Claustro - Board Game

Group: Claustro_1

Luís Filipe da Silva Jesus - up202108683 - 50%
Miguel Diogo Andrade Rocha - up202108720 - 50%

Installation and Execution

After installing SICStus Prolog, you can start the game with the following steps:

1. Launch the SICStus Prolog environment.

2. Consult the play.pl file that contains the game source code by using the File menu in the SICStus Prolog GUI or by entering the consult command in the SICStus Prolog console followed by the path of the file.
consult('path/to/play.pl').

3. Once the play.pl file has been consulted, start the game by typing the command:

play.

Game Description

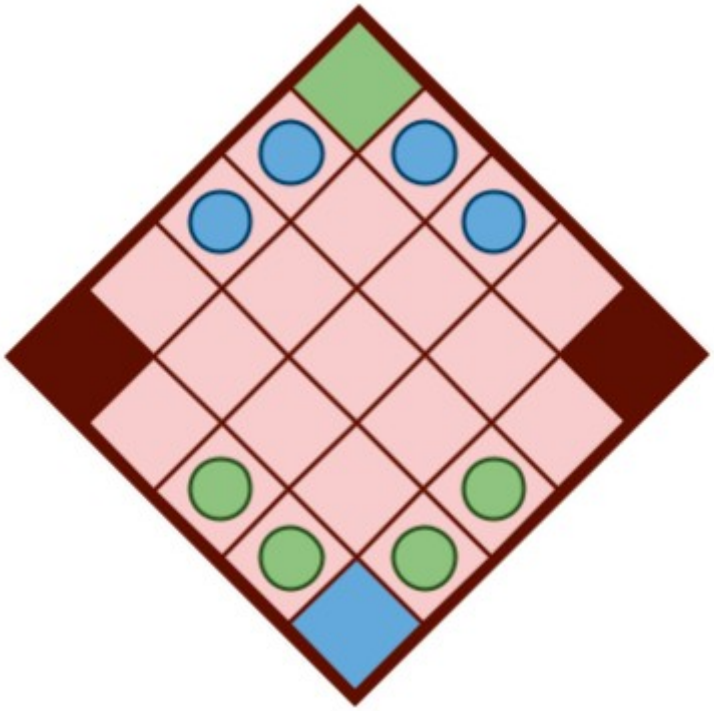
Board

•Claustro typicaly uses a 5x5 board/grid. However we made it possible for the user to choose the width and height of the board. Usually, the board is diagonally oriented between the two players. The two corners closest to each player are each other players' goal.

•The other two corners of the board are neutral zones, where no pieces can be placed.

Pieces

•Each player has (width - 3) + (height - 3) pieces. Initially, each piece is placed next to each other starting in each square orthogonally adjacent to the player starting corner.



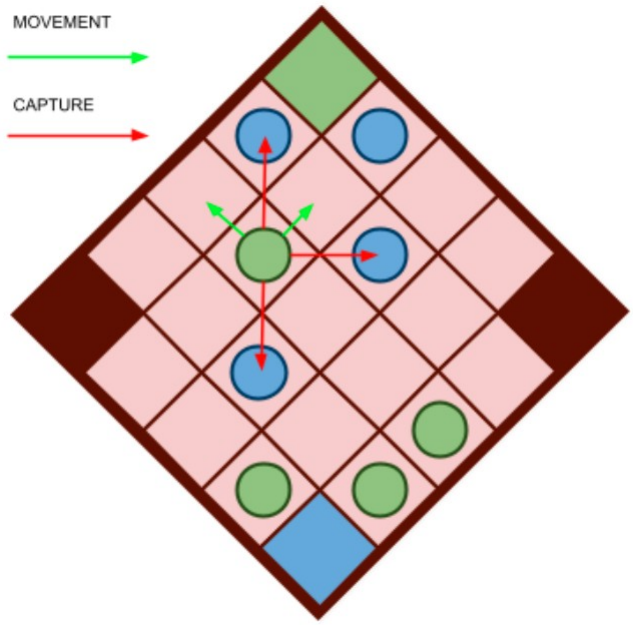
To disinguish between the two players, each player has a different color, for example, green and blue.

Gameplay

Players take turns moving one of their pieces. In each turn, a player can move one of its pieces in the following ways:

- Move: Move a piece one square orthoganally towards the goal (considering it is not occupied).

- Capture: Captures are made by jumping one square (where there is an opponent's piece) diagonally. The captured piece is not removed from the board, but placed in any of the other free squares in the board, by the player who captured it, before the next turn.



