GoRoGo

Progress Report

Master in Informatics and Computing Engineering

Logic Programming

Group GoRoGo_3:
David Alexandre Gomes Reis - up201607927
Dinis Falcão Leite Moreira - up201503092

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

October 15, 2017

1 The game

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.

1.1 History

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.

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.

1.2 Rules

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.

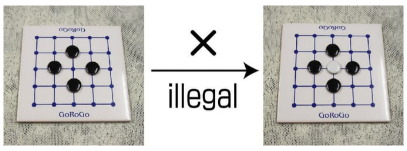


Figure 1: Illegal to place a piece surrounded by enemies

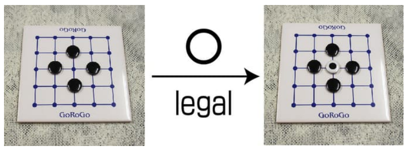


Figure 2: Legal to place a henge piece surrounded by enemies

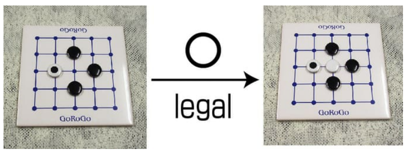


Figure 3: Legal to place a white piece if surrounded by enemies and a henge piece

1.3 Sources

1. Kickstarter: GoRoGo - An artisanal variation on the classic game of GO
2. Logygames: GoRoGo the new game of GO

2 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]]
```

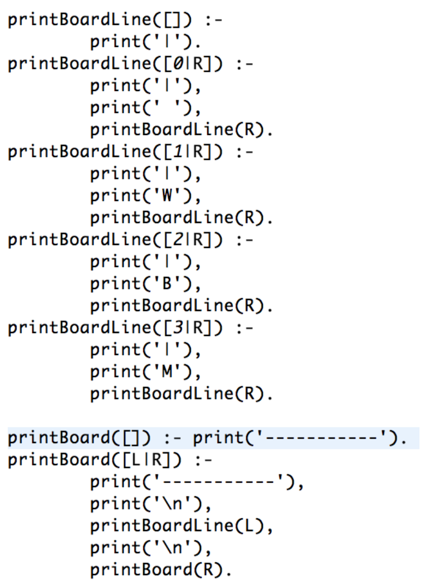
Figure 5: Mid-game state

```
[[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

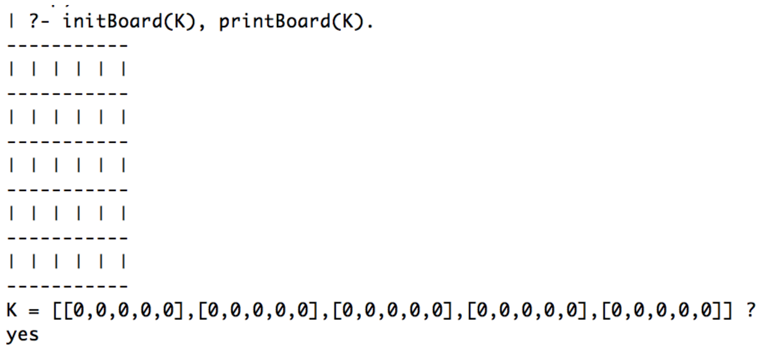
3 Game State Text Display

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.



