FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Mestrado Integrado em Engenharia Informática e Computação

Sunday, November 17 2019

Echek

Programação em Lógica
2019/2020

3MIEIC02_Echek_2:

Gonçalo Marantes Monteiro up201706917@fe.up.pt
Simão Santos up201504695@fe.up.pt

Index

Introduction

This project is being developed in the SICStus Prolog Development System, and is the the first of a two part project regarding the course Logic Programming (Programação em Lógica), a 3rd year course of the Integrated Masters Degree in Informatics and Computing Engineering (MIEIC). The aim of this project is to develop a board game using the programming language Prolog, based on moving pieces (possible moves) and win/defeat conditions (final states).

Game Description

History

Echek is part of a 5 board games collection called Cut and Play: Collection of free micro games created by Léandre Proust in 2019. It was first introduced on kickstarter and has over 600 backers who have pledged over €4000 to help bring this collection to life. This collection is composed of: Echek, Exo, YSNP, Ants and Supéro.

Picture 1: Cut and Play: Collection of free micro games

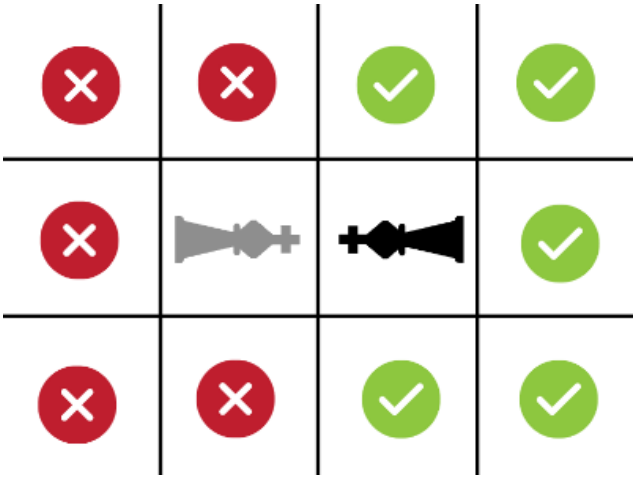
The goal of this collection was to offer games with simple rules, little material and easy to make. Since the beginning of 2019, Proust has created and published a new minimalist board game each month: Echek on February, Exo in March, YSNP in April, Ants in May and Supéro in June. 20,000 copies of these games were printed and distributed for free in France by Philibert, the French leader in board games sales.

Rules

Echek is a chess-inspired game which is played with only 12 cards and without an actual board. Each player has 6 pieces (King, Queen, Tower, Bishop, Horse, and Pawn) with chess-like moves and they play on a 4x4 dynamic board. The game begins with both Kings face to face, and each turn players can move a piece or put a piece into play. The game ends when a King is surrounded.

Placing and Moving

The player chooses a piece from his hand and places it in the playing area. When placing, there are two important things to consider. First, it must be placed next (by one side or by one corner) to one of his own pieces already on the board. And second, the piece can't be placed on one of the 4 squares adjacent (top, bottom, left and right) to the opposing King. In the picture below we see the 2 rules in play.



Picture 2: Early game moves

Regarding moving the player chooses one of his pieces already in the playing area and moves it according to the rules of this piece. After moving, all the pieces in game must always be connected by a corner or a side, otherwise the move is not allowed.

Movements and special "powers"

A Pawn can move one square horizontally or vertically. When this piece is placed in the playing area, it can move immediately if desired.

A Knight or Horse in Echek, just like the Knight in Chess, moves to a square that is two squares away horizontally and one square vertically, or two squares vertically and one square horizontally. The complete move therefore looks like the letter "L". It can jump over all other pieces to its destination square.

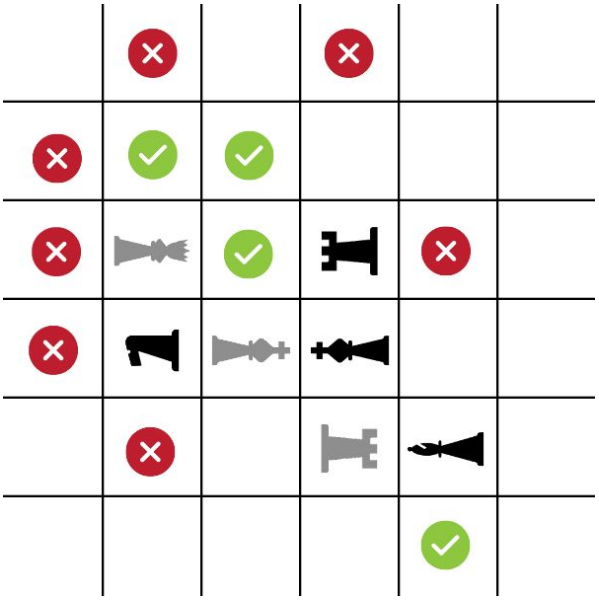
A Tower can move any number of squares horizontally or vertically. If this piece is in the playing area, only once per game, you can spend your turn to swap it with your king (exchange their positions). Similar to the rook move in Chess.

