GoRoGo

Final Report

Master in Informatics and Computing Engineering

Logic Programming

Grupo GoRoGo_3:
David Alexandre Gomes Reis - up201607927

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

November 12, 2017

Summary

GoRoGo is a two player game based on the old Chinese classic game Go, played on a 5x5 board with 25 pieces total. The main challenges of this project were to fully implement the game's rules and to flexibly allow both human and AI players with different difficulty levels.

We chose to represent the board using a two-dimensional array and different numbers for the different pieces (0- Empty 1- White 2- Black 3- Henge). This choice allowed us to easily implement complex game mechanics, such as detecting invalid moves and finding the best possible moves in a certain situation.

We developed 3 levels of difficulty for the AI. The first one (easiest) plays random moves. The second one is able to protect his own pieces, stopping the opponent from eating them. The third one (hardest) is not only able to protect his own pieces but will also eat the opponents pieces when it has the chance to do so.

Any combination of players is supported (Eg. AI 1 vs Player, AI 2 vs AI 3, Player vs Player, etc.)

Another challenge we faced was the end game/win detection, which will be described in detail in the next sections.

Contents

1 Introduction

The main objective of this project was to use the Prolog programming language and its features to develop a board game with a complex set of rules and an AI.

This report is divided into a two sections, the first of which explains some history about the game, as well as its the rules. The second section is about our implementation of the game in Prolog, and is divided into a few subsections which explain in further detail the most vital parts of the game. The third section talks about how we receive and parse user input. The fourth and last section draws a few conclusions from the whole project.

2 The Game: GoRoGo

GoRoGo is based on the popular chinese game Go, it uses the same tactics but the strategy used to win the game is different. The board is just 5x5 compared to the 19x19 board in Go, and there's the new neutral "Henge" pieces.

GoRoGo is a new version of Go, it was created in 2016 and it was designed to be a less intimidating version for beginners, but it can be an interesting variant even for experienced players. The game was tested at the Tokyo Game Market in December of 2016 and it received outstanding reviews and a high demand. [1]

The original game of Go is one of the oldest games known to exist, supposedly it was created by the chinese emperor Yao (2356-2255 BC) to teach his son discipline, concentration and balance.

GoRoGo is played on a 5x5 grid board. There are 3 different types of pieces, the white (10 pieces, played by one player), the black (10 pieces, played by the other player), and the "henge" pieces, these 5 pieces are both white and black, they exchange color depending on whose turn to play it is, they are white when White is playing and black when Black is playing. Each piece is played one time and cannot be moved (except when it is won by the opponent).

The game is started by the White player who has to place a henge piece in any position on the board, the remaining 4 henge pieces are separated by the two players, 2 for each one. These pieces cannot be the player's final play and must be played before.

Black then plays and the player's turn then alternates between Black and White. For a piece or group of pieces not to be captured, they need to have at least one free intersection linked by paths. If there's no free intersection adjacent to the piece or group of pieces and they are surrounded by the opponents pieces, they are captured by the opponent.

You can place your pieces in any intersection you want except if that intersection is already surrounded by the opponent's pieces, this is only allowed if that play leads to one or more of the opponent's pieces to be captured. However, henge pieces can be placed anywhere in the board when played. Figures 1, 2 and 3 show examples of these situations.

If a player has no possible plays, he loses the game. The game ends when all the 25 pieces have been played, and the player who has captured more of the opponent's pieces wins. If both players have captured the same number of the opponent's pieces, the White player wins. [2]

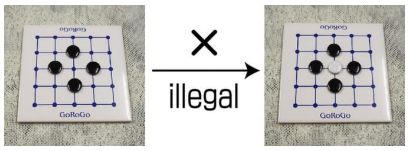

Figure 1: Illegal to place a piece surrounded by enemies

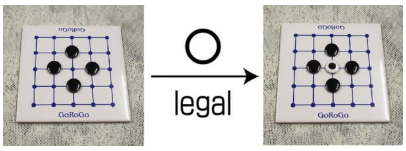

Figure 2: Legal to place a henge piece surrounded by enemies

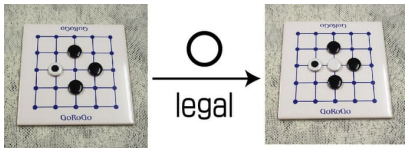

Figure 3: Legal to place a white piece if surrounded by enemies and a henge piece

