2º Trabalho Prático de PFL

Taacoca - Grupo 2 (3LEIC06)

- up202004735 - António Ferreira (50%)
- up202006137 - Guilherme Almeida (50%)

Installation and Execution

To play **Taacoca**, alongside the folder with the source code of the game, you need to have installed, on your computer, the 4.7.1 version of the **SICStus Prolog** interpreter. In order to load up the game, now in the interpreter, you'll need to consult the **"play.pl"** file, located in the source root directory:

```
?- consult(./play.pl).
```

Finally, to start the game, you'll have to run the predicate **play/0**, which will boot up the main menu of Taacoca:

```
?- play.
```

Game Description

Board

One of the characteristics of Taacoca that distinguishes itself from other board games is its **hexagonal board with 5 spaces/cells per side**. The **bottom row** is the home row of **player 1**, while the **top row** is the home row of **player 2**.

Initially, each player has 12 pieces. Here's a representation of initial state of the board:

```
    ABCDEFGHIJKLMNOPQ
     _________
1     |o|o|o|o|o|
2    |_|o|o|o|o|_|
3   |_|_|o|o|o|_|_|
4  |_|_|_|_|_|_|_|_|
5 |_|_|_|_|_|_|_|_|_|
6  |_|_|_|_|_|_|_|_|
7   |_|_|+|+|+|_|_|
8    |_|+|+|+|+|_|
9     |+|+|+|+|+|

    ABCDEFGHIJKLMNOPQ
```

Gameplay

Starting with player 1, players will take turns moving any three of their pieces one space forward (if a player has less than three pieces, then they must move all of the remaning pieces, if that's not possible, they lose the game).

The chosen pieces **do not need to be connected to each other** but they **must move in the same direction** (left or right). A player cannot move their pieces to a space/cell if it is **occupied with another piece of theirs.** If some of the target spaces/cells (or all of them) are occupied with enemy's pieces then the opponent's pieces in question are **captured and removed from the board.**

The first player to **reach the home row of the opponent** or **capture all of their pieces** wins. **No draws are possible in Taacoca.**

Game Logic

Game state internal representation

The **game state** is made up of the game board and the pieces of each player.

- On one hand, the **board** is represented by a **list of lists**: each list is a row on the board and each element of these lists represents a space/cell of the board. A **space** can be an **empty space ('_')**, a **piece of player 1 ('+')** or a **piece of player 2 ('o')**.
- On the other hand, the **pieces** of each player are handled using the predicate *piece/3*. This predicate will store information about which player it belongs to and its current X and Y coordinates.
- There's also many other predicates that are used to ensure the rules and logic of the game: **valid_spaces/2** (used to check if a piece can move to a certain cell or not), **valid_direction/1** (used to check if the user inputed a valid direction) and **next_piece/3** (used in the hard difficulty game modes, where the next X and Y coordinates of a space are stored, as well as the weight of that next move: **3 - only space available // 2 - enemy piece // 1 - empty space // 0 - not valid**).

Initial state

Board

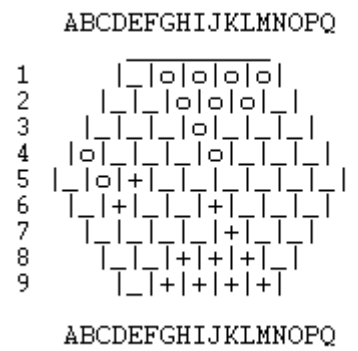
```
   ABCDEFGHIJKLMNOPQ
     _____________
1      |o|o|o|o|o|
2     |_|o|o|o|o|_|
3    |_|_|o|o|o|_|_|
4   |_|_|_|_|_|_|_|_|
5  |_|_|_|_|_|_|_|_|_|
6   |_|_|_|_|_|_|_|_|
7    |_|_|+|+|+|_|_|
8     |_|+|+|+|+|_|
9      |+|+|+|+|+|

   ABCDEFGHIJKLMNOPQ
```

Pieces left

```
piece(1, 5, 9).
piece(1, 7, 9).
piece(1, 9, 9).
piece(1, 11, 9).
piece(1, 13, 9).
piece(1, 7, 7).
piece(1, 6, 8).
piece(1, 8, 8).
piece(1, 10, 8).
piece(1, 12, 8).
piece(1, 9, 7).
piece(1, 11, 7).
piece(2, 5, 1).
piece(2, 7, 1).
piece(2, 9, 1).
piece(2, 11, 1).
piece(2, 13, 1).
piece(2, 6, 2).
piece(2, 8, 2).
piece(2, 10, 2).
piece(2, 12, 2).
piece(2, 7, 3).
piece(2, 9, 3).
piece(2, 11, 3).
```

Intermediate state

Board



Pieces left

```
piece(1, 7, 9).
piece(1, 9, 9).
piece(1, 11, 9).
piece(1, 13, 9).
piece(1, 8, 8).
piece(1, 10, 8).
piece(1, 12, 8).
piece(1, 11, 7).
piece(2, 7, 1).
piece(2, 9, 1).
piece(2, 11, 1).
piece(2, 13, 1).
piece(2, 8, 2).
piece(2, 10, 2).
piece(2, 12, 2).
piece(2, 9, 3).
piece(1, 10, 6).
piece(1, 4, 6).
piece(1, 5, 5).
piece(2, 2, 4).
piece(2, 3, 5).
piece(2, 10, 4).
```

