Resolução de Problema de Decisão usando Programação em Lógica com Restrições: Puzzle Loop

Bruno Sousa – up20104145 e João Gonçalves – up201604245

Faculdade de Engenharia da Universidade do Porto

Resumo: Este artigo serve de complemento ao segundo trabalho realizado para a cadeira Programação Lógica que teve como objetivo a Resolução de um Problema de Decisão/Otimização usando Programação em Lógica com Restrições, sendo este problema o "Puzzle Loop".

Keywords: Programação Lógica, Puzzle Loop, SICStus.

1 Introdução

Na escolha de um problema de Decisão usando Programação em Logica com Restrições o nosso grupo optou pelo Puzzle Loop.

O Puzzle Loop consiste na distribuição, num tabuleiro, das peças de xadrez, de forma a que cada peça ataque outra , sendo que esta peça também não poderá atacar nem uma peça igual a si nem mais do que uma peça ao mesmo tempo. O movimento para cada peça vai ser o mesmo que está estipulado para o jogo xadrez.

Este artigo vai apresentar a seguinte estrutura:

- **2-Descrição do Problema:** Descrição com detalhe o problema de otimização ou decisão em análise.
- **3-Abordagem:** Descrição da modelação do problema como um PSR, de acordo com as subsecções seguintes:
    - **3.1-Variáveis de Decisão:** Descrição das variáveis de decisão e os seus domínios.
    - **3.2-Restrições:** Descrição das restrições rígidas e flexíveis do problema e a sua implementação utilizando o SICStus Prolog.
    - **3.3-Estratégia de Pesquisa:** Descrição da estratégia de etiquetagem (*labeling*) utilizada ou implementada, nomeadamente no que diz respeito à ordenação de variáveis e valores.

- **4-Visualização da Solução:** Explicação dos predicados que permitem visualizar a solução em modo de texto.

- **5-Resultados:** Exemplos de aplicação em instâncias do problema com diferentes dimensões e análise dos resultados obtidos.

- **6-Conclusões:** Conclusões retiradas deste projeto, resultados obtidos, vantagens e limitações da solução proposta, aspetos a melhorar.

- **7-Bibliografia:** Bibliografia consultada durante o desenvolvimento do trabalho.

- **8-Anexos:** Código fonte para uma melhor compreensão do artigo.

2 Descrição do Problema

O Puzzle Loop é um puzzle 2D que lida com tamanhos diferentes de tabuleiros e números diferentes de peças.

Para a resolução deste, primeiro vai ser escolhido ou gerado aleatoriamente o problema e só depois, com o uso de restrições, este vai ser resolvido.

Relativamente ao puzzle em si, as peças que estão disponíveis para este problema são o cavalo, o rei, o bispo, a torre e a rainha.

Tal como referido anteriormente, o objetivo deste problema é reorganizar as peças de forma a que cada peça ataque uma, e apenas uma, peça diferente da própria. Cada peça também terá de estar a ser atacada por outra peça diferente formando assim um loop, como se pode ver pela figura 1. Para além disso, cada peça não poderá atacar uma peça do mesmo tipo.

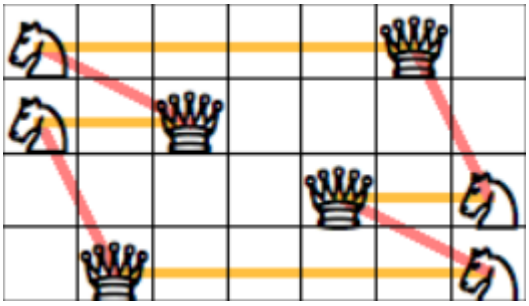

Figura 1- Resolução para um tabuleiro 4x7 com 4 cavalos e 4 rainhas.

3 Abordagem

Para a resolução deste problema é utilizada uma lista que guarda todas as posições do tabuleiro. O tamanho desse tabuleiro, bem como o número e tipo de peças a colocar nele são escolhidas ou geradas aleatoriamente.

Após a atribuição das restrições, a matriz vai conter 0's para os índices em que não se apresenta nenhuma peça, e o id da peça nos índices em que se encontra a peça correspondente a esse id.

O id do cavalo (n) é 1, o id do bispo (b) é 2, o id do rei (k) é 3, o id da torre (r) é 4 e, por fim, o id da rainha (q) é 5.

Para o exemplo encontrado na figura 1,
Lista = [1, 0, 0, 0, 0, 5, 0, 1, 0, 5, 0, 0, 0, 0, 0, 0, 0, 0, 5, 0, 1, 0, 5, 0, 0, 0, 0, 1].

