U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Stacks

Trabalho Prático 1

Programação Funcional e Lógica

Turma 4 - Stacks_4

Carlos Manuel da Silva Costa (up202004151@fe.up.pt) – Contribuição: 50%

Tomás Henrique Ribeiro Coelho (up202108861@fe.up.pt) – Contribuição: 50%

3 de novembro de 2023

Índice

Instalação e Execução

Para ambos os ambientes, Windows ou Linux, primeiramente fazer o download dos ficheiros no zip PFL_TP1_T04_Stacks4 e descompactar. Consultar o ficheiro stacks.pl ou através da linha de comandos ou através do SICSTUS. Após consultar o ficheiro, basta usar o predicado play/0.

```
| ?- play.
```

Descrição do Jogo

O jogo é jogado num tabuleiro de 5x5, com um total de 25 espaços. Cada jogador recebe 10 peças, que podem ser vermelhas ou pretas. As peças começam empilhadas em grupos de 2 nos respetivos espaços iniciais, adjacentes ao tabuleiro principal. O objetivo do jogo é colocar quatro ou mais das suas peças nos espaços iniciais do adversário ou eliminar seis ou mais das peças do adversário. Além disso, é importante referir que, apesar de ser descrito como um tabuleiro de 5x5, na prática funciona como um tabuleiro de 7x5, uma vez que os espaços iniciais desempenham um papel crucial no jogo.

Este jogo tem bastantes regras que os utilizadores devem cumprir e que foram implementadas, aqui estão as principais:

1 - O primeiro jogador a jogar é sempre o jogador que tem as peças Pretas.
3 - Na vez de um jogador, ele tem a opção de jogar com uma stack de 1, 2, 3 peças e fazer, respetivamente, 3, 2, 1 movimentos. Ou então pode também mexer duas peças individuais, mas apenas 2 espaços cada.
4 - As stacks podem ir até 4 peças, sendo que stacks de 4 não podem ser movidas como uma inteira.
5 - Um movimento pode ser feito para qualquer direção vertical, horizontal ou diagonal, bem como qualquer combinação destas direções.
6 - Novas stacks podem ser criadas ao mexer uma stack para um quadrado onde já exista uma stack pertencente ao jogador (dar stack).
7 - Novas stacks também podem ser criadas ao mexer peças de uma stack existente e não mexer a stack inteira (dar destack).
8 - É possível fazer ataques sozinhos a peças inimigas movendo para a casa da stack inimiga ou então fazer ataques combinados com as outras peças do jogador desde que,

antes do último movimento de ataque, a peça que ataca esteja adjacente à peça inimiga, e também adjacente às outras peças com as quais pretende combinar. O resultante destes ataques é a diferença entre o "poder" da peça atacada e da(s) que ataca(m).
9 – O jogo termina quando um dos jogadores tiver eliminado 6 ou mais peças inimigas, ou então quando chegar às home spaces do inimigo com quatro ou mais peças.

A lista completa de regras pode ser encontrada em STACKS_Rules_of_Play.pdf, retirada de Stacks | Stacks: Rules of Play | File | BoardGameGeek.

Lógica do Jogo

Representação interna do estado de jogo

O nosso GameState representa o estado atual do jogo. É composto por uma lista de elementos que representam o tabuleiro atual. Em cada ciclo de jogo para além do tabuleiro também é indicado qual é o jogador atual que está a jogar, sendo este RED ou BLACK. Cada elemento da lista que representa o tabuleiro pode ter uma das seguintes opções: empty, black(1), black(2), black(3), black(4), red(1), red(2), red(3), red(4). Empty significa que nenhuma peça se encontra naquele espaço. Black(1) significa que uma stack de apenas 1 peça preta se encontra naquele espaço e assim sucessivamente. Ao dar display para o utilizador, deixamos o espaço em branco para representar um espaço vazio e para os restantes usamos, respetivamente, B1,B2,B3,B4,R1,R2,R3,R4 para representar tanto a cor como o número de peças na stack. As linhas do tabuleiro vão de 1 a 7 e as colunas de "a" a "e" para tornar o input mais user-friendly. Exemplos:

Estado inicial:

```
initial_state([
    [red(2), red(2), red(2), red(2), red(2)],
    [empty, empty, empty, empty, empty],
    [empty, empty, empty, empty, empty],
    [empty, black(1), red(1), empty, empty],
    [empty, black(1), empty, black(1), black(1)],
    [empty, empty, empty, empty, empty],
    [black(2), black(1), black(2), black(2), black(2)]
]).
```

```
    a     b     c     d     e
  -------------------------------
1 | R2  | R2  | R2  | R2  | R2  |   Max movement per stack:
  -------------------------------
2 |     |     |     |     |     |
  -------------------------------
3 |     |     |     |     |     |   Stack of 4: cannot move
  -------------------------------
4 |     |     |     |     |     |   Stack of 3: one space
  -------------------------------
5 |     |     |     |     |     |   Stack of 2: two spaces
  -------------------------------
6 |     |     |     |     |     |   Stack of 1: three spaces
  -------------------------------
7 | B2  | B2  | B2  | B2  | B2  |
  -------------------------------
CURRENT PLAYER:black
```

Estado intermediário:

```
initial_state([
    [empty, red(2), empty, black(1), red(1)],
    [empty, black(1), empty, empty, red(1)],
    [empty, empty, empty, empty, empty],
    [red(1), empty, empty, empty, empty],
    [empty, black(1), black(2), empty, black(2)],
    [black(1), empty, empty, red(1), red(1)],
    [empty, red(1), red(1), red(1), empty]
]).
```

```
    a     b     c     d     e
  -------------------------------
1 |     | R2  |     | B1  | R1  |   Max movement per stack:
  -------------------------------
2 |     | B1  |     |     | R1  |
  -------------------------------
3 |     |     |     |     |     |   Stack of 4: cannot move
  -------------------------------
4 | R1  |     |     |     | B1  |   Stack of 3: one space
  -------------------------------
5 |     | B1  | B2  |     | B2  |   Stack of 2: two spaces
  -------------------------------
6 | B1  |     |     | R1  | R1  |   Stack of 1: three spaces
  -------------------------------
7 |     | R1  | R1  | R1  |     |
  -------------------------------
CURRENT PLAYER:red
```

Estado final:

```
initial_state([
    [empty, red(1), empty, black(1), red(1)],
    [empty, black(1), black(1), empty, red(2)],
    [empty, empty, empty, empty, empty],
    [empty, empty, empty, red(1), black(1)],
    [empty, empty, empty, empty, black(2)],
    [black(1), empty, empty, empty, red(1)],
    [black(2), red(2), red(1), red(1), empty]
]).
```

```
    a    b    c    d    e
  -------------------------
1 |    | R1 |    | B1 | R1 |   Max movement per stack:
  -------------------------
2 |    | B1 | B1 |    | R2 |
  -------------------------
3 |    |    |    |    |    |   Stack of 4: cannot move
  -------------------------
4 |    |    |    | R1 | B1 |   Stack of 3: one space
  -------------------------
5 |    |    |    |    | B2 |   Stack of 2: two spaces
  -------------------------
6 | B1 |    |    |    | R1 |   Stack of 1: three spaces
  -------------------------
7 | B2 | R2 | R1 | R1 |    |
  -------------------------
Red Wins!
```

Visualização do estado do jogo:

Após iniciar o jogo o utilizador é apresentado o seguinte menu, onde é possível escolher que tipo de jogo quer fazer, humano contra humano, humano contra o computador ou então computador contra computador.

```
| ?- play.
Welcome to Stacks!
1. Human vs Human
2. Human vs Bot
3. Bot vs Bot
Select an option (1/2/3):
```

Em qualquer dos casos em que esteja envolvido um computador, imediatamente a seguir é apresentada ao utilizador a possibilidade de escolher a dificuldade do computador, esta que pode ser 1 ou 2, sendo a dificuldade 1 respetiva a um computador que faz moves escolhidos à sorte e a dificuldade 2 respetiva a um compuador que faz moves consoante o nosso algoritmo greedy implementado.

```
Select an option (1/2/3): 3.
Difficulty:
1 - Random Moves
2 - Greedy Moves
```

Em qualquer caso em que haja algum tipo de input por parte do utilizador este é verificado e validado, sendo que, se o input colocado for incorreto, é pedido um novo input.

```
Difficulty:
1 - Random Moves
2 - Greedy Moves
|: 6.
Invalid input.
Difficulty:
1 - Random Moves
2 - Greedy Moves
|:
```

Para ser mostrado o tabuleiro ao utilizador, é utilizado o predicado display_game/1:

```
display_game(Board) :-
    nl,
    display_columns,
    display_rows(Board, 1),
    display_separator.
```

