U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Programação Funcional e em Lógica

Módulo 2 – Programação em Lógica

Realizado por:

Frederico Manuel Alves Pereira Oliveira Lopes (up201904580)

Pedro Jorge Ribeiro Botelho de Moniz Pacheco (up201806824)

Grupo:

Petteia (Simple)_5

Instalação e execução

Linux:

- Compilar ficheiro menu ([menu]);
- Correr função play (play.).

Windows:

- Compilar ficheiro menu (consult(*path_to_menu*))
- Correr função play (play.).

Descrição do jogo

O jogo de tabuleiro Petteia é um dos jogos mais antigos do mundo, tendo surgido na Grécia Antiga. Apresenta várias variações, sendo a presente no nosso trabalho a versão Petteia Simples.

Esta variação é jogada num tabuleiro 8x12, estando as peças dos jogadores a ocupar a primeira e a última fila, e tem as seguintes regras:

- É jogado por dois jogadores adversários, que jogam à vez;
- Duas peças não podem ocupar o mesmo espaço, simultaneamente;
- O jogador só pode mover uma peça por jogada;
- O jogador pode mover cada peça o número de casas desejado, na horizontal OU vertical (diagonal não);
- Cada peça não pode passar por cima de outra peça durante o movimento;
- As peças são capturadas quando ficam entre duas peças do adversário, horizontal OU verticalmente (diagonal não);
- Peças capturadas são removidas do tabuleiro;
- O jogo termina quando o número estipulado de capturas tiver sido atingido por um dos jogadores.

As regras mais pormenorizadas do jogo podem ser consultadas aqui: https://www.nestorgames.com/rulebooks/PETTEIA_EN.pdf

Lógica do jogo

Representação interna do estado do jogo:

O estado atual é representado por uma lista de listas, em que cada uma destas linhas corresponde a uma linha do tabuleiro.

Estado inicial:

```
starting_board([
    [b,b,b,b,b,b,b,b,b,b,b],
    [clear,clear,clear,clear,clear,clear,clear,clear,clear,clear,clear],
    [clear,clear,clear,clear,clear,clear,clear,clear,clear,clear,clear],
    [clear,clear,clear,clear,clear,clear,clear,clear,clear,clear,clear],
    [clear,clear,clear,clear,clear,clear,clear,clear,clear,clear,clear],
    [clear,clear,clear,clear,clear,clear,clear,clear,clear,clear,clear],
    [clear,clear,clear,clear,clear,clear,clear,clear,clear,clear,clear],
    [w,w,w,w,w,w,w,w,w,w,w]
    ]).
```

Exemplo de Estado Intermédio:

```
([
    [b,clear,b,b,b,b,b,clear,b,b,clear,b],
    [clear,clear,clear,clear,clear,clear,clear,clear,clear,clear,clear,clear],
    [clear,b,clear,clear,clear,b,clear,clear,clear,clear,b,clear],
    [clear,clear,clear,clear,clear,clear,w,clear,clear,clear,clear,clear],
    [clear,clear,clear,clear,clear,clear,clear,clear,clear,clear,w,clear],
    [clear,clear,clear,clear,w,clear,clear,clear,clear,clear,clear,clear],
    [clear,clear,clear,clear,clear,clear,clear,clear,clear,clear,clear,clear],
    [w,w,w,w,clear,w,clear,w,w,clear,w,w]
    ]).
```

Exemplo de Estado Final:

([

[b,clear,b,b,b,b,b,clear,b,b,clear,b],

[clear,clear,clear,clear,clear,clear,clear,clear,clear,clear,clear,clear],

[clear,clear,clear,clear,clear,b,clear,clear,clear,clear,clear,clear],

[clear,clear,clear,clear,clear,clear,clear,clear,clear,clear,clear,clear],

[clear,clear,clear,clear,clear,clear,clear,clear,clear,clear,w,clear],

[clear,clear,clear,clear,w,clear,w,clear,clear,clear,clear,clear],

[clear,clear,clear,clear,clear,clear,clear,clear,clear,clear,w,clear],

[w,w,w,w,clear,w,clear,w,w,clear,w,clear]

]).

Legenda:

w – peça branca (W)

b – peça preta (B)

clear – espaço em branco

Sempre que um jogador faz uma jogada, o tabuleiro instanciado no início do jogo é atualizado, sendo que as listas que o integram são substituídas por outras listas.

A gestão de quem deve jogar e de quantas peças foram capturadas por cada jogador, ao longo do jogo, é feito por variáveis no loop principal do jogo.

```
game_loop(Board, Player, WhiteCaptures, BlackCaptures, Level)
```

Player é o jogador da ronda atual, WhiteCaptures e BlackCaptures são as peças que o respetivo jogador capturou até ao momento.

Visualização do estado de jogo:

Visto que o estado atual do nosso jogo é representado pelo tabuleiro, no início de todas as jogadas é chamada a função write_board(Board).