3.1 Variáveis de Decisão

A solução tem, ao todo, N variáveis de decisão totais, sendo N o número de espaços no tabuleiro, ou seja, o tamanho da matriz.

Para o domínio da matriz de solução este vai estar desde 0 até ao maior id das duas peças escolhidas, visto que os índices da matriz só vão assumir 3 valores diferentes, sendo estes os id's para cada peça/espaço vazio.

No final, na geração de soluções (em labelling), como a solução é única (apenas se repete simetricamente) apenas é obtida uma solução.

3.2 Restrições

Podemos organizar as diferentes restrições para este prolema em 5 restrições:

- **Apenas podes existir N peças de cada tipo**: Para este caso, é restringido para cada tipo que a contagem desse tipo de peça no tabuleiro tem de ser igual a N (utilizando o predicado **count**).

- **Cada peça não pode atacar outra igual à própria:** Para esta restrição são calculadas, com o movimento específico de cada peça, as posições que a peça está a atacar e restringe-se que não podem estar peças com id igual ao da peça em causa, nas posições que esta está a atacar.

- **Cada peça pode apenas estar a atacar uma peça:** Para este caso, novamente são calculadas as posições que a peça está a atacar e restringe-se que uma, e apenas uma, dessas posições pode conter uma peça, sendo esta peça obrigatoriamente uma com id diferente da peça que a está a atacar.

- **Cada peça tem de estar a ser atacada por apenas uma peça:** Para este caso, são calculadas todas as posições de onde a peça de outro tipo a poderia estar a atacar e restringe-se que uma, e apenas uma, dessas posições pode conter uma peça do outro tipo.

- **Os ataques das peças têm de formar um *loop*:** Para este caso, para cada posição que não estiver em branco, é guardada a posição da peça que esta ataca num tabuleiro auxiliar e o seu índice numa lista auxiliar**.** Após terem sido aplicadas todas as outras restrições é criada uma lista, utilizando o tabuleiro e a lista auxiliares, uma nova lista em que para cada índex, o seu valor será o índex da lista que ele estará a atacar. Para essa lista é restringido que tem de formar um circuito hamiltoniano (utilizando o predicado **circuit**).

- Nos casos em que a peça pode-se deslocar multiplicas casas na mesma direção, esta não está a atacar peças em que pelo caminho até estas, se encontra outra peça.

3.3 Estratégia de Pesquisa

Para a decisão da estratégia de pesquisa, foi testado o problema mais complexo que encontramos (“Place 4 knights and queens on a 4x7 chess board.”) com todas as estratégias e verificada qual a melhor (a mais rápida / com menos *backtracks*).

Resultados das estratégias de seleção de variáveis:

- **leftmost*:*** 4,97 segundos com 5766 *backtracks*.
- **ff:** 5,28 segundos com 5766 *backtracks*.
- **ffc:** 2,56 segundos com 3152 *backtracks*.
- **min:** mais de um minuto (paramos o programa pois já não seria a melhor estratégia).
- **max:** 4,99 segundos com 18109 *backtracks*.

Resultados das estratégias de ordenação de valores:

- **up:** 4,75 segundos com 5766 *backtracks*.
- **down:** 0,51 segundos com 1506 *backtracks*.

Resultados das estratégias de *branching*:

- **step:** 4,75 segundos com 5766 *backtracks*.
- **enum:** 5,11 segundos com 24460 *backtracks*.
- **bisect:** 4,77 segundos com 9557 *backtracks*.

Depois da análise dos resultados foram escolhidas as estratégias **ffc**, **down** e **step**.

4 Visualização da Solução

Para visualizer a solução foi implementado o predicado **display_board(+List,+Line,+Cols)**, que recebendo uma lista com o tabuleiro com a respresentação das peças referida acima, a linha do tabuleiro em que se pretende começar a represetanção e o numero de colunas do tabuleiro, mostra o tabuleiro.

Exemplo para o problema "Place 4 knights and queens on a 4x7 chess board.":

Figura 2- Visualização de um tabuleiro 4x7 com 4 cavalos e 4 rainhas.

A sua implementação em Prolog:

```
display_board([],_,_):-
  nl,
  nl,
  !.

display_board(Matrix,Cols,Cols):-
  !,
  nl,
  display_nl(0,Cols),
  nl,
  display_board(Matrix,0,Cols).

display_board([Head|Tail],Col,Cols):-
  pieceCode(Piece,Head),
  write(Piece),
  write('|'),
  Next is Col + 1,
  display_board(Tail,Next,Cols).
```

Para cada posição chama a função **pieceCode(-Piece,+Code)**, para obter a representação de cada peça a partir do seu id.