Final state

Board

```
Congrats, player 1 won!

   ABCDEFGHIJKLMNOPQ
     _________
1      |+|o|o|o|_|
2     |+|_|o|o|_|_|
3    |_|_|_|o|+|_|_|
4   |_|_|_|_|_|_|_|o|
5  |_|_|_|_|_|_|o|_|o|_|
6   |_|o|_|_|_|_|_|_|
7    |_|_|_|_|+|_|_|
8     |_|+|+|+|+|_|
9      |_|_|+|+|+|

   ABCDEFGHIJKLMNOPQ
```

Pieces left

```
piece(1, 9, 9).
piece(1, 11, 9).
piece(1, 13, 9).
piece(1, 8, 8).
piece(1, 10, 8).
piece(1, 12, 8).
piece(1, 11, 7).
piece(2, 7, 1).
piece(2, 9, 1).
piece(2, 11, 1).
piece(2, 8, 2).
piece(2, 10, 2).
piece(2, 9, 3).
piece(2, 11, 5).
piece(1, 11, 3).
piece(2, 16, 4).
piece(2, 15, 5).
piece(2, 4, 6).
piece(1, 5, 1).
piece(1, 4, 2).
piece(1, 6, 8).
```

Game state visualization

The visual and interaction aspects of the game are focused on two main modules: the `play` module and the `board` module.

Play

This module is responsible for handling all of the aspects regarding the menus of the game. With that in mind, there's many predicates to highlight:

`main_menu`

- Predicate responsible for printing out the main menu of the game

```
=========================================================

Welcome to Taacoca!

1. Play
2. Rules
3. Exit

Please enter your choice:
|:
```

`menu_choice(+Choice)`

- Predicate responsible for handling the choice given by the user

`play_menu`

- Predicate responsible for printing out the play menu of the game (where the user can select which game mode they want to play)

```
1. Player vs. Player
2. Player vs. AI
3. AI vs. AI

Please enter your choice:
|:
```

`play_menu_choice(+Choice)`

- Predicate responsible for handling the choice given by the user. If necessary, depending on the game mode, the user will be asked to choose the level of difficulty and what player he wants to play as.

Board

The main focus of this module is to handle the logic regarding the visual aspects and manipulation of the game board.

`initial_state(-Board)`

- Predicate that gets the initial board of the game

`display_game(+GameState)`

- Predicate used to display the current game state/board of the game

Misc

Although they are included in other modules, there are other predicates that are also essential for user input and validation of the user data that should be mentioned:

`get_direction(-Dir)`

- Predicate in which the user specifies the direction he wants to move the pieces

`check_cords(+[XInput,YInput],+Player,-X,-Y)`

- Predicate thats checks if the user given coordenates are valid

```
choose_pieces_rec(+Gamestate,+Player,?Pieces,-ChosenPieces,?N)
```

- Recursive predicate where the user will chose which pieces they want to move

Move execution

During a user's turn, the process of executing a move is handled by the **move/3** predicate. In this, will be initially asked in which direction he wants to move the pieces (left or right).

```
move(+Gamestate,+Player,-NewGamestate)
```

Next, the number of iterations will be calculated (**N**), using the **get_number_plays/2** predicate. This variable represents the number of possible pieces a user can move during a turn (in every turn, the player moves 3 pieces. However, if they have less then 3 pieces, they can only move the remaining pieces).

```
get_number_plays(+Player,-N)
```

After that, the user will get to choose the N pieces he wants to move. This is done using the **choose_pieces/7** predicate, in which the user inputs the coordinates of the pieces, the coordinates are validated, the coordinates of the new spaces/cells are obtained and checked to see if they're valid and the board will be prepared for the move.

```
choose_pieces(+Gamestate,+Player,+Dir,+Pieces,-NewPieces,-NewGamestate,?N)
```

The actual movement of the pieces occurs in the **move_pieces/5** predicate:

```
move_pieces(+Gamestate,+Player,+Coordinates,-NewGamestate,?N)
```

Finally, to finish the turn, attacks (an enemy's piece is consumed during the move of one of the user's pieces) will be checked using the **check_attacks/2** predicate:

```
check_attacks(+Player,+Coordinates,?N)
```

Game Over

The strategy for checking if the game ended was to create the predicate **game_over/1**:

```
game_over(+Player)
```

The predicate will succeed if the game has ended.

Since the game has **two ways to win** (by reaching the enemy's side or by making the enemy run out of pieces), the predicate gameOver/1 needed to have a **rule for each win condition**.

For the first rule we defined the predicate **check_reached_other_side/1**, which used the predicate **findall/3** to find all pieces belonging to the player that are on the enemy's base. If this predicate returns a non-empty list, then the player has reached the enemy's side and the game is over.

