369_1

Isabel André Amaral - up202006677 (50%)
Milena Luísa Pereira Gouveia - up202008862 (50%)

1. Installation and Execution

In order to run the game you must have SICSus Prolog 4.7.1 installed in your machine.

On the SICStus interpreter consult de file *play.pl* in the project's *src* directory. Then, run the predicate play/0 to start the game.

2. Game Description

The game should be played on a 9x9 square board. For a faster and simpler game, a 6x6 board can also be used. However, our interactive menu allows to choose and play on a board of any size.

The board is initially empty. One player plays with white pieces and the other with black pieces. The player with white pieces always goes first.

At each turn, each player drops a stone at one of the board's empty positions. For every 3 stones in any line, either orthogonal or diagonal, the player gets 1 point. For every 6 stones, gets 2 points. For every 9 stones, gets 3 points. Pieces of the same colour only need to be in the same line, they don't need to be consecutive.

The game ends when the board is full and the player with the higher score wins.

3. Game Logics

3.1. Internal Representation of the Game State

The **GameState** stores information about the current state of the game. It consists of a list containing the board size, the board itself, the game level, the scores of the players with the white and black pieces and the player that will play next.

The **Board** is represented as a two-dimensional list, i.e. a matrix, which stores the positions of the pieces belonging to each player. The board is updated as each player makes their moves, with the pieces being placed in the positions chosen by the players. Each position of the board is represented as a character, `w` for white pieces, `b` for black and `e` for an empty space.

Some examples of the **GameState** during the game:

- <u>Initial game state:</u>

  ```
  [3, [[e,e,e],[e,e,e],[e,e,e]], 1, 0, 0, w]
  ```

  By examining the GameState list, we can see that the board size is 3, the board is currently empty, the game level is 1, and both white and black pieces have 0 points. The player with the white pieces is the next one to make a move.

- <u>Intermediate game state:</u>

  ```
  [3, [[w,b,e],[w,b,e],[w,b,e]], 1, 1, 1, w]
  ```

  In this GameState, the board size is 3, there are 3 empty positions of the board and the level is 1. Both white and black pieces have scored 1 point. The player with the white pieces will make the next move.

- <u>Final game states:</u>

  ```
  [3, [[w,b,w],[w,b,b],[w,b,w]], 1, 1, 1, b]
  ```

  In this final GameState, the board size is 3 and all positions of the board are occupied. The game level is 1 and both white and black pieces have scored 1 point.

3.2. Game State Visualization

The visualization of the state of the game at a given moment is done using the predicate `display_game(+GameState)` defined in *display.pl*, which uses some auxiliary predicates also defined in that file. This predicate is used to, in each game cycle iteration, display the current board and pieces placed in it and the current score of each player. This is done in an intuitive way so that the player may easily evaluate what's the best position for the next move and input said position in a line-column format.

The state of the game is initialized based on the input information received from the user through the predicate `menu(-GameType, -Level, -Size)` defined in *menu.pl*, which uses auxiliary predicates to choose the type of the game (h/h, h/pc or pc/pc), its level (1 or 2 for games involving at least one computer) and the size of the board.

After the inputs for these fields are collected from the user, the game state is initialized using the predicate `initial_state(+Size, +Level, -GameState)`. The **Size** and **Level** fields are initialized with the user's choices, the **Board** is initialized as a Size x Size matrix with all positions empty, the **WhitePlayer** and **BlackPlayer**, referring to each of the players' scores, are initialized to 0 and **NextPlayer** is initialized to **`w`** since the player with the white pieces always goes first.

3.3. Move Execution

The execution of a valid move, obtaining the new state of the game, is achieved using the predicate `move(+GameState, +Move, -NewGameState)` defined in *move.pl*.

This predicate first uses the `get_next_player/2` and `get_board/2` predicates to extract the player whose turn it is and the current board. The player's piece is then placed on the board using the predicate `insert_piece_into_board(+Board, +Piece, +Line, +Column, -NewBoard)` defined in *move.pl*, which inserts the piece at the specified position according to the Move variable, which is in the [Line, Column] format.

