Asterismo

Relatório Final TP1

Mestrado Integrado em Engenharia Informática e Computação

Programação em Lógica

Turma 4

Daniel Ferreira Brandão up201705812@fe.up.pt
Pedro Miguel Braga Barbosa Lopes Moás up201705208@fe.up.pt

17 de Novembro de 2019

Resumo

O projeto consiste na implementação do jogo Asterismo em PROLOG. É um jogo competitivo para dois jogadores, no qual o objetivo é capturar peças, mantendo as restantes seguras. PROLOG é uma linguagem de programação com um paradigma diferente ao que estamos habituados, no entanto este projeto ajudou-nos a à sua melhor compreensão e utilização de novas abordagens.

Todas as regras do jogo original foram implementadas com sucesso em 4 modos de utilização: Humano vs Humano, Humano vs CPU, CPU vs Humano e CPU vs CPU.

1. Introdução

Este trabalho foi desenvolvido no âmbito da unidade curricular Programação em Lógica, 3º ano do Mestrado Integrado em Engenharia Informática e Computação, tendo como objetivo aprofundar o conhecimento teórico e prático da matéria lecionada referente à linguagem de programação PROLOG. Selecionamos o jogo de tabuleiro Asterismo devido à jogabilidade simples de compreender, e também pela ideia de trabalhar com um tabuleiro de padrões hexagonais.

O relatório está estruturado da seguinte forma:

2. O jogo Asterismo

2.1. Descrição

Asterismo é um jogo de tabuleiro para dois jogadores, criado em 2019 por Giuliano Polverari. É jogado num tabuleiro hexagonal com 63 peças (21 azuis, 21 amarelas e 21 vermelhas), aleatoriamente colocadas no início do jogo, formando uma árvore compacta. O objetivo de cada jogador é colecionar um certo número de peças mantendo todas as peças no tabuleiro seguras.

2.2. Regras

Uma peça é considerada segura quando está adjacente a pelo menos três peças ou a duas a sua cor.

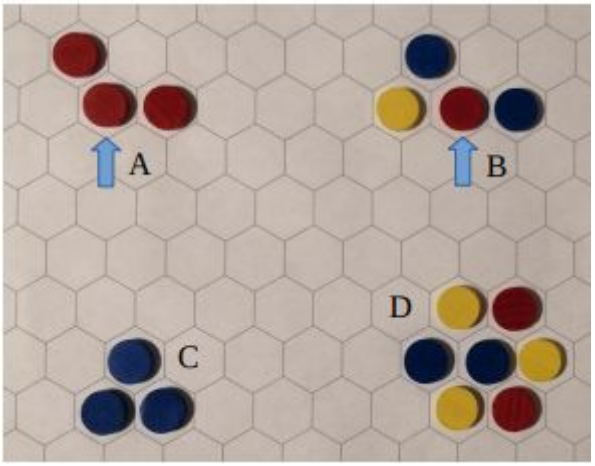

Figura 1 - Nos Casos A e B, as setas apontam para peças seguras, sendo que as outras não têm esse estatuto. No exemplo C e D todas as peças estão numa posição segura.

Para preparar o tabuleiro coloca-se, aleatoriamente, cada uma das 63 peças num hexágono diferente, de forma que, a partir do centro do tabuleiro, se crie uma árvore compacta de peças. Não podem ser criadas secções separadas. As peças que não resultarem seguras são removidas.

Figura 2 - Exemplo válido de configuração inicial do tabuleiro. A peça marcada com um X não se encontra segura, logo será removida.

