Imperial College London – Department of Computing

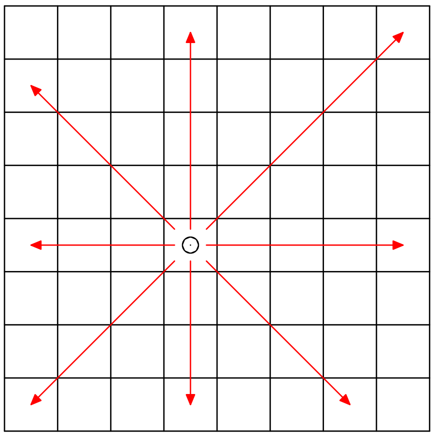
276: Introduction to Prolog
Exercise 5: N Queens

(Based on an Exercise and Solution by Robert Craven)

Solutions and Comments

November 2013

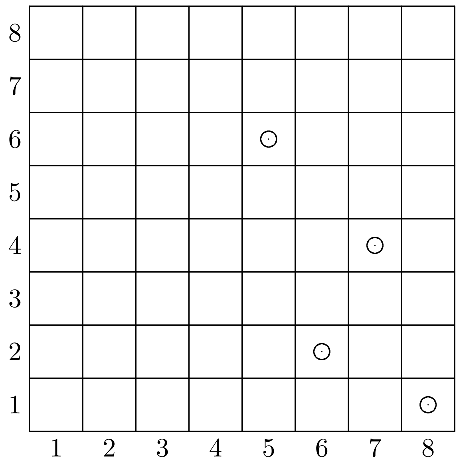
A chessboard is an 8×8 grid; the chess piece known as a 'queen' can attack along any horizontal, vertical and diagonal, as shown below:



The problem: place 8 queens on a chessboard, in such a way that no queen is attacking any other queen. Suppose that a queen in the m^{th} column and the n^{th} row is represented by the Prolog term `q(m,n)` (like Cartesian co-ordinates); a configuration of the board is to be represented by a Prolog list. For example, the list

```
[q(5,6),q(6,2),q(7,4),q(8,1)]
```

would represent the partially-completed non-solution:



Notice that if all solutions are forced to have the 'canonical' form

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

then we do *not* need to check that there are no vertical attacks. (Why?)

1. Write a program `no_attack/2`, which succeeds on a call

```
?- no_attack(PartialSoln, q(M,N))
```

when a queen can be placed in the m^{th} column and n^{th} row without falling under attack from any of the queens in `PartialSoln`. (`PartialSoln` will be ground.) For example (with reference to the grid above):

```
?- no_attack([q(7,4),q(8,1)], q(6,1)).
no
?- no_attack([q(7,4),q(8,1)], q(6,2)).
yes
```

You can assume that solutions are in the canonical form described above, so that checking for vertical attacks is unnecessary. You should assume that none of the queens in `PartialSoln` attack each other.

Solution. There are many different solutions to each of these exercises. Here are a few possibilities and variations.

```
% ----- no_attack/2  (Rob Craven)

no_attack([], _).
no_attack([q(X1,Y1)|Rest], q(X,Y)) :-
  Y \= Y1,
  Y1-Y =\= X1-X,
  Y1-Y =\= X-X1,
  no_attack(Rest, q(X,Y)).
```

Alternatively:

```
% ----- no_attack/2  (alternative)

no_attack(Partial, Queen) :-
  \+ (member(Qx, Partial),
      attack_pair(Qx, Queen)
     ).

attack_pair(q(X,_), q(X,_)).
attack_pair(q(_,Y), q(_,Y)).
attack_pair(q(X,Y), q(X1,Y1)) :-
  Y1-Y =:= X1-X.
attack_pair(q(X,Y), q(X1,Y1)) :-
  Y1-Y =:= X-X1.
```

The first clause for `attack_pair/2` is not strictly necessary but I will include it since it is cheap and perhaps makes things clearer.

`no_attack/2` above could also be written equivalently as:

```
% ----- no_attack/2  (alternative, equivalently)

no_attack(Partial, Queen) :-
  forall(member(Qx, Partial),
         \+ attack_pair(Qx, Queen)
        ).
```

Here is one more method, that combines `attack_pair` and `member` into one. This the solution I prefer (because it will be easiest to modify and generalise later).

```
% ----- no_attack/2

no_attack(Partial, Queen) :-
  \+ attack(Partial, Queen).

attack([Q|_], Queen) :-
  attack_pair(Q, Queen).
attack([_|Rest], Queen) :-
  attack(Rest, Queen).
```

2. Using your answer from (**1**), write a program `queens8/1`, which takes as argument a list representing an empty grid, and returns a solution to the 8 Queens problem. You should have a `Prolog` fact `template/1` in the program, to give the 'canonical' form of the solution, so that

```
  ?- template(Sol), queens8(Sol).
```

will give solutions.