After placing the piece, the board is updated using `update_board/3`, and the player's points are updated using `update_points/3`, which adds the points gained from their move to their current score. These points are computed by summing the vertical, horizontal and diagonal points generated by the move. The next player is then updated (`update_next_player/2`).

3.4. Computer Move

In h/pc and pc/pc games, the move to be played by the computer is chosen using the predicate `choose_move(+GameState, +Player, +Level, -Move)` defined in the *move.pl* file and depends on the level of the game being played.

In level 1, the computer's moves are chosen randomly among all the empty positions using the `random/3` predicate from the `random` module.

In level 2, the computer's moves are chosen using a greedy algorithm that evaluates which move may guarantee more points in the current game cycle iteration. If there aren't, at the moment, any moves that will guarantee points, the move is, once again, chosen randomly among all the empty positions, similarly to what happens in level 1. The algorithm works in the following way:

- From all the lines in the board, we select the one that guarantees more points if a piece is played anywhere in that line. Ex: In a 9x9 board with 7 lines with 1 black piece each, 1 line with 2 black pieces and 1 line with 6 black pieces, the line with the 2 black pieces will be chosen because a move in that line would make 3 black pieces and guarantee the computer 1 point. If there is more than one line that guarantees the same number of points, the first one is chosen.

- From all the columns in the board, we select the one that guarantees more points if a piece is played anywhere in that column.
- From all the diagonals in the board, we select the one that guarantees more points if a piece is played anywhere in that diagonal.
- Among the best line, the best column and the best diagonal, we select the one that guarantees more points if a piece is played in that line/column/diagonal. If there is a tie, the choice is made by the following order: line, column, diagonal.
- The computer will play anywhere in the selected line/column/diagonal. Since the point is earned independently from where in the line/column/diagonal the move is made, it's chosen randomly among all the empty positions in that line/column/diagonal.

3.5. Listing of Valid Moves

The predicate `valid_moves(+GameState, +Player, -ListOfMoves)` defined in *move.pl,* obtains a list of all the valid moves that a player can make.

It first uses `get_board/2` to obtain the board and finds every empty position in that board. This is done by recursively going through the rows and columns of the board and adding the positions of any empty spaces, denoted by the **`e`** character, to a list. It is important to note that the valid moves do not depend on the player, so an auxiliary predicate without the Player argument is used.

3.6. Game State Evaluation and Game Over

The predicate `game_over(+GameState, -Winner)` defined in *369Game.pl,* verifies if the game has ended and, if so, identifies the winner.

This predicate first uses the `valid_moves/3` predicate, defined in *move.pl,* to check if there are any valid moves left for the current player. If not, then the game is over.

Next, it uses the predicate `value(+GameState, +Player, -Value),` defined in *points.pl,* to get the score of each player and passes these values to an helper predicate which determines the winner by comparing the players' scores, returning the character **`w`** for the player with the white pieces or **`b`** for the player with black ones. If both players have the same points, the character **`t`** for tie is returned.

Finally, the predicate `game_over/2` calls `display_winner/1` to display a user-friendly message indicating the winner.

4. Conclusion

The game described in the game description section was successfully implemented. Its biggest limitation, which also corresponds to our biggest implementation difficulty, lies in the greedy algorithm used in level 2 to choose the best move for the computer. The algorithm implemented only looks at the move that will immediately guarantee the most points, instead of evaluating the best strategy in the long run or even trying to block positions that could be useful for the other player, which reduces its cleverness.

However, the group didn't correctly understand the rules of the "real" 369 game until our version was fully implemented and it was too late to completely change its logic. In reality, the game we should have implemented had pieces of only one colour instead of different colours for the different players. Even so, we believe the logic wouldn't be much different if we had implemented it like this from the start and, if anything, our version ended up adding a little bit of complexity to the project.

5. Bibliography

- https://www.di.fc.ul.pt/~jpn/gv/369.htm