This procedure shall be repeated until one of the following conditions is met:

•A player successfully moves one of his pieces on his goal (opponent starting corner). In this case, he wins the game.

•A player cannot move nor capture opponent pieces with any of his pieces. In this case, he wins the game.

•Both players repeat the same move 3 times each. In this case, the last player to move wins the game.

•Source

Game Logic

Internal Game State Representation:

•The main tuple representing the game state is of the form (Turn, MoveHistory, Board):
  •Turn: An integer that indicates which player's turn it is. If the number is odd, then it represents the green player's turn.
  •MoveHistory: A list that records the moves made during the game. This can be used to undo moves or replay the game.
  •Board: A list of lists, where each sublist represents a row on the board. The elements of these sublists represent individual squares on the board, which can contain atoms such as empty, blue, green,

blueGoal, greenGoal, and neutral. These atoms represent different pieces or states of a square on the board.

•Dynamic Board Size: The initial_state/2 predicate initializes the game board. It takes the dimensions of the board and applies a series of transformations (setBoard, setSpecialSquares, setInitialPieces) to set up the initial game board with pieces and special squares.

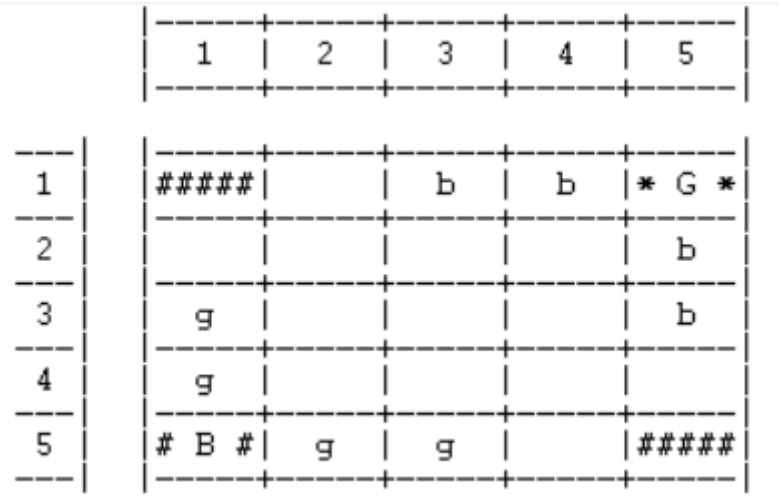
Examples of board states:

•Initial State: All pieces are in their starting positions.

```
[
   [neutral, empty, blue, blue, greenGoal],
   [empty, empty, empty, empty, blue],
   [green, empty, empty, empty, blue],
   [green, empty, empty, empty, empty],
   [blueGoal, green, green, empty, neutral]
]
```

Below is a visual representation of the initial board state:

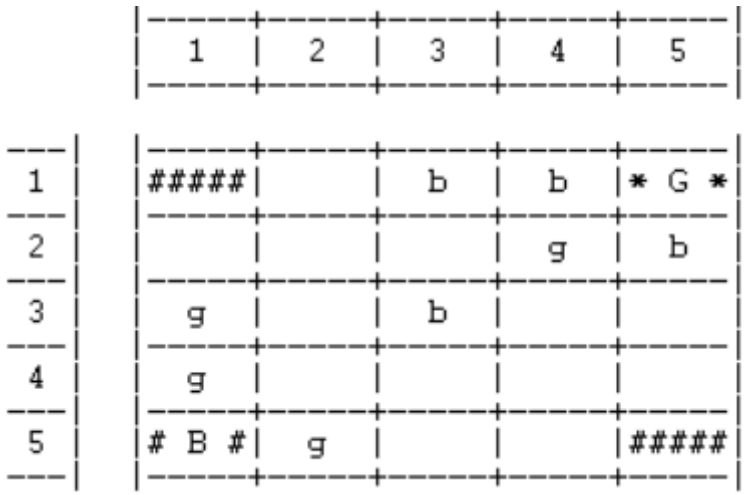
•



Intermediate State: Some pieces have been moved from their starting positions.

```
   [
      [neutral, empty, blue, blue, greenGoal],
      [empty, empty, empty, green, blue],
      [green, empty, empty, blue, empty],
      [green, empty, empty, blue, empty],
      [blueGoal, green, empty, empty, neutral]
   ]
```