Solution. The following solution by Rob Craven is *not* tail recursive, but represents one of the most intuitive ways to solve the problem.

```
% ----- queens8/1  (Rob Craven)

queens8([]).
queens8([q(X,Y)|Rest]) :-
  queens8(Rest),
  member(Y, [1,2,3,4,5,6,7,8]),
  no_attack(Rest, q(X,Y)).

% ----- template/1

template([q(1,_),q(2,_),q(3,_),q(4,_),q(5,_),q(6,_),q(7,_),q(8,_)]).
```

Note how this works: `template/1` gives the general form of the solution. It is the job of `queens8/1` to produce values for the columns by instantiating the variables.

How to make it tail recursive? Many ways. See discussion below.

Note Instead of

```
  member(Y, [1,2,3,4,5,6,7,8]),
```

we could write

```
  in_range(1,8,Y),
```

where `in_range/3` is defined as follows:

```
% ----------- in_range/3

in_range(Min, Max, Min) :-
  Min =< Max.  % to be on the safe side

in_range(Min, Max, X) :-
  Min < Max,
  NextMin is Min + 1,
  in_range(NextMin, Max, X).
```

3. Construct a one-line query which gives you the number of solutions to the 8 Queens problem.

Solution. This is easy. Given the previous answers, you can just write:

```
?- template(X), findall(X, queens8(X), Sols), length(Sols, N).
```

If you cannot assume that your program does not generate duplicate solutions, then

```
?- template(X), setof(X, queens8(X), Sols), length(Sols, N).
```

There is no point doing the (much more expensive) `setof` instead of `findall` if you don't have to.

4. Write a program `print_board/1` which will output a picture of a *completed* solution, given the canonical list representation of the solution as input. Something like the following should result:

```
_ _ _ _ _ _ _ Q
_ Q _ _ _ _ _ _
_ _ _ Q _ _ _ _
Q _ _ _ _ _ _ _
_ _ _ _ _ _ Q _
_ _ _ _ Q _ _ _
_ _ Q _ _ _ _ _
_ _ _ _ _ Q _ _
```

Prolog I/O primitives are *not* an examinable part of the course but what follows is a typical bit of procedural Prolog code.

Solution. There are many ways to do this. Here is one way, using `forall/2`:

```
% ----- forall/2

forall(X, Y) :-  \+ (X, \+ Y).

% ----- print_board/1 (Rob Craven)

print_board(Board) :-
  nl,
  forall( member(Row, [8,7,6,5,4,3,2,1]),
          ( member(q(Col,Row),Board),
            print_row(1, Col, 8)
          )
    ),
  nl.

% ----- print_row/3

print_row(End, _, Max) :-
  End > Max,  % could put a 'cut' here
  nl.
print_row(Col, QueenCol, Max) :-
  Col =< Max,    % could be omitted if 'cut' above
  (
    Col = QueenCol
    ->
      write('Q ')
    ;
      write('_ ')
  ),
  NextCol is Col + 1,
  print_row(NextCol, QueenCol, Max).
```

`print_board/1` could also have been expressed equivalently as follows:

```
% ----- print_board/1 (Rob Craven, equivalently)

print_board(Board) :-
  nl,
  forall( member(Row, [8,7,6,5,4,3,2,1]),
          forall( member(q(Col,Row),Board),
                  print_row(1, Col, 8)
                )
     ),
  nl.
```

And also like this:

```
% ----- print_board/1 (Rob Craven, equivalently again)

print_board(Board) :-
  nl,
  forall( ( member(Row, [8,7,6,5,4,3,2,1]),
            member(q(Col,Row),Board)
          ),
          print_row(1, Col, 8)
    ),
  nl.
```

Pick whichever you think is clearest.

Question: Why are they equivalent?

```
forall( P, forall( Q, R ) )
```

is the same as

```
\+ ( P, \+ forall( Q, R ) )
```

is the same as

```
\+ ( P, \+ \+ ( Q, \+ R ) )
```

is the same as

```
\+ ( P, ( Q, \+ R ) )
```

is the same as (conjunction is associative)

```
\+ ( ( P, Q), \+ R )
```

which is the same as

```
forall( ( P, Q), R )
```

In logic:

$$\begin{array}{rl} & \forall x\,(\,P(x) \rightarrow \forall y\,(\,Q(x,y) \rightarrow R(x,y)\,)\,) \\ \equiv & \forall x\,\forall y\,(\,P(x) \rightarrow (\,Q(x,y) \rightarrow R(x,y)\,)\,) \\ \equiv & \forall x\,\forall y\,(\,(P(x) \wedge Q(x,y)) \rightarrow R(x,y)\,) \end{array}$$

Rob Craven's

```
forall( P, ( Q, R ) )
```

is not *logically* equivalent but equivalent *procedurally* — but only because R is

```
print_row(1, Col, 8)
```

and so

- never fails
- is only used for its side-effects (printing)

We can also write `print_row` differently, by exploiting the fact that the board is represented in canonical form.

