README.md

Functional and Logic Programming - 1st Pratical Assignment

Game Theme

[Tactigon Board Game]

Group Description

Group Name: **Tactigon_4**

Group Members:

- Marco André Pereira da Costa - up202108821 - 50% contribution
- Tiago Filipe Castro Viana - up201807126 - 50% contribution

Installation and Execution

In order to install and execute the game, you must download ***PFL_TP1_T03_Tactigon_4.ZIP*** and extract it. Then, inside the *src* directory, consult the ***main.pl*** file through SICStus Prolog 4.8. Finally, run the following command:

```
?- play.
```

To improve the game experience, we recommend maximizing the SICStus Prolog window and changing the font as follows:

- Font: Consolas
- Style: Normal
- Size: 11

The game is available for Windows and Linux.

Game Description

Tactigon is a board game for two players, played on an irregular hexagon board. Each player starts with 13 pieces. The game starts with a default board configuration, and the players take turns moving their pieces and resolving any combat that may result from that movement.

General movement rules:

- Pieces can move along any path and in any direction up to their maximum spaces.
- Pieces can't jump over other pieces.*
- Maximum spaces are equal to the number of sides of the piece.**

* - This rule can be changed by applying the advanced rule 1.
** - This rule can be changed by applying the advanced rule 2.

Pieces can be one of this four types:

- **Circle** - 1 side
- **Triangle** - 3 sides
- **Square** - 4 sides
- **Pentagon** - 5 sides

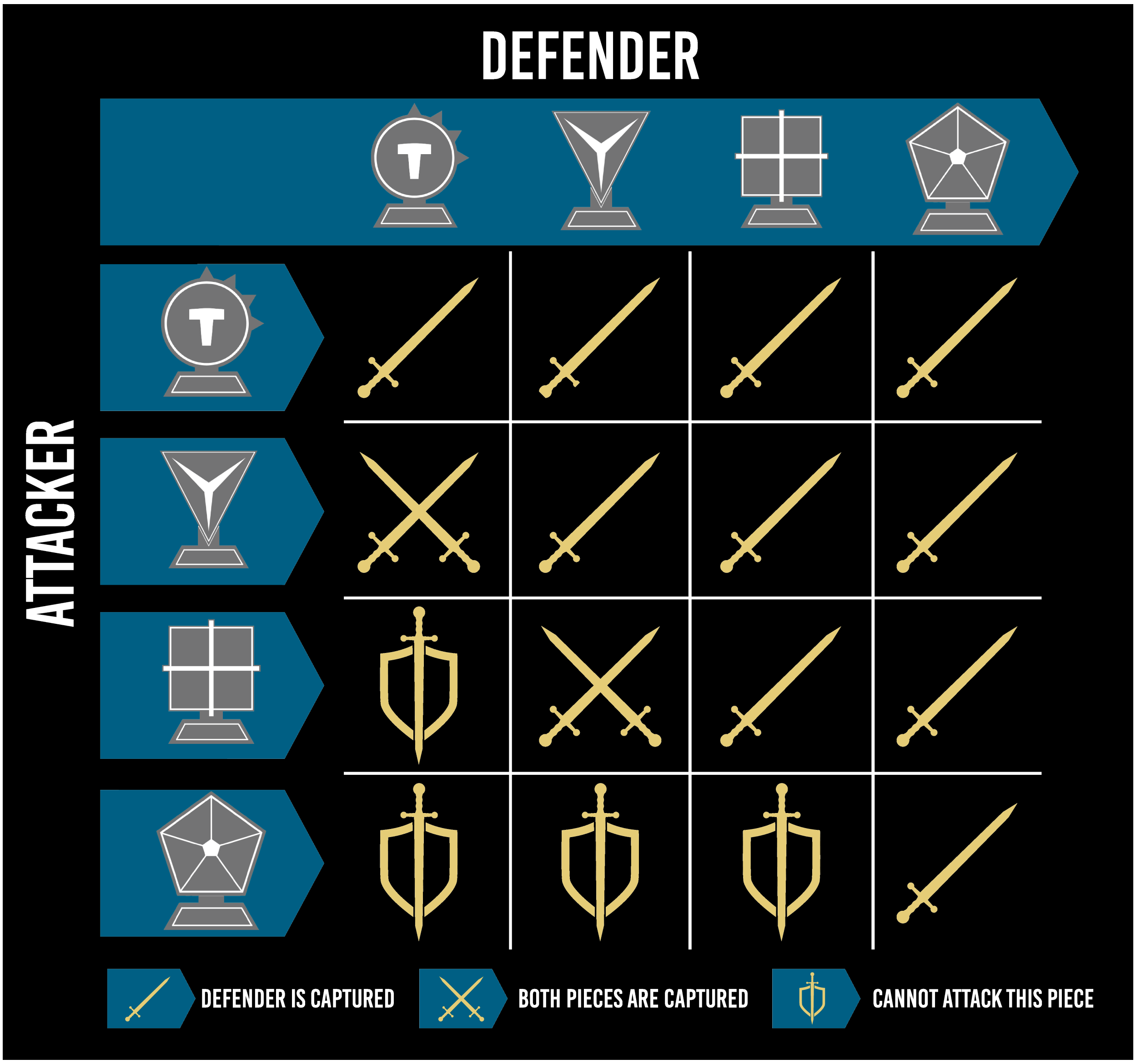

Figure 1 - Combat Table

The pieces can combat the opponent's pieces by moving to a tile occupied by an opposing piece.

Combat has two outcomes:

- The defending piece is captured, marked by a sword icon in Figure 1.
- Both pieces are captured, marked by a two swords crossing icon in Figure 1.

The sword and shield icon represents certain combats that can't occur in the game. Captured pieces are removed from the board and can't be used for the rest of the game.

The victory can be achieved by **capturing the opponent's pentagon** or by occupying **both gold tiles** at the **end** of the **opponent's turn**.

Two optional advanced rules can be applied to the game:

1. *Square pieces can jump over other pieces, except for opposing squares. A "jumped" tile still counts towards the piece's move limit.*

2. *Pieces that start a turn on a gold tile can move an additional space on that turn.*

For more information about the game, please consult the official website.
For more information about the game rules, please consult the How to Play or the Rulebook.

