**Faculdade de Engenharia da Universidade do Porto**


# Relatório Final PLOG

## *Manalath*

2018/2019

Grupo *Manalath_1*

Susana Maria de Sousa Lima, up201603634

Gonçalo Regueiras dos Santos, up201603265

# Índice

# Introdução

No âmbito da unidade curricular de Programação em Lógica do 3º ano do MIEC, foi proposta a implementação de um jogo de tabuleiro em PROLOG, com diferentes modos de jogo: Jogador contra Jogador, Jogador contra Computador e Computador contra Computador. Nos modos de interação com o computador, pelo menos dois níveis de dificuldade deveriam ser incluídos.

A interface de jogo deveria ser implementada em modo de texto.

O jogo escolhido, de entre uma variedade de opções apresentadas no enunciado, foi o *Manalath*.

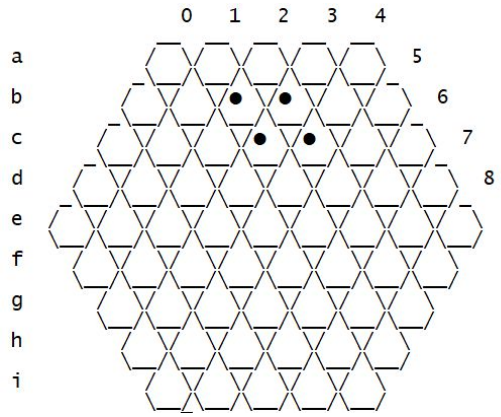
# Descrição do Jogo

*Manalath* é um jogo de tabuleiro abstrato, de dois jogadores, da família dos jogos do tipo *Yavalath*. Foi criado em 2012 por Dieter Stein e Néstor Romeral Andrés e incluído no livro "Yavalath & Co" da nestorgames. O tempo médio de jogo é 30 minutos.

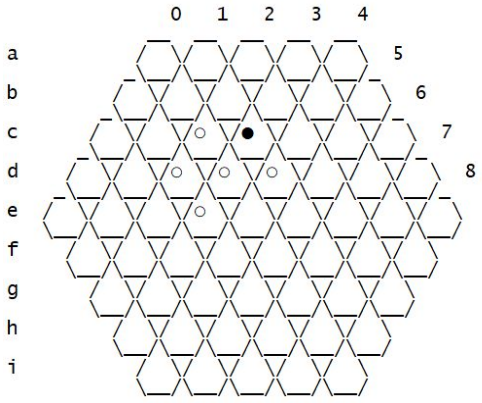
*Manalath* é  jogado num tabuleiro hexagonal com 61 espaços. Cada jogador tem 30 peças de uma cor atribuída (geralmente preto e branco).

Começa com um tabuleiro vazio e, em cada turno, o jogador atual escolhe uma peça de qualquer cor (própria ou do oponente) e coloca-a num espaço vazio do tabuleiro. No entanto, nunca pode colocar uma peça de tal forma que um grupo de mais de 5 peças seja criado. Um grupo é formado por peças vizinhas, que possuam um lado em comum.

Grupo de 4 peças pretas




Grupo de 5 peças brancas




Se no final do turno de um jogador houver um grupo de 5 peças da sua cor, este ganha. Por outro lado, se houver um grupo de 4, o jogador perde. Se ambas as situações ocorrerem no final da ronda, o jogador perde.

Caso não hajam mais jogadas possíveis, o jogo termina em empate.

# Lógica do Jogo

## Representação do estado do jogo

Os dois elementos fundamentais para a representação do estado do jogo são o tabuleiro e os jogadores. As funções diretamente relacionadas com o tabuleiro foram implementadas no ficheiro *board.pl* e as diretamente relacionadas com os jogadores, no ficheiro *player.pl*

O tabuleiro é representado internamente através de uma lista de termos, em que cada termo representa uma célula do tabuleiro. Cada termo, designado por cell - *cell(X,Y,C)* - possui uma coordenada X, uma coordenada Y e um estado/cor, sendo que:

- *X*: representa o número da coluna diagonal da esquerda para a direita (numerada apenas por número pares)
- *Y*: representa o número da linha do tabuleiro
- *C*: identifica o estado da célula
    - *emptyCell*: caso esteja vazia
    - *whitePiece*: caso esteja ocupada por uma peça branca
    - *blackPiece*: caso esteja ocupada por uma peça preta

