Flügelrad

UC - Programação Funcional e em Lógica

Turma 2

Luis Vieira Relvas (up202108661) *(Contribuição: 70%)*

Válter Ochôa de Spínola Catanho Castro (up201706545) *(Contribuição 30%)*

Índice

Instalação e Execução do Jogo:

Para a execução do jogo apenas é necessário consultar o ficheiro proj.pl e, de seguida, fazer uma chamada ao predicado *play/0*.

Descrição do Jogo:

Flugelrad é um jogo de tabuleiro, composto por 7 hexágonos. A cada vértice do hexágono está atribuída uma posição sendo que os valores das posições poderão ir desde 1 a 30. As bolas na fase inicial são distribuídas de uma forma alternada ao redor do tabuleiro. Para melhor compreensão, poderá visualizar a seguinte imagem que representa o estado do jogo após algumas movimentações:

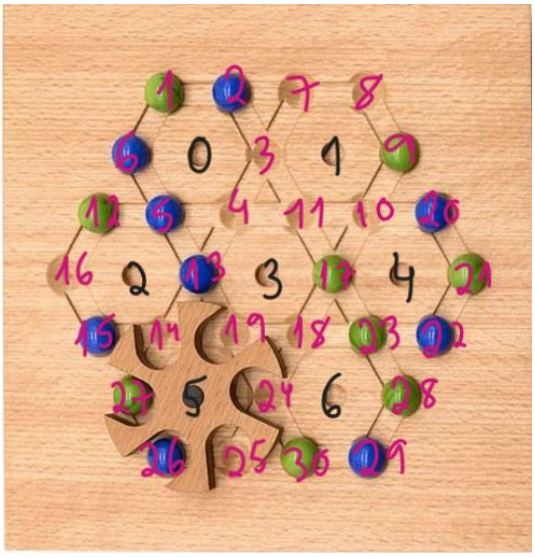

Imagem 1 – Representação da Board.

Legenda:
Preto – Número do Hexágono
Rosa – Número da Posição

A cada jogador estão atribuídas 9 bolas, sendo que as bolas do Jogador1 serão as Amarelas e as bolas do Jogador2 serão as Vermelhas. O objetivo do jogo é posicioná-las de modo a obter uma sequência de 6 bolas consecutivas.

Para cada jogada, estão estabelecidas algumas restrições:

1) O jogador só poderá escolher um hexágono valido (Valores entre 0 e 6);
2) Ao rodar o Hexágono este não pode ficar na mesma posição.

Lógica do Jogo:

Para representar o estado do jogo, recorremos ao argumento ***GameState***. O ***GameState*** é utilizado em todas as fases do nosso jogo, seja para garantir o começo da partida, a atualização do estado da ***Board*** ou até mesmo para terminar o Jogo.

Dessa forma o ***GameState*** é composto por 3 elementos:

1) ***Board***: Matriz com dimensões 7x6 que representa o estado inicial do Jogo. É importante salientar que cada linha representa um hexágono.

```
% Your original matrix.
board(I,[
[1,2,3,4,5,6],  % 0
[7,8,9,10,11,3], % 1
[12,5,13,14,15,16], % 2
[4,11,17,18,19,13], % 3
[10,20,21,22,23,17], % 4
[14,19,24,25,26,27], % 5
[18,23,28,29,30,24] % 6
]).
```

---

2) ***ChangeBoard***: Matriz com dimensoes 7x13 que representa a disposição dos elementos presentes na Board, só que na perspetiva do Jogador.

```
% My original matrix but in the Users Perspective
check(U,[
    [0,0,0,1,-,2,0,7,-,8,0,0,0],
    [0,0,6,0,0,0,3,0,0,0,9,0,0],
    [0,12,-,5,-,4,-,11,-,10,-,20,0],
    [16,0,0,0,13,0,0,0,17,0,0,0,21],
    [0,15,-,14,-,19,-,18,-,23,-,22,0],
    [0,0,27,0,0,0,24,0,0,0,28,0,0],
    [0,0,0,26,-,25,0,30,-,29,0,0,0]
]).
```

---

3) ***Player***: Jogador que está a realizar a jogada.

