Programação em Lógica
Intermediate Report

Group: Echek_3

Daniel Gazola Bradaschia : up201700494
Gustavo Speranzini Tosi Tavares : up201700129

FEUP : MIEIC
17 de Novembro de 2019

Index

1. Introduction

The following project was developed for Logic Programing for Master in Informatics and Computer Engineering. It consists in implementing a board game for two players in *Prolog.* The game of choice was Echeck, a modified version of traditional chess

2. The Echeck

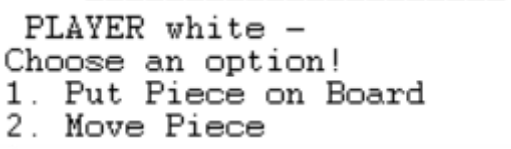
Echeck is a modern variant of the traditional chess game created by Léandre Proust on 4 May of 2019. As a result of being a variant, echeck has different rules, however the goal remains the same: to capture the enemy king while protecting your own.

2.1 The Rules

The game consists of a 4x4 board, totaling in 16 squares, played by 2 players, each with their respective 6 pieces of white or black colors. If a player is playing against a computer a choice of difficulty will be presented, hard being a normal player while an easy computer won't be allowed to use special moves.

At the beginning of the game, the kings are placed in the center of the board. The game begins by the white team. A player has 2 options in his turn:

1) Place a new piece on the board, from those who are available.
2) Move a piece already present on the board.

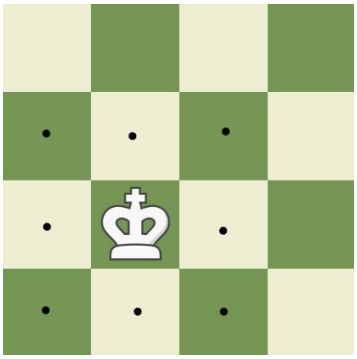


Options In a Turn.

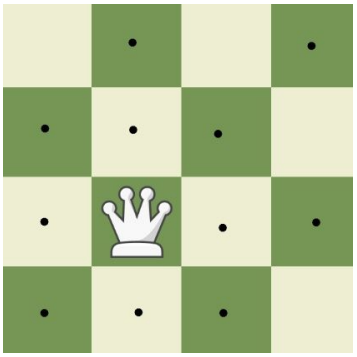
The pieces have the respective types:

- *The King*: Can move in a single space in any direction (horizontal, vertical, diagonal). If this piece is surrounded the game ends.
- *The Queen*: Can move any number of spaces in any direction (horizontal, vertical, diagonal).
- *The Tower*: Can move any number of spaces horizontally or vertically. If present on the board, this piece can switch places with the king.
- *The Bishop*: Can move any number of spaces diagonally.
- *The Horse*: Can move with the traditional “L” movement. This piece is capable of jumping over other board pieces.
- *The Pawn*: Can move a single space forward or diagonally forward.

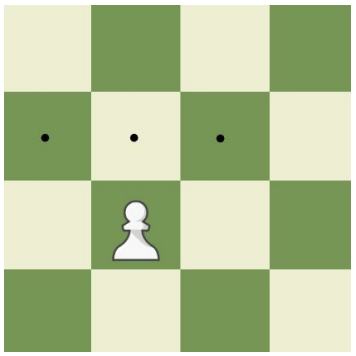
Pieces that can move any number of spaces (Queen, Bishop and Tower), can jump over any pieces. The game ends when a player has his King surrounded by all 4 cardinal sides. The following images should serve as an example of how each piece can move:



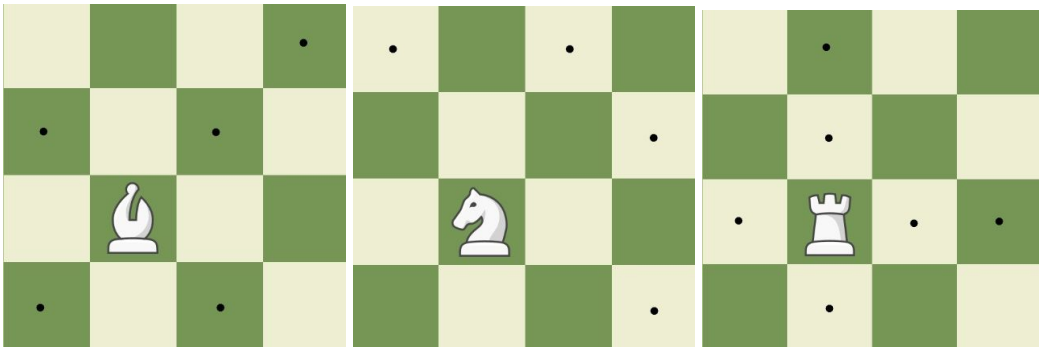
King Movements



Queen Movements



Pawn Movements



Bishop Movements Horse Movements Tower Movements

3. Game Logic

3.1 Internal Representation

To represent the possible states of the game, the approach chosen was the list of lists, in which the internal lists represents the board lines, each with 4 elements that symbolizes squares.

Each square is represented by:
{LINE, COLUMN, PIECE, PLAYER}

Line and Column are the coordinates of the square.
Player can be white, black or none.