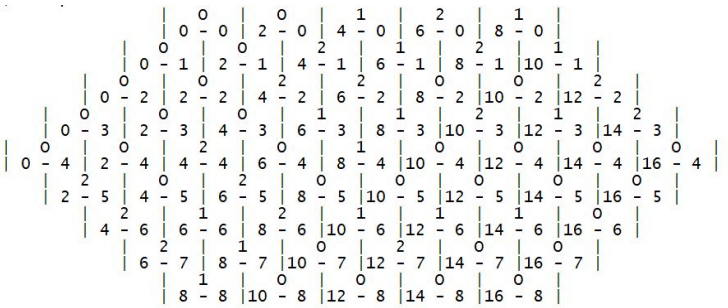
Por outro lado, um jogador é representado por um termo dinâmico - *player(PlayerId, PlayerColor, CurrentlyPlaying,Bot)* - no qual:

- *PlayerId* : representa o identificador fixo do jogador, como existem dois jogadores varia entre 1 e 2;
- *PlayerColor*: representa a cor fixa/atribuída do jogador, ao jogador 1 é inicialmente atribuída a cor *blackPiece* e ao jogador 2 a cor *whitePiece*;
- *CurrentlyPlaying*: indica se o jogador é o jogador atual, isto é, representa o jogador que está a jogar em cada jogada. Se o seu valor for 1 significa que o jogador em questão é o jogador atual, se não, o seu valor é 0.
- *Bot*: indica se um jogador é um *bot* ou um humano, no primeiro caso o seu valor é 1, no segundo é 0:
    - No modo Jogador contra Jogador, ambos os jogadores são humanos pelo que o valor de *Bot* dos dois é 0;
    - No modo Jogador contra Computador, o primeiro jogador é humano (*Bot* igual a 0) e o segundo é *bot* (*Bot* igual a 1)
    - No modo Computador contra Jogador, o primeiro jogador é *bot* (*Bot* igual a 1) e o segundo é humano (*Bot* igual a 0)
    - No modo Computador contra Computador, ambos os jogadores são *bots* pelo que o valor de *Bot* dos dois é 1.

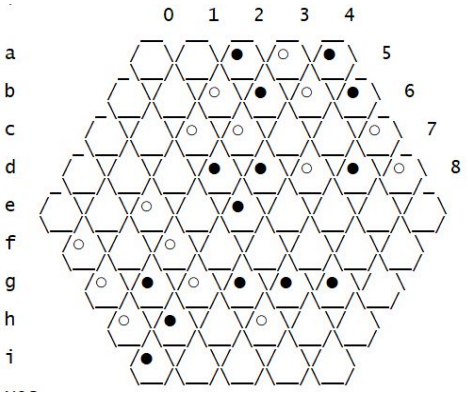
## Visualização do tabuleiro

Face ao relatório intermédio, a visualização do tabuleiro sofreu grandes alterações para melhorar a usabilidade e a fácil percepção das casas adjacentes e do tabuleiro em geral. O tabuleiro tornou-se mais hexagonal e a representação das peças, anteriormente *0*, *1* ou *2*, é agora feita com pequenos círculos[1] das respetivas cores das peças, ⬤ e ○ (no SICStus apresentam o mesmo tamanho). O predicado *print_board(+Board)* é o responsável por dar início ao processo de desenho do tabuleiro, imprimindo os índices das primeiras colunas e o "topo" da primeira linha. A partir daí, o predicado *print_lines(+Board, +CurrentLineNumber, +TotalNumberLines)* chama-se a si mesmo recursivamente, imprimindo linha por linha. Em cada linha, é impressa uma letra, seguida dos hexágonos correspondentes a essa linha. Na metade superior do tabuleiro, são ainda impressos os topos do primeiro e do último hexágono e o índice da coluna que surgirá na linha seguinte.
Para evitar um erro de representação do SICStus, em que caso o caractere correspondente à peça preta fosse o último a ser desenhado, outro símbolo era escrito na tela em vez do correto, no fim de cada linha é desenhado um caractere invisível, corrigindo assim esse erro.
Como se pode ver pelas seguintes imagens, as melhorias entre a representação antiga e a nova são inegáveis, tornando a experiência do utilizador muito mais agradável e *user-friendly*.