Esta função faz um ciclo em que chama a função write_line(Line, Id) para cada iteração.

Na primeira iteração é passado o Id 0 que indica ao write_line que deve imprimir a linha dos índices. Nas seguintes iterações, o write_line imprime as listas do Board, uma a uma.

A função write_line tem um funcionamento semelhante, com a diferença de que na primeira iteração imprime o índice da coluna e nas seguintes iterações imprime a coluna propriamente dita.

A função write_board permite às listas terem tamanho variável, contudo, esta funcionalidade não foi utilizada devido a natureza do nosso jogo.

Execução de Jogadas:

A execução da jogada começa com o loop principal a determinar de quem é a vez de jogar. Depois disto, o estado atual do tabuleiro é mostrado ao utilizador para este poder escolher que jogada quer fazer.

A escolha do movimento dá-se de seguida, sendo que para a execução de uma jogada é necessário o jogador escolher uma peça (seleciona através da coluna e fila) e o local para onde a quer mover.

```
starting_pos(Board, ValidColumn, ValidRow, Player) :-
    write('Choose a piece to move\n'),
    write('Column\n'),
    checkInputColumn(IsValidC, Column),
    write('Row\n'),
    checkInputRow(IsValidR, Row),

    nth0(Row, Board, RowList),
    nth0(Column, RowList, Element),
    (Element == Player
        ->ValidRow = Row, ValidColumn = Column, !, true
        ;   write('Invalid,   that   is   not   your   piece!\n'),
starting_pos(Board, ValidColumn, ValidRow, Player)
    ).
```

São feitas validações para a peça que o jogador escolhe, de forma a garantir que a peça existe e que lhe pertence, e para onde o jogador a quer mover, com o objetivo de verificar se o movimento é válido.

Para verificação da validade de uma jogada, usamos a função checkLegalMove(+Board, +OriginColumn, +OriginRow, +DestinationColumn, +DestinationRow, -ReturnBooleanValue), onde o argumento Board será o tabuleiro atual e a forma como as peças estão dispostas no mesmo aquando da verificação da validade da jogada em questão. OriginColumn e OriginRow indicam a posição da peça a ser movida pela sua coluna e fila, respetivamente. DestinationColumn e DestinationRow indicam para que casa o jogador quer que a peça se mova, e ReturnBooleanValue retorna dois valores, sendo 'True' quando o movimento é legal, e 'False' quando é ilegal.

A verificação começa por ver se o movimento corresponde a um movimento vertical ou horizontal, e caso nenhum dos dois se verifique, retornará 'False'. No caso de um dos dois movimentos se verificar, o programa irá verificar se é um movimento para a frente da posição inicial (quando é um movimento vertical, numa posição superior à posição prévia, quando é um movimento horizontal, numa posição mais à direita à posição prévia).

De seguida, obtêm a fila ou coluna em questão, começando por verificar se alguma peça se encontra na posição de destino, e caso tal seja verdadeiro, a função retorna 'False'. Posteriormente, o programa verifica se existe alguma peça entre a posição inicial e a posição de destino, devolvendo 'False' se houver. Caso nenhuma das condições se verifique (peça no destino e peça entre origem e destino), a função retorna 'True'.

Depois disto, se a jogada for inválida, novas informações são pedidas ao jogador, e por outro lado, se a jogada for válida, o movimento é executado.

Para executar o movimento, a lista que continha a antiga posição é substituída por uma nova lista semelhante à anterior, com a diferença que tem aquela posição com clear". Depois disto, um processo semelhante é executado na posição final, mas desta vez é acrescentada uma peça. No caso de movimento horizontal, só uma lista tem de ser substituída.

Após o movimento ser feito, é necessário determinar se alguma captura ocorreu. Para isto foi criada a função check_captures(Board, NewBoard, WCapture, BCapture), que é chamada no final de cada jogada. Esta função é composta por várias partes, começando por verificar que existiu alguma captura feita horizontalmente, e para isso é chamada a função check_horizontal_captures(Board, TempBoard, 0, HWCapture, HBCapture) que, por sua vez, chama as funções white_capture e black_capture. A função white_capture analisa todo o tabuleiro à procura de peças pretas no meio de peças brancas horizontalmente. Se nenhuma for encontrada então é retornada uma cópia do board e, se por outro lado, ocorrer alguma captura, então a lista que contém a linha com a peça capturada é substituída por uma nova lista sem a peça capturada, WCapture

passa a ser 1 para indicar que ocorreu uma captura e o Board com a nova lista é retornado. A função black_board faz o mesmo para as peças pretas.

Depois de as capturas horizontais serem verificadas, é necessário fazer o mesmo para as verticais. A função check_vertical_captures tem um funcionamento bastante semelhante com a função horizontal, com a diferença que, desta vez a função itera através das colunas, em vez das filas e por isso foi também necessário criar novas funções para a captura propriamente dita (white_capture_vertical e black_capture_vertical) visto que a substituição da peça capturada no tabuleiro tem um funcionamento diferente.

