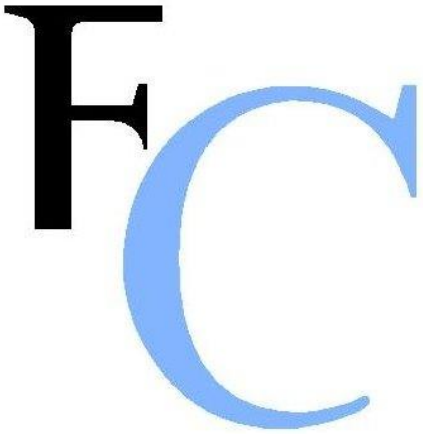


FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Jogo do Galo

INTELIGÊNCIA ARTIFICIAL

201404878 André Carvalho

201405219 Miguel Correia

Conteúdo

1. Introdução

Anteriormente estudamos na unidade curricular de Inteligência Artificial problemas de busca. Este tipo de problemas não envolvem a presença de um oponente, oponente este cuja jogada não é conhecida. Então, nesta parte da matéria vamos introduzir os jogos com oponentes e respetivos algoritmos que os possam solucionar.

Os jogos com oponentes estão diretamente ligados à incerteza, não por falta de informação, mas sim por não se saber qual será a próxima jogada do adversário. As diferenças principais entre os problemas de busca e os jogos com oponentes são:

- Espaço de busca muito grande;
- Tempo para cada jogada.

Os jogos com oponentes mais conhecidos são o Jogo do Galo, o xadrez, o jogo das Damas, o Gamão, entre outros. Um jogo pode-se definir como uma espécie de problema de busca, sendo caraterizado por:

- Estado inicial: configuração inicial do tabuleiro, bem como a indicação de quem deve de jogar;
- Função para gerar os sucessores, função esta que retorna uma lista de pares com as jogadores possíveis e o respetivo estado resultante;
- Teste onde se verifica se o jogo chega ao fim;
- Função utilidade que devolve o valor número do jogo, como por exemplo, “ganhou”, “perdeu”, “empatou”.

Este trabalho está relacionado com um dos exemplos anteriores, mais concretamente, o jogo do Galo, também conhecido como Tic-Tac-Toe. Neste jogo o que fazemos é atribuir um valor em relação à situação de jogo nas folhas das árvores formadas por todas as possíveis jogadas. Após a avaliação do estado das folhas da árvore, é aplicado um dos algoritmos possíveis para a resolução de um jogo com oponentes, o Minimax, que irá permitir a escolha das jogadas com maior hipótese de levar o jogador à vitória, mesmo sabendo que o oponente terá uma ação que pode não ser totalmente prevista.

Outro algoritmo possível para resolver este tipo de problemas é o algoritmo Alfa-Beta, algoritmo este que se trata de uma melhoria do algoritmo Minimax, tornando-o mais eficiente

2. Algoritmo Min-Max

O Min-Max é um algoritmo para a resolução de jogos com oponentes, cujo principal objetivo é a minimização da perda máxima possível, e segue os seguintes passos:

1. Gera a árvore de procura desde o nó inicial até aos nós terminais, sendo que estes podem ser impostos por um limite de profundidade ou mesmo um nó terminal;
2. Aplica a função de utilidade a cada nó terminal para determinar o respetivo valor;
3. Usa a função utilidade dos nós terminais para determinar através de um processo de backup a utilidade dos nós no nível imediatamente acima na árvore de procura;
4. Caso seja MIN a jogar o valor calculado é o mínimo dos nós do nível inferior;
5. Se for MAX a jogar, o valor calculado é o máximo.
6. Continua a usar o processo de backup um nível de cada vez até atingir o nó inicial.
7. Quando chega ao nó inicial, faz a jogada correspondente ao valor determinado para esse nó.

O algoritmo apresenta complexidade temporal de $O(b^m)$, sendo m a profundidade máxima da árvore e b o fator de ramificação da mesma.

Inicialmente é essencialmente pesquisa em profundidade, sendo usada posteriormente uma fila de nós recursivamente.

Para problemas do quotidiano este algoritmo apresenta um custo de tempo geralmente inaceitável, no entanto serve de base a outros métodos mais realistas bem como de suporte à análise matemática de jogos.

O algoritmo, apresenta-se abaixo descrito em psedo-código:

```
1.  function MINIMAX_DECISION(state) returns an action
2.  inputs: state -> estado corrente no jogo
3.      v <- MAX-VALUE(state)
4.      return the action in SUCCESSORS(state) with value v
5.
6.  function MAX-VALUE(state) returns a utility value
7.      if TERMINAL_TEST(state) then return UTILITY(state)
8.          v <- -infinito
9.      for s in SUCCESSORS(state) do
10.         v <- MAX(v, MIN-VALUE(s))
11.     return v
12.
13. function MIN-VALUE(state) returns a utility value
14.     if TERMINAL_TEST(state) then return UTILITY(state)
15.         v <- infinito
16.     for s in SUCCESSORS(state) do
17.         v <- MIN(v, MAX-VALUE(s))
18.     return v
```

3. Algoritmo Alfa-Beta

O Alfa-Beta trata-se de um algoritmo para a resolução de jogos com oponentes, cujo principal objetivo é calcular o Minimax corretamente sem analisar todos os nós da árvore.