3 Game Logic

3.1 Game State Data Structure

Given the fact that GoRoGo is a board game and Prolog has very good support for lists, we decided to implement the game's board using multiple lists within a list. Each list represents a line on the board. This translates to a list containing 5 lists, each containing 5 different elements, which represent the current piece that currently sits on that position. Since there are 3 pieces on the board, we decided to assign them numbers. 0 for an empty position, 1 for whites, 2 for blacks and 3 for the mixed "henge" pieces. The game starts with an empty board. The rule in Figure 4 could be used to initialize an empty board. Figure 5 shows a possible mid-game state and Figure 6 shows a possible state where the game has ended.

```
initBoard([[0,0,0,0,0],
           [0,0,0,0,0],
           [0,0,0,0,0],
           [0,0,0,0,0],
           [0,0,0,0,0]]).
```

Figure 4: Rule to initialize the board

```
[[1,0,1,0,0],
 [0,1,2,2,0],
 [2,0,3,0,0],
 [0,1,0,0,0],
 [0,3,1,0,0]]
```

Figure 5: Mid-game state

```
[[1,0,0,0,0],
 [0,1,3,0,0],
 [0,0,0,0,1],
 [0,1,0,0,0],
 [0,0,1,3,0]]
```

Figure 6: End-game state

3.2 Game State Text Display

In order to provide a good experience to the player, we chose to display the current game state by printing the Game State line by line, while replacing the numbers that represent each piece (0,1,2 and 3) for a letter or a space. We also print a grid between all the positions in order to provide better separation, especially between empty positions. Figure 7 is our current implementation of the board printing. Figures 8, 9 and 10 are the output of printing the example

game states provided in the end of the previous section, by changing the rule *initBoard* to the desired game state.

```
printBoardLine([]) :-
        print('|').
printBoardLine([0|R]) :-
        print('|'),
        print(' '),
        printBoardLine(R).
printBoardLine([1|R]) :-
        print('|'),
        print('W'),
        printBoardLine(R).
printBoardLine([2|R]) :-
        print('|'),
        print('B'),
        printBoardLine(R).
printBoardLine([3|R]) :-
        print('|'),
        print('M'),
        printBoardLine(R).

printBoard([]) :- print('-----------').
printBoard([L|R]) :-
        print('-----------'),
        print('\n'),
        printBoardLine(L),
        print('\n'),
        printBoard(R).
```

Figure 7: Predicate to print the board

```
| ?- initBoard(K), printBoard(K).
-----------
| | | | | |
-----------
| | | | | |
-----------
| | | | | |
-----------
| | | | | |
-----------
| | | | | |
-----------
K = [[0,0,0,0,0],[0,0,0,0,0],[0,0,0,0,0],[0,0,0,0,0],[0,0,0,0,0]] ?
yes
```

Figure 8: Output of printing the initial board

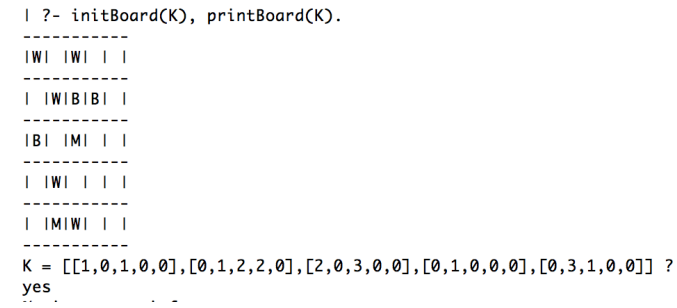

Figure 9: Output of printing the example mid-game board

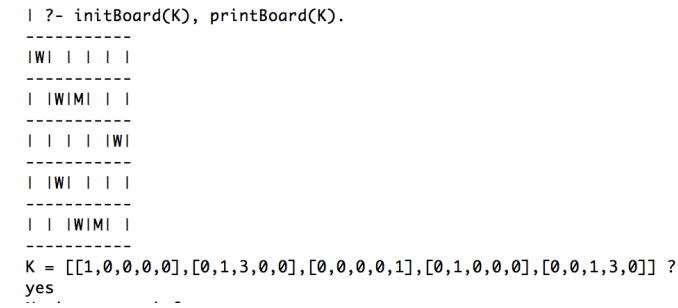

Figure 10: Output of printing the example end-game board

3.3 Obtaining Possible Moves

For a move to be valid in GoRoGo, the spot where we want to place the piece must be empty and it mustn't be surrounded by enemy pieces. For the purpose of verifying if a certain place of the board is empty we developed the predicate getPos(Line,Column,Board,Piece), and by giving it the Line, the Column, the Board and 0 as the piece (empty space) we get true if the spot is empty and false if it is already occupied. We also implemented the predicate surrounded(Line,Column,Board,CurrentPlayer,Piece) which checks if a certain piece on the board would be surrounded. The CurrentPlayer and the Piece are necessary since the Henge pieces change their color on every turn, so a piece can be surrounded in one player's turn and not surrounded in the other's.

