Trike

Neste projeto foi feita a implementação do jogo Trike, utilizando a linguagem prolog.

Turma 08 grupo Trike_5:

- Francisco Pires da Ana up202108762 (50%)
- José Pedro Evans up202108818 (50%)

Trike: https://boardgamegeek.com/boardgame/307379/trike
(https://boardgamegeek.com/boardgame/307379/trike)

Instalação e Execução

Para a instalação do jogo é necessário fazer download do ficheiro FL_TP1_T08_Trike5.zip e descompactá-los. Dentro do diretório src, consultar o ficheiro main.pl (http://main.pl), através da linha de comandos ou pela UI do SicstusProlog 4.7.1. O jogo está disponível em ambientes Windows e Linux.

O jogo inicia-se com o predicado play/0.

Descrição do Jogo

O Trike é um jogo de estratégia pensado para 2 jogadores. O objetivo é tentar encurralar o jogador adversário, levando o peão a ficar sem jogadas válidas e rodeado pelo maior número possível de peças da tua cor.

Regras

- Trike é um jogo abstrato, jogado num tabuleiro equilátero, triangular, de tamanho variável (na implmentação damos a opção de variar o tamanho entre 6 e 9).
- É jogado com um peão neutro, e peças de 2 cores (utilizamos os símbolos ‘X’ e ‘O’).
- O peão pode movimentar-se livremente em linha reta pelas 6 direções, desde que não passe por uma casa ocupada por outro peão (não pode saltar).
- Quando não há mais movimentos possíveis o jogo acaba, e ganha quem tiver mais pontos, isto é, quem tiver mais peças da sua cor adjacentes ao peão(incluindo a casa onde o mesmo se encontra).
- Na primeira jogada, o primeiro jogador escolhe um sítio qualquer no tabuleiro para jogar, o segundo jogador pode escolher seguir normalmente o jogo, ou ficar com a cor

escolhida na primeira jogada e passar o seu turno.

Tivemos em conta as regras escritas pelo criador do jogo em BoardGeek (https://boardgamegeek.com/boardgame/307379/trike).

Game Logic

Representação do Estado do Jogo

Para guardar o estado do jogo, utilizamos a variável GameState, que é utilizada em vários predicados, e formada por uma lista de 3 elementos:

1. BoardState - Representa o estado do tabuleiro num momento do jogo, e é constituído por uma matriz (lista de listas), triangular, isto é, em que o numero de elementos vai aumentando quanto maior o índice. O tamanho, ou seja, o número de linhas que terá o triângulo, é definido pelo utilizador nas configurações iniciais, podendo variar entre 6 e 9. Inicialmente é criado com todas as posições ocupadas por *empty*, sendo que depois os jogadores serão representados por *player1*, *player2*, *bot1* e/ou *bot2*.
2. Player - Representa o jogador que está a jogar no momento (átomo player1, player2, bot1, bot2).
3. Row-Col - Representa o par coluna linha, sendo a linha uma letra e a coluna um número, onde o peão (jogada atual) se encontra.

A principal função responsável por criar o BoardState é a init_state, onde se define que símbolo estará utilizado a cada jogador e se criar a lista de listas com as corretas dimensões:

```
init_state(Size, Acc, Board) :-
    asserta(state_char(empty, ' ')),
    asserta(state_char(player1, 'X')),
    asserta(state_char(player2, 'O')),
    Size > 0,
    length(Row, Size),
    maplist(=(empty), Row),
    NewSize is Size - 1,
    init_state(NewSize, [Row|Acc], Board).
```

Esta função cria recursivamente uma lista de listas (BoardState) que será utilizada para gerar o GameState, que recebe ainda um player e a posição atual.

