



UNIVERSIDADE  
DE ÉVORA

# Relatório do 3º trabalho Inteligência Artificial

**Trabalho realizado por:**

- Henrique Rosa, nº 51923
- Diogo Matos, nº 54466

28/06/2024

## 1 - QUATRO EM LINHA

**A)** Após análise da logística do jogo, podemos concluir que é bastante análogo ao jogo do galo, apenas sendo jogado num tabuleiro maior e com 4 peças em linha para ganhar, ao invés de 3. Isto acaba por tornar a contagem das peças mais difícil, e como tal, decidimos usar, como estrutura de dados para a representação do problema, uma lista de 42 (6 x 7) tópicos com três valores: as duas coordenadas da posição a que se refere, e o seu valor. No início do jogo, todos os valores começam a “v” as e vão sendo preenchidas por “b” ou “p”, para brancas ou pretas respectivamente, durante o decorrer do jogo. A definição de estado também contém uma variável que designa o jogador a jogar no estado. Assim, a representação de estados em prolog seria a seguinte:

```
estado_inicial(e( [(1,7,v),(2,7,v),(3,7,v),(4,7,v),(5,7,v),(6,7,v),
(1,6,v),(2,6,v),(3,6,v),(4,6,v),(5,6,v),(6,6,v),
(1,5,v),(2,5,v),(3,5,v),(4,5,v),(5,5,v),(6,5,v),
(1,4,v),(2,4,v),(3,4,v),(4,4,v),(5,4,v),(6,4,v),
(1,3,v),(2,3,v),(3,3,v),(4,3,v),(5,3,v),(6,3,v),
(1,2,v),(2,2,v),(3,2,v),(4,2,v),(5,2,v),(6,2,v),
(1,1,v),(2,1,v),(3,1,v),(4,1,v),(5,1,v),(6,1,v)], b ) ).
```

**B)** Um estado é terminal se existirem 4 peças em sequência e da mesma cor, ou caso o tabuleiro fique cheio. Dado que este jogo tem muitos estados terminais, foi feito um predicado terminal dependente em funções auxiliares.

O predicado **terminal/1** é o seguinte:

```
terminal(e(L,_)):- conta4pecas(L,b) ; conta4pecas(L,p).
terminal(e(L,_)):- \+member( (_,_,v),L).
```

O predicado terminal depende do predicado **conta4pecas/2**, apresentado abaixo:

```
conta4pecas(L,P):- linhas(L,P,1) ;
colunas(L,P,1) ;
tabuleiroX(M), M1 is M+1 , diagonais(L,P,1,-M1);
tabuleiroX(S), S1 is S+1, antidiags(L,P,1,-S1).
```

O predicado acima depende dos predicados **linhas**, **colunas**, **diagonais** e **antidiagonais**. Resumidamente, estes predicados vão verificar as suas secções específicas do tabuleiro (linhas verifica as linhas, colunas as colunas, etc.), contando as peças e sucedendo se nalguma dessas áreas houver 4 peças de igual cor em sequência (ver ficheiro game.pl, linhas 68-109).

**C)** Como o jogo que estamos a definir é análogo ao jogo do galo, também a sua função de utilidade seria análoga, como tal, decidimos implementar uma função de utilidade idêntica à que a professora implementou no jogo do galo, no ficheiro galo.pl. A única diferença seria o valor retornado, para efeitos de, aquando da execução do jogo, os valores dos estados terminais serem sempre maiores do que os valores de estados não terminais, conforme as funções de utilidade e avaliação, respectivamente - Isto para dar prioridade à escolha de um estado terminal (ou seja, ganhar o jogo). Como tal, definimos o predicado `valor/2` como o seguinte:

```
valor(e(L,_),0,P):- \+ member(v,L),!.  
  
valor(E,V,P):-terminal(E),  
    X is P mod 2, (X== 1,V=4;X==0,V= -4).
```

**D)** Implementámos o operador de transição de estados da seguinte forma:

```
op1(e(L,J),(C),e(L1,J1)):- inv(J,J1), subs(v,J,L,L1,1,C).  
inv(p,b).  
inv(b,p).  
  
subs(A,J, [(X,Y,A)|R], [(X,Y,J)|R],C,C).  
subs(A,J, [(X,Y,B)|R], [(X,Y,B)|S],N,C):- M is N+1, subs(A,J,R,S,M,C).
```

Como a nossa representação de estado utiliza coordenadas para definir as posições, fizemos uma pequena alteração no ficheiro minmax.pl para facilitar os testes. Começámos por implementar uma função de conversão entre o número de casa e coordenadas dessa mesma casa (já que a utilização do minmax dado nas aulas retorna o número da casa). As alterações foram as seguintes:

```
%este predicado converte entre numero de casa e coordenadas da casa, no tabuleiro  
%apenas para facilitar testes com o minmax  
casa_pos(e([(X,Y,_)|T],_), (X,Y),F,F).  
casa_pos(e([(X,Y,_)|T],_), (RX,RY),F,I):- I1 is I+1, casa_pos(e(T,_), (RX,RY),F,I1).
```