A Bishop can move any number of squares diagonally. This piece may exceed the borders of the playing area (it is not considered in the borders of the playing area).

A King can move one square in any direction (horizontally, vertically, or diagonally). When this piece is surrounded, you lose the game.

A Queen can move any number of squares in any direction (horizontally, vertically, or diagonally). When this piece is surrounded you lose it, removing it permanently from the game.

Pieces that move any number of squares (Queen, Tower or Bishop) can not jump over pieces of the opposing color, with the exception of the Horse, but they can jump over pieces of the same color to get to its destination square.



Picture 3: Possible white Queen moves

Ending a Match

The game ends as soon as a King is surrounded on all four sides (up, down, left, and right) by pieces of any color or by the border of the game area. The player whose king is surrounded loses the game.

Notes about the rules

In the rules when the board shrunk from a 4x4 to a 3x4 by having its left column removed, for example, it was not clear whether the only way to restore it to its full size (4x4) was to reinstate that missing column or, in addition to that possibility, we could add a new column on the right side, by moving a piece there.

Since we weren not getting a definitive answer on that, we decided to contact the game's creator, *Leandre Proust*. He was more that happy to help and was also very

excited we were making a “dematerialized version”, as he put it, of his game. He explained that the board could grow in any direction, after having being shrunk.

In addition, he asked if we could provide, when completed, a running version of the game. Since the game is being written in prolog and has no graphical interface, we don’t think Leandre will know what is coming.

Game Logic in Prolog

Game Representation

Mentioned previously, Echek is played on a 4x4 dynamic board, which means that the board is not stationary and it can grow or shrink depending on the positioning of the pieces. This introduces a new challenge when defining the board.

The examples below show the code of 3 boards and their representation.

Initial State

```
initialBoard([
    [empty, empty, empty],
    [empty, king-black, empty],
    [empty, king-white, empty],
    [empty, empty, empty]
]).
```

Intermediate State

```
intermediateBoard([
    [empty, empty, empty, empty, empty],
    [empty, empty, empty, bishop-black, empty],
    [empty, tower-black, king-black, tower-white, empty],
    [empty, empty, king-white, empty, empty],
    [empty, queen-white, horse-black, empty, empty],
    [empty, empty, empty, empty, empty]
]).
```

Final State

```
finalBoard([
    [empty, empty, empty, empty, empty, empty],
    [empty, empty, queen-black, empty, empty, empty],
    [empty, tower-black, king-black, tower-white, bishop-black, empty],
    [empty, empty, king-white, empty, empty, empty],
    [empty, queen-white, horse-black, horse-white, empty, empty],
    [empty, empty, empty, empty, empty, empty]
]).
```

Board Preview in Text Mode

The numbers and letters in alphabetical order represent the coordinates of the board and the uppercase letters representing the pieces are the white ones, while their lowercase counterparts are the black ones.

Pieces Representation

```
% Pieces
translate(king, 'k').
translate(queen, 'q').
translate(bishop, 'b').
translate(tower, 't').
translate(horse, 'h').
translate(pawn, 'p').
translate(empty, '.').

% Color
translate(black, 'B').
translate(white, 'W').
```

Picture 4: Game pieces representation

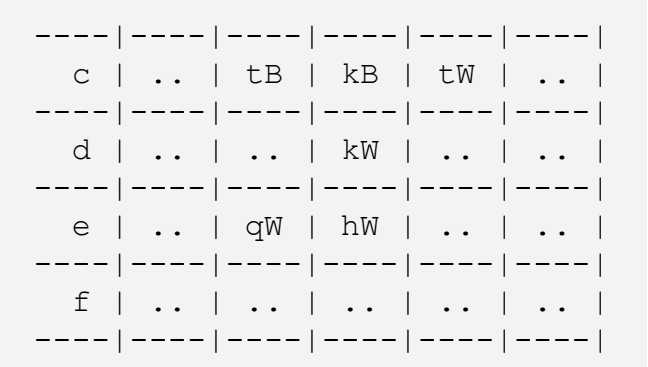
Initial State

```
    | 1  | 2  | 3  |
---|----|----|----|
 a | .. | .. | .. |
---|----|----|----|
 b | .. | kB | .. |
---|----|----|----|
 c | .. | kW | .. |
---|----|----|----|
 d | .. | .. | .. |
---|----|----|----|
```