O display_columns atribui letras a cada coluna. O display_rows constrói, de forma recursiva, as linhas do tabuleiro tendo em conta cada peça que consta na lista.

```
% display_columns/0
% Display the column labels on the console.
display_columns :-
    write('     a    b    c    d    e\n').

% display_separator/0
% Display a separator line on the console.
display_separator :-
    write('  -------------------------\n').


% display_rows(+Board, +RowNumber)
% Display the rows of the game board, including piece positions and a movement restriction tip in the right side.
display_rows([], 8).
display_rows([Row | Rest], RowNumber) :-
    display_separator,
    write(RowNumber),
    write(' '),
    display_row(Row),
    (RowNumber =:= 1 -> write('   Max movement per stack: ') ; true),
    (RowNumber =:= 3 -> write('   Stack of 4: cannot move') ; true),
    (RowNumber =:= 4 -> write('   Stack of 3: one space') ; true),
    (RowNumber =:= 5 -> write('   Stack of 2: two spaces') ; true),
    (RowNumber =:= 6 -> write('   Stack of 1: three spaces') ; true),
    nl,
    NextRowNumber is RowNumber + 1,
    display_rows(Rest, NextRowNumber).
```

Validação de jogadas e Execução

O jogo funciona à base de ciclos sendo que em cada ciclo é indicado o jogador que está a jogar atualmente, é escolhido um movimento válido, o movimento é feito, gerando um novo GameState, o próximo jogador a jogar é trocado e é apresentado o novo tabuleiro. Após tudo isto é inicializado um novo ciclo. A cada novo ciclo é verificado se o jogo terminou de alguma das duas maneiras possíveis.

```
game_cycle(GameState-Player):-
    game_over(GameState, Winner), !.

game_cycle(GameState-Player):-
    write('CURRENT PLAYER:'),
    write(Player), nl,
    choose_move(GameState, Player, Move, TwoMovesGamestate),
    move(TwoMovesGamestate, Move, NewGameState),
    next_player(Player, NextPlayer),
    display_game(NewGameState),
    !,
    game_cycle(NewGameState-NextPlayer).
```

No caso do nosso jogo, o utilizador pode tomar dois caminhos distintos quando quer fazer a sua jogada, se quer mover apenas uma stack de 1, 2 ou 3 (até 3, 2, 1 movimentos, respetivamente) ou então se quer mover duas stacks de 1 até 2 movimentos.

```
    a    b    c    d    e
  -------------------------
1 | R2 | R2 | R2 | R2 | R2 |   Max movement per stack:
  -------------------------
2 |    |    |    |    |    |
  -------------------------
3 |    |    |    |    |    |   Stack of 4: cannot move
  -------------------------
4 |    |    |    |    |    |   Stack of 3: one space
  -------------------------
5 |    |    |    |    |    |   Stack of 2: two spaces
  -------------------------
6 |    |    |    |    |    |   Stack of 1: three spaces
  -------------------------
7 | B2 | B2 | B2 | B2 | B2 |
  -------------------------
CURRENT PLAYER:black
Choose one of the turn possibilities:
1 - Move a single stack.
2 - Move two individual pieces up to 2 spaces each.
|:
```

O nosso choose_move trata desta situação:

```
choose_move(GameState, Player, Move, TwoMovesGamestate) :-
    repeat,
    write('Choose one of the turn possibilities:') ,nl,
    write('1 - Move a single stack.') ,nl,
    write('2 - Move two individual pieces up to 2 spaces each.'),nl,
    read(Input),
    (integer(Input) ->
        (Input =:= 1 -> single_move(GameState, Player, Move, TwoMovesGamestate)
        ; Input =:= 2 -> double_move(GameState, Player, Move, TwoMovesGamestate)
        ;
          fail
        )
    ;
      fail
    )
    .
```

