



Dualma Board Game

Programação Funcional e Lógica

Dualma_2

Licenciatura em Engenharia Informática e Computação

Diogo Sarmento up202109663@fe.up.pt Contribuição; 50% Rodrigo Póvoa up202108890@fe.up.pt Contribuição: 50%

Índice

1. Instalação e Execução
2. Descrição do Jogo
3. Lógica do Jogo 3.1 - Representação Interna do estado do Jogo 3.2 - Visualização do estado do Jogo 3.3 - Validação e execução dos movimentos 3.4 - Lista dos movimentos válidos 3.5 - Fim do Jogo 3.6 - Avaliação do tabuleiro 3.7 - Jogada do computador
4. Conclusões
5. Bibliografia

1. Instalação e execução

Antes de começar a jogar o jogo, é necessário fazer download do zip PFL_TP1_T05_Dualma_2.ZIP e dar unzip. Depois, abrir o terminal SICStus 4.8 e consultar o ficheiro main.pl. Tanto o Windows como o Linux funcionam. Recomenda-se aumentar o tamanho da fonte para uma melhor jogabilidade (Exemplo: Font: Consolas ; Font Style: Regular; Size: 20).

2. Descrição do Jogo

DuAlma é um jogo de tabuleiro estratégico para dois jogadores, cada um com 11 discos de cor preta ou branca, com o objetivo de criar uma fila de três discos da mesma cor no topo de uma pilha com altura 2. O jogo desenrola-se num tabuleiro hexagonal composto por 61 hexágonos, com 5 hexágonos de cada lado, e segue um conjunto distinto de regras.

O jogo está dividido em duas fases: "Placement Phase" e "Movement Phase". Durante a primeira fase, os jogadores alternam-se para colocar seus discos em qualquer espaço vazio no tabuleiro, exceto no centro. O Preto começa a colocar um disco de cada cor, seguido pelo Branco. O hexágono central permanece vazio durante toda esta fase. Assim que todos os discos são colocados, o jogo passa para a próxima fase.

Na 2ª fase, começando pelo Branco, os jogadores movem seus discos em linhas retas, cobrindo uma distância igual à altura da pilha da qual fazem parte. As pilhas podem consistir num único disco ou dois discos um em cima do outro, formando uma pilha com altura 2. Apenas o disco no topo de uma pilha é ativo e pode mover-se. Um disco pode pousar num hexágono vazio ou em cima de outro disco, independentemente da sua cor. No entanto, a altura total de uma pilha não pode exceder 2 discos, e nenhum disco pode sair do tabuleiro. Além disso, o disco no topo de uma pilha com altura 2 pode realizar um "salto", permitindo que se mova para qualquer espaço vazio adjacente.

A condição de vitória é alcançada quando um jogador forma uma fila de três pilhas com altura 2 da sua cor no topo. Esta vitória também pode ser alcançada se forem criadas várias filas de discos ativos da mesma cor com altura 2. Mesmo que mais de três pilhas com altura 2 com a cor do jogador sejam formadas, ainda assim conta como vitória. Para mais informações:

https://s3.amazonaws.com/geekdo-files.com/bgg368377?response-content-disposition=inline%3B%20filename%3D%22DuAlma.pdf%22&response-content-type=application%2Fpdf&X-Amz-Content-Sha256=UNSIGNED-PAYLOAD&X-Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-Credential=AKIAJYFNCT7FKCE4O6TA%2F20231102%2Fus-east-1%2Fs3%2Faws4_request&X-Amz-Date=20231102T142345Z&X-Amz-SignedHeaders=host&X-Amz-Expires=120&X-Amz-Signature=e2b0e267d54830c403a6ec58014f7ea9251eb73bad341725def27c7bd792f8bc Ou dentro do zip do projeto.