For the second rule we defined the predicate **check_other_player_number_pieces/1**, which checks if the other player has any pieces left. This predicate uses the **value/2** predicate to get the number of pieces of a determinate player. If this player has no pieces, we conclude that the adversary won and the game is over.

Finally, the **game_over/1** predicate uses these two rules to determine if the game has ended by checking both conditions. If either of the two rules succeeds, then the game is over.

List of valid moves:

The first step for the creation of the bot was to create the predicate **valid_moves/4**:

```
valid_moves(+Board, +Player, +[X,Y], -PossiblePieces)
```

The predicate **valid_moves/4** is used to determine the possible set of moves for each piece, based on the player and the restrictions imposed. It takes into consideration the type of piece (empty space or enemy piece) that is present in the next position and assigns a weight to each move accordingly. If the next move is an empty space, the weight is 1. If the next move is the piece of the enemy player, the weight is 2. If there is only one possible move, that move is given the weight of 3 and the other move, which is invalid, is given the weight of 0.

The **valid_moves/4** predicate has multiple clauses to handle different cases, depending on the type of piece present in the next position and the player.

We also have the **get_valid_moves/6** predicate, which is used to find the valid moves for a given set of pieces in the AI vs AI mode. This predicate receives the current game state (Gamestate), the player, a list of the coordinates of the chosen pieces ([[X,Y]|Tail]), an accumulator for storing the valid moves (Acc), a variable to store the list of all valid possible next moves for each chosen piece (AllValidMoves), and a variable to store the number of iterations (N).

```
get_valid_moves(+Gamestate, +Player, +[[X,Y]|Tail], ?Acc,-AllValidMoves, ?N)
```

Once the **valid_moves/4** predicate has determined the possible moves and assigned weights to each position, the **get_valid_moves/6** predicate adds these moves to the accumulator (Acc) and continues to the next piece in the list. When all the chosen pieces have been processed, the ** gget_valid_moves/6** predicate returns the list of all valid possible

Game state evaluation

The strategy we decided to use for our game, since it has a continous flow and 2 defined ways of winning was to use the **value/2** predicate to check the amount of pieces of a certain player.

```
value(Player,Value)
```

Computer move

The strategy for deciding the computer moves was to create the predicate **choose_move/4**:

```
choose_move(+GameState, +Player, +Difficulty, -NewGamestate)
```

Note: `Difficulty` is a binary value passed which defines the difficulty of the game. `1` should be passed for the easy mode and `2` for the hard mode.

This predicate uses *move_pieces/5* and it repeats this process until N pieces that can move are selected.

- **Easy**:

    - The strategy envolved in an easy computer's turn is very similar to the one of the user. The only difference is in the method of choosing the pieces to move. In this case, an easy AI will choose N different random pieces using the **choose_pieces_computer/7** predicate. `choose_pieces_computer(+Gamestate,+Player,+Dir,+Pieces,-NewPieces,-NewGamestate,?N)`

- **Hard**:

    - In this game mode, the AI will use a greedy algorithm to determine the best N possible pieces to move. For this game, taking into consideration its rules and main objectives, we came to the conclusion that, during the process of choosing the pieces, the best ones are the ones closest to the enemy's home row - **choose_best_pieces/3**. `choose_best_pieces(+Player,-ChosenPieces,?N)`
    - Then, using the **get_valid_moves/6** predicate, all the possible moves, for each piece, will be obtained and stored in a list. In addition, the coordinates of each possible move is stored, as it was already said in the beginning, in a **next_piece/3** predicate, alongside the weight given to each move. `get_valid_moves(+Gamestate,+Player,+Coordinates,?Acc,-AllValidMoves,?N)`
    - To determine in which direction to move the pieces, in the **find_next_pieces/5** predicate, the sum of the weights of each direction will be calculated. If the weight of the next left pieces is bigger or equal to the total weight of the right next pieces, the pieces will move to the left. Otherwise, they will move to the right. `find_next_pieces(+Gamestate,+Player,+PossiblePieces,-NewPieces,?N)`

Conclusion

The board game *Taacoca* was successfully implemented in SICStus Prolog 4.7. The game can be played in 3 different modes:

- Player vs Player
- Player vs Computer
- Computer vs Computer

One of the main challenges that we faced during the development was the initial process of translating the game's logic into suitable Prolog code.

Additionally, while we had initially intended to include the option for players to choose a variable board size, we were unable to find a feasible method to implement this feature using our current board representation. Despite this limitation, the implemented version of the game is fully functional and provides an enjoyable gaming experience for players.

One potential area for improvement in the Taacoca game is the algorithm currently used by the computer player. As it stands, the algorithm only considers the immediate consequences of each move, leading to a less sophisticated playstyle that may not always result in the optimal outcome. To address this issue, we could consider implementing an alternative level with a more advanced algorithm such as minimax, which takes into account the potential long-term consequences of each move in order to maximize the chances of winning the game. This would add a greater level of depth and challenge to the game, making it more engaging for players.

Sources

- https://www.iggamecenter.com/en/rules/taacoca
- https://sicstus.sics.se/documentation.html