Unit 14 Backtracking

Structure:

14.1 Introduction

In the previous unit, we discussed the concept of dynamic programming and now in the search for fundamental principles of algorithm design, we will discuss about backtracking, which represents one of the most general techniques. Many problems which deal with searching for a set of solutions or which ask for an operational solution satisfying some constraints can be solved using the backtracking formulation. The name backtrack was first coined by D.H Lehmer in the 1950s. Early workers who studied the process were R.J. Woler, who gave an algorithmic account of it in 1960, and S. Golomb and L.Barmert who presented a very general description of it in the form of applications.

Objectives

After studying this unit, you should be able to:

- analyze the 8-queen problem
- solve the problem of the Sum of subsets
- use the applications of the Knapsack problem.

14.2 Backtracking Strategy

In many applications of the backtrack method, the desired solution is expressible as an *n*tuple ($x_1,……..,x_n$) where the x_is are chosen from some finite set s_i often the problem to be solved calls for finding one vector that maximizes a criterion function $p(x_1,....x_n)$. Sometimes it seeks all vectors that satisfy *P*. For example, sorting the array of integers in a [*1:n*] is a problem whose solution is expressible by an *n*-tuple, were x_i is the index in a of i^{th} smallest element. The criterion function *P* is the inequality $a[x_i] \leq a[x_i+1]$ for $1 \leq i < n$. The set S_i is finite and includes the integer *1*-through *n*. Through sorting is not usually one of the problems solved by backtracking, it is one example of a familiar problem whose solution can be formulated as an *n*-tuple. In this unit we study a collection of problems whose solution are best done using backtracking.

Suppose m_i is the size of set S_i. Then there are $m=m_1\ m_2\ \ldots\ldots.\ m_n$ n-tuples that are possible candidates for satisfying the function *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 its virtue, the ability to yield the same answer, for fewer in trials. Its basic idea is to build up the solution vector on component at a time and to use modified criterion functions $P_i\{x_1, \ldots.. x_i)$ (some times called bound functions) to test whether the vector being formed has any chance of success. The major advantage of this method is that 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 requires that all the solution satisfy a complex set of constraints. For any problem, these constraints can be divided into two categories; explicit and implicit.

Definition

Explicit constraints: Are rules that restrict each x_i, to take a value only from a given set,

common examples of explicit constraints

are $x_i > 0$ or S_i = {all non negative real no}

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

$d_i \leq x_i < u_i$ 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*.

Definition

The implicit constraints are tuples 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.

Self Assessment Question

1. ——————— are rules that restrict each x_i, to take a value only from a given set.

14.3 8-Queens Problem

A classic combinational problem is to place eight queens on *8x8* chessboard so that no two "attack" that is, so that no two of them are on the same row, colours or diagonal. Let us number the rows and columns of the chessboard *1* through *8* (Fig 14.1). The queens can also be numbered *1* through *8*. Since each queen must be on a different row, we can without loss of generality assume that queen *i* is to be placed on row *i*. All solutions to the 8-queens problem can therefore be represented as 8-tuples $(x_1 \ldots\ldots x_8)$, where x_i is the column on which queen *i* is placed. The explicit constraints using this formulation are $S_i = \{1, 2, 3, 4, 5, 6, 7, 8\}; 1 \le i \le 8$. Therefore, the solution space consists of 8^8-tuples. The implicit constraints for this problem are that 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. We see later how to formulate the second constraint in terms of the x_i. Expressed as an *8*-tuple, the solution in Fig. 14.1 is (4, 6, 8, 2, 7, 1, 3, 5).

	1	2	3	4	5	6	7	8
1				Q				
2						Q		
3								Q
4		Q						
5							Q	
6	Q							
7			Q					
8					Q			

Fig. 14.1: 8-Queens Problem

Algorithm 14.1

```
Algorithm Place (k, i)
// Returns true if a queen can be placed
// in k^th row and i^th columns otherwise
// it returns false, x[ ] is a global array
// whose first (k – 1) values have been set
// Abs (r) returns the absolute value of r
{
   For (i:=1) to k – l) do
         if ((x [j] – l // Two in the same column
               or (Abs (x [j] = Abs (j – k)))
               // or in the same diagonal
               Then return false.
   Return true:
}
Algorithm NQueens (k, n)
// Using backtracking this procedure prints all
// possible placements of n queens on an
// n x n chessboard so that they are
// non attacking
{
   for I : = 1 to n do
{
         if Place (k,j) then
         {
               x [ k] : = i;
               if (k=n) then write (x[1:n])
               else Nqueen (k+1, n);
         }
   }
}
```

At this point, we can see how effective function Nqueen is over the brute force approach. For an *8x8* chessboard, there are $\binom{64}{8}$ possible ways to place *8* pieces or approximately *4.4* billion *8* tuples to examine. However, by allowing only placements of queens on distinct rows and columns, we require the examination of at most *8!* or only *40,3208*-tuples.

