Área Departamental de Engenharia de Electrónica e Telecomunicações e de Computadores

Trabalho Prático 1

47283 : Ricardo Duarte Cardoso Bernardino (a47283@alunos.isel.pt)

47249 : Miguel Henriques Couto de Almeida (a47249@alunos.isel.pt)

48270 : Daniel Lopes Pina (a48270@alunos.isel.pt)

Relatório para a Unidade Curricular de Inteligência Artificial
da Licenciatura em Engenharia Informática e de Computadores

Professor : Doutor Nuno Miguel da Costa de Sousa Leite

23 – 4 – 2023

Contents

List of Figures

1

HEX

1.1 Descrição do Problema

No trabalho em questão foi proposto o desenvolvimento de um jogo de tabuleiro chamado *Hex*, recorrendo à linguagem de programação puramente Lógica de nome *Prolog* [2].

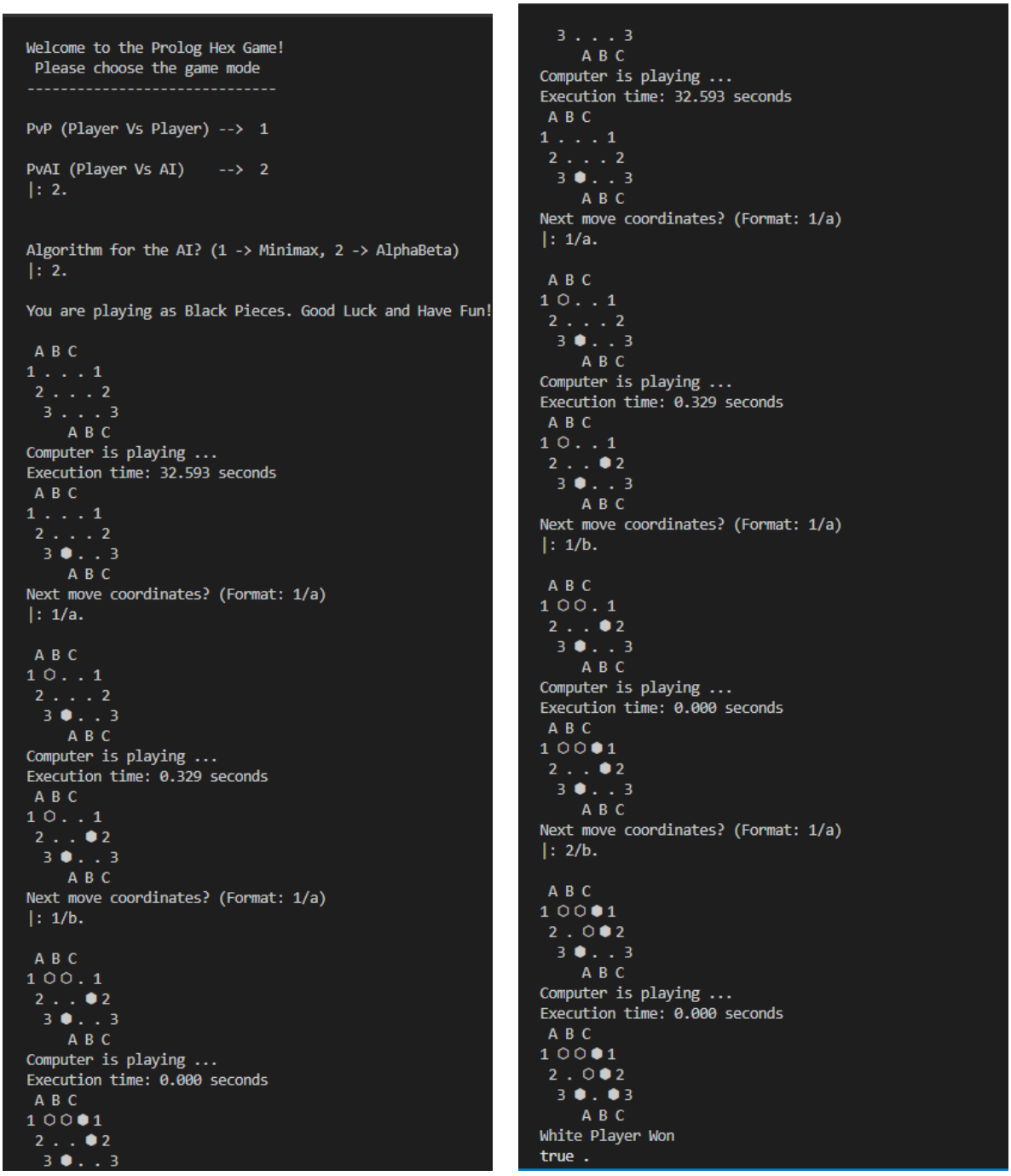
Um dos requisitos exigidos foi que o jogo suportasse a opção de jogar contra outro jogador, como também que suportasse a opção descrita como jogar contra o computador, neste caso recorrendo à implementação de uma Inteligência Artificial e tirando partido das suas decisões enquanto jogadas no tabuleiro.

