Chess Loop Puzzle

Resolução de Problema de Decisão usando Programação em Lógica com Restrições

Nuno Lopes and Francisco Ferreira

Faculdade de Engenharia da Universidade do Porto

Resumo Este artigo complementa o segundo projecto da Unidade Curricular de Programação em Lógica, do Mestrado Integrado em Engenharia Informática e de Computação. O projecto consiste num programa, escrito em Prolog, capaz de resolver qualquer Chess Loop Puzzle, que é um problema de decisão.

Keywords: chess loop, sicstus, clpfd, prolog, feup

1 Introdução

O objetivo do trabalho era a implementação de uma abordagem a um problema de decisão ou otimização em Prolog com restrições. Assim sendo, o grupo teve a oportunidade de optar por um destes dois temas, sendo que o grupo optou por implementar o problema Chess Loop, que é um Puzzle 2D (problema de decisão).

O puzzle consiste num tabuleiro de dimensões variadas com dois tipos de peças do famoso jogo de Xadrez, entre as quais estão presentes o Rei, a Rainha, a Torre, o Cavalo e o Bispo. Este artigo explica detalhadamente o funcionamento deste puzzle, fornecendo também uma explicação sobre como visualizar o tabuleiro e a respetiva solução. Finalmente, existe ainda uma secção onde serão fornecidas estatisticas de resolução de diferentes puzzles.

2 Descrição do Problema

Em cada puzzle do tipo Chess Loop, o usuário é pedido que introduza num tabuleiro de Xadrez dois tipo de peças sendo que cada peça ataque uma e uma só peça do outro tipo de peças (e nenhuma do mesmo tipo) e que os ataques formem um loop.

3 Abordagem

Na implementação do puzzle em Prolog o grupo optou por utilizar uma lista que é manipulada ao longo do programa. No entanto, para facilitar a visualização das soluções encontradas e confirmar a veracidade destas a representação do puzzle é feita através de uma lista de listas, assemalhando-se bastante ao formato de um tabuleiro.

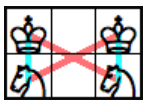


Figura 1. Exemplo de uma solução num tabuleiro de 2x3 com 2 Reis e 2 Cavalos

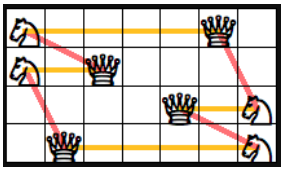


Figura 2. Exemplo de uma solução num tabuleiro de 4x7 com 4 Rainhas e 4 Cavalos

Figura 3. Visualização do exemplo da figura 1

3.1 Variáveis de Decisão

Como foi dito em cima, o grupo trabalha os tamanhos e os dominios com uma lista e não com a lista de listas, que é uma mera representação.

Assim, cada solução tem tamanho igual ao número de colunas do tabuleiro multiplicado pelo número de linhas do mesmo. Relativamente ao domínio da mesma, este vai de 1 até ao tamanho da lista. No que diz respeito às peças, o domínio de cada uma delas vai de 1 ao tamanho da lista de soluções, pois diz respeito à posição dessa mesma peça na lista.

```
solveBoard(NumRows, NumColumns, NumPieces, TypePiece1,
    TypePiece2) :-
     BoardSize is NumRows*NumColumns,
     length(Piece1A, NumPieces),
     length(Piece2A, NumPieces),
     length(Piece1B, NumPieces),
     length(Piece2B, NumPieces),

     domain(Piece1A, 1, BoardSize),
     domain(Piece1B, 1, BoardSize),
     domain(Piece2A, 1, BoardSize),
     domain(Piece2B, 1, BoardSize),
     (...)
```

Figura 4. Excerto de Código onde são definidos os domínios e os tamanhos das peças do jogo.

3.2 Restrições

1. Nenhuma peça pode ter a mesma posição:
   As posições das peças estão guardadas em listas por tipo de peça, para além da lista com todas as posições que representa o tabuleiro. Assim, com recurso ao predicado all_different garantimos que não há peças nas mesmas posições.

```
all_different(Piece1A),
all_different(Piece1B),
all_different(Piece2A),
all_different(Piece2B),
```

Figura 5. Excerto de Código onde é implementada a restrição relativa a peças em diferentes posições.

2. Para cada peça garatir que:
   (a) Não pode atacar nenhuma peça do mesmo tipo (**pieceNoAttackSelf**)
   Esta restrição é feita tendo em conta a lista de peças de um certo tipo e com o predicado noAttackPositions são eliminadas as peças do mesmo tipo que não estejam em posições seguras.
   (b) Tem que atacar uma e uma só peça do outro tipo (**pieceAttack**)
   Esta restrição é feita tendo em conta a lista de peças de um certo tipo e com o predicado AttackPositions que restringe cada peça a atacar uma das peças da equipa adversária, com a ajuda de um contador.

```
% Piece A attack constraints
pieceNoAttackSelf(Piece1A, NumColumns, TypePiece1),
pieceAttack(Piece1A, Piece2A, NumColumns,
    TypePiece1),

% Piece B attack constraints
pieceNoAttackSelf(Piece2B, NumColumns, TypePiece2),
pieceAttack(Piece2B, Piece1B, NumColumns,
    TypePiece2),
```

Figura 6. Excerto de Código onde são implementadas restrições relativas aos ataques das peças.