5 Resultados

Exemplo de solução de problemas diferentes:

```
| ?- chessloop(1).
Place 2 knights and kings on a 2x3 chess board.

k| |k|
------
n| |n|

yes
```

```
| ?- chessloop(7).
Place 4 bishops and knights on a 3x5 chess board.

b| |b| | |
----------
 | |n|n|b|
----------
 |b|n|n| |

yes
| ?- chessloop(13).
Place 3 knights and queens on a 3x6 chess board.

q| | |n| | |
------------
 | |n|n| |q|
------------
 |q| | | | |

yes
```

Estátisticas para cada um dos problemas encontrados:

Problema	Tempo (em segundos)	Backtracks
"Place 2 bishops and rooks on a 2x3 chess board."	0.0	2
"Place 2 knights and kings on a 2×3 chess board."	0.0	2
"Place 3 knights and kings on a 4×5 chess board."	0.14	492
"Place 2 rooks and kings on a 2×4 chess board."	0.0	19

“Place 3 rooks and kings on a 4×5 chess board.”	0.14	307
“Place 2 bishops and knights on a 3×3 chess board.”	0.0	2
“Place 4 bishops and knights on a 4×4 chess board.”	0.07	466
“Place 3 bishops and knights on a 3×5 chess board.”	0.0	79
“Place 4 bishops and kings on a 4×6 chess board.”	1.13	2782
“Place 4 bishops and kings on a 5×5 chess board.”	0.13	250
“Place 3 knights and rooks on a 3×4 chess board.”	0.0	37
“Place 5 knights and rooks on a 3×8 chess board.”	0.12	468
“Place 2 knights and queens on a 2×4 chess board.”	0.0	4
“Place 3 knights and queens on a 3×6 chess board.”	0.03	148
“Place 3 knights and queens on a 4×5 chess board.”	0.02	86
“Place 4 knights and queens on a 4×7 chess board.”	0.54	1506

6 Conclusões

Este projeto teve como objetivo a resolução de Problema de Decisão usando Programação em Lógica com Restrições utilizando, para tal, os conhecimentos adquiridos na unidade curricular de Programação Lógica.

Neste trabalho fomos superando as dificuldades encontradas., nomeadamente nas restrições.

Em suma, estamos satisfeitos com o resultado deste trabalho e consideramos que este projeto contribuiu bastante para uma melhor compreensão de restrições em Prolog.

7 Bibliografia

Chess Loop Puzzles
lib-clpfd - SICStus Prolog

8 Anexos

Código Fonte:

Ficheiro *chessloop.pl*:

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

:- reconsult('puzzles.pl').
:- reconsult('display.pl').
:- reconsult('pieces.pl').
:- reconsult('king.pl').
:- reconsult('knight.pl').
:- reconsult('rook.pl').
:- reconsult('bishop.pl').
:- reconsult('queen.pl').

buildPositionsChildren(_,_,[],Indexes,Indexes,Posi-
tions,Positions):-
  !.

%Gets valid positions value in a direction
buildPositionsChildren(Matrix,Cols,[Head|Tail],Temp-
Ind,Indexes,Temp,Positions):-
  [Line,Col] = Head,
  Line > 0,
  length(Matrix,Size),
  Line =< Size div Cols,
  Col > 0,
  Col =< Cols,
  !,
  Index is ((Line - 1) * Cols) + Col,
  element(Index,Matrix,Element),
```

```
  append(TempInd,[Index],NextInd),
  append(Temp,[Element],Next),
  buildPositionsChildren(Matrix,Cols,Tail,NextInd,In-
dexes,Next,Positions).

buildPositionsChildren(Matrix,Cols,[_|Tail],TempInd,In-
dexes,Temp,Positions):-
  buildPositionsChildren(Matrix,Cols,Tail,TempInd,In-
dexes,Temp,Positions).

buildPositions(_,_,[],Indexes,Indexes,Positions,Posi-
tions):-
  !.

%Gets valid positions value in all directions
buildPositions(Matrix,Cols,[Head|Tail],TempInd,In-
dexes,Temp,Positions):-
  Head \= [[]],
  buildPositionsChildren(Matrix,Cols,Head,[],Child-
Indexes,[],ChildPositions),
  ChildIndexes \= [],
  !,
  append(TempInd,ChildIndexes,NextInd),
  append(Temp,[ChildPositions],Next),
  buildPositions(Matrix,Cols,Tail,NextInd,In-
dexes,Next,Positions).

buildPositions(Matrix,Cols,[_|Tail],TempInd,In-
dexes,Temp,Positions):-
  buildPositions(Matrix,Cols,Tail,TempInd,In-
dexes,Temp,Positions).

subCountCode(_,_,[],CodeCount):-
  !,
  CodeCount #= 0.