Game Logic

Internal Game State Representation

Board - The board is represented by a list of *Positions*. Each *Position* is represented by 2 elements: a *Piece* and the *Tile* where the *Piece* is located. Each *Tile* consists of the coordinates (X, Y) on the board. The minimum and maximum values for the X coordinate is defined for each line, and there can only be tiles inside those limits. Finally, the board also has *Gold Tiles*, which are represented by the corresponding (X, Y) coordinates on the board with the predicate **gold_tile/2**.

Player - The game has only two players, cian and red, represented by the corresponding atoms. The first player to move is chosen randomly, and after each turn, the current player is changed to the other player using the **other_player/2** predicate.

The **GameState** is represented by a list with the **Board** and the current **Player** at a given time in the game. The **GameState** does not contain a list of pieces that each player has captured, since they are removed from the board and can't be used for the rest of the game.

This is the representation of the board in the **initial** game state, where each piece is located in its starting position:

```
% board(+State, -Board)
% Unifies Board with the board at the current State for starting a game or for de
board(initial,
[
    position(cian-circle-1, tile(3, 0)),
    position(cian-circle-2, tile(1, 1)),
    position(cian-square-1, tile(2, 1)),
    position(cian-triangle-1, tile(3, 1)),
    position(cian-square-2, tile(4, 1)),
    position(cian-circle-3, tile(5, 1)),
    position(cian-triangle-2, tile(2, 2)),
    position(cian-pentagon-1, tile(3, 2)),
    position(cian-triangle-3, tile(4, 2)),
    position(cian-circle-4, tile(1, 3)),
    position(cian-square-3, tile(3, 3)),
    position(cian-circle-5, tile(5, 3)),
    position(cian-circle-6, tile(3, 4)),
    position(red-circle-1, tile(3, 6)),
```

```prolog
        position(red-circle-2, tile(1, 7)),
        position(red-triangle-1, tile(2, 7)),
        position(red-square-1, tile(3, 7)),
        position(red-triangle-2, tile(4, 7)),
        position(red-circle-3, tile(5, 7)),
        position(red-square-2, tile(2, 8)),
        position(red-pentagon-1, tile(3, 8)),
        position(red-square-3, tile(4, 8)),
        position(red-circle-4, tile(1, 9)),
        position(red-triangle-3, tile(3, 9)),
        position(red-circle-5, tile(5, 9)),
        position(red-circle-6, tile(3, 10))
    ]
    ).
```

board.pl

```
   |X |X0|X1|X2|X3|X4|X5|X6|  |
---|--------------------------|---
Y  |            __            |Y
Y0 |         __/CC\__         |Y0
Y0 |      __/  \__/  \__      |Y0
Y1 |     /CC\__/CT\__/CC\     |Y1
Y1 |     \__/CS\__/CS\__/     |Y1
Y2 |     /  \__/CP\__/  \     |Y2
Y2 |     \__/CT\__/CT\__/     |Y2
Y3 |     /CC\__/CS\__/CC\     |Y3
Y3 |     \__/  \__/  \__/     |Y3
Y4 |   __/  \__/CC\__/  \__   |Y4
Y4 |  /  \__/  \__/  \__/  \  |Y4
Y5 |  \__/GT\__/  \__/GT\__/  |Y5
Y5 |  /  \__/  \__/  \__/  \  |Y5
Y6 |  \__/  \__/RC\__/  \__/  |Y6
Y6 |     \__/  \__/  \__/     |Y6
Y7 |     /RC\__/RS\__/RC\     |Y7
Y7 |     \__/RT\__/RT\__/     |Y7
Y8 |     /  \__/RP\__/  \     |Y8
Y8 |     \__/RS\__/RS\__/     |Y8
Y9 |     /RC\__/RT\__/RC\     |Y9
Y9 |     \__/  \__/  \__/     |Y9
Y10|        \__/RC\__/        |Y10
Y10|           \__/           |Y10
---|--------------------------|---
   |X |X0|X1|X2|X3|X4|X5|X6|  |
```

Figure 2 - Initial Board

This is a possible representation of the board in an **intermediate** game state. The pieces are located in different positions than the initial game state, some of the pieces were captured, but no player has won yet, since both players still have their pentagon and the gold tiles [(1,5) and (5,1)] are not both occupied by the same player:

```prolog
% board(+State, -Board)
% Unifies Board with the board at the current State for starting a game or for de
board(intermediate,
[
    position(cian-circle-1, tile(4, 5)),
    position(cian-square-1, tile(5, 7)),
    position(cian-triangle-1, tile(5,2)),
    position(cian-circle-3, tile(5, 1)),
    position(cian-pentagon-1, tile(3, 0)),
    position(cian-triangle-3, tile(0, 5)),
    position(cian-circle-6, tile(5, 5)),
    position(red-circle-1, tile(3, 2)),
    position(red-square-1, tile(5, 6)),
    position(red-pentagon-1, tile(3, 5)),
    position(red-circle-4, tile(2, 3)),
    position(red-triangle-1, tile(3, 9)),
    position(red-circle-6, tile(1, 5))
]
).
```

board.pl

```
   |X |X0|X1|X2|X3|X4|X5|X6|  |
---|--------------------------|---
Y  |            __            |Y
Y0 |         __/CP\__         |Y0
Y0 |      __/  \__/  \__      |Y0
Y1 |     /  \__/  \__/CC\     |Y1
Y1 |     \__/  \__/  \__/     |Y1
Y2 |     /  \__/RC\__/CT\     |Y2
Y2 |     \__/  \__/  \__/     |Y2
Y3 |     /  \__/  \__/  \     |Y3
Y3 |     \__/RC\__/  \__/     |Y3
Y4 |   __/  \__/  \__/  \__   |Y4
Y4 |  /  \__/  \__/  \__/  \  |Y4
Y5 |  \__/RC\__/RP\__/CC\__/  |Y5
Y5 |  /CT\__/  \__/CC\__/  \  |Y5
Y6 |  \__/  \__/  \__/RS\__/  |Y6
Y6 |     \__/  \__/  \__/     |Y6
Y7 |     /  \__/  \__/CS\     |Y7
Y7 |     \__/  \__/  \__/     |Y7
Y8 |     /  \__/  \__/  \     |Y8
Y8 |     \__/  \__/  \__/     |Y8
Y9 |     /  \__/RT\__/  \     |Y9
Y9 |     \__/  \__/  \__/     |Y9
Y10|        \__/  \__/        |Y10
Y10|           \__/           |Y10
---|--------------------------|---
   |X |X0|X1|X2|X3|X4|X5|X6|  |
```

