Quixo Game

Written in Prolog

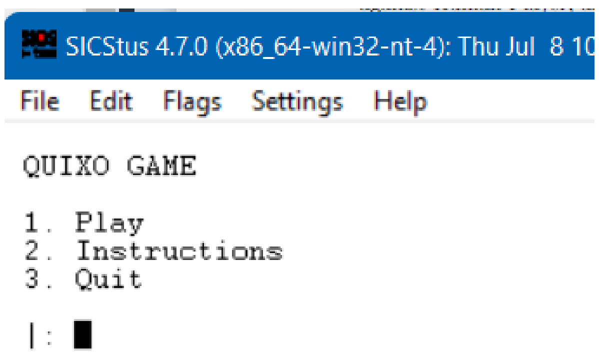
Turma 9, Grupo 7

Rui Pedro Mendes Moreira (up201906355)
Sérgio Rodrigues da Gama (up201906690)

22 de Janeiro 2022

1 Installation and Execution

In order to start our game, the only thing needed to setup is ***SICStus Prolog compiler***. Then the user only has to consult (import and compile) our ***main.pl*** file, using the same program stated before. Then, to start the game, it is needed to call the predicate **play/0**, which will start the execution, by displaying the main menu, and asking for the user to input an option.



```
QUIXO GAME

1. Play
2. Instructions
3. Quit

|:
```

Fig 1 - Main Menu Display

2 Game Description

Quixo is a board game similar to Tic-Tac-Toe, and it is actually based on that same game. However, instead of being played in a 3 by 3 board, this game has a 5 by 5 grid and the rules of the game are also quite different.

The players have to take a cube that is blank or is bearing his symbol (typically a cross or a nought) from the outer ring of the board. Then if it his blank rotates it to show his symbol and adds it to the grid by pushing it into one of the rows from which it was removed. After some plays, the pieces slowly go from blank to the players symbols.

When one of the players reach an entire row, column or diagonal with their symbol, that same player wins the game and the gameplay ends.

3 Game Logic

3.1 Internal representation of the game

In order to represent the state of the game, we went for a 2D Array. Storing each board element, and to differentiate each player, player1 is represented by **X** and player2 is represented by **O**.

In the first instance, the Board starts with the rows empty. The user has the possibility to choose between 4 options. Human Player against Human Player, Human Player against Computer, Computer against Human Player, and Computer against Computer.

The board is represented in the following structure:

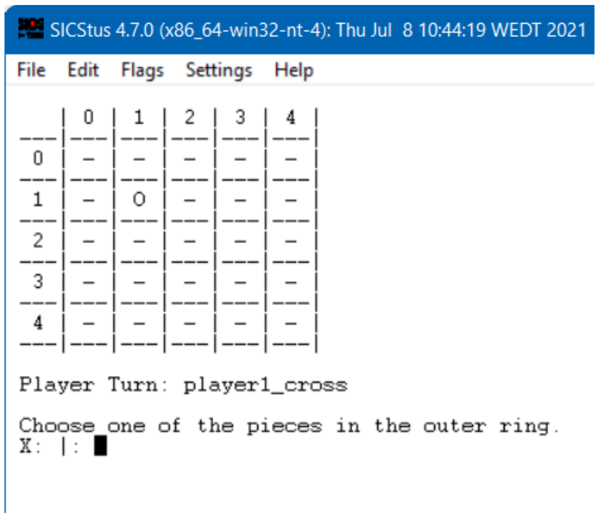
```
initial_board([
    [empty,empty,empty,empty,empty],
    [empty,nought,empty,empty,empty],
    [empty,empty,empty,empty,empty],
    [empty,empty,empty,empty,empty],
    [empty,empty,empty,empty,empty]
    ]).
```

3.2 Visualization of the game state

Initially, we provide the User a Main menu, where he is able to choose the type of game he wants to play, as described in the section above. We, also ask the user to choose the AI difficulty, but since only the random bot has been implemented, the only difficulty working is the **Easy Level**.

- **symbol()**. Converts the internal representation of the player Symbol, to allow the user interface to display the correct game symbol.
- **white_char(+Char, +Padding)**. Prints a white character to the console, with an associated padding.
- **print_symbol(+String,+Padding)**. Prints the player symbol in the board with a padding associated to it.
- **print_header()**. Prints to the console, the board header with the column indexes.
- **print_row(+Row)**. Prints to the console, the rows in the board, with the row index.
- **print_board(+Board, +RowIndex)**. Displays the board by calling **print_row(+Row)** predicate for each row of the boar.
- **display_game**. Calls **print_header()**, **print_board(+Board, +RowIndex)** to the display the current board GameState.

