Faculdade de Engenharia da Universidade do Porto

Programação em Lógica - Trabalho Prático 1

Aplicação em Prolog para um Jogo de Tabuleiro - Nudge

Grupo Nudge_1:

João Paulo Monteiro Leite - up201705312
Márcia Isabel Reis Teixeira - up201706065

2019/2020

Resumo

Este trabalho teve como objetivo principal a implementação do jogo de tabuleiro *Nudge* usando a linguagem *Prolog*. Foram implementados três modos de jogo: jogador *vs* jogador, jogador *vs* computador e computador *vs* computador, sendo que o computador apresenta diferentes níveis de dificuldade, um mais básico e efetivamente aleatório, e outro que recorre a uma avaliação do estado do jogo para tomar a melhor decisão possível.

A realização deste projeto colocou o desafio de trabalhar com uma linguagem declarativa, bastante diferente das linguagens com que costumamos interagir. A solução implementada recorreu a predicados *built-in* e a predicados criados pelo grupo, utilizando estratégias gananciosas e de *backtracking.*

Em suma, este trabalho permitiu assim consolidar os conhecimentos adquiridos nas aulas teóricas e práticas, resultando no que consideramos um trabalho satisfatório com uma interface de utilização que proporciona uma experiência agradável.

Índice

1. Introdução

Este projeto teve como objetivo a implementação em *Prolog* de um jogo de tabuleiro, *Nudge*, com três modos de jogo distintos, jogador contra jogador, jogador contra computador ou computador contra computador. O trabalho foi desenvolvido em ambiente SICStus.

Este relatório encontra-se dividido nos seguintes pontos:

- **O Jogo *Nudge*:** Descrição do jogo e das suas regras;
- **Lógica do Jogo:** Descrição da implementação da lógica do jogo, dividida nas seguintes secções:
    - **Representação do Estado do Jogo:** Descrição da representação interna do jogo e exemplos de tabuleiros em diferentes estados;
    - **Visualização do Tabuleiro:** Descrição dos predicados de visualização do tabuleiro;
    - **Lista de Jogadas Válidas:** Descrição dos predicado de validação de jogadas e construção da lista de jogadas válidas;
    - **Execução de Jogadas:** Explicação sobre o validamento e execução de jogadas;
    - **Final do Jogo:** Descrição dos predicados que verificam o fim do jogo;
    - **Avaliação do Tabuleiro:** Descrição da implementação do predicado que atribuí um valor a cada possível jogada;
    - **Jogada do Computador:** Descrição dos predicados de decisão da jogada do computador para os diferentes níveis de dificuldade.
- **Conclusões:** Conclusões tiradas do projeto.

2. O Jogo *Nudge*

2.1. História

Nudge é um jogo de estratégia abstrata, com regras simples mas uma mecânica inovadora. Destaca-se principalmente pela preocupação ambiental na sua produção e embalamento, sendo as peças feitas de bioplástico biodegradável, e o tabuleiro de cartão totalmente reciclado.

2.2. Regras

O jogo deve ser jogado por 2 jogadores, e é constituído por um tabuleiro quadrado com 5x5 posições, por 3 discos brancos e 3 discos pretos.

Iniciar o jogo

Há 4 disposições dos discos possíveis para iniciar o jogo, ilustradas na figura 1.

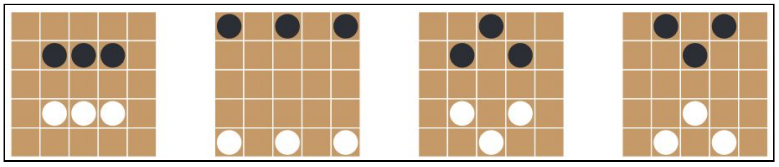

Figura 1 - Posições iniciais do jogo

O primeiro jogador é escolhido aleatoriamente através do lançamento de uma moeda ou de um dos discos. Os jogadores vão depois alternando a vez entre si, fazendo dois movimentos a cada jogada.

Cada disco pode ser movido um quadrado a cada movimento, em qualquer direção que não diagonal (frente, trás ou lados). Uma fila de vários discos pode ser movida longitudinalmente também um quadrado por movimento. Na mesma jogada não é permitido fazer um movimento e depois outro que faça retornar à posição original.