3. As peças têm que formar um loop.
   Esta restrição consistiu na implementação de duas listas temporárias que auxiliam no estabelecimento de um circuito que tem em conta a posição de origem.

```
checkForLoop(Piece1A, Piece2A, Piece1B, Piece2B,
    NumPieces),
```

Figura 7. Excerto de Código onde é implementada a restrição relativa ao loop das peças.

4 Visualização da Solução

De forma sucinta, a visualização da solução é feita convertendo a lista com as soluções para uma lista de listas com o mesmo formato que o tabuleiro pretendido(mesmo número de linhas e de colunas). Posteriormente procede-se a impressão da lista de listas, linha a linha e da esquerda para a direita, substituindo os tipos de peças pelos respetivos carateres em unicode e as posições sem um tipo de peça definido são substituidas por espaços. Nas figuras seguintes é possivel ver exemplos do resultado final onde o utilizador vai obter a solução.

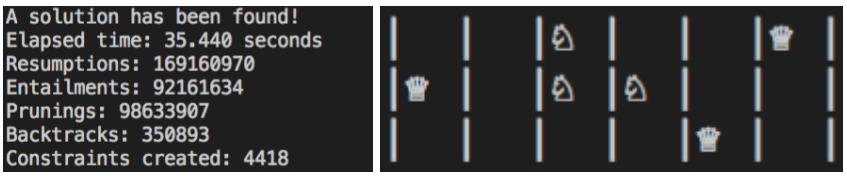
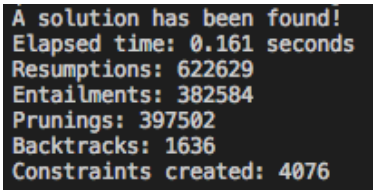


Figura 8. Visualização de uma solução de num tabuleiro de 3 por 6, com 3 Cavalos e 3 Rainhas

5 Resultados

Os resultados obtidos pelo grupo foram bastante conclusivos como pode ser facilmente verificado na figura abaixo.



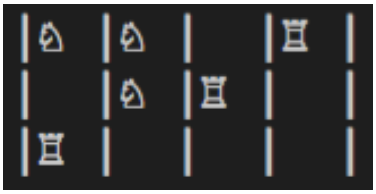
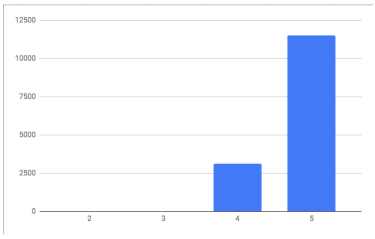
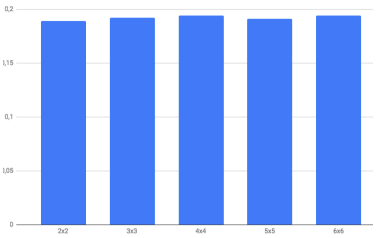


Figura 9. Visualização de uma solução de num tabuleiro de 3 por 4, com 3 Cavalos e 3 Torres



2	0,01
3	12
4	3138
5	11500

Figura 10. Variação do número de peças relativamente aos tempos de pesquisa de soluções (em segundos), mantendo o tamanho do tabuleiro e o tipo das peças



2x2	0,189
3x3	0,192
4x4	0,194
5x5	0,191
6x6	0,194

Figura 11. Variação do tamanho do tabuleiro relativamente aos tempos de pesquisa de soluções (em segundos), mantendo o mesmo número de peças (2 peças) e do tipo das peças

Podemos concluir que o número de peças de cada tipo tem um enorme impacto nos tempos de busca de soluções, pois é verificado um aumento exponencial destes mesmos tempos com um simples aumento do número de peças.

Relativamente a fatores como o tipo de peças e o tamanho do tabuleiro concluiu-se que as suas variações não são significativas nos tempos de pesquisa de solução por si só.

6 Conclusões

Com o término deste projeto o grupo conclui que o uso de Prolog com restrições é útil em certas situações.

Tal como no projeto anterior, verificámos em primeira mão que relativamente a outras linguagens de programação, o Prolog simplifica no desenvolvimento de certos aspetos em programas mais complexos, mas por outro lado também dificulta aspetos que noutras linguagens seriam praticamente instantâneos, como por exemplo, a simples representação de uma lista de listas. Concluindo, foi verificada a eficácia do Prolog, se bem que comparativamente ao primeiro projeto e com a adição de restrições, o grupo notou uma ligeira diminuição da velocidade de execução na obtenção de soluções.

