PLOG 2020/2021 - TP1

Group: Jin_Li_2

Turma 3

Name	Number	E-Mail
Pedro Varandas da Costa Azevedo da Ponte	201809694	up201809694@fe.up.pt
Mariana Oliveira Ramos	201806869	up201806869@fe.up.pt

Installation and Execution

To correctly run and execute the game, you just need to, using SICstus Prolog, define the working directory of SICStus in the game folder and use the instruction consult('jin_li.pl') ou ['jin_li.pl'], followed by the predicate jin_li.

The Game - JinLi

Jin Li is a strategy game for 2 players. The players each control two fish in a pond (7x7 board), either two red fish (RF) or two yellow fish (YF). Besides the fishes each player also has 10 stones stored. Players take turns during the game moving their fish.

- Each player's Koi start in the corner squares closer to the player;
- On his turn, a player must either swim one of his fish and drop a stone or jump over a stone.
- A fish swims to an empty square adjacent (orthogonally or diagonally) to its current location. The stones are placed in an empty square. If a player has run out of stones then he does not drop after swimming.
- When jumping over a stone the jump must be along a straight line (orthogonally or diagonally).

After his turn, the player scores one point for each other fish adjacent to his fish's new location (the player can score 0, 1, 2, or 3 points on one turn). Keep track of the score using the scoring tracks placed on the top and bottom of the board. The first player to score 10 points wins.

Source Rules

Game Logic

Internal representation of the GameState

Board

To represent the cells of the board, we decided to use lists within a list. Each list inside the main list represents a line and each element of this list is the cell content. The content of a cell can be one out of three characters:

- **RF** - Red player's koi,
- **YF** - Yellow player's koi,
- **' '** - empty cell.

Some Possible Situation Representations

Initial Situation:

```
initialBoard([
[red,empty,empty,empty,empty,empty,red],
[empty,empty,empty,empty,empty,empty,empty],
[empty,empty,empty,empty,empty,empty,empty],
[empty,empty,empty,empty,empty,empty,empty],
[empty,empty,empty,empty,empty,empty,empty],
[empty,empty,empty,empty,empty,empty,empty],
[yellow,empty,empty,empty,empty,empty,yellow]
]).

     0   1   2   3   4   5   6
   |---|---|---|---|---|---|---|
A  |RF |   |   |   |   |   |RF |
   |---|---|---|---|---|---|---|
B  |   |   |   |   |   |   |   |
   |---|---|---|---|---|---|---|
C  |   |   |   |   |   |   |   |
   |---|---|---|---|---|---|---|
D  |   |   |   |   |   |   |   |
   |---|---|---|---|---|---|---|
E  |   |   |   |   |   |   |   |
   |---|---|---|---|---|---|---|
F  |   |   |   |   |   |   |   |
   |---|---|---|---|---|---|---|
G  |YF |   |   |   |   |   |YF |
   |---|---|---|---|---|---|---|

   yellow turn.
   Yellow player has 10 stones to play.
   Red player has 10 stones to play.
   Yellow Score: 0.
   Red Score: 0.
```

Intermediate Situation:

```
intermediateBoard([
[empty,empty,empty,stone,empty,empty,empty],
[empty,empty,empty,stone,stone,empty,empty],
[empty,empty,red,empty,empty,empty,stone],
[empty,stone,yellow,red,empty,stone,empty],
[stone,empty,stone,stone,empty,yellow,empty],
[empty,empty,stone,empty,stone,empty,empty],
[empty,empty,empty,stone,empty,empty,empty]
]).

    0   1   2   3   4   5   6
  |---|---|---|---|---|---|---|
A |   |   |   | O |   |   |   |
  |---|---|---|---|---|---|---|
B |   |   |   | O | O |   |   |
  |---|---|---|---|---|---|---|
C |   |   |RF |   |   |   | O |
  |---|---|---|---|---|---|---|
D |   | O |YF |RF |   | O |   |
  |---|---|---|---|---|---|---|
E | O |   | O | O |   |YF |   |
  |---|---|---|---|---|---|---|
F |   |   | O |   | O |   |   |
  |---|---|---|---|---|---|---|
G |   |   |   | O |   |   |   |
  |---|---|---|---|---|---|---|

  yellow turn.
  Yellow player has 4 stones to play.
  Red player has 4 stones to play.
  Yellow Score: 2.
  Red Score: 3.
```