Using these predicates, we implemented validMove(Line, Column, Line1, Column1, Board, CurrentPlayer, Piece) which recursively traverses the whole board until it can find a validMove. This predicate can be used both to get a valid move and to check if a certain move is valid.

3.4 Move Execution

Once the player or the AI has chosen a move, the move is verified using the predicates described in the previous section and, if valid, it is applied to the board. For that purpose, we implemented the predicate movePlayer(Line, Column, OldBoard, NewBoard, Piece) which replaces the empty space in the OldBoard by the Piece in the NewBoard.

Every time a move is made, we need to check if any piece has been eaten

or if the game has ended. To check for eaten pieces we implemented the predicate removeEaten(Line, Column, OldBoard, NewBoard, CurrentPlayer) which recursively traverses the whole board and uses the surrounded(Line, Column, Board, CurrentPlayer, Piece) predicate, described in a previous section, to determine if a piece must be removed or not. Using the predicate removePiece(Line, Column, OldBoard, NewBoard) which removes a piece in a given Line and Column, it removes all the pieces that are surrounded by enemy/henge pieces. The NewBoard is the board for the next turn.

We also keep track of how many pieces a player has using the predicate countPieces(CurrentPlayer, Piece, WPieces, WMixed, BPieces BMixed, WPieces1, WMixed1, BPieces1, BMixed1), since we need this information to determine the end of the game. WPieces are the White pieces, WMixed are the Henge pieces that belong to the White player, and similarly BPieces are the Black pieces and BMixed are the Henge pieces that belong to the Black player.

To check if the game has ended, we developed a few predicates that will be explained in the next section.

3.5 End Game Check

As stated in the previous section, we must verify if the game has ended after every move. The game ends when both players run out of pieces or when a player has nowhere to put his piece. There is also a rule that states that a player loses if his last piece is a Henge piece. To check all of these rules, we implemented a predicate endGame(Board, WPieces, WMixed, BPieces, BMixed, Count) that is True if the game has ended and False if it hasn't. It also prints on the screen who the winner was, and why the game ended. WPieces, WMixed, BPieces and BMixed are the number of pieces that each player has, similar to the countPieces predicate described in the previous section.

If WPieces = WMixed = BPieces = BMixed = 0 it means that there are no more pieces to play so the game must end. The predicate endGame traverses the board to check how many pieces each player has on the board, using Count. Count is a variable that is matched to 0 in the first endGame call, and is incremented every time a white piece is found, and decremented every time a black piece is found. If the final value of Count is 0 or above, white wins, otherwise, black wins.

Another possible end game situation is when there are no valid moves for a player. This is checked using the validMoves predicate described in a previous section. If that predicate is false, then the player loses.

The last possible end game situation is when the last piece of a player is a Henge piece. This is easily checked using the variables WPieces, WMixed, BPieces, and BMixed. If WPieces unifies to 0 and WMixed unifies to 1, then White loses. The same logic is applied to the Black pieces.

3.6 AI Move Choice

The AI has 3 levels of difficulty. The easiest level plays random, valid moves. The medium level is able to defend its pieces when they are a move away from being eaten. The hardest level is able to eat opponent pieces, and defend its own. This means that every level of difficulty falls back to the previous level if a move is not available. For example, on the hardest level, if there is no opponent piece that could be eaten i n that move, then it falls back to the medium level of difficulty and tries to defend its own pieces. If there are no pieces at risk of

being eaten in the next turn, then it falls back to the previous level of difficulty, playing a random move.

To achieve this mechanic, we implemented the predicate playerInput(PlayerType, CurrentPlayer, Board, Line, Column, Piece, WPieces, WMixed, BPieces, BMixed). PlayerType is 0 if it is a Human player, 1 if it is the easiest AI, 2 if it is the medium AI, and 3 if it is the hardest AI.

