Chess Loop Puzzles
Resolução de Problema de Decisão usando Programação em Lógica com Restrições

Juliana Marques e Helena Montenegro

FEUP-PLOG, Turma 3MIEIC06, Chess Loop_4

Faculdade de Engenharia da Universidade do Porto,
Rua Roberto Frias, sn, 4200-465 Porto, Portugal

Resumo. O projeto foi desenvolvido no Sistema de Desenvolvimento SICStus Prolog no âmbito da unidade curricular de Programação em Lógica, cujo objetivo é resolver um problema de decisão implementando restrições. O problema de decisão escolhido é baseado num puzzle lógico denominado *chess loop*. Este puzzle tem como principal objetivo o posicionamento de duas peças num tabuleiro de xadrez, de tal forma que cada peça ataque apenas uma outra peça de tipo diferente, formando através dos ataques um *loop*. Através da manipulação de predicados disponibilizados pelo SICStus Prolog, mostramos, neste artigo que foi possível a resolução deste problema de forma eficiente.

Palavras-chave: restrições, xadrez, SICtus, prolog

1 Introdução

O objetivo deste trabalho foi implementar a resolução de um problema de decisão ou otimização em prolog com restrições. O problema escolhido, trata-se de um problema de decisão que consiste na resolução dum puzzle lógico, chess loop.

Este artigo descreve detalhadamente a abordagem do grupo para implementar uma resolução capaz de resolver este problema para qualquer dimensão do tabuleiro, assim como para qualquer tipo de peça, bem como a análise detalhada da mesma.

Para além disso, será mostrada a interface usada para visualizar e demonstrar de forma percetível as soluções encontradas para cada caso do problema.

De seguida, serão analisadas as estatísticas de resolução do problema com diferentes complexidades e diferentes labelings.

E finalmente, a conclusão do projeto realizado.

2 Descrição do Problema

O problema chess loop puzzles consiste em posicionar diferentes tipos de peças do xadrez, como a rainha, o rei, o bispo, o cavalo e a torre, num tabuleiro com dimensões variáveis de forma a que cada peça ataque, de acordo com o seu movimento característico, uma outra peça de tipo diferente, formando um ciclo.

Para encontrar a solução para estes problemas é necessário fornecer as dimensões do tabuleiro, os dois tipos diferentes de peças assim como a quantidade de cada uma.

3 Abordagem

3.1 Variáveis de Decisão

A variável de decisão utilizada na função principal foi uma lista representante do tabuleiro do jogo. Os índices da lista representam a posição das diferentes células do tabuleiro, enquanto que o valor na lista representa o tipo de peça. Cada peça de xadrez é representada por um número inteiro:

0 – célula vazia;
1 – torre;
2 – bispo;
3 – cavalo;
4 – rei;
5 – rainha;

O domínio da nossa variável de decisão é, por tanto, um valor entre 0 e 5.

Numa das restrições é gerada uma lista que contém as posições das células onde as peças foram posicionadas formando o ciclo. Esta lista é também uma variável de decisão utilizada depois numa outra restrição, como explicado no seguinte tópico.

Para o movimento de cada peça, fizemos funções cuja variável de decisão era uma lista com apenas dois elementos, representando a posição inicial e final do movimento da peça, sendo o domínio entre 1 e o número de células do tabuleiro.

3.2 Restrições

Em primeiro lugar, foi necessário restringir a quantidade de cada peça que existe no tabuleiro, tendo sido utilizada a função ***count()*** para este efeito.

Depois, aplicou-se a restrição de que nenhuma das colunas ou linhas nas bordas do tabuleiro pode estar vazia, de modo a respeitar as dimensões do tabuleiro pedidas em cada caso do problema.

A restrição relativa ao posicionamento de cada peça foi feita de modo recursivo, na função ***restrict_movement()***, com o objetivo de os movimentos das peças formarem um ciclo.

Para formar o ciclo, verificamos que em cada iteração, a peça era colocada na posição de destino da iteração anterior e que na última iteração, o destino da última peça posicionada coincidia com a posição da primeira peça posicionada.

Na função recursiva mencionada no parágrafo anterior foi necessário restringir os movimentos de cada peça, o que foi implementado nas funções intituladas ***[nome_da_peça]move()***.

Na função ***choose_move()***, é chamada a função do movimento relativa à peça recebida como argumento.

Finalmente, na função ***restriction_cant_move()*** foi implementada a restrição de que cada célula que se consegue alcançar com um movimento de uma peça tem de estar vazia, à exceção da célula que contém a peça de diferente tipo necessária.

Para tal, utilizamos uma lista gerada na função da restrição anterior, que contém em cada célula a posição das peças posicionadas formando um ciclo, para verificarmos que uma peça não podia atacar outras peças.

Para definir quais as peças que não podiam ser atacadas, criamos o predicado ***make_list()*** que origina uma lista baseada na anteriormente mencionada, onde desta foram eliminados a posição da própria peça e a posição seguinte, tal como peças para as quais se podia alcançar com o movimento mas cujo movimento este necessitaria de um salto por cima da outra peça.

Para cada um dos movimentos de cada peça criamos um predicado ***not_[peça]_move()*** que restringe todos os movimentos que a peça não pode fazer.

3.3 Estratégia de Pesquisa

Na ponderação da heurística que escolhemos para o projeto, verificamos que **occurrence** era a que mais se aplicava. No tema Chess Loop Puzzles, diferentes peças estão sujeitas a diferentes restrições, existindo peças muito mais restringidas que outras. Deste modo, faz sentido utilizar uma heurística cuja escolha da variável se dá pelo número de restrições. A heurística de escolha de valor utilizada foi **enum**.