O jogador pode também com um movimento arrastar um disco do adversário (“nudge”). Para o fazer, o jogador deve ter uma fila de discos longitudinalmente maior do que a do adversário (por exemplo, 2 discos contra 1 do adversário).

Os movimentos válidos e inválidos encontram-se ilustrados na figura 2, com explicações em inglês.

O primeiro jogador a conseguir arrastar um disco do adversário para fora do tabuleiro é o vencedor.

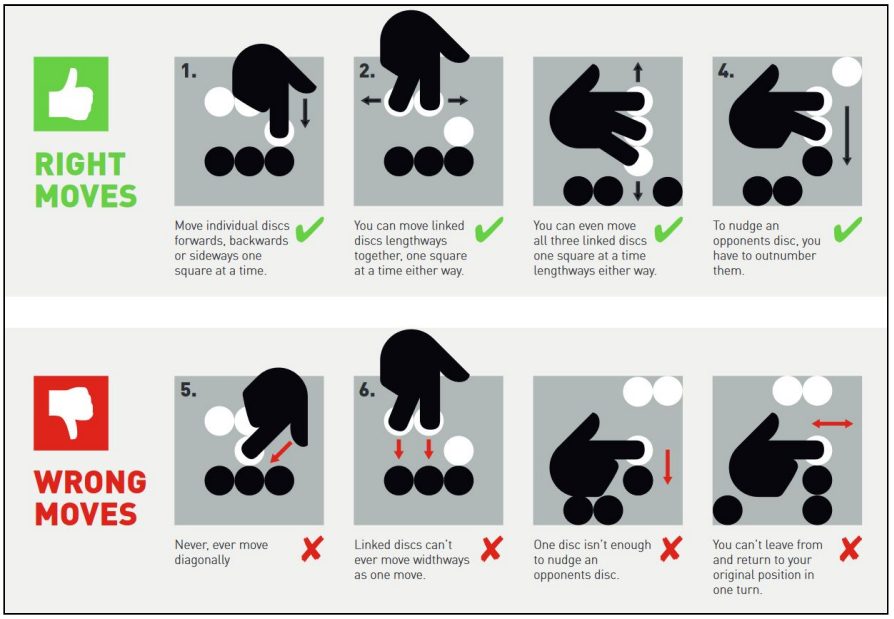

Figura 2 - Ilustração e explicação dos movimentos válidos e inválidos

A história e regras do jogo, bem como as figuras 1 e 2, foram retiradas do site oficial do *Nudge* (link na bibliografia).

3. Lógica do Jogo

3.1. Representação do Estado de Jogo

O estado do tabuleiro do jogo é representado, internamente, por uma lista de listas. O tabuleiro usado tem 5 listas, cada uma delas correspondente a uma fila, e cada lista tem 5 números, que indicam o estado de cada célula. Estes números podem ser 0, caso a célula esteja vazia, 1 caso esteja ocupada por um disco branco, e 2 caso esteja ocupada por um disco preto. Apesar do tabuleiro usado ter tamanho fixo 5:5, os predicados implementados permitem tabuleiros de tamanho variáveis.

O jogador atual é guardado também numa variável que vai sendo passada entre os vários predicados, podendo o seu valor ser 1, caso seja a vez do jogador dos discos brancos, ou 2, caso seja a vez do jogador dos discos pretos.

Seguem-se exemplificações de vários estados do tabuleiro em ProLog (inicial, intermédio e final), acompanhadas de imagens que mostram um tabuleiro de Nudge nesse estado (figuras 3-5, retiradas também do site oficial do Nudge).