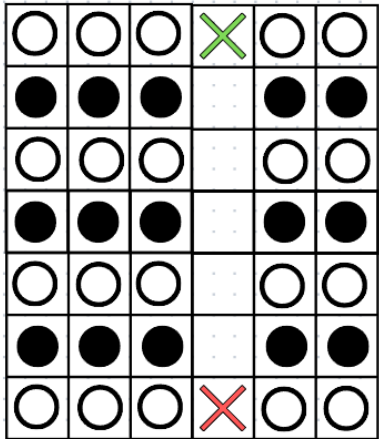
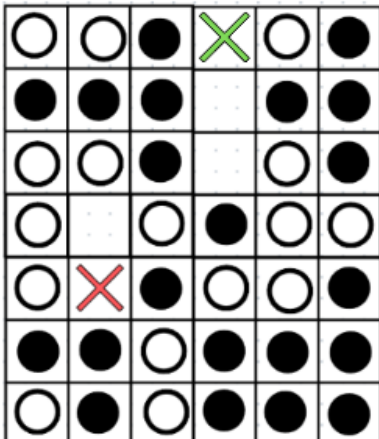
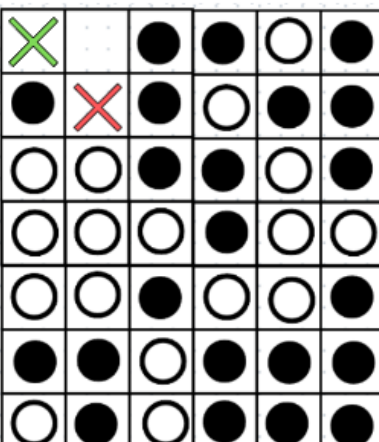
Com tudo isto, podemos utilizar o algoritmo do ficheiro minmax.pl para definir qual a melhor jogada para cada estado. De seguida, começámos a testar a nossa definição com vários estados diferentes.

Apresentamos abaixo três estados, sempre na jogada das peças brancas. Cada exemplo contém a lista que pertence à sua representação em prolog, o primeiro output obtido na chamada do predicado g(game), e um esquema que facilita a visualização do estado em questão, com a melhor jogada no output.

Exemplo	Estado	Out	Esquema
1	<pre>[ (1,7,b), (2,7,b), (3,7,b), (4,7,v), (5,7,b), (6,7,b),   (1,6,p), (2,6,p), (3,6,p), (4,6,v), (5,6,p), (6,6,p),   (1,5,b), (2,5,b), (3,5,b), (4,5,v), (5,5,b), (6,5,b),   (1,4,p), (2,4,p), (3,4,p), (4,4,v), (5,4,p), (6,4,p),   (1,3,b), (2,3,b), (3,3,b), (4,3,v), (5,3,b), (6,3,b),   (1,2,p), (2,2,p), (3,2,p), (4,2,v), (5,2,p), (6,2,p),   (1,1,b), (2,1,b), (3,1,b), (4,1,v), (5,1,b), (6,1,b) ]</pre>	4,1	
2	<pre>[ (1,7,b), (2,7,b), (3,7,p), (4,7,v), (5,7,b), (6,7,p),   (1,6,p), (2,6,p), (3,6,p), (4,6,v), (5,6,p), (6,6,p),   (1,5,b), (2,5,b), (3,5,p), (4,5,v), (5,5,b), (6,5,p),   (1,4,b), (2,4,v), (3,4,b), (4,4,p), (5,4,b), (6,4,b),   (1,3,b), (2,3,v), (3,3,p), (4,3,b), (5,3,b), (6,3,p),   (1,2,p), (2,2,p), (3,2,b), (4,2,p), (5,2,p), (6,2,p),   (1,1,b), (2,1,p), (3,1,b), (4,1,p), (5,1,p), (6,1,p) ]</pre>	2,3	
3	<pre>[ (1,7,v), (2,7,v), (3,7,p), (4,7,p), (5,7,b), (6,7,p),   (1,6,p), (2,6,v), (3,6,p), (4,6,b), (5,6,p), (6,6,p),   (1,5,b), (2,5,b), (3,5,p), (4,5,p), (5,5,b), (6,5,p),   (1,4,b), (2,4,b), (3,4,b), (4,4,p), (5,4,b), (6,4,b),   (1,3,b), (2,3,b), (3,3,p), (4,3,b), (5,3,b), (6,3,p),   (1,2,p), (2,2,p), (3,2,b), (4,2,p), (5,2,p), (6,2,p),   (1,1,b), (2,1,p), (3,1,b), (4,1,p), (5,1,p), (6,1,p) ],</pre>	2,6	

Quanto ao segundo exemplo, não conseguimos entender porque é que o algoritmo considerou (2,3) a melhor jogada, sendo que (4,6) seria objetivamente melhor, pois qualquer jogada que não essa levaria à vitória do oponente.