On the hardest level, this predicate, called with PlayerType unified to 3, uses the predicate goodMove(0, 0, Line, Column, Board, CurrentPlayer) to retrieve a move that would eat an opponent piece, which would be unified with Line and Column. If there are no such moves available, it falls back to playerInput(2, CurrentPlayer, Board, Line, Column, Piece, WPieces, WMixed, BPieces, BMixed), which is the medium level AI.

The medium level AI uses once more the predicate goodMove(0, 0, Line, Column, Board, CurrentPlayer), but this time it unifies CurrentPlayer with the opponent player number, in order to obtain the positions of the board where the opponent could eat a piece. In short, it tries to find the possible "good moves" of the opponent and if it finds any, it places a piece on that same spot to prevent the opponent's move next turn. If there are no such moves available, it falls back to playerInput(1, CurrentPlayer, Board, Line, Column, Piece, WPieces, WMixed, BPieces, BMixed), which is the easiest level of AI.

The easiest level of AI simply plays a random move, randomizing the type of piece and the spot where it's placed until it finds a valid spot. There are always valid spots when this predicate runs, since the end game predicate checks for that after every move.

4 User Text Interface

To start the game, the user simply needs to invoke the predicate play(PlayerWhiteType, PlayerBlackType) where PlayerWhiteType is the type of the white player and PlayerBlackType is the type of the black player (0 for Human, 1 for Easy AI, 2 for Medium AI, 3 for Hard AI). After that, the game starts. To choose the next move, the predicate playerInput(PlayerType, CurrentPlayer, Board, Line, Column, Piece, WPieces, WMixed, BPieces, BMixed) is called, where PlayerType is the type of player specified in the beginning of the game through PlayerWhiteType and PlayerBlackType. This means that the same predicate is used both for the human and the AI players.

If PlayerType unifies to 0, it means that the CurrentPlayer is a human. In that case, the program first asks (through the read() and write() predicates) what the type of piece that player wants to use. If the player still has more than one of those pieces available (which is checked using the variables WPieces, WMixed, BPieces, BMixed), then the program moves on, otherwise it keeps asking for a type of piece until the player gives an available one.

Next, the program asks the player the Line and the Column where the player wants to place the chosen piece. Once again, we repeat this input sequence until the player gives a valid Line and Column according to the rules of the game.

Piece, Line and Column are unified with the players choice, passing that information to the main game loop.

5 Conclusions

With this project we were able to take advantage of Prolog's capabilities to implement complex rules and game logic that would have been less efficient and require a lot more code with most other imperative languages.

We believe we were able to meet all the requirements that were proposed to us with this project, but a few details could still be further improved, such as the performance of the AI when deciding a move (sometimes takes over a second). Adding another level of AI between the easiest and medium that have already been described would also be a good improvement, since we feel there is a big difficulty gap between those two types of AI.

Bibliography

[1] Kickstarter: GoRoGo - An artisanal variation on the classic game of GO, https://www.kickstarter.com/projects/1287555371/gorogo-a-variation-on-the-classic-game-of-go

[2] Logygames: GoRoGo the new game of GO, http://www.logygames.com/english/GoRoGo.html

