Faculdade de Engenharia da Universidade do Porto

Push Fight

Practical Work 1

Programação Funcional e em Lógica
Class 14 - Push Fight 6

Luís Miguel Lima Tavares - up202108662@fe.up.pt Contribution: 50%
Rodrigo Campos Rodrigues - up202108847@fe.up.pt Contribution: 50%

Porto, 05/11/23

Table of Contents

Installation and Execution

To install the Push Fight game, it is initially necessary to download the files found in PFL_TP1_T14_PushFight6.zip and extract them. In the src directory, access the file push_fight.pl using the command line or directly through the UI of Sicstus Prolog 4.8. The game can be run on Windows and Linux operating systems and is started with the play/0 predicate.

Description of the Game

The object of Push Fight is to push one of your opponent's game pieces off the board.

Players take turns placing their five pieces (three with square tops, two with sphere tops) on their half of the board, with white placing first. White then takes the first turn, and players alternate turns after that. A turn consists of 0-2 moves, then a push of one space; a player can move any of their pieces orthogonally — changing direction if desired — any number of spaces as long as those spaces are empty. A player can push only with one of their square top pieces, and they push both friendly and opposing pieces one space. After pushing, the player places the red anchor piece on the piece that pushed; this piece cannot be pushed on the opponent's next turn.

Players continue taking turns until someone pushes an opponent's piece off either end of the board. (The sides are ridged, and pieces cannot be pushed over them.)

The rules and functioning of the game were consulted in BoardGameGeek.

Game Logic

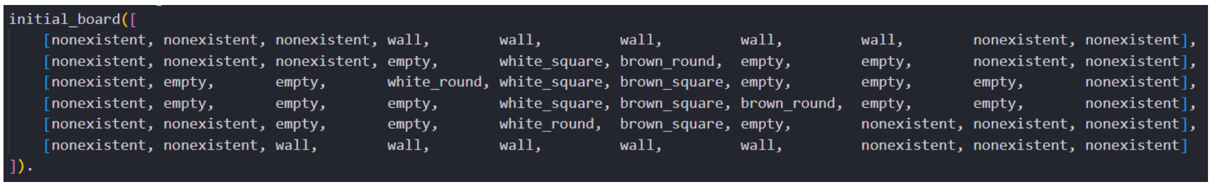
Internal Game State Representation

The internal game state in this Prolog program is represented by a list structure **GameState([Board, Player, MovesLeft, [WhitePiecesLeft, BrownPiecesLeft], Anchor, AnchorPosition])**. This structure encapsulates all the necessary components of the game's current status. The Board is represented as a list of lists, with each inner list representing a row on the game board and different atoms indicating the pieces. Player indicates the current player's turn, using atoms such as white or brown to signify which player is to move next. MovesLeft is an integer showing the number of moves the current player has left in their turn.

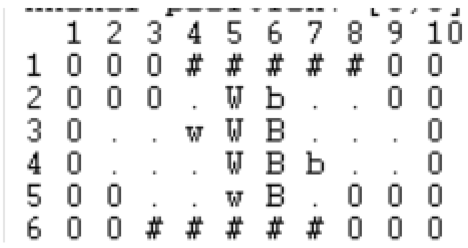
The **PiecesLeft sub-list [WhitePiecesLeft, BrownPiecesLeft]** contains the count of remaining pieces for each player, providing an easy way to determine if a player has pieces to place on the board or if the game is nearing completion. Each player has two types of pieces, the square pieces and the round pieces. Anchor represents a special cell that can only be on top of square pieces on the board, which has strategic importance in the game, and AnchorPosition specifies the exact location of this anchor on the game board.

The game board has 7 different atoms, 4 different atoms to represent the players' pieces (**white_round**, **white_square**, **brown_round**, **brown_square**). The **empty** atom represents any free space on the board, the **wall** atom represents a wall on the board, and, finally, the **nonexistent** atom represents spaces that are out of the board (these atoms are used to check if any piece is out of bounds).