The piece can be occupied by the following options:

- *none*: empty square
- *king:* white and black kings.
- *queen*: white and black queens.
- *tower*: white and black towers.
- *bishop*: white and black bishops.
- *horse*: white and black horses.
- *pawn*: white and black pawns.

3.2 Text Mode Preview

The following is the code that prints the board, recursively, in a user friendly way:

```
printBoard(TAB):-
        printSeparatorIndex, nl,
        printSeparatorLINE, nl,
        printMatrix(TAB, 1),
        printSeparatorLINE.

printMatrix([], 5).

printMatrix([H|T], N):-
        write('  '),
        N1 is N+1,
        write(N),
        write(' | '),
        printLINE(H),
        nl,
        printMatrix(T, N1).

printLINE([]).

printLINE([H|T]):-
        translate(H, S),
        write(S),
        write(' | '),
        printLINE(T).

printSeparatorLINE:-
        write('    ---------------------').

printSeparatorCOLumn:-
        write('|     |     |     |     |').

printSeparatorIndex:-
        write('      1    2    3    4 ').
```

Function to Display the Board

It should also be noted that a mechanism of translation was developed to turn the internal representation more user friendly as well. Each piece is identified by its first letter of the name followed by an Upper Case letter of its team.

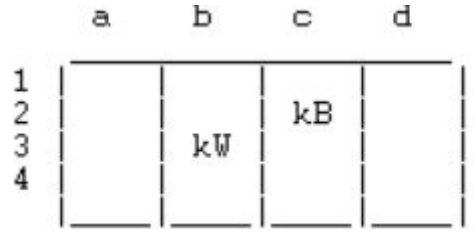
```
translate({_,_,none, none},S) :- S='  '.
translate({_,_,king, black},S) :- S='kB'.
translate({_,_,king, white},S) :- S='kW'.
translate({_,_,queen, black},S) :- S='qB'.
translate({_,_,queen, white},S) :- S='qW'.
translate({_,_,bishop, black},S) :- S='bB'.
translate({_,_,bishop, white},S) :- S='bW'.
translate({_,_,tower, black},S) :- S='tB'.
translate({_,_,tower, white},S) :- S='tW'.
translate({_,_,horse, black},S) :- S='hB'.
translate({_,_,horse, white},S) :- S='hW'.
translate({_,_,pawn, black},S) :- S='pB'.
translate({_,_,pawn, white},S) :- S='pW'.

translate(1, _, S, _, _):- S='queen'.
translate(2, _, S, _, _):- S='bishop'.
translate(3, _, S, _, _):- S='tower'.
translate(4, _, S, _, _):- S='horse'.
translate(5, _, S, _, _):- S='pawn'.
translate(6, PLAYER, _, TAB, STATE) :- display_game(TAB, PLAYER, STATE).
```

Translate Mechanism

List of Game States:

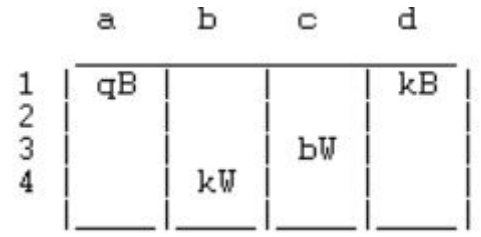
- Initial Game State:

[[{1,1,none,none}, {1,2,none,none}, {1,3,none,none}, {1,4,none,none}],
[{2,1,none,none}, {2,2,none,none}, {2,3,king,black}, {2,4,none,none}],
[{3,1,none,none}, {3,2,king,white}, {3,3,none,none}, {3,4,none,none}],
[{4,1,none,none}, {4,2,none,none}, {4,3,none,none}, {4,4,none,none}]]



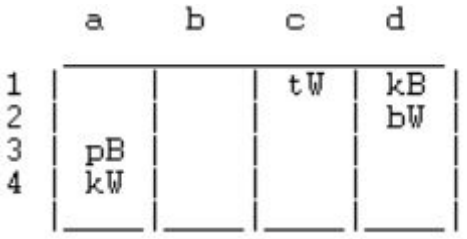
- Possible Intermediate Game State:

  [[{1,1,queen,black}, {1,2,none,none}, {1,3,none,none}, {1,4,king,black}],
  [{2,1,none,none}, {2,2,none,none}, {2,3,bishop,white}, {2,4,none,none}],
  [{3,1,none,none}, {3,2,king,white}, {3,3,none,none}, {3,4,none,none}],
  [{4,1,none,none}, {4,2,none,none}, {4,3,none,none}, {4,4,none,none}]]



- Possible End Game State:

  [[{1,1,none,none}, {1,2,none,none}, {1,3,tower,white}, {1,4,king,black}],
  [{2,1,none,none}, {2,2,none,none}, {2,3,king,black}, {2,4,bishop,white}],
  [{3,1,pawn,black}, {3,2,king,white}, {3,3,none,none}, {3,4,none,none}],
  [{4,1,king,white}, {4,2,none,none}, {4,3,none,none}, {4,4,none,none}]]