Referências

1. Chess Loop rules,
   https://www2.stetson.edu/~efriedma/puzzle/chessloop/
2. SICStus Prolog, https://sicstus.sics.se/
3. url SWI-Prolog, http://www.swi-prolog.org/

Anexo

Código fonte

chessloop.pl

```
:- use_module(library(lists)).
:- use_module(library(clpfd)).
:- use_module(library(random)).

:- include('board.pl').
:- include('moves.pl').
:- include('solver.pl').
:- include('utils.pl').

re :-
    reconsult('chessloop').

chessloop :-
    nl,
    write('------------------------ CHESS LOOP PUZZLE------------------------'), nl, nl,
    write('> To solve a board and see the solution visually use:'), nl,
    write('solveBoard(NumRows, NumColumns, NumPieces, TypePiece1, TypePiece2).'), nl, nl,
    write('     Example:'), nl,
    write('     - Place 2 knights and kings on a 2x3 chess board'), nl,
    write('     solveBoard(2, 3, 2, kings, knights).'), nl, nl,

    write('> To generate a random problem and a solution use:'), nl,
    write('chessloopRandom.'), nl, nl,

    write('> If you want to check all solutions manually use:'), nl,
    write('solveBoard(NumRows, NumColumns, NumPieces, TypePiece1, TypePiece2, Pieces1List,
        Pieces2List).'), nl, nl.

chessloopRandom :-
    random(2, 7, NumRows),
    random(2, 7, NumColumns),
    random(2, 5, NumPieces),
    random(1, 5, NumTypePiece1),
    random(1, 5, NumTypePiece2),
    nth1(NumTypePiece1, [kings, knights, queens, bishops, rooks], TypePiece1),
    nth1(NumTypePiece2, [kings, knights, queens, bishops, rooks], TypePiece2),

    nl,
    write('- Placing '), write(NumPieces), write(' '), write(TypePiece1), write(' and '), write(
        TypePiece2),
    write(' on a '), write(NumRows), write('x'), write(NumColumns), write(' chess board.'), nl,
    nl,
```

```
solveBoard(NumRows, NumColumns, NumPieces, TypePiece1, TypePiece2).
```