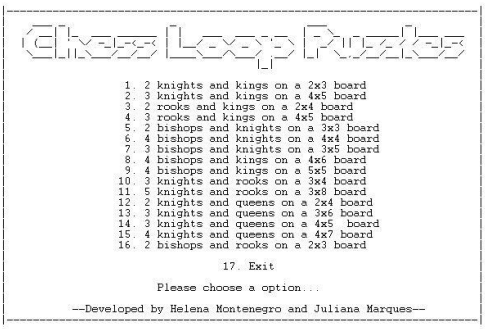
4 Visualização da Solução

A aplicação é iniciada através do comando "menu.". A partir daí aparece uma interface fácil de utilizar que contém todas as opções de problemas que se encontram no enunciado do problema de decisão Chess Loop Puzzles. Em anexo encontram-se os resultados encontrados para cada puzzle, tal como a sua solução, para efeitos de comparação.

Na visualização, utilizamos o predicado ***display_board()*** que mostra o tabuleiro com o posicionamento das peças obtido, de modo semelhante às imagens mostradas no enunciado para cada caso.

Aliada a esta função, criamos também a função ***make_board_matrix()*** que torna a lista representante do tabuleiro numa matriz.

Mostramos também o tempo que o programa demorou a encontrar a solução visualizada.



5 Resultados

A seguinte tabela foi originada ao testar diferentes tamanhos de tabuleiros com o mesmo tipo de peças. Em específico, foram utilizadas as opções 12, 13, 14 e 15 do menu da aplicação, cujas peças eram o cavalo e a rainha.

Tamanho do tabuleiro	Tempo (segundos)
2X4	0.109
3X6	0.219
4X5	0.281
4X7	0.390

A partir da tabela anterior, podemos concluir que quanto maior for o tabuleiro, mais tempo a aplicação demora a posicionar as peças, apesar de a diferença ser bastante pequena.

A seguinte tabela destina-se à análise do tempo resultante de utilizar as diversas combinações de heurísticas para escolha de variáveis e escolha de valores aplicadas a um tabuleiro.

Para uma maior diferenciação nos valores do tempo, escolhemos a opção 10 do menu, cujo tabuleiro é 3x4 e as peças posicionadas são cavalos e torres. Os valores de contagem do tempo encontram-se em segundos.

	leftmost	**min**	**max**	**ff**	**ffc**	**occurrence**	**max_regret**
step	0.638	1.402	0.302	0.140	20.604	0.142	0.141
enum	0.744	1.282	0.317	0.172	21.578	0.156	0.156
bisect	0.675	1.549	1.656	0.141	20.929	0.141	0.140
middle	29.716	36.061	0.277	29.959	23.414	0.140	14.920
median	31.726	39.576	0.274	29.938	22.786	0.172	12.940

Comparando apenas as heurísticas para a escolha de variáveis, podemos observar uma grande diferença negativa entre ***ffc*** e todas as outras, dado que apresenta valores temporais acima dos 20's em qualquer combinação.

Entre as melhores heurísticas, encontra-se ***occurrence***, a utilizada na entrega deste projeto, ***max_regret*** e ***ff***.

Comparando agora as heurísticas para a escolha de valores, observamos que, de modo geral, ***step***, ***enum*** e ***bisect*** apresentam valores idênticos. Já ***middle*** e ***median*** apresentam, de maneira geral, uma grande diferença em relação a todas as outras heurísticas.

No entanto, podemos observar que aliadas à heurística ***max***, ***middle*** e ***median*** apresentam-se com melhores tempos que as restantes e que, utilizadas com a heurística ***occurrence***, apresentam resultados semelhantes às outras heurísticas.

Avaliando a tabela de um modo geral, podemos concluir que a heurística ***occurrence*** é melhor que todas as outras, sendo a única coluna cujos todos os valores são inferiores a meio segundo. Todas as heurísticas de escolha de valor associadas a esta apresentam resultados idênticos, pelo que, para o nosso projeto, qualquer uma delas é eficiente.

6 Conclusões e Trabalho Futuro

Após a realização deste trabalho conclui-se que a linguagem Prolog, mais especificamente os módulos de resolução de restrições são bastante poderosos permitindo a resolução de uma ampla variedade de questões de decisão e otimização, resolvendo problemas de alguma complexidade bastante rápido.

Depois de entendido o funcionamento por detrás das variáveis de decisão, formas de restringir o domínio de variáveis, a maneira como o ***labeling*** funciona e, por fim, o potencial de todos os predicados que a biblioteca 'clpfd' disponibiliza, tornou-se mais fácil a resolução do problema proposto.

Os resultados obtidos demonstram que, no caso do nosso projeto, a heurística ***occurrence*** é a mais eficiente. Isto porque, tal como mencionamos no tópico 3.3, o nosso trabalho exige diferentes restrições para diferentes peças, pelo que começar a posicionar peças com maiores restrições limita mais as possíveis soluções e origina tempos menores.

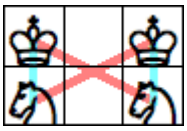
A solução implementada pelo nosso grupo correspondeu às expetativas, sendo capaz de resolver qualquer tipo de puzzle deste tipo de forma eficiente.

7 Referências

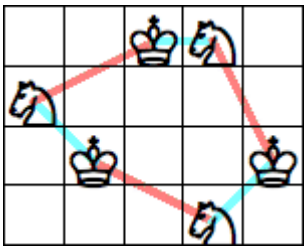
1. Chees loops, https://www2.stetson.edu/~efriedma/puzzle/chessloop/
2. Slides da disciplina sobre Plr, https://moodle.up.pt/pluginfile.php/83295/mod_resource/content/6/PLR_SICStus.pdf
3. SICStus Prolog, https://sicstus.sics.se/