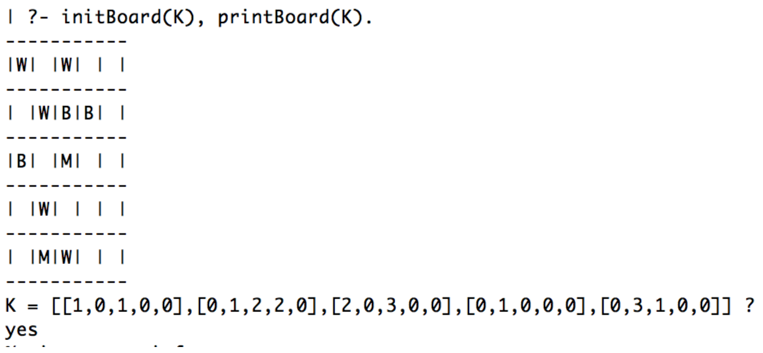
```
printBoardLine([]) :-
        print('|').
printBoardLine([0|R]) :-
        print('|'),
        print(' '),
        printBoardLine(R).
printBoardLine([1|R]) :-
        print('|'),
        print('W'),
        printBoardLine(R).
printBoardLine([2|R]) :-
        print('|'),
        print('B'),
     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



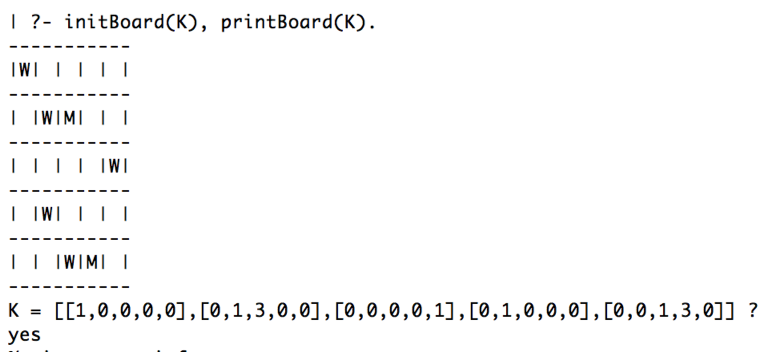
```
| ?- initBoard(K), printBoard(K).
-----------
| | | | | |
-----------
| | | | | |
-----------
| | | | | |
-----------
| | | | | |
-----------
| | | | | |
-----------
K = [[0,0,0,0,0],[0,0,0,0,0],[0,0,0,0,0],[0,0,0,0,0],[0,0,0,0,0]] ?
yes
```

Figure 8: Output of printing the initial board



```
| ?- initBoard(K), printBoard(K).
-----------
|W| |W| | |
-----------
| |W|B|B| |
-----------
|B| |M| | |
-----------
| |W| | | |
-----------
| |M|W| | |
-----------
K = [[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]] ?
yes
```

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



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

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

4 Possible Moves

The only requirement for a move to be valid in GoRoGo is that the slot where we place the piece is empty, as long as it's not surrounded by enemy pieces. Also, already placed pieces cannot move, besides leaving the board when they are surrounded by enemy pieces. For this reason, a possible "move" predicate could be *movePlayer(X, Y, CurrentState, NewState, Type)* where *X* and *Y* are the position on the board where we wish to place the piece, *CurrentState* is the current game state, *NewState* is the new game state after trying to place the piece, and finally *Type* is the type of piece we wish to place (1-3). This predicate needs only to check if the place (X, Y) is already occupied, and if not, replace the white space for the *Type* in the *NewState*. This predicate could be used in a few different ways. Given *X*, *Y*, the *CurrentState* and the *Type*, this predicate should return the *NewState*, however, given the *CurrentState* and the *Type*, this predicate could also return (through *X* and *Y*) all the possible moves. The actual usage of *movePlayer* depends only on our implementation of said predicate.

We also need the predicate *removeEaten(currentState, NewState, EatenPieces)* which removes the eaten pieces on each round, and increments the number of eaten pieces accordingly (important for end game detection, which is described below).

To detect the end of the game, we need to check if all the pieces on the board belong to one player and if the number of pieces currently on the board plus the number of eaten pieces (tracked by the predicate *removeEaten*) equals the total number of pieces given to each player at the start of the game. In that case, the player that still has pieces on the board is the winner. The predicate *checkWin(CurrentBoard,EatenPieces,Winner)*, where Winner equals 0 if no one has one yet, 1 if the player has won, and 2 if the "AI" has won.

In order to make the "AI" a decent player we could also have a predicate *goodMove(X, Y, CurrentState, Type)* which, given the *CurrentState* and the piece *Type*, return in *X* and *Y* a position which "eats" a piece of the opponent. In case there are no possible moves where the "AI" eats one of the opponent's pieces, it's also important to have a *badMove(X, Y, CurrentState, Type)* predicate which calculates all the spots where the piece we want to place would be eaten. If there are no good moves or bad moves on a turn, the "AI" should then randomly pick a position on the board and place it's piece there, using a predicate *randomMoves(X, Y, CurrentState, Type).* When choosing a move, the “AI” should try these predicates in this order.

In summary, the “AI” should first try to eat one of the opponent’s pieces, using *goodMove*. If there are no such moves, it should then simply avoid being eaten, using *badMove*. Again, if there are no such moves, it should then place a piece in a random slot, using *randomMove*.