1.2 Jogo

1.2.1 Descrição do Jogo

Hex é um jogo de tabuleiro jogado a 2 jogadores em uma grade hexagonal, teoricamente de qualquer tamanho desde que o número de colunas coincida com o número de linhas. Os jogadores jogam ambos colocando uma peça da sua cor em uma única posição dentro do tabuleiro do jogo. O objetivo final é formar um caminho entre as suas peças que ligue os lados opostos do tabuleiro com peças da mesma cor, antes, obviamente, que o seu adversário conecte os seus lados primeiro. O primeiro jogador a completar a sua ligação ganha o jogo. Os quatro cantos de cada hexágono das bordas pertencem a ambos os lados adjacentes. No jogo ***Hex*** não existe a possibilidade de o jogo acabar em um empate, logo o resultado final vai ser sempre a vitória para um dos participantes.

1.2.2 Funcionamento do Jogo



(a) Início do Jogo (b) Conclusão do Jogo

Figura 1.1: Jogo em Curso

1.3 Algoritmos

1.3.1 *Depth-First Search*

Este algoritmo foi implementado com a finalidade de averiguar a condição de fim de jogo, para analisar o tabuleiro presente e concluir se este reúne as condições para o jogo ser terminado, ou seja, se um dos jogadores já conseguiu construir um caminho de peças entre ambos os seus lados.

O algoritmo *Depth-First Search (DFS)* é um algoritmo de pesquisa em um grafo que percorre o grafo explorando o máximo possível em profundidade. Isto significa que o algoritmo visita o primeiro vizinho de um vértice antes de visitar o segundo vizinho e assim sucessivamente, aprofundando-se na pesquisa até atingir um vértice sem filhos, momento esse em que volta a trás para visitar os outros vértices.

```
1  dfs_full_board(Board, Player, X/Y, Visited) :-
2      length(Board, Size),
3      \+ member(X/Y, Visited),
4      get_neighbours(Board, Player, X/Y, NeighbourList),
5      member(Pos, NeighbourList),
6      (
7      other_side(Size, Pos, Player)
8      ;
9      dfs_full_board(Board, Player, Pos, [X/Y | Visited])
10     ).
```

A função começa por verificar o tamanho do tabuleiro e se a posição atual (X/Y) não está na lista de posições visitadas, de seguida obtém a lista de posições vizinhas da posição atual que pertencem ao jogador atual (*Player*) através da função auxiliar *get_neighbours*.

A partir da lista de vizinhos, a função verifica se algum deles é uma posição que pertence a uma das extremidades do tabuleiro, representando a última coluna/linha objetivo do jogador em questão *(other_side)*, e se for, a função retorna sem voltar a explorar mais a partir da posição, caso contrário, a função chama-se recursivamente com a nova posição como a posição atual, adicionando a posição anterior (X/Y) à lista de nós visitados.

Esse processo continua até que todas as posições possíveis tenham sido visitadas ou que uma posição na extremidade do tabuleiro seja encontrada.

```
1  get_neighbours(Board, Player, X/Y, Neighbors) :-
2    length(Board, Size),
3    BoardSize is Size + 1,
4    RowBelow is X + 1,
5    RowAbove is X - 1,
6    ColLeft is Y - 1,
7    ColRight is Y + 1,
8    findall(Neighbor, (
9    between(ColLeft, ColRight, Col),
10   between(RowAbove, RowBelow, Row),
11   Row >= 1,
12   Row < BoardSize,
13   Col>= 1,
14   Col < BoardSize,
15   get_elem_board(Board, Row/Col, ResElem),
16   check_same_player(Player, ResElem),
17   + (Row =:= X, Col =:= Y),
18   + (Row =:= X - 1, Col =:= Y - 1),
19   + (Row =:= X + 1, Col =:= Y + 1),
20   Neighbor = Row/Col
21   ), Neighbors).
```

A função usa o predicado *findall* para gerar uma lista de vizinhos válidos a partir das coordenadas calculadas. O predicado *between* é usado para gerar um intervalo de valores para linhas e colunas vizinhas, e, em seguida, são verificados os limites do tabuleiro.