8 Anexos

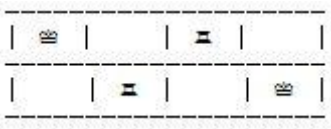
8.1 2 cavalos e reis num tabuleiro 2x3



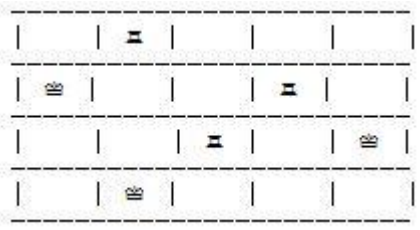
8.2 3 cavalos e reis num tabuleiro 4x5

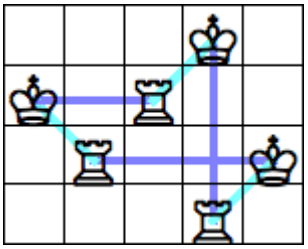


8.3 2 torres e reis num tabuleiro 2x4

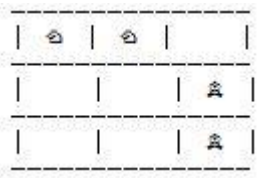


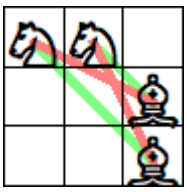
8.4 3 torres e reis num tabuleiro 4x5



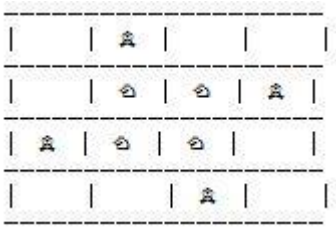


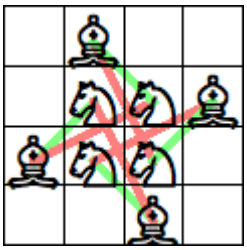
8.5 2 cavalos e bispos num tabuleiro 3x3



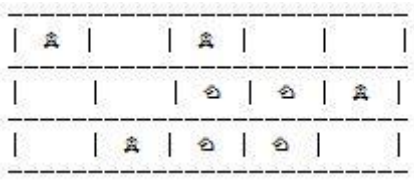


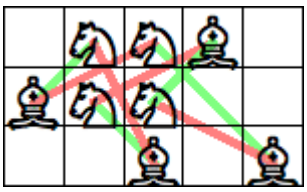
8.6 4 cavalos e bispos num tabuleiro 4x4



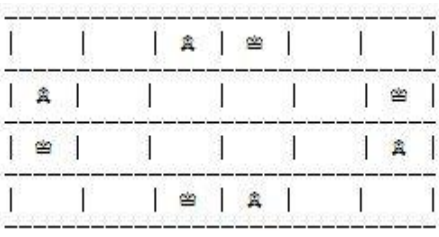


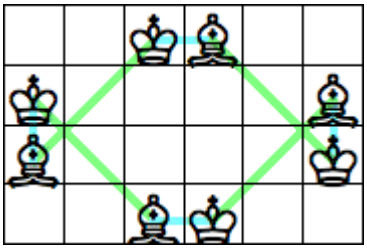
8.7 4 cavalos e bispos num tabuleiro 3x5



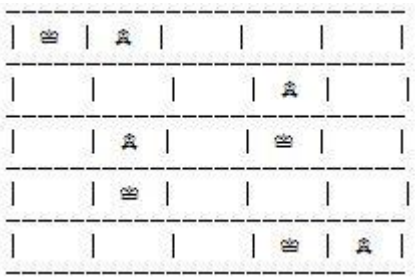


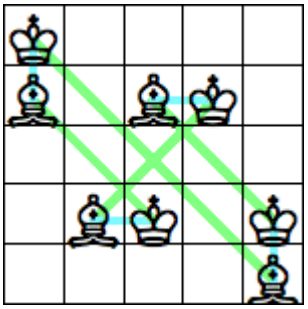
8.8 4 bispos e reis num tabuleiro 4x6



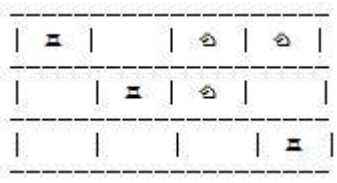


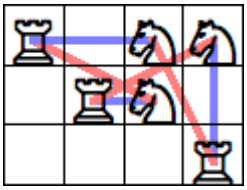
8.9 4 bispos e reis num tabuleiro 5x5



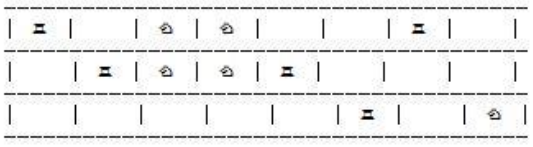


8.10 3 cavalos e torres num tabuleiro 3x4

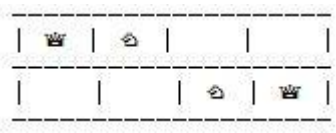




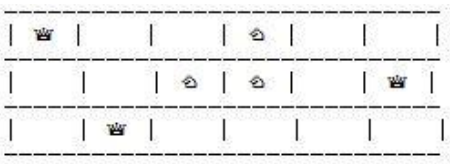
8.11 5 cavalos e torres num tabuleiro 3x8

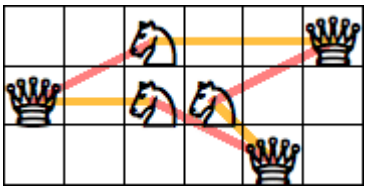


8.12 2 cavalos e rainhas num tabuleiro 2x4

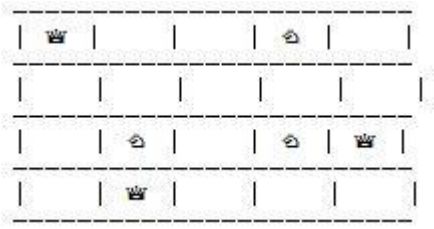


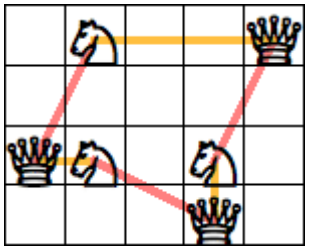
8.13 3 cavalos e rainhas num tabuleiro 3x6



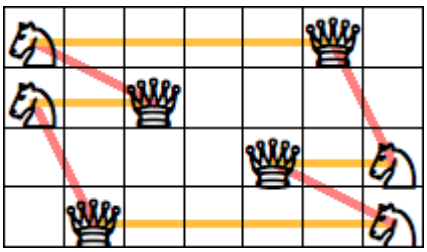


8.14 3 cavalos e reis num tabuleiro 4x5





8.15 4 cavalos e reis num tabuleiro 4x7



8.16 Código Fonte

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

/*------------------------------MENU------------------------------*/
/*Function that starts the menu.*/
menu :-
    print_options,
    read(Input),
    analise_input(Input).