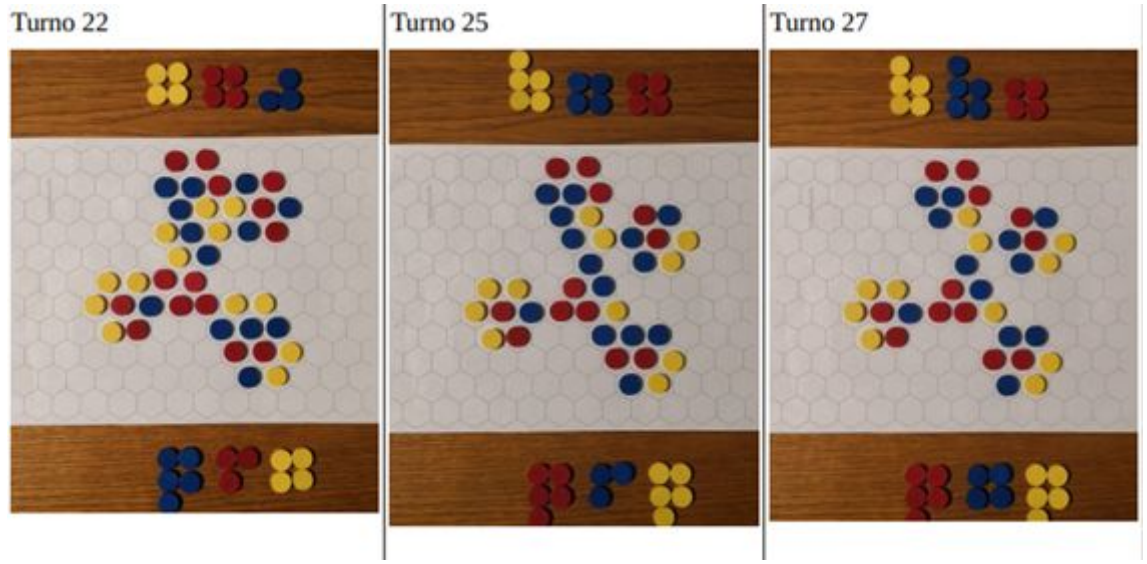

Figura 3 - Turnos 22, 25 e 27 de um exemplo de jogo de asterismo

O jogo prossegue em turnos. Um de cada vez, cada jogador escolhe uma peça e junta-a à sua colheita, sendo apenas válido quando se certifica que todas as outras peças se mantêm seguras. O primeiro jogador a colecionar 5 peças de cada cor (15 peças no total) ganha.

3. Lógica do Jogo

O jogo inicia-se através do predicado *play/0*. Este inicializa o tabuleiro do jogo, pede ao utilizador para introduzir uma dificuldade e um modo de jogo, efetua o ciclo de jogo e, por fim, imprime uma mensagem a indicar o vencedor.

O ciclo do jogo é implementado por *gameloop(+Game, +Mode, +Difficulty, +Player, -Winner),* e consiste nos seguintes passos:

1. Se o jogo terminou, imprime o seu estado e termina a função, caso contrário, continua.
2. Imprime o estado do jogo
3. Obtém uma jogada válida do CPU/Utilizador, dependendo do modo de jogo e do jogador atual
4. Efetua a jogada
5. Obtém o próximo jogador
6. Faz *gameloop* com os valores novos

O extrato de código abaixo mostra a definição do predicado em PROLOG:

```
gameloop(Game, Mode, Difficulty, Player, Winner):-
    display_game(Game, Player),
    get_move(Game, Mode, Difficulty, Player, Move),
    display_move(Move, Player),
    move(Move, Player, Game, NewGame),
    next_player(Player, NewPlayer),
    gameloop(NewGame, Mode, Difficulty, NewPlayer, Winner).
```

3.1. Representação do Estado do Jogo

O estado do jogo é representado por uma lista de dois elementos. Em primeiro lugar, uma lista de listas com o conteúdo de cada posição do tabuleiro, de modo a tornar o seu processamento relativamente simples, sendo que, para determinar quais são as posições que rodeiam um hexágono, é apenas necessário verificar se a fila é par ou ímpar. O segundo elemento da estrutura é uma lista com duas listas, cada uma utilizada para representar as peças capturadas por cada jogador.

Representação de um exemplo de estado inicial:

```
[
     [0,0,0,0,0,1,1,0,0,0,0,0],
    [0,0,0,0,0,2,2,1,2,1,0,0],
     [0,0,0,0,1,2,3,3,1,2,3,0],
    [0,0,0,0,0,3,2,3,2,1,3,0],
     [0,0,0,2,1,3,2,2,2,3,0,0],
    [0,0,3,3,3,1,1,2,2,2,0,0],
     [0,0,3,0,2,1,1,3,3,1,0,0],
    [0,0,0,3,1,3,3,2,2,2,2,0],
     [0,0,0,3,1,3,3,1,1,3,0,0],
    [0,0,0,0,1,2,1,1,2,3,0,0],
     [0,0,0,0,0,0,0,0,1,0,0,0]
]-[[],[]]
```