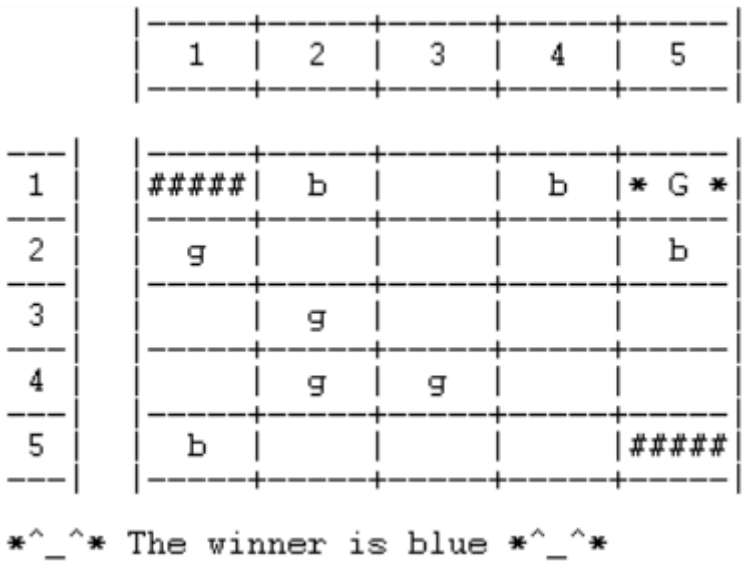
Below is a visual representation of the intermediate board state:



•Final State: The game has reached an end condition, like all pieces of one color have been captured. For example blue reached blueGoal

```
[
    [neutral, blue, empty, blue, greenGoal],
    [green, empty, empty, empty, blue],
    [empty, green, empty, empty, empty],
    [green, green, green, empty, empty],
    [blue, empty, empty, empty, neutral]
]
     ^
     |
(blueGoal position)
```

Below is a visual representation of the final board state:



Game State Visualization

•display_game(+GameState) is a predicate responsible for visualizing the current game state. It takes the current game state tuple and displays the board and other relevant information to the player. This includes displaying the columns and the pieces on the board using ASCII characters or symbols to represent different pieces.
•The game state visualization makes use of auxiliary predicates such as displayCols/1, headerBorder/1, and displayBoard/3 to create a user-friendly display of the board. These predicates are designed to be flexible, accommodating boards of various sizes and configurations.

Menu and Interaction

•Input Validation: Inside the gameloop, the system prompts the user for input and validates it, ensuring that the moves and selections made are within the acceptable range and logical for the game's current state, according to these predicates:

•moveTypeChoice/2: Presents the options for moving a piece or capturing a piece.
•askMoveType/3: Asks the user to select a move type, validating the move based on the game state.
•choosePiece/3: Allows the player to choose a piece to move, with the system handling both human and bot players.
•askBoardPosition/4: Prompts for a board position to move the piece and ensures the input is within the allowed range.
•askReplacePosition/4: In the event of a capture, this allows the user or bot to choose where to place the captured piece (must be an empty square).
•choose_move/4: The process of selecting a move during the game, handling different phases such as move execution, capture, and automated move selection based on the player type (human or bot).

Move Validation and Execution

•The predicate responsible for move execution is move/3, taking the current game state and a proposed move, produces a new game state that reflects the move. However, the proposed move is validated before calling move/3 while asking for user inputs as explained before or choosing random moves for the easy bot.

Move Validation

•Before executing a move, it is validated using a set of rules defined for the game. These include:

•Movement: A piece can move ortogonally in the direction of the playerGoal to an adjacent square if it is empty or to a goal square.

•Capture: A piece can capture an opponent's piece if the opponent's piece is placed in an adjacent diagonal square.

•Game Over Conditions: The Game Over Conditions are checked in the gameloop after a move is executed. These conditions would end the game, such as reaching the opponent's goal line and having no valid moves available or both players making the same move 3 times in a row.

Execution of Move

- Once a move has been validated, the move/3 predicate will:

•Update the board to reflect the piece's new position.
•Capture any opponent pieces if the move is a capturing move.
•Update any state information that is dependent on the move, such as the turn number or scoring.
•Return the new game state.

Helpers and Utilities:

•Several helper predicates are used in conjunction with move/3:

•`canMove/6` and `canCapture/6` for checking the validity of non-capturing and capturing moves, respectively.
•`valid_move/3` and `valid_moves/3` for finding all valid moves for a piece or player.
•`gameOver/2` for checking if a game-ending condition has been reached.
•`change_cell/5` for modifying the board state by changing the value of a specified cell.