/*Function which calls the decision problem with the right arguments according to the user's input.*/
analise_input(16):-statistics(walltime, [Start,_]),
    chess_loops(3,2,[1,2],[2,2]),
    statistics(walltime, [End,_]),
     Time is End - Start,
     format('~3d seconds.~n', [Time]),
    menu.

analise_input(1):-statistics(walltime, [Start,_]),
    chess_loops(3,2,[4,3],[2,2]),
    statistics(walltime, [End,_]),
     Time is End - Start,
     format('~3d seconds.~n', [Time]),
    menu.

analise_input(2):-
statistics(walltime, [Start,_]),
    chess_loops(5,4,[3,4],[3,3]),
    statistics(walltime, [End,_]),
     Time is End - Start,
     format('~3d seconds.~n', [Time]),
    menu.

analise_input(2):-
statistics(walltime, [Start,_]),
    chess_loops(5,4,[3,4],[3,3]),
    statistics(walltime, [End,_]),
     Time is End - Start,
     format('~3d seconds.~n', [Time]),
    menu.

analise_input(3):-statistics(walltime, [Start,_]),
    chess_loops(4,2,[4,1],[2,2]),
    statistics(walltime, [End,_]),
     Time is End - Start,
     format('~3d seconds.~n', [Time]),
    menu.

analise_input(4):-statistics(walltime, [Start,_]),
    chess_loops(5,4,[1,4],[3,3]),
    statistics(walltime, [End,_]),
     Time is End - Start,
     format('~3d seconds.~n', [Time]),
    menu.

analise_input(5):-statistics(walltime, [Start,_]),
    chess_loops(3,3,[3,2],[2,2]),
    statistics(walltime, [End,_]),
     Time is End - Start,
     format('~3d seconds.~n', [Time]),
    menu.
```

```
analise_input(6):-statistics(walltime, [Start,_]),
    chess_loops(4,4,[2,3],[4,4]),
    statistics(walltime, [End,_]),
     Time is End - Start,
     format('~3d seconds.~n', [Time]),
    menu.

analise_input(7):-statistics(walltime, [Start,_]),
    chess_loops(5,3,[2,3],[4,4]),
    statistics(walltime, [End,_]),
     Time is End - Start,
     format('~3d seconds.~n', [Time]),
    menu.

analise_input(8):-statistics(walltime, [Start,_]),
    chess_loops(6,4,[2,4],[4,4]),
    statistics(walltime, [End,_]),
     Time is End - Start,
     format('~3d seconds.~n', [Time]),
    menu.

analise_input(9):-statistics(walltime, [Start,_]),
    chess_loops(5,5,[4,2],[4,4]),
    statistics(walltime, [End,_]),
     Time is End - Start,
     format('~3d seconds.~n', [Time]),
    menu.

analise_input(10):-statistics(walltime, [Start,_]),
    chess_loops(4,3,[1,3],[3,3]),
    statistics(walltime, [End,_]),
     Time is End - Start,
     format('~3d seconds.~n', [Time]),
    menu.

analise_input(11):-statistics(walltime, [Start,_]),
    chess_loops(8,3,[1,3],[5,5]),
    statistics(walltime, [End,_]),
     Time is End - Start,
     format('~3d seconds.~n', [Time]),
    menu.

analise_input(12):-statistics(walltime, [Start,_]),
    chess_loops(4,2,[5,3],[2,2]),
    statistics(walltime, [End,_]),
     Time is End - Start,
     format('~3d seconds.~n', [Time]),
    menu.

analise_input(13):-statistics(walltime, [Start,_]),
    chess_loops(6,3,[5,3],[3,3]),
    statistics(walltime, [End,_]),
     Time is End - Start,
     format('~3d seconds.~n', [Time]),
    menu.
```

```
analise_input(14):-statistics(walltime, [Start,_]),
    chess_loops(5,4,[5,3],[3,3]),
    statistics(walltime, [End,_]),
    Time is End - Start,
    format('~3d seconds.~n', [Time]),
    menu.

analise_input(15):-
 statistics(walltime, [Start,_]),
    chess_loops(7,4,[3,5],[4,4]),
    statistics(walltime, [End,_]),
    Time is End - Start,
    format('~3d seconds.~n', [Time]),
    menu.

analise_input(17):-
    write('Exiting...'), nl.

analise_input(_) :-
    write('\nError: that option does not exist.\n\n'),
    menu.