Representação de um exemplo de estado final, saindo o jogador 1 vitorioso.

```
 [
      [0,0,0,0,0,0,0,0,0,0,0,0],
     [0,0,0,0,0,0,0,1,2,0,0,0],
      [0,0,0,0,0,0,2,1,2,0,0,0],
     [0,0,0,0,0,0,0,3,0,3,0,0],
      [0,0,1,1,0,0,0,2,2,3,0,0],
     [0,0,1,3,3,0,0,0,3,3,0,0],
      [0,0,3,0,3,0,0,0,1,0,0,0],
     [0,0,0,2,1,1,0,3,1,0,0,0],
      [0,0,0,2,1,3,2,1,2,0,0,0],
     [0,0,0,0,0,0,2,2,2,0,0,0],
      [0,0,0,0,0,0,0,0,0,0,0,0]
 ]-[[1,1,1,1,1,2,2,2,2,2,3,3,3,3,3],[1,1,1,1,1,2,2,2,2,3,3,3,3,3]]
```

3.1.1. Criação do tabuleiro

Como o estado inicial do tabuleiro é aleatório, criou-se um algoritmo que o preenche com um certo número configurável de peças de cada cor.

```
/** ---- Settings ---- **/
pieceConfig(21-21-21).
```

Este algoritmo cria um formato de tabuleiro aleatório que tenta espalhar as peças de uma maneira semelhante ao que aconteceria na realidade (largar as peças ao mesmo tempo no meio do tabuleiro e espalhar criando um formato compacto), colocando as diferentes peças aleatoriamente. O algoritmo divide-se em duas fases:

1. Escolhe-se uma posição central (aleatoriamente entre ⅖ e ⅗ de cada dimensão do tabuleiro), colocando-se peças na fronteira das peças colocadas previamente, ou seja, na primeira iteração colocam-se peças à volta do centro, na segunda à volta das peças colocadas na primeira, etc.
2. No final de um número de iterações definido, é escolhida uma peça aleatória das recentemente colocadas, preenchendo os seus vizinhos e adicionando-os a essa lista para a próxima iteração, repetindo o processo até as peças por colocar acabarem, obtendo-se um formato aleatório, mas não demasiado caótico.

No final, é ainda utilizado o predicado *remove_invalid_pieces(+Board,-FinalBoard),* de modo a remover peças que possam ter ficadas inseguras.

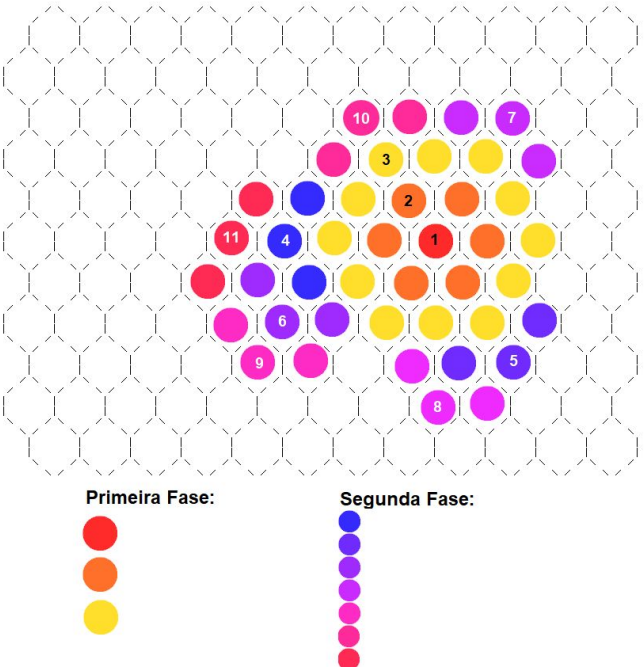

Figura 4 - Exemplo de funcionamento do algoritmo de colocação de peças

3.2. Visualização do Tabuleiro

De modo a visualizar o jogo é utilizado o predicado *display_game(+Game, +Player),* que imprime o tabuleiro, as peças de cada jogador e o jogador da ronda atual.

A função *display_board(+Board)* percorre a lista de listas, havendo, para cada fila de hexágonos, funções para cada linha de impressão. É também guardada a paridade do número da fila a ser impressa, para que sejam feitos os ajustes necessários. De modo a que não ocorram descontinuidades no tabuleiro, foi também desenvolvida uma função de *display* para a primeira e última fila de hexágonos.

