Trike

Group: Trike_1

Identification of the group elements and Contribution

- João Pedro Moreira Costa (up202108714@.up.pt) - 50%
- Rafael Neves Teixeira (up202108831@up.pt) - 50%

Installation and Execution

To play the game it is required to have installed at least the 4.8.0 version of SICStus Prolog and the folder `src` that contains the code for the game functioning.

After having the requirements mentioned previously, on the SICStus interpreter, we need to consult the `play.pl` file located in the folder src:

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

If using Windows, we can click on the options `File` -> `Consult` -> select the `play.pl` file and then run the play predicate in the interpreter:

```
?- play.
```

Description of the Game

- `Game Board:` Trike is played on an equilateral triangular hexagon-tessellated grid but since there isn't a good way to represent it in code language we had to opt out for a pyramid grid.
- `Number of Players:` The game is played by 2 players. The first player start initially with the black pieces and the second player with the white ones. Colours can be switched after the pie rule.
- `Game Components:` Black/white checkers are used to represent the two players (represented as `B` and `W`, respectively). The last checker (pinned checker) is used to determine the following possible moves, vertically, horizontally and diagonally until a piece or the end of the board is found.
- `Game Objective:` The primary objective is to trap the pinned checker, and at the end of the game, accumulate as many points as possible.
- `Movement:` Players take turns moving the pinned checker around the board. Passing is not allowed. The pawn can move any number of empty points in a straight line, in any direction, but cannot land on or jump over occupied points.
- `Game Progression:` When a player moves the piece, they must first place a checker of their own colour onto the destination point and then move the pawn on top of it.
- `Winning:` The game ends when the pawn becomes trapped. The player with the most points wins.
- `Scoring:` At the game's conclusion, each player scores one point for every checker of their own colour that is adjacent to or underneath the pawn.

- `Pie Rule:` Before the game begins, the first player selects a colour and places a checker on any point of the board. At this point, the second player has a one-time opportunity to switch sides rather than make a regular move.

Sources

> **Game Website and Rules:** (https://boardgamegeek.com/boardgame/307379/trike)

> **Online Gameplay:** (https://pt.boardgamearena.com/gamepanel?game=trike)

Game Logic

Internal Game State Representation

GameState is represented as a list with 2 elements, the current Player and the current Board. Board is also represented as a list but it includes sublists that represent rows of the board. Each element of those sublists, represents an element in a column. There can be 4 diferent values on the board:

- `0` represents an empty space - printed on the board as `.`
- `p` represents a playable space - printed on the board as `x`
- `b` represents a space occupied by a black piece - printed on the board as `B`
- `w` represents a space occupied by a white piece - printed on the board as `W`

Here are some representations of the different states on the game:

Initial State

```
GameState = [b,
    [0],
    [0,0],
    [0,0,0],
    [0,0,0,0],
    [0,0,0,0,0],
    [0,0,0,0,0,0],
    [0,0,0,0,0,0,0],
    [0,0,0,0,0,0],
    [0,0,0,0,0],
    [0,0,0,0],
    [0,0,0],
    [0,0],
    [0]
].
```

Intermediate State

```
GameState = [w,
    [0],
```

```
    [0,b],
    [0,w,0],
    [b,b,0,w],
    [0,0,0,0,0],
    [0,0,0,0,0,0],
    [0,0,0,0,0,0,0],
    [0,0,0,0,0,0],
    [0,0,0,0,0],
    [0,b,b,0],
    [w,0,w],
    [0,0],
    [0]
].
```

Final State

```
GameState = [b,
    [0],
    [0,b],
    [0,w,0],
    [b,b,0,w],
    [0,0,0,0,0],
    [0,0,0,0,0,0],
    [0,0,0,0,0,0,0],
    [0,0,0,w,0,0],
    [0,0,b,w,b],
    [0,b,b,w],
    [w,0,w],
    [0,0],
    [0]
].
```

Game State Visualization

Game Menu

The game menu is displayed like this:

```
      _________  ______ ______
     /_  __/ _ \\/  _/ //_/ __/
      / / / , _// // ,< / _/
     /_/ /_/|_/___/_/|_/___/

    1. Player Vs Player
    2. Player Vs Computer (Easy)
    3. Player Vs Computer (Hard)
    4. Computer Vs Computer
```

```
5. Instructions
0. Quit

_____________________________
SELECT YOUR OPTION!
```

An option is picked by typing a number followed by a `.` and pressing the `Enter` key.

The first 4 options correspond to dynamic stages of the program and the last one to a static page.