/*Function that prints the menu in the console.*/
print_options :- nl,
    write('        |-------------------------------------------------------------------------|'),nl,
    write('        |     ___ _                  _                    ___          _          |'),nl,
    write('        |    / __| |_  ___ ______ | |  ___  ___ _ __  | _ \\_  _ _____| |___ ___  |'),nl,
    write('        |   | (__| \' \\/ -_|_-<_-< | |__/ _ \\/ _ \\ \'_ \\ |  _/ || |_ /_ / / -_|_-<  |'),nl,
    write('        |    \\___|_||_\\___/__/__/ |____\\___/\\___/ .__/ |_|  \\_,_/__/__|_\\___/__/  |'),nl,
    write('        |                                       |_|                               |'),nl,
    write('        |                                                                         |'),nl,
    write('        |                1. 2 knights and kings on a 2x3 board                    |'),nl,
    write('        |                2. 3 knights and kings on a 4x5 board                    |'),nl,
    write('        |                3. 2 rooks and kings on a 2x4 board                      |'),nl,
    write('        |                4. 3 rooks and kings on a 4x5 board                      |'),nl,
    write('        |                5. 2 bishops and knights on a 3x3 board                  |'),nl,
    write('        |                6. 4 bishops and knights on a 4x4 board                  |'),nl,
    write('        |                7. 3 bishops and knights on a 3x5 board                  |'),nl,
    write('        |                8. 4 bishops and kings on a 4x6 board                    |'),nl,
    write('        |                9. 4 bishops and kings on a 5x5 board                    |'),nl,
    write('        |               10. 3 knights and rooks on a 3x4 board                    |'),nl,
    write('        |               11. 5 knights and rooks on a 3x8 board                    |'),nl,
    write('        |               12. 2 knights and queens on a 2x4 board                   |'),nl,
    write('        |               13. 3 knights and queens on a 3x6 board                   |'),nl,
    write('        |               14. 3 knights and queens on a 4x5  board                  |'),nl,
    write('        |               15. 4 knights and queens on a 4x7 board                   |'),nl,
    write('        |               16. 2 bishops and rooks on a 2x3 board                    |'),nl,
    write('        |                                                                         |'),nl,
    write('        |                                17. Exit                                 |'),nl,
    write('        |                                                                         |'),nl,
    write('        |                      Please choose a option...                          |'),nl,
    write('        |                                                                         |'),nl,
    write('        |          --Developed by Helena Montenegro and Juliana Marques--         |'),nl,
    write('        |-------------------------------------------------------------------------|'),nl,
    nl,nl,nl.
```

```
/*--------------------------DISPLAY-------------------------------*/
/*Function to display the board received as argument*/
display_board([H|T]) :-
    length(H, Len),
    display_board([H|T], Len).

display_board([], Len) :- write(' '), display_sep(Len).
display_board([H|T], Len) :-
    write(' '),
    display_sep(Len),
    write(' |'),
    display_line(H),
    display_board(T, Len).

/*Auxiliar function to display separator between lines*/
display_sep(0) :- nl.
display_sep(Len) :- write('-----'), L is Len-1, display_sep(L).

/*Auxiliar function to display the line of the board received as argument*/
display_line([]) :- nl.
display_line([H|T]) :-  translate_symbol(H, Symbol), write(' '), write(Symbol), write(' |'), display_line(T).

/*Function to translate numerical values in board to easy to recognize symbols*/
translate_symbol(0, '  ').
translate_symbol(1, Char) :- char_code(Char, 9814).
translate_symbol(2, Char) :- char_code(Char, 9815).
translate_symbol(3, Char) :- char_code(Char, 9816).
translate_symbol(4, Char) :- char_code(Char, 9812).
translate_symbol(5, Char) :- char_code(Char, 9813).

/*------------------------END-OF-DISPLAY--------------------------*/
```

```
/*------------------------MAIN-FUNCTION--------------------------*/
/*Main funcion which recieves the number of each pawn in the loop and the board's dimensions and does the chess loop.
The variable is the board that will contain the elements, represented by an array.
The numbers represent:
 0 - empty cell
 1 - rook
 2 - bishop
 3 - horse
 4 - king
 5 - queen
*/
chess_loops(TamX, TamY, Pawns, Quantity) :-
    TamTot is TamX*TamY,
    length(Board, TamTot),
    domain(Board, 0, 5),

%Restriction 1: restricts number of each type of pawn in the board
    restrict_number_pawns(Board, Pawns, Quantity, 0, NCells),
    WhiteCells is TamTot-NCells,
    count(0, Board, #=, WhiteCells),
```

```
%Restriction 2: there cant be an empty row or column on the edges
    getOuterColumn(TamX, TamY, OuterColumn, Board),
    length(OuterColumn, OuterLen),
    count(0, OuterColumn, #<, OuterLen),
    getInnerColumn(TamX, TamY, InnerColumn, Board),
    length(InnerColumn, InnerLen),
    count(0, InnerColumn, #<, InnerLen),
    length(FirstRow, TamX),
    append(FirstRow, _, Board),
    count(0, FirstRow, #<, TamX),
    length(LastRow, TamX),
    append(_, LastRow, Board),
    count(0, LastRow, #<, TamX),

%Restriction 3: restricts the place where each pawn can be (further explained in the function that applies it)
    nth1(1, Quantity, Quant),
    restrict_movement(Quant, Pawns, Board, TamX, TamY, List),

%Restriction 4: with one movement, one pawn can reach one and one only other pawn
    restriction_cant_move(Board, Pawns, List, TamX, TamY, 1),
    append(Board, List, B), !,

%End of restrictions
    labeling([occurrence, enum], B),
    make_board_matrix(Board, BoardMatrix, [], TamX),
    display_board(BoardMatrix).

/*Restriction that from one cell with a pawn only one movement can reach a cell with a pawn, the rest of the cells
reachable with one movement have to be empty.*/
restriction_cant_move(Board, [Pawn, _], List, TamX, TamY, Count) :-
    length(List, Count),
    element(Count, List, Place),
    element(1, List, Place3),
    make_list(Pawn, Count, Place, Place3, TamX, TamY, List, ListToSend, 1),
    cant_move(Board, Pawn, Place, ListToSend, TamX, TamY).
restriction_cant_move(Board, [Pawn, Pawn2], List, TamX, TamY, Count) :-
    element(Count, List, Place),
    C is Count+1,
    element(C, List, Place3),
    make_list(Pawn, Count, Place, Place3, TamX, TamY, List, ListToSend, 1),
    cant_move(Board, Pawn, Place, ListToSend, TamX, TamY),
    restriction_cant_move(Board, [Pawn2, Pawn], List, TamX, TamY, C).