Picture 4: Board initial state

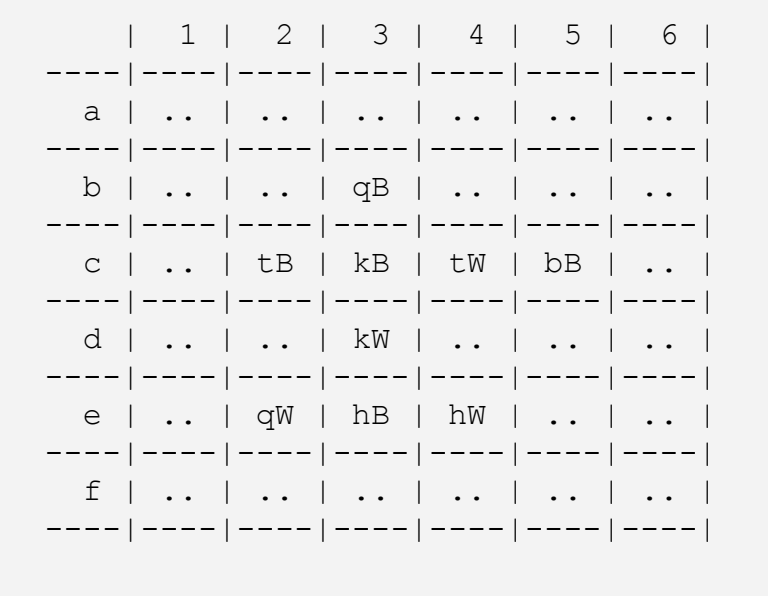
Intermediate State

```
    |  1 |  2 |  3 |  4 |  5 |
----|----|----|----|----|----|
  a | .. | .. | .. | .. | .. |
----|----|----|----|----|----|
  b | .. | .. | .. | bB | .. |
```



Picture 5: Board intermediate state

Final State



Picture 6: Board final state

List of Valid Moves

```
valid_moves([Row | Board], Piece-Color, PossibleMoves):-
    length(Row, MaxColNumber),
    length([Row | Board], MaxRowNumber),
    Goal =
    (
        between(1, MaxColNumber, ColNum),
        between(1, MaxRowNumber, RowNum),
        canMove(Board, ColNum-RowNum, Piece-Color)
    ),
    setof(ColNum-RowNum, Goal, PossibleMoves).
```

When it is our turn to play we have two options, we can either place a piece on the board or move an existing piece to a new cell. For this reason we need two predicates to get the list of possible moves, since the rules for placing and moving are different.

Essentially, we go through all the cells on the board and place or move the desired piece to its new cell and see if the final board is valid. By using *setof* we can achieve this easily.

```
canMove(Board, NewColNum-NewRowNum, Piece-Color):-
    isInsideBoard(Board, NewColNum-NewRowNum), % check if NewColNum-NewRowNum are inside
the board
    getCellCoords(Board, OldColNum-OldRowNum, Piece-Color),
    isEmptyCellCoords(Board, NewColNum-NewRowNum), % check if coord is empty
    canPieceMove(Board, Piece-Color, OldColNum-OldRowNum, NewColNum-NewRowNum), % checks
if NewCoords are achievable from OldCoords
    % detect if there are enemies in path
    replaceCell(Board, empty-empty, OldColNum-OldRowNum, EmptyBoard), % replace old cell
with empty
    replaceCell(EmptyBoard, Piece-Color, NewColNum-NewRowNum, NewBoard), % put piece in
new cell
    rearrangeBoard(NewBoard, ArrangedBoard), % rearrange board
    getCellCoords(ArrangedBoard, ArrangedColNum-ArrangedRowNum, Piece-Color),
    !,
    \+notAdjacent(ArrangedBoard, ArrangedColNum-ArrangedRowNum), % check if coord has any
adjacent pieces
    !,
    isValidBoard(ArrangedBoard). % check if final board is empty
```