Para o caso de o jogador querer utilizar a primeira opção:

```
single_move(GameState,Player, Move, TwoMovesGamestate) :-
    repeat,
    write('Select a piece or stack (e.g., a1): '),
    read(FromInput),
    %i
    (integer(FromInput) -> write('Invalid input format. Please use the format "a1" or similar.'), nl, fail ; true),

    (user_input_to_coordinates(FromInput, (FromRow, FromCol)) -> true ; write('Invalid input format. Please use the format "a1" or similar.'), nl,
        fail),

    nth1(FromRow, GameState, Row),
    nth1(FromCol, Row, Piece),
    %ii
    ( (Player = black, Piece = black(_)) ;
      (Player = red, Piece = red(_)) ),

    piece_value(Piece, Val),

    %iii
    (((FromRow =\= 1, Player = black);
     (FromRow =\= 7, Player = red) -> true);
     (write('That is not allowed.'), nl, fail)          % NAO FAZ SENTIDO SEU FILHO DA PUTA
    ),

    write('Select a destination (e.g., b2): '),
    read(ToInput),      % Read the coordinate for the destination

    % ATAUQE COMBINADO GOES HERE

    %iv
    (integer(ToInput) -> write('Invalid input format. Please use the format "a1" or similar.'), nl, fail ; true),

    (user_input_to_coordinates(ToInput, (ToRow, ToCol)) -> true ; write('Invalid input format. Please use the format "a1" or similar.'), nl,
        fail),

,
%v

(((Val > 1) -> write('How many pieces do you want (e.g., 1, 2, 3, 4): '), read(NPiecesInput)); NPiecesInput = 1),

%vi
(\+integer(NPiecesInput) -> write('Invalid input format.'), nl, fail ; true),

%vii
(NPiecesInput =< Val),

%viii
calculate_possible(NPiecesInput, Possible),
(abs(FromRow - ToRow) =< Possible, abs(FromCol - ToCol) =< Possible),

nth1(ToRow, GameState, RowEnemy),
nth1(ToCol, RowEnemy, PieceEnemy),
piece_value(PieceEnemy, ValEnemy),


%ix
find_possible_paths(GameState, FromRow, FromCol, ToRow, ToCol, Possible, Paths, Player),
(Paths \= []),

%x
find_adjacent_pieces(GameState,Player,ToRow,ToCol,AdjacentPieces),
(((Player = black, PieceEnemy = red(_), AdjacentPieces \= [[Piece-(FromRow,FromCol)]], AdjacentPieces \= []) ;
(Player = red, PieceEnemy = black(_), AdjacentPieces \= [[Piece-(FromRow,FromCol)]], AdjacentPieces \= [])) ->
is_possible_combinateds(GameState, Player,NPiecesInput, NewVal,AdjacentPieces, FromRow, FromCol, ToRow, ToCol, Paths, GameStateCombinated)
 ;
true ),


 (
    (integer(NewVal), Yoo is NewVal, TwoMovesGamestate = GameStateCombinated);
    (Yoo is NPiecesInput, TwoMovesGamestate = GameState)
 ),

 %xi
 ( (Player = black, PieceEnemy = black(_), (Yoo+ValEnemy) =< 4,
    NewValue is Yoo + ValEnemy,
    NewValue2 is Val-NPiecesInput,
    PieceTo = black(NewValue),
    (NPiecesInput - Val =:= 0 -> PieceFrom = empty ; PieceFrom = black(NewValue2))
    ) ;
   (Player = red, PieceEnemy = red(_), (Yoo+ValEnemy) =< 4,
    NewValue is Yoo + ValEnemy,
    NewValue2 is Val-NPiecesInput,
    PieceTo = red(NewValue),
    (NPiecesInput - Val =:= 0 -> PieceFrom = empty ; PieceFrom = red(NewValue2))
    ) ;
   (Player = red, (PieceEnemy = black(_); PieceEnemy = empty), (Yoo > ValEnemy),
     NewValue is Yoo,
     NewValue2 is Val-NPiecesInput,
     PieceTo = red(Yoo),
     ((NPiecesInput =:= Val -> PieceFrom = empty; PieceFrom = red(NewValue2)))
     ) ;
   (Player = black, (PieceEnemy = red(_); PieceEnemy = empty), (Yoo > ValEnemy),
    NewValue is Yoo,
    NewValue2 is Val-NPiecesInput,
    PieceTo = black(Yoo),
    (NPiecesInput - Val =:= 0 -> PieceFrom = empty ; PieceFrom = black(NewValue2))
    )
 ),

 From = (FromRow, FromCol),
 To = (ToRow, ToCol),
 Move = (From, To, PieceFrom, PieceTo)
 .
```