A função verifica se a posição vizinha é ocupada pelo jogador atual, se a posição vizinha não é a posição atual, e se a posição vizinha não está em nenhuma das diagonais em relação à posição atual.

Por fim, a função cria um par (*Row*/*Col*) para cada posição vizinha válida e adiciona-o à lista de vizinhos que é retornada.

1.3.2 *Minimax*

Para colocar o computador a jogar de forma inteligente contra o jogador precisávamos de um algoritmo que nos desse a melhor jogada com base no atual estado do tabuleiro

[1], portanto com base no que tinha sido estudado durante as aulas fomos implementar o algoritmo *Minimax*.

O algoritmo *minimax* é um algoritmo que toma decisões normalmente usado em jogos de dois jogadores, como as damas ou o jogo do galo. O objetivo do algoritmo é determinar qual é melhor jogada a fazer, assumindo que o adversário também está a jogar de forma a obter o melhor resultado.

O algoritmo *minimax* funciona porque explora a árvore do jogo completa, que representa todas as jogadas e resultados possíveis do jogo. Em cada nível da árvore, o algoritmo analisa todas as jogadas possíveis para o jogador atual e avalia os estados de jogo que resultaram, logo depois, o algoritmo escolhe a jogada que leva à pontuação mais alta, assumindo que o adversário vai sempre escolher a jogada que leva à pontuação mais baixa. Esta função é chamada recursivamente até que o algoritmo atinja o fim do jogo, momento esse em que atribui finalmente uma pontuação ao estado final do jogo.

A função de pontuação (*static-val*) é super importante para o funcionamento do algoritmo *minimax*, pois é esta função que atribui o valor do estado do jogo na perspetiva do jogador que está a jogar.

Logo após o algoritmo avaliar todos as jogadas possíveis e os estados de jogo finais que resultam, ele decide jogar no vértice que leva à pontuação mais alta. Esta jogada é assumida como a melhor jogada a ser feita, partindo do princípio que o adversário também vai escolher a jogada que leva à pontuação mais baixa.

```
1  minimax( Pos, BestSucc, Val) :-
2    moves( Pos, PosList), !,     % Legal moves in Pos produce PosList
3    best( PosList, BestSucc, Val)
4    ; % Or
5    staticval( Pos, Val).        % Pos has no successors: evaluate statically
6
7  best( [Pos], Pos, Val) :-
8    minimax( Pos, _, Val), !.
9  best( [Pos1 | PosList], BestPos, BestVal) :-
10   minimax( Pos1, _, Val1),
11   best( PosList, Pos2, Val2),
12   betterof( Pos1, Val1, Pos2, Val2, BestPos, BestVal).
13
14 betterof( Pos0, Val0, Pos1, Val1, Pos0, Val0) :- % Pos0 better than Pos1
15    min_to_move( Pos0),     % MIN to move in Pos0
16    Val0 > Val1, !          % MAX prefers the greater value
```

```
17    ; % Or
18    max_to_move( Pos0),     % MAX to move in Pos0
19    Val0 < Val1, !.         % MIN prefers the lesser value
20  betterof( Pos0, Val0, Pos1, Val1, Pos1, Val1). % Otherwise Pos1 better than Pos0
21
22  moves(Pos, PosList):-
23    Pos = (Board, Player),
24    length(Board, Size),
25    next_player(Player, NextPlayer),
26    \+ terminal_node(Board),
27    findall(CurrElem, (
28      simulated_play(Board, Player, NewBoard),
29      CurrElem = (NewBoard, NextPlayer)
30    ), PosList),
31    \+ PosList = [], !
32    ;
33    fail.
34
35  staticval(Pos, Val) :-
36    Pos = (Board, Player),
37    (is_game_over(Board, 'w'), Val = 1,! % Black Won
38    ;
39    Val = -1). % White Won
40
41  min_to_move(Pos) :-
42    Pos = (Board, Player),
43    Player == 'b'.
44
45  max_to_move(Pos) :-
46    Pos = (Board, Player),
47    Player == 'w'.
```

1.3.3 *Alpha-Beta Pruning*

Decidimos implementar o algoritmo *alpha-beta pruning* pela necessidade de otimizar o algoritmo *minimax* e reduzir o tempo de pesquisa [1], pois em certos cenários durante o jogo o algoritmo estava a calcular jogadas desnecessárias. O *alpha-beta pruning* reduz

o número de vértices que precisam de ser calculados durante a pesquisa pela melhor jogada, porque remove vértices que são irrelevantes para o resultado final do jogo.

