coloring is carried out by invoking the statement mcoloring(1);

```
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.
}
```

Example:

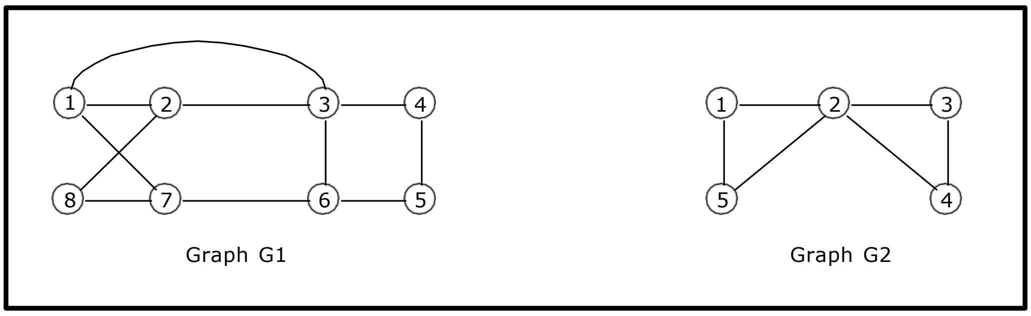
Color the graph given below with minimum number of colors by backtracking using state space tree.



A 4-node graph and all possible 3-colorings

Hamiltonian Cycles:

Let G = (V, E) be a connected graph with n vertices. A Hamiltonian cycle (suggested by William Hamilton) is a round-trip path along n edges of G that visits every vertex once and returns to its starting position. In other 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 the v_i are distinct expect for v_1 and v_{n+1}, which are equal. The graph G_1 contains the Hamiltonian cycle 1, 2, 8, 7, 6, 5, 4, 3, 1. The graph G_2 contains no Hamiltonian cycle.



Two graphs to illustrate Hamiltonian cycle

The backtracking solution vector $(x_1, \ldots \ldots x_n)$ is defined so that x_i represents the i^{th} visited vertex of the proposed cycle. If k = 1, then x_1 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 $x_1, x_2, \ldots, x_{k-1}$ and v is connected by an edge to k_{x-1}. The vertex x_n can only be one remaining vertex and it must be connected to both x_{n-1} and x_1.

Using NextValue algorithm we can 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).

The traveling salesperson problem using dynamic programming asked for a tour that has minimum cost. This tour is a Hamiltonian cycles. For the simple case of a graph all of whose edge costs are identical, Hamiltonian will find a minimum-cost tour if a tour exists.

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);
}
```

0/1 Knapsack:

Given n positive weights w_i, n positive profits p_i, and a positive number m that is the knapsack capacity, the problem calls for choosing a subset of the weights such that:

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

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.

Bounding functions are needed to kill some live nodes without expanding them. A good bounding function for this problem is obtained by using an upper bound on the value of the best feasible solution obtainable by expanding the given live node and any of its descendants. If this upper bound is not higher than the value of the best solution determined so far, than that live node can be killed.

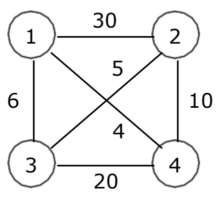
We continue the discussion using the fixed tuple size formulation. If at node Z the values of x_i, $1 \le i \le k$, have already been determined, then an upper bound for Z can be obtained by relaxing the requirements $x_i = 0$ or 1.

(Knapsack problem using backtracking is solved in branch and bound chapter)

7.8 Traveling Sale Person (TSP) using Backtracking:

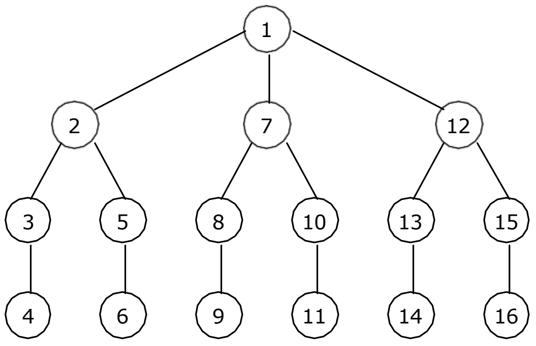
We have solved TSP problem using dynamic programming**.** In this section we shall solve the same problem using backtracking.

Consider the graph shown below with 4 vertices.



A graph for TSP

The solution space tree, similar to the n-queens problem is as follows:



We will assume that the starting node is 1 and the ending node is obviously 1. Then 1, {2, … ,4}, 1 forms a tour with some cost which should be minimum. The vertices shown as {2, 3, …. ,4} forms a permutation of vertices which constitutes a tour. We can also start from any vertex, but the tour should end with the same vertex.

Since, the starting vertex is 1, the tree has a root node R and the remaining nodes are numbered as depth-first order. As per the tree, from node 1, which is the live node, we generate 3 braches node 2, 7 and 12. We simply come down to the left most leaf node 4, which is a valid tour {1, 2, 3, 4, 1} with cost 30 + 5 + 20 + 4 = 59. Currently this is the best tour found so far and we backtrack to node 3 and to 2, because we do not have any children from node 3. When node 2 becomes the E-node, we generate node 5 and then node 6. This forms the tour {1, 2, 4, 3, 1} with cost 30 + 10 + 20 + 6 = 66 and is discarded, as the best tour so far is 59.

Similarly, all the paths from node 1 to every leaf node in the tree is searched in a depth first manner and the best tour is saved. In our example, the tour costs are shown adjacent to each leaf nodes. The optimal tour cost is therefore 25.