Os predicados *disp_p1_pawns(+P1pawns)* e disp_p2_pawns(+P2pawns) recebem as peças capturadas pelo respetivo jogador (uma lista), mostrando-as na consola recorrendo à função disp_pawns(+Pawns), mostrada abaixo.

```
disp_pawns([]).
disp_pawns([Pawn | Pawns]):-
    disp_pawn(Pawn),
    write(' '),
    disp_pawns(Pawns).

disp_pawn(0):- write(' ').
disp_pawn(1):- write('O').
disp_pawn(2):- write('X').
disp_pawn(3):- write('T').
```

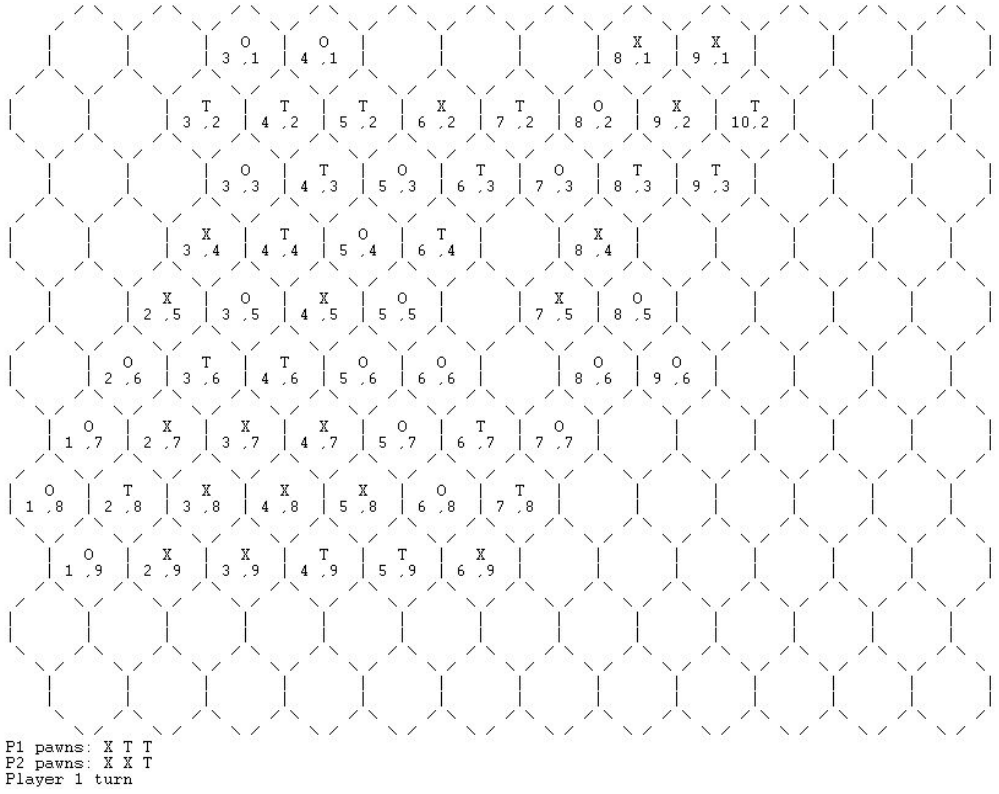

Figura 5 - Exemplo da consola após display_game

Para obter jogadas ou outra informação fornecida pelo utilizador, utiliza-se a função *read*, validando-se sempre os dados recebidos. Caso estes sejam inválidos, é repetido o processo de leitura, imprimindo-se previamente uma mensagem de erro.

Exemplo para leitura de dificuldade do CPU:

```
read_difficulty(0, 0).
read_difficulty(Mode, Difficulty):-
    Mode \= 0,
    repeat,
        write('Choose CPU difficulty:'), nl,
        write('- 0: Easy'), nl,
        write('- 1: Normal'), nl,
        catch(read(Difficulty), _Error, bad_difficulty_format),
        validate_difficulty_format(Difficulty),
    !.

validate_difficulty_format(Difficulty):- integer(Difficulty), Difficulty
>= 0, Difficulty =< 1, !.
validate_difficulty_format(_):- bad_difficulty_format.

bad_difficulty_format:- write('Couldn\'t read difficulty.'), nl, fail.
```

3.3. Lista de Jogadas Válidas

O predicado utilizado para obter a lista de jogadas Válidas é *valid_moves(+Board, -ListOfMoves)*. Como as jogadas possíveis são as mesmas para ambos os jogadores, poise as peças em jogo não pertencem a nenhum jogador, não é necessário o argumento Player. Esta função chama o predicado *get_valid_moves(+Board, +X-Y, -ListOfMoves),* que percorre o tabuleiro com auxílio de *next_cell(+Board, +X-Y, -NextX-NextY)*, adicionando à lista de jogadas válidas as que passam com sucesso o predicado *valid_move(+X-Y, +Board)*.