Additionally, we show the User the Current GameState, by showing the current player turn and the current board state using the predicates **turn_player(+Player)** and **display_game()**.


SICStus 4.7.0 (x86_64-win32-nt-4): Thu Jul 8 10:44:19 WEDT 2021
File Edit Flags Settings Help

```
    | 0 | 1 | 2 | 3 | 4 |
----|---|---|---|---|---|
 0  | - | - | - | - | - |
----|---|---|---|---|---|
 1  | - | O | - | - | - |
----|---|---|---|---|---|
 2  | - | - | - | - | - |
----|---|---|---|---|---|
 3  | - | - | - | - | - |
----|---|---|---|---|---|
 4  | - | - | - | - | - |
----|---|---|---|---|---|

Player Turn: player1_cross

Choose one of the pieces in the outer ring.
X: |:
```

Fig 2 - Board and Turn Display (Human vs Human game mode)

3.3 Execution of Plays

- Given that our Board is stored in a 2D array, we decided that the plays done by the Users would be represented by two numbers, respectively representing the column and row index of the spot or board piece he wants to move.
- In order to get the next move, we use the predicate **make_play(+GameMode, +Player, +Board, -NewBoard)**, which can have 2 outcomes.
  1. If the player is Human, we read and validate the move.
  2. If the Player is AI, we choose a random move that is valid.
- To validate the move in our Game, we use the predicate **is_valid_move(+X, +Y, +MoveX, +MoveY)**, that evaluates the move inputted, or chosen randomly, and checks if the move is valid according to the game rules definition. To do so, we first create a predicate **is_valid_play(+X, +Y, +Board, +Player)** that verifies the piece chosen from the borders of the board. If the player can choose that board piece, we call the next predicate, to validate the move.
- After having the move validated we use the predicate **make_move(+X, +Y, +MoveX, +MoveY, +Board, +Player, -NewBoard)** to apply it, and to update the board, updating the game state.

3.4 Game Final

The end of the game is reached when a player has 5 pieces in a line, whether in row, column or a diagonal. In order to check that, we use the predicate **game_over(+Board, +Player1, +Player2)** which will iterate through the Board, and check whether or not the player has 5 pieces in a line, by using the predicate **win(+Board, +Player)**.

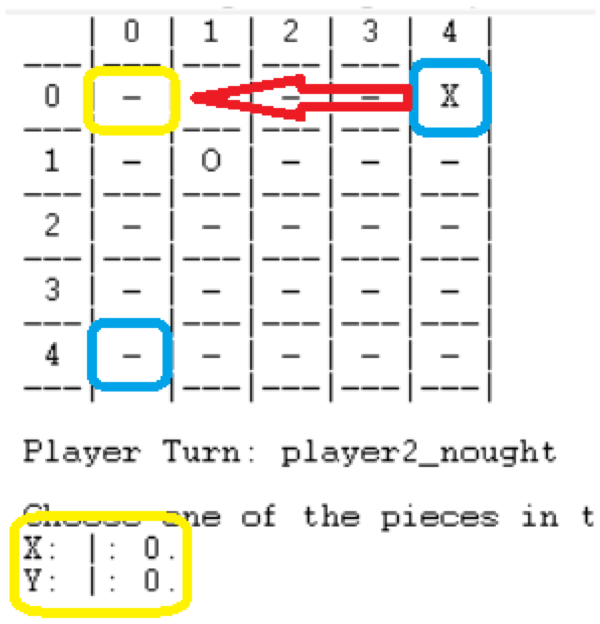
3.5 Valid Plays

In order to validate the plays that are made in the game we separate them into two types. The plays on the grid edges and the rest of the plays.

3.5.1 Edge Cases

In this cases the plays are done from the coordinates (0, 0), (4, 0), (0, 4) and (4, 4). So they are only in two rows simultaneously, so there are also only two possible moves from this positions.

To illustrate this type of play better, we will analyze the play that is being made in the *Fig 3.* In this example, the player chose the cube in (0, 0). Therefore, he has 2 possible moves, one where the cube goes to the right (4, 0) and another where the cube goes to the bottom (0, 4). When the cube goes to the right, the cube that was already there has to switch places from position (4, 0) to the position (3, 0). However, if the player chose to move to the bottom, no cube as to switch places, because, that spot (0,4) is empty.