Figure 3 - Intermediate Board

And finally, a possible representation of the board in the **final** game state, where the cian player has won the game by capturing the red player's pentagon:

```
% board(+State, -Board)
% Unifies Board with the board at the current State for starting a game or for de
board(final,
[
    position(cian-circle-1, tile(5, 1)),
    position(cian-pentagon-1, tile(5, 6)),
    position(red-circle-1, tile(3, 5))
]
).
```

board.pl

```
   |X |X0|X1|X2|X3|X4|X5|X6|  |
---|--------------------------|---
Y  |            __            |Y
Y0 |         __/  \__         |Y0
Y0 |      __/  \__/  \__      |Y0
Y1 |     /  \__/  \__/CC\     |Y1
Y1 |     \__/  \__/  \__/     |Y1
Y2 |     /  \__/  \__/  \     |Y2
Y2 |     \__/  \__/  \__/     |Y2
Y3 |     /  \__/  \__/  \     |Y3
Y3 |     \__/  \__/  \__/     |Y3
Y4 |   __/  \__/  \__/  \__   |Y4
Y4 |  /  \__/  \__/  \__/  \  |Y4
Y5 |  \__/GT\__/RC\__/GT\__/  |Y5
Y5 |  /  \__/  \__/  \__/  \  |Y5
Y6 |  \__/  \__/  \__/CP\__/  |Y6
Y6 |     \__/  \__/  \__/     |Y6
Y7 |     /  \__/  \__/  \     |Y7
Y7 |     \__/  \__/  \__/     |Y7
Y8 |     /  \__/  \__/  \     |Y8
Y8 |     \__/  \__/  \__/     |Y8
Y9 |     /  \__/  \__/  \     |Y9
Y9 |     \__/  \__/  \__/     |Y9
Y10|        \__/  \__/        |Y10
Y10|           \__/           |Y10
---|--------------------------|---
   |X |X0|X1|X2|X3|X4|X5|X6|  |
```

Figure 4 - Final Board

Game State Visualization

In the main menu, the user can choose to start a new game, change settings or exit the game.

```
% menu/0
% Displays the menu and processes the user input. 1 to start the game, 2 to chang
menu :-
    display_menu,
    get_option(1, 3, 'Select an option', 'option', Option),
    processMenuOption(Option).

% processMenuOption(+Option)
% Processes the user input
processMenuOption(1) :-
    board_size(_, Size),
    initial_state(Size, [Board, Player]),
    game_loop([Board, Player]),
    !.

processMenuOption(2) :-
    change_settings,
    !,
    menu.

processMenuOption(3) :-
    clear_screen,
    !.
```

main.pl

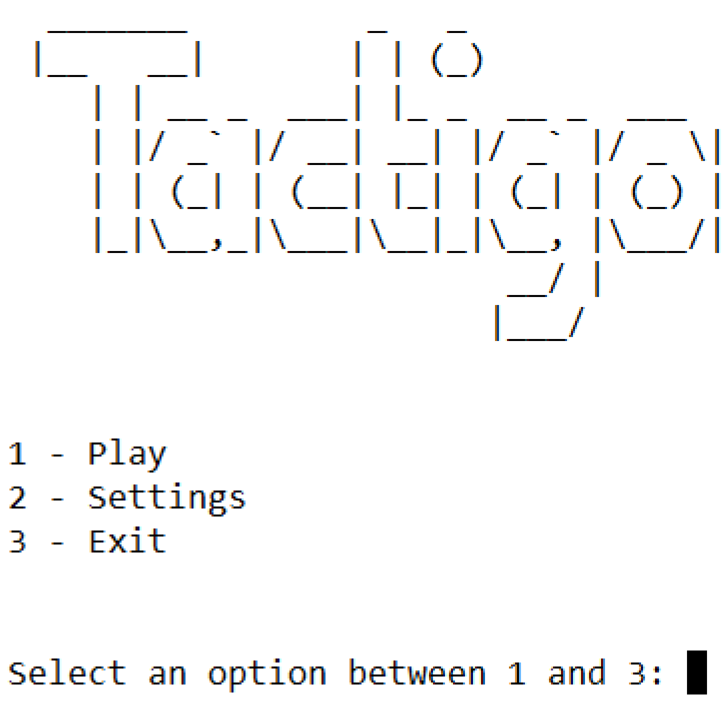

Figure 5 - Menu

The input is validated using the **get_option/5** predicate, which asks the user for an option between two values and reads the input. If the input is valid, the predicate returns the input. If not, the predicate asks the user for a new input. The request and the error messages are defined by the *Objective* and *Error* elements, respectively:

```
% get_option(+MinValue, +MaxValue, +Objective, +Error, -Option)
% Given an objective, unifies Option with the value given by user input between M
get_option(MinValue, MaxValue, Objective, _, Option):-
    format('~a between ~d and ~d: ', [Objective, MinValue, MaxValue]),
    read_number_input(Option),
    between(MinValue, MaxValue, Option),
    !.

get_option(MinValue, MaxValue, Objective, Error, Option):-
    format('Invalid ~a.~n', [Error]),
    get_option(MinValue, MaxValue, Objective, Error, Option).
```

utils.pl

If the user chooses to start a new game, the game will start with the defined settings. The predicate **initial_state/2** is responsible for creating the initial game state, based on the given board size.

```
% initial_state(+Size, -GameState)
% Returns the initial game state for a given board size
initial_state(Size, [Board, Player]) :-
    board_size(_, Size),
    !,
    board(initial, Board),
    findall(P, player(P), Players),
    random_member(Player, Players).

initial_state(Size, [Board, Player]) :-
    create_new_board(Size),
    !,
    board(initial, Board),
    findall(P, player(P), Players),
    random_member(Player, Players).
```

logic.pl

This predicate verifies if the board size is the current board size defined in the settings. If so, the initial game state is created using the initial board. If not, a new board is created using the **create_new_board/1** predicate and the initial game state is created using the new board.