3. Lógica do Jogo

3.1 - Representação Interna do estado do Jogo

O estado do Jogo, GameState, é formado por:

- Board: Matriz "Hexagonal" com tamanho indicado pelo utilizador no menu. Contém átomos a representar espaço vazio(empty), átomos a representar a cor das peças e se é uma stack de 1 ou 2 (white, black, whiteblack, blackwhite, whitewhite, blackblack).
- Player: Jogador que joga o turno atual (1 ou 2).

Para além disso, também estamos a usar Dynamic para guardar outros "factos" que também são importantes para o jogo e não faz sentido representar na GameState. É o caso do Nome do jogador, Dificuldade do bot, nº de discos a ser colocados no tabuleiro, jogador branco e jogador preto.

```
% name_of_the_player(+Player, -Name)
% Find the Players name
:- dynamic name_of_the_player/2.

% bot_difficulty(+Bot, -Difficulty)
% Find the Bot difficulty
:- dynamic bot_difficulty/2.
```

```

% disks(+Player, -Disks)
% Finds the number of disks yet to be placed
:- dynamic disks/2.

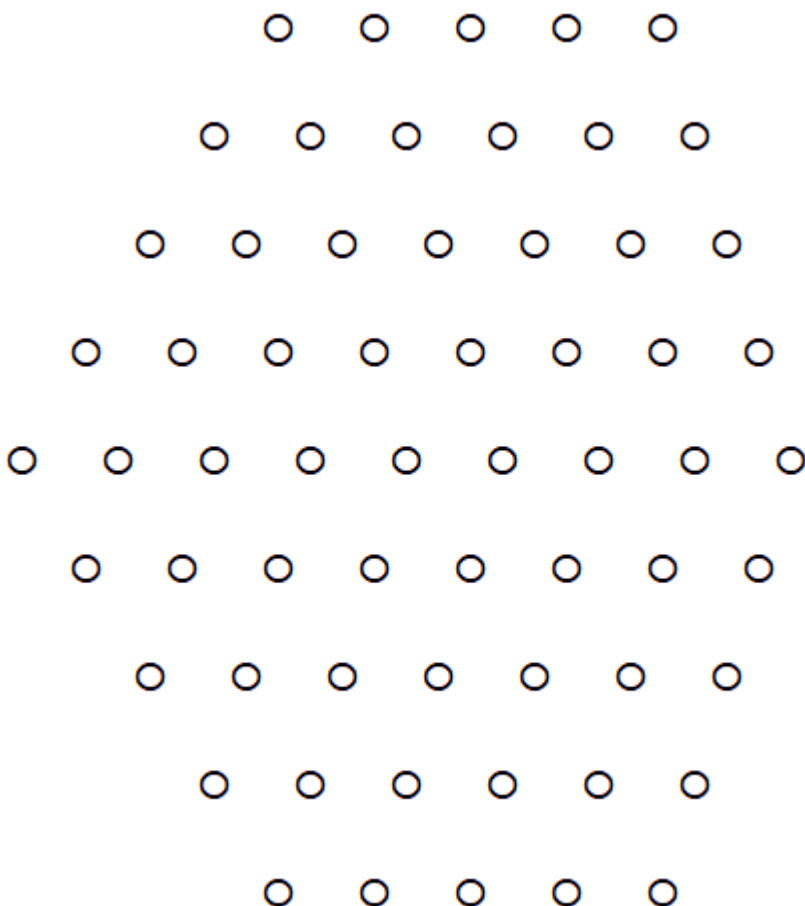
% white(+Player)
% Finds which Player is White
:- dynamic white/1.

% black(+Player)
% Finds which Player is Black
:- dynamic black/1.

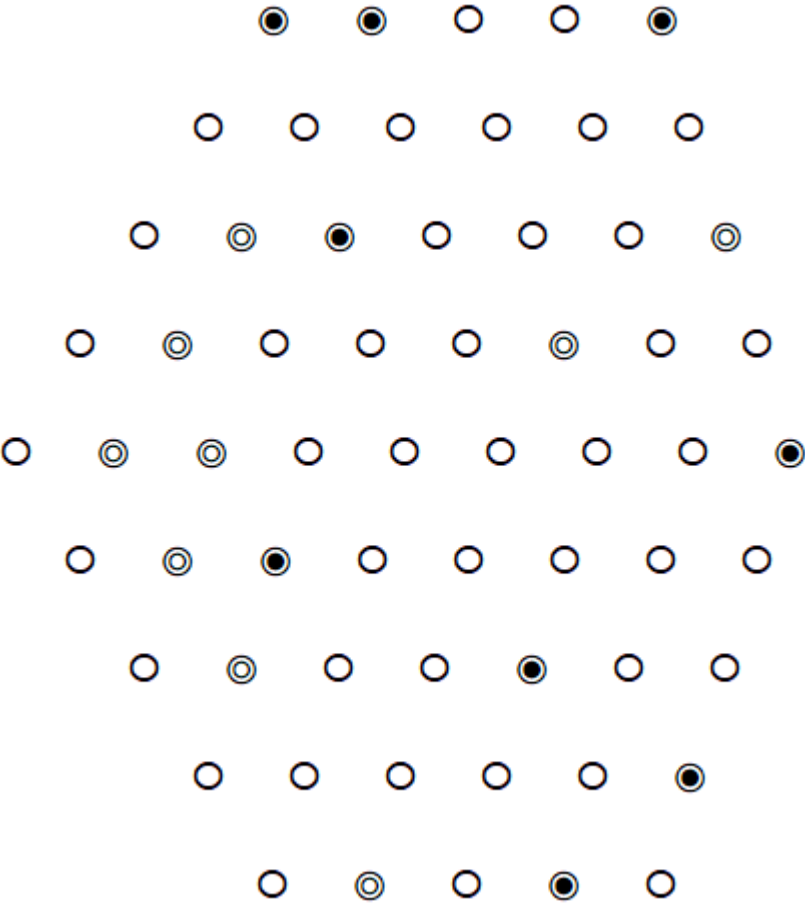
% board(+Size,+Matrix)
board(5,[
    [empty, empty, empty, empty, empty],
    [empty, empty, empty, empty, empty, empty],
    [empty, empty, empty, empty, empty, empty, empty],
    [empty, empty, empty, empty, empty, empty, empty, empty],
    [empty, empty, empty, empty, empty, empty, empty, empty, empty],
    [empty, empty, empty, empty, empty, empty, empty, empty],
    [empty, empty, empty, empty, empty, empty, empty],
    [empty, empty, empty, empty, empty]
]).

