Quixo

Ângela Coelho (up201907549) and Nuno Castro (up202003324)
from 3MIEIC04, Quixo_1

Bachelor in Informatics and Computing Engineering 2021/2022 – 1 st Semester

Introduction

The main goal of this project is to recreate the board game Quixo at a computational level using the **Prolog** language. We will start with the problem description where we explain the game rules and our approach to implement it in the context of this course.

Game Description

The board is a NxN matrix composed of pieces in the shape of a cube. Each piece has one side with a cross and the other side with a circle and the rest of them are blank. The game starts with all of the pieces with the blank side facing up. Each player has a symbol associated.

On a turn, the current player can play a blank piece, by facing his symbol up, or a piece that already has his symbol facing up. Then, after taking that piece from the board, there will be a free space that will be filled by pushing the other pieces orthogonally and placing the chosen piece in the free space created after pushing the other pieces.

Therefore, as the game proceeds, the board will be filled with pieces containing crosses and circles and ends whenever a player formas an orthogonal or diagonal line of his pieces, with this person being the winner.

In order to understand and know more about this game, we used this site as reference.

Installation and Execution

This project was developed using SICStus 4.7 and Windows 11. However, it can be also executed in Linux distributions.

To download SICStus 4.7 visit the following site and choose the one suitable for your Operative System. After the installation, it will ask for license information which will vary according to the OS as follows:

- **Unix**
  Site name: student.fe.up.pt

License code: hh32-bkq6-qssc-cpxq-g7d3
Expiration date: permanent

- **Darwin**
  Site name: student.fe.up.pt
  License code: 33be-bkq6-qssc-da4m-o9sc
  Expiration date: permanent

- **Win32**
  Site name: student.fe.up.pt
  License code: be8a-bkq6-qssc-ct96-se8b
  Expiration date: permanent

After installing **SICStus 4.7** you just need to follow these steps in order to play the game.

- **Windows**
  1. Open the SICStus executable.
  2. In the top menu click File > Consult and then choose the *main.pl*.
  3. After consulting the game main file it is possible to start playing the game by typing ***play.*** in the console.

- **Linux**
  1. Open a new terminal and type

Game Logic

Game State Intern Representation

The game state is composed by a ***Board*** and a ***CurrentPlayer***. The ***Board*** which represents a NxN matrix is essentially a list of lists each one representing a row. It starts with zeros representing the pieces facing up the blank face and then, whenever a move is done, the respective piece changes to the symbol associated with the ***CurrentPlayer***. ***CurrentPlayer*** can be 1 or 2 representing *Player 1* or *Player 2*, respectively.

Game State Visualization

The game starts with an initial state represented by the predicate ***initial_state(+Size, -GameState)*** that, as said previously, starts with the ***GameState*** being composed by the ***Board*** filled with zeros and the ***CurrentPlayer*** being the *Player 1* which is represented by the cross symbol. This predicate receives the board size which can be 3, 4 or 5

To represent the ***GameState*** visually it is used the predicate ***display_game(+GameState)*** which shows a grid with the corresponding pieces and the identifiers for the rows and columns in order to make the game more user friendly.

Game Move

In order to make a move the player has to select a piece through its coordinates in the board. Then, a verification of the possible directions in which the player can push the other pieces and place his is made. After this verification, those directions become available for the player to choose and make his move. Horizontal movement happens obtaining the row in which the movement happens, deleting the selected piece from that row and appending it at the head or tail depending on the direction. Vertical movement occurs overwriting, recursively, the piece of a row with the piece of the previous or next one, depending on the direction of the movement. After the push is done the player's piece is placed.

End Game

Despite not working properly in the game, predicates to verify the end of a game were made.

To verify if a player completed a line, the predicate **linesWin** would analyze the board row by row and then for each row the elements would be compared with the player's symbol.

To verify if a player completed a column, the predicate **colWin** would analyze the board column index by column index and then for each one, the rows would be iterated and the elements compared to the player's symbol.

To verify if a player completed a diagonal, the predicates **diagonalWin** and **diagonal2Win** would analyze the board, iterate simultaneously the row and column and then compare the elements to the player's symbol.

Conclusion

The project was very challenging at first since it was developed in a language very different from what we are used to work with during our academic path. However, this doesn't mean it wasn't interesting, in fact, quite the opposite: it helped us understand the power of these type of programming languages and ideas.

We are happy with our results, but we also know that the project could have been more explored if only we had more time, however the available time didn't contribute at all. Because of this we weren't able to implement the option of a player to play against the computer nor the option of two computers playing against each other.

Despite this, as said before, we were able to learn much more about this type of technology and get comfortable with, bearing in mind that there is always room for improvement and learning, as seen.