A Prolog Code

```
initBoard([[0,0,0,0,0],
           [0,0,0,0,0],
           [0,0,0,0,0],
           [0,0,0,0,0],
           [0,0,0,0,0]]).

/*Checks if there are still enough pieces for a certain combination of
Player/Piece*/
pieceAllowed(1,1,WPieces,_,_,_,Est) :-
        \+ initBoard(Est), /*White needs to start with henge*/
        WPieces > 0.
pieceAllowed(1,3,_,WMixed,_,_,_) :-
        WMixed > 0.

pieceAllowed(2,2,_,_,BPieces,_,_) :-
        BPieces > 0.
pieceAllowed(2,3,_,_,_,BMixed,_) :-
        BMixed > 0.

/*Checks if a certain coordinate is inside the board*/
insideTable(Value):-
        Value > -1,
        Value < 5.

/* Player controlled move decision */
playerInput(0,PlayerNo,Est,Linha,Coluna,Tipo,WPieces,WMixed,BPieces,BMixed) :-
        repeat,
                print('Type of piece: '),
                read(Tipo),
                number(Tipo),
                pieceAllowed(PlayerNo,Tipo,WPieces,WMixed,BPieces,BMixed,Est), !,
        repeat,
                print('Line: '),
                read(Linha),
                number(Linha),
                Linha > -1,
                Linha < 5,
                print('Column: '),
                read(Coluna),
                number(Coluna),
                Coluna > -1,
                Coluna < 5,
                getPos(Linha,Coluna,Est,0),
                \+ surrounded(Linha,Coluna,Est,PlayerNo,Tipo), !.

/* CPU controlled move decision, used if good move available */
playerInput(3,PlayerNo,Est,Linha,Coluna,Tipo,WPieces,WMixed,BPieces,BMixed) :-
        goodMove(0,0,Linha,Coluna,Est,PlayerNo),
        repeat,
                random(1,4,Tipo),
```

```
                pieceAllowed(PlayerNo,Tipo,WPieces,WMixed,BPieces,BMixed,Est), !.

playerInput(3,PlayerNo,Est,Linha,Coluna,Tipo,WPieces,WMixed,BPieces,BMixed) :-
        !, playerInput(2,PlayerNo,Est,Linha,Coluna,
         Tipo,WPieces,WMixed,BPieces,BMixed).

/* CPU controlled move decision, used by player 1 to protect his pieces*/
playerInput(2,1,Est,Linha,Coluna,Tipo,WPieces,WMixed,BPieces,BMixed) :-
        goodMove(0,0,Linha,Coluna,Est,2),
        repeat,
                random(1,4,Tipo),
                pieceAllowed(1,Tipo,WPieces,WMixed,BPieces,BMixed,Est), !.

/* CPU controlled move decision, used by player 2 to protect his pieces*/
playerInput(2,2,Est,Linha,Coluna,Tipo,WPieces,WMixed,BPieces,BMixed) :-
        goodMove(0,0,Linha,Coluna,Est,1),
        repeat,
                random(1,4,Tipo),
                pieceAllowed(2,Tipo,WPieces,WMixed,BPieces,BMixed,Est), !.

playerInput(2,PlayerNo,Est,Linha,Coluna,Tipo,WPieces,WMixed,BPieces,BMixed) :-
        !, playerInput(1,PlayerNo,Est,Linha,Coluna,
         Tipo,WPieces,WMixed,BPieces,BMixed).

/* CPU controlled move decision, random */
playerInput(1,PlayerNo,Est,Linha,Coluna,Tipo,WPieces,WMixed,BPieces,BMixed) :-
        repeat,
                random(1,4,Tipo),
                pieceAllowed(PlayerNo,Tipo,WPieces,WMixed,BPieces,BMixed,Est), !,
        repeat,
                random(0,5,Linha),
                random(0,5,Coluna),
                getPos(Linha,Coluna,Est,0),
                \+ surrounded(Linha,Coluna,Est,PlayerNo,Tipo), !.

/* Updates piece counts */
countPieces(1,1,WPieces,WMixed,BPieces,BMixed,WPieces1,WMixed1,BPieces1,BMixed1) :-
        WPieces1 is WPieces - 1,
        WMixed1 is WMixed,
        BPieces1 is BPieces,
        BMixed1 is BMixed.

countPieces(1,3,WPieces,WMixed,BPieces,BMixed,WPieces1,WMixed1,BPieces1,BMixed1) :-