Código que inicializa o tabuleiro, criando um tabuleiro com as peças numa das posições iniciais permitidas pelo jogo:

```
initBoard(Board):-
    Board=[[0,0,0,0,0],
           [0,1,1,1,0],
           [0,0,0,0,0],
           [0,2,2,2,0],
           [0,0,0,0,0]].
```

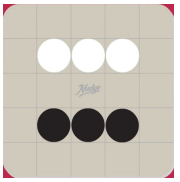

Figura 3

Exemplificação de um possível estado intermédio:

```
Board=[[0,0,0,0,0],
       [0,0,0,1,1],
       [0,0,2,1,0],
       [0,2,2,0,0],
       [0,0,0,0,0]].
```

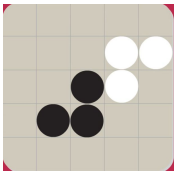

Figura 4

Exemplificação de um possível estado final (dado que o jogo acaba no momento em que uma peça é empurrada para fora do tabuleiro, não se considera que seja necessário guardar as peças que estão fora do tabuleiro):

```
Board=[[0,0,0,0,0],
       [0,0,0,0,0],
       [0,0,0,0,0],
       [0,0,0,2,1],
       [0,2,0,1,1]].
```

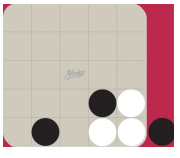

Figura 5

3.2. Visualização do Tabuleiro

O programa foi desenvolvido em ambiente SICStus, utilizando a *font* “Terminal”.

A visualização do tabuleiro é feita através do predicado displayGame(+Board, +Player, +Move). Este predicado começa por chamar printHorizontalCoordinates(+BoardLine, +LetterCode), que irá imprimir as coordenadas horizontais representadas por letras, começando em A e continuando até atingir o número de colunas. De seguida chama getBoardDimensions(+Board, -Lines, -Columns), que retorna o número de linhas e colunas do Board, passando estes valores ao predicado printBoard(+Board, +LineNr, +Columns), que imprime o tabuleiro recursivamente. Por fim recorre ao predicado printPlayer(+Player, +Move), que imprime o símbolo do jogador atual (‘B’ para o jogador dos discos pretos e ‘W’ para o jogador dos discos brancos) assim como o número do movimento atual.

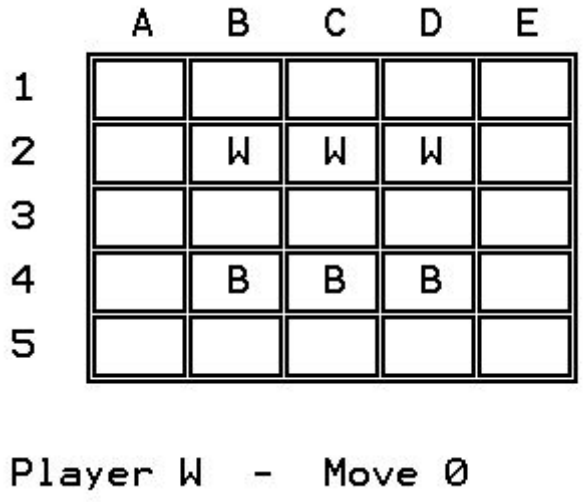

Figura 6 - Output produzido por displayGame(+Board, +Player, +Move), mostrando neste caso o tabuleiro no seu estado inicial.

No predicado printBoard(+Board, +LineNr, +Columns), Board corresponde ao tabuleiro (lista de listas) e LineNr ao código ASCII correspondente ao caractere do número da linha a ser impressa no momento, ou seja, a primeira linha terá LineNr 49, a segunda LineNr 50 e assim sucessivamente.
printBoard imprime o tabuleiro recursivamente, imprimindo os divisores entre linhas (ou a margem superior, no caso da primeira linha) e as coordenadas verticais (1-nº de linhas), usando o predicado printLine(+Line) para imprimir a linha recursivamente e, por fim, efetua a chamada recursiva.
O caso base de printBoard(+Board, +LineNr, +Columns) é o caso em que Board é uma lista vazia; nesta situação é impressa apenas a margem inferior do tabuleiro.

O predicado printLine(+Line) recebe uma linha do tabuleiro, ou seja, uma lista, e imprime-a, começando por imprimir o divisor das células (linha dupla vertical),

usando depois o predicado printCell(+Cell) para imprimir o conteúdo da célula e, por fim, efetua a chamada recursiva.
O case base de printLine(+Line) é o caso em que Line é uma lista vazia, caso em que é impresso apenas o divisor das células.