Antiga representação



Nova representação




O sistema de coordenadas inicial, apesar de facilitar bastante os cálculos, era pouco intuitivo e obrigava a colocar as coordenadas dentro das casas do tabuleiro, o que o tornava pouco apelativo e confuso. Por esse motivo, foi criado outro sistema de coordenadas, em que cada linha é representada por uma letra e cada "coluna" (na diagonal) tem o respetivo índice por cima. Sempre que é preciso fazer alguma conversão entre as coordenadas *LetterNumber* mostradas ao utilizador e as coordenadas X-Y internas, são utilizados os predicados *coordsToUser*(+*Board*, +*X*, +*Y*, -*Letter*, -*Number*) e *userToCoords*(+*Board*, +*Letter*,

[1] A correta representação dos caracteres especiais está assegurada se for utilizada a fonte *Lucida Console* no SICstus 4.4.1

*+Number*, *-X*, *-Y*) que convertem uma notação na outra e verificam que as coordenadas são válidas e que existe uma célula no tabuleiro com essas coordenadas.

## Lista de Jogadas Válidas

No jogo em questão, uma jogada é classificada como válida, se a peça sobre a qual incide se encontra vazia, se não formar um grupo de mais de 5 elementos de uma das cores disponíveis e não exceder o número máximo de peças da cor jogadas, 30 peças por cor.

De modo a obter a lista de jogadas válidas para um certo tabuleiro, o predicado *valid_moves(+Board, -ListOfMoves)* foi implementado. As jogadas na lista são representadas no formato *(X,Y,Color)*, em que *X* e *Y* são as coordenadas da célula do tabuleiro e *Color* é a cor/estado da célula.

Este predicado é usado em duas situações distintas do jogo. Por um lado permite validar a possibilidade de jogada do jogador atual, caso não existam jogadas válidas para ambos os jogadores, o jogo termina em empate. Por outro lado, como retorna a lista de jogadas válidas para o tabuleiro de jogo, é usado na parte da escolha de jogadas do *Bot*.

## Execução de Jogadas

Para uma jogada ser executada é necessário obter informação sobre a célula do tabuleiro onde jogar e a cor da peça a jogar.

No caso do jogador ser um humano, essa informação é pedida ao utilizador com o formato *LetterNumber* (exemplo *a0*, linha *a*, coluna 0), representativo das coordenadas da célula no tabuleiro, e *Color*, representativo da cor do peça (*blackPiece*, *black* ou *b* para cor preta, *whitePiece*, *white* ou *w* para a cor branca). A informação obtida é então validada. Caso seja inválida, nova informação é pedida ao utilizador. Isto acontece quando a informação obtida do utilizador é errada, por exemplo coordenadas ou cor inexistentes, ou quando a jogada não é válida (situação explicada no ponto anterior).

No caso do jogador ser um *bot* essa informação é obtida através do predicado *choose_move*, explicado na secção “Jogada do Computador” do relatório.

Após a obtenção de coordenadas e cor da peça a jogar válidas, a jogada é executada.

O predicado *move(+Move, +Board, -NewBoard)* foi implementado com o objetivo de executar uma jogada, *Move*, no formato de lista de 3 elementos: coordenadas X e Y e a cor/estado da célula. Retorna o tabuleiro atualizado.

Após a jogada o jogador é atualizado recorrendo ao predicado *switchCurrentPlayer*, que altera os termos dinâmicos que representam os jogadores, trocando os campos de *CurrentlyPlaying* de 1 para 0 e vice-versa.

## Avaliação do Tabuleiro e Final do Jogo

Como forma de avaliar o estado do tabuleiro foi implementado o predicado *value(+Board, -Value)*. Este predicado avalia o estado do tabuleiro (*Board*):

- *Value* recebe o valor do identificador do jogador que ganhou, caso isso aconteça;
- *Value* recebe o valor -1, caso não existam jogadas válidas possíveis;
- *Value* recebe o valor 0 se o jogo não tiver terminado.

O predicado *game_over(+Board, +Winner)* foi implementado para, após cada jogada, verificar se o jogo terminou ou não, e caso tenha terminado em que condições terminou. Para esse efeito, chama o predicado *value*. No ciclo de jogo é chamado após o predicado *move* referido no ponto anterior.