Estado de Jogo Inicial

```
GameState([[[empty],[empty,empty],[empty,empty,empty],
[empty,empty,empty,empty],[empty,empty,empty,empty,empty],
[empty,empty,empty,empty,empty,empty],
[empty,empty,empty,empty,empty,empty,empty],
[empty,empty,empty,empty,empty,empty,empty,empty]],player1])
```

Exemplo de GameState inicial, de um jogo com tabuleiro de size 8. O player1 é sempre o primeiro a jogar, como é possível ver no estado de jogo inicial. De notar também que ainda não há coordendas guardadas, já que é o primeiro estado.

Estado de Jogo Intermédio

```
GameState([[[empty],[player1,empty],[empty,player2,empty],
[empty,empty,empty,empty],[empty,empty,empty,empty,empty],
[empty,empty,empty,empty,empty,empty],
[empty,empty,empty,empty,empty,empty,empty],
[empty,empty,empty,empty,empty,empty,empty,empty]],player1,C-2])
```

> Exemplo de GameState depois de uma jogada feita pelo player1 (‘X’) e uma jogada pelo player2 (‘O’), sendo a última em C-2, e sabendo que o segundo jogador não quis trocar de lados.

Estado do Jogo Final

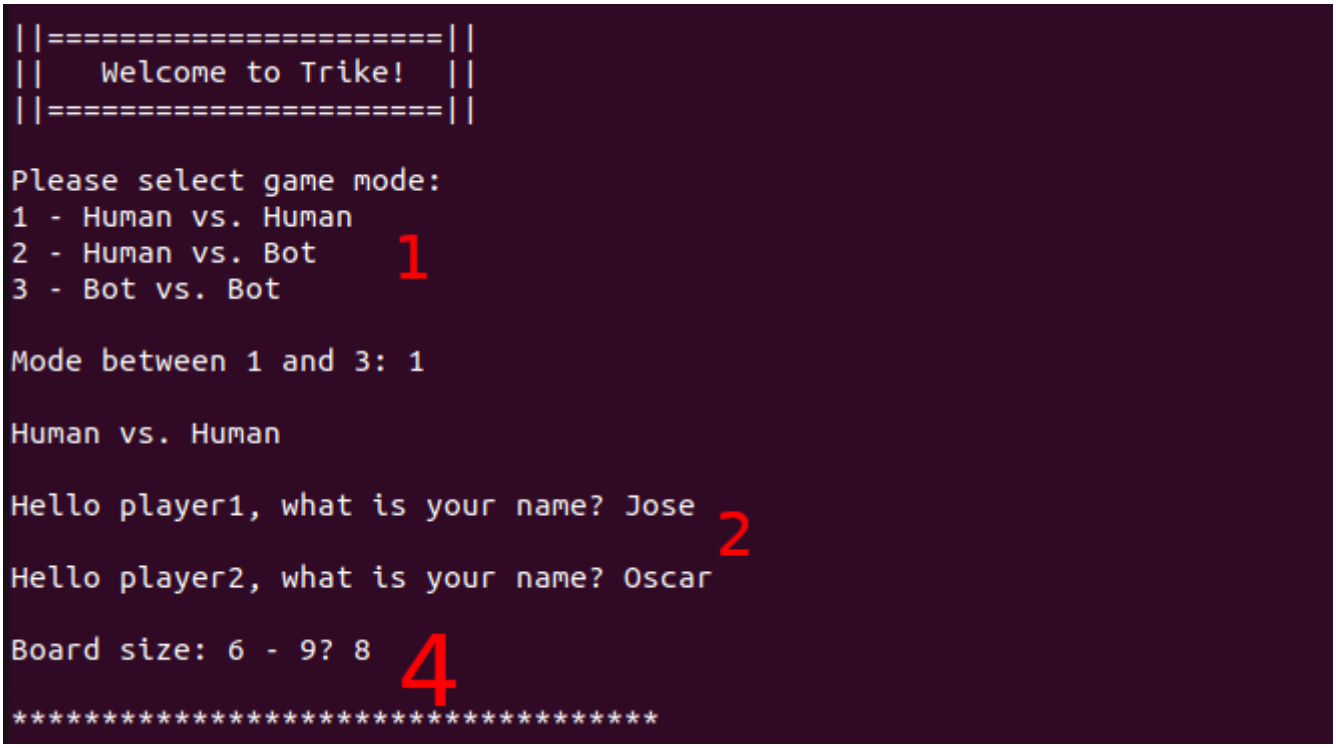
```
GameState([[[empty],[player1,empty],[player2,player2,empty],
[player1,player2,player1,empty],[empty,empty,empty,empty,empty],
[empty,empty,empty,empty,empty,empty],
[empty,empty,empty,empty,empty,empty,empty],
[empty,empty,empty,empty,empty,empty,empty,empty]],player1,C-1]
```