```

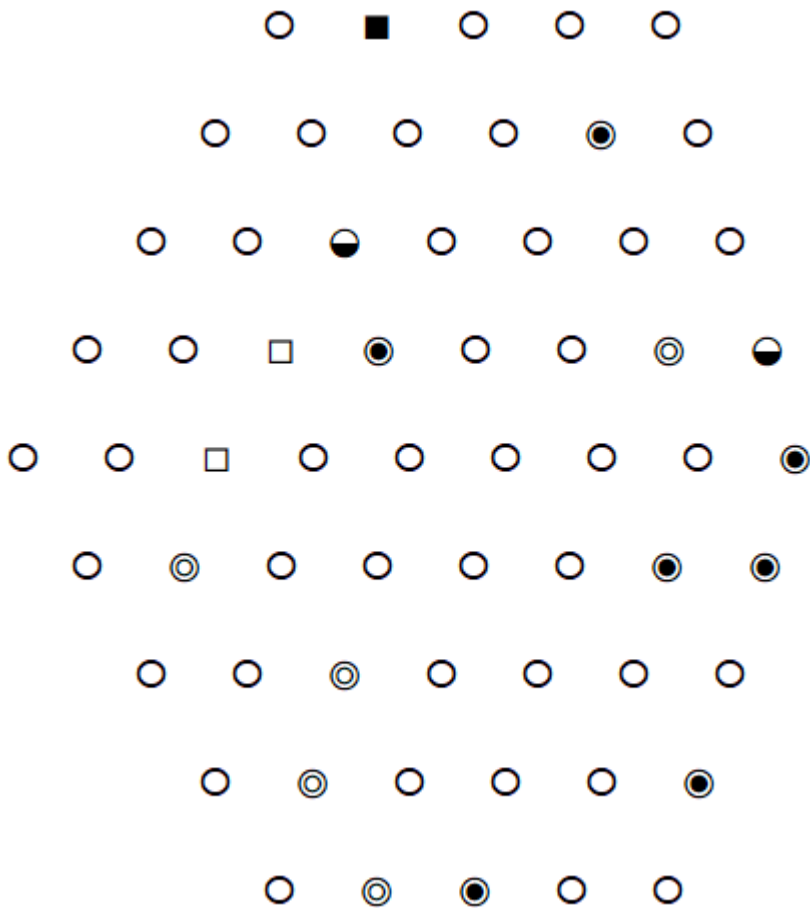
GameState Inicial:



GameState Intermédio



GameState Final:



3.2 - Visualização do Estado do Jogo

Ao correr `play`, no terminal, é apresentado aos jogadores uma tela, onde selecionam o modo do jogo a jogar, que pode ser `PlayerVsPlayer`, `PlayerVsBot`, `BotVsBot`. De seguida pedem os nomes dos jogadores, se existir. Posteriormente, vai pedir o nível de dificuldade e o jogador que inicia a partida, bem como o tamanho do tabuleiro.

Nos inputs em que se pede um número, é feita a validação usando o `between()`. Escrever qualquer outro número que não seja válido leva à repetição do input. Nos inputs em que é pedido Nome ou uma coordenada (ex: a1) é usada o predicado `getLine(Result,Acc)` para filtrar.

Para obter o input do `GameMode`, `BotDifficulty`, `StartPlayer` e `BoardSize`, utilizamos os seguintes predicados que usam o `read()`.

```
% getPlayerName(-Player)
% Asks the player for his name and returns it in Player
getPlayerName(Player):-
    format('\nHello ~a, what is your name? ', [Player]),
    getLine(Name, []),
    asserta(name_of_the_player(Player, Name)),
    clear_input.

% getGameMode(-GameMode)
% Asks the player for the game mode and returns it in GameMode
```

```

getGamemode(Gamemode):-
    repeat,
    read(Gamemode),
    between(1,3,Gamemode),
    clear_input,
    !.

% getBotDifficulty(+Bot)
% Asks the player for the bot difficulty and applies it to Bot
getBotDifficulty(Bot):-
    repeat,
    read(BotDifficulty),
    between(1,2,BotDifficulty),
    asserta(bot_difficulty(Bot, BotDifficulty)),
    clear_input.

% getStartPlayer(-StartPlayer)
% Asks the player for the start player and returns it in StartPlayer
getStartPlayer(StartPlayer):-
    repeat,
    read(StartPlayer),
    between(1,2,StartPlayer),
    asserta(white(StartPlayer)),
    (StartPlayer == 1 -> asserta(black(2)); asserta(black(1))),
    clear_input.

% getBoardSize(-Size)
% Asks the player for the board size and returns it in Size
getBoardSize(Size):-
    repeat,
    write('\nChoose the size of the board (5 or 6):'),
    read(Size),
    member(Size, [5,6]),
    clear_input.