/*Function that goes through the not instantiates list of positions of each pawn and returns the list of positions that
the pawn received as argument (whose index in the list is Count, whose position is Place and which attacks the pawn
in the position Place3) can't attack.*/
make_list(_, _, _, _, _, [], [], _).
make_list(Pawn, Count, Place, Place3, TamX, TamY, [H1|T1], List, Aux) :-
    length([H1|T1], Count), Aux = 1, A is Aux+1,
    make_list(Pawn, Count, Place, Place3, TamX, TamY, T1, List, A).
make_list(Pawn, Count, Place, Place3, TamX, TamY, [_|T1], List, Aux) :-
    Aux = Count, A is Aux+1,
    make_list(Pawn, Count, Place, Place3, TamX, TamY, T1, List, A).
make_list(Pawn, Count, Place, Place3, TamX, TamY, [_|T1], List, Aux) :-
    Aux is Count+1, A is Aux+1,
    make_list(Pawn, Count, Place, Place3, TamX, TamY, T1, List, A).
make_list(Pawn, Count, Place, Place3, TamX, TamY, [H1|T1], List, Aux) :- Pawn \= 3, Pawn \=4,
    translate_position_coords(Place, X1, Y1, TamX, TamY),
    translate_position_coords(H1, X2, Y2, TamX, TamY),
    translate_position_coords(Place3, X3, Y3, TamX, TamY),
    ((X2#=X1 #/\ Y2#\=Y1 #/\ X3#=X1 #/\ Y3#\=Y1 #/\ Place3 #> Place #/\ H1 #> Place3) #\/
    (X2#=X1 #/\ Y2#\=Y1 #/\ X3#=X1 #/\ Y3#\=Y1 #/\ Place3 #< Place #/\ H1 #< Place3) #\/
    (X2#\=X1 #/\ Y2#=Y1 #/\ X3#\=X1 #/\ Y3#=Y1 #/\ Place3 #> Place #/\ H1 #> Place3) #\/
    (X2#\=X1 #/\ Y2#=Y1 #/\ X3#\=X1 #/\ Y3#=Y1 #/\ Place3 #< Place #/\ H1 #< Place3) #\/
    (X2#=X1+A #/\ Y2#=Y1+B #/\ A#\= 0 #/\ X3#=X1+B #/\ Y3#=Y1+B #/\ B#\= 0 #/\ Place3 #< Place #/\ H1 #< Place3) #\/
    (X2#=X1+A #/\ Y2#=Y1-A #/\ A#\= 0 #/\ X3#=X1+B #/\ Y3#=Y1-B #/\ B#\= 0 #/\ Place3 #< Place #/\ H1 #< Place3)) #<=> Val,
    interpret_val(Val, Pawn, Count, Place, Place3, TamX, TamY, [H1|T1], List, Aux).
make_list(Pawn, Count, Place, Place3, TamX, TamY, [H1|T1], [H2|T2], Aux) :-
    A is Aux+1,
    H2 #= H1,
    make_list(Pawn, Count, Place, Place3, TamX, TamY, T1, T2, A).
```

```
/*Function auxiliary to the one above, to ensure the restriction done before #<=> is imposed. It's useful for
cases where for the pawn to arrive at another pawn it has to jump. In these cases the other pawn can be at a place
reachable from the initial pawn with one movement.*/
interpret_val(Val, Pawn, Count, Place, Place3, TamX, TamY, [H1|T1], [H2|T2], Aux) :-
    (H2 #= H1) #\ Val,
    A is Aux+1,
    make_list(Pawn, Count, Place, Place3, TamX, TamY, T1, T2, A).

/*Function that goes through the list generated in make_list() and ensures that the Pawn can't reach any of
the pawns positioned in the elements of this list. Place refers to the place where the Pawn is positioned.*/
cant_move(_, _, _, [], _, _).
cant_move(Board, Pawn, Place, [Place1 | Rest], TamX, TamY) :-
    not_choose_move(Pawn, Place, Place1, TamX, TamY),
    cant_move(Board, Pawn, Place, Rest, TamX, TamY).

/*Function that restricts the position of the pawn received as argument.
Recursively, it will choose a pawn of type received as argument in Pawn. In one movement the pawn has to reach an enemy pawn, so
the function gets the position of two pawns in the board, one of the received type and other of any other type and checks if
in one movement the first pawn can get to the second pawn's position by calling the function for the respective type of movement.*/
restrict_movement(0, _, _, _, _, []).
restrict_movement(Quantity, [Pawn, Pawn2], Board, TamX, TamY, ListToReturn) :-
    Quantity > 0,
    element(Place, Board, Pawn),
    element(Place2, Board, Pawn2),
    element(Place3, Board, Pawn),
    Place3 #\= Place,
    choose_move(Pawn, Place, Place2, TamX, TamY, Board),
    choose_move(Pawn2, Place2, Place3, TamX, TamY, Board),
    List = [Place, Place2],
    Q is Quantity-1,
    restrict_movement(Q, List, Place3, Place, [Pawn,Pawn2], Board, TamX, TamY, ListToReturn).

/*This function is called for the second and so on iterations, to make sure that the recursivity isn't applied to the same pawn as
the previous iteration.*/
restrict_movement(0, ListToReturn, _, _, _, _, _, _, ListToReturn).
restrict_movement(Quantity, List, Place, FirstPlace, [Pawn, Pawn2], Board, TamX, TamY, ListToReturn) :-
    Quantity > 0,
    element(Place2, Board, Pawn2),
    diff_place(Place2, List),
    element(Place3, Board, Pawn),
    place3_place1(Place3, Place, FirstPlace, Quantity, List),
    choose_move(Pawn, Place, Place2, TamX, TamY, Board),
    choose_move(Pawn2, Place2, Place3, TamX, TamY, Board),
    append(List, [Place], L),
    append(L, [Place2], L4),
    Q is Quantity-1,
    restrict_movement(Q, L4, Place3, FirstPlace, [Pawn, Pawn2], Board, TamX, TamY, ListToReturn).