> Exemplo de GameState onde o player2 (Oscar) joga com ‘O’ e ganha, já que tem 3 peças a rodear a cada C-1.

Visualização do Estado do Jogo

Depois da execução do predicado play/0, o utilizador é convidado a escolher alguns parâmetros que determinarão as características da partida:

1. Modo (Humano/Humano, Humano/Bot e Bot/Bot);
2. Nome dos Jogadores;
3. Tipo do bot (No caso de ser contra o bot)
4. Tamanho do Tabuleiro;



Exemplo de possível de início de jogo entre dois humanos, Jose e Oscar.

Em todos os casos são utilizados mecanismos de validação de input para garantir que o código não quebra por input inválido. Por exemplo, no predicado get_option/4, garantimos que não é escolhido um numero fora dos limites supostos:

```
get_option(Min,Max,Context,Value):-
    format('~a between ~d and ~d: ', [Context, Min, Max]),
    repeat,
    read_number(Value),
    between(Min, Max, Value), !.
```

Para mostrar o estado do jogo, utilizamos o predicado display_game(+BoardState), que dado o estado do tabuleiro o imprime usando a função display_board(+BoardState), do ficheiro board.pl (http://board.pl). Estes são os principais predicados envolvidos na sua visualização:

```
display_board(BoardState) :-
    length(BoardState, N),
    DeltaX is N - 1,
    draw_board(DeltaX, 1, BoardState, 'A').

% draw_board(+DeltaX, +NRet, +BoardState, +Letter)
% Prints the board.
draw_board(-1, _, _, _).
draw_board(DeltaX, NRet, [Line|Others], Letter) :-
    draw_board_line(DeltaX, NRet, Line, Letter),
    NRet1 is NRet + 1,
    DeltaX1 is DeltaX - 1,
    next_char(Letter, Letter1),
    draw_board(DeltaX1, NRet1, Others, Letter1).

% draw_board_line(+DeltaX, +NRet, +LineState, +Letter)
% Prints a line of the board.
% DeltaX is the number of empty spaces to print before the board line.
% NRet is the number of the cells in the line.
% LineState is a list of the states of the cells in the line.
% Letter is the letter of the line.
draw_board_line(DeltaX, NRet, LineState, Letter) :-
    NEspacos is DeltaX * 4,
    print_n(NEspacos, ' '), draw_up_line(NRet, Letter, 1),
    print_n(NEspacos, ' '), draw_up2_line(NRet),
    print_n(NEspacos, ' '), draw_middle_line(NRet, LineState),
    print_n(NEspacos, ' '), draw_down_line(NRet).
```

No predicado draw_board_line/4, são utilizadas funções auxiliares para definir diferentes linhas, cuja legenda é:

```
/*

            |__ C1 __|__ C2 __|__ C3 __| <- up line
            |        |        |        | <- up2 line
            |        |        |        | <- middle line
            |________|________|________| <- down line

*/
```