We can estimate the number of nodes that will be generated by *N*Queens. The bounding function is static and no change is made to the function as the search proceeds. In addition, all nodes on the same level, of the separate space level, state space tree have the same degree.

As required placement of each queen on the chessboard was chosen randomly with each choice we kept track of the number of columns a queen could legitimately be placed on. These numbers are listed beneath each chessboard. The number following the vector represents the value that function estimate would produce from these sizes. The coverage of these five trials is *1625*.

So that estimated no. of unbounded nodes is only *2.34%* of the total no of nodes in the *8* queen state space tree.

Self Assessment Questions

2. The total no of nodes in the *8*-queen state space is ——————.
3. For an *8x8* chessboard there are —————possible ways to place *8* pieces.

14.4 Sum of Subsets

Given positive numbers w_i, $1 \leq i \leq n$, and m, this problem calls for finding all subsets of the w_i whose sums are m. For example, if $n = 4$, $(w_1, w_2, w_3, w_4) = (11, 13, 24, 7)$, 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 as described by the vector $(1,2,4)$ and $(3, 4)$. In general, all solutions are k-tuples $(x_1, x_2 \ldots\ldots x_k)$, $1 \leq k \leq n$, and different solutions may have different sized tuples. The explicit constraints $x_j \in \{j|j$ is an integer and $1 \leq j \leq n\}$. The implicit requires that no two be the same and that the sum of the corresponding will be m. Since we wish to avoid generating multiple instance of the same subject (e.g $(1, 2, 4)$ and $(1, 4, 2)$ represent the same subject), another implicit constraints that is imposed is $x_i<x_{i+1}$, $I \leq i < k$.

In another formulation of the sum of subsets problem, each solution subject is represented by *n*-tuple $(x_1, x_2, \ldots.. x_n)$ such that $x_i \in \{0,1\}, 1 \leq i \leq n$. Then, $x_i = 0$, if w_i is not chosen then the above instance are *(1, 1, 0, 1)* and *(0, 0, 1, 1)*. The formulation express all solutions using fixed sized tuple. Thus, we conclude that there may be several ways to formulate a problem so that all solutions are tuples that satisfy some constraints. One can verify that for both of the above formulations, the solutions space consists of 2^n distinct tuples.

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. For a given solution space, many tree organizations may be possible. The next two examples examine some of the ways to organize a solution into a tree.

We have two possible formulations of the solutions space for the sum of subsets problem. Fig 14.2 and Fig 14.3 show a possible tree organization for each of these formulations for the case $n=4$. The tree of Fig 14.2 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 solution space. The solution space is defined by all paths from the root node to any node in the tree, since any such path corresponds to a subset satisfying the explicit constraints. The possible paths are ( ) (this corresponds to the empty path from the root to itself), (*1*), (*1,2*), (*1,2,3*), (*1,2,3,4*), (*1,2,4*), (*1,3,4*), (*2*), (*2,3*) and so on. Thus, left most subtree defines all subsets containing w_i, the next subtree defines all subsets containing w_2 but not w_1, and so on.

The tree of Fig 14.3 corresponds to the fixed tuples size formulation. Edges from level *i* nodes to level $i + I$ nodes are labeled with the value of x_i, which is either zero or one. All paths from the root to a leaf node define the solution space. The left subtree of the root defines all subsets containing w_I, the right subtree defines all subsets not containing w_I and so on. Now there are 2^4 leaf nodes which represent *16* possible tuples.