/*Function that defines the relationship between the cell to which the second pawn can move to (Place) and the cell where the first pawn that was
position is (FirstPlace), as well as the cell to which the first pawn that was positioned in the same iteration is (Place1).
If the function restrict_movement is in the last iteration the place is the place resulting of the movement of the last pawn positioned and,
as such, has to overlap with the FirstPlace, where the first cell to be positioned is.
If the function is at any other iteration but the last, the Place has to be a cell where no other cell was positioned so far.*/
place3_place1(Place, _,FirstPlace, Q, _) :- Q#=1, Place #= FirstPlace.
place3_place1(Place, Place1, _, Q, List) :- Q#>1, Place #\= Place1, diff_place(Place, List).

/*Function that checks if the first argument is not a member of the list received as the second argument (similar to member, but with
restrictions).*/
diff_place(_, []) :- !.
diff_place(Place, [PlaceDiff|Rest]) :-
    Place #\= PlaceDiff,
    diff_place(Place, Rest).

/*Function that applies restriction to the number of pawns in the board.
It receives the Board, the list of Paws that should exist in the board, the list of quantities of the pawns in the board, an auxiliary
value meant to help discover the last argument to be returned which is the number of pawns of any type in the board.*/
restrict_number_pawns(_, [], [], Count, Count).
restrict_number_pawns(Board, [Pawn1|RestOfPawns], [Quant1|RestOfQuants], AuxCount, Count) :-
    count(Pawn1, Board, #=, Quant1),
    Aux is AuxCount + Quant1,
    restrict_number_pawns(Board, RestOfPawns, RestOfQuants, Aux, Count).
```

```
/*Auxiliar function that gets the last column of the board in order to use it in the chess_loops to check if it's not filled with empty cells.
It receives the dimensions of the board (TamX and TamY) and the board and it returns the last column.*/
getOuterColumn(_, 0, [], _).
getOuterColumn(TamX, TamY, [H|T], Board) :-
    Pos is TamX * TamY,
    element(Pos, Board, H),
    Tam is TamY-1,
    getOuterColumn(TamX, Tam, T, Board).

/*Auxiliar function that gets the first column of the board in order to use it in the chess_loops to check if it's not filled with empty cells.
It receives the dimensions of the board (TamX and TamY) and the board and it returns the first column.*/
getInnerColumn(_, 0, [], _).
getInnerColumn(TamX, TamY, [H|T], Board) :-
    Pos is TamX * TamY - TamX + 1,
    element(Pos, Board, H),
    Tam is TamY-1,
    getInnerColumn(TamX, Tam, T, Board).

/*Funtion that turns the board's list into a matrix.
It receives the Board, and the size of a row of the board and it returns the matrix.*/
make_board_matrix([], BoardMatrix, BoardMatrix, _).
make_board_matrix(Board, BoardMatrix, AuxBoard, TamX) :-
    get_line(Board, Rest, TamX, Line),
    append(AuxBoard, [Line], AuxToSend),
    make_board_matrix(Rest, BoardMatrix, AuxToSend, TamX).

/*Auxiliar function that get's one line of the board and returns the rest of the board.*/
get_line(Board, Board, 0, []).
get_line([H|T], Rest, TamX, [H1|T1]) :-
    H1 = H,
    Tam is TamX - 1,
    get_line(T, Rest, Tam, T1).


/*----------------------END-OF-MAIN-FUNCTION----------------------*/

/*-------------------------PAWNS-MOVEMENTS-------------------------*/

/*Function that depending on the Pawn received as first argument calls its type of movement*/
choose_move(1, Place1, Place2, TamX, TamY, _) :- rook_move(Place1, Place2, TamX, TamY).
choose_move(2, Place1, Place2, TamX, TamY, _) :- bishop_move(Place1, Place2, TamX, TamY).
choose_move(3, Place1, Place2, TamX, TamY, _) :- horse_move(Place1, Place2, TamX, TamY).
choose_move(4, Place1, Place2, TamX, TamY, _) :- king_move(Place1, Place2, TamX, TamY).
choose_move(5, Place1, Place2, TamX, TamY, _) :- queen_move(Place1, Place2, TamX, TamY).

/*Function that ensures that the Place2 is a cell that the Pawn can't reach with one movement*/
not_choose_move(1, Place1, Place2, TamX, TamY) :- not_rook_move(Place1, Place2, TamX, TamY).
not_choose_move(2, Place1, Place2, TamX, TamY) :- not_bishop_move(Place1, Place2, TamX, TamY).
not_choose_move(3, Place1, Place2, TamX, TamY) :- not_horse_move(Place1, Place2, TamX, TamY).
not_choose_move(4, Place1, Place2, TamX, TamY) :- not_king_move(Place1, Place2, TamX, TamY).
not_choose_move(5, Place1, Place2, TamX, TamY) :- not_queen_move(Place1, Place2, TamX, TamY).

/*Function that translates a cell's position (Pos) in an list to its coordinates in a matrix (X and Y)*/
translate_position_coords(Pos, X, Y, TamX, _) :-
    P #= Pos - 1,
    X #= mod(P, TamX) + 1,
    Y #= P // TamX + 1.

/*The following functions have as arguments:
 Pos1 - the position in the Board array in which the pawn is supposed to be
 Pos2 - a position to which the pawn can move from Pos1
 TamX - the number of columns of the board
 TamY - the number of rows of the board*/

/*Function that defines the bishop's movement.*/
bishop_move(Pos1, Pos2, TamX, TamY) :-
    translate_position_coords(Pos1, X1, Y1, TamX, TamY),
    translate_position_coords(Pos2, X2, Y2, TamX, TamY),
    A#\=0,
    X2 #= X1 + A,
    (Y2 #= Y1 + A #\/ Y2 #= Y1 - A).