```

Já para obter o nome do jogador utilizamos o `getLine` em vez do `read()` para aceitar caracteres até ao `newline`.

```

% getLine(-Result, +Acc)
% Reads a line from the input and returns it in Result
getLine(Result, Acc) :-
    get_char(Char),
    Char \= '.',
    Char \= '\n',
    append(Acc, [Char], Acc1),
    getLine(Result, Acc1).
getLine(Result, Acc) :-
    atom_chars(Result, Acc).

% getPlayerName(-Player)
% Asks the player for his name and returns it in Player

```

```
getPlayerName(Player):-
    format('\nHello ~a, what is your name? ', [Player]),
    getLine(Name, []),
    asserta(name_of_the_player(Player, Name)),
    clear_input.
```

Após escolhermos o tamanho do tabuleiro, inicializamos o board com o `init_state/2`. Existem 2 tamanhos de tabuleiro: um hexágono de 5x5 de lado ou 6x6 de lado.

```
% configuration(-GameState)
% Initialize GameState with Board and Player Turn
configurations([Board,Player]):-
    drawStartMenu,
    getGamemode(Gamemode),
    !,
    option(Gamemode),
    startPlayer(Player),
    getBoardSize(Size),
    init_state(Size,Board).
```

Após configurarmos o `GameState` e os predicados necessários, podemos começar a imprimir o tabuleiro do nosso jogo. Cada vez que é feita uma interação, o board é atualizado.

```
% display_game(+GameState)
% Displays the board.
display_game([Board,_]):-
    length(Board, BoardLength),
    True_BoardLength is (BoardLength + 1) ,
    print_board_rows(Board,1,True_BoardLength), nl,
    true.
```

O predicado `print_board_rows` vai ser responsável por imprimir cada linha do tabuleiro, tendo em conta o seu formato hexagonal e que tipo de 'estado' está cada uma das posições, isto é, se tem uma peça branca, uma peça preta... usando o predicado `write_hexagon_tile/1`.

```
print_board_rows(_, NumRows, NumRows):-!.
print_board_rows(Board, NumRows, MaxRows):-
    length(Board, BoardLength),
    NumSpaces is abs(NumRows - (BoardLength + 1) // 2),
    NumSpaces1 is NumSpaces*2,
    nl,
    print_spaces(NumSpaces1),
    nth1(NumRows, Board, Row),
    write_row(Row),
    nl,
    NextNumRows is NumRows + 1,
    print_board_rows(Board, NextNumRows, MaxRows).
```

```

% print_spaces(+N)
% Prints N spaces.
print_spaces(0).
print_spaces(N):-
    N > 0,
    write(' '),
    NextN is N - 1,
    print_spaces(NextN).

% write_row(+Row)
% Writes a row of the board.
write_row([]).
write_row([H|T]):-
    write_hexagon_tile(H),
    write(' '),
    write_row(T).

% write_hexagon_tile(+Tile)
% Writes a tile of the board according to the atom.
write_hexagon_tile(empty):-
    put_code(9711).
write_hexagon_tile(white):-
    write('w').
write_hexagon_tile(black):-
    write('b').
write_hexagon_tile(whiteblack):-
    write('g').
write_hexagon_tile(blackwhite):-
    write('G').
write_hexagon_tile(whitewhite):-
    write('W').
write_hexagon_tile(blackblack):-
    write('B').

```

3.3 - Validação e execução dos movimentos

O nosso jogo contém 2 ciclos de jogo, um para cada fase (Placement e Movement), onde o caso de paragem no primeiro ciclo é ausência de peças para jogar e no segundo ciclo é a vitória de um dos jogadores.

Primeiro Ciclo:

```

game_cycle_first_phase(Gamestate, NewGamestate4):-
    clear,
    display_game(Gamestate),
    (disks(WhitePieces, BlackPieces), WhitePieces = 0, BlackPieces = 0) ->
        write('No more pieces available for both players. Moving to the next
phase. '), nl,
        NewGamestate4 = Gamestate
;

```



```

print_stats(Gamestate),
put_piece_input(Gamestate, Put, white),
put_piece_move(Gamestate, Put, NewGamestate, white),
display_game(NewGamestate),
put_piece_input(NewGamestate, Put2, black),
put_piece_move(NewGamestate, Put2, NewGamestate2, black),
change_player(NewGamestate2, NewGamestate3),
game_cycle_first_phase(NewGamestate3, NewGamestate4).