By picking the option `0` the following text is displayed:

```
          _________  ______ ______
        /_  __/ _ \\/  _/ //_/ __/
         / / / , _// // ,< / _/
        /_/ /_/|_/___/_/|_/___/

Thank you for playing. Hope to see you soon!
```

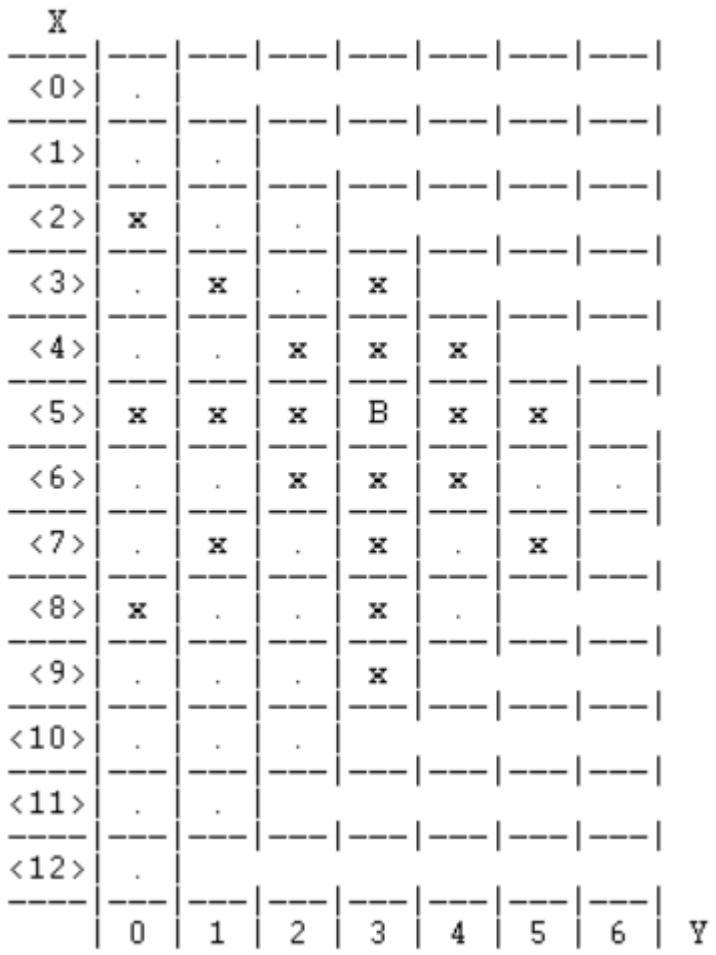
Board

Once we start a game, the board is displayed like this:

```
  X
-----|---|---|---|---|---|---|---|
 <0> | . |
-----|---|---|---|---|---|---|---|
 <1> | . | . |
-----|---|---|---|---|---|---|---|
 <2> | . | . | . |
-----|---|---|---|---|---|---|---|
 <3> | . | . | . | . |
-----|---|---|---|---|---|---|---|
 <4> | . | . | . | . | . |
-----|---|---|---|---|---|---|---|
 <5> | . | . | . | . | . | . |
-----|---|---|---|---|---|---|---|
 <6> | . | . | . | . | . | . | . |
-----|---|---|---|---|---|---|---|
 <7> | . | . | . | . | . | . |
-----|---|---|---|---|---|---|---|
 <8> | . | . | . | . | . |
-----|---|---|---|---|---|---|---|
 <9> | . | . | . | . |
-----|---|---|---|---|---|---|---|
<10> | . | . | . |
-----|---|---|---|---|---|---|---|
<11> | . | . |
-----|---|---|---|---|---|---|---|
<12> | . |
-----|---|---|---|---|---|---|---|
     | 0 | 1 | 2 | 3 | 4 | 5 | 6 | Y
```

To display it we use the predicate `display_game(+GameState)`, printing the board and whose turn it is to play.

Also, everytime someone plays, the board is updated to show every move that is possible to take using the predicate `valid_moves(+GameState, +PlayerPos, -ListOfMoves)`:



Move Validation and Execution

The `move(+GameState, +Move, -NewMove, -NewGameState)` predicate takes the current game state, the move to take, the new move in case an incorrect one was given and the new game state resulting from that move.