Validação e Execução das Jogadas

> Depois da primeira jogada, o jogo é constituído essencialmente pelo game_cycle(+GameState), que termina apenas quando se atinge a condição de game_over, ou seja, quando a lista de jogadas possíveis está vazia:

```
game_over(BoardState, CurrX, CurrY, Winner):-
    write('Game Over!\n'),
    check_winner(BoardState, CurrX, CurrY, Winner),
    congratulate(Winner), nl,nl.

% game_cycle(+BoardState, +Player, +CurrRow-CurrCol)
% Main game cycle
game_cycle([BoardState, Player, CurrRow-CurrCol]):-
    write([BoardState,Player, CurrRow-CurrCol]), nl,
    display_game(BoardState),
    get_moves(BoardState, CurrRow, CurrCol, Moves),
    (Moves = [] -> game_over(BoardState, CurrRow, CurrCol, Winner) ; true),
    (Moves \= [] ->
        print_turn_before(Player),
        print_curr_position(CurrRow-CurrCol),
        write('Possible moves: \n'),
        print_moves(Moves),
        get_move(Row-Col, Moves, Player, BoardState),
        move(BoardState, Row-Col, Player, [NewBoardState, NewPlayer]),
        print_turn_after(Player, Row-Col),
        game_cycle([NewBoardState, NewPlayer, Row-Col])
    ; true).
```

> É importante notar que a segunda jogada tem de permitir a troca de lados, algo que está contemplado no predicado begin_game/2, quando se avalia a intenção do jogador trocar de lados.

A lista de jogadas é obtida a partir do predicado get_moves/4, que computa todas as jogadas possíveis nas 6 direções, usando as funçoes auxiliares presentes no ficheiro utils.pl (http://utils.pl), e que consideram a jogada válida quando:

1. As coordenadas estão dentro do tabuleiro (predicado in_bounds/3).
2. As coordenadas fazem parte de uma das casas para onde o peão pode andar (todas as 6 direções, em linha reta, desde que não haja obstáculos).

```
% get_moves_no(+BoardState, +Row, +Col, -Moves)
% Gets the possible moves of the piece in the coordinates (Row,Col) of the
board and direction NO.
get_moves_no(BoardState, Row, Col, Moves):-
    char_code(Row, RowCode),
    RowCode1 is RowCode - 1,
    char_code(Row1, RowCode1),
    Col1 is Col - 1,

    length(BoardState, BoardSize),
    (in_bounds(Row1, Col1, BoardSize) ->
        (get_piece(BoardState, Row1, Col1, Piece),
        (Piece == empty ->
            Moves = [Row1-Col1|Moves1],
            get_moves_no(BoardState, Row1, Col1, Moves1) ;
            Moves = [])
        ) ;
        Moves = []
    ).
```

O predicado get_moves_no/4 computa todas as possíveis posições para mover o peão na direção noroeste. O predicado get_moves/4 junta todas as posições de todas as direções numa só lista **Moves**, que é impressa para que o jogador possa saber as posições para onde pode jogar, juntamente com a posição atual do peão.

O predicado move/4 é responsável por mover o peão para a casa válida escolhida pelo utilizador, trocando por fim o jogador para o próximo turno.

```
% put_piece(+BoardState, +Row-Col, +Piece, -NewBoardState)
% Puts a piece in the given position
put_piece(BoardState, Row-Col, Piece, NewBoardState):-
    char_code('A', ACode),
    char_code(Row, RowCode),
    OffsetRow is RowCode - ACode,
    nth0(OffsetRow, BoardState, RowList),
    Col1 is Col - 1,
    replace(Col1, Piece, RowList, NewRowList),
    replace(OffsetRow, NewRowList, BoardState, NewBoardState).

% move(+BoardState, +Row-Col, +Player, -[NewBoardState, NewPlayer])
% Moves a piece to the given position and switches the player
move(BoardState, Row-Col, Player, [NewBoardState, NewPlayer]):-
    put_piece(BoardState, Row-Col, Player, NewBoardState),
    other_player(Player, NewPlayer).
```