solver.pl

```
% ---------------- BOARD SOLVER PREDICATE ----------------

solveBoard(NumRows, NumColumns, NumPieces, TypePiece1, TypePiece2) :-
    BoardSize is NumRows*NumColumns,
    length(Piece1A, NumPieces),
    length(Piece2A, NumPieces),
    length(Piece1B, NumPieces),
    length(Piece2B, NumPieces),

    domain(Piece1A, 1, BoardSize),
    domain(Piece1B, 1, BoardSize),
    domain(Piece2A, 1, BoardSize),
    domain(Piece2B, 1, BoardSize),

    all_different(Piece1A),
    all_different(Piece1B),
    all_different(Piece2A),
    all_different(Piece2B),

    % Check if the lists have same elements
    same_elements_list(Piece1A, Piece1B),
    same_elements_list(Piece2A, Piece2B),

    % Piece A attack constraints
    pieceNoAttackSelf(Piece1A, NumColumns, TypePiece1),
    pieceAttack(Piece1A, Piece2A, NumColumns, TypePiece1),

    % Piece B attack constraints
    pieceNoAttackSelf(Piece2B, NumColumns, TypePiece2),
    pieceAttack(Piece2B, Piece1B, NumColumns, TypePiece2),

    % Check if the positions form a loop
    checkForLoop(Piece1A, Piece2A, Piece1B, Piece2B, NumPieces),

    append(Piece1A, Piece1B, Piece1),
    append(Piece2A, Piece2B, Piece2),

    append(Piece1, Piece2, Result),

    statistics(walltime, _),
    labeling([], Result),
```

```
    statistics(walltime, [_, ElapsedTime | _]),
        format('A solution has been found!~nElapsed time: ~3d seconds', ElapsedTime), nl,
    fd_statistics, nl,

    !, displayBoard(NumColumns, NumRows, Piece1A, TypePiece1, Piece2A, TypePiece2).


% ---------- EACH TYPE OF PIECE HANDLER ----------

% Verify Piece can't attack Piece of the same type
pieceNoAttackSelf([A|Rest], NumColumns, Type) :-
    forEachPiece(A, Rest, NumColumns, Type),
    pieceNoAttackSelf(Rest, NumColumns, Type).
pieceNoAttackSelf([], _, _).

forEachPiece(A, [B|Rest], NumColumns, Type) :-
    verifyNoPieceAttack(A, B, NumColumns, Type),
    forEachPiece(A, Rest, NumColumns, Type).
forEachPiece(_, [], _, _).

verifyNoPieceAttack(A, B, NumColumns, Type) :-
    getMatrixPositionPLR(A, NumColumns, PieceRow, PieceColumn),
    getMatrixPositionPLR(B, NumColumns, OtherRow, OtherColumn),
    noAttackPositions(Type, PieceColumn, PieceRow, OtherColumn, OtherRow).

verifyNoPieceAttack(A, B, NumColumns, Type, C) :-
    getMatrixPositionPLR(A, NumColumns, PieceRow, PieceColumn),
    getMatrixPositionPLR(B, NumColumns, OtherRow, OtherColumn),
    getMatrixPositionPLR(C, NumColumns, MiddlePieceRow, MiddlePieceColumn),
    noAttackPositions(Type, PieceColumn, PieceRow, OtherColumn, OtherRow, MiddlePieceColumn,
        MiddlePieceRow).


% Verify if the Piece attack only one of the other type of Piece
pieceAttack(Piece, Other, NumColumns, Type) :-
    pieceAttackIterator(Piece, Other, NumColumns, [], Type).

pieceAttackIterator([A|Piece], [B|Other], NumColumns, Previous, Type) :-
    verifyPieceAttack(A, B, NumColumns, Type),
    pieceNoAttackOthers(A, Other, NumColumns, Type, B),
    pieceNoAttackOthers(A, Previous, NumColumns, Type, B),
    append([B], Previous, NewPrevious),
    pieceAttackIterator(Piece, Other, NumColumns, NewPrevious, Type).
pieceAttackIterator([], [], _, _, _).

pieceNoAttackOthers(_, [], _, _, _).
pieceNoAttackOthers(A, [C|Other], NumColumns, Type, B) :-
    verifyNoPieceAttack(A, C, NumColumns, Type, B),
    pieceNoAttackOthers(A, Other, NumColumns, Type, B).
```

```
verifyPieceAttack(A, B, NumColumns, Type) :-
    getMatrixPositionPLR(A, NumColumns, PieceRow, PieceColumn),
    getMatrixPositionPLR(B, NumColumns, OtherRow, OtherColumn),
    attackPositions(Type, PieceColumn, PieceRow, OtherColumn, OtherRow).



% ---------- ATTACK LOOP VERIFICATION ----------

checkForLoop(Piece1A, Piece2A, Piece1B, Piece2B, NumLoops) :-
    element(1, Piece1A, FirstPiece),
    loop(Piece1A, Piece2A, Piece1B, Piece2B, FirstPiece, FirstPiece, NumLoops).


loop(Piece1A, Piece2A, Piece1B, Piece2B, NextPiece, FirstPiece, 1) :-
    element(Pos1A, Piece1A, NextPiece),
    element(Pos1A, Piece2A, NextPiece2),
    element(Pos2B, Piece2B, NextPiece2),
    element(Pos2B, Piece1B, NextPiece3),

    NextPiece3 #= FirstPiece.

loop(Piece1A, Piece2A, Piece1B, Piece2B, NextPiece, FirstPiece, Counter) :-
    Counter #> 1,
    element(Pos1A, Piece1A, NextPiece),
    element(Pos1A, Piece2A, NextPiece2),
    element(Pos2B, Piece2B, NextPiece2),
    element(Pos2B, Piece1B, NextPiece3),

    NextPiece3 #\= FirstPiece,

    Counter1 is Counter-1,
    loop(Piece1A, Piece2A, Piece1B, Piece2B, NextPiece3, FirstPiece, Counter1).



% ---------- BOARD SOLVER PREDICATE TO SEE/VERIFY ALL SOLUTIONS ----------
solveBoard(NumRows, NumColumns, NumPieces, TypePiece1, TypePiece2, FinalPiece1, FinalPiece2) :-
    BoardSize is NumRows*NumColumns,
    length(Piece1A, NumPieces),
    length(Piece2A, NumPieces),
    length(Piece1B, NumPieces),
    length(Piece2B, NumPieces),

    domain(Piece1A, 1, BoardSize),
    domain(Piece1B, 1, BoardSize),
```

```
domain(Piece2A, 1, BoardSize),
domain(Piece2B, 1, BoardSize),

all_different(Piece1A),
all_different(Piece1B),
all_different(Piece2A),
all_different(Piece2B),

% Check if the lists have same elements
same_elements_list(Piece1A, Piece1B),
same_elements_list(Piece2A, Piece2B),

% Piece A attack constraints
pieceNoAttackSelf(Piece1A, NumColumns, TypePiece1),
pieceAttack(Piece1A, Piece2A, NumColumns, TypePiece1),

% Piece B attack constraints
pieceNoAttackSelf(Piece2B, NumColumns, TypePiece2),
pieceAttack(Piece2B, Piece1B, NumColumns, TypePiece2),

% Check if the positions form a loop
checkForLoop(Piece1A, Piece2A, Piece1B, Piece2B, NumPieces),

% Constraints to avoid some repeated solutions
% ------------------------------------------------------------
sorted(Piece1A),
different_lists(Piece1A, Piece1B, NumPieces),
% ------------------------------------------------------------

append(Piece1A, Piece1B, Piece1),
append(Piece2A, Piece2B, Piece2),

append(Piece1, Piece2, Result),

statistics(walltime, _),
labeling([], Result),
statistics(walltime, [_, ElapsedTime | _]),

sort(Piece1A, FinalPiece1),
sort(Piece2A, FinalPiece2),

nl, format('A solution has been found!~nElapsed time: ~3d seconds', ElapsedTime), nl,
fd_statistics, nl.
```