Initial State of the Game

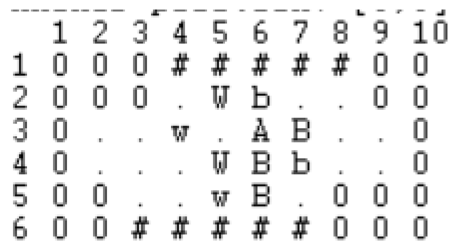


```
initial_board([
    [nonexistent, nonexistent, nonexistent, wall,         wall,         wall,         wall,         wall,        nonexistent, nonexistent],
    [nonexistent, nonexistent, nonexistent, empty,        white_square, brown_round,  empty,        empty,       nonexistent, nonexistent],
    [nonexistent, empty,       empty,       white_round,  white_square, brown_square, empty,        empty,       empty,       nonexistent],
    [nonexistent, empty,       empty,       empty,        white_square, brown_square, brown_round,  empty,       empty,       nonexistent],
    [nonexistent, nonexistent, empty,       empty,        white_round,  brown_square, empty,        nonexistent, nonexistent, nonexistent],
    [nonexistent, nonexistent, wall,        wall,         wall,         wall,         wall,         nonexistent, nonexistent, nonexistent]
]).
```



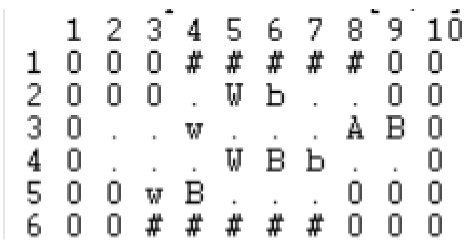
```
   1 2 3 4 5 6 7 8 9 10
1  0 0 0 # # # # # 0 0
2  0 0 0 . W b . . 0 0
3  0 . . w W B . . . 0
4  0 . . . W B b . . 0
5  0 0 . . w B . 0 0 0
6  0 0 # # # # # 0 0 0
```

Intermediate State of the Game



```
   1 2 3 4 5 6 7 8 9 10
1  0 0 0 # # # # # 0 0
2  0 0 0 . W b . . 0 0
3  0 . . w . A B . . 0
4  0 . . . W B b . . 0
5  0 0 . . w B . 0 0 0
6  0 0 # # # # # 0 0 0
```

Final State of the Game



```
   1 2 3 4 5 6 7 8 9 10
1  0 0 0 # # # # # 0 0
2  0 0 0 . W b . . 0 0
3  0 . . w . . . A B 0
4  0 . . . W B b . . 0
5  0 0 w B . . . 0 0 0
6  0 0 # # # # # 0 0 0
```

