not attack each other. It will be interesting to compare various ideas for programming this problem. Therefore we will present three programs based on somewhat different representations of the problem.

4.5.1 Program 1

First we have to choose a representation of the board position. One natural choice is to represent the position by a list of eight items, each of them corresponding to one queen. Each item in the list will specify a square of the board on which the corresponding queen is sitting. Further, each square can be specified by a pair of coordinates (X and Y) on the board, where each coordinate is an integer between 1 and 8. In the program we can write such a pair as

X/Y

where, of course, the '/' operator is not meant to indicate division, but simply combines both coordinates together into a square. Figure 4.6 shows one solution of the eight queens problem and its list representation.

Having chosen this representation, the problem is to find such a list of the form

[X1/Y1, X2/Y2, X3/Y3, ..., X8/Y8]

which satisfies the no-attack requirement. Our procedure **solution** will have to search for a proper instantiation of the variables **X1, Y1, X2, Y2, ..., X8, Y8**. As we know that all the queens will have to be in different columns to prevent vertical attacks, we can immediately constrain the choice and so make the search task easier. We can thus fix the X-coordinates so that the solution list will fit the following, more specific template:

[1/Y1, 2/Y2, 3/Y3, ..., 8/Y8]

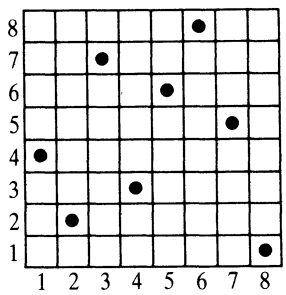

Figure 4.6 A solution to the eight queens problem. This position can be specified by the list **[1/4, 2/2, 3/7, 4/3, 5/6, 6/8, 7/5, 8/8]**.

We are interested in the solution on a board of size 8 by 8. However, in programming, in general, the key to the solution is often in considering a more general problem. Paradoxically, it is often the case that the solution for the more general problem is easier to formulate than that for the more specific, original problem; then the original problem is simply solved as a special case of the more general problem.

The creative part of the problem is to find the correct generalization of the original problem. In our case, a good idea is to generalize the number of queens (the number of columns in the list) from 8 to any number, including zero. The **solution** relation can then be formulated by considering two cases:

Case 1 The list of queens is empty: the empty list is certainly a solution because there is no attack.

Case 2 The list of queens is non-empty: then it looks like this:

[X/Y | Others]

In case 2, the first queen is at some square **X/Y** and the other queens are at squares specified by the list **Others**. If this is to be a solution then the following conditions must hold:

(1) There must be no attack between the queens in the list **Others**; that is, **Others** itself must also be a solution.

(2) X and Y must be integers between 1 and 8.

(3) A queen at square **X/Y** must not attack any of the queens in the list **Others**.

To program the first condition we can simply use the **solution** relation itself. The second condition can be specified as follows: Y will have to be a member of the list of integers between 1 and 8 – that is, **[1,2,3,4,5,6,7,8]**. On the other hand, we do not have to worry about X since the solution list will have to match the template in which the X-coordinates are already specified. So X will be guaranteed to have a proper value between 1 and 8. We can implement the third condition as another relation, **noattack**. All this can then be written in Prolog as follows:

```
solution( [X/Y | Others] ) :-
  solution( Others),
  member( Y, [1,2,3,4,5,6,7,8] ),
  noattack( X/Y, Others).
```

It now remains to define the **noattack** relation:

```
noattack( Q, Qlist)
```

Again, this can be broken down into two cases:

(1) If the list **Qlist** is empty then the relation is certainly true because there is no queen to be attacked.

(2) If **Qlist** is not empty then it has the form **[Q1 | Qlist1]** and two conditions must be satisfied:

   (a) the queen at Q must not attack the queen at Q1, and

   (b) the queen at Q must not attack any of the queens in **Qlist1**.

To specify that a queen at some square does not attack another square is easy: the two squares must not be in the same row, the same column or the same diagonal. Our solution template guarantees that all the queens are in different columns, so it only remains to specify explicitly that:

- the Y-coordinates of the queens are different, and
- they are not in the same diagonal, either upward or downward; that is, the distance between the squares in the X-direction must not be equal to that in the Y-direction.

Figure 4.7 shows the complete program. To alleviate its use a template list has

```
solution( [] ).

solution( [X/Y | Others] ) :-          % First queen at X/Y, other queens at Others
  solution( Others),
  member( Y, [1,2,3,4,5,6,7,8] ),
  noattack( X/Y, Others).              % First queen does not attack others

noattack( _, [] ).                     % Nothing to attack

noattack( X/Y, [X1/Y1 | Others] ) :-
  Y =\= Y1,                            % Different Y-coordinates
  Y1-Y =\= X1-X,                       % Different diagonals
  Y1-Y =\= X-X1,
  noattack( X/Y, Others).

member( X, [X | L] ).

member( X, [Y | L] ) :-
  member( X, L).

% A solution template

template( [1/Y1,2/Y2,3/Y3,4/Y4,5/Y5,6/Y6,7/Y7,8/Y8] ).
```

Figure 4.7 Program 1 for the eight queens problem.

been added. This list can be retrieved in a question for generating solutions. So we can now ask

```
?- template( S), solution( S).
```

and the program will generate solutions as follows:

```
S = [ 1/4, 2/2, 3/7, 4/3, 5/6, 6/8, 7/5, 8/1];
S = [ 1/5, 2/2, 3/4, 4/7, 5/3, 6/8, 7/6, 8/1];
S = [ 1/3, 2/5, 3/2, 4/8, 5/6, 6/4, 7/7, 8/1];
...
```

Exercise

4.6 When searching for a solution, the program of Figure 4.7 explores alternative values for the Y-coordinates of the queens. At which place in the program is the order of alternatives defined? How can we easily modify the program to change the order? Experiment with different orders with the view of studying the executional efficiency of the program.

4.5.2 Program 2

In the board representation of program 1, each solution had the form

```
[ 1/Y1, 2/Y2, 3/Y3, ..., 8/Y8]
```

because the queens were simply placed in consecutive columns. No information is lost if the X-coordinates were omitted. So a more economical representation of the board position can be used, retaining only the Y-coordinates of the queens:

```
[ Y1, Y2, Y3, ..., Y8]
```

To prevent the horizontal attacks, no two queens can be in the same row. This imposes a constraint on the Y-coordinates. The queens have to occupy all the rows 1, 2, ..., 8. The choice that remains is the *order* of these eight numbers. Each solution is therefore represented by a permutation of the list

```
[1,2,3,4,5,6,7,8]
```

Such a permutation, S, is a solution if all the queens are safe. So we can write:

```
solution( S) :-
  permutation( [1,2,3,4,5,6,7,8], S),
  safe( S).
```