E.g.:

```
% create_new_board(+Size)
% Creates a new board with Size lines, with a default number of columns for that
create_new_board(11) :-
    clear_board,
    assert(board_size(7, 11)),
    assert_list([
        line(0, 2, 4),
        line(1, 1, 5),
        line(2, 1, 5),
        line(3, 1, 5),
        line(4, 0, 6),
        line(5, 0, 6),
        line(6, 1, 5),
        line(7, 1, 5),
        line(8, 1, 5),
        line(9, 1, 5),
        line(10, 3, 3)
    ]),
    assert_list([
        gold_tile(1, 5),
        gold_tile(5, 5)
    ]),
    assert(board(initial,
    [
        ...
    ])),
    assert(board(intermediate,
    [
        ...
    ])),
    assert(board(final,
    [
        ...
    ])).
```

board.pl

With the game state created, the game loop starts. The **display_game/1** predicate is called and board is drawn. The board is displayed using the **draw_board/1** predicate, which displays the board in the terminal, with the pieces in their current positions.

```
% display_game(+GameState)
% Displays the game and all its elements
display_game([Board, _]) :-
    clear_screen,
    nl,
    draw_board(Board),
```

```
    display_legend,
    nl,nl.
```

interface.pl

The **draw_board/1** predicate divides the board in 3 parts:

1. **Header** - Displays the header of the board, with the column numbers.
2. **Board** - Displays the board and pieces of the board, with the lines numbers.
3. **Footer** - Displays the footer of the board, with the column numbers.

```
% draw_board(+Board)
% Draws the board
draw_board(Board) :-
    board_size(_, Y),
    MaxY is 2*Y - 1,
    draw_header,                    % '   |X |X0...|   ', nl, '---|-----...|---',
    draw_first_line(Board),
    draw_board_aux(Board, 0, MaxY),
    draw_footer.                    % '---|-----...|---', nl, '   |X |X0...|   ',
```

board.pl

The predicates **draw_board_aux/3** is responsible for drawing the board and pieces of the board via **draw_board_line/2**. The **draw_first_line/1** and **draw_board_line/2** predicates are similar. The first one draws the first line of the board, which is different from the other lines, since it has no tiles, but only the top underscores of the tiles.

Each actual line of the board takes 2 lines in the terminal. The program builds a list of "states" for each tile in the line.

```
% build_line(+Board, +CurrentY, +Y, -Line)
% Builds a Line, which is a list of draw predicates that represent a part of line
build_line(Board, CurrentY, Y, Line) :-
    build_line(Board, CurrentY, Y, 0, [draw(start, _)], Line).

% build_line(+Board, +CurrentY, +Y, +X, +Aux, -Line)
% Builds a Line, which is a list of draw predicates that represent a part of line
build_line(_, _, _, X, Aux, Line) :-
    board_size(X, _),
    append(Aux, [draw(none, _)], Line),
    !.

build_line(Board, CurrentY, Y, X, Aux, Line) :-
    \+ tile(X, CurrentY),
    1 is Y mod 2,
    1 is X mod 2,
```

```
    NY is CurrentY + 1,
    tile(X, NY),
    append(Aux, [draw(startBottom, _)], Aux1),
    X1 is X + 1,
    !,
    build_line(Board, CurrentY, Y, X1, Aux1, Line).

build_line(Board, CurrentY, Y, X, Aux, Line) :-
    \+ tile(X, CurrentY),
    0 is Y mod 2,
    0 is X mod 2,
    PY is CurrentY - 1,
    tile(X, PY),
    append(Aux, [draw(bottom, _)], Aux1),
    X1 is X + 1,
    !,
    build_line(Board, CurrentY, Y, X1, Aux1, Line).

build_line(Board, CurrentY, Y, X, Aux, Line) :-
    \+ tile(X, CurrentY),
    append(Aux, [draw(none, _)], Aux1),
    X1 is X + 1,
    !,
    build_line(Board, CurrentY, Y, X1, Aux1, Line).

build_line(Board, CurrentY, Y, X, Aux, Line) :-
    0 is Y mod 2,
    0 is X mod 2,
    !,
    PY is CurrentY - 1,
    (
        tile(X, PY) -> append(Aux, [draw(bottom, _)], Aux1) ;
        append(Aux, [draw(startBottom, _)], Aux1)
        ),
    X1 is X + 1,
    build_line(Board, CurrentY, Y, X1, Aux1, Line).

build_line(Board, CurrentY, Y, X, Aux, Line) :-
    0 is Y mod 2,
    1 is X mod 2,
    !,
    tile_to_string(Board, tile(X, CurrentY), String),
    append(Aux, [draw(top, String)], Aux1),
    X1 is X + 1,
    build_line(Board, CurrentY, Y, X1, Aux1, Line).

build_line(Board, CurrentY, Y, X, Aux, Line) :-
    1 is Y mod 2,
    0 is X mod 2,
    !,
    tile_to_string(Board, tile(X, CurrentY), String),
    append(Aux, [draw(top, String)], Aux1),
    X1 is X + 1,
```

```
    build_line(Board, CurrentY, Y, X1, Aux1, Line).

build_line(Board, CurrentY, Y, X, Aux, Line) :-
    1 is Y mod 2,
    1 is X mod 2,
    !,
    append(Aux, [draw(bottom, _)], Aux1),
    X1 is X + 1,
    build_line(Board, CurrentY, Y, X1, Aux1, Line).
```