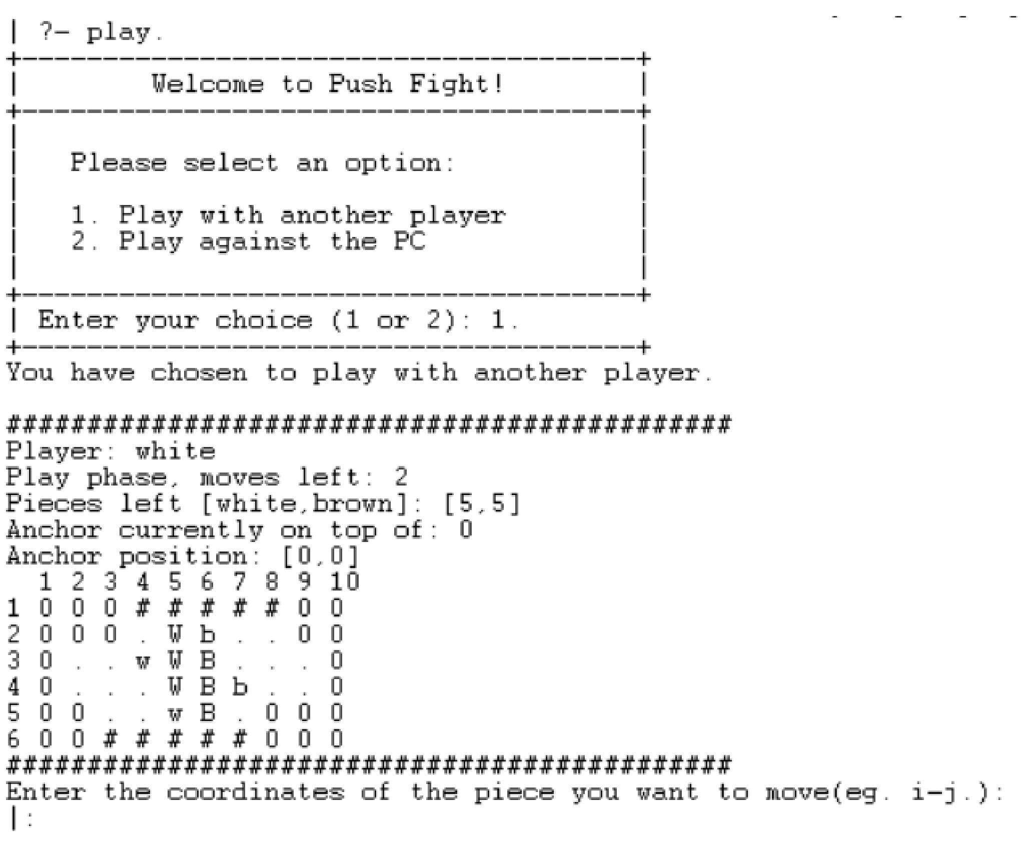
Game State Visualization

The game state visualization in this Prolog implementation is encapsulated within the **display_game/1** predicate, which serves as the central mechanism for displaying the current state to the user. This predicate takes the

GameState list as an argument, which contains the current board state, the active player, the number of moves left, pieces left for each player, and the anchor's position and status. It provides a user-friendly console-based visualization of the game state, starting with a decorative border, followed by key game information such as the current player, phase of the game (play or push phase), the number of pieces left for each player, and the position and status of the anchor.

The **display_board/1** predicate is responsible for showing the board layout, where it uses column and row identifiers to assist players in understanding the game board. Cells on the board are represented by different characters: walls by **#**, empty cells by **.**, player pieces by **w, W, b, B**, depending on the color and shape, and the anchor by **A**. The visual representation of the game board is intuitive, making use of different letters and symbols to easily distinguish between various types of cells.

The game starts with an engaging menu system introduced by the **play/0** predicate, which displays a stylized menu and prompts the user to choose between playing with another human or against the PC. Input validation is handled by **read_choice/1**, which repeatedly asks the user to enter a valid choice until one is provided. Upon receiving a valid input, the **perform_action/1** predicate proceeds with the game's initiation based on the user's choice.



```
| ?- play.
+---------------------------------------+
|        Welcome to Push Fight!         |
+---------------------------------------+
|                                       |
|   Please select an option:            |
|                                       |
|   1. Play with another player         |
|   2. Play against the PC              |
|                                       |
+---------------------------------------+
| Enter your choice (1 or 2): 1.
+---------------------------------------+
You have chosen to play with another player.

##############################################
Player: white
Play phase, moves left: 2
Pieces left [white,brown]: [5,5]
Anchor currently on top of: 0
Anchor position: [0,0]
  1 2 3 4 5 6 7 8 9 10
1 0 0 0 # # # # # 0 0
2 0 0 0 . W b . . 0 0
3 0 . . w W B . . . 0
4 0 . . . W B b . . 0
5 0 0 . . w B . 0 0 0
6 0 0 # # # # # 0 0 0
##############################################
Enter the coordinates of the piece you want to move(eg. i-j.):
|:
```

Move Validation and Execution

The **move/3** predicate is responsible for both validating and executing a move within a game. Here is the detailed explanation of what this predicate does:

- **Move Representation:** A move is represented as a list with the format **[FromRow, FromCol, ToRow, ToCol]**, which indicates the start position and the end position of a move on the game board.