Algumas das regras implementadas (identificadas no código):

i. Validação input
ii. Jogador só pode mover uma peça da sua cor
iii. Peças que já chegaram ao espaço de uma casa inimiga não podem ser movidas.
iv. Validação input
v. Caso a stack que queira mexer tenha um tamanho superior a 1, pode escolher quantas peças dessa stack quer mexer, caso contrário, mexe apenas 1.
vi. Validação input
vii. O número de peças que quer mexer deve ser igual ou inferior ao tamanho da stack que escolheu.
viii. O destino que escolheu deve ser alcançável com o número de movimentos máximos possíveis com a quantidade de peças que o utilizador decidiu mexer.
ix. Deve haver pelo menos 1 caminho possível para a peça chegar ao lugar que o utilizador escolheu com o número máximo de movimentos que é permitido. No cálculo destes caminhos são tidas em conta todas as restrições incluindo o facto de, num movimento diagonal, a peça não pode passar por entre duas peças inimigas. Todos os caminhos são guardados numa lista sendo que cada caminho é guardado numa lista de posições (x,y).
x. (Ataque Combinado) Para um ataque, aqui é verificado se, para cada path possível, na penúltima posição da peça antes da posição da peça inimiga, existem peças adjacentes tanto à peça atacante como à peça inimiga. Se sim, é perguntado ao utilizador se quer realizar um ataque combinado. Novamente, se o utilizador decidir usar um ataque combinado, será apresentada de uma maneira user-friendly as diferentes peças com que ele pode combinar e agrupar, sendo que o utilizador pode escolher combinar com apenas uma ou então mais peças, tudo desde que estas estejam todas adjacentes à peça atacante e à peça inimiga. Tudo isto é tido em consideração e é apresentado ao utilizador as diferentes opções possíveis e as conseguintes caso ele queira combinar mais do que uma peça.
xi. Recálculo da peça que estava na posição inicial e da peça que vai estar na posição final.
xii. Outro aspeto importante é que peças inimigas nos espaços de casa do utilizador não podem ser comidas.

Para o double_move/4, temos algo semelhante, mas o utilizador só pode mover peças individuais e dois espaços cadas.

Move/3:-

```
move(GameState, Move, NewGameState) :-
    % Split the move into separate components
    Move = (From, To, PieceFrom, PieceTo),

    From = (FromRow, FromCol),
    To = (ToRow, ToCol),

    nth1(ToRow, GameState, ToRowList),
    nth1(ToCol, ToRowList, EnemyPiece),

    piece_value(PieceTo, PieceToValue),
    piece_value(EnemyPiece, EnemyPieceValue),

    (PieceTo = red(_) -> Player = red ; Player = black),

    NewValue is (EnemyPieceValue - (PieceToValue - EnemyPieceValue)),

    retreat_positions(Player, ToRow, ToCol, RetreatPositions, GameState, NewValue),
    remove_empty_lists(RetreatPositions, RetreatPositionsFixed), nl,
    (
        ((PieceTo = red(_),
         EnemyPiece = black(_), NewValue > 0) ->
         retreat_positions(Player, ToRow, ToCol, RetreatPositions, GameState, NewValue),
         remove_empty_lists(RetreatPositions, RetreatPositionsFixed), nl,
         repeat,
         (
            (RetreatPositionsFixed \= [] -> (write('Choose a position to retreat the piece to: (  '),
        write_retreat(RetreatPositionsFixed),
        write(')'), nl,
        read(RetreatInput),
        (integer(RetreatInput) -> write('Invalid input format. Please use the format "a1" or similar.'), nl, fail ; true),
        (user_input_to_coordinates(RetreatInput, (RetreatRow, RetreatCol)) -> true ; write('Invalid input format. Please use the format "a1" or similar.'), nl,
        fail),
         member([RetreatRow, RetreatCol], RetreatPositionsFixed),
        nth1(RetreatRow, GameState, RetreatRowList),
        nth1(RetreatCol, RetreatRowList, RetreatSpace),
        piece_value(RetreatSpace, RetreatSpaceVal),
        (RetreatSpaceVal + (EnemyPieceValue - (PieceToValue - EnemyPieceValue))) < 4,
        ((RetreatSpace \= empty -> replace(GameState, RetreatRow, RetreatCol, black(EnemyPieceValue - (PieceToValue - EnemyPieceValue) + RetreatSpaceVal), GameState0)
        ;
    replace(GameState, RetreatRow, RetreatCol, black(EnemyPieceValue - (PieceToValue - EnemyPieceValue)), GameState0))
    );GameState0 = GameState)) ; GameState0 = GameState
     )
     );
    ((PieceTo = black(_),
     EnemyPiece = red(_), NewValue > 0) ->
     retreat_positions(Player, ToRow, ToCol, RetreatPositions, GameState, NewValue),
     remove_empty_lists(RetreatPositions, RetreatPositionsFixed), nl,
     repeat,
    (
        (RetreatPositionsFixed \= [] -> (write('Choose a position to retreat the piece to: (  '),
     write_retreat(RetreatPositionsFixed),
     write(')'), nl,
     read(RetreatInput),
     (integer(RetreatInput) -> write('Invalid input format. Please use the format "a1" or similar.'), nl, fail ; true),
    (user_input_to_coordinates(RetreatInput, (RetreatRow, RetreatCol)) -> true ; write('Invalid input format. Please use the format "a1" or similar.'), nl,
    fail),
     member([RetreatRow, RetreatCol], RetreatPositionsFixed),
     nth1(RetreatRow, GameState, RetreatRowList),
     nth1(RetreatCol, RetreatRowList, RetreatSpace),
     piece_value(RetreatSpace, RetreatSpaceVal),
     (RetreatSpaceVal + (EnemyPieceValue - (PieceToValue - EnemyPieceValue))) < 4,
     ((RetreatSpace \= empty -> replace(GameState, RetreatRow, RetreatCol, red(EnemyPieceValue - (PieceToValue - EnemyPieceValue) + RetreatSpaceVal), GameState0)
     ;
     replace(GameState, RetreatRow, RetreatCol, red(EnemyPieceValue - (PieceToValue - EnemyPieceValue)), GameState0))
     ); GameState0 = GameState)) ; GameState0 = GameState
     )
     );
     GameState0 = GameState
 ),


 % Create the new board with the piece moved
 replace(GameState0, FromRow, FromCol, PieceFrom, TempGameState),
 replace(TempGameState, ToRow, ToCol, PieceTo, NewGameState).
```