%Counts if code apears (1) or not (0) in a list
subCountCode(CountCode,OtherCode,[Head|Tail],CodeCount):-
  domain([CodeCount],0,1),
  subCountCode(CountCode,OtherCode,Tail,Temp),
  (Head #= CountCode #/\ CodeCount #= 1)
  #\/
  (Head #= 0 #/\ CodeCount #= Temp)
  #\/
  (Head #= OtherCode #/\ CodeCount #= 0).
```

```
countCode(_,_,[],_,CodeCount):-
  !,
  CodeCount #= 0.

%Counts in how many directions CountCode appears
countCode(CountCode,OtherCode,List,PositionCode,Co-
deCount):-
  [Head|Tail] = List,
  length(List,Max),
  domain([CodeCount],0,Max),
  subCountCode(CountCode,OtherCode,Head,Temp),
  countCode(CountCode,OtherCode,Tail,PositionCode,Rest),
  CodeCount #= Temp + Rest.

matrixtoList([],List,List):-
  !.

%Builds a list from a matrix
matrixtoList([Head|Tail],Temp,List):-
  append(Temp,Head,Next),
  matrixtoList(Tail,Next,List).

restrict(_,_,_,Line,Lines,_,_,_,_,_):-
  Line > Lines,
  !.

restrict(Matrix,Code1,Code2,Line,Lines,Col,Cols,Loop-
Matrix,Loop,LoopIndex):-
  Col > Cols,
  !,
  NextLine is Line + 1,
  restrict(Matrix,Code1,Code2,NextLine,Lines,1,Cols,Loop-
Matrix,Loop,LoopIndex).

%Applies problem restrictions to all board positions.
restrict(Matrix,Code1,Code2,Line,Lines,Col,Cols,Loop-
Matrix,Loop,LoopIndex):-
  Index is ((Line - 1) * Cols) + Col,
  element(Index,Matrix,Element),
  attackPositions(Matrix,Code1,Line,Col,Cols,Indexes1,At-
tackPositions1),
  attackPositions(Matrix,Code2,Line,Col,Cols,Indexes2,At-
tackPositions2),
  countCode(Code2,Code1,AttackPositions1,Code1,Attack1),
```

```
  matrixtoList(AttackPositions1,[],ListAttackPositions1),
  element(LoopIndex1,ListAttackPositions1,LoopCode2),
  element(LoopIndex1,Indexes1,Index1),
  countCode(Code1,Code2,AttackPositions2,Code2,Attack2),
  matrixtoList(AttackPositions2,[],ListAttackPositions2),
  element(LoopIndex2,ListAttackPositions2,LoopCode1),
  element(LoopIndex2,Indexes2,Index2),
  countCode(Code1,Code2,AttackPositions1,Code1,Defend1),
  countCode(Code2,Code1,AttackPositions2,Code2,Defend2),
  !,
  element(Index,LoopMatrix,LoopMatrixElement),
  element(LoopIndex,Loop,LoopElement),
    (Element #= 0 #/\ NewLoopIndex #= LoopIndex)
    #\/
    (Element #= Code1 #/\ LoopCode2 #= Code2 #/\ Loop-
MatrixElement #= LoopIndex #/\ LoopElement #= Index1 #/\
NewLoopIndex #= LoopIndex + 1 #/\ Attack1 #= 1 #/\ De-
fend1 #= 0 #/\ Defend2 #= 1)
    #\/
    (Element #= Code2 #/\ LoopCode1 #= Code1 #/\ Loop-
MatrixElement #= LoopIndex #/\ LoopElement #= Index2 #/\
NewLoopIndex #= LoopIndex + 1 #/\ Attack2 #= 1 #/\ De-
fend2 #= 0 #/\ Defend1 #= 1),
  NextCol is Col + 1,
  restrict(Ma-
trix,Code1,Code2,Line,Lines,NextCol,Cols,Loop-
Matrix,Loop,NewLoopIndex).

