Faculdade de Engenharia da Universidade do Porto

Relatório Projeto PLOG: SHÕBU

Programação em Lógica 2019/2020

PLOG_TP4_RI_SHÕBU1:

Alexandre Carqueja: up201705049@fe.up.pt Henrique Santos: up201706898@fe.up.pt

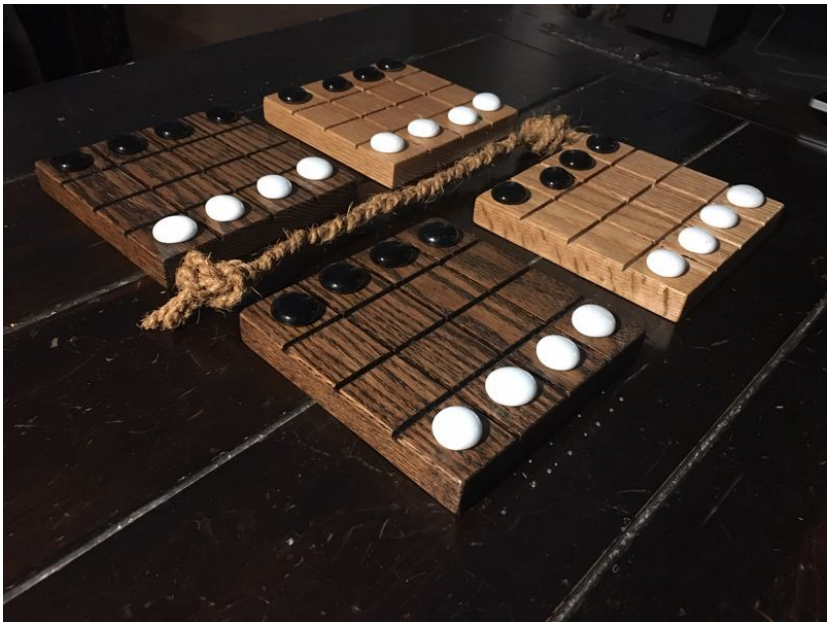
Índice

1. Introdução

Este trabalho consiste na implementação do jogo de tabuleiro “Shõbu” seguindo o paradigma de programação funcional, utilizando a linguagem Prolog. Foi desenvolvido em SWI-Prolog, e implementa as regras de movimentação e vitória do jogo, permitindo jogos entre humano-humano, humano-computador e computador-computador.

2. Descrição do Jogo

Shõbu é um jogo de 2 jogadores constituído por 4 “mini tabuleiros” de 2 cores (preto e branco), estes tabuleiros estão separados por uma corda no meio, sendo que cada jogador fica com 1 tabuleiro de cada cor do seu lado da corda.



Em cada um dos tabuleiros estão 4 pedras de cada jogador (4 brancas e 4 pretas). O jogador com as pedras pretas começa, e em cada turno são feitas 2 ações:

- Num dos tabuleiros do seu lado da corda, mexe uma das suas pedras em qualquer direção (mesmo diagonal) em, no máximo, 2 casas, sendo que não pode passar por nenhuma outra pedra no seu caminho.
- A seguir repete a mesma ação em qualquer outra pedra sua num dos 2 tabuleiros da outra cor. Desta vez a pedra pode passar por outras pedras no seu caminho, empurrando-as na direção em que se move (exceto se houver outra pedra atrás a impedir que a pedra seja empurrada). Se alguma pedra cair fora do tabuleiro é eliminada.

O objetivo é conseguir eliminar todas as pedras do oponente num dos 4 tabuleiros.

Informação: https://boardgamegeek.com/boardgame/272380/shbu

Vídeo de tutorial: https://www.youtube.com/watch?v=PN5899RUzVk

3. Lógica do Jogo

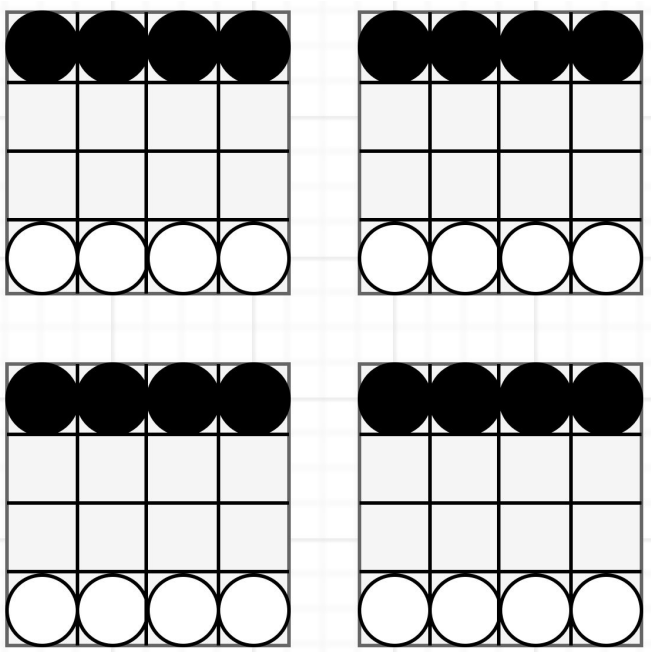
3.1. Representação do Estado de Jogo:

Para a representação dos tabuleiro usamos uma matriz de matrizes, em que a matriz principal é de 2 por 2 e em que cada matriz dentro dessa matriz corresponde a um tabuleiro de X por X.

Desde a entrega intermédia, foi adicionada a possibilidade de boards com tamanho variável, criados pelo predicado create_board(+Size, -Board). O código correspondente aos boards e funções de criação dos mesmos encontra-se no ficheiro “boards.pl”.

Estado inicial:

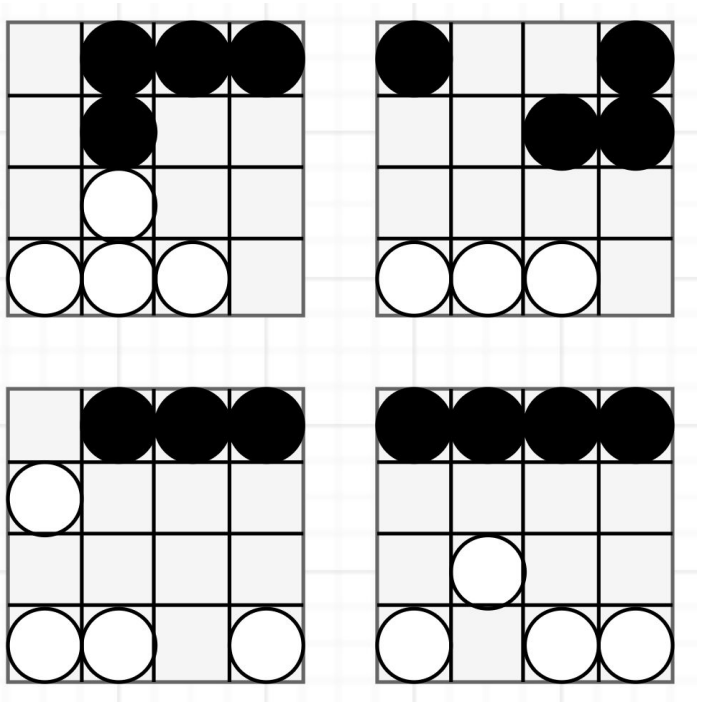
```
initialBoard([
                [
                    [
                        [b, b, b, b],
                        [e, e, e, e],
                        [e, e, e, e],
                        [w, w, w, w]
                    ],
                    [
                        [b, b, b, b],
                        [e, e, e, e],
                        [e, e, e, e],
                        [w, w, w, w]
                    ]
                ],
                [
                    [
                        [b, b, b, b],
                        [e, e, e, e],
                        [e, e, e, e],
                        [w, w, w, w]
                    ],
                    [
                        [b, b, b, b],
                        [e, e, e, e],
                        [e, e, e, e],
                        [w, w, w, w]
                    ]
                ]
            ]).
```