O predicado *next_cell* foi definido em prolog da seguinte forma:

```
next_cell(_, X-_, 0-0):- X =< 0, !.
next_cell(_, _-Y, 0-0):- Y =< 0, !.
next_cell(Board, X-Y, NextX-Y):-
    get_board_width(Board, Width),
    get_board_height(Board, Height),
    X < Width, Y =< Height, !, % If (X < width && y <= height)
    NextX is X + 1.            %     return (x+1, y)

next_cell(Board, _-Y, 1-NextY):-
    get_board_height(Board, Height),
    Y < Height, !, % Else if (Y < height)
    NextY is Y + 1. %acabaar
next_cell(_, _-_, 0-0).
```

valid_move(+X-Y,+Board) verifica se a remoção da peça na posição X-Y é válida. Inicialmente, é verificado se existe alguma peça nessa posição. Caso exista, remove a peça do tabuleiro e utiliza o predicado *get_coords_around(X-Y, CoordList),* para obter a lista de coordenadas à volta da peça que removida, de forma a verificar se todas as peças vizinhas continuam seguras após a sua remoção, utilizando *are_pawns_safe(+CoordList, +Board)*. No final, é necessário verificar se, ao remover a peça, as peças restantes não se tenham dividido em secções separadas, criando mais do que uma única "ilha" de peças. Para fazer essa verificação, é utilizado o predicado *check_single_section(+Board)* que, utilizando um algoritmo *flood fill*, conclui se existem conjuntos de peças que não estejam ligados a outras.

Abaixo mostram-se excertos de código em PROLOG, contendo os predicados *valid_move* e *flood_fill*.

```
valid_move(X-Y, Board):-
    \+get_pawn_at(X-Y, Board, 0),
    remove_pawn_at(X-Y, Board, NewBoard),
    get_coords_around(X-Y, CoordList),
    are_pawns_safe(CoordList, NewBoard),
    check_single_section(NewBoard).


flood_fill(Board,X-Y,NewBoard):-
    \+get_pawn_at(X-Y, Board,0),
    \+get_pawn_at(X-Y, Board,-1), !,
    replace_pawn_at(X-Y,-1,Board,NewBoard0),

    get_coords_around(X-Y,[A,B,C,D,E,F]),
    flood_fill(NewBoard0, A, NewBoard1),
    flood_fill(NewBoard1, B, NewBoard2),
    flood_fill(NewBoard2, C, NewBoard3),
    flood_fill(NewBoard3, D, NewBoard4),
    flood_fill(NewBoard4, E, NewBoard5),
    flood_fill(NewBoard5, F, NewBoard).
```

3.4. Execução de Jogadas

Para validação e execução de jogadas é usado o predicado *move(+X-Y, +Player, +Game, -NewGame),* que devolve um novo estado de jogo com a jogada executada, caso esta seja válida.

Em primeiro lugar, *move* verifica se a jogada recebida é válida, através do predicado *valid_move(+X-Y,+Board)* explicado na secção anterior. De seguida, é chamado o predicado *get_pawn_at(+X-Y,+Board, -Pawn)*, para obter a peça que se encontra na posição da jogada, e é adicionada à lista de peças capturadas pelo jogador, utilizando o predicado *add_pawn_to_player(+Pawn, +Player, +Pawns, -NewPawns)*.

Finalmente, a peça escolhida é removida do tabuleiro com *remove_pawn_at(+X-Y, +Board, -NewBoard),* que devolve um novo tabuleiro no qual a posição da jogada é substituída por um 0, que representa a posição vazia.

