MA410 Prolog Practical: CSPs - The N-Queens Problem

1 Introduction

Both N-Queens and Sudoku are examples of Constraint Satisfaction Problems (or CSPs for short).

A CSP is defined as a triple $\langle X, D, C\rangle$:

X	set of variables
$\mathcal{D}$	domain of values
C	Constraints of form $\langle t, R\rangle$ where t = tuples of variables R= tuples of relations.

E.g. 4-Queens problem:

$X = \{Q_1, Q_2, Q_3, Q_4\}$
$\mathcal{D} = \{(1, 2, 3, 4)^4\}$
$C = Q_1 \neq Q_2 \neq Q_3 \neq Q_4$ and
for $i = 0...4$ and $j = (i + 1)...4, k = j - i$,
$Q_i \neq Q_j + k$ and $Q_i \neq Q_j - k$.

We use a forward-checking with backtracking technique in prolog to solve the N-Queens problem.

∘ Download and edit `lab_nqs.pl`.

2 Create Initial Board Positions

We need to define a blank `N` × `N` board.

Each position `P` on the board will be a coupled list of the form `[R,C]` where `R` denotes row and `C` column.

Write predicates:

(a) `listNums(N1,N2,L)`: `L`= [`N1`,...,`N2`].

(b) `boardPos([R,C],N)`: `[R,C]` is a board position on an N × N board.

(c) `allBoardPos(N,L)`: `L`=`[[1,1],....,[N, N]]`.

3 Attacking Predicates

We create predicates related to checking whether positions attack each other or not.

(a) Write the following predicates:

	returns true if ***P1=`[R1,C1]`*** & ***P2=`[R2,C2]`*** are
`on_same_col([R1,C1],[R2,C2])`	on same column.
`on_same_row([R1,C1],[R2,C2])`	on same row.
`on_same_diag([R1,C1],[R2,C2])`	on same diagonal.
`attack(P1,P2)`	attack each other.

For all other values they return false.

Test your code in prolog, e.g.

```
?- on_same_col([2,3],[1,3]).
?- on_same_diag([1,4],[2,3]).
?- attack([1,2],[4,5]).
```

What happens if you type: `?- attack([2,3], P).`

(b) Create predicate `attack_list_restricted(L,P1,P2)` where `P1`, `P2` are restricted to `L` (a list of positions) and then checks if they attack each other.

4 Placing Queens on Board

(a) *We start with a general position `[R,C]` & try remove all positions from our current available list to create an updated list.*

Create `tryPlaceQueen([R,C], LAv, LNewAv)` where `LAv` is list of available spaces before, and `LNewAv` is list obtained from `LAv` by removing all positions in `LAv` attacked by `[R,C]` .

(b) *Next, we create a recursive predicate to go through each column in turn so as to place a queen:*

`tryColsInTurn(C, LAv, LTkn, N)` where `C` = current column, `LAv` = available positions, `LTkn` = taken positions, `N` = final column.

5 Solving N-Queens

(a) *To solve the N-Queens problem, we create the board and then starting at C = 1, run the predicate `tryColsInTurn` stopping at C=N.*

Create predicates:

i. `solveNQs(N,LTaken)`: `LTaken` = list of Queen positions solving N × N problem.

ii. `numNQSols(N,M)`: `M` = no. of `N`-Queens solutions

(We write code to print solutions to N-Queens problem.)

iv. `showQueen(P,L)`: writes `[Q]` to screen if `P` ∈ `L` and otherwise writes `[ ]`.

v. `printPositions(L,[R,C],N)`: writes `[Q]` at position `[R,C]` if `[R,C]` ∈ `L`, and otherwise prints `[ ]`. At end of each row, it prints a new line. Repeats process until position `[N,N]`.

vi. `solveNQs(N)`: solves N-Queen's problem and writes solution to screen.

(b) Switch font in Prolog to `Courier` and try solve for different values of `N`.