b: black piece; w: white piece; e: empty

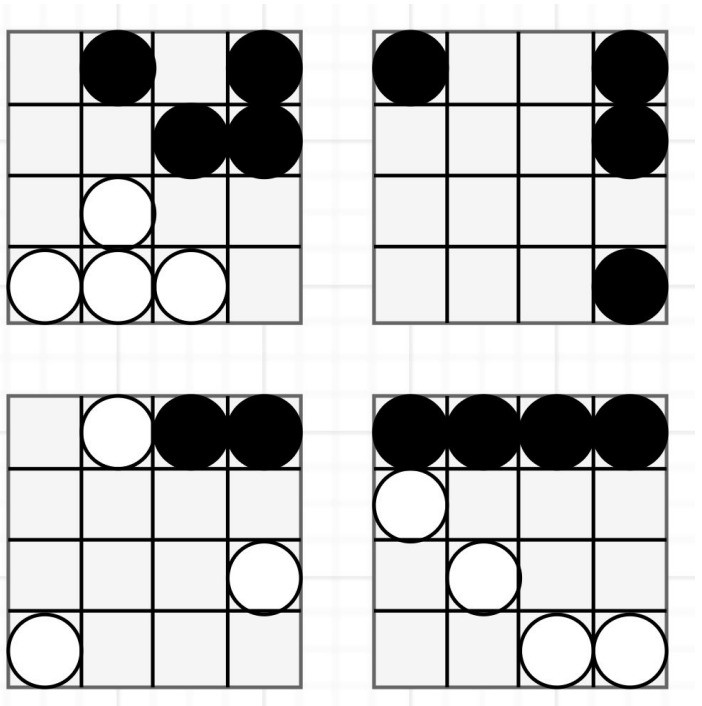
Estado intermédio:

```
intermediateBoard([
                [
                    [
                        [e, b, b, b],
                        [e, b, e, e],
                        [e, w, e, e],
                        [w, w, w, e]
                    ],
                    [
                        [b, e, e, b],
                        [e, e, b, b],
                        [e, e, e, e],
                        [w, w, w, e]
                    ]
                ],
                [
                    [
                        [e, b, b, b],
                        [w, e, e, e],
                        [e, e, e, e],
                        [w, w, e, w]
                    ],
                    [
                        [b, b, b, b],
                        [e, e, e, e],
                        [e, w, e, e],
                        [w, e, w, w]
                    ]
                ]
            ]).
```



Estado Final:

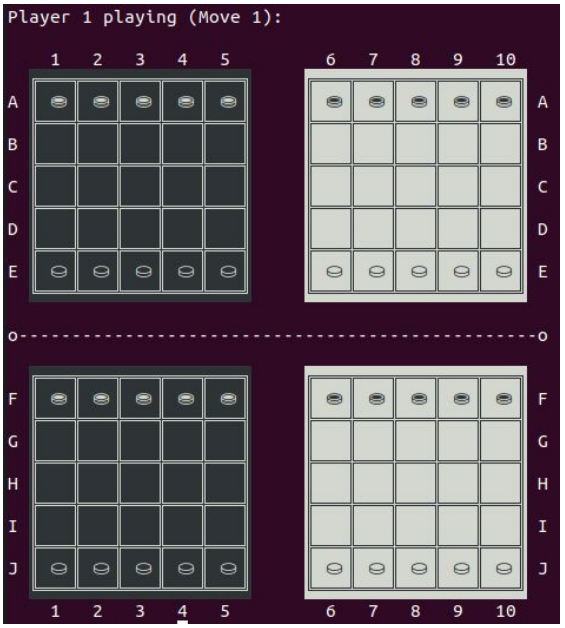
```
endBoard([
                [
                    [
                        [e, b, e, b],
                        [e, e, b, b],
                        [e, w, e, e],
                        [w, w, w, e]
                    ],
                    [
                        [b, e, e, b],
                        [e, e, e, b],
                        [e, e, e, e],
                        [e, e, e, b]
                    ]
                ],
                [
                    [
                        [e, w, b, b],
                        [e, e, e, e],
                        [e, e, e, w],
                        [w, e, e, e]
                    ],
                    [
                        [b, b, b, b],
                        [w, e, e, e],
                        [e, w, e, e],
                        [e, e, w, w]
                    ]
                ]
            ]).
```



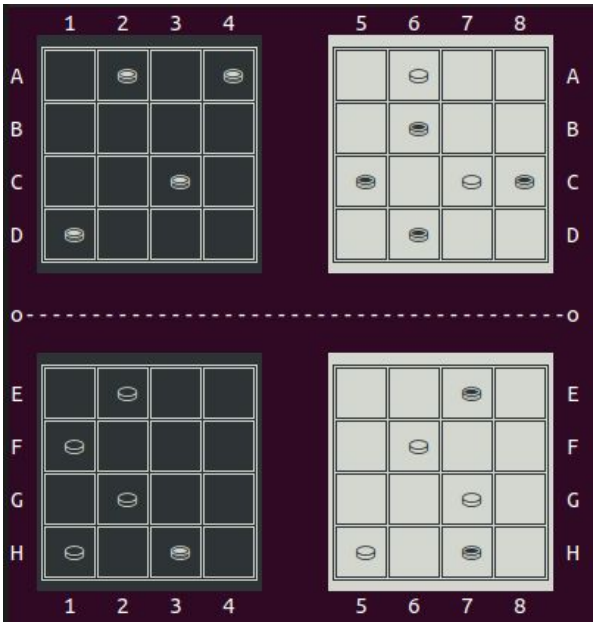
3.2. Visualização do tabuleiro

O jogo é displayed com recurso ao predicado **display_game(+Board, +Player, +Move)**, onde +Board é a matriz representativa do estado do board, +Player é o número do jogador(1/2), e +Move é a etapa de jogada (1/2). Esta chama duas vezes o predicado **display_board_pair(+BoardPair, +NRow, +NBoard)**, que por sua vez chama os predicado **print_row([Cell|Rest])** e **print_divisor(N)** recursivamente.