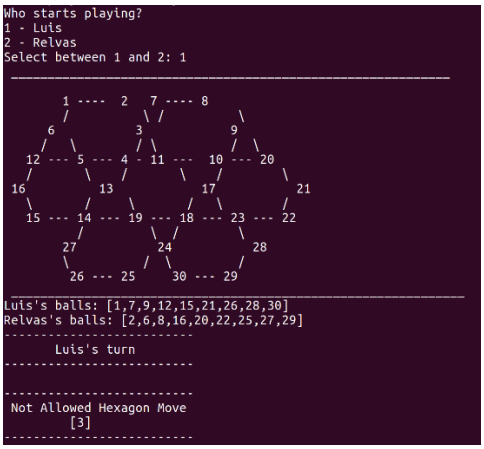

Imagem 2 – Representação do Estado Inicial da Board

Visualização do estado do jogo:

Assim que o predicado *play* é chamado, antes de ser iniciado o Jogo, o(s) utilizador(s) tem de configurar a partida que pretendem realizar. Dessa forma ser-lhe-ão pedidas informações relativamente aos seguintes campos:

1) Modo (Humano vs. Humano / Humano vs. Bot / Bot vs. Bot);
2) Nome dos Jogadores;
3) Dificuldade do Bot;
4) Qual será o Jogador que irá iniciar a partida;

Para todos os casos são realizadas verificações para o Input para garantir que os valores introduzidos pelo utilizador(s) são sempre válidos. A validação do input para o Modo de Jogo e para a dificuldade do Bot é garantida pelo predicado *get_input/4*.

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

Contrariamente, para garantirmos o input para o nome dos jogadores é utilizado o predicado *name_of/2*.

```
set_name(Player):-
    format('Hello ~a, what is your name? ', [Player]),
    read_string(Name, []),
    asserta(name_of(Player, Name)).
```

Após todas as verificações estarem concluídas está na altura de inicializarmos o estado inicial das nossas Matrizes Board e ChangeBoard e do Player que irá começar a jogar, utilizando o predicado *init_state/2*.

```
initial_state(Board,ChangeBoard):-
    board(I,Board),
    check(U,ChangeBoard).
```

Assim que o predicado *configurations/1* esteja completo poderemos então dar início ao Jogo!

```
configurations([Board,ChangeBoard,Player]):-
    flugelrad,
    set_mode,
    choose_player(Player),
    initial_random_state,
    initial_state(Board,ChangeBoard).
```

Relativamente á Visualização do Estado do Jogo, o predicado responsável por realizar esta ação é o *display_board/3*.Este predicado a cada movimento realizado no Jogo irá atualizar o seu estado dependendo sempre do ***Board***.

Este predicado irá percorrer todas as linhas e colunas da Matriz ***Board*** e irá construir uma nova Matriz ***ChangeBoard*** com alguns detalhes a nível de design para melhor perceção ao nível do Jogador.

Execução de Jogadas:

- O jogo estará sempre a correr o mesmo ciclo (*game_cycle\1*) e só parará em caso de vitoria de algum do(s) utilizador(s).

```
% game_cycle(+GameState)

game_cycle(GameState):-
    game_over(GameState,Winner),
    get_final(Choose),
    ((Choose == 0 ->
        clear_data,
        halt
    );
```

```
    reset_game).

% Game Loop when the game is not over
% game_cycle(+GameState)
game_cycle(GameState):-
    pos(V,K),
    set_list(Index,First),
    display_Player(GameState,Player),nl,
    retractall(connected(_,_)),
    valid_moves(GameState,ListOfMoves),
    display_Not_Allowed(GameState,ListOfMoves),
    choose_move(GameState,Number,Times,ListOfMoves),
    nth0(0,GameState,Board1),
    nth0(Number,Board1,T),
    common_elements(GameState, T, CommonPairs),
    hexagon_update(GameState),
    rotate_hexagon(Number,Times,RotatedList,GameState),
    move(T,GameState,CommonPairs,RotatedList,UpdatedBoard),
    NewGameState = [UpdatedBoard,ChangeBoard,Player],
    display_Board(NewGameState,NewBoard,0),
    NewGameState1 = [UpdatedBoard,NewBoard,Player],
    separate_lists_by_value(K,Alists,Vlists,_),
    separate_lists_by_value(K,Alists1,Vlists1,_),
    display_Balls(NewGameState1,Alists,Vlists),
    (((Player == First) ->
    (iterate_common(NewGameState1,Alists,Alists1,0)));
    iterate_common(NewGameState1,Vlists,Vlists1,0)),
    iterate_connected(NewGameState1),
    change_player(Player,NextPlayer),
    NewGameState2 = [UpdatedBoard,NewBoard,NextPlayer],
    game_cycle(NewGameState2).
```