```

O `print_stats` serve para indicar quem é que joga naquele turno, enquanto que o `put_piece_input` pede e valida o input do utilizador para colocar uma peça. Já o `put_piece_move` coloca a peça no tabuleiro e vai “atualizar” o tabuleiro. Finalizando, é feita a troca do turno através do `change_player`.

Consideramos um movimento válido se:

- A linha (row) está dentro dos limites, tal como a coluna (column) , embora seja aplicada um bocado de matemática devido ao formato hexagonal do tabuleiro.
- Seja uma posição ‘empty’, isto é, não está já colocada outra peça no local.
- Não seja o centro do tabuleiro.

Segundo Ciclo:

```

game_cycle_second_phase(Gamestate):-
    display_game(Gamestate),
    check_game_over(Gamestate, Winner),
    winner_screen(Gamestate, Winner).

game_cycle_second_phase(Gamestate):-
    print_stats(Gamestate),
    move_piece(Gamestate, NewGamestate),
    change_player(NewGamestate, NewGamestate2),
    game_cycle_second_phase(NewGamestate2).

```

O primeiro predicado corresponde ao fim do jogo, isto é, se alguém chegou a uma condição de vitória (3 in line with stack) e mostra o ecrã do fim do jogo juntamente com quem ganhou o jogo.

O segundo predicado corresponde ao movimento das peças existentes no tabuleiro e a sua validação.

O conjunto de movimentos válidos são aqueles que respeitam as seguintes condições:

- Encontra-se dentro dos limites do tabuleiro.
- Na posição inicial, tem que existir uma peça 'ativa' que seja da mesma cor que o jogador que vai jogar.
- Na posição final, tem que ter espaço livre para colocar mais uma peça.
- Deve encontrar-se a uma distancia de 1 u.c ou 2 u.c , se a peça a ser movida estiver numa stack.

3.4 - Lista de Jogadas Válidas

Na primeira fase, a lista de jogadas válidas é obtida a partir do predicado `valid_moves/2`, usando o `findall/3`. Só chamamos este predicado no caso dos Bots, já que os jogadores humanos vão inserir as suas peças.

```

% valid_moves(+Board, -ListOfMoves)
% Gets all the valid moves for a board
valid_moves(Board, ListOfMoves):-
    length(Board, Size),
    Upper is 97 + Size,
    findall([R, C], (
        between(1,Size,C),
        between(97, Upper, R),
        valid_row(R, Size),
        valid_column(C, R, Size),
        not_in_center([R, C], Size),
        is_empty(Board, R, C)
    ), ListOfMoves).

```

No caso da segunda fase, usamos o `valid_moves_second_cycle/3`, usando o `findall/3`. Só chamamos este predicado no caso dos Bots, já que os jogadores humanos vão inserir as suas peças. Primeiramente, verificamos se existe a peça naquela posição, se é da mesma cor do jogador, se é possível mexê-lo e se o destino está 'livre'.

```

valid_moves_second_cycle(Board, Player, ListOfMoves):-
    white(Player),
    length(Board, Size),
    Side_Size is ((Size+1)//2),
    findall([R-C,R1-C1], (
        between(1,Size,C),
        between(1, Size, R),
        inBoard_int(R-C,Board),
        between(1,Size,C1),
        between(1,Size,R1),
        inBoard_int(R1-C1,Board),
        find_values(whiteblack, whitewhite, Board, Indices),
        find_values(white, white, Board, Indices2),
        append(Indices,Indices2,Indices3),
        member(R-C,Indices3),
        check_if_piece_is_from_player(R-C,Player,Board,Range),
        get_distance_in_line(R-C,R1-C1,Side_Size,Range1),
        Range1 == Range,
        check_if_destination_is_clear_or_has_one_level(R1-C1,Board)
    ), ListOfMoves).

valid_moves_second_cycle(Board, Player, ListOfMoves):-
    black(Player),
    length(Board, Size),
    Side_Size is ((Size+1)//2),
    findall([R-C,R1-C1], (
        between(1,Size,C),
        between(1, Size, R),
        inBoard_int(R-C,Board),
        between(1,Size,C1),
        between(1,Size,R1),