**E)** Implementamos a pesquisa alfa-beta no ficheiro alfa\_beta.pl. Em vez de valores infinitos para representar majorantes e minorantes dos valores obtidos pelos jogadores, utilizamos 50 e -50, pois o prolog não conseguia utilizar inf e -inf com operações aritméticas necessárias ao funcionamento do código. Abaixo apresentaremos uma tabela com os exemplos já testados, com os tempos de execução dos algoritmos minmax e corte alfa-beta, representados no esquema do exemplo como uma cruz vermelha para o minmax e uma cruz verde para o alfa beta.

Exemplo	Estado	Minmax	Alfa-Beta
1		<p>Tempo: 293.611 seg (4.9 minutos) Estados Visitados: 13699 Output: 4,1</p>	<p>Tempo: 0.599 seg  Estados Visitados: 27  Output: 4,7</p>
2		<p>Tempo: 7.062 seg  Estados Visitados: 325  Output: 2,3</p>	<p>Tempo: 0.599 seg  Estados Visitados: 15  Output: 4,7</p>
3		<p>Tempo: 0.34 seg  Estados Visitados: 15  Output: 2,6</p>	<p>Tempo: 0.035 seg  Estados Visitados: 0  Output: 1,7</p>

**F)** A função de avaliação que definimos tem por base o mesmo tipo de lógica que o predicado terminal, mas os predicados auxiliares que criamos, ao invés de verificarem constantemente se existem 4 peças em linha, conta o número máximo de peças em sequência nas linhas, colunas, diagonais e anti-diagonais num estado, e retorna-o. Como tal, definimos o predicado aval:

```
aval(e(L,P),Val):-  linhasA(L,P,1,VL) ,  
                    colunasA(L,P,1,VC) ,  
                    tabuleiroX(M), M1 is M+1 , diagonaisA(L,P,1,-M1,VD),  
                    tabuleiroX(S), S1 is S+1, antidiagsA(L,P,1,-S1,VAD),!,  
                    H1 is max(VL,VC), H2 is max(H1, VD), Val is max(H2,VAD).
```

O predicado acima depende dos predicados `linhasA`, `colunasA`, `diagonaisA` e `antidiagonaisA`. Como referido acima, estes predicados têm a mesma lógica dos predicados auxiliares do predicado terminal, que falamos na alínea B). A única diferença seria que, ao invés de estar constantemente a verificar se existem 4 peças em sequência, estes predicados vão contar o número de peças máximo. (ver ficheiro `game.pl`, linhas 113-148).

Só nos foi possível implementar o corte em profundidade no algoritmo minimax, e mesmo assim os resultados dados pelo algoritmo deixam um bocado a desejar, tendo tempos similares ao minmax, mas retornando sempre as coordenadas da última casa que ainda não tenha sido instanciada, logo, decidimos incluir na entrega a nossa tentativa no ficheiro `corte.pl`, mas não iremos fazer a análise e comparação de tempo e espaço.

**G) & H)** Não nos foi possível responder às alíneas G) e H) por falta de tempo. Pedimos perdão.

## 2 - NIM

A versão que escolhemos no jogo do NIM consiste numa fila de 9 peças. No seu turno, cada jogador pode retirar uma ou duas peças, sem restrição de posição (não importa se o jogador retira a peça do fim, do meio ou do início da fila). Perde o jogador que retirar a última peça.

**A)** Os estados podem ser representados da seguinte forma: `e(L,J)`, em que `L` é um inteiro que representa o número de elementos na fila, enquanto que `J` contém ou 1 ou 2, e representa o jogador a jogar naquele estado.

**B)** O estado terminal é bastante simples, e consiste em verificar se o número de peças restante é menor que 1. Isto porque o jogo só acaba caso não haja mais peças.

```
terminal(e(A,_)):- A<1.
```

**C)** Como as jogadas entre os dois jogadores são alternadas, os valores da função de utilidade seriam exatamente os mesmos comparativamente ao jogo do quatro em linha, logo, decidimos reutilizar o predicado valor/3 que já tínhamos implementado anteriormente.

```
valor(E,V,P):-terminal(E),  
    X is P mod 2, (X== 1,V=1;X==0,V= -1).
```

**D)** Implementamos o agente inteligente no predicado **playnim/0** da seguinte forma:

```
playnim:-estado_inicial(E),playnim(E).  
  
playnim(e(N,1)):-N<0, nl,write('ganhou o computador').  
playnim(e(N,2)):-N<0, nl,write('ganhou o jogador').  
playnim(e(N,2)):-expor(N),nl, read(Op), N1 is N-Op, playnim(e(N1,1)).  
playnim(e(N,1)):-expor(N),nl, minimax_decidir(e(N,1),Op), (Op==terminou-> Op is 1; Op is Op),  
    write(Op),nl, N1 is N-Op, playnim(e(N1,2)) ;  
    Op=1, write(Op), nl, N1 is N-Op, playnim(e(N1,2)).  
  
expor(0):-write(0),!.  
expor(N):- S is N-1, expor(S), write(0).
```

Basta compilar o ficheiro nim.pl com o ficheiro do algoritmo desejado e jogar, executando a chamada ao predicado **playnim/0**.

**E)** Não nos foi possível responder a esta alínea por falta de tempo. Pedimos perdão.