Since the board is in 'canonical form' it will look something like this

```
[q(1,5), q(2,3), q(3, 8), q(4,6), ... ]
```

So to print row N we just recurse down the list representing the board looking for an entry q(_,N), as follows:

```
% ----- print_board/1

print_board(Board) :-
  nl,
  forall( member(Row, [8,7,6,5,4,3,2,1]),
          print_row(Board, Row)
        ),
  nl.

% ----- print_row/2 (recursive)

print_row([], _Row) :-
  nl.
print_row([q(X,Y)|RestBoard], Row) :-
  (
   Y = Row
   ->
      write('Q ')
   ;
      write('_ ')
  ),
  print_row(RestBoard, Row).
```

Discussion: Some alternative (tail recursive) solutions

First, as above, using a 'template' for the final solution:

```
  ?- template(Sol), queens8_tr(Sol, []).
```

The second argument in `queens8_tr/2` represents the partial solution constructed so far, initially `[]`.

```
% ----- queens8_tr/2   % tail recursive

queens8_tr([], _PartialSol).

queens8_tr([q(X,Y)|Rest], PartialSol) :-
  member(Y, [1,2,3,4,5,6,7,8]),
  no_attack(PartialSol, q(X,Y)),
  queens8_tr(Rest, [q(X,Y)|PartialSol]).
```

Note(1) As written above, the `PartialSol` will be built up in reverse order, from position of the 8th queen back to the 1st. That does not matter because `no_attack/2` as written does not depend on the order that items appear in its `PartialSol` argument.

Note(2) We do not need an additional argument to represent the final solution, nor any call in the base case to reverse the partial solution into the right order. The 'template' already stores the solution. `queens8_tr/2` simply fills in values for the variables in the template.

Note(3) We can improve the efficiency slightly. Instead of generating all candidate values from 1 to 8 at every step, let us keep a list of candidate column numbers (columns not used so far).

```
% slightly optimised version

queens8_tr_opt(Sol) :-
  queens8_tr_opt(Sol, [], [1,2,3,4,5,6,7,8]).

% ----- queens8_tr_opt/3

queens8_tr_opt([], _Sol, _CandCols). % _CandCols will be []

queens8_tr_opt([q(X,Y)|Rest], PartialSol, CandCols) :-
  select(Y, CandCols, ColsRem),
  no_attack(PartialSol, q(X,Y)),
  queens8_tr_opt(Rest, [q(X,Y)|PartialSol], ColsRem).

% ---------- select/3

select(X, [X|Y], Y).

select(X, [U|Y], [U|Z]) :-
  select(X, Y, Z).
```

Note(4) The above was just for illustration. We do not need to keep an explicit list of candidate column values remaining: the candidate values are just those that have not been used in `PartialSol` so far.

```
% slightly optimised version (improved)

queens8_tr_opt(Sol) :-
  queens8_tr_opt(Sol, []).
```

```
% ----- queens8_tr_opt/2

queens8_tr_opt([], _Sol).

queens8_tr_opt([q(X,Y)|Rest], PartialSol) :-
  in_range(1, 8, Y),
      % alternatively:
      % member(Y, [1,2,3,4,5,6,7,8]),
  \+ member(q(_,Y), PartialSol),
  no_attack(PartialSol, q(X,Y)),
  queens8_tr_opt(Rest, [q(X,Y)|PartialSol], ColsRem).
```

Discussion: an alternative (tail recursive) solution

Suppose we don't use a 'template'. Let us solve the problem by building up a partial solution. It is easiest to go from the right, i.e. placing 8th queen in column 8, then 7th queen in column 7, …

```
% ------ queens8_tr/1  (recursive, no template)
queens8_tr(Sol) :-
  queens8_tr(8, [], Sol).

% -------  queens8_tr/3

% queens8_tr(M, PartialSoln, Final)
% place queen M;
% PartialSoln has queens in columns M+1 .. 8

queens8_tr(0, Sol, Sol).

queens8_tr(M, PartialSoln, Final) :-
  M > 0,
  in_range(1, 8, Y),
      % alternatively:
      % member(Y, [1,2,3,4,5,6,7,8]),
  \+ member(q(_,Y), PartialSoln),   % optional
  no_attack(PartialSoln, q(M,Y)),
  NextM is M-1,
  queens8_tr(NextM, [q(M,Y)|PartialSoln], Final).
```

Solve the problem by means of the query:

```
 ?- queens8_tr(Sol).
```

The printing of the board etc. is unchanged.