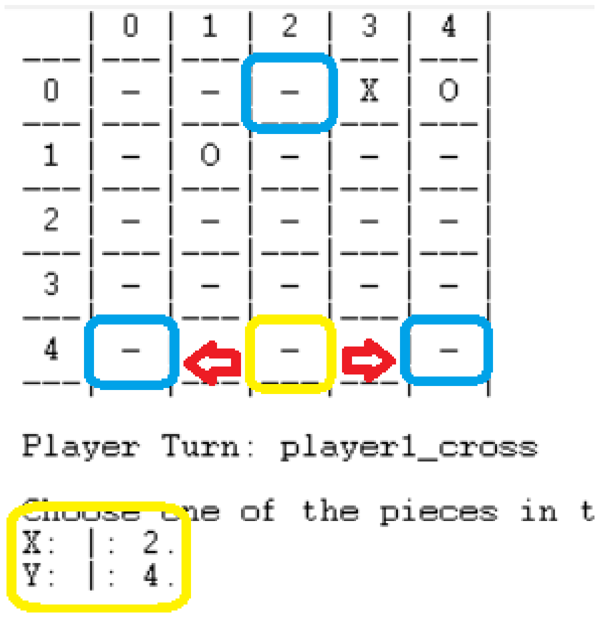
```
   | 0 | 1 | 2 | 3 | 4 |
---|---|---|---|---|---|
 0 | - | - | - | - | X |
---|---|---|---|---|---|
 1 | - | O | - | - | - |
---|---|---|---|---|---|
 2 | - | - | - | - | - |
---|---|---|---|---|---|
 3 | - | - | - | - | - |
---|---|---|---|---|---|
 4 | - | - | - | - | - |
---|---|---|---|---|---|

Player Turn: player2_nought

Choose one of the pieces in t
X: |: 0.
Y: |: 0.
```

Fig 3 - Edge Case Play

3.5.2 The other Cases

In this cases the plays are done from the rest of the coordinates, from the outer ring of the board. Therefore, they always share 3 rows, making it possible to move the cubes to 3 different spots.

Like in the previous topic, to illustrate this type of play better, we will analyze the play that is being made in the *Fig 4*. In this example, the player chose the cube in (2, 4). Therefore, he has 3 possible moves, one where the cube goes to the right (4, 4), another where the cube goes to the top (2, 0), and another where the cube goes to the left (0, 4). Like in the edge cases, when the moves to a spot that is already taken, those cubes have to switch places to the opposite direction from where the cube as come.



```
    | 0 | 1 | 2 | 3 | 4 |
 ---|---|---|---|---|---|
  0 | - | - | - | X | O |
 ---|---|---|---|---|---|
  1 | - | O | - | - | - |
 ---|---|---|---|---|---|
  2 | - | - | - | - | - |
 ---|---|---|---|---|---|
  3 | - | - | - | - | - |
 ---|---|---|---|---|---|
  4 | - | - | - | - | - |
 ---|---|---|---|---|---|

Player Turn: player1_cross

Choose one of the pieces in t
X: |: 2.
Y: |: 4.
```

Fig 4 - Not Edge Case Play

3.6 Computer Move

To execute a computer move, we use the predicate **pc_move(_, _, _, _, +Board, +Player, -NewBoard)**. The computer randomly chooses one of the valid moves that we defined in the predicate **valid_pc_moves(+X, +Y, +MoveX, +MoveY)**.

4 Conclusion

The main goal of this project was to apply the Logical Programming methodology using the language Prolog.

Since we are used to different paradigms, this project challenged us to re-think on the problems we had and how to solve them.

A next step to improve the game would be to include a natural language parsing to get user inputs. For example, specifying a move from (0, 2) to (0, 4) could be done by writing move right 2, and to include a bot using either a greedy algorithm that evaluated the weight of the moves, or implement a minimax algorithm.

In conclusion, the main goal of the project was achieved and by doing it we improved our knowledge in Logical Programming and Prolog.

5 Participation

The following table is relative to the group distribution of work and each individuals effort.

Sérgio da Gama	50%
Rui Moreira	50%

References

[1] Quixo https://boardgamegeek.com/boardgame/3190/quixo