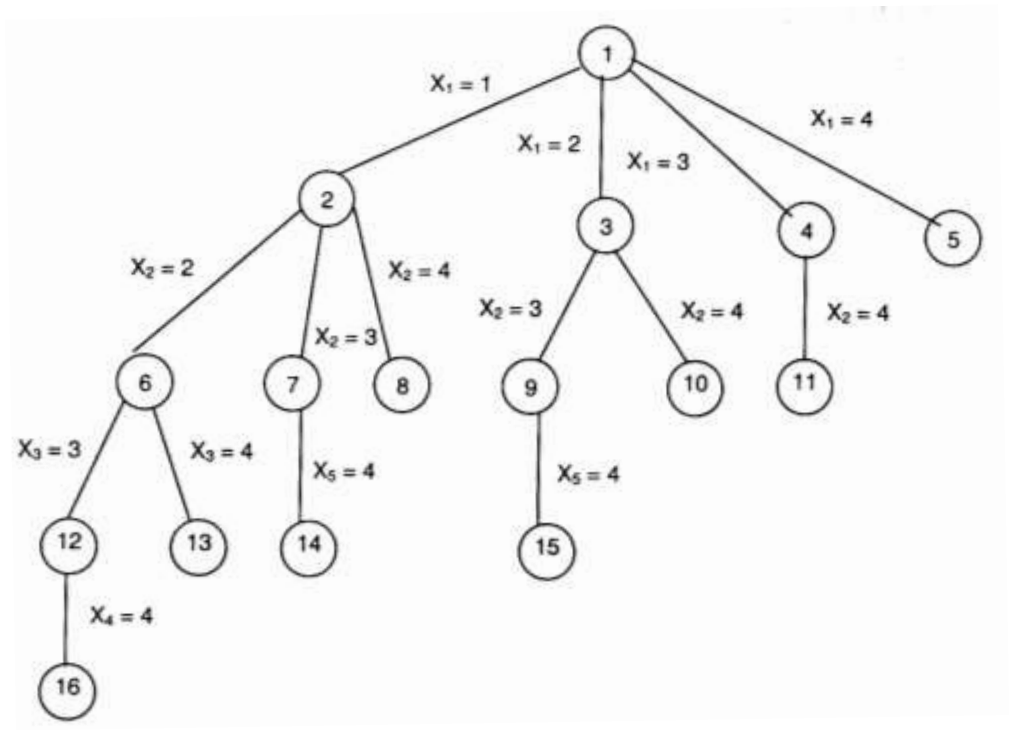

Fig. 14.2: A possible space organization for the sum of subsets problem. Nodes are numbered as in breath first search.

At this point, it is useful to develop some terminology regarding tree organization of solution spaces. Each node in this tree defines a problem state. All paths from the root to other nodes define the state space of the problem. Solution states are those problem states s for which the path from the root to s defines a tuple in the solution space. In the tree Fig 14.2, all nodes are solutions states whereas in the tree Fig 14.3, only leaf nodes are solution states. Answer states are those solution states s for which the path from the root to s defines a tuple that is a member of the set of solutions (i.e. it satisfies the implicit constraints) of the problem.

The tree organization of the solution space is referred to as the state space tree.

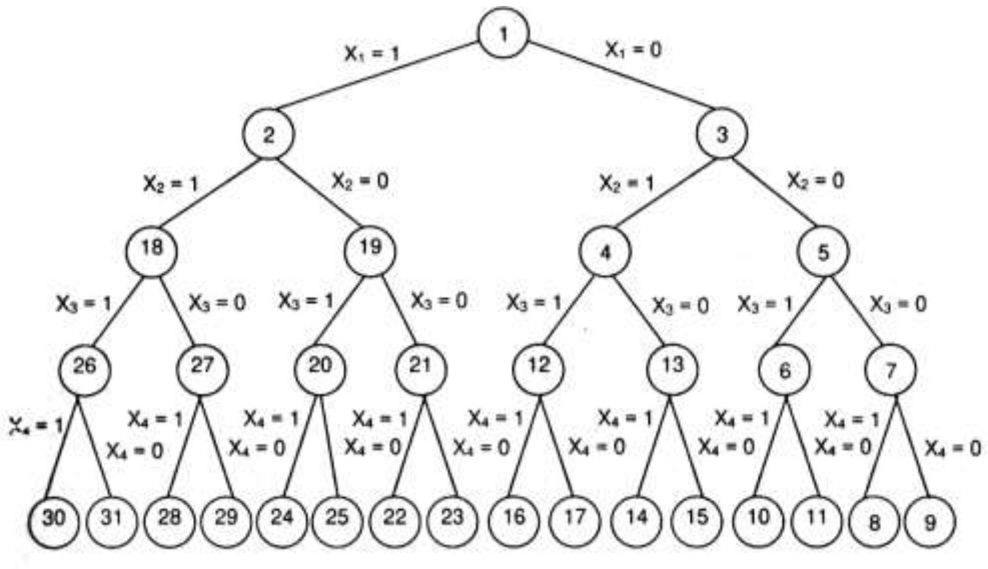

Fig. 14.3: Corresponds to the fixed tuples size formulation.