O código correspondente à visualização do tabuleiro encontra-se no ficheiro “display.pl”.



Tabuleiro 5x5 inicial



Tabuleiro 4x4 final

3.3. Lista de jogadas válidas

A validação de uma jogada foi separada em dois predicados com o mesmo nome e aridade diferentes: **valid_moves(+Board, +Player, -ListOfMoves)** e **valid_moves(+Board, +Player, +LastMove, -ListOfMoves)**.

O primeiro predicado, de aridade 3, valida jogadas na primeira fase de jogo, ou seja, a primeira jogada que o jogador realiza sobre um dos seus dois homeboards. Como tal, este retorna em ListOfMoves todos os movimentos de 1/2 espaços possíveis que não colidem com outras peças e permitem uma segunda jogada num tabuleiro da cor oposta, sendo os seus elementos da forma **[Yi/Xi,Yf/Xf]**.

O segundo predicado, de aridade 4, recebe como argumento extra um "LastMove", que especifica o movimento tomado na primeira fase de jogo, sob a qual a segunda fase é dependente. Este irá retornar todos os movimentos iguais a LastMove, e desta vez pode empurrar peças inimigas (desde que não haja outras a bloquear o caminho). A ListOfMoves, em vez de retornar uma lista com elementos do tipo [Yi/Xi,Yf/Xf], retorna uma lista com elementos do tipo **[[Yi/Xi, Yf/Xf], PiecePushed]**, em que PiecePushed é também um elemento do tipo **[Yi/Xi,Yf/Xf]** ou apenas **[Yi/Xi]**, dependendo de se é arrastado para outra posição ou se é arrastado para fora do board, respetivamente.

Em ambos os predicados é utilizado o findall para nos preencher a lista de movimentos possíveis com todos os movimentos que são aceites para aquela jogada.

No caso do primeiro movimento, é gerada uma peça que esteja dentro do tabuleiro com recurso a **inside_board(+TotalSize, -Yi/Xi)**, é depois verificado se essa posição corresponde a uma peça do jogador, e de seguida é gerada uma posição válida para essa peça com o predicado **check_move1_destination(+Board, +Yi/Xi, -Yf/Xf)**, este gera posições que estejam a 1 ou 2 casas de distância desde que não haja nenhuma outra pela entre a posição inicial e a final. Como estamos a utilizar findall para esta parte, o resultado final é que todas as posições aceites são adicionadas à lista. Finalmente verifica-se se esse trajeto não cria um cenário impossível na jogada a seguir, testando se a Lista de jogadas possíveis num board em que esta jogada teria sido executada não é vazia.

No caso do segundo movimento, da mesma maneira que no primeiro, é gerada uma peça do jogador com **inside_board(+TotalSize, -Yi/Xi)** e de seguida passa-se a gerar as posições finais possíveis com **check_move2_destination(+Board, +LastMove, +Yi/Xi, -Yf/Xf, +PieceType, -PiecePushed)**. Neste predicado o trajeto de Yi/Xi Yf/Xf tem de ser com a mesma direção que o do movimento anterior (LastMove), se essa condição se confirmar passa-se a validar o trajeto com o predicado **check_if_path_is_valid(+Yi/Xi,**

+Yf/Xf, +SmallBoard, +PieceType, +Bx, +By, -PiecePushed), que confirma se entre as 2 posições se passa por alguma peça inimiga no caminho, e no caso de se passar verifica se é possível empurrá-la.

3.4. Execução de jogadas

A cada jogada, é pedido um input ao jogador, sob a forma de linha/coluna (por exemplo, o input E11 para a linha 4 coluna 10). Após ser pedido tanto a origem como o destino do movimento, este é comparado com a lista de jogadas válidas obtida através da chamada dos predicados valid_move. Caso a jogada esteja presente na lista de movimentos válidos, é chamado o predicado **move(+Move, +Board, -OutBoard)**, que recebe a jogada e o board atual, e retorna um novo board com a jogada aplicada. Este predicado, como já recebe uma jogada validada anteriormente, apenas tem de realizar a mudança dos tiles, com recurso a duas chamadas ao predicado **set_tile(+InBoard, -OutBoard, +Line, +Col, +Symbol, -PastSymbol)**. Este irá alterar o tile com as coordenadas especificadas por (Line, Col) no board de input, retornando em OutBoard o novo board e em PastSymbol o tile que se encontrava anteriormente naquela posição.