        WPieces1 is WPieces,
        WMixed1 is WMixed - 1,
        BPieces1 is BPieces,
        BMixed1 is BMixed.

countPieces(2,2,WPieces,WMixed,BPieces,BMixed,WPieces1,WMixed1,BPieces1,BMixed1) :-
        WPieces1 is WPieces,
        WMixed1 is WMixed,
        BPieces1 is BPieces - 1,
```

```
        BMixed1 is BMixed.

countPieces(2,3,WPieces,WMixed,BPieces,BMixed,WPieces1,WMixed1,BPieces1,BMixed1) :-
        WPieces1 is WPieces,
        WMixed1 is WMixed,
        BPieces1 is BPieces,
        BMixed1 is BMixed - 1.

/* Initial game starter call: 0- Human 1- Easy AI 2- Medium AI 3- Hard AI */
play(Player1,Player2) :-
        use_module(library(random)),
        initBoard(Board),
        printBoard(Board),
        play(Player1,Player2,Board,0,10,3,10,2,1).

/* Recursive game Loop for whites */
play(Player1,Player2,Board,EatenPieces,WPieces,WMixed,BPieces,BMixed,1) :-
        playerInput(Player1,1,Board,Linha1,Coluna1,Tipo1,
         WPieces,WMixed,BPieces,BMixed),
        movePlayer(Linha1,Coluna1, Board, Board1, Tipo1),
        countPieces(1,Tipo1,WPieces,WMixed,BPieces,BMixed,
         WPieces1,WMixed1,BPieces1,BMixed1),
        removeEaten(0,0,Board1,Board2,EatenPieces,EatenPieces1,1),
        printBoard(Board2),
        (endGame(Board2,WPieces1,WMixed1,BPieces1,BMixed1, 0) ;
        play(Player1,Player2,Board2,EatenPieces1,
         WPieces1,WMixed1,BPieces1,BMixed1,2)).

/* Recursive game Loop for blacks */
play(Player1,Player2,Board,EatenPieces,WPieces,WMixed,BPieces,BMixed,2) :-
        playerInput(Player2,2,Board,Linha1,Coluna1,Tipo1,
         WPieces,WMixed,BPieces,BMixed),
        movePlayer(Linha1,Coluna1, Board, Board1, Tipo1),
        countPieces(2,Tipo1,WPieces,WMixed,BPieces,BMixed,
         WPieces1,WMixed1,BPieces1,BMixed1),
        removeEaten(0,0,Board1,Board2,EatenPieces,EatenPieces1,2),
        printBoard(Board2),
        (endGame(Board2,WPieces1,WMixed1,BPieces1,BMixed1, 0) ;
        play(Player1,Player2,Board2,EatenPieces1,
         WPieces1,WMixed1,BPieces1,BMixed1,1)).

/* Checks if/Gets a move that eats a piece */
goodMove(L, C, _, _, _, _) :-
        (L > 5 ; C > 5),
        !, fail.
goodMove(5, Coluna, Linha1, Coluna1, Board, PlayerNo) :-
        ColunaNova is Coluna+1,
        goodMove(0,ColunaNova,Linha1,Coluna1, Board, PlayerNo).
goodMove(Linha, Coluna, Linha, Coluna, Board, PlayerNo) :-
        getPos(Linha, Coluna, Board, 0),
        removeEaten(0,0,Board,_,0,N1,PlayerNo),
        movePlayer(Linha, Coluna, Board, Board1, PlayerNo),
```

```
        \+ surrounded(Linha,Coluna,Board1,PlayerNo,PlayerNo),
        removeEaten(0,0,Board1,_,0,N,PlayerNo),
        N > N1,
        print('sugested '),nl,
        print(Linha),nl,
        print(Coluna),nl.
goodMove(Linha, Coluna, Linha1, Coluna1, Board, PlayerNo) :-
        LinhaNova is Linha+1,
        goodMove(LinhaNova,Coluna,Linha1,Coluna1, Board, PlayerNo).

/* Checks if/Gets a move is valid */
validMove(L, C, _, _, _, _, _) :-
        (L > 5 ; C > 5),
        !, fail.
validMove(5, Coluna, Linha1, Coluna1, Board, PlayerNo, Type) :-
        ColunaNova is Coluna+1,
        validMove(0,ColunaNova,Linha1,Coluna1, Board, PlayerNo,Type).
validMove(Linha, Coluna, Linha, Coluna, Board, PlayerNo,Type) :-
        getPos(Linha, Coluna, Board, 0),
        \+ surrounded(Linha,Coluna,Board,PlayerNo,Type).
validMove(Linha, Coluna, Linha1, Coluna1, Board, PlayerNo,Type) :-