Por último, o predicado printCell(+Cell) imprime apenas o símbolo associado ao conteúdo da célula: no caso da célula vazia (0) é um espaço, um disco branco (1) é a letra W e um disco preto (2) é a letra B.

Para tornar o código mais legível e organizado existem também os predicados printDivider(+NrColumns), printTopBorder(+NrColumns) e printBottomBorder(+NrColumns), que apenas imprimem os divisores entre filas, a margem superior e a margem inferior do tabuleiro, respetivamente.
Para impressão no ecrã são usados os predicados *built-in* put_code(+Code) e write(+Term).

Ao iniciar o jogo com o predicado “play”, são chamados os predicados chooseBoard(-BoardNumber) e readGameMode(-Mode, -Level).

chooseBoard permite escolher de entre 4 configurações iniciais do tabuleiro.

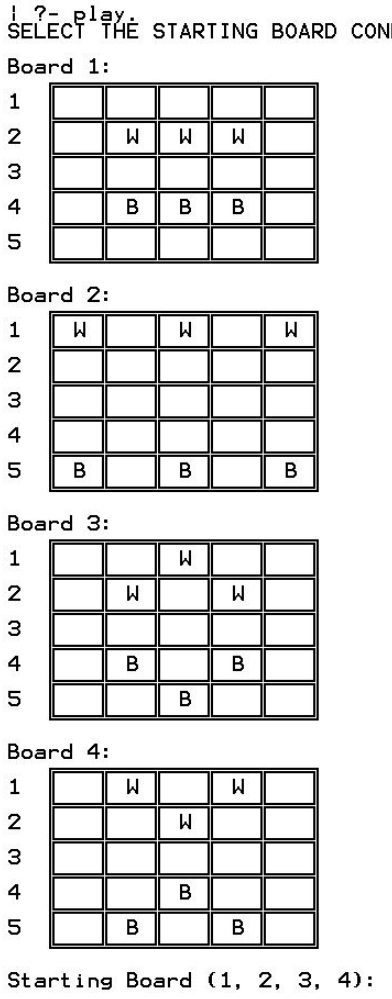

Figura 7 - Output produzido por chooseBoard(-BoardNumber).

readGameMode permite ao utilizador selecionar entre 3 modos de jogo, jogador contra jogador, computador contra computador, ou jogador contra computador.

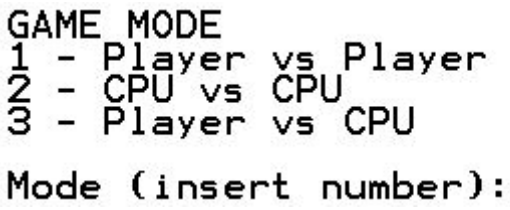

Figura 8 - Output produzido por readGameMode(-Mode, -Level).

Caso seja selecionado um modo de jogo que não jogador *vs* jogador, é também possível selecionar o nível de AI.

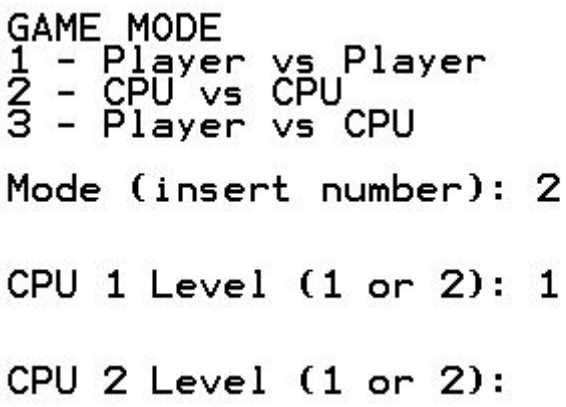

Figura 9 - Escolha do nível das AI.

Para realizar uma jogada, o utilizador recebe um conjunto de instruções, apresentadas pelo predicado readMove(+Board, +Player, +MoveNr, +OriginalBoard, -Move). Este começa por ler o tipo de jogada que irá ser feita (mover disco ou mover linha), seguida da posição (coordenada horizontal e vertical) e direção do movimento. Segue-se depois uma chamada a isPlayerMoveValid(+Board, +Player, +OriginalBoard, +Move) que irá validar a jogada submetida. Caso não seja válido, repete-se readMove.