O algoritmo *alpha-beta* mantém dois valores: o *alpha* e o *beta*, que são usados para representar a pontuação mínima que o jogador *max* garante e a pontuação máxima que o jogador *min* garante. Inicialmente, *alpha* é definido como menos infinito e *beta* é definido como mais infinito.

Durante a pesquisa, quando o algoritmo avalia um vértice, ele atualiza o valor de *alpha* ou *beta*, dependendo se o jogador que estiver a jogar for o jogador *max* ou o jogador *min*. Se for o jogador *max*, o *alpha* será atualizado para o máximo de *alpha* e para a pontuação do vértice atual. Se o jogador que estiver a jogar for o jogador *min*, o *beta* será atualizado para o mínimo de *beta* e para a pontuação do vértice atual.

Em cada vértice, o algoritmo verifica se o vértice atual pode ser cortado comparando *alpha* e *beta*. Se *alpha* for maior ou igual a *beta*, então o algoritmo corta o resto da subárvore porque tem a certeza que o outro jogador não vai escolher esse caminho, pois iria resultar em uma pontuação pior do que uma pontuação já encontrada em um caminho diferente. Tal coisa só é possível porque o algoritmo pressupõe que o adversário vai sempre escolher a jogada que leva à pontuação mais baixa para o jogador atual.

```
1  alphabeta( Pos, Alpha, Beta, GoodPos, Val) :-
2     moves( Pos, PosList), !,
3     boundedbest( PosList, Alpha, Beta, GoodPos, Val)
4     ; % Or
5     staticval( Pos, Val).      % Static value of Pos
6
7  boundedbest( [Pos | PosList], Alpha, Beta, GoodPos, GoodVal) :-
8     alphabeta( Pos, Alpha, Beta, _, Val),
9     goodenough( PosList, Alpha, Beta, Pos, Val, GoodPos, GoodVal).
10
11 goodenough( [], _, _, Pos, Val, Pos, Val) :- !.   % No other candidate
12 goodenough( _, Alpha, Beta, Pos, Val, Pos, Val) :-
13    min_to_move( Pos), Val > Beta, !   % Maximizer attained upper bound
14    ; % Or
15    max_to_move( Pos), Val < Alpha, !. % Minimizer attained lower bound
16 goodenough( PosList, Alpha, Beta, Pos, Val, GoodPos, GoodVal) :-
17    newbounds( Alpha, Beta, Pos, Val, NewAlpha, NewBeta),     % Refine bounds
18    boundedbest( PosList, NewAlpha, NewBeta, Pos1, Val1),
19    betterof( Pos, Val, Pos1, Val1, GoodPos, GoodVal).
20
```

```
21 newbounds( Alpha, Beta, Pos, Val, Val, Beta) :-
22   min_to_move( Pos), Val > Alpha, !.        % Maximizer increased lower bound
23 newbounds( Alpha, Beta, Pos, Val, Alpha, Val) :-
24   max_to_move( Pos), Val < Beta, !.         % Minimizer decreased upper bound
25 newbounds( Alpha, Beta, _, _, Alpha, Beta). % Otherwise bounds unchanged
26
27 betterof( Pos, Val, Pos1, Val1, Pos, Val) :-  % Pos better than Pos1
28   min_to_move( Pos), Val > Val1, !
29   ; % Or
30   max_to_move( Pos), Val < Val1, !.
31 betterof( _, _, Pos1, Val1, Pos1, Val1). % Otherwise Pos1 better
```

1.4 Conclusão

1.4.1 Resultados

Com a escolha de implementar esta otimização conseguimos reduzir, tendo como exemplo os computadores dos alunos, os tempos de computação da melhor jogada em quase 40%, o que resultou em um ganho significativo de eficiência do algoritmo à medida que fomos escolhendo jogar em um tabuleiro com dimensões progressivamente maiores, porque inicialmente estávamos a testar em um tabuleiro de três colunas por três linhas.

1.4.2 Aprendizagem

Este trabalho prático foi excelente para dar os primeiros passos na linguagem *Prolog* e fazer uso da mesma construir um jogo relativamente simples, contudo podendo tirar partido de algoritmos tais como os que foram usados para também ter uma primeira experiência com uma inteligência artificial.

Referências

[1] Stuart Russel Peter Norvig, *Artificial Intelligence: A Modern Approach, Global Edition 4th ed.* Pearson, 2021.

[2] Ivan Bratko, *Prolog Programming for Artificial Intelligence 4th ed.* Addison-Wesley, 2011.