        LinhaNova is Linha+1,
        validMove(LinhaNova,Coluna,Linha1,Coluna1, Board, PlayerNo, Type).

/* Checks if game ended and declares winner*/
endGame([],0,1,_,_,Count) :-
        Count > 0,
        print('Black wins! (Last piece must not be henge)').
endGame([],_,_,0,1,Count) :-
        Count > 0,
        print('White wins! (Last piece must not be henge)').
endGame([],0,0,0,0,Count) :-
        Count > 0,
        print('Whites win!').
endGame([],0,0,0,0,Count) :-
        Count < 0,
        print('Blacks win!').
endGame([],0,0,0,0,0) :-
        print('Tie! (Which means white wins!)').
endGame([[]|Xs],0,0,0,0,Count) :-
        endGame(Xs,0,0,0,0, Count).
endGame([[1|Xs]|Xss],0,0,0,0, Count) :-
        !,
        NewCount is Count+1,
        endGame([Xs|Xss],0,0,0,0, NewCount).
endGame([[2|Xs]|Xss],0,0,0,0, Count) :-
        !,
        NewCount is Count-1,
        endGame([Xs|Xss],0,0,0,0, NewCount).
endGame([[_|Xs]|Xss],0,0,0,0, Count) :-
        endGame([Xs|Xss],0,0,0,0, Count).
endGame(Board,W,H,_,_,_):-
```

```
        (W > 0 ; H > 0),
        (\+ validMove(0,0,L,C,Board,1,1) ; W =< 0),
        (\+ validMove(0,0,L,C,Board,1,3) ; H =< 0),nl,
        print('Black wins! (White has no possible moves!)').
endGame(Board,_,_,B,H,_):-
        (B > 0 ; H > 0),
        (\+ validMove(0,0,L,C,Board,2,2) ; B =< 0),
        (\+ validMove(0,0,L,C,Board,2,3) ; H =< 0),nl,
        print('White wins! (Black has no possible moves!)').

/*Prints a line of the board*/
printBoardLine([]) :-
        print('|').
printBoardLine([0|R]) :-
        print('|'),
        print(' '),
        printBoardLine(R).
printBoardLine([1|R]) :-
        print('|'),
        print('W'),
        printBoardLine(R).
printBoardLine([2|R]) :-
        print('|'),
        print('B'),
        printBoardLine(R).
printBoardLine([3|R]) :-
        print('|'),
        print('H'),
        printBoardLine(R).

/*Prints the board*/
printBoard(Est) :-
        nl,
        print(' |0|1|2|3|4|'),
        nl,
        printBoard(0,Est).
printBoard(_,[]) :- print('------------'), nl.
printBoard(Linha,[L|R]) :-
        print('------------'),
        print('\n'),
        print(Linha),
        printBoardLine(L),
        print('\n'),
        LinhaNova is Linha+1,
        printBoard(LinhaNova,R).

/*Removes a piece from the board*/
removePiece(0,0,[[_|LR]|R],[[0|LR]|R]).
removePiece(0, Coluna, [[L|LR]|R], [[L|LR1]|R1]) :-
        Coluna \= 0,
        ColunaNova is Coluna-1,
        removePiece(0, ColunaNova, [LR|R], [LR1|R1]).
```

```
removePiece(Linha,Coluna,[L|R],[L|R1]) :-
        Linha \= 0,
        LinhaNova is Linha-1,
        removePiece(LinhaNova,Coluna,R,R1).

/*Moves a piece to an empty spot*/
movePlayer(0,0,[[0|LR]|R],[[Tipo|LR]|R], Tipo).
movePlayer(0, Coluna, [[L|LR]|R], [[L|LR1]|R1], Tipo) :-
        Coluna \= 0,
        ColunaNova is Coluna-1,
        movePlayer(0, ColunaNova, [LR|R], [LR1|R1], Tipo).
movePlayer(Linha,Coluna,[L|R],[L|R1], Tipo) :-
        Linha \= 0,
        LinhaNova is Linha-1,
        movePlayer(LinhaNova,Coluna,R,R1,Tipo).

/*Gets the piece on a certain position of the board*/
getPos(N,P,_,3) :-
        (N < 0 ; N > 4);
        (P < 0 ; P > 4).
getPos(0,0,[[Peca|_]|_], Peca).
getPos(0,Coluna,[[_|R]|RR],Peca) :-
        Coluna > 0,
        ColunaNova is Coluna-1,
        getPos(0,ColunaNova,[R|RR],Peca).
getPos(Linha,Coluna,[_|R],Peca) :-
        Linha > 0,
        LinhaNova is Linha-1,