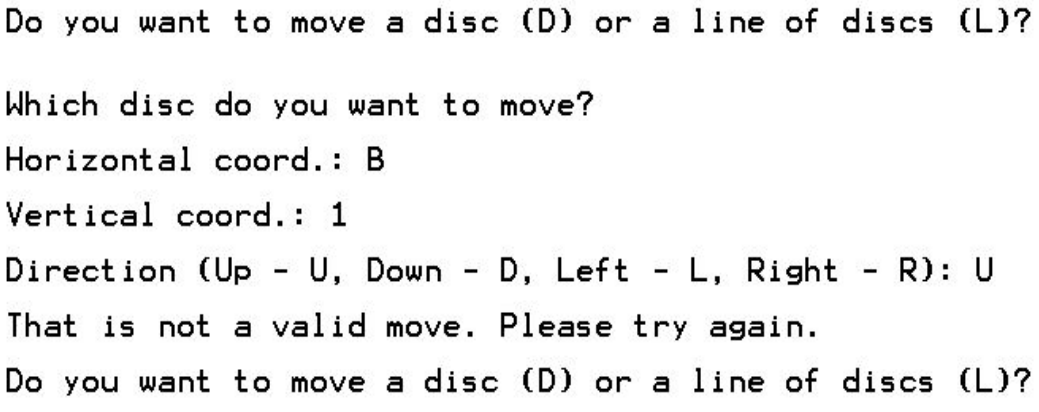

Figura 10 - Jogada inválida.

A maioria dos inputs do utilizador são tratados pelo predicado getCodeInput(-Code), que lê da *input* stream, utilizando o predicado *built-in* "read_line", verificando se o tamanho lido tem comprimento 1 (pois todos os *inputs* são caracteres), e caso se tratem de letras minúsculas transformar em maiúsculas para tornar o programa mais *user-friendly*.

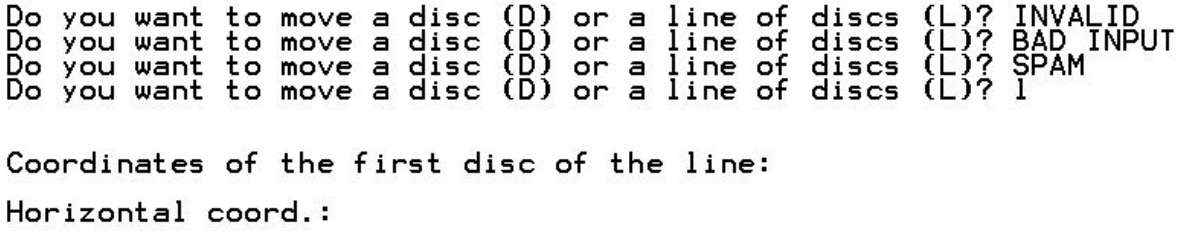

Figura 11 - Demonstração de inputs inválidos e *case-insensitive*.

3.3. Lista de Jogadas Válidas

Para encontrar a lista de todas as jogadas válidas, recorremos ao predicado validMoves(+Board, +Player, +OriginalBoard, -ListOfMoves). Este, por sua vez, faz duas chamadas ao predicado *built-in* setof(+Template, +Goal, -Set), uma para cada tipo de move (D e L). O setof recebe como "Template" uma lista com um "Move" ([Horizontal, Vertical, Direction, Type], sendo Type "D" ou "L"). "Goal" será uma chamada ao predicado isPlayerMoveValid(+Board, +Player, +OriginalBoard, +Move), explicado com mais detalhe na secção seguinte. O terceiro argumento de setof é uma lista onde serão guardados todos os moves válidos do tipo especificado. No fim, faz-se um *append* de ambas as listas, e coloca-se o resultado em "ListOfMoves".

3.4. Execução de Jogadas

O ciclo principal de jogadas é feito pelo predicado game(+Mode, +Level, +Board, +OriginalBoard, +Player, +Move, +Winner). Este irá dar display do board, verificar se o jogo terminou e executar os movimentos, chamando-se a si mesmo recursivamente.

