# Declarative Programming
# CS-205
# Part II: Logic Programming (Prolog)

Monika Seisenberger, Ulrich Berger

Department of Computer Science
College of Science
Swansea University

CS-205 - Search Strategies and Puzzles

## Search Strategies

One of Prolog's salient features is its built-in search mechanism.

In the ideal case this means that it suffices to specify a problem by a list of program clauses. Prolog will then automatically search for solutions and find them if they exist.

In many cases the search can be divided in two distinct phases:

- A *generating phase* where one specifies potential solutions in such a way that Prolog will find or generate all such potential solutions.
- A *testing phase* where the potential solutions are tested whether they are actual solutions. This test often involves arithmetic calculations.

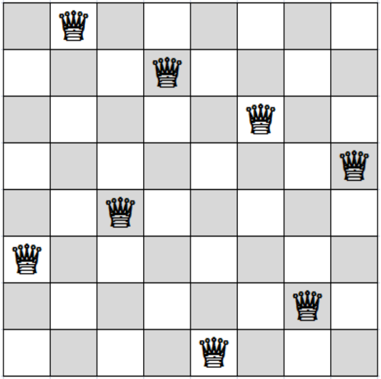
## Declarative vs. Procedural Programs

In a purely *declarative* logic program the generating phase and the testing phase are strictly separated and one speaks of a *generate-and-test* logic program.

Such programs are usually very short, elegant and easy to understand. However, they may be inefficient since too many potential solutions are generated most of which are not actual solutions.

In order to obtain better efficiency one may try to interleave the two phases in order to weed out unsuitable potential solutions at an early stage and thus narrow down the search space. Such programs have a more *procedural* character.

## Example: The *N* Queens Problem



Eight queens not attacking each other (codepumkin.com)

## Preliminaries

We need to place $N$ queens on an $N \times N$ chess board so that they don't attack each other.

We first exclude horizontal and vertical attacks. That is, we make sure that no two queens are on the same row or column. We therefore only generate potential solutions that are already free from horizontal and vertical attacks. We introduce some predicates that help achieving this.

## Selection from a list with remainder

`mem_rem(X, L, R)` means that `X` is a member of `L` and `R` is what remains of `L` if `X` is taken away.

```
mem_rem(H, [H|T], T).
mem_rem(X, [H|T], [H|R]) :- mem_rem(X, T, R).
```

Try:

```
?- mem_rem(2,[1,2,3],R).
?- mem_rem(X,[1,2,3],R).
```

## Generating permutations

`perm(L,P)` means that `P` is a permutation of `L`, that is, a rearrangement of `L`.

```
perm([], []).
perm(L, [X|P]) :-
  mem_rem(X, L, R),
  perm(R, P).
```

Try:

```
?- perm([1,2,3],P).
```

## Generating intervals

`interval(Left,Right,L)` means that

```
            L = [Left,Left+1,...,Right].
```

```
interval(Left,Right,[]) :- Left > Right.
interval(Left,Right,[Left|L]) :-
  Left =< Right,
  Left1 is Left + 1,
  interval(Left1,Right,L).
```

Try:

```
?- interval(1,4,L).
```

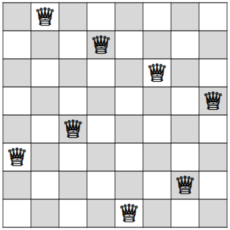
## Representing queens

We represent a queen that sits on

*row* `X` (counting from top to bottom) and

*column* `Y` (counting from left to right)

by the term `q(X,Y)`.



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

## Displaying queens

```
d_queens(Qs) :- length(Qs,N), display_queens(N,Qs).

display_queens(_,[]).
display_queens(N,[q(_,Y)|Qs]) :-
  display_queen(N,1,Y), display_queens(N,Qs).

display_queen(N, K, _) :- K > N, nl.
display_queen(N, K, Y) :-
  K =< N, q_or_dot(K, Y, C), write(C),
  K1 is K + 1, display_queen(N, K1, Y).

q_or_dot(K,Y,' Q') :- K =:= Y.
q_or_dot(K,Y,' .') :- K =\= Y.

?- d_queens([q(1,2),q(2,1),q(3,3)]).
 . Q .
 Q . .
 . . Q
```