restrict(Matrix,Code1,Code2,Line,Lines,Col,Cols,Loop-
Matrix,Loop,LoopIndex):-
  Index is ((Line - 1) * Cols) + Col,
  element(Index,Matrix,Element),
  attackPositions(Matrix,Code1,Line,Col,Cols,_,Attack-
Positions1),
  attackPositions(Matrix,Code2,Line,Col,Cols,Indexes2,At-
tackPositions2),
  countCode(Code1,Code2,AttackPositions2,Code2,Attack2),
  matrixtoList(AttackPositions2,[],ListAttackPositions2),
  element(LoopIndex2,ListAttackPositions2,LoopCode1),
  element(LoopIndex2,Indexes2,Index2),
  countCode(Code1,Code2,AttackPositions1,Code1,Defend1),
  countCode(Code2,Code1,AttackPositions2,Code2,Defend2),
  !,
  element(Index,LoopMatrix,LoopMatrixElement),
  element(LoopIndex,Loop,LoopElement),
```

```
    (Element #= 0 #/\ NewLoopIndex #= LoopIndex)
    #\/
    (Element #= Code2 #/\ LoopCode1 #= Code1 #/\ Loop-
MatrixElement #= LoopIndex #/\ LoopElement #= Index2 #/\
NewLoopIndex #= LoopIndex + 1 #/\ Attack2 #= 1 #/\ De-
fend2 #= 0 #/\ Defend1 #= 1),
  NextCol is Col + 1,
  restrict(Ma-
trix,Code1,Code2,Line,Lines,NextCol,Cols,Loop-
Matrix,Loop,NewLoopIndex).

restrict(Matrix,Code1,Code2,Line,Lines,Col,Cols,Loop-
Matrix,Loop,LoopIndex):-
  Index is ((Line - 1) * Cols) + Col,
  element(Index,Matrix,Element),
  attackPositions(Matrix,Code1,Line,Col,Cols,Indexes1,At-
tackPositions1),
  attackPositions(Matrix,Code2,Line,Col,Cols,_,Attack-
Positions2),
  countCode(Code2,Code1,AttackPositions1,Code1,Attack1),
  matrixtoList(AttackPositions1,[],ListAttackPositions1),
  element(LoopIndex1,ListAttackPositions1,LoopCode2),
  element(LoopIndex1,Indexes1,Index1),
  countCode(Code1,Code2,AttackPositions1,Code1,Defend1),
  countCode(Code2,Code1,AttackPositions2,Code2,Defend2),
  !,
  element(Index,LoopMatrix,LoopMatrixElement),
  element(LoopIndex,Loop,LoopElement),
    (Element #= 0 #/\ NewLoopIndex #= LoopIndex)
    #\/
    (Element #= Code1 #/\ LoopCode2 #= Code2 #/\ Loop-
MatrixElement #= LoopIndex #/\ LoopElement #= Index1 #/\
NewLoopIndex #= LoopIndex + 1 #/\ Attack1 #= 1 #/\ De-
fend1 #= 0 #/\ Defend2 #= 1),
  NextCol is Col + 1,
  restrict(Ma-
trix,Code1,Code2,Line,Lines,NextCol,Cols,Loop-
Matrix,Loop,NewLoopIndex).

restrict(Matrix,Code1,Code2,Line,Lines,Col,Cols,Loop-
Matrix,Loop,LoopIndex):-
  Index is ((Line - 1) * Cols) + Col,
  element(Index,Matrix,Element),
  Element #= 0,
  NextCol is Col + 1,
```

```
  NewLoopIndex #= LoopIndex,
  restrict(Ma-
trix,Code1,Code2,Line,Lines,NextCol,Cols,Loop-
Matrix,Loop,NewLoopIndex).

buildCircuit(_,_,_,Index,Size):-
  Index > Size,
  !.

%Builds the circuit needed to ensure "loop"
buildCircuit(Circuit,LoopMatrix,Loop,Index,LoopSize):-
  element(Index,Circuit,Element),
  element(Index,Loop,LoopElement),
  element(LoopElement,LoopMatrix,CircuitElement),
  Element #= CircuitElement,
  NextIndex is Index + 1,
  buildCircuit(Circuit,LoopMatrix,Loop,NextIndex,Loop-
Size).