board.pl

After that, the predicate **draw_hexagons/2** is responsible for using that list of "states" to draw the line in the terminal, with specific characters for each transition and each state.

```
% draw_hexagons(+Line, +LastState)
% Draws the hexagons of a line, and updates the LastState
draw_hexagons([], _) :- !.

draw_hexagons([H|T], LastState) :-
    draw_hexagon(LastState, H),
    draw_hexagons(T, H).

% draw_hexagon(+LastState, +State)
% Logic for drawing a hexagon, depending on the LastState and the State
draw_hexagon(draw(top, _), draw(none, _)) :-
    write('\\  '),
    !.

draw_hexagon(draw(bottom, _), draw(none, _)) :-
    write('/  '),
    !.

draw_hexagon(_, draw(none, _)) :-
    write('   '),
    !.

draw_hexagon(_, draw(start, _)) :-
    write('  '),
    !.

draw_hexagon(_, draw(bottom, _)) :-
    write('\\__'),
    !.

draw_hexagon(_, draw(top, none)) :-
    write('/  '),
    !.
draw_hexagon(_, draw(top, PrintType)) :-
    format('/~w', [PrintType]),
```

```
    !.

draw_hexagon(draw(top, _), draw(startBottom, _)) :-
    write('\\__'),
    !.

draw_hexagon(_, draw(startBottom, _)) :-
    write(' __'),
    !.
```

board.pl

During the game, the move input is validated using the **ask_move/2** predicate, which asks the user for two pairs of coordinates and reads the input. During the process, there are error messages and a possibility to cancel the move input. To get the coordinates, the predicate **get_move_input/1** is used, which asks for coordinates in the format X-Y and reads the input:

```
% get_move_input(-Coordinates)
% Reads a move from user input, in format X-Y, and unifies it with Coordinates
get_move_input(X-Y) :-
    read_number_del(X, 45),
    read_number_del(Y, 10),
    !.
```

utils.pl

The **read_number_del/2** predicate reads a number until a delimiter is found.

If the user chooses to change settings, the user is asked to select:

1. **Board Size**
2. **Cian Difficulty**
3. **Red Difficulty**
4. **Advanced Rules**

```
Select an option between 1 and 3: 2
Board Size:
1 - 11 lines, 7 columns (Default board size)
2 - 13 lines, 9 columns
3 - 15 lines, 11 columns
Select an option between 1 and 3: 1         <--- 1.
Player cian is:
1 - Level 1 Bot (Random)
2 - Level 2 Bot (Greedy)
3 - Human
Select an option between 1 and 3: 3         <--- 2.
Player red is:
```

```
1 - Level 1 Bot (Random)
2 - Level 2 Bot (Greedy)
3 - Human
Select an option between 1 and 3: 3         <--- 3.
Advanced Rules:
1 - Square pieces can jump over other pieces, except for opposing squares. A "jum
2 - Pieces that start a turn on a gold tile can move an additional space on that
Options:
1 - Advanced Rule 1
2 - Advanced Rule 2
3 - Both Advanced Rules
4 - None
Select an option between 1 and 4: 4         <--- 4.
```

Move Validation and Execution

The game runs in a loop where each iteration corresponds to a turn. The only stop condition of this loop is the victory of one of the players:

```
% game_loop(+GameState)
% Main game loop
game_loop(GameState) :-
    game_over(GameState, Winner),
    !,
    display_game(GameState),
    display_winner(Winner),
    !,
    menu.

game_loop(GameState) :-
    display_game(GameState),
    process_turn(GameState, NewGameState),
    !,
    game_loop(NewGameState).
```

main.pl

The **process_turn/2** predicate is responsible for processing the turn of the current player. If the current player is human, this predicate will ask for a move, validate the input and call the predicate **move/3**, if the input is valid. If the chosen move is not valid, the **invalid_move/0** predicate prints a warning message in the terminal and the user is asked to choose another move (this is repeated until the chosen move is valid). If the current player is a bot, this predicate will choose a valid move (depending on the difficulty level) and make the move:

```
% process_turn(+GameState, -NewGameState)
% Processes the turn of the current player
```

```
process_turn([Board, Player], [NewBoard, NewPlayer]) :-
    difficulty(Player, 3), % Human player
    !,
    repeat,
    invalid_move, % Display an invalid move message if the move is invalid
    get_move([Board, Player], OX-OY-DX-DY), % Get a move from the user
    move([Board, Player], OX-OY-DX-DY, [NewBoard, NewPlayer]),
    !.

process_turn([Board, Player], [NewBoard, NewPlayer]) :-
    difficulty(Player, Difficulty), % Computer player
    !,
    choose_move([Board, Player], Player, Difficulty, OX-OY-DX-DY), % Get a move f
    move_aux([Board, Player], OX-OY-DX-DY, [NewBoard, NewPlayer]),
    !.
```

main.pl

The **get_move/2** predicate is responsible for asking the user for a move and reading the input.

The **move/3** predicate is responsible for validating the move, executing the move if it is valid, and returning the new game state. This predicate starts by validating the move using the **validate_move/2** predicate. If the move is valid, the predicate **move_aux/3** is called to execute the move:

```
% move(+GameState, +Move, -NewGameState)
% Validates a move, makes the move when valid and returns the new game state
move([Board, Player], OX-OY-DX-DY, [NewBoard, NewPlayer]) :-
    validate_move([Board, Player], OX-OY-DX-DY), % Check if the move is valid
    move_aux([Board, Player], OX-OY-DX-DY, [NewBoard, NewPlayer]).
```

logic.pl

The **validate_move/2** predicate is responsible for validating the move. It starts by checking if the chosen piece is, in fact, on the board and if it belongs to the current player:

```
% validate_move(+GameState, +Move)
% Checks if a move is valid
validate_move([Board, Player], OX-OY-DX-DY) :-
    member(position(Piece, tile(OX, OY)), Board), % Check if the piece exists in
    piece_info(Piece, Player, _), % Check if the piece belongs to the player
    valid_move_for_piece([Board, Player], Piece, OX-OY-DX-DY).
```

logic.pl

After that, the predicate **valid_move_for_piece/3** is called to check if the move is valid for the chosen piece.