Executing Plays

Since Echek is a turn-based game, the main game predicate is very simple, each player has its own individual predicate for choosing a piece, a cell and making a move.

```
gameLoop(Board, Player1, Player2):-
    whitePlayerTurn(Board, Player1, NewBoard),
    checkQueensTrapped(NewBoard, NewWhiteBoard), % check if queens need to be removed
    printBoard(NewWhiteBoard),
    (
        % game over
        (
```

```
            game_over(NewWhiteBoard, Winner), declareGameOver(Winner)
        );
        % if not, continue playing
        (
            blackPlayerTurn(NewWhiteBoard, Player2, FinalBoard),
            checkQueensTrapped(FinalBoard, FinalBlackBoard), % check if queens need to be
        removed
            printBoard(FinalBlackBoard),
            (
                % game over
                (
                    game_over(FinalBlackBoard, Winner), declareGameOver(Winner)
                );
                % if not continue playing
                (
                    gameLoop(FinalBlackBoard, Player1, Player2)
                )
            )
        )
    ).
```

```
whitePlayerTurn(Board, person, NewBoard):-
    nl, write('-------- PLAYER WHITE --------'), nl, nl,
    readGameInput(Board, Piece, NewColNum-NewRowNum),
    getCellCoords(Board, OldColNum-OldRowNum, Piece-white),
    play(Board, Piece-white, OldColNum-OldRowNum, NewColNum-NewRowNum, NewBoard),
    printBoard(NewBoard).
```

In *whitePlayerTurn* (the *blackPlayerTurn* is the same but for black pieces), the predicate *readGameInputs* (seen below) verifies the input written. (The *readRow* is the same as *readCols* but for rows)

```
readGameInput(Board, Piece, ColNum-RowNum):-
    readPiece(Piece),
    readCol(Board, ColNum),
    readRow(Board, RowNum).
```

```
% --- Pieces --- %
readPiece(Piece):-
    write('> Piece Name: '),
    read(ReadPiece),
    (
        % piece is valid
        (
            validPiece(ReadPiece),
            Piece = ReadPiece
        );
        % or try again
        (
            write('ERROR: Please enter Piece name again.'), nl,
            readPiece(Piece)
        )
    ).
```

Then, still in the *whitePlayerTurn*, we get the coordinates of the piece selected, with the predicate: *getCellCoords*, since we know the board and the piece selected. It is important to note that the selected piece may not be in the board. In that case *OldColNum-OldRowNum* in *getCellCoords* is equal to 0-0.

Then in the *move* predicate, where the real stuff happens, with the Board given, we want to move the Piece from *OldColNum-OldRowNum* to *NewColNum-NewRowNum*, thus creating a *NewBoard*.

We implemented a kind of loop, where to move on, the player must automatically insert a valid input. If the move is not valid, the *play* predicate is called again, after (again) receiving the coordinates to where the player wishes to move. When the player finally inserts

```
% movement
move(Board, Piece-Color, OldColNum-OldRowNum, NewColNum-NewRowNum, NewBoard):-
    getPossibleMoves(Board, Piece-Color, PossibleMoves),
    (
        % either move is valid
        (
            member(NewColNum-NewRowNum, PossibleMoves),
            movePiece(Board, Piece-Color, OldColNum-OldRowNum, NewColNum-NewRowNum,
        NewBoard)
        );
        % or try again
        (
            write('ERROR: Invalid move'), nl,
            readGameInput(Board, NewPiece, NewNewColNum-NewNewRowNum),
            getCellCoords(Board, NewOldColNum-NewOldRowNum, NewPiece-Color),
            move(Board, NewPiece-Color, NewOldColNum-NewOldRowNum,
        NewNewColNum-NewNewRowNum, NewBoard)
        )
    ).
```

End of a Game

When verifying the end of a game, we check if the king is surrounded by pieces or the board's border, as explained previously in the rules.

```
isTrapped([Row | Board], ColNum-RowNum):-
    length(Row, NumCols), length([Row | Board], NumRows),
    ColNum > 1, RowNum > 1,
    ColNum < NumCols, RowNum < NumRows,
    RowNumUpper is RowNum - 1,  RowNumLower is RowNum + 1,
    ColNumLeft is ColNum - 1,  ColNumRight is ColNum + 1,

    \+isEmptyCellCoords([Row | Board], ColNum-RowNumUpper), % check if upper coord is
        empty
    \+isEmptyCellCoords([Row | Board], ColNum-RowNumLower), % check if lower coord is
        empty
    \+isEmptyCellCoords([Row | Board], ColNumLeft-RowNum), % check if left coord is empty
    \+isEmptyCellCoords([Row | Board], ColNumRight-RowNum). % check if right coord is
        empty
```

We check if the upper, lower, left and right cell are not empty, when the king is not either in the corners or the sides. If they are not empty, it means the king is surrounded which in turn means game over for the player whose king is surrounded.