Inside that function, we use the predicate `check_if_valid(+Board, +X, +Y)` to check if the move to take is inside the board and if it corresponds to a valid move:

```
check_if_valid(Board, X, Y) :-
    is_inside(Board, X, Y),
    nth0(X, Board, Row),
    nth0(Y, Row, p).
```

If it is valid, a piece of the corresponding player is placed on the board using the predicate `replace(Board, PointX, PointY, Player, TempBoard)`, the playable moves are remove using the predicate `clean_playables(+TempBoard, -NewBoard)`, the player playing switches and the board with the play made

without the possible moves is displayed `display_game(+GameState)`. But, if the move selected isn't valid, a recursive call is made until the player picks a correct play.

For the bot, a predicate `choose_move(+GameState, +Level, -Move)` is called which objective is to pick a valid move using the algorithm according to the difficulty picked initially for the computer.

List of Valid Moves

The `valid_moves(+GameState, +PlayerPos, -ListOfMoves)` predicate takes 3 arguments, the current game state, the current player position and a list with all the possible moves that is returned. That list represents the board and is printed showing the player where he can play. This predicate calls the `swap(+PlayerX, +PlayerY, +Board, -ListOfMoves)` predicate which function is to replace on the board all the empty spaces that could represent a valid move with a `p`. To achieve this, this predicate travels the board vertically, horizontally and diagonally until it reaches the end of the board or finds a `b` or `w`, while replacing every `0` space with a `p`.

```
valid_moves([CurPlayer|Board], [PlayerX, PlayerY], ListOfMoves) :-
    swap(PlayerX, PlayerY, Board, ListOfMoves),
    display_game([CurPlayer|ListOfMoves]).
```

In addition to this, we created a `clean_playables(+Board, -NewBoard)` predicate that replaces every `p` from the board with an empty space `0` so that the players can have an easier view of the current state of the game:

```
clean_playables([], []).
clean_playables([Row|RestOfRows], [NewRow|NewRest]) :-
    replace_p_in_row(Row, NewRow),
    clean_playables(RestOfRows, NewRest).

replace_p_in_row([], []).
replace_p_in_row([p|Rest], [0|NewRest]) :-
    replace_p_in_row(Rest, NewRest).
replace_p_in_row([X|Rest], [X|NewRest]) :-
    X \= p,
    replace_p_in_row(Rest, NewRest).
```

End of Game

After the first play, the game is always being ran in a loop inside the predicate `gameplay(+GameState, +PlayerPos, FinalScore, Winner)` (Player Vs Player) or `gameplay_bot(+GameState, +PlayerPos, +Level, -FinalScore, -Winner)` (Player Vs Bot) or `gameplay_bot_vs_bot(+GameState, +PlayerPos, -FinalScore, -Winner)` (Bot Vs Bot). Inside these predicates there is another predicate that is always checking if the game has ended after every move that was made, namelly `(game_over(GameState)`, that receives the board with all the possible moves:

```
game_over([Player|ListOfMoves]) :-
    check_board(ListOfMoves).
```

- `check_board(ListOfMoves)`: check if there is a 'p' in the board.

This predicate ends the game if there are no cells marked with `p` (playable spaces) on the board and if this predicate checks for true (there are playable spaces), it means that the game hasn't ended and so the following play is proceeded, but if it checks for false (no playable spaces), the predicate `calculate_final_score(+GameState], +PlayerPos, -FinalScore, -Winner)` is called, identifying the player who won and retrieving the final score that corresponds the number of pieces belonging to that player which are below or around the last piece played.

When the game finishes, this is an example of what is displayed:

```
   X
-----|---|---|---|---|---|---|---|
 <0>|   .   |
-----|---|---|---|---|---|---|---|
 <1>|   B   |   .   |
-----|---|---|---|---|---|---|---|
 <2>|   .   |   B   |   .   |
-----|---|---|---|---|---|---|---|
 <3>|   .   |   W   |   B   |   .   |
-----|---|---|---|---|---|---|---|
 <4>|   W   |   W   |   B   |   .   |   B   |
-----|---|---|---|---|---|---|---|
 <5>|   B   |   B   |   B   |   W   |   W   |   W   |
-----|---|---|---|---|---|---|---|
 <6>|   W   |   W   |   W   |   .   |   B   |   W   |   .   |
-----|---|---|---|---|---|---|---|
 <7>|   B   |   W   |   .   |   .   |   W   |   B   |
-----|---|---|---|---|---|---|---|
 <8>|   W   |   B   |   .   |   .   |   W   |
-----|---|---|---|---|---|---|---|
 <9>|   .   |   .   |   W   |   B   |
-----|---|---|---|---|---|---|---|
<10>|   .   |   B   |   B   |
-----|---|---|---|---|---|---|---|
<11>|   B   |   W   |
-----|---|---|---|---|---|---|---|
<12>|   B   |
-----|---|---|---|---|---|---|---|
     |   0   |   1   |   2   |   3   |   4   |   5   |   6   |   Y

Player w turn

Player b is the WINNER!!!
Scored 2 points.
```