If the **advanced rule 2** is applied, the predicate **valid_move_for_piece/3** will check if the chosen piece is on a gold tile, in order to determine if the piece can move an additional space:

```
% valid_move_for_piece(+GameState, +Piece, +Move)
% Checks if a move is valid for a particular piece
valid_move_for_piece([Board, Player], Piece, OX-OY-DX-DY) :-
    rules(2),
    gold_tile(OX, OY),
    piece_info(Piece, _, Type),
    movement(Type, N),  % N is the maximum number of steps for this type of piece
    N1 is N + 1, % N1 is the maximum number of steps increases by 1 because of ad
    valid_move_dfs([Board, Player], N1, Piece, DX-DY, OX-OY).
```

logic.pl

If the **advanced rule 2** is not applied, this predicate starts by checking the type of the chosen piece, in order to determine the maximum number of spaces that the piece can move:

```
valid_move_for_piece([Board, Player], Piece, OX-OY-DX-DY) :-
    piece_info(Piece, _, Type), % Get the type of the piece
    movement(Type, N),  % N is the maximum number of steps for this type of piece
    valid_move_dfs([Board, Player], N, Piece, DX-DY, OX-OY).
```

logic.pl

After determining the maximum number of spaces that the piece can move, the predicate **valid_move_dfs/5** is called to check if the move is valid for the chosen piece. This predicate uses a depth-first search algorithm to check if the move is valid. The depth-first search algorithm starts by checking if the chosen piece can move to the destination tile in a single step. If it can, there are two possible outcomes: the destination tile is empty or the destination tile is occupied by an opposing piece. If the destination tile is empty, the move is valid. If the destination tile is occupied by an opposing piece, the move is valid if the combat is possible (i.e., the current player's piece can capture the opposing piece or the combat ends in a draw where both pieces are captured):

```prolog
% valid_move_dfs(+GameState, +N, +Piece, +DestinationTile, +CurrentTile)
% Checks if a move is valid for a particular piece using DFS, being N the number
valid_move_dfs([Board, _], N, _, DX-DY, CX-CY) :-
    N > 0,
    adjacent(tile(CX, CY), tile(DX, DY)), % Check if the destination tile is adja
    \+ member(position(_, tile(DX, DY)), Board). % Check if the destination tile

valid_move_dfs([Board, Player], N, Piece, DX-DY, CX-CY) :-
    N > 0,
    adjacent(tile(CX, CY), tile(DX, DY)), % Check if the destination tile is adja
    member(position(Defender, tile(DX, DY)), Board),
    other_player(Player, DefenderPlayer),
    piece_info(Defender, DefenderPlayer, DefenderType),  % Check if the destinati
    piece_info(Piece, Player, Type),
    combat(Type, DefenderType, _). % Check if the piece can attack the opponent's
```

logic.pl

If the chosen piece can't move to the destination tile in a single step, the depth-first search algorithm will move to an adjacent tile of the current piece and call itself recursively, decreasing the number of steps left to take by 1. This process will continue until the maximum number of spaces that the piece can move is reached (when N is equal to 0) and all possible moves are checked, in depth. Within the recursive calls, if there are still moves left to take, the algorithm will check if the destination tile is adjacent to the current tile and if it is empty or occupied by an opposing piece. If the destination tile is empty, the move is valid. If the destination tile is occupied by an opposing piece and the combat is possible, the move is valid. If there are no more moves left to take, the move is invalid.

If the **advanced rule 1** is applied, the predicate **valid_move_dfs/5** will also check if the chosen piece is a square, in order to determine if the piece can jump over other pieces (except for opposing squares):

```prolog
% Special case for the square piece when advanced rule 1 is enabled.
% Square pieces can jump over other pieces, except for opposing squares.
% A "jumped" tile still counts as a step.
valid_move_dfs([Board, Player], N, Player-square-Id, DX-DY, CX-CY) :-
    rules(1),
    N > 0,
    N1 is N - 1,
    other_player(Player, Opponent), % Get the opponent of the player
    findall(X-Y, adjacent(tile(CX, CY), tile(X, Y)), AdjacentTiles),  % Get all t
    member(CX1-CY1, AdjacentTiles), % Get an adjacent tile
    \+ member(position(Opponent-square-_, tile(CX1, CY1)), Board), % Check if the
    valid_move_dfs([Board, Player], N1, Player-square-Id, DX-DY, CX1-CY1).
```

logic.pl

By default, pieces can't jump other pieces. Therefore, if the **advanced rule 1** is not applied, the algorithm will only make the recursive call if the chosen adjacent tile is empty:

```
valid_move_dfs([Board, Player], N, Piece, DX-DY, CX-CY) :-
    N > 0,
    N1 is N - 1,
    findall(X-Y, adjacent(tile(CX, CY), tile(X, Y)), AdjacentTiles), % Get all th
    member(CX1-CY1, AdjacentTiles), % Get an adjacent tile
    \+ member(position(_, tile(CX1, CY1)), Board), % Check if the adjacent tile i
    valid_move_dfs([Board, Player], N1, Piece, DX-DY, CX1-CY1).
```

logic.pl

After validating the move, the **move_aux/3** predicate is called to execute it. This predicate starts by checking if the player's piece is on the board, in the given origin coordinates. After that, it checks if there is an opposing piece on the destination tile. If there is, and if the combat results in a draw, both pieces are removed from the board and the current player is changed to the opponent player (using the **other_player/2** predicate). If the combat results in a victory for the player's piece or if there isn't an opposing piece on the destination tile, both tiles are cleared, the player's piece is moved to the destination tile and the current player is changed to the opponent player (using the **other_player/2** predicate). The new game state, including the updated board and the new current player, is returned:

```
% move_aux(+GameState, +Move, -NewGameState)
% Moves a piece from one tile to another, and returns the new game state
move_aux([Board, Player], OX-OY-DX-DY, [NewBoard, NewPlayer]) :-
    member(position(Piece, tile(OX, OY)), Board),
    member(position(Defender, tile(DX, DY)), Board),
    piece_info(Piece, _, Type),
    piece_info(Defender, _, DefenderType),
    combat(Type, DefenderType, none), % Check if the combat results in a draw
    !,
    delete(Board, position(Piece, tile(OX, OY)), Board1),
    delete(Board1, position(Defender, tile(DX, DY)), NewBoard),
    other_player(Player, NewPlayer). % Change the current player

move_aux([Board, Player], OX-OY-DX-DY, [NewBoard, NewPlayer]) :-
    member(position(Piece, tile(OX, OY)), Board), % Get the piece to move
    !,
    delete(Board, position(Piece, tile(OX, OY)), Board1),
    delete(Board1, position(_, tile(DX, DY)), Board2),
```

```
    append(Board2, [position(Piece, tile(DX, DY))], NewBoard), % Add the piece to
    other_player(Player, NewPlayer). % Change the current player
```