5. There is a redundancy in the data structures we are using to represent our chessboards. As the m^{th} element of our list represents the m^{th} column, we could alter our terms so that, for example, instead of

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

we have

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

This gives a much more compact representation. In a new file, rewrite your program to use this new representation. (New file so we can use the same predicates without name clashes.)

Every solution will thus be a *permutation* of

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

Solution. Here is a solution without a 'template'. One could do one with a 'template' too but I will not bother. The only thing we need to change is `attack/2` since that is the only component of the program that depends on how the board is represented.

```
% ----------- queens8_tr/1 and queens8_tr/3

queens8_tr(Sol) :-
  queens8_tr(8, [], Sol).

% queens8_tr(M, PartialSoln, Final)
% place queen M where
% PartialSoln has placed queens M+1 .. 8

queens8_tr(0, Sol, Sol).

queens8_tr(M, PartialSoln, Final) :-
  M > 0,
  in_range(1,8,Y),
      % alternatively:
      % member(Y, [1,2,3,4,5,6,7,8]),
  \+ member(Y, PartialSoln),  % optional, for efficiency
  Mx is M + 1,
  \+ attack(PartialSoln, Mx, q(M,Y)),
  NextM is M-1,
  queens8_tr(NextM, [Y|PartialSoln], Final).


% ----------- attack/3

% attack(PartialSoln, Mx, Queen)
% PartialSoln has queens placed in columns Mx .. 8

attack([Q|_], Mx, Queen) :-
  attack_pair(q(Mx,Q), Queen).
attack([_|Rest], Mx, Queen) :-
  MxNext is Mx + 1,
  attack(Rest, MxNext, Queen).


% ----------- attack_pair/2

attack_pair(q(X,_), q(X,_)).
attack_pair(q(_,Y), q(_,Y)).
attack_pair(q(X,Y), q(X1,Y1)) :-
  Y1-Y =:= X1-X.
attack_pair(q(X,Y), q(X1,Y1)) :-
  Y1-Y =:= X-X1.
```

For printing the Board:

```
% ----------- in_range_desc/3

in_range_desc(Min, Max, Max) :-
  Min =< Max.  % to be on the safe side
in_range_desc(Min, Max, X) :-
  Min < Max,
  NextMax is Max - 1,
  in_range_desc(Min, NextMax, X).


% -----------  print_board/1

print_board(Board) :-
  length(Board, N),
  nl,
  forall( in_range_desc(1,N,Row),
          print_row(Board, Row)
        ),
  nl.

% -----------  print_row/2

print_row([], _Row) :-
  nl.
print_row([Y|RestBoard], Row) :-
  (
   Y = Row
   ->
      write('Q ')
   ;
      write('_ ')
  ),
  print_row(RestBoard, Row).
```

Note that we could dispense with `in_range/3`. We need `in_range_desc/3` to print the board rows in the correct order, highest to lowest, but whether we use `in_range/3` or `in_range_desc/3` to generate candidate values in `queens8_tr/3` makes no difference. Both will generate all solutions eventually, though in a different order.

6. The 8 Queens problem can be generalized to the 'n Queens' problem, where instead of an 8×8 chessboard, there is an $n \times n$ grid and n queens. Modify your answer for (**5**) to solve the n Queens problem.

Solution. The generalisation of the previous program to n queens is trivial. (One could do one with a 'template' too but I will not bother.)

```
% ----------- queensN_tr/2 and queensN_tr/3

queensN_tr(N, Sol) :-
  integer(N), N > 0,    % to be on the safe side
  queensN_tr(N, [], Sol).

% queensN_tr(M, PartialSoln, Final)
% place queen M where
% PartialSoln has placed queens M+1 .. N

queensN_tr(0, Sol, Sol).

queensN_tr(M, PartialSoln, Final) :-
  M > 0,
  in_range(1,N,Y),    % this is the only change!
  \+ member(Y, PartialSoln),  % optional, for efficiency
  Mx is M + 1,
  \+ attack(PartialSoln, Mx, q(M,Y)),
  NextM is M-1,
  queensN_tr(NextM, [Y|PartialSoln], Final).
```

`attack/3` and `attack_pair/2` are unchanged (they do not depend on number of queens).

The printing of the board is unchanged (my version).