A execução de jogadas é realizada pelo predicado move, no entanto, a sua validação é feita antes, ou pelo predicado readMove(+Board, +Player, +MoveNr, +OriginalBoard, -Move), caso seja uma jogada de um humano, ou pelo predicado validMoves(+Board, +Player, +OriginalBoard, -ListOfMoves), caso seja a AI, ambos recorrendo ao predicado isPlayerMoveValid(+Board, +Player, +OriginalBoard, +Move).

Para verificar se uma jogada é válida utilizamos o predicado isPlayerMoveValid(+Board, +Player, +OriginalBoard, +Move), que recebe o jogador atual, o estado do *Board*, o *Board* original (pois a primeira e segunda jogadas não

podem ser inversas) e a jogada atual. Primeiro recorre-se ao predicado validCoords(+Board, +Horizontal, +Vertical), para verificar se as coordenadas estão dentro dos limites do *board*. De seguida, chama-se getPosition(+Board, +Horizontal, +Vertical, -Content), que retorna o conteúdo na posição Horizontal:Vertical, conteúdo esse que vai ser comparado com o player para verificar se se trata de uma peça do mesmo. Testa-se o movimento na direção do movimento, utilizando o predicado positionFromDirection(+Direction, +Coordinates, +Increment, -NewH, -NewV), que retorna as novas coordenadas após a jogada, coordenadas estas que vão ser testadas com validCoords. Uma vez que o estado de jogo após o segundo movimento não pode ser igual ao estado inicial da primeira jogada, utiliza-se checkResetBoard(+Board, +OriginalBoard, +Move), para comparar os *boards* após a jogada. No fim de todas as validações, caso o tipo de movimento seja "D" (disco), verifica-se se a posição para onde se está a mover está vazia, finalizando-se um movimento válido. Caso o movimento seja "L" (linha), valida-se uma linha com validateLineMove(+Board, +Player, +MoveInfo).

validateLineMove, utiliza o predicado countLineOfDiscs(+Board, +StartingCoordinates, +Direction, +Disc, -NumberOfDiscs) para contar o número de discos "Disc" que existem nessa linha, começando em "StartingCoordinates" e na direção "Direction". Caso o número seja superior a 1, confirma-se que o movimento é do tipo linha. Move-se a peça do início da fila, para a posição após o fim da mesma, verificando se as coordenadas dessa posição são válidas (validCoords). Por fim, verifica-se se essa posição está livre, caso sim, termina a validação. Caso seja uma peça do oponente, contam-se as suas peças, caso a fila do jogador seja maior que a do oponente, o movimento é válido.

O predicado move(+Move, +Board, -NewBoard), recebe uma jogada, o estado atual do jogo e retorna um novo estado de jogo. Consoante o tipo do "Move", utilizam-se os predicados simpleMove(+Move, +Board, -NewBoard) (para mover discos), ou multipleMove(+Move, +Board, -NewBoard) (para mover linhas).

simpleMove, move a peça da sua posição atual, para a posição indicada pelo "Move", devolvendo "NewBoard".

multipleMove, conta o número de discos do jogador, move o primeiro elemento da linha para a posição imediatamente a seguir ao fim da mesma. Caso essa posição seja uma peça do jogador inimigo, aplica-se o mesmo processo, movendo a primeira peça da fila do oponente para o fim. Caso essa posição seja inválida, a peça é removida do tabuleiro.

3.5. Final do Jogo

A verificação de fim de jogo é realizada pelo predicado gameOver(+Board, -Winner), após o display do *board*, e antes da jogada seguinte. gameOver recebe como argumento o estado do jogo, e retorna o número representativo do jogador vencedor. Caso este número seja diferente de 0, um jogador venceu o jogo.

Para verificar se o jogo terminou, o predicado gameOver chama countBoardPieces(+Board, +Piece, -Number), que vai contar o número de peças presentes no *board* para cada um dos jogadores. Caso o número de peças de um jogador seja diferente do outro, o jogador com maior número de peças em jogo será o vencedor.