Já no move/3 em si, recebemos no Move a posição da qual a peça parte e a peça que temos que lá colocar bem como a posição destinatária e a peça que temos que lá calcular, tudo calculado previamente. Outra situação que verificamos é quando a diferença entre a peça que atacou e a peça atacada é menor do que o valor da peça atacada, esta não desaparece. Por exemplo, uma stack de 3 ataca uma stack de 2. Como a diferença entre ataques é apenas 1, a stack atacada deve perder apenas uma peça, e as

restantes peças (neste caso apenas 1) devem recuar uma casa à escolha do jogador atacante. Quando isto acontece, nós oferecemos ao utilizador as opções possíveis de recuo de uma maneira user-friendly, sendo que a peça não poderá recuar para uma casa com uma peça inimiga nem para um espaço com uma peça da mesma cor se a junção das duas resultasse numa stack com um valor maior do que 4. Quando a peça comida está numa homespace, ao invés de recuar, vai para uma das homespaces ao lado da qual se encontra. Apesar de ser raro acontecer esta situação, é possível a peça a recuar não ter nenhuma posição válida para recuar, pois pode se verificar, por exemplo, todas os 3 espaços de recuo estarem ocupados por uma peça inimiga. Apesar de este caso específico não ser tratado nas regras oficiais do jogo, nós decidimos que quando isto acontece, as restantes peças também são eliminadas. No final do move trocamos o GameState pelo novo GameState.

Lista de jogadas válidas

Este predicado foi usado unicamente para a movimentação dos bots. O validate_move segue as mesmas regras descritas acima para o choose_move com a exceção de não haver inputs do utilizador. Para haver jogos distintos no caso dos bots nível 1, usamos o tempo atual do computador para a definição da seed que diz qual é o random_move a ser escolhido, assim, quando corremos os jogos obtemos sempre um resultado diferente e de uma maneira diferente.

```
valid_moves(GameState, Moves, Player):-
    now(X),
    setrand(X),
    findall(Move,  (between(1, 5, FromCol), between(1, 7, FromRow),
            between(1, 5, ToCol), between(1, 7, ToRow), between(1, 4, NPiecesMoving),
            validate_move(GameState,Player, FromRow-FromCol, ToRow-ToCol,NPiecesMoving, Move)
            ), Moves),
            \+length(Moves, 0), !.
```