moves.pl

```
% ---------------------- PIECES MOVES HANDLERS ----------------------

% Attack

attackPositions(kings, PieceColumn, PieceRow, OtherColumn, OtherRow) :-
    attackPositionsKing(PieceColumn, PieceRow, OtherColumn, OtherRow).

attackPositions(knights, PieceColumn, PieceRow, OtherColumn, OtherRow) :-
    attackPositionsKnight(PieceColumn, PieceRow, OtherColumn, OtherRow).

attackPositions(rooks, PieceColumn, PieceRow, OtherColumn, OtherRow) :-
    attackPositionsRook(PieceColumn, PieceRow, OtherColumn, OtherRow).

attackPositions(bishops, PieceColumn, PieceRow, OtherColumn, OtherRow) :-
    attackPositionsBishop(PieceColumn, PieceRow, OtherColumn, OtherRow).

attackPositions(queens, PieceColumn, PieceRow, OtherColumn, OtherRow) :-
    attackPositionsQueen(PieceColumn, PieceRow, OtherColumn, OtherRow).


% No Attack

noAttackPositions(kings, PieceColumn, PieceRow, OtherColumn, OtherRow) :-
    noAttackPositionsKing(PieceColumn, PieceRow, OtherColumn, OtherRow).

noAttackPositions(knights, PieceColumn, PieceRow, OtherColumn, OtherRow) :-
    noAttackPositionsKnight(PieceColumn, PieceRow, OtherColumn, OtherRow).

noAttackPositions(rooks, PieceColumn, PieceRow, OtherColumn, OtherRow) :-
    noAttackPositionsRook(PieceColumn, PieceRow, OtherColumn, OtherRow).

noAttackPositions(bishops, PieceColumn, PieceRow, OtherColumn, OtherRow) :-
    noAttackPositionsBishop(PieceColumn, PieceRow, OtherColumn, OtherRow).

noAttackPositions(queens, PieceColumn, PieceRow, OtherColumn, OtherRow) :-
    noAttackPositionsQueen(PieceColumn, PieceRow, OtherColumn, OtherRow).


noAttackPositions(kings, PieceColumn, PieceRow, OtherColumn, OtherRow, MiddlePieceColumn,
    MiddlePieceRow) :-
    noAttackPositionsKing(PieceColumn, PieceRow, OtherColumn, OtherRow, MiddlePieceColumn,
        MiddlePieceRow).

noAttackPositions(knights, PieceColumn, PieceRow, OtherColumn, OtherRow, MiddlePieceColumn,
    MiddlePieceRow) :-
    noAttackPositionsKnight(PieceColumn, PieceRow, OtherColumn, OtherRow, MiddlePieceColumn,
        MiddlePieceRow).

noAttackPositions(rooks, PieceColumn, PieceRow, OtherColumn, OtherRow, MiddlePieceColumn,
    MiddlePieceRow) :-
```

```
    noAttackPositionsRook(PieceColumn, PieceRow, OtherColumn, OtherRow, MiddlePieceColumn,
        MiddlePieceRow).

noAttackPositions(bishops, PieceColumn, PieceRow, OtherColumn, OtherRow, MiddlePieceColumn,
    MiddlePieceRow) :-
    noAttackPositionsBishop(PieceColumn, PieceRow, OtherColumn, OtherRow, MiddlePieceColumn,
        MiddlePieceRow).

noAttackPositions(queens, PieceColumn, PieceRow, OtherColumn, OtherRow, MiddlePieceColumn,
    MiddlePieceRow) :-
    noAttackPositionsQueen(PieceColumn, PieceRow, OtherColumn, OtherRow, MiddlePieceColumn,
        MiddlePieceRow).


% ---------------------- KING MOVES ----------------------

attackPositionsKing(KingColumn, KingRow, OtherColumn, OtherRow) :-
    (OtherColumn #= KingColumn #/\ OtherRow #= KingRow+1) #\
    (OtherColumn #= KingColumn #/\ OtherRow #= KingRow-1) #\
    (OtherColumn #= KingColumn+1 #/\ OtherRow #= KingRow+1) #\
    (OtherColumn #= KingColumn+1 #/\ OtherRow #= KingRow-1) #\
    (OtherColumn #= KingColumn-1 #/\ OtherRow #= KingRow+1) #\
    (OtherColumn #= KingColumn-1 #/\ OtherRow #= KingRow-1) #\
    (OtherColumn #= KingColumn+1 #/\ OtherRow #= KingRow) #\
    (OtherColumn #= KingColumn-1 #/\ OtherRow #= KingRow).

noAttackPositionsKing(KingColumn, KingRow, OtherColumn, OtherRow) :-