%Solve a chessloop puzzle with a given ID
%If the id is negative, generates a random puzzle
chessloop(ID):-
  puzzle(ID,Num,Piece1,Piece2,Lines,Cols),
  !,
  display_puzzle(Num,Piece1,Piece2,Lines,Cols),
  pieceCode(Piece1,Code1),
  pieceCode(Piece2,Code2),
  Size is Lines * Cols,
  length(Matrix,Size),
  Max is max(Code1, Code2),
  domain(Matrix,0,Max),
  count(Code1,Matrix,#=,Num),
  count(Code2,Matrix,#=,Num),
  length(LoopMatrix,Size),
  length(Loop,Size),
  restrict(Matrix,Code1,Code2,1,Lines,1,Cols,Loop-
Matrix,Loop,1),
  CircuitSize is Num * 2,
  length(Circuit,CircuitSize),
  buildCircuit(Circuit,LoopMatrix,Loop,1,CircuitSize),
  circuit(Circuit),
  labeling([ff,down,step],Matrix),
  !,
  display_board(Matrix,0,Cols).
```

Ficheiro *puzzles.pl*:

```
%Puzzle Information
%In case ID is negative generates a random puzzle
puzzle(ID,Num,Piece1,Piece2,Lines,Cols):-
  ID < 0,
  !,
  repeat,
  random(2,6,Lines),
  random(3,9,Cols),
  random(2,6,Num),
  random(1,6,Code1),
  random(1,6,Code2),
  Code1 =\= Code2,
  !,
  pieceCode(Piece1,Code1),
  pieceCode(Piece2,Code2).

puzzle(0,2,r,b,2,3).
puzzle(1,2,n,k,2,3).
puzzle(2,3,n,k,4,5).
puzzle(3,2,r,k,2,4).
puzzle(4,3,r,k,4,5).
puzzle(5,2,b,n,3,3).
puzzle(6,4,b,n,4,4).
puzzle(7,4,b,n,3,5).
puzzle(8,4,b,k,4,6).
puzzle(9,4,b,k,5,5).
puzzle(10,3,n,r,3,4).
puzzle(11,5,n,r,3,8).
puzzle(12,2,n,q,2,4).
puzzle(13,3,n,q,3,6).
puzzle(14,3,n,q,4,5).
puzzle(15,4,n,q,4,7).
```

Ficheiro *display.pl*:

```
%Displays puzzle information
display_puzzle(Num,Piece1,Piece2,Lines,Cols):-
  write('Place '),
  write(Num),
  write(' '),
  writePiece(Piece1),
```

```
  write('s and '),
  writePiece(Piece2),
  write('s on a '),
  write(Lines),
  write('x'),
  write(Cols),
  write(' chess board.\n\n').

display_nl(Cols,Cols):-
  !.

%Displays a new board line
display_nl(Col,Cols):-
  !,
  write('--'),
  Next is Col + 1,
  display_nl(Next,Cols).

display_board([],_,_):-
  nl,
  nl,
  !.

display_board(Matrix,Cols,Cols):-
  !,
  nl,
  display_nl(0,Cols),
  nl,
  display_board(Matrix,0,Cols).

%Displays the chess board
display_board([Head|Tail],Col,Cols):-
  pieceCode(Piece,Head),
  write(Piece),
  write('|'),
  Next is Col + 1,
  display_board(Tail,Next,Cols).
```

Ficheiro *pieces.pl*:

```
%Gets piece ID
pieceCode(' ',0).
pieceCode(n,1).
```

```
pieceCode(b,2).
pieceCode(k,3).
pieceCode(r,4).
pieceCode(q,5).

%Writes piece description
writePiece(n):-
  write('knight').

writePiece(b):-
  write('bishop').

writePiece(k):-
  write('king').

writePiece(r):-
  write('rook').

writePiece(q):-
  write('queen').

%Returns in Positions all attacks positions a given piece
has from [Line, Col] and their indexes in Indexes
attackPositions(Matrix,3,Line,Col,Cols,Indexes,Posi-
tions):-
  !,
  attackPositionsKing(Matrix,Line,Col,Cols,Indexes,Posi-
tions).

attackPositions(Matrix,1,Line,Col,Cols,Indexes,Posi-
tions):-
  !,
  attackPositionsKnight(Matrix,Line,Col,Cols,Indexes,Po-
sitions).

attackPositions(Matrix,4,Line,Col,Cols,Indexes,Posi-
tions):-
  !,
  attackPositionsRook(Matrix,Line,Col,Cols,Indexes,Posi-
tions).

attackPositions(Matrix,2,Line,Col,Cols,Indexes,Posi-
tions):-
  !,
```

```
  attackPositionsBishop(Matrix,Line,Col,Cols,Indexes,Po-
sitions).

attackPositions(Matrix,5,Line,Col,Cols,Indexes,Posi-
tions):-
  !,
  attackPositionsQueen(Matrix,Line,Col,Cols,Indexes,Posi-
tions).
```

Ficheiro *king.pl*:

```
%Returns in Positions all attacks positions king has from
[Line, Col] and their indexes in Indexes
attackPositionsKing(Matrix,Line,Col,Cols,Indexes,Posi-
tions):-
  LastLine is Line - 1,
  NextLine is Line + 1,
  LastCol is Col - 1,
  NextCol is Col + 1,
  UpLeft = [LastLine,LastCol],
  Up = [LastLine,Col],
  UpRight = [LastLine,NextCol],
  Left = [Line,LastCol],
  Right = [Line,NextCol],
  DownLeft = [NextLine,LastCol],
  Down = [NextLine, Col],
  DownRight = [NextLine,NextCol],
  Possible = [[UpLeft], [Up], [UpRight], [Left], [Right],
[DownLeft], [Down], [DownRight]],
  buildPositions(Matrix,Cols,Possible,[],Indexes,[],Posi-
tions).
```

Ficheiro *knight.pl*:

```
%Returns in Positions all attacks positions knight has
from [Line, Col] and their indexes in Indexes
attackPositionsKnight(Matrix,Line,Col,Cols,Indexes,Posi-
tions):-
  LastLastLine is Line - 2,
  LastLine is Line - 1,
  NextLine is Line + 1,
  NextNextLine is Line + 2,
```

```
  LastLastCol is Col - 2,
  LastCol is Col - 1,
  NextCol is Col + 1,
  NextNextCol is Col + 2,
  UpUpLeft = [LastLastLine,LastCol],
  UpUpRight = [LastLastLine,NextCol],
  UpLeftLeft = [LastLine,LastLastCol],
  UpRightRight = [LastLine,NextNextCol],
  DownDownLeft = [NextNextLine,LastCol],
  DownDownRight = [NextNextLine,NextCol],
  DownLeftLeft = [NextLine,LastLastCol],
  DownRightRight = [NextLine,NextNextCol],
  Possible = [[UpUpLeft], [UpUpRight], [UpLeftLeft], [Up-
RightRight], [DownDownLeft], [DownDownRight],
[DownLeftLeft], [DownRightRight]],
  buildPositions(Matrix,Cols,Possible,[],Indexes,[],Posi-
tions).
```

Ficheiro *rook.pl*:

```
buildPossibleUp(0,_,Up,Up):-
  !.

%Return in Up all possible board positions above [Line,
Col]
buildPossibleUp(Line,Col,Temp,Up):-
  Pos = [Line,Col],
  append(Temp,[Pos],NextUp),
  Next is Line - 1,
  buildPossibleUp(Next,Col,NextUp,Up).

buildPossibleLeft(0,_,Left,Left):-
  !.

%Return in Up all possible board positions to the left of
[Line, Col]
buildPossibleLeft(Col,Line,Temp,Left):-
  Pos = [Line,Col],
  append(Temp,[Pos],NextLeft),
  Next is Col - 1,
  buildPossibleLeft(Next,Line,NextLeft,Left).

buildPossibleRight(Col,Cols,_,Right,Right):-
  Col > Cols,
```

```
  !.

%Return in Up all possible board positions to the right
of [Line, Col]
buildPossibleRight(Col,Cols,Line,Temp,Right):-
  Pos = [Line,Col],
  append(Temp,[Pos],NextRight),
  Next is Col + 1,
  buildPossibleRight(Next,Cols,Line,NextRight,Right).

buildPossibleDown(Line,Lines,_,Down,Down):-
  Line > Lines,
  !.

%Return in Up all possible board positions below [Line,
Col]
buildPossibleDown(Line,Lines,Col,Temp,Down):-
  Pos = [Line,Col],
  append(Temp,[Pos],NextDown),
  Next is Line + 1,
  buildPossibleDown(Next,Lines,Col,NextDown,Down).

%Returns in Positions all attacks positions rook has from
[Line, Col] and their indexes in Indexes
attackPositionsRook(Matrix,Line,Col,Cols,Indexes,Posi-
tions):-
  length(Matrix,Size),
  Lines is Size div Cols,
  LastLine is Line - 1,
  NextLine is Line + 1,
  LastCol is Col - 1,
  NextCol is Col + 1,
  buildPossibleUp(LastLine,Col,[],Up),
  buildPossibleLeft(LastCol,Line,[],Left),
  buildPossibleRight(NextCol,Cols,Line,[],Right),
  buildPossibleDown(NextLine,Lines,Col,[],Down),
  Possible = [Up,Left,Right,Down],
  buildPositions(Matrix,Cols,Possible,[],Indexes,[],Posi-
tions).
```

Ficheiro *bishop.pl*:

```
buildPossibleUpLeft(0,_,UpLeft,UpLeft):-
  !.

buildPossibleUpLeft(_,0,UpLeft,UpLeft):-
  !.