O jogo termina quando um dos jogadores ganhar ou não existirem mais jogadas válidas, ocorrendo um empate.

## Jogada do Computador

Como referido na introdução do relatório e, de acordo com o estipulado no enunciado, foram implementados dois níveis de jogo para o *bot* (aplicados no modo Jogador contra Computador e Computador contra Computador.

O nível *easy* implementa jogadas semi aleatórias, isto é, se existir uma jogada que impeça o bot de perder ou que lhe permita vencer ele executa essa jogada, caso contrário as jogadas são escolhidas aleatoriamente. O nível *hard* escolhe a melhor jogada possível de entre uma lista de jogadas, neste caso, escolhe a jogada que tenha uma pontuação menor.

Na escolha de uma jogada, em primeiro lugar, são obtidas todas as jogadas válidas para o tabuleiro atual através do predicado *valid_moves* (mencionado anteriormente). Estas jogadas são analisadas no predicado *analyse_validMoves* - *analyse_validMoves(+Board, +Player_Color, +ListOfMoves, +SkipStep,+Tmp, -Cells)* - que pontua um lista de jogadas possíveis de melhor para pior. Este predicado analisa vários casos possíveis de jogadas:

- se a jogada a ser analisada levar à vitória para o jogador atual, é pontuada com o valor de -600;

- se a jogada a ser analisada levar à derrota do jogador atual, é pontuada com o valor de 500;
- se a célula da jogada a ser analisada possuir quatro vizinhos da cor do oponente, processa os seus vizinhos, sendo que, se um deles possuir zero ou um vizinhos da cor do oponente, cria uma nova jogada para o primeiro vizinho do vizinho que seja válido e não leve à vitória do adversário. Esta jogada criada é pontuada com -550. A jogada principal, que não foi pontuada, volta a ser analisada, saltando este passo (de modo a evitar um ciclo infinito)
- se a jogada a ser analisada impedir a vitória do oponente, é pontuada com o valor de -500;
- se a célula da jogada a ser analisada possuir mais do que um vizinho da cor do oponente, a jogada é pontuada com o valor do número de vizinhos da cor do oponente multiplicado por -20;
- se a célula da jogada a ser analisada possuir mais do que dois vizinhos da cor do jogador atual, procura a primeira célula vizinha que seja válida e cria uma jogada para essa célula com o valor de -30. A jogada principal, que não foi pontuada, volta a ser analisada, saltando este passo (de modo a evitar um ciclo infinito)
- se a célula da jogada a ser analisada possuir mais do que um vizinho da cor do jogador atual , a jogada é pontuada com o valor do número de vizinhos da cor do jogador multiplicado por -20;
- se nenhum destes casos ocorrer, a jogada é pontuada com -5. caso sua cor seja a do jogador atual, e 0. caso a sua cor seja a cor oposta ( prioriza a cor do jogador sobre a cor contrária)

Ambos os níveis, *easy* e *hard,* chamam este predicado, no entanto escolhem a jogada da lista resultante de forma diferente, como descrito anteriormente.

# Conclusões

A implementação do jogo *Manalath* em PROLOG exigiu um grande esforço por parte dos elementos do grupo uma vez que não tinham experiência com o paradigma de programação declarativa e foi o primeiro projeto realizado nesta linguagem. No entanto, permitiu expandir o conhecimento sobre a linguagem e consolidar a teoria lecionada nas aulas da unidade curricular.

Embora todos os objetivos para o trabalho tenham sido cumpridos, uma melhoria adicional ao mesmo seria modificar o nível *hard* do *bot* de modo a procurar a melhor jogada tendo em consideração as possíveis jogadas seguintes.

# Bibliografia


*Board Games*. s.d. https://www.boardgamegeek.com/wiki/page/thing:127993 (acedido em Outubro de 2018).

"Manalath." *nestorgames*. s.d. https://nestorgames.com/rulebooks/MANALATH_EN.pdf (acedido em Outubro de 2018).

*The opinionated Gamers*. s.d. https://opinionatedgamers.com/2018/04/24/james-nathan-yavalath-and-manalath// (acedido em Outubro de 2018).

*Patel*, Amit. "Hexagonal Grids." *RedBlobGames*. Março 2013. https://www.redblobgames.com/grids/hexagons/ (acedido em Outubro de 2018).