Final do Jogo:

Devido à simplicidade do sistema de pontos do nosso jogo, consideramos que casos base para a função do loop principal do jogo eram suficientes para determinar se o jogo tinha terminado. Estes são atingidos quando um dos jogadores alcança o número de pontos estipulado e assim, termina a execução do jogo com uma mensagem de parabéns ao vencedor.

```
game_loop(Board,_,7,_,_) :-
    write_board(Board),
    write('White player won!').
```

Lista de Jogadas Válidas:

A implementação da função valid_moves(Board, ListOfValidMoves) começa por receber como argumento Board, o tabuleiro atual do jogo, ou seja, o estado atual do jogo. Ao receber este argumento, passa-o à função auxiliar iterBoardFinal(Board, StartingRow, StartingColumn, AuxList, FinalList), sendo StartingRow e StartingColumn a posição no tabuleiro onde irá começar a iterar, seguido por uma lista auxiliar vazia, e devolve o resultado em FinalList. A função itera as filas do tabuleiro de forma a encontrar peças, e quando as encontra, verifica as jogadas possíveis para cada uma, com ajuda da função checkValidMovesForPiece(Board, Piece, ListOfValidMoves), que por sua vez recebe também o Board, tal como a peça em questão e devolve os movimentos válidos dessa peça, numa lista de nome ListOfValidMoves. A função checkValidMovesForPiece usa uma série de quatro (4) funções para verificação de movimentos possíveis e legais, sendo elas: iterRowFront, iterRowBack, iterColumnFront, iterColumnBack, que iteram uma fila em posições superiores e inferiores no tabuleiro, e também iteram uma coluna tanto em posições superiores como inferiores no tabuleiro.

Depois de iterar todas as filas à procura de peças e verificar todos os movimentos possíveis para cada uma, o argumento ListOfValidMoves da função valid_moves, tomará o valor que for retornado da função iterBoardFinal em FinalList.

A lista ListOfValidMoves é constituída por sua vez, por sublistas na forma de [Origin, Destination], em que Origin corresponde à Row (fila) e Column (coluna) inicial, onde se encontra a peça, e Destination corresponde à Row (fila) e Column (coluna) final para onde a peça se pode deslocar, constituindo esse movimento a um movimento legal.

Jogada do Computador:

Para cada jogada do computador, é criada uma lista com as respetivas peças e as suas posições e outra lista com as posições que estão vazias no tabuleiro. As funções que determinam as posições destas peças são:

get_pieces(Board, Row, 12, List, FinalList, Color)

empty_places(Board, Row, 12, List, FinalList)

Estas funções iteram pelas várias linhas do Board e pelos vários elementos de cada linha. Sempre que o elemento é igual ao pretendido, é guardada a sua posição e no fim todas as posições são retornadas na FinalList.

Antes da jogada ser executada, é feita a seleção aleatória de uma peça e de um espaço vazio no tabuleiro, se a jogada for válida é executada, se não o for, então uma nova peça e uma nova posição final são escolhidas.

```
ai_play(Board, NewBoard, Color) :-
    get_pieces(Board, 0, 0, List, BlackList, Color),
    empty_places(Board, 0, 0, List1, ClearList),
    %get initial position
    random_member(Initial, BlackList),
    nth0(0, Initial, Row),
    nth0(1, Initial, Column),
    %get final position
    random_member(Final, ClearList),
    nth0(0, Final, FinalRow),
    nth0(1, Final, FinalColumn),
    %check if move is valid and move if so
```

```
    checkLegalMove(Board, Column, Row, FinalColumn, FinalRow,
ReturnBooleanValue),
    (  ReturnBooleanValue == 'True'
    -> move(Board, Column, Row, FinalColumn, FinalRow, NewBoard)
    ;  !, ai_play(Board, NewBoard, Color)
    ).
```

Conclusões

Este projeto, apesar de bastante grande e complexo para uma nova linguagem, foi bastante útil para perceber como programas feitos em outros paradigmas de programação podem ser igualmente conseguidos com programação em lógica. Foi também possível perceber algumas vantagens do paradigma de programação em lógica face aos outros, principalmente no que toca a algoritmos, que necessitam de instruções lógicas complexas.

Uma das dificuldades encontradas foi a adaptação do conteúdo do enunciado ao nosso tema, sendo que em alguns casos foi difícil implementar o conteúdo da maneira desejada, dado a lógica do jogo e o que foi pedido no enunciado serem incompatíveis, dificuldade que por vezes foi ultrapassada e por outras não.

Para além disso, não foi possível integrar no projeto todas as características que queríamos, como foi o caso do nível 2 da AI, que por falta de tempo foi impossível de conseguir.

Ainda assim, consideramos que o objetivo do projeto foi conseguido e o resultado final foi satisfatório.