logic.pl

List of Valid Moves

The predicate **valid_moves/3** is responsible for returning a list of all possible moves for a player in the current game state. This predicate checks all the moves for each piece of the player in the current game state, using the **validate_move/3** predicate, and returns a list with all the valid moves. This list is obtained by using the **setof/3** predicate, as we don't want to have repeated moves in the list:

```
% valid_moves(+GameState, +Player, -Moves)
% Gets all the valid moves for the Player in the current game state
valid_moves([Board, Player], Player, Moves) :-
    setof(OX-OY-DX-DY, [Board, Player]^validate_move([Board, Player], OX-OY-DX-DY
```

logic.pl

If this predicate is called for the player that is not the current player, it will return an empty list:

```
% No moves available for the player that is not the current player
valid_moves([_, Player], Opponent, Moves) :-
    Opponent \= Player,
    Moves = [].
```

logic.pl

End of Game

The predicate **game_over/2** is responsible for checking if the game is over and, in case it is, returning the winner. At first, this predicate checks if the current player's pentagon was captured. If so, the **other_player/2** predicate is used to get the opponent player that will be returned as the winner. If the current player's pentagon wasn't captured, the predicate will check if the opponent's pentagon was captured. If so, the current player is returned as the winner:

```
% game_over(+GameState, -Winner)
% Checks if the game is over and returns the winner
game_over([Board, Player], Winner) :-
    \+ member(position(Player-pentagon-_, tile(_, _)), Board),
```

```
    other_player(Player, Winner).

game_over([Board, Player], Player) :-
    other_player(Player, Opponent),
    \+ member(position(Opponent-pentagon-_, tile(_, _)), Board).
```

logic.pl

If neither of the pentagons was captured, the predicate will check if all gold tiles are occupied by the current player. If so, and as this predicate is called in the beginning of the game loop, this means that the current player was occupying both gold tiles at the end of the opponent's turn, and so the current player is returned as the winner:

```
game_over([Board, Player], Player) :-
    findall(X-Y, gold_tile(X, Y), GoldTiles),
    findall(X-Y, (member(position(Player-_-_, tile(X, Y)), Board), member(X-Y, Go
    length(GoldTiles, N),
    length(GoldTilesWithPlayer, N).
```

logic.pl

Game State Evaluation

The **value/3** predicate is responsible for evaluating the current game state for the evaluated player.

```
% value(+GameState, +EvaluatedPlayer, -Value)
% Evaluate the value of the game state for the EvaluatedPlayer
value([Board, Player], EvaluatedPlayer, Value) :-
    evaluate_advantage([Board, _], EvaluatedPlayer, Advantage), % Get the advanta
    closest_to_opponent_pentagon([Board, _], EvaluatedPlayer, DistanceToOpponentP
    other_player(EvaluatedPlayer, Opponent),
    closest_to_opponent_pentagon([Board, _], Opponent, DistanceToPlayerPentagon),
    Distance is DistanceToPlayerPentagon - DistanceToOpponentPentagon,
    wins_game([Board, Player], EvaluatedPlayer, Wins), % Check if the EvaluatedPl
    Value is Wins + Advantage + Distance.
```

logic.pl

At first, the **value/3** predicate will call the **evaluate_advantage/3** predicate to determine the advantage of the game state for the evaluated player and return it. This predicate will check the number of pieces of the evaluated player and the number of pieces of the opponent player. The advantage of the evaluated player is equal to the difference between the number of pieces of the evaluated player and the number of pieces of the opponent player. If the opponent player has more pieces than the evaluated player, the advantage will be negative, indicating that the move could not be very favorable for the evaluated player. If both players have the same number of pieces, the advantage is equal to 0:

```
% evaluate_advantage(+GameState, +EvaluatedPlayer, -Advantage)
% Evaluate the advantage of the game state for the EvaluatedPlayer
evaluate_advantage([Board, _], EvaluatedPlayer, Advantage) :-
    count_player_pieces([Board, _], EvaluatedPlayer, NumPlayerPieces), % Count th
    other_player(EvaluatedPlayer, Opponent),
    count_player_pieces([Board, _], Opponent, NumOpponentPieces), % Count the num
    Advantage is NumPlayerPieces - NumOpponentPieces.
```

logic.pl

After determining the advantage, the **value/3** predicate will call the **closest_to_opponent_pentagon/3** predicate to determine the distance between the evaluated player's closest piece and the opponent's pentagon. This predicate will check all the pieces of the evaluated player and return the distance between the closest piece and the opponent's pentagon. If the evaluated player has no pieces or if the opponent has no pentagon, the distance is equal to 0:

```
% closest_to_opponent_pentagon(+GameState, +EvaluatedPlayer, -Distance)
% Unifies Distance with the distance between the closest piece of the EvaluatedPl
closest_to_opponent_pentagon([Board, _], EvaluatedPlayer, 0) :-
    other_player(EvaluatedPlayer, Opponent),
    \+ member(position(Opponent-pentagon-_, _), Board).

closest_to_opponent_pentagon([Board, _], EvaluatedPlayer, 0) :-
    \+ member(position(EvaluatedPlayer-_-_, _), Board).

closest_to_opponent_pentagon([Board, _], EvaluatedPlayer, MinDistance) :-
    findall(Position, (member(position(EvaluatedPlayer-_-_, Position), Board)), P
    other_player(EvaluatedPlayer, Opponent),
    member(position(Opponent-pentagon-_, OpponentPiecePosition), Board),
    setof(Distance, Position^PlayerPiecesPositions^OpponentPiecePosition^(member(
```