        getPos(LinhaNova,Coluna,R,Peca).

/*Checks if board position is surrounded by enemies*/
surrounded(Linha,Coluna,Est,1,1) :-
        LinhaAcima is Linha - 1,
        LinhaAbaixo is Linha + 1,
        ColunaFrente is Coluna + 1,
        ColunaTras is Coluna - 1,
        (getPos(LinhaAcima,Coluna, Est, 2) ; \+insideTable(LinhaAcima)),
        (getPos(LinhaAbaixo,Coluna, Est, 2) ; \+insideTable(LinhaAbaixo)),
        (getPos(Linha,ColunaFrente, Est, 2) ; \+insideTable(ColunaFrente)),
        (getPos(Linha,ColunaTras, Est, 2) ; \+insideTable(ColunaTras)).

surrounded(Linha,Coluna,Est,1,2) :-
        LinhaAcima is Linha - 1,
        LinhaAbaixo is Linha + 1,
        ColunaFrente is Coluna + 1,
        ColunaTras is Coluna - 1,
        (getPos(LinhaAcima,Coluna, Est, 1);getPos(LinhaAcima,Coluna, Est, 3) ;
        \+insideTable(LinhaAcima)),
        (getPos(LinhaAbaixo,Coluna, Est, 1);getPos(LinhaAbaixo,Coluna, Est, 3) ;
        \+insideTable(LinhaAbaixo)),
        (getPos(Linha,ColunaFrente, Est, 1);getPos(Linha,ColunaFrente, Est, 3) ;
        \+insideTable(ColunaFrente)),
```

```
        (getPos(Linha,ColunaTras, Est, 1);getPos(Linha,ColunaTras, Est, 3) ;
        \+insideTable(ColunaTras)).

surrounded(Linha,Coluna,Est,2,1) :-
        LinhaAcima is Linha - 1,
        LinhaAbaixo is Linha + 1,
        ColunaFrente is Coluna + 1,
        ColunaTras is Coluna - 1,
        (getPos(LinhaAcima,Coluna, Est, 2); getPos(LinhaAcima,Coluna, Est, 3) ;
        \+insideTable(LinhaAcima)),
        (getPos(LinhaAbaixo,Coluna, Est, 2); getPos(LinhaAbaixo,Coluna, Est, 3) ;
        \+insideTable(LinhaAbaixo)),
        (getPos(Linha,ColunaFrente, Est, 2); getPos(Linha,ColunaFrente, Est, 3) ;
        \+insideTable(ColunaFrente)),
        (getPos(Linha,ColunaTras, Est, 2); getPos(Linha,ColunaTras, Est, 3) ;
        \+insideTable(ColunaTras)).

surrounded(Linha,Coluna,Est,2,2) :-
        LinhaAcima is Linha - 1,
        LinhaAbaixo is Linha + 1,
        ColunaFrente is Coluna + 1,
        ColunaTras is Coluna - 1,
        (getPos(LinhaAcima,Coluna, Est, 1) ; \+insideTable(LinhaAcima)),
        (getPos(LinhaAbaixo,Coluna, Est, 1) ; \+insideTable(LinhaAbaixo)),
        (getPos(Linha,ColunaFrente, Est, 1) ; \+insideTable(ColunaFrente)),
        (getPos(Linha,ColunaTras, Est, 1) ; \+insideTable(ColunaTras)).

/*Removes all the eaten pieces from the board*/
removeEaten(5,0,New,New,EatenPiecesNew,EatenPiecesNew, _) :- !.
removeEaten(Linha,5,Old,New,EatenPiecesOld,EatenPiecesNew, PlayerNo):-
        Linha < 5,
        LinhaNova is Linha+1,
        removeEaten(LinhaNova,0,Old,New,EatenPiecesOld,EatenPiecesNew,PlayerNo).
removeEaten(Linha,Coluna,Old,NewNew,EatenPiecesOld,EatenPiecesNew, PlayerNo) :-
        Linha < 5,
        Coluna < 5,
        getPos(Linha,Coluna,Old,Tipo),
        surrounded(Linha,Coluna,Old,PlayerNo,Tipo),
        removePiece(Linha,Coluna,Old,New),
        ColunaNova is Coluna+1,
        EatenPiecesInc is EatenPiecesOld + 1,
        removeEaten(Linha,ColunaNova,New,NewNew,
         EatenPiecesInc,EatenPiecesNew,PlayerNo).
removeEaten(Linha,Coluna,Old,New,EatenPiecesOld,EatenPiecesNew, PlayerNo) :-
        Linha < 5,
        Coluna < 5,
        getPos(Linha,Coluna,Old,Tipo),
        \+ surrounded(Linha,Coluna,Old,PlayerNo,Tipo),
        ColunaNova is Coluna+1,
        removeEaten(Linha,ColunaNova,Old,New,
         EatenPiecesOld,EatenPiecesNew,PlayerNo).
```