```

```

inBoard_int(R1-C1,Board),
find_Values(blackblack, blackwhite, Board, Indices),
find_Values(black, black, Board, Indices2),
append(Indices,Indices2,Indices3),
member(R-C,Indices3),
check_if_piece_is_from_player(R-C,Player,Board,Range),
get_distance_in_line(R-C,R1-C1,Side_Size,Range1),
Range1 == Range,
check_if_destination_is_clear_or_has_one_level(R1-C1,Board)
), ListOfMoves).

```

3.5 - Fim do Jogo

No segundo ciclo do jogo, verificamos se existem condições para uma vitória para um dos jogadores, isto é, se existe pelo menos um 3 em linha de 2 stack. Caso exista, o jogo acaba e impresso no ecrã o tabuleiro final e o vencedor do jogo.

```

game_cycle_second_phase(Gamestate):-
    display_game(Gamestate),
    check_game_over(Gamestate,Winner),
    winner_screen(Gamestate,Winner).

```

3.6 Avaliação do tabuleiro

No primeiro ciclo, a cada jogada verificamos se a peça que o bot vai colocar é da mesma cor que o bot. Caso seja, o algoritmo procurar uma posição válida em que a soma das distâncias relativamente a outras peças seja a mais baixa possível, de modo a ter um melhor posicionamento. Já se a cor for diferente, o bot vai usar um algoritmo semelhante mas nesse caso vai procurar em que a soma das distancias seja a máxima (usamos o sort/2 para encontrar o mínimo ou o máximo)

```

% calculate_distances(+EmptyRow, EmptyColumn, +OwnPieces, -Distances)
% Calculates the distance between a tile and all the pieces of a player
calculate_distances([EmptyRow, EmptyColumn], OwnPieces, Distances):-
    findall(Distance, (
        member([OwnRow, OwnColumn], OwnPieces),
        manhattan_distance([OwnRow, OwnColumn], [EmptyRow, EmptyColumn], Distance)
    ), Distances).

% evaluate_proximity(+EmptyPositions, +OwnPieces, -BestMove)
% Evaluates the best move to place a piece
evaluate_proximity(EmptyPositions, OwnPieces, BestMove):-
    findall([SumDistances, [Row, Column]], (
        nth1(_, EmptyPositions, [Row, Column]),
        calculate_distances([Row, Column], OwnPieces, Distances),
        sum_list(Distances, SumDistances)
    ), Moves),
    sort(Moves, SortedMoves),

```

```

write('Sorted Moves: '), write(SortedMoves), nl,
nth1(_, SortedMoves, BestMove).

% manhattan_distance(+Position1, +Position2, -Distance)
% Calculates the manhattan distance between two positions
manhattan_distance([Row1, Column1], [Row2, Column2], Distance):-
    Distance is abs(Row1 - Row2) + abs(Column1 - Column2).

```

DIOGO AQUÍ

3.7 Jogada Do Computador

Foram criadas 2 dificuldades de bot: Random e Hard.

Na primeira fase, o Bot Random, escolhe de forma random uma das jogadas possíveis e válidas a partir da lista de movimentos válidos.

```

% getCoordsRandom(-[Row, Column],+Board)
% Gets a random position to put a piece
getCoordsRandom([Row, Column],Board):-
    valid_moves(Board, ListOfMoves),
    random_member([Row,Column],ListOfMoves).