- O *game_cycle/1* é composto por vários predicados, no entanto, o único que requer algum tipo de input é o *choose_move/3*.

    Se o(s) utilizador(s) é *Humano*: escolher o número do hexágono (***Number***) que pretende rodar e a quantidade de vezes que pretende rodar (***Times***).

    Se o(s) utilizador é *Bot Random*: o valor escolhido para o ***Number*** e para o ***Times*** será totalmente aleatório utilizando o predicado *random_betweenN/3* e *random_betweenT/3*, respetivamente, tendo sempre em consideração os valores válidos para a execução do movimento.

    Se o(s) utilizador é um *Bot Greedy*: a partir do predicado *value/1* irá prever qual será a melhor jogada de modo a conectar o maior número de bolas possível.

Independentemente do modo de jogo escolhido o procedimento após o predicado *choose_move/3* é sempre o mesmo.

- Após o predicado *choose_move/3*, recorremos ao predicado *common_elements/3*. Começamos então por verificar quais são os elementos em comum entre o hexágono escolhido e os restantes presentes na ***Board***. Serão guardados os valores comuns num *map-list* denominado por ***CommonPairs*** onde a Chave representa o número do hexágono e o Valor representa o Valor em comum.

- Por seguimento, encontramos dois predicados *hexagon_update/1* e *rotate_hexagon/4*. Estes são utilizados para: estabelecer uma concordância entre as posições do hexágono escolhido e as posições do hexágono que estão presentes na ***Board***; para exercer a rotação proveniente do predicado *choose_move/3* e armazenar na variável ***RotatedList***.

- Após a variável ***RotatedList*** estar atribuída podemos então fazer uma comparação entre as posições das bolas antes da rotação e depois da rotação, sendo responsável por esta ação o predicado *move/5*. Este predicado serve para modificar os valores dos elementos em comum, calculados anteriormente e guardados na map-list ***CommonPairs***, pelos novos valores que estão nas novas posições comuns. Após a iteração sobre todos os pares existentes no ***CommonPairs***, alteramos então o valor do nosso hexágono na ***Board*** pela ***RotatedList***, para perceberes melhor os predicados acima mencionados repara no seguinte exemplo:

Se considerarmos a Matriz ***Board*** acima definida e se o Hexagono escolhido for o número 0, *Hexagon(0,[1,2,3,4,5,6])*, então os seguintes pares farão parte da ***CommonPairs*** : *(1-3,2-5,3-4)*. De seguida, e considerando que o número de vezes que o hexágono vai rodar é igual a 3, então a ***RotatedList*** *= [4,5,6,1,2,3],* tendo em conta que a Rotação é feita no sentido contrário aos ponteiros do relógio. Agora, podemos então, comparar os valores do hexágono inicial *[1,2,3,4,5,6]* com os valores do hexagono após a rotação *[4,5,6,1,2,3]*. Verificamos que as posições em comum com os hexagonos 1, 2 e 3 possuem novos valores 6, 2 e 1. Para finalizar o predicado *move/5* vai fazer a atualização dos valores da ***Board*** e colocar na variável ***UpdatedBoard***.