- **Move Validation:** The code first extracts the piece at the **FromRow** and **FromCol** positions on the board. It then checks if this piece belongs to the current player by verifying if the piece is a member of the list of pieces associated with the **Player**.

- **Move Execution - Play Phase:** If the player still has moves left (**MovesLeft > 0**), the predicate checks if the **From** and **To** positions are the same. If they are, the move is considered a skip, and the game state is updated accordingly with one less move left. If the positions are different, the predicate then verifies if there is a clear path from the start to the end position using the **is_clear_path/5** predicate. If the path is clear, the piece is moved, and the board is updated, otherwise, an error message is shown, and the original game state is returned.

- **Move Execution - Push Phase:** When no moves are left, it enters the push phase. Here, it first ensures that the piece to be pushed is a square piece. Then it verifies if the push is valid by checking for a valid neighboring piece and that the push can be completed in the given direction with the **valid_neighbor_push/5** and **check_push_direction/5** predicates. If the push is valid, it is executed, the next player is determined with the **next_player/2** predicate, and the game state is updated to reflect the new board and the switch of turns.

- **Error Handling:** At every decision point, if the condition for a valid move is not met, the predicate outputs an error message indicating why the move was not valid and returns the original GameState.

List of Valid Moves

The **valid_moves/3** predicate in the game generates a list of possible moves for a selected player based on the current state of the game. This state includes the board layout, the active player, the number of moves left, the count of remaining pieces, and information about the game's anchor. The predicate differentiates between two distinct phases: the push stage (when **MovesLeft** is 0) and the regular move stage.

During the push stage, only the player's square pieces can push, and a move is considered valid if it involves pushing a piece adjacent to the player's own. The predicate iterates over all square pieces of the active player (white or brown), checking for potential pushes. For each piece, it determines if there's a neighboring piece that can be pushed and if the direction of the push is valid. A comprehensive list of such moves is generated using the **findall/3** construct, which compiles a list of all [FromRow, FromCol, ToRow, ToCol] moves that meet the criteria.

When it is not the push stage, the predicate considers all types of pieces the player controls. It uses **findall/3** to gather every move from the source to a destination where the path is clear, and the destination cell is empty. This allows the predicate to produce a list of all theoretically possible moves without executing them.

End of Game

The **game_over/2** predicate is a crucial component of the game logic, designed to evaluate whether the end conditions of the game have been met and, if so, to identify the winner. It takes the current game state as input, which includes the board configuration, the player whose turn it is, the number of moves left, the pieces left for both players, and the status of the anchor and its position. The predicate specifically checks the count of pieces left for each pla yer; if the white player has fewer than 5 pieces remaining, the game concludes with the brown player as the victor, indicated by setting Winner to brown and succeeding with true. Conversely, if the brown player's pieces fall below the threshold of 5, the white player is declared the winner. If neither of these conditions is met, the game is not over, and the predicate fails with false, indicating that the gameplay continues. After every push phase this predicate is called in order to check if the conditions for the game to conclude are met.

Computer Plays

Regarding the computer plays, we have only implemented the level 1 difficulty. In this difficulty the move made by the computer is made by choosing a random move from a list of valid moves available.

Conclusions

The Push Fight game was implemented with success in Prolog. It can be played in two modes: Player vs Player and Player vs Bot. When playing against the computer there is only a level of difficulty available. All the interactions are robust and assure the correct flow of the game.

The most challenging aspect of the project was the implementation of the push mechanic, where we had to change the position of various pieces at once.

With this project we consolidated the contents taught in class.

Bibliography

- https://boardgamegeek.com/boardgame/54221/push-fight