    (OtherColumn #\= KingColumn #\/ OtherRow #\= KingRow+1) #/\
    (OtherColumn #\= KingColumn #\/ OtherRow #\= KingRow-1) #/\
    (OtherColumn #\= KingColumn+1 #\/ OtherRow #\= KingRow+1) #/\
    (OtherColumn #\= KingColumn+1 #\/ OtherRow #\= KingRow-1) #/\
    (OtherColumn #\= KingColumn-1 #\/ OtherRow #\= KingRow+1) #/\
    (OtherColumn #\= KingColumn-1 #\/ OtherRow #\= KingRow-1) #/\
    (OtherColumn #\= KingColumn+1 #\/ OtherRow #\= KingRow) #/\
    (OtherColumn #\= KingColumn-1 #\/ OtherRow #\= KingRow).

noAttackPositionsKing(KingColumn, KingRow, OtherColumn, OtherRow, _, _) :-
    (OtherColumn #\= KingColumn #\/ OtherRow #\= KingRow+1) #/\
    (OtherColumn #\= KingColumn #\/ OtherRow #\= KingRow-1) #/\
    (OtherColumn #\= KingColumn+1 #\/ OtherRow #\= KingRow+1) #/\
    (OtherColumn #\= KingColumn+1 #\/ OtherRow #\= KingRow-1) #/\
    (OtherColumn #\= KingColumn-1 #\/ OtherRow #\= KingRow+1) #/\
    (OtherColumn #\= KingColumn-1 #\/ OtherRow #\= KingRow-1) #/\
    (OtherColumn #\= KingColumn+1 #\/ OtherRow #\= KingRow) #/\
    (OtherColumn #\= KingColumn-1 #\/ OtherRow #\= KingRow).


% ---------------------- KNIGHT MOVES ----------------------
```

```
attackPositionsKnight(KnightColumn, KnightRow, OtherColumn, OtherRow) :-
    (OtherColumn #= KnightColumn+2 #/\ OtherRow #= KnightRow+1) #\
    (OtherColumn #= KnightColumn+2 #/\ OtherRow #= KnightRow-1) #\
    (OtherColumn #= KnightColumn-2 #/\ OtherRow #= KnightRow+1) #\
    (OtherColumn #= KnightColumn-2 #/\ OtherRow #= KnightRow-1) #\
    (OtherRow #= KnightRow+2 #/\ OtherColumn #= KnightColumn+1) #\
    (OtherRow #= KnightRow+2 #/\ OtherColumn #= KnightColumn-1) #\
    (OtherRow #= KnightRow-2 #/\ OtherColumn #= KnightColumn+1) #\
    (OtherRow #= KnightRow-2 #/\ OtherColumn #= KnightColumn-1).

noAttackPositionsKnight(KnightColumn, KnightRow, OtherColumn, OtherRow) :-
    (OtherColumn #\= KnightColumn+2 #\/ OtherRow #\= KnightRow+1) #/\
    (OtherColumn #\= KnightColumn+2 #\/ OtherRow #\= KnightRow-1) #/\
    (OtherColumn #\= KnightColumn-2 #\/ OtherRow #\= KnightRow+1) #/\
    (OtherColumn #\= KnightColumn-2 #\/ OtherRow #\= KnightRow-1) #/\
    (OtherRow #\= KnightRow+2 #\/ OtherColumn #\= KnightColumn+1) #/\
    (OtherRow #\= KnightRow+2 #\/ OtherColumn #\= KnightColumn-1) #/\
    (OtherRow #\= KnightRow-2 #\/ OtherColumn #\= KnightColumn+1) #/\
    (OtherRow #\= KnightRow-2 #\/ OtherColumn #\= KnightColumn-1).

noAttackPositionsKnight(KnightColumn, KnightRow, OtherColumn, OtherRow, _, _) :-
    (OtherColumn #\= KnightColumn+2 #\/ OtherRow #\= KnightRow+1) #/\
    (OtherColumn #\= KnightColumn+2 #\/ OtherRow #\= KnightRow-1) #/\
    (OtherColumn #\= KnightColumn-2 #\/ OtherRow #\= KnightRow+1) #/\
    (OtherColumn #\= KnightColumn-2 #\/ OtherRow #\= KnightRow-1) #/\
    (OtherRow #\= KnightRow+2 #\/ OtherColumn #\= KnightColumn+1) #/\
    (OtherRow #\= KnightRow+2 #\/ OtherColumn #\= KnightColumn-1) #/\
    (OtherRow #\= KnightRow-2 #\/ OtherColumn #\= KnightColumn+1) #/\
    (OtherRow #\= KnightRow-2 #\/ OtherColumn #\= KnightColumn-1).


% ---------- ROOK MOVES ----------

attackPositionsRook(RookColumn, RookRow, OtherColumn, OtherRow) :-
    (OtherColumn #= RookColumn #\ OtherRow #= RookRow).

noAttackPositionsRook(RookColumn, RookRow, OtherColumn, OtherRow) :-
    (OtherColumn #\= RookColumn #/\ OtherRow #\= RookRow).

noAttackPositionsRook(RookColumn, RookRow, OtherColumn, OtherRow, MiddlePieceColumn,
    MiddlePieceRow) :-
    (OtherColumn #\= RookColumn #/\ OtherRow #\= RookRow) #\