Fim do Jogo

> Quando não há mais jogadas disponíveis, é utilizado o predicado check_winner/4 para recolher todas as peças adjacentes ao peão que se encontra na última casa. Finalmente, são contadas as que pertencem a cada jogador retornando um vencedor que é apresentado numa mensagem final, seguido da terminação do programa.

```
% check_winner(+BoardState, +Line, +N, -Winner)
% Unifies the Winner with the player that won the game
check_winner(BoardState, Line, N, Winner):-
    get_piece(BoardState, Line, N, Piece),
    get_adjacent_pieces(BoardState, Line, N, AdjacentPieces),
    count(player1, AdjacentPieces, Player1Count),
    count(player2, AdjacentPieces, Player2Count),
    (Player1Count > Player2Count -> Winner = player1
    ; Player1Count < Player2Count -> Winner = player2
    ; Winner = Piece).
```

```
Game Over!

Congratulations Oscar, you won!
```

Jogos contra o computador

Existe ainda a possibilidade de jogar contra com o computador, em dificuldades distintas:

1. O bot escolhe um movimento aleatoriamente, entre aqueles presentes na lista **Moves**.
2. O bot utiliza uma estratégia greedy para fazer as jogadas

```
% choose_move(+Player, +Moves, -Row-Col, +Difficulty, +BoardState)
% In easy mode, chooses a random move
choose_move(_, Moves, Row-Col, 1, _):-
    choose(Moves, Row-Col).

% In hard mode, chooses the best move among the possible ones
choose_move(Player, Moves, Row-Col, 2, BoardState):-
    choose_best_move(Player, Moves, Row-Col, BoardState).
```

O predicado choose_move/5 escolhe um movimento válido aleatoriamente se a Difficulty for 1 e seguindo uma heurística greedy em caso de Difficuly 2.

```
% choose(+List, -Elt)
% chooses a random element from List
choose([], []).
choose(List, Elt) :-
        length(List, Length),
        random(0, Length, Index),
        nth0(Index, List, Elt).

 choose_best_move(+Player, +Moves, -Row-Col, +BoardState)
% Chooses the move which gives the player the most adjacent pieces of its
type
choose_best_move(_, [Move], Move, _).
choose_best_move(Player, [CurrRow-CurrCol|T], BestMove, BoardState):-
    get_adjacent_pieces(BoardState, CurrRow, CurrCol, AdjacentPieces),
    count(Player, AdjacentPieces, PlayerCount),
    choose_best_move(Player, T, PrevBestMove, BoardState),
    get_adjacent_pieces(BoardState, PrevBestRow, PrevBestCol,
PrevAdjacentPieces),
    count(Player, PrevAdjacentPieces, PrevPlayerCount),
    (PlayerCount > PrevPlayerCount -> BestMove = CurrRow-CurrCol; BestMove =
PrevBestMove).
```

> O predicado choose_best_move/4 não se encontra ainda funcional, mas a ideia seria o bot preferir sempre a jogada que maximizasse o número de peças adjacentes á sua. Desta forma, o tabuleiro seria tanto melhor quanto mais peças adjacentes o jogador tiver. Esta heurística é evidentemente simplista para a complexidade do jogo em questão, mas seria interessante testar os resultados da apicação da mesma, podendo testá-la contra outras heurísticas simples.

Conclusão

O desenvolvimento do jogo Trike em Prolog mostrou-se um grande desafio. As partes mais desafiantes da sua conceção foram a representação de elementos gráficos utilizando listas e strings, algo que é por si só pouco prático. Além disto, o desenvolvimento da nossa heurística acabou por não se ver concluído por falta de tempo. Existem ainda ligeiros problemas com a validação de inputs em algumas exceções que podem ser tratados posteriormente.

Em suma foi um projeto que nos apresentou uma paradigma de programação distinto e pouco intuitivo no primeiro contacto, mas que com o devido trabalho se mostrou capaz (a até prático em algumas partes da lógica do programa, como a recolha de movimentos válidos) de implementar o jogo Trike de forma funcional.

Referências

BoardGeek (https://boardgamegeek.com/boardgame/307379/trike), nomeadamente as threads iniciadas pelo criador do jogo sobre estratégia e instruções de jogo.