logic.pl

After calling the **closest_to_opponent_pentagon/3** predicate for the evaluated player, the **value/3** predicate will also call this predicate for the opponent player, in order to determine the distance between the opponent's closest piece and the evaluated player's pentagon. We want the evaluated player's pentagon to be as far as possible from the opponent's closest piece, and the evaluated player's closest piece to be as close as possible to the opponent's pentagon. Therefore, the distance between the evaluated player's closest piece to the opponent's pentagon is subtracted from the distance between the opponent's closest piece to the evaluated player's pentagon. This difference is saved saved as *Distance* and is used afterwards to help determining the value of the game state for the evaluated player:

```
%(...)
closest_to_opponent_pentagon([Board, _], EvaluatedPlayer, DistanceToOpponentPenta
other_player(EvaluatedPlayer, Opponent),
closest_to_opponent_pentagon([Board, _], Opponent, DistanceToPlayerPentagon), % G
Distance is DistanceToPlayerPentagon - DistanceToOpponentPentagon
%(...)
```

*logic.pl (in **value/3** predicate)*

At last, the predicate **value/3** will call the **wins_game/3** predicate to check if the evaluated player wins the game in the current game state. This predicate will call the **game_over/2** predicate to check if the game is over and if the evaluated player is the winner. If so, the predicate will return a value of **1000**. If the game is not over yet, the predicate will return a value of **0**:

```
% wins_game(+GameState, +EvaluatedPlayer, -Value)
% Checks if the EvaluatedPlayer wins the game and returns the value of the game s
wins_game([Board, Player], EvaluatedPlayer, Value) :-
    game_over([Board, Player], EvaluatedPlayer), % Check if the EvaluatedPlayer w
    Value is 1000,
    !.

wins_game([_, _], _, 0).
```

logic.pl

Finally, the **value/3** predicate will sum the advantage, the distance and the result of the **wins_game/3** predicate and return it as the value of the game state for the evaluated player:

```
%(...)
Value is Wins + Advantage + Distance.
```

```
%(...)
```

*logic.pl (in **value/3** predicate)*

Computer Plays

When processing a turn in the game loop, if the current player is a computer player, the **process_turn/2** predicate will call the **choose_move/4** predicate:

```
process_turn([Board, Player], [NewBoard, NewPlayer]) :-
    difficulty(Player, Difficulty), % Computer player
    !,
    choose_move([Board, Player], Player, Difficulty, OX-OY-DX-DY), % Get a move f
    move_aux([Board, Player], OX-OY-DX-DY, [NewBoard, NewPlayer]),
    !.
```

main.pl

If the difficulty level of the computer player is **1** (the bot chooses a valid random move), the **choose_move/4** predicate will call the **valid_moves/3** predicate to get a list of all possible valid moves for the current computer player in the current game state. After that, the predicate **random_member/2** will choose a random move from the list of possible moves and return it:

```
% choose_move(+GameState, +Player, +Level, -Move).
% Chooses a move for the difficulty level 1 (random) bot
choose_move([Board, Player], Player, 1, Move) :-
    valid_moves([Board, Player], Player, Moves), % Get all the valid moves
    random_member(Move, Moves). % Choose a random move
```

logic.pl

If the difficulty level of the computer player is **2** (the bot chooses the best valid move at that time), the **choose_move/4** predicate will also start by calling the **valid_moves/3** predicate to get a list of all possible valid moves for the current computer player in the current game state. After that, the predicate **setof/3** will be used to get a list of all possible moves and the values of the corresponding game states, ordered by value. This list will be sorted in descending order, after the **reverse/2** predicate, and the move with the highest value will be selected using the **select_value_move/3** predicate:

```
% choose_move(+GameState, +Player, +Level, -Move).
% Chooses a move for the difficulty level 2 (greedy) bot
choose_move([Board, Player], Player, 2, Move) :-
    valid_moves([Board, Player], Player, Moves), % Get all valid moves for the pl
```

```
    setof(Value-CurrentMove, [Board, Player]^[NewBoard, NewPlayer]^(member(Curren
    reverse(ValuesMoves, ReversedValuesMoves), % Reverse the list of values and m
    ReversedValuesMoves = [MaxValue-_|_], % Get the highest value
    select_value_move(ReversedValuesMoves, MaxValue, Move), % Select a move with
    !.
```

logic.pl

As more than one move can have the highest value, the **select_value_move/3** predicate is used to select a move with the highest value possible for the current computer player in the current game state. This predicate will get a list with all the moves with the highest value and choose a random move from that list, returning it:

```
% select_value_move(+ValuesMoves, +Value, -Move)
% Selects a random move with the given value
select_value_move(ValuesMoves, Value, Move) :-
    findall(M, (member(Value-M, ValuesMoves)), Moves), % Get all the moves with t
    random_member(Move, Moves), % Choose a random move with the given value
    !.
```

logic.pl

Conclusions

Tactigon was implemented successfully, with all the required features in Prolog. The game can be played by two human players, by a human player against a computer player or computer player vs computer player. The computer player can play in two different difficulty levels: random and greedy. The game can be played in three different board sizes: 11 lines, 7 columns; 13 lines, 9 columns; and 15 lines, 11 columns. The game can also be played with two advanced rules: square pieces can jump over other pieces, except for opposing squares; and pieces that start a turn on a gold tile can move an additional space on that turn. Each interaction with the user is robust.

During the development of this project, there were two components that stood out as the most challenging: the implementation of the predicates to draw the board and the implementation of the predicates to validate, execute and choose moves. By implementing these components, we were able to learn more about Prolog and its logic programming paradigm, and to consolidate the concepts developed in theoretical and practical classes.

Bibliography

The set up and the rules of the game were consulted in **Tactigon's official website** and in the **Tactigon's rulebook**:

- Tactigon - How to Play
- Tactigon - Rulebook

For more information about hexagonal grids and hexagonal coordinates, the **Red Blob Games** website was consulted:

- Hexagonal Grids - Red Blob Games