Checking the Board

To see if a board is valid, there are several rules (as explained in the game description section) which we will go through.

Firstly, every piece must be touching one another, either side by side or by a corner, we take care of this rule while placing and moving pieces, so in the end all pieces are connected.

Secondly, the maximum size of the board is 4x4, which means the actual maximum size of our board is 6x6 since we have to account for the borders.

```
% all the pieces are touching one another if the board has no empty rows or cols
besides the borders
isValidBoard([Row | Board]):-
    % test if rows are empty
    length([Row | Board], RowNum), ActualRowNum is RowNum - 1,
    isValidBoardRows([Row | Board], ActualRowNum),
    % test if cols are empty
    length(Row, ColNum), ActualColNum is ColNum - 1,
    isValidBoardCols([Row | Board], ActualColNum),
    % check board size
    between(1, 6, RowNum),
    between(1, 6, ColNum).
```

Since the board is completely dynamic and evolves with the players' moves, everytime a move is made we need arrange the board in a way that there are always empty borders. This way, the players always have cells to make moves.

```
rearrangeBoard(Board, NewBoard):-
    checkTop(Board, TopBoard), % add or remove top row
    checkBot(TopBoard, BotBoard), % add or remove bot row
    checkLeft(BotBoard, LeftBoard), % add or remove left row
    checkRight(LeftBoard, NewBoard). % add or remove right row
```

Computer Moves

Essentially the computer decision making algorithm is divided into two parts, choosing a piece and choosing where to put it. This is valid for both placing and moving.

```
choose_move(Board, Level, Piece-Color, OldColNum-OldRowNum, NewColNum-NewRowNum):-
   pickPiece(Board, Level, Piece-Color, OldColNum-OldRowNum),
   pickMove(Board, Level, Piece-Color, OldColNum-OldRowNum, NewColNum-NewRowNum).

pickPiece(Board, 1, Piece-Color, OldColNum-OldRowNum):-
   pieceList(PieceList),
   random_member(Piece, PieceList),
   getCellCoords(Board, OldColNum-OldRowNum, Piece-Color).

pickMove(Board, 1, Piece-Color, OldColNum-OldRowNum, NewColNum-NewRowNum):-
   valid_places(Board, Piece-Color, PossibleMoves),
   random_member(NewColNum-NewRowNum, PossibleMoves).
```

The computer chooses a random piece from the piece list and then chooses a random valid move.

Unfortunately we were not able to finish level 2 of the decision making algorithm for the computer controlled player, so the decision is always random.

Conclusion

With this project we had to redefine our notion of programming, since we were dealing with a different mindset. All we've known is imperative programming so adapting our problem solving skills to the declarative point of view was challenging. Not only did we have to deal with slow development, with our planning with prolog, we were dealt an unlucky hand with the theme of the work.

Echek is not as simple to implement as we initially thought and the rules weren't all that explicit, although it was not our first choice, we believed it would be a bit more simple, since it was based on chess. Although we had our struggles, it was a great opportunity to learn a different paradigm while doing something entertaining.

In the end there were a couple of things we weren't able to finish due to time constraints: the level 2 of the computer algorithm, which we mentioned above, and the special rule of the pawn, bishop and tower. The pawn's ability to immediately move when placed, the tower's rook-like switch with the king and how the bishop can move out of the border.

Bibliography

The site where we got the information about the rules and history. Searching by Echek in the page, you'll find the game - https://www.kickstarter.com/projects/924686715/cut-and-play-collection-of-free-micro-games

The first draft of the rules, written in english - https://www.fabrikludik.fr/jeuxGratuit/game/Echek-vo.pdf

The final version of the rules, written in french - http://www.fabrikludik.fr/jeuxGratuit/echekRegle.pdf

The email of the creator, Leandre Proust, which we contacted to enlighten us on some rules - proust.leandre@gmail.com