Final Situation:

```
finalBoard([
[empty,yellow,empty,stone,empty,empty,empty],
[empty,stone,yellow,stone,stone,red,red],
[empty,empty,stone,stone,stone,stone,stone],
[empty,stone,stone,empty,empty,stone,stone],
[stone,empty,stone,stone,stone,empty,empty],
[empty,empty,stone,empty,stone,empty,empty],
[empty,empty,empty,stone,empty,empty,empty]
]).

    0   1   2   3   4   5   6
  |---|---|---|---|---|---|---|
A |   |YF |   | O |   |   |   |
  |---|---|---|---|---|---|---|
B |   | O |YF | O | O |RF |RF |
  |---|---|---|---|---|---|---|
C |   |   | O | O | O | O | O |
  |---|---|---|---|---|---|---|
D |   | O | O |   |   | O | O |
  |---|---|---|---|---|---|---|
E | O |   | O | O | O |   |   |
  |---|---|---|---|---|---|---|
F |   |   | O |   | O |   |   |
  |---|---|---|---|---|---|---|
G |   |   |   | O |   |   |   |
  |---|---|---|---|---|---|---|

  red turn.
  Yellow player has 0 stones to play.
  Red player has 0 stones to play.
  Yellow Score: 9.
  Red Score: 10.

  GAME ENDED
  Red player wins the game!
```

Atoms

Code	Meaning
code(empty, ' ')	*Empty square of the board*
code(red, 'RF')	*Red Fish*
code(yellow, 'YF')	*Yellow Fish*
code(stone, ' O')	*Stone*

GameState Visualization

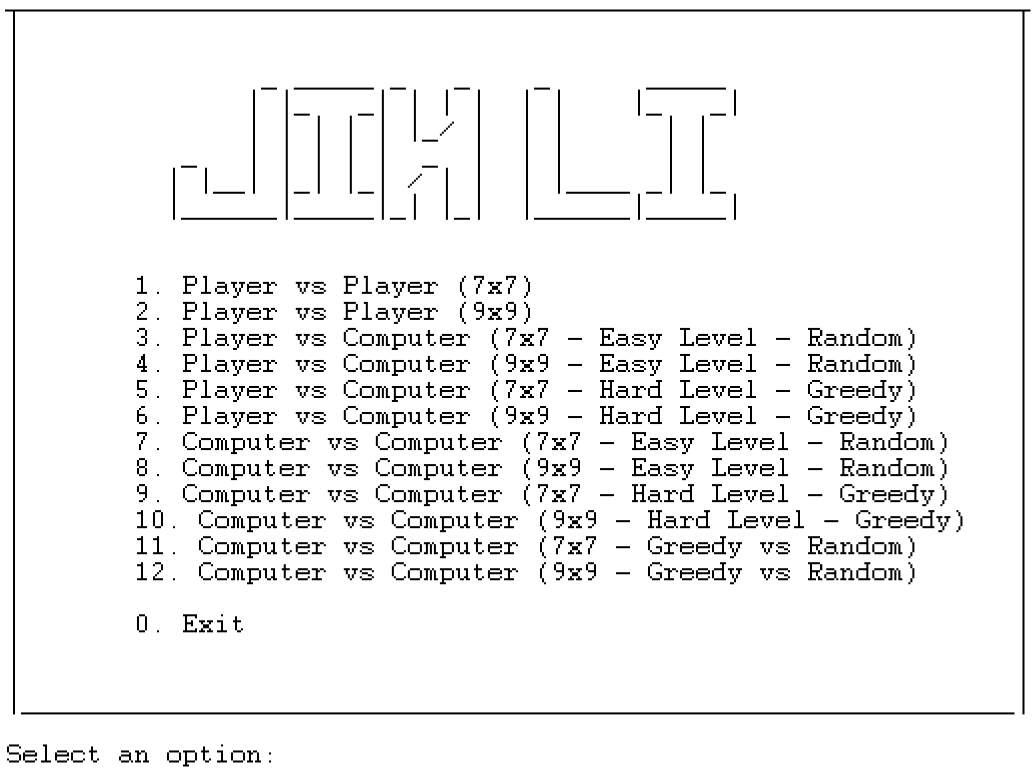
Menu

When initiating the game with the predicate `jin_li/0`, the main menu is displayed with options about the board size, the game type, the bot's difficulties and exiting the game. In order to select an option, the user must press the corresponding number, dot and enter. The inputs are validated by the predicate `checkOption/2` and read by the predicate `selectAction(+Option)`. After that, depending on the option chosen, one of the following predicates is called:

- `play(+Size)` which initializes the game Player vs Player;
- `playPVsComputer(+Size, +Mode)` which initializes the game Player vs Computer with Mode 'random' or 'greedy';
- `playComputerVsComputer(+Size, +Mode)` which initializes the game Computer vs Computer with Mode 'random' or 'greedy'. Each one of the predicates above chooses randomly the first player to move their koi. And after that calls the main game loop `start_game(-GameState, -Player, -YellowStones; -RedStones, -YellowScore, -RedScore, +Size)`.