```

No caso do Bot Hard, é utilizado o algoritmo explicado no capítulo anterior, priorizando colocar as peças da própria cor o mais perto possível das outras e do centro, e fazer o contrário se tiver peças de outra cor.

```

getCoordsHard([Row, Column], [Board, Player], Color):-
    (white(Player) -> ColorPlayer = white; black(Player) -> ColorPlayer = black),
    (Color = ColorPlayer ->
        get_own_pieces(Board, OwnPieces, ColorPlayer),
        (OwnPieces = [] ->
            length(Board, Size),
            CenterRow is 97 + Size // 2,
            CenterColumn is Size // 2 + 1,
            choose_random_move([CenterRow, CenterColumn], [Row, Column]),
            format('Best Move: ~w\n', [[Row, Column]]))
        ;
        valid_moves(Board, EmptyPositions),
        evaluate_proximity(EmptyPositions, OwnPieces, [_, [Row, Column]]),
        format('Best Move: ~w\n', [[Row, Column]]))
    )
;
(ColorPlayer = white -> ColorAdversary = black; ColorAdversary = white),
get_own_pieces(Board, OpponentPieces, ColorAdversary),
(OpponentPieces = [] ->
    length(Board, Size),
    CenterRow is 97 + Size // 2,
    CenterColumn is Size // 2 + 1,

```

```

        choose_random_move_other( Board, [Row, Column]),
        format('Best Move: ~w\n', [[Row, Column]])
    ;
    write('Own Pieces: '), write(OpponentPieces), nl,
    valid_moves(Board, EmptyPositions),
    findall([DistanceNeg, [Row, Column]], (
        member([Row, Column], EmptyPositions),
        calculate_distances([Row, Column], OpponentPieces, Distances),
        sum_list(Distances, Distance),
        DistanceNeg is -Distance % Negate distance for sorting
    ), Moves),
    sort(Moves, SortedMoves),
    nth1(_, SortedMoves, [_, [Row, Column]]),
    format('Best Move: ~w\n', [[Row, Column]])
)
).

%choose_random_move_other(+Board, -[Row, Column])
% Chooses a random tile to place a piece
choose_random_move_other( Board, [Row, Column]):-
    length(Board, Size),
    ExtremeCorner is 97 + Size - 1,
    ExtremeCorner2 is ceiling(Size/2),
    Corner1 = [97, 1],
    Corner2 = [97, ExtremeCorner2],
    Corner3 = [ExtremeCorner, 1],
    Corner4 = [ExtremeCorner, ExtremeCorner2],
    Corners = [Corner1, Corner2, Corner3, Corner4],
    random_member([Row, Column], Corners).

%choose_random_move(+[CenterRow, CenterColumn], -[Row, Column])
% Chooses a random tile to place a piece around the center of the board
choose_random_move([CenterRow, CenterColumn], [Row, Column]):-
    SurroundingTiles = [[-1, 0], [1, 0], [0, -1], [0, 1], [-1, -1], [1, -1]],
    random_member([RRow,RColumn],SurroundingTiles),
    Row is CenterRow + RRow,
    Column is CenterColumn + RColumn.

```

DIOGO AQUÍ

Conclusões

Embora com algumas dificuldades iniciais e sem acesso a um debug decente, conseguimos implementar o nosso jogo Dualma. Além disso conseguimos ser criativos e conseguimos fazer não só com tabuleiro de tamanho fixo mas dinâmico,isto é 2 tabuleiros diferentes que usam o mesmo código, o que aumentou a complexidade do programa, para além de ter que fazer basicamente 2 AI Hard diferentes (um para cada fase). Acreditamos que conseguimos demonstrar o nosso empenho neste trabalho, aplicando os conhecimentos obtidos nas aulas teóricas, práticas e estudo individual.

Bibliografia

Funcionamento do jogo, bem como o tabuleiro (png):

<https://boardgamegeek.com/boardgame/398462/dualma/files>