%Return in UpLeft all possible board positions in upleft
diagonal from [Line, Col]
buildPossibleUpLeft(Line,Col,Temp,UpLeft):-
  Pos = [Line,Col],
  append(Temp,[Pos],NextUpLeft),
  NextLine is Line - 1,
  NextCol is Col - 1,
  buildPossibleUpLeft(NextLine,NextCol,NextUpLeft,Up-
Left).

buildPossibleUpRight(_,_,0,UpRight,UpRight):-
  !.

buildPossibleUpRight(Col,Cols,_,UpRight,UpRight):-
  Col > Cols,
  !.

%Return in UpRight all possible board positions in up-
right diagonal from [Line, Col]
buildPossibleUpRight(Col,Cols,Line,Temp,UpRight):-
  Pos = [Line,Col],
  append(Temp,[Pos],NextUpRight),
  NextLine is Line - 1,
  NextCol is Col + 1,
  buildPossibleUpRight(NextCol,Cols,NextLine,Nex-
tUpRight,UpRight).

buildPossibleDownLeft(_,_,0,DownLeft,DownLeft):-
  !.

buildPossibleDownLeft(Line,Lines,_,DownLeft,DownLeft):-
  Line > Lines,
  !.

%Return in DownLeft all possible board positions in
downleft diagonal from [Line, Col]
```

```
buildPossibleDownLeft(Line,Lines,Col,Temp,DownLeft):-
  Pos = [Line,Col],
  append(Temp,[Pos],NextDownLeft),
  NextLine is Line + 1,
  NextCol is Col - 1,
  buildPossibleDownLeft(Next-
Line,Lines,NextCol,NextDownLeft,DownLeft).

buildPossibleDownRight(Line,Lines,_,_,DownRight,Down-
Right):-
  Line > Lines,
  !.

buildPossibleDownRight(_,_,Col,Cols,DownRight,Down-
Right):-
  Col > Cols,
  !.

%Return in DownRight all possible board positions in
downright diagonal from [Line, Col]
buildPossibleDownRight(Line,Lines,Col,Cols,Temp,Down-
Right):-
  Pos = [Line,Col],
  append(Temp,[Pos],NextDownRight),
  NextLine is Line + 1,
  NextCol is Col + 1,
  buildPossibleDownRight(Next-
Line,Lines,NextCol,Cols,NextDownRight,DownRight).

%Returns in Positions all attacks positions bishop has
from [Line, Col] and their indexes in Indexes
attackPositionsBishop(Matrix,Line,Col,Cols,Indexes,Posi-
tions):-
  length(Matrix,Size),
  Lines is Size div Cols,
  LastLine is Line - 1,
  NextLine is Line + 1,
  LastCol is Col - 1,
  NextCol is Col + 1,
  buildPossibleUpLeft(LastLine,LastCol,[],UpLeft),
  buildPossibleUpRight(NextCol,Cols,LastLine,[],UpRight),
  buildPossibleDownLeft(Next-
Line,Lines,LastCol,[],DownLeft),
  buildPossibleDownRight(Next-
Line,Lines,NextCol,Cols,[],DownRight),
```

```
  Possible = [UpLeft,UpRight,DownLeft,DownRight],
  buildPositions(Matrix,Cols,Possible,[],Indexes,[],Posi-
tions).
```

Ficheiro *queen.pl*:

```
%Returns in Positions all attacks positions queen has
from [Line, Col] and their indexes in Indexes
attackPositionsQueen(Matrix,Line,Col,Cols,Indexes,Posi-
tions):-
  length(Matrix,Size),
  Lines is Size div Cols,
  LastLine is Line - 1,
  NextLine is Line + 1,
  LastCol is Col - 1,
  NextCol is Col + 1,
  buildPossibleUpLeft(LastLine,LastCol,[],UpLeft),
  buildPossibleUp(LastLine,Col,[],Up),
  buildPossibleUpRight(NextCol,Cols,LastLine,[],UpRight),
  buildPossibleLeft(LastCol,Line,[],Left),
  buildPossibleRight(NextCol,Cols,Line,[],Right),
  buildPossibleDownLeft(Next-
Line,Lines,LastCol,[],DownLeft),
  buildPossibleDown(NextLine,Lines,Col,[],Down),
  buildPossibleDownRight(Next-
Line,Lines,NextCol,Cols,[],DownRight),
  Possible = [UpLeft,Up,Up-
Right,Left,Right,DownLeft,Down,DownRight],
  buildPositions(Matrix,Cols,Possible,[],Indexes,[],Posi-
tions).
```