All boards have a variable size that is passed to the game with **Size**.

All the predicates mentioned in this section can be found in the file menu.pl.



```
1. Player vs Player (7x7)
2. Player vs Player (9x9)
3. Player vs Computer (7x7 - Easy Level - Random)
4. Player vs Computer (9x9 - Easy Level - Random)
5. Player vs Computer (7x7 - Hard Level - Greedy)
6. Player vs Computer (9x9 - Hard Level - Greedy)
7. Computer vs Computer (7x7 - Easy Level - Random)
8. Computer vs Computer (9x9 - Easy Level - Random)
9. Computer vs Computer (7x7 - Hard Level - Greedy)
10. Computer vs Computer (9x9 - Hard Level - Greedy)
11. Computer vs Computer (7x7 - Greedy vs Random)
12. Computer vs Computer (9x9 - Greedy vs Random)

0. Exit

Select an option:
```

Board

In order to have a user-friendly game visualization, we decided to represent the game pieces with some symbols: **RF** for red fishes, **YF** for yellow fishes, **O** for stones, and **' '** for empty spaces. To do it, we use a predicate called `code(Value, Symbol)`.

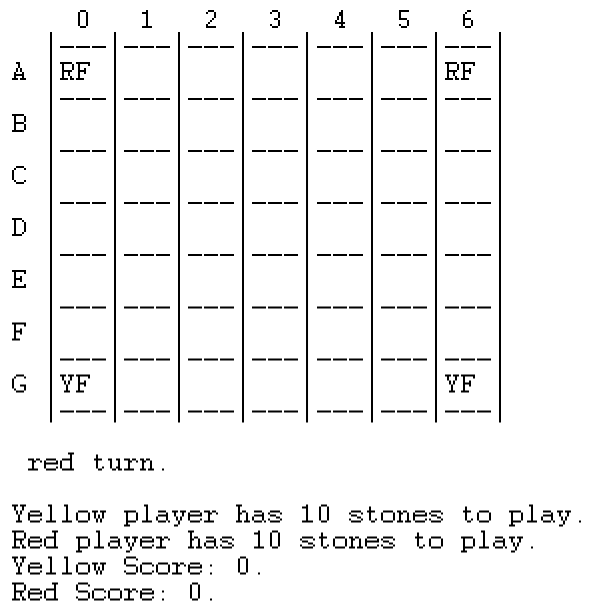
In order to create a board with a chosen size, we have implemented `generateRandomBoard(-GameBoard, +Size)`, that builds the board, row by row, calling the predicates `buildBoard/4` and `buildRow/4`.

To print the board, we only have to call `display_board/2` that uses the predicates:

- `printBoard(+GameBoard)` - calls predicates that will draw the header and the board;
- `printHeader/1` - prints the header that contains the number of the columns and the row separator;
- `printBoard/3` - prints the rest of the boards using predicates `letter/4` to get the respective row letter identifier, `print_line/1` that prints the respective row passed by argument, `print_cell/1` that displays a cell and `printRowSeparator/2`.

All the predicates mentioned in this section can be found in the file display.pl.

Initial game visualization example:

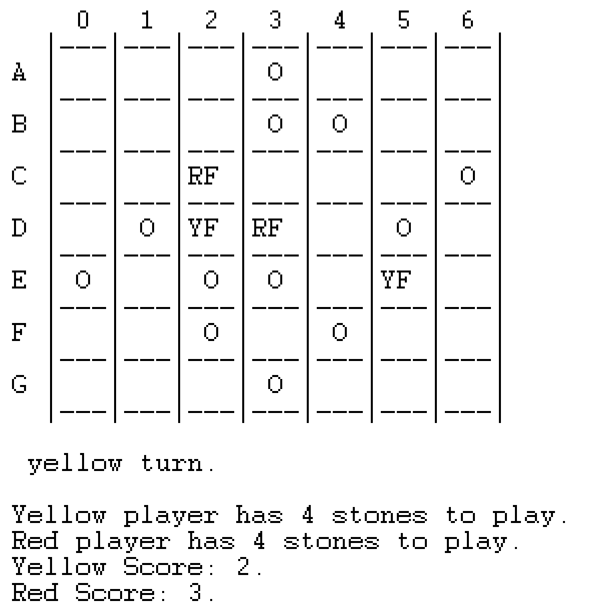


```
     0   1   2   3   4   5   6
   |___|___|___|___|___|___|___|
A  |RF |   |   |   |   |   |RF |
   |___|___|___|___|___|___|___|
B  |   |   |   |   |   |   |   |
   |___|___|___|___|___|___|___|
C  |   |   |   |   |   |   |   |
   |___|___|___|___|___|___|___|
D  |   |   |   |   |   |   |   |
   |___|___|___|___|___|___|___|
E  |   |   |   |   |   |   |   |
   |___|___|___|___|___|___|___|
F  |   |   |   |   |   |   |   |
   |___|___|___|___|___|___|___|
G  |YF |   |   |   |   |   |YF |
   |___|___|___|___|___|___|___|

 red turn.

Yellow player has 10 stones to play.
Red player has 10 stones to play.
Yellow Score: 0.
Red Score: 0.
```

Intermediate game visualization example:



```
     0   1   2   3   4   5   6
   |___|___|___|___|___|___|___|
A  |   |   |   | O |   |   |   |
   |___|___|___|___|___|___|___|
B  |   |   |   | O | O |   |   |
   |___|___|___|___|___|___|___|
C  |   |   |RF |   |   |   | O |
   |___|___|___|___|___|___|___|
D  |   | O |YF |RF |   | O |   |
   |___|___|___|___|___|___|___|
E  | O |   | O | O |   |YF |   |
   |___|___|___|___|___|___|___|
F  |   |   | O |   | O |   |   |
   |___|___|___|___|___|___|___|
G  |   |   |   | O |   |   |   |
   |___|___|___|___|___|___|___|

 yellow turn.

Yellow player has 4 stones to play.
Red player has 4 stones to play.
Yellow Score: 2.
Red Score: 3.
```

Final game visualization example:

```
     0   1   2   3   4   5   6
   |---|---|---|---|---|---|---|
 A |   |YF |   | O |   |   |   |
   |---|---|---|---|---|---|---|
 B |   | O |YF | O | O |RF |RF |
   |---|---|---|---|---|---|---|
 C |   |   | O | O | O | O | O |
   |---|---|---|---|---|---|---|
 D |   | O | O |   |   | O | O |
   |---|---|---|---|---|---|---|
 E | O |   | O | O | O |   |   |
   |---|---|---|---|---|---|---|
 F |   |   | O |   | O |   |   |
   |---|---|---|---|---|---|---|
 G |   |   |   | O |   |   |   |
   |---|---|---|---|---|---|---|

  yellow turn.

 Yellow player has 0 stones to play.
 Red player has 0 stones to play.
 Yellow Score: 9.
 Red Score: 10.
```

Valid Moves

The `valid_moves(+GameState, +Player, -ListOfMoves, +Size)` returns on ListOfMoves a list of moves in the format `[[Fish1Row, Fish2Row, [MoveRow1, MoveColumn1], [MoveRow2, MoveColumn2], ...] [Fish2Row,Fish2Column,[MoveRow3,MoveColumn3], [MovRow4-MovColumn4], ...]]`. This predicate first calls `getPlayerPos(+GameState, +Player, -ListOfPositions, +Size)` that goes through the board and gets the position of all the player's pieces, the positions are stored in Positions.

Then it is called `getAllPossibleMoves(+GameState, +Palyer, +ListOfPositions, -ListOfMoves, +Size)` that using `getMoves(+GameState, +InitRow, +InitColumn, -Moves, +Size)`, sees all the possible moves (with `getAdjacentes(+GameState, +Row, +Col, -Adj, +Size)` and `getPossibleJumps(+GameState, +InitRow, +InitColumn, +ListInt, +ListAux, -ListRes)`). This verification consists of checking the existence of an empty cell in any surrounding position including the ones after jumping.

All the predicates mentioned in this section can be found in the file game_logic.pl. They are mainly used on the game loop that you can find in play.pl

---

Making moves

The `move(GameState, Move, MidGameState, Player, Jump, Size)` asks the player for position inputs that are validated using the predicates `checkRow/3` and `checkColumn/3`. After that, the predicate `validateContent/7` is called to check if the player is selecting one of their pieces. If any of these verifications fail, the predicate asks again for the input. If it succeeds the predicate `selectSpot(+GameState, +Player, -MidGameState, +InitRow, +InitColumn, -NewRow, -NewColumn, -Jump, +Size)` is called asking the user for inputs and validating if it is an empty cell to put the koi using the predicate `validateMove/10`.