- Aquando da alteração na ***Board*** das novas posições das bolas nos respetivos hexágonos, ação realizada pelo predicado *display_board/3*, verificamos então quais as bolas da mesma cor que estão conectadas utilizando o predicado *iterate_common/3*. Este, irá utilizar uma das seguintes listas, de acordo com a ordem de começo escolhida pelo utilizador:

1) ***Alists: [1,7,9,12,15,21,26,28,30];***
2) ***Vlists: [2,6,8,16,20,22,25,27,29];***

- O objetivo é então, fixando o ***Header*** , iterar sobre todas as possibilidades comparando com os restantes valores da Lista. Para garantir a conexão entre duas bolas é utilizado o predicado *check_iterate/3*. Este recebe da lista um ***Header*** e um ***Valor*** e, de seguida, vai verificar, tendo em conta as suas posições, se é possível estabelecer uma conexão entre as duas. Se for estabelecida uma conexão, atribuímos ao predicado *connected/2* o ***Valor*** e o ***Header*** da seguinte forma: *connected(Fixo,Valor)* e *connected(Valor,Fixo)* por se tratar de uma ligação bidirecional.

- Para concluir o predicado *iterate_connected/1* utilizando uma *depth-first-search* vai verificar se, tendo em conta as conexões atribuídas no predicado anterior, existem 6 bolas da mesma cor consecutivas.

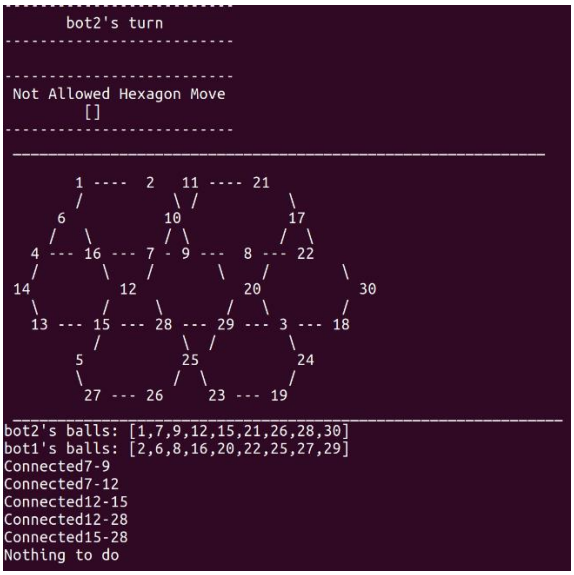

Imagem 3 -Representação Da Board durante o Jogo

Lista de Jogadas Válidas:

Flugelrad é um jogo que não requer restrições no movimento complexas. Desse modo as únicas restrições impostas para a realização do movimento são:

1) A escolha do número do hexágono e o número de vezes que pretendemos rodar o mesmo. Atenção que o Utilizador não pode movimentar um hexágono cujas posições estejam todas *'empty'.*

```
get_hexagon(GameState, Number,ListOfMoves) :-
    repeat,
    get_input(0,6,'Please choose the Number of the Hexagon you want to
rotate',Number),
    \+ member(Number,ListOfMoves),
    repeat.


get_number(GameState, Times) :-
```

```
    repeat,
    get_input(1,5,'Please choose how many times you want to rotate the
Hexagon',Times).
```

```
% Auxiliary predicade to help check If there are any hexagons with all
the elements empty

check_empty(Y, Board, Elists) :-
    nth0(Y, Board, Row),
    between(0, 5, X),
    nth0(X, Row, Value),
    \+ member(Value, Elists).


% predicade that receives a GameState and a List of Hexagons that are not
allowed to move because they have all the elements empty
% and returns in ListOfMoves the Hexagons that are allowed to move
% valid_moves(+GameState,-ListOfMoves)
valid_moves([Board, _, Player], ListOfMoves) :-
    pos(V, K),
    separate_lists_by_value(K, Alists, Vlists, Elists),
    % Initialize an empty list to accumulate values of Y
    findall(Y, (
        between(0, 6, Y),
        \+ check_empty(Y, Board, Elists)
    ), ListOfMoves).
```

Final de Jogo:

O final do jogo é verificado a cada iteração do *game_cycle/1* pelo predicado *game_over/2*. O jogo termina quando um jogador conseguir colocar 6 bolas da mesma cor em posições consecutivas.

```
game_over([_,_,Other],Winner) :-
    change_player(Other,Winner),
    ((winner(Winner,1) ; winner(Winner,1)) ->
        name_of(Winner, Name),
        write(' _____________________________________'),nl,
        write(' |                                   |'),nl,
        write(' |            YOU WON                |'),nl,
        write(' |                                   |'),nl,
        write(' |                                   |'),nl,
        write(' |                                   |'),nl,
        format('|     ~a won the game!\n            |', [Name]),
        write(' |                                   |'),nl,
        write(' |                                   |'),nl,
        write(' |                                   |'),nl,
        write(' |                                   |'),nl,
        write(' _____________________________________'),nl
        ).
```

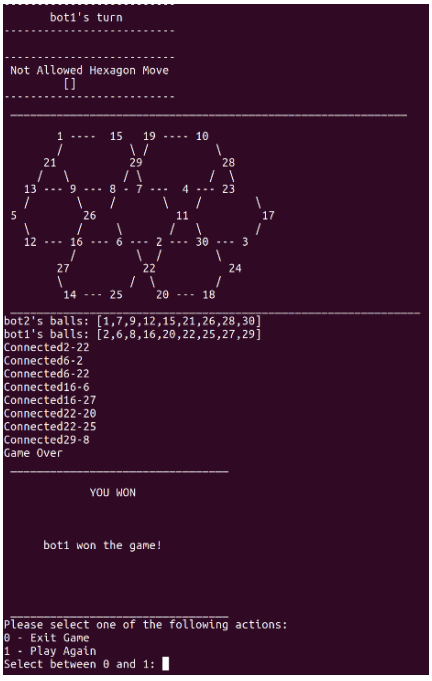

Imagem 4 – Representação Final do Jogo

Jogada do Computador:

O computador possui 2 níveis de dificuldade:

1) *Random*;
2) *Greedy*;

- Na dificuldade *Random* o Computador escolhe sempre jogadas aleatórias tendo sempre em conta as restrições de movimento.

- Na dificuldade *Greedy* o Computador irá escolher, auxiliado pelo predicado *value/1*, a jogada que conectará mais bolas consecutivas. Para atender a este pedido o Computador irá pesquisar sobre todas as jogadas possíveis e de seguida realizar essa jogada.

Os predicados chamados em *value/1* são iguais ao que são chamados no *game_cycle/1*. No entanto, existem algumas diferenças aquando da chamada dos mesmos.

- Nos predicados *display_Board/3*, *iterate_common/3* e *check_iterate/4* repara que no game_cycle/1 o último argumento destes predicados é chamado com o valor 0, por outro lado, no *value/1* é chamado com o valor 1. Esta diferença permite-nos saber de onde estamos a chamar o predicado. Desse modo, enquanto o Computador *Greedy* está a iterar até atingir a melhor jogada possível, não estamos a ocupar os nossos recursos com iterações que não tem relevância.

Como já referido anteriormente, é possível jogar contra o Computador em qualquer um dos modos. É também possível realizar um jogo entre dois computadores, sendo que, neste caso, o utilizador escolhe a dificuldade dos *Bots*.

Conclusão:

Todas as implementações feitas sobre os predicados têm por base os conhecimentos lecionados nas aulas teóricas e praticas da **UC - Programação Funcional e em Lógica**.

Ao longo de todo o processo, fomos capazes de desenvolver um pensamento diferente da vasta maioria de linguagens com que já interagimos anteriormente.

A nossa implementação do jogo cumpre os requisitos, isto porque, o jogo foi desenvolvido com sucesso. No entanto, há sempre espaço para melhorias:

1) O algoritmo *Greedy* utilizado no Computador na dificuldade 2: tendo em conta que é apenas capaz de prever a melhor jogada possível numa profundidade de 1, é incapaz de tornar as suas escolhas imprevisíveis e não lineares;

2) Não foi também possível desenvolver uma função concreta para avaliar o estado atual do ***Board***, uma vez que a definição de melhor jogada comparativamente com outra não é trivial.

Uma limitação do nosso projeto é o facto de não ser possível alterar o tamanho do ***Board***, visto que resultaria numa derivação do jogo inicialmente proposto relativamente às regras.

Bibliografia:

https://boardgamegeek.com/boardgame/400097/flugelrad