List Of Valid Moves

valid_move((Player, X1, Y1), (MoveType, X2, Y2), Board).

- •This predicate takes a move and a board as input and decides if the move is valid based on the current state of the game.

- •There are two types of moves:

  - •Normal move (MoveType is 0): The move is valid if canMove/6 is satisfied.
  - •Capture move (MoveType is 1): The move is valid if canCapture/6 is satisfied.

valid_moves((Turn, _, Board), Player, ListOfMoves).

- •This predicate generates all valid moves for a given player during their turn.

  - •First, the size of the board is obtained using getBoardSize/3.
  - •The predicate uses findall/3 to generate a list of all valid normal moves (ListOfMoves0 - Moves) and capture moves (ListOfMoves1 - Captures).
  - •After generating both lists, they are concatenated using append/3 into a single list (ListOfMoves2).
  - •Finally, the list is sorted to give the final ListOfMoves, ensuring that no duplicate moves are present.

End Of Game

- •The predicate game_over(+GameState, -Winner) is used to check if the game is over and who won the game. It takes the current game state and returns the winner of the game or false if the game is not over.

- •The game is over if one of the following conditions is met:

  - •A player has reached the opponent's goal.
  - •A player has no valid moves available.
  - •Both players have made the same move 3 times in a row.

Game State Evaluation

•The predicate value(+GameState, +Player, -Value) is used to evaluate the game state from the perspective of a player and returns a value for the state.

•The value of a game state is calculated as follows:

•The distance is calculated by the distance/3 predicate, which uses the Manhattan distance formula to calculate the distance between two points.

•the sum of the 1/distances of each Player pieces from the playerGoal.

•the sum of the 1/distances of each Opponent pieces from the opponentGoal.

•the difference between the two sums is the value of the game state.

Computer Plays

Easy Bot Move Selection

•To determine its move, the bot first generates a random position on the board, then checks if it is a piece of its color and then utilizes the predicate valid_moves_piece((Turn, _, Board), (Player, X1, Y1), ListOfMoves), which generates a list of all valid moves available to that piece. From this list, the bot chooses a random move, thereby introducing an element of unpredictability to its gameplay strategy.
•The easy bot is not concerned with the value of a move, only that it is valid. Therefore, it does not use the value/3 predicate to evaluate the game state.
•If a captureMove is chosen, the bot will also choose a random empty position to place the captured piece.

Greedy Bot Move Selection

•The greedy bot picks the move that maximizes its position advantage, comparing the current GameState value with the potential new value after the move (new GameState).

•To determine its move, the greedy bot utilizes the predicate `mostValueableMove(GameState, Piece, Move)`.

•Selects the move with the highest score for the current player's piece based on move score calculations:

`0,9 * (1/Distance_from_Goal) + 0,1 * Can_Capture`

•If a `captureMove` is chosen, the bot will also choose the empty position that maximizes its position advantage to place the captured piece. Which is done by the predicate `furthestPosition(Player, Gamestate, X, Y)`, that selects the furthest empty position from the opponent's goal.
•It is done inside the `choose_move(+GameState, +Player, +Level, -Move)` predicate, and aftet that, it uses the `value/3` predicate to calculate the score before and after the potential move.

•If the new game state score is greater than or equal to the current score, it chooses that move.

Conclusions

The game developed has demonstrated functional efficacy in executing the designated board game with the AI component capable of interacting within the game's parameters as required. However, there were some aspects that couldn't be implemented due to time constraints and the complexity of the task:

•The first opportunity to improve lies within the heuristic approach of the greedy bot. The current implementation operates on a short-term outlook, making decisions based on the immediate optimal move. This approach lacks the strategic depth of considering future moves and their potential impact, which could be improved by integrating a lookahead algorithm pruning to forecast the consequences of current actions.

•Another aspect that could be improved is the user interface, specifically by adding the functionality to navigate back through menus during gameplay. This would enhance the user experience by providing more flexibility and allowing users to correct inadvertent selections without restarting the program.

•Lastly, the representation of the board as a traditional square grid could be transformed into a diagonal (losango) layout to better reflect the actual movements and distances between pieces on the board. However,

this change would be purely aesthetic and does not affect the functionality of the game.

These enhancements would form the basis of the roadmap for the next phase of development, with a focus on creating a more sophisticated AI opponent and a more user-friendly interface, thereby elevating the overall quality and engagement of the game.