## Avoiding diagonal attacks

- Checking that there is no diagonal attack between two queens

```
safe(q(X, Y), q(X1, Y1)) :-
  abs(X-X1) =\= abs(Y-Y1).
```

- Checking that a queen is not attacked diagonally by a list of queens

```
safe_all(_, []).
safe_all(Q, [Q1|Qs]) :-
  safe(Q, Q1),
  safe_all(Q, Qs).
```

- No diagonal attacks between lists of lists of queens

```
safe_all_all([]).
safe_all_all([Q|Qs]) :-
  safe_all(Q, Qs),
  safe_all_all(Qs).
```

## Declarative solution

Following the generate-and-test strategy we generate all placements of queens that avoid horizontal or vertical attacks and then check that there are no diagonal attacks either.

```
queens_dec(N, Qs) :-
  interval(1, N, Xs),
  perm(Xs, Ys),
  make_queens(Xs, Ys, Qs), % generate potential sol.
  safe_all_all(Qs).   % test that no diagonal att.

make_queens([], [], []).
make_queens([X|Xs], [Y|Ys], [q(X,Y)|Qs]) :-
  make_queens(Xs, Ys, Qs).
```

# A solution of the 12 Queens Problem

```
?- queens_dec(12,Qs), d_queens(Qs).

Q . . . . . . . . . . .
 . . Q . . . . . . . . .
 . . . . Q . . . . . . .
 . . . . . . . Q . . . .
 . . . . . . . . . Q . .
 . . . . . . . . . . . Q
 . . . . . Q . . . . . .
 . . . . . . . . . . Q .
 . Q . . . . . . . . . .
 . . . . . . Q . . . . .
 . . . . . . . . Q . . .
 . . . Q . . . . . . . .

Qs = [q(1, 1), q(2, 3), q(3, 5), q(4, 8),
      q(5,10), q(6,12), q(7, 6), q(8,11),
      q(9, 2), q(10,7), q(11,9), q(12,4)]
```

## Procedural (and more efficient) solution

Idea: We iteratively place a queen in each row making sure it is not attacked by the previously placed queens.

```
queens_pro(N, Qs) :-
  interval(1, N, Ys),
  queens_part(N, 1, Ys, [], Qs).
```

`queens_part(N,X,Ys,PQs,Qs)` means that `Qs` is a list of queens on rows `X` to `N` with `Y`-coordinates in `Ys` and such that they are not attacked by the previously placed queens `PQs`.

```
queens_part(N,X,_,_,[]) :- X > N.

queens_part(N, X, Ys, PQs, [Q|Qs]) :-
  X =< N,
  mem_rem(Y, Ys, Ys1),
  Q = q(X, Y),
  safe_all(Q, PQs),
  X1 is X + 1,
  queens_part(N, X1, Ys1, [Q|PQs], Qs).
```

# A solution of the 20 Queens Problem

```
?- queens_pro(20,Qs), d_queens(Qs).
 Q . . . . . . . . . . . . . . . . . . .
 . . Q . . . . . . . . . . . . . . . . .
 . . . . Q . . . . . . . . . . . . . . .
 . Q . . . . . . . . . . . . . . . . . .
 . . . Q . . . . . . . . . . . . . . . .
 . . . . . . . . . . . . Q . . . . . . .
 . . . . . . . . . . . . . . Q . . . . .
 . . . . . . . . . . . Q . . . . . . . .
 . . . . . . . . . . . . . . . . . Q . .
 . . . . . . . . . . . . . . . . . . . Q
 . . . . . . . . . . . . . . . . Q . . .
 . . . . . . . . Q . . . . . . . . . . .
 . . . . . . . . . . . . . . . Q . . . .
 . . . . . . . . . . . . . . . . . . Q .
 . . . . . . . Q . . . . . . . . . . . .
 . . . . . . . . . Q . . . . . . . . . .
 . . . . . . Q . . . . . . . . . . . . .
 . . . . . . . . . . . . . Q . . . . . .
 . . . . . Q . . . . . . . . . . . . . .
 . . . . . . . . . . Q . . . . . . . . .
```