Game State Evaluation

The `value(+Board, +Row, +Col, -Value)` predicate takes four arguments: the current Board, the coordinates of a playable piece and a Value that determines the quality of the play.

```
value(Board, Row, Col, Value) :-
    clean_playables(Board, NewBoard),
    swap(Row, Col, NewBoard, ListOfMoves),
    count_p(ListOfMoves, Value).
```

This predicate calls the predicate `swap(+Row, +Col, +NewBoard, -ListOfMoves)` to determine and mark on the board the playable places that are possible from the coordinate (Row, Col) given and the predicate `count_p(+ListOfMoves, -Value)` to determine how many playable spaces exist from that coordinate. This predicate will be useful for the bot to determine the move with the best `Value` by iterating trough all the possible plays from a given position.

Computer Plays

All the bots resort to the predicate `choose_move(+GameState, +Level, -Move)` to pick a valid move according to the difficulty (Level) picked at the game menu:

```
choose_move([_Player|Board], Level, [PointX, PointY]) :-
    ((Level = 3) ->
        count_p(Board, Count),
        ((Count = 1) -> get_p_coordinates(Board, [(PointX, PointY)]);
            hard(Board, PointX, PointY)
        )
        ;
        count_p(Board, Count),
        ((Count = 1) -> get_p_coordinates(Board, [(PointX, PointY)]);
            choose_random_p(Board, PointX, PointY)
        )
    ).
```

If `Level` corresponds to `3`, the `hard(+Board, -PointX, -PointY)` predicate is called. Oherwise, the `choose_random_p(+Board, -PointX, -PointY)` predicate is called to run the algorithm with easiest difficulty available on the game, which strategy is to pick a playable space, marked with `p`, randomly.

To make up a good strategy for an algorithm for the hard difficulty of the bot, after playing and studying the game Trike a bit, we realised that one of best possible moves to make to play on the cell with the most available spaces possible. This way, the play takes more of a defensive aproach since it always places a piece where the opponent isn't trying to end the game. With that, we decided to implement our hardest algorithm around that strategy, so the predicate `hard(+Board, -PointX, -PointY)` does exactly what was mentioned before by picking the move with the best Value:

```
hard(Board, Row, Col) :-
    get_p_coordinates(Board, PList),
    process_p(Board, PList, Values),
    max_position1(Values, Index),
    custom_nth1(Index, PList, (Row, Col)).
```

```
process_p(_, [], []).
process_p(Board, [(Row,Col)|Res], [Value|Values]) :-
    value(Board, Row, Col, Value),
    process_p(Board, Res, Values).

value(Board, Row, Col, Value) :-
    clean_playables(Board, NewBoard),
    swap(Row, Col, NewBoard, ListOfMoves),
    count_p(ListOfMoves, Value).
```

- `get_p_coordinates(+Board, -PList)`: checks all the cells of the board and returns all the playable ones (marked with `p`)
- `process_p(+Board, +PList, -Values)`: process each element in the list and store the values in a new list. This predicate iterates through all the playable spaces, invoking the value(+Board, +Row, +Col, -Value) predicate to calculate the number of playable positions if a piece is placed on any of these available spaces and stores this count in the `Values` variable
- `max_position1(+Values, -Index)`: find the position of the maximum value on the list, this is, the play that will have the most playable spaces
- `custom_nth1(+Index, +PList, -(Row, Col)`: retrieve the position of the element with the maximum score

---

Conclusions

We are satisfied with the final result of this project, since at the start of the development of this game we felt pressed about how much time we had to finish it but thankfully we ended up completing all the main functionalities that were planned, even if it took us a lot of time and work. The major difficulties encountered by us while making the project were creating an intuitive board in the SicStus terminal since we don't have many options for how to display it, the debugging of the code since it gets stressful in this programming language and the implementation of some predicates with higher complexity. Regardless, the development of this game enriched our knowledge in prolog development and made us realize that it was important for us to confront such a different programming environment since it helped us understand how to approach complex problems and structure code in a more logical and rule-driven manner.

Possible improvements

If we had more time, we would like to have:

- Implemented the choice of playing in different board sizes
- Improved the algorithm for hard bot difficulty so that it could detect when the opponent could trap it's pieces and prevent it

---

Bibliography

- Slides from the teorical lessons
- https://sicstus.sics.se/sicstus/docs/latest4/html/sicstus.html/