/*Function that ensures that the cell Pos2 is not reachable with the bishop's movement from the cell Pos1*/
not_bishop_move(Pos1, Pos2, TamX, TamY) :-
    translate_position_coords(Pos1, X1, Y1, TamX, TamY),
    translate_position_coords(Pos2, X2, Y2, TamX, TamY),
    (X2 #= X1 + A #/\
    (Y2 #\= Y1 + A #/\ Y2 #\= Y1 - A)).
```

```
/*Function that defines the rook's movement.*/
rook_move(Pos1, Pos2, TamX, TamY) :-
    translate_position_coords(Pos1, X1, Y1, TamX, TamY),
    translate_position_coords(Pos2, X2, Y2, TamX, TamY),
    A#\=0,
    ((X1 #= X2 #/\ Y1 #= Y2+A) #\/
    (X1 #= X2 + A #/\ Y1 #= Y2)).

/*Function that ensures that the cell Pos2 is not reachable with the rook's movement from the cell Pos1*/
not_rook_move(Pos1, Pos2, TamX, TamY) :-
    translate_position_coords(Pos1, X1, Y1, TamX, TamY),
    translate_position_coords(Pos2, X2, Y2, TamX, TamY),
    X1 #\= X2 #/\ Y1 #\= Y2.
```

```
/*Function that defines the horse's movement.*/
horse_move(Pos1, Pos2, TamX, TamY) :-
    translate_position_coords(Pos1, X1, Y1, TamX, TamY),
    translate_position_coords(Pos2, X2, Y2, TamX, TamY),
    (X1 #= X2 + 2 #/\ Y1 #= Y2 + 1) #\/
    (X1 #= X2 + 2 #/\ Y1 #= Y2 - 1) #\/
    (X1 #= X2 + 1 #/\ Y1 #= Y2 + 2) #\/
    (X1 #= X2 + 1 #/\ Y1 #= Y2 - 2) #\/
    (X1 #= X2 - 2 #/\ Y1 #= Y2 - 1) #\/
    (X1 #= X2 - 2 #/\ Y1 #= Y2 + 1) #\/
    (X1 #= X2 - 1 #/\ Y1 #= Y2 - 2) #\/
    (X1 #= X2 - 1 #/\ Y1 #= Y2 + 2).

/*Function that ensures that the cell Pos2 is not reachable with the horse's movement from the cell Pos1*/
not_horse_move(Pos1, Pos2, TamX, TamY) :-
    translate_position_coords(Pos1, X1, Y1, TamX, TamY),
    translate_position_coords(Pos2, X2, Y2, TamX, TamY),
    (X1 #\= X2+2 #/\
    X1 #\= X2-2 #/\
    X1 #\= X2+1 #/\
    X1 #\= X2-1) #\/
    (X1 #= X2 + 2 #/\ Y1 #\= Y2 + 1 #/\ Y1 #\= Y2 - 1) #\/
    (X1 #= X2 + 1 #/\ Y1 #\= Y2 + 2 #/\ Y1 #\= Y2 - 2) #\/
    (X1 #= X2 - 2 #/\ Y1 #\= Y2 - 1 #/\ Y1 #\= Y2 + 1) #\/
    (X1 #= X2 - 1 #/\ Y1 #\= Y2 - 2 #/\ Y1 #\= Y2 + 2).

/*Function that defines the king's movement.*/
king_move(Pos1, Pos2, TamX, TamY) :-
    translate_position_coords(Pos1, X1, Y1, TamX, TamY),
    translate_position_coords(Pos2, X2, Y2, TamX, TamY),
    ((X2 #= X1 + 1 #\/ X2 #= X1 - 1) #/\ (Y2 #= Y1 + 1 #\/ Y2 #= Y1 - 1)) #\/
    ((X2 #= X1 + 1 #\/ X2 #= X1 - 1) #/\ Y2 #= Y1) #\/
    (X2 #= X1 #/\ (Y2 #= Y1 + 1 #\/ Y2 #= Y1 - 1)).

/*Function that ensures that the cell Pos2 is not reachable with the king's movement from the cell Pos1*/
not_king_move(Pos1, Pos2, TamX, TamY) :-
    translate_position_coords(Pos1, X1, Y1, TamX, TamY),
    translate_position_coords(Pos2, X2, Y2, TamX, TamY),
    ((X2 #= X1 + 1 #\/ X2 #= X1 - 1) #/\ (Y2 #\= Y1 + 1 #/\ Y2 #\= Y1 - 1) #/\ Y2 #\= Y1) #\/
    X2 #> X1+1 #\/ X2 #< X1-1 #\/
    (X2 #= X1 #/\ (Y2 #\= Y1 + 1 #/\ Y2 #\= Y1 - 1)).

/*Function that defines the queen's movement.*/
queen_move(Pos1, Pos2, TamX, TamY) :-
    translate_position_coords(Pos1, X1, Y1, TamX, TamY),
    translate_position_coords(Pos2, X2, Y2, TamX, TamY),
    ((X2 #= X1 + A) #/\ (Y2 #= Y1 + A #\/ Y2 #= Y1 - A)) #\/
    (X1 #= X2 #/\ Y1 #\= Y2) #\/ (X1 #\= X2 #/\ Y1 #= Y2).

/*Function that ensures that the cell Pos2 is not reachable with the queen's movement from the cell Pos1*/
not_queen_move(Pos1, Pos2, TamX, TamY) :-
    translate_position_coords(Pos1, X1, Y1, TamX, TamY),
    translate_position_coords(Pos2, X2, Y2, TamX, TamY),
    X2 #\= X1  #/\ Y2 #\= Y1  #/\
    (X2 #= X1 + A #/\ Y2 #\= Y1 + A #/\ Y2 #\= Y1 - A).

/*----------------------END_OF_PAWNS-MOVEMENTS--------------------*/
```