3.3 Valid Moves

The process of validating a move is made through two different instances, first we ask the user or computer to choose a piece on the board:

```
choosePiece(TAB, {LINE, COL, TYPE, PLAYER}, pvp):-
        write(' Select Piece (Row/Column): '),
        read(LINE/COL),
        verifyPosition(TAB, {LINE, COL, TYPE, PLAYER}).

choosePiece(TAB, {LINE, COL, TYPE, PLAYER}, pvp):-
        clearScreen, printBoard(TAB),
        nl, write('Invalid Position. Try again!'), nl,
        choosePiece(TAB, {LINE, COL, TYPE, PLAYER}, pvp).
```

Choose Piece

Once the input of row and column happens, *verifyPosition* will check if the desired position has a piece that belongs to the respective player. In the negative case the process repeats itself, while in positive case, we go to the movement of our piece.
If the user should wish to list every position available to move, the following function should be used:

```
valid_moves(TAB, {LINE, COL, TYPE, PLAYER}, L):-
        findall({X,Y}, move({LINE, COL, TYPE, PLAYER},{X,Y} ,TAB, _), L).
```

valid moves

3.4 Movement

We initiate the movement by calling the following:

```
movePiece(TAB, {LINE, COL, TYPE, PLAYER}, NEWTAB, pvp):-
        write(' Select Destination (Row/Column) '),
        read(LINEEND/COLEND),
        verifyEndPosition(TAB,{LINEEND,COLEND},{LINE, COL, TYPE, PLAYER},NEWTAB).
```

move Piece

While the function above reads the row and column of destination, the following function is responsible to validate and execute the move:

```
verifyEndPosition(TAB,{LINEEND,COLEND},{LINE, COL, TYPE, PLAYER},NEWTAB):-
        verifyMoveInsideBoard(LINEEND, COLEND),
        verifyNotPiece(TAB, {LINEEND, COLEND, _, _}),
        verifyMove({LINE, COL, TYPE, PLAYER}, {LINEEND,COLEND}),
        move({LINE, COL, TYPE, PLAYER}, {LINEEND,COLEND}, TAB, NEWTAB).
```

verify the destination

In *verifyEndPosition,* we will break into four different process that will validate if the move is inside the board, verify if the space is free to be occupied and verify if the movement is happening within the limitations of the selected piece, respectively. Finally, once everything is verified, the move will be executed.

3.5 End of Game

Echeck ends once a king from any team is surrounded in all of his cardinal directions.

```
game_over(TAB, black, _):-
        getKingPos({PL,PC,king,white}, TAB),
        FRONT is PL-1,
        \+verifyVictory(TAB,FRONT,PC,{PL, PC, king,white}),
        BACK is PL+1,
        \+verifyVictory(TAB,BACK,PC,{PL, PC, king,white}),
        RIGHT is PC+1,
        \+verifyVictory(TAB,PL,RIGHT,{PL, PC, king,white}),
        LEFT is PC-1,
        \+verifyVictory(TAB,PL,LEFT,{PL, PC, king,white}),
        display_game(TAB, black, win).

game_over(TAB, white, _):-
        getKingPos({PL,PC,king,black}, TAB),
        FRONT is PL-1,
        \+verifyVictory(TAB,FRONT,PC,{PL,PC,king,black}),
        BACK is PL+1,
        \+verifyVictory(TAB,BACK,PC,{PL,PC,king,black}),
        RIGHT is PC+1,
        \+verifyVictory(TAB,PL,RIGHT,{PL,PC,king,black}),
        LEFT is PC-1,
        \+verifyVictory(TAB,PL,LEFT,{PL,PC,king,black}),
        display_game(TAB, white, win).

game_over(TAB, black, STATE):-
        display_game(TAB, white, STATE).

game_over(TAB, white, STATE):-
        display_game(TAB, black, STATE).
```

As the code above shows, two functions have been created per team, in which *verifyVictory* check if the king is surrounded in his front, back, left and right. If it is, *display_game* will be called with withe a new STATE, indicating victory, else the game shall continue.

3.6 Computer Movement

The computer movement is made by generating random numbers in the functions previously described, however, should a user want to check an automated move, the function *choose_move* is available.It should be noted that two levels of difficulty have been assigned, easy being just random movement while hard allows the computer to execute special movements, such as swapping *king* and *tower.*

4. Conclusions

The development of this project was especially useful to help consolidate our knowledge in *prolog,* as well as put it into practice.

The group is overall happy with the final result, but is aware that there is room for improvement, mainly in the area of automated play, in which we wish we were able to create a bigger difference between difficulties. In the end, we believed we have achieved our goal of learning and developing in *prolog,* even among the difficulties of staying in the set parameters and lack of experience.

The biggest problem however lies in the game itself. Being such a new game and being in kickstart production meant that some rules were confusing or straight right questionable, which is why we wished we had more time to iron out some decisions made through the development.

5. Bibliografy

[1]Game:https://www.boardgamegeek.com/boardgame/274302/echek

[2]PlayBook:http://www.fabrikludik.fr/protozone/jeux/echek1540381546/regle-du-jeu-print-and-play-1540382406.pdf

[3]Images:https://www.chess.com/pt-BR/analysis