Suppose we are given distinct positive number and we desire to find all combinations of these numbers whose sums are m. This is called the sum of subsets problem. The example in above shows how we could formulate this problem using either fixed or variable sized tuples. We consider a back tracking solution using the fixed tuple size strategy. 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.

The children of any node in Fig 14.3 are easily generated. For a node at level *i,* the left child corresponds to $x_i = 1$ and the right to $x_i = 0$.

A simple choice for the bounding functions B_k *(*x_1*, ……. *x_i*)* = true if

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

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

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

The bounding functions we use are therefore B_k $(x_1, \ldots\ldots\ldots\ldots\ldots, x_k)$ = 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 algorithms 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 all that is needed is to directly use either of the backtracking scheme, a simpler algorithms results if we tailor either of these schemes to the problem at hand. This simplification results from the realization that if $x_k = 1$, then

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

For simplicity, we redefine the recursive scheme. The resulting algorithms is SumOfSub (Algorithm 14.2)

Algorithm 14.2

```
Algorithm SumOfSub (s,k,r)
// Find all subsets of w(k....n) that sum to m
// The values of x[j], 1 ≤ 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
// non decreasing order It is assumed that
// w[l] ≤ m and $\sum_{i=1}^{n} w[i] \geq m$
{
// Generate left child. Note: S+tw[j] ≤ m
// since $B_{k-1}$ is true.
x [k] : = 1;
If ($s + w[k] = m$) then write ($x [1: k]$);

// sub set found there is no recursive call
// here as $w[j] > 0$, $1 \le j \le n$
else if ($s + w[k] + w [k + 1] \le m$)
          then SumOfSub ($s + w[k], k + 1, r – w[k]$);
// Generate right child and evaluate $B_k$
if($s + r – w [k] \ge m$) and ($s + w [k + 1] \le m$ ) then
{
x[k]:=0
SumOfSub ($s, k + 1, r — w [k]$):
          }
}
```

Algorithm SumOfSub avoids computing $\sum_{i=1}^{k} w_i k_i$ *and* $\sum_{i=k+1}^{n} w_i$ each time by keeping those values in variables *s* and *r* respectively. The algorithm assumes $w_i \le m$ *and* $\sum_{i=1}^{n} w_i \ge m.$ The initial call is Sum Of Sub $\left(0, 1, \sum_{i=1}^{n} w_i\right)$.

It is interesting to note that the algorithm does not explicitly use the test $k > n$ to terminate the recursion. This test is not needed as an entry to the algorithm $s \ne m$ and $s + r \ge m$. Hence, $r \ne 0$ and so *k* can be no greater than *n*. Also note that in the else if statement, since $s + w_k < m$ and $s + r \ge m$, it follows that $r \ne w_k$ and hence $k + 1 \le n$, observe also that if $s + w_k = m$, then $x_{k+1}, \ldots\ldots\ldots\ldots\ldots\ldots x_n$ must be zero. These zeros are omitted from the output. In next statement we do not test for $\sum_{i=1}^{k} w_i x_i + \sum_{i=k+1}^{n} w_i \ge m$ as we already know $s + r \ge m$ and $x_k + 1$.

Self Assessment Questions

4. A simple choice for the bounding functions B_k $(x_1, \ldots\ldots x_i)$ is true if ——————
5. Given positive numbers w_i, $1 \le i \le n$, and *m*, the problem calls for finding all ——————of the w_i whose sums are *m*

14.5 Knapsack Problem

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

$$\sum_{1 \le i \le n} w_i \ x_i \le m \ and \ \sum_{1 \le i \le n} P_i x_i \text{ is minimized.}$$

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 source is the same as that for the sum of subset problem. Two possible tree organizations are possible, One corresponds to the fixed tuple size formulation and the other to the variable tuple size formulation. Backtracking algorithms for the knapsack problem can be arrived at using either of these two state space trees. Regardless of which is used, bounding functions are needed to help 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, then that live node can be killed.