Final do Jogo

A cada ciclo de jogo são calculadas o número de peças vermelhas e pretas. Se o número de peças pretas for inferior a 4, o jogador da cor Vermelho ganha e o jogo acaba. Se o número de peças vermelhas for inferior a 4, o jogador da cor Preta ganha e o jogo acaba. Outra maneira de ganhar é através da colocação de peças nos espaços de casa do inimigo. Esse cálculo também é feito: se 4 ou mais peças pretas estiverem na fila 1, o jogo termina e o jogador da cor Preta ganha e, do mesmo modo, se 4 ou mais peças vermelhas estiverem na fila 7, o jogo termina e o jogador da cor Vermelha ganha.

```
game_over(GameState, Winner) :-
    sum_red_pieces(GameState, SumRedTotal), !,
    sum_black_pieces(GameState, SumBlackTotal), !,
    sum_red_pieces_on_row7(GameState, SumRed7), !,
    sum_black_pieces_on_row1(GameState, SumBlack1), !,

    (   (SumRedTotal < 5, Winner = 'Black') ->
        write('Black Wins!'), nl
    ;   (SumBlackTotal < 5, Winner = 'Red') ->
        write('Red Wins!'), nl
    ;   (SumRed7 > 3, Winner = 'Red') ->
        write('Red Wins!'), nl
    ;   (SumBlack1 > 3, Winner = 'Black') ->
        write('Black Wins!'), nl
    ).
```

Avaliação do tabuleiro

Esta função é importante para os bots nível 2 onde é usado um algoritmo greedy para calcular qual o melhor move de todos os moves disponíveis para o bot realizar. O move mais valorizado é aquele que permite chegar a uma vitória, seja por chegar com mais de 4 peças à casa inimiga ou por eliminar 6 ou mais peças inimigas. Caso um move vitorioso não exista, é valorizado atacar peças inimigas para as eliminar. Caso não sejam possíveis ataques, é valorizado aproximar peças das casas inimigas. Para isso, existe, para cada cor, um valor associado a cada fila do tabuleiro. Sendo que quanto menor o valor final, melhor é a jogada, quanto mais próxima a fila das casas inimigas, menor valor esta tem.
O valor é calculado multiplicando o número de peças da fila * o valor da fila. Quanto a moves de ataque, por cada peça comida é subtraído 50 ao valor do move. Quando é um move vitorioso, este é colocado imediatamente o melhor valor.

Jogadas dos computadores

Para o bot nível 1, este escolhe um move à sorte de todos os valid_moves que tem disponíveis. Algo que também fizemos foi mudar constantemente a seed com a hora do computador em que está a correr para que fosse possível observar o decorrer de diferentes jogos.

```
now(X),
setrand(X),

choose_move_bot(GameState, Player, Level, Move):-
    valid_moves(GameState, Moves, Player),
    choose_move_bot(Level, Player, GameState, Moves, Move).

choose_move_bot(1,_Player, _GameState, Moves, Move):-
    random_select(Move, Moves, _Rest).
```

Para o bot nível 2, através do nosso algoritmo greedy e da avaliação do tabuleiro descrita acima, o bot irá sempre escolher a melhor jogada de acordo com o nosso algoritmo.

```
choose_move_bot(2,Player, GameState, Moves, Move):-
    setof(Value-Mv, NewState^( member(Mv, Moves),
    move_bot(GameState, Mv, NewState),
    evaluate_board(GameState, NewState, Player, Value)), [_V-Move|_]).
```

Conclusões

O jogo Stacks foi realizado com sucesso. Foram consideradas todos os possíveis erros e casos especiais e foram tratados de acordo com as regras oficiais. A maior dificuldade deste projeto foi, sem dúvida, a análise de todas as regras e perceber bem o que o autor do jogo pretendia com as mesmas. A grande quantidade de regras e de acontecimentos possíveis numa única jogada levou a alguns casos especiais em que foi necessária atenção extra e levou também a alguma dificuldade em traduzir o jogo para o utilizador de uma maneira percetível para o mesmo enquanto se usava uma quantidade de inputs reduzida para não tornar o jogo maçador. É realmente um jogo que se demora a aprender e a masterizar devido à enorme quantidade de jogadas possíveis que um utilizador pode fazer num dado momento. O facto de haver algum espaço para interpretação das regras também pode levar a alguma ambiguidade, mas temos a perceção de que foram implementadas da melhor maneira.

Bibliografia

Descrição do jogo: Stacks | Board Game | BoardGameGeek

Regras do jogo: Stacks | Stacks: Rules of Play | File | BoardGameGeek