3.5. Final do jogo

O final do jogo é calculado de forma simples, devido à simplicidade deste critério nas próprias regras do jogo: caso haja algum small board em que não existam mais peças de um jogador, esse jogador perde. Como tal, o predicado **game_over(+Board, -Winner)** apenas chama o predicado **exists_both_piece_types(+SmallBoard)** nos quatro boards pequenos, que por sua vez chama os predicados **has_black_piece(+SmallBoard)** e **has_white_piece(+SmallBoard)** nestes. Caso algum dos boards não verifique o predicado exists_both_piece_types, são feitas verificações adicionais com os predicados has_black_piece e has_white_piece para determinar o vencedor do jogo.

3.6. Avaliação do tabuleiro

A avaliação do tabuleiro é feito de maneira diferente para as duas partes da jogada, na primeira parte o tabuleiro é avaliado de acordo com o número de opções de eliminar peças adversárias que se tem na parte seguinte se se realizar este movimento, versus o número de opções que o adversário tem para fazer o mesmo. Nesta parte é utilizado o predicado **value(+Board, +Player, +Move, -Value)**, em que o Player tem de estar na primeira parte do seu turno. A contabilização de peças que se pode comer é feita no predicado **countEatMoves(+ListOfMoves, Count**).

na segunda parte da jogada o tabuleiro é avaliado de acordo com a jogada que permite criar um board mais desequilibrado a nosso favor, ou seja, com recurso ao predicado **valueSmallBoard(+Board, +Player, -Value)**, são avaliados cada um dos dos pequenos Boards individualmente, e dá-se uma pontuação maior por cada board que esteja com uma diferença entre peças nossas versus peças do adversário maior.

Assim garante-se que em cada jogada completa se tem sempre o número ótimo de opções para eliminar peças em relação ao nosso adversário, e que estamos sempre a enfraquecer o adversário nos locais que são mais vantajosos para nós. No entanto é um algoritmo mais focado no ataque e menos da defesa.

3.7. Jogada do Computador

A escolha de uma jogada pela parte do computador é feita através dos predicados **choose_move(+Board, +Level, +Player, -Move)** e **choose_move(+Board, +Level, +Player, +LastMove, -Move).**

Os dois predicados choose_move têm aridades diferentes pois, tal como na obtenção de jogadas possíveis, é assim que diferenciamos a fase de jogada.

Foram implementados dois níveis de dificuldade, 0 (random) e 1 (greedy). Na dificuldade 0, a escolha é feita ao obter todas as jogadas válidas e escolhendo uma ao acaso.

Na dificuldade 1, obtemos todas as jogadas válidas, avaliamos os boards resultantes de cada uma delas, e escolhemos a jogada cujo valor retornado pelo predicado value seja máximo, ou, no caso de existir mais que uma jogada com o valor máximo, uma das jogadas dessa lista aleatoriamente. Isto é conseguido através dos predicados **store_value_list(+Board, +ListOfMoves, +Player, -ListOfValues)**, que obtém uma lista de valores de cada board quando se aplica o move correspondente da lista ListOfMoves, e dos predicados **max_list(List, Max)** e **get_random_max_move(+ListOfMoves, +List, +Max, -Move)**, que em conjunto obtêm uma das jogadas de maior valor possível. É também de salientar que até a data todos os jogos feitos entre 2 computadores, um de nível 1 e outro de nível 0, foram sempre ganhos pelo computador de nível 1.

4. Conclusões

Podemos assim concluir que após a realização do projeto, temos um maior conhecimento da praticabilidade de programação funcional, assim como das suas limitações.

Consideramos que existem alguns aspetos a melhorar, principalmente em termos de eficiência das jogadas de computador no nível mais elevado, isto porque, devido a este fazer muitas verificações de jogadas possíveis (tanto dele como do adversário), cada jogada não é imediata e demora cerca de 2 segundos a ser executada (o que neste contexto não consideramos um grande incômodo, já que demora menos do que um humano demoraria).

Em suma, consideramos ter alcançado todos os objetivos propostos para a realização deste projeto, que nos permitiu uma maior perceção do potencial da programação funcional.

5. Bibliografia

-Sterling, Leon. S.; Shapiro, Ehud Y. - “The Art of Prolog : Advanced Programming Techniques”

-Manual de SWI-Prolog: https://www.swi-prolog.org/pldoc/doc_for?object=manual

-Regras de Shobu: https://www.boardgamegeek.com/boardgame/272380/shbu