3.6. Avaliação do Tabuleiro

A avaliação do valor de um tabuleiro é feita pelo predicado value(+Board, +Player, -Value).
Um tabuleiro pode ter 4 valores diferentes:

- 3, se o jogador ganhar com esse tabuleiro (tiver mais peças que o adversário);
- 2, se o tabuleiro permitir que o jogador faça um movimento para vencer;
- 1, se o tabuleiro for neutro;
- 0, se o tabuleiro permitir que o adversário faça um movimento para ganhar (e o jogador perder).

O predicado value utiliza outros predicados para verificar as situações descritas acima e atribuir o valor a Value adequadamente.
Inicialmente, o predicado value utiliza o predicado gameOver para verificar se o jogador vence com esse tabuleiro. Caso isso não se verifique, usa o predicado isWinningPosition(+Board, +Player) para verificar se o adversário pode efetuar movimentos para vencer. Por fim, verifica com o isWinningPosition se o jogador pode efetuar movimentos para vencer. Caso esta situação também não se verifique, Value é 1, indicando um tabuleiro neutro.

3.7. Jogada do Computador

A jogada do computador é escolhida com o predicado chooseMove(+Board, +Player, +OriginalBoard, +Level, -Move). O parâmetro Level indica os níveis de dificuldade e o modo escolhido pelo utilizador no ínicio:

- 0 - jogador vs jogador (não é relevante);
- 1 - jogador vs computador de nível 1, ou computador de nível 1 vs computador de nível 1;
- 2 - jogador vs computador de nível 2, ou computador de nível 2 vs computador de nível 2;
- 3 - computador de nível 1 com discos brancos vs computador de nível 2 com discos pretos;
- 4 - computador de nível 2 com discos brancos vs computador de nível 1 com discos pretos.

Consoante o Player e o Level, o predicado chooseMove utiliza o predicado moveAILevel1(+Board, +Player, +OriginalBoard, +Move) ou moveAILevel2(+Board, +Player, +OriginalBoard, +Move), caso se trate de uma AI de nível 1 ou de nível 2, respetivamente.

A AI de nível 1, implementada no predicado moveAILevel1, apenas seleciona um movimento aleatório de todos os movimentos possíveis encontrados com validMoves, utilizando o predicado da biblioteca *random* random_member(-X, +List:list).

A AI de nível 2, implementada no predicado moveAILevel2, cria uma nova lista de listas de movimentos a partir da lista obtida com validMoves, usando o predicado listMovesByValue(+Board, +Player, +ListOfMoves, +ListByValues). Nesta lista de listas os movimentos estão divididos de forma ordenada pelo valor do tabuleiro resultante, sendo que a primeira lista corresponde aos movimentos de maior valor (3) e a última aos de menor valor (0). O predicado listMovesByValue percorre a lista de movimentos de forma recursiva, verifica o valor de cada movimento com o predicado value, e coloca-o (append) na lista de ListByValues correspondente. Por fim, o predicado moveAILevel2 escolhe um movimento aleatório de uma das listas, começando pelas lista de movimentos de maior valor e, se estiver vazia, avançando para as listas de menor valor.

4. Conclusões

Apesar de trabalhoso, o trabalho dedicado ao projeto ajudou a expandir e interiorizar os conhecimentos de Prolog através da aplicação dos mesmos na realização de um programa mais complexo.
Consideramos que todos os objetivos foram cumpridos, de acordo com os critérios estabelecidos para este projeto.
Uma das maiores dificuldades encontradas no desenvolvimento do projeto foi encontrar um bom critério de avaliação do tabuleiro, a ser utilizado pela AI inteligente. No final, esta dificuldade foi maioritariamente superada, sendo a AI capaz de fazer jogadas geralmente ótimas. Algo que podia ser melhorado seria passar o número do movimento atual para o predicado de avaliação, de modo a poder escolher melhor a próxima jogada.

Concluindo, o trabalho foi realizado com sucesso, cumprindo os requisitos e expandindo o nosso conhecimento da linguagem Prolog.

Bibliografia

- nudgegames.co.uk - site do jogo *Nudge*
- https://sicstus.sics.se/documentation.html - documentação SICStus
- https://sicstus.sics.se/sicstus/docs/4.3.0/html/sicstus/lib_002dlists.html - documentação sobre a biblioteca de listas