    (OtherColumn #= RookColumn #/\ OtherColumn #= MiddlePieceColumn #/\ OtherRow #< MiddlePieceRow
        #/\ MiddlePieceRow #< RookRow) #\
    (OtherColumn #= RookColumn #/\ OtherColumn #= MiddlePieceColumn #/\ OtherRow #> MiddlePieceRow
        #/\ MiddlePieceRow #> RookRow) #\
    (OtherRow #= RookRow #/\ OtherRow #= MiddlePieceRow #/\ OtherColumn #< MiddlePieceColumn #/\
        MiddlePieceColumn #< RookColumn) #\
```

```
    (OtherRow #= RookRow #/\ OtherRow #= MiddlePieceRow #/\ OtherColumn #> MiddlePieceColumn #/\
        MiddlePieceColumn #> RookColumn).


% ------------------- BISHOP MOVES --------------------

attackPositionsBishop(BishopColumn, BishopRow, OtherColumn, OtherRow) :-
    abs(BishopColumn - OtherColumn) #= abs(BishopRow - OtherRow).

noAttackPositionsBishop(BishopColumn, BishopRow, OtherColumn, OtherRow) :-
    abs(BishopColumn - OtherColumn) #\= abs(BishopRow - OtherRow).

noAttackPositionsBishop(BishopColumn, BishopRow, OtherColumn, OtherRow, MiddlePieceColumn,
    MiddlePieceRow) :-
    (abs(BishopColumn - OtherColumn) #\= abs(BishopRow - OtherRow)) #\
    (abs(BishopColumn - OtherColumn) #= abs(BishopRow - OtherRow) #/\ abs(BishopColumn -
        MiddlePieceColumn) #= abs(BishopRow - MiddlePieceRow) #/\ MiddlePieceColumn #>
        BishopColumn #/\ OtherColumn #> MiddlePieceColumn) #\
    (abs(BishopColumn - OtherColumn) #= abs(BishopRow - OtherRow) #/\ abs(BishopColumn -
        MiddlePieceColumn) #= abs(BishopRow - MiddlePieceRow) #/\ MiddlePieceColumn #<
        BishopColumn #/\ OtherColumn #< MiddlePieceColumn).


% ------------------- QUEEN MOVES --------------------

attackPositionsQueen(QueenColumn, QueenRow, OtherColumn, OtherRow) :-
    (OtherColumn #= QueenColumn #/\ OtherRow #\= QueenRow) #\
    (OtherColumn #\= QueenColumn #/\ OtherRow #= QueenRow) #\
    (abs(QueenColumn - OtherColumn) #= abs(QueenRow - OtherRow)).

noAttackPositionsQueen(QueenColumn, QueenRow, OtherColumn, OtherRow) :-
    (OtherColumn #\= QueenColumn #\/ OtherRow #= QueenRow) #/\
    (OtherColumn #= QueenColumn #\/ OtherRow #\= QueenRow) #/\
    (abs(QueenColumn - OtherColumn) #\= abs(QueenRow - OtherRow)).

noAttackPositionsQueen(QueenColumn, QueenRow, OtherColumn, OtherRow, MiddlePieceColumn,
    MiddlePieceRow) :-
    ((OtherColumn #\= QueenColumn #/\ OtherRow #\= QueenRow) #\
    (OtherColumn #= QueenColumn #/\ OtherColumn #= MiddlePieceColumn #/\ OtherRow #<
        MiddlePieceRow #/\ MiddlePieceRow #< QueenRow) #\
    (OtherColumn #= QueenColumn #/\ OtherColumn #= MiddlePieceColumn #/\ OtherRow #>
        MiddlePieceRow #/\ MiddlePieceRow #> QueenRow) #\
    (OtherRow #= QueenRow #/\ OtherRow #= MiddlePieceRow #/\ OtherColumn #< MiddlePieceColumn #/\
        MiddlePieceColumn #< QueenColumn) #\
    (OtherRow #= QueenRow #/\ OtherRow #= MiddlePieceRow #/\ OtherColumn #> MiddlePieceColumn #/\
        MiddlePieceColumn #> QueenColumn)) #/\
    ((abs(QueenColumn - OtherColumn) #\= abs(QueenRow - OtherRow)) #\
```

```
    (abs(QueenColumn - OtherColumn) #= abs(QueenRow - OtherRow) #/\ abs(QueenColumn -
        MiddlePieceColumn) #= abs(QueenRow - MiddlePieceRow) #/\ MiddlePieceColumn #> QueenColumn
        #/\ OtherColumn #> MiddlePieceColumn) #\
    (abs(QueenColumn - OtherColumn) #= abs(QueenRow - OtherRow) #/\ abs(QueenColumn -
        MiddlePieceColumn) #= abs(QueenRow - MiddlePieceRow) #/\ MiddlePieceColumn #< QueenColumn
        #/\ OtherColumn #< MiddlePieceColumn)).
```