Considera-se um dado nó n, em que MAX tem a possibilidade de jogar para esse nó. Caso MAX detete a possibilidade de escolher um nó melhor do que o nó n no nível em que se encontro o nó pai do nó n ou em algum nível acimda desse, então o nó n nunca será alcançado durante o jogo. Como consequência, quando a informação sobre o nó n for suficiente, explorando alguns dos seus descendentes, pode-se eliminar o nó n.

O algoritmo Alfa-beta, tal como o minimax, é do tipo pesquisa em profundidade, pelo que em cada instante é apenas necessário considerar os nós ao longo de um ramo da árvore de pesquisa.

Alfa é o valor da melhor escolha encontrada até àquele momento, ao longo de um ramo, para MAX. Por sua vez, Beta é o valor da melhor escolha encontrada até àquele momento, ao longo de um ramo, para MIN.

O Alfa-beta atualiza o valor de Alfa e Beta ao longa da procura e corta uma sub-árvore após a chamada recursiva mal se encontre os piores valores de Alfa e Beta.

O algoritmo, apresenta-se abaixo descrito em psedo-código:

```
1.  function ALPHA-BETA-SEARCH(state) returns an action
2.      v <- MAX-VALUE(state,-inf,+inf)
3.      return the action in SUCCESSORS(state) with value v
4.
5.  function MAX-VALUE(state,alfa,beta) returns a utility value
6.      if TERMINAL_TEST(state) then return UTILITY(state)
7.          v <- -infinito
8.      for s in SUCCESSORS(state) do
9.          v <- MAX(v, MIN-VALUE(s,alfa,beta))
10.     if (v >= beta ) then return v % momento da poda
11.         alfa <- MAX(alfa,v)
12.     return v
13.
14. function MIN-VALUE(state,alfa,beta) returns a utility value
15.     if TERMINAL_TEST(state) then return UTILITY(state)
16.         v <- +infinito
17.     for s in SUCCESSORS(state) do
18.         v <- MIN(v, MAX-VALUE(s,alfa,beta))
19.     if (v <= alfa ) then return v % momento da poda
20.     beta <- MIN(beta,v)
21. return v
```

4. Jogo do Galo

O jogo do galo é dos jogos mais populares e conhecidos. Trata-se de um jogo extramente simples, que se carateriza por:

- O tabuleiro é uma matriz de 3x3;
- Dois jogadores escolhem uma marcação para jogar, sendo normalmente utilizado um círculo (O) e um X;
- Os jogadores vão alternando a sua vez de jogar, jogando uma marcação de cada vez num espaço que se encontre vazio;
- O objetivo é conseguir 3 peças iguais em linha, quer seja na horizontal, vertical ou diagonal e, ao mesmo tempo, quando possível, impedir o adversário de ganhar na próxima jogada.
- Um jogador ganha o jogo quando alcança o objetivo.

O jogador deve jogar tendo em conta as seguintes prioridades:

- Ganhar, completando a linha;
- Bloquear para impedir o adversário de completar a linha e como tal ganhar o jogo;
- Bloquear jogadas que proporcionem ao adversário mais do que uma posição de vantagem;
- Jogar no centro;
- Jogar no canto oposto ao do adversário;
- Jogar no canto vazio;
- Jogar no lado vazio.

5. Min-Max e Alfa-Beta aplicados ao Jogo do Galo

Neste trabalho optamos por utilizar a linguagem C++, pois estamos familiarizados com a mesma desde o último trabalho. Além disto, o C++ assim como a maior parte das linguagens, contem muitas bibliotecas e APIs que nos ajudam bastante, como por exemplo <chrono>, apesar de ser de uma versão mais recente do C++, mais propriamente a versão 11, permite-nos calcular o tempo de execução de uma parte do código, para podermos comprar os dois algoritmos.

Entre os dois algoritmos já referidos e explicados, a nível de código não são muito diferentes entre si, o Alfa-Beta é uma otimização do Minimax. Estes dois algoritmos, podem ser implementados de 2 formas cada um, das quais uma delas os algoritmos percorrem a árvore toda (até aos casos de vitoria, derrota ou empate) ou podemos limitar a profundidade. Como já mostramos, cada um dos algoritmos depende da função utilidade que será diferente entre estas duas formas.

Contudo optamos, por dar a opção de escolha ao utilizador entre estes algoritmos: Minimax, Minimax com limite de profundidade, Alfa-Beta e Alfa-Beta com limite de profundidade