If all the verifications succeed, then it is called `replaceValueMatrix/5` that replaces the old koi's position for an empty space and put the koi in the selected position, obtaining a new GameState -

MidGameState.

Given that each player may or may not put a stone after a move we included the predicate `canPutStone(+NumStones, +MidGameState, +Player, -FinalGameState, -NumStonesFinal, +Jump,+Mode, +Size)` that checks if the player still has stones left and did not jump. If those conditions are met then the predicate `selectSpotStone(+GameState, -Player, -FinalGameState, +Size)` is called asking the user for inputs and validating if it is an empty cell using the predicate `validateStoneSpot/6`.

All the predicates mentioned in this section can be found in the file inputs.pl except for the `canPutStone/8` that can be found in game_logic.pl and `replaceValueMatrix/5` utils.pl.

Board Evaluation

The evaluation of the board is made using the `value(+GameState, +Player, +FinalRow, +FinalCol, -Value, +Size)`. Although this predicate has a different amount of arguments than those requested by the teachers, we implemented the predicate this way to optimize the evaluation. Using the predicates getAdjacentes and calculateScore, we just unified the variable *Value* with the *Score* obtained. This way we can move the kois to the possible place that will add the most score.

All the predicates mentioned in this section can be found in the file game_logic.pl.

Computer Move

To choose a computer move we use the predicate `choose_move(+GameState, +Player, +Level, -Move, Size)`, where `Level` will be 'random' or 'greedy', the two difficulties we implemented in our game.

First, `valid_moves/4` (explained above) is used to get all the possible moves for both kois, and then we will choose a move from `ListOfMoves` according to the level, explained in the sections below.

All the predicates mentioned in this section can be found in the file ai.pl.

Level 1 - Random

In level 1, the koi and the respective move chosen will be random using `random_member(-Elem, +List)`.

Level 2 - Greedy

In level 2 the move will be greedy, choosing the best move in the current turn. In this case both `getMovesValuesBot/7` and `selectBestMove/5` select a move using `findall(+Template, +Generator, -List)`. In the `Generator` the`value/4` predicate, explained above, is used to evaluate the board after a move. This way these predicates end up returning all possible moves to the current player. The `List` is `Value-InitialPosition-FinalPosition-Index`. Finally, the list is sorted, using `sort(+List1, -List2)`, being in ascending order of Values and `reverse(+List, -Last)` is used so we can get the move with the highest value.

Placing Stones

Given that each player may or may not put a stone after a move we created the predicate `choose_stone(+MidGameState, -FinalGameState, +Player, +Level, +Size)` that is called after checking if the player can actually place a stone on the board. This predicate receives a Level just like the chose_move. If the Level is 'random' then the stone will be placed on an empty cell of the matrix. Otherwise, if the Level is 'greedy', we decided to go for a sabotaging approach. The predicate `choose_move/5` is called passing the oponent Player. This way we are able to put a stone in one of the positions where the opponent would make more points.

In level 1, the koi and the respective move chosen will be random using `random_member(-Elem, +List)`.

Game Over

At the end of every turn, two predicates are called:

- `calculateScore(+FinalGameState, +Adj, +Score, -FinalScore)` that updates the score of the player who just played, analyzing the current GameState with the help of the previously called predicate `getAdjacentes/5`;
- `checkGameOver(+Player, -NextPlayer, +Score)` in order to check if the game is over, according to the rules already presented. This predicate evaluates whether the current Player won or not by evaluating his current Score. If the Score is 10, means the game ended and in that case, NextPlayer is unified with the term 'end'. Otherwise, the variable NextPlayer becomes the next player to play.

All the predicates mentioned in this section can be found in the file game_logic.pl.

Conclusion

We feel that this project developed our knowledge and understanding of Prolog. Since Prolog was a brand new language to us, we had some difficulties starting the disenrollment of the game, however we quickly became accustomed to the syntax.

For our knowledge, we didin't encouter any bugs or limitations.

For possible improvements, we could implement more difficulty modes or made our bot moore intelligent: for example, to choose a place for a stone, the bot could take in account not only the possible moves of the enemy, but also his own situation and possible next moves that would make him score more points in the next turn.

Bibliography

- SICStus Documentation;
- Moodle slides;
- StackOverflow to check out some doubts that appeared while we were developing this project.