board.pl

```
% ---------- GET BOARD POSITION WITH CONSTRAINT LOGIC PROGRAMMING ----------

getMatrixPositionPLR(Position, NumColumns, PosRow, PosColumn) :-
    getColumnPLR(Position, NumColumns, PosColumn),
    getRowPLR(Position, NumColumns, PosRow).

getColumnPLR(Position, NumColumns, PosColumn) :-
    ((Position mod NumColumns) #= 0 #/\ PosColumn #= NumColumns) #\/
    ((Position mod NumColumns) #\= 0 #/\ PosColumn #= (Position mod NumColumns)).

getRowPLR(Position, NumColumns, PosRow) :-
    ((Position mod NumColumns) #= 0 #/\ PosRow #= Position/NumColumns) #\
    ((Position mod NumColumns) #\= 0 #/\ PosRow #= (Position/NumColumns +1)).


% ---------- GET BOARD POSITION ----------

getMatrixPosition(Position, NumColumns, PosRow, PosColumn) :-
    getColumn(Position, NumColumns, PosColumn),
    getRow(Position, NumColumns, PosRow).

getColumn(Position, NumColumns, PosColumn) :-
    (Position mod NumColumns) =:= 0, !,
    PosColumn is NumColumns;
    PosColumn is (Position mod NumColumns).

getRow(Position, NumColumns, PosRow) :-
    floor(Position/NumColumns) =:= ceiling(Position/NumColumns), !,
    PosRow is round(Position/NumColumns);
    PosRow is floor(Position/NumColumns)+1.


% ---------- BOARD DISPLAY ----------

symbol(empty, S) :- S=' '.
symbol(kings, S) :- S='\x2654\'.
symbol(knights, S) :- S='\x2658\'.
```

```
symbol(queens, S) :- S='\x2655\'.
symbol(bishops, S) :- S='\x2657\'.
symbol(rooks, S) :- S='\x2656\'.

replaceBoard([],_,NewBoard,NewBoard).
replaceBoard([H|T], Type, Board, NewBoard) :-
    replace(Board, H, Type, NewBoard1),
    replaceBoard(T, Type, NewBoard1, NewBoard).

generateEmptyBoard(N, Board) :-
    generateBoard(N, [], Board).

generateBoard(0, Board, Board).
generateBoard(N, BoardIterator, Board) :-
    N > 0,
    append([empty], BoardIterator, NewBoard),
    N1 is N-1,
    generateBoard(N1, NewBoard, Board).

writeBoard([], _).
writeBoard([H|T], NumColumns, Counter) :-
    X is floor(Counter/NumColumns),
    Y is ceiling(Counter/NumColumns),
    X == Y, !,
    printSquare(H), write(' |'), nl,
    Counter1 is Counter+1,
    writeBoard(T, NumColumns, Counter1);
    printSquare(H),
    Counter1 is Counter+1,
    writeBoard(T, NumColumns, Counter1).

printSquare(Piece) :-
    symbol(Piece, PieceSymbol),
    write(' |'),
    write(PieceSymbol).

displayBoard(NumColumns, NumRows, Pieces1, TypePiece1, Pieces2, TypePiece2) :-
    Size is NumColumns*NumRows,
    generateEmptyBoard(Size, Board),
    replaceBoard(Pieces1, TypePiece1, Board, NewBoard),
    replaceBoard(Pieces2, TypePiece2, NewBoard, NewBoard1), !,
    writeBoard(NewBoard1, NumColumns, 1).
```

utils.pl

```
% checks if two lists with the same size have the same elements
```

```
same_elements_list([], _).
same_elements_list([A|Rest], ListB) :-
    element(_, ListB, A),
    same_elements_list(Rest, ListB).


% checks if two lists with the same size have the same elements in a different order
different_lists(ListA, ListB, ListLength) :-
    different_lists_iterator(ListA, ListB, Counter),
    MaxValue #= ListLength-1,
    count(1, Counter, #<, MaxValue).

different_lists_iterator([A|ListA], [B|ListB], [X|Xs]) :-
    (A #= B) #<=> X,
    different_lists_iterator(ListA, ListB, Xs).
different_lists_iterator([], [], []).


% Makes sure if a List is in ascending order
sorted([A,B|R]) :-
    A #< B,
    sorted([B|R]).
sorted([_]).

% Replaces the element in index I on a List by the element X
replace([_|T], 1, X, [X|T]).
replace([H|T], I, X, [H|R]):- I > 0, NI is I-1, replace(T, NI, X, R), !.
replace(L, _, _, L).
```