A definição do predicado *move* em PROLOG é:

```
move(X-Y, Player, Board-Pawns, NewBoard-NewPawns):-
    valid_move(X-Y, Board),
    get_pawn_at(X-Y, Board, Pawn),
    add_pawn_to_player(Pawn, Player, Pawns, NewPawns),
    remove_pawn_at(X-Y, Board, NewBoard).
```

3.5. Final do Jogo

Para verificar se o jogo acabou e identificar o vencedor, é utilizado o predicado *game over(+Game,-Winner),* que tem como input o estado do jogo atual e, caso o jogo tenha acabado, retorna o vencedor ou 0 em caso de empate. A condição de final do jogo corresponde a uma das seguintes:

- Um jogador completa a condição de vitória (capturar 5 peças de cada tipo)
- Não existem mais jogadas válidas, forçando o empate.

De forma a averiguar se um jogador é vencedor, utiliza-se o predicado *check_winner(+Pawns),* que retorna com sucesso caso Pawns contenha pelo menos 5 peças de cada tipo. Para verificar se não existem mais jogadas possíveis, verifica-se se a lista devolvida por *valid_moves* é vazia.

A função *check_winner* foi definida em PROLOG da seguinte forma:

```
check_winner(Pawns):-
    count_element(Pawns, 1, Res1), Res1 > 4,
    count_element(Pawns, 2, Res2), Res2 > 4,
    count_element(Pawns, 3, Res3), Res3 > 4.
```

3.6. Avaliação do Tabuleiro

De modo a avaliar o estado de jogo, é usado o predicado *value(+Board, +Player, -Value)*, que recebe como input o tabuleiro e o jogador, devolvendo a avaliação. O critério utilizado para o jogo tem em conta o número de peças capturadas "úteis", ou seja, se um jogador tiver mais de 5 peças de um tipo, o excesso não conta para a avaliação, pois não faz com que jogador esteja mais perto da condição de vitória.

Para obter a avaliação, é utilizado o predicado auxiliar pawn_value(+Pawns, -Value), que, dado um conjunto de peças, retorna a sua avaliação, somando todas as peças úteis, definido em PROLOG como mostrado abaixo:

```
pawn_value(Pawns, Value):-
    count_element(Pawns, 1, Num1s),
    count_element(Pawns, 2, Num2s),
    count_element(Pawns, 3, Num3s),
    Value is min(Num1s, 5) + min(Num2s, 5) + min(Num3s, 5).
```

3.7. Jogada do Computador

De forma a que seja possível jogar contra o computador, é utilizado o predicado *choose_move(+Game,+Player,+Difficulty,-Move)*, que recebe como argumentos o jogo, o jogador atual, a dificuldade e devolve a escolha. O predicado começa por chamar *valid_moves* para obter a lista de movimentos e, de seguida, para as dificuldades 0 ou 1, vai obter uma jogada aleatória da lista de movimentos válidos, ou a melhor jogada no momento, respetivamente.

Para obter a melhor jogada no momento, usa-se o predicado *best_move(+MoveList, +Game, +Player, -BestMove-BestValue)*. Trata-se de um predicado recursivo, que percorre a lista de movimentos, comparando o resultado de cada iteração e guardando o movimento com melhor avaliação, com o auxílio de *move_value(+Move, +Player, +Game, -Value).*

A definição de *best_move* em PROLOG foi feita da seguinte forma:

```
best_move([Move], Game, Player, Move-Value):- move_value(Move, Player, Game,
Value).
best_move([Move | Moves], Game, Player, BestMove-BestValue):-
    best_move(Moves, Game, Player, BestMoveRest-BestValueRest),
    move_value(Move, Player, Game, Value),
    compare_moves(Move-Value, BestMoveRest-BestValueRest, BestMove-BestValue).

compare_moves(Move1-Value1, _Move2-Value2, Move1-Value1):- Value1 >= Value2.
compare_moves(_Move1-Value1, Move2-Value2, Move2-Value2):- Value1 < Value2.
```

4. Conclusões

A realização deste projeto teve como objetivo aprofundar e pôr em prática o conhecimento que adquirimos ao longo das aulas. Trabalhar com PROLOG revelou-se desafiante no início do projeto, devido ao diferente paradigma de linguagem de programação em que se encontra, mas rapidamente nos acostumámos e aprendemos a utilizá-lo, conseguindo concluir todas as funcionalidades que planeámos.

O próximo passo para melhorar a execução do jogo seria adicionar níveis de dificuldade ao CPU, que previssem duas ou mais jogadas à frente.