We continued 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 requirement $x_i = 0$ on *1* to $0 \le x_i \le 1$ for $k + 1 \le i \le n$ and using the greedy algorithm to solve the relaxed problem. Function Bound (c_p, c_w, k) determines an upper bound on the best solution obtained by expanding any zone z at level of the state space tree. The object weights and profits are *w[i]* and *p[i]*. It is assumed that $\dfrac{p(i)}{w[i]} \ge \dfrac{p(i+1)}{w(i+1)}\ ; 1 \le i \le n$

```
Algorithm 14.3
Algorithm Bound ($c_p$, $c_w$, $k$)
// $c_p$ is the current profit, $c_w$ is the current
// weight total $k$ is the index of the last removed
// item : and $m$ is the knapsack size
{
b:=$c_p$;  c:= $c_w$;
for $i = k + 1$ to $n$ do
{
c: = c + w [i]
if (c < m) then b:= b + p [i]
else return b + (cl – (c-m)) / w[i] * p [i];
} return b:
{
```

From bound it follows that the bound for a feasible left child of a node z is the same as that for z. Hence, the bounding function need not be used whenever the backtracking algorithm makes a move to the left child of a node. The resulting algorithm is Bknap. It was obtained from the recursive backtracking scheme, Initially set f_p: = *-1*: This algorithm is invoked as Bknap (1, 0, 0):

When $f_p \neq —1$, $x[i]$, $1 \leq i \leq n$ is such that $\sum_{i=1}^{n} p[i]\, x[i] = f_p$. The path $y[i]$, $1 \leq i \leq k$, is the path to the current node. The current weight

$$c_w = \sum_{i=1}^{k-1} w[i]\, y[i] \;\; and \;\; c_p = \sum_{i=1}^{k-1} p[i]\, y[i]$$

```
Algorithm 14.4
Algorithm Bknap ($k$, $c_p$, $c_w$)
// $m$ is the size of the knapsack; is the
// no of weights and profits $w[\,]$ and $p[\,]$
// are the weights and profits
// p [i] /w [i] ≥ p[i +1] $f_w$ $\frac{p[i]}{w[i]} \geq p[i+1] \cdot f_w$ is the final
// weight of knapsack: $f_p$ is the final
// maximum profit x[k] = 0 if w[k] is not
// 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 ($c_p$, $c_w$, k) ≥ $f_p$) then
{
y[k]: = 0; if (k < n) then Bknop (k + 1, $c_p$, $c_w$):
If (($c_p$ > $f_p$) and (k = n) then
{
        $f_p$ : = $c_p$ : $f_w$ = $c_w$
        for j : =  1 to k do
        x[j] : = y[j]:
        }
    }
}
```

Self Assessment Question

6. If the object weights and profits are *w[i]* and *p[i]*, It is assumed that ———

14.6 Summary

- In this unit, we studied that any problem can be divided into two categories explicit and implicit.
- We also studied in this unit a classic combinational problem called as 8-queens problem.

- The concept of sum of subsets was discussed with suitable examples.
- The Knapsack problem and its algorithm was also discussed with suitable explanation

14.7 Terminal Questions

1. Write briefly the concept of 8-Queens Problem
2. Write the Algorithm of sum of Subsets
3. Briefly explain the Knapsack Problem

14.8 Answers

Self Assessment Questions

1. Explicit constraints
2. $1 + \sum_{j=0}^{7} \left[\prod_{i=0}^{j} (8 - i) \right] = 69,281$
3. $\begin{pmatrix} 64 \\ 8 \end{pmatrix}$
4. $\sum_{i=1}^{k} w_i \, x_i + \sum_{i=k+1}^{n} w_i \geq m$
5. subsets
6. $\frac{p(i)}{w[i]} \geq \frac{p(i+1)}{w(i+1)} \; ; 1 \leq i \leq n$

Terminal Questions

1. A classic combinational problem is to place eight queens on *8x8* chessboard so that no two “attack” that is, not two of them are on the same row, colours or diagonal. Let us number the rows and columns of the chessboard *1* through *8* (Fig 14.1). The queens can also be numbered *1* through *8*. Since each queen must be on a different row, we can without loss of generality assume that queen *i* is to be placed on row *i*. All solutions to the 8-queens problem can therefore be represented as 8-tuples $(x_1 \ldots\ldots x_8)$, where x_i is the column on which queen *i* is placed. The explicit constraints using this formulation are $S_i = \{1, 2, 3, 4, 5, 6, 7,$

8}; $1 \le i \le 8$. Therefore the solution space consists of 8^8-tuples. The implicit constraints for this problem are that 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. (Refer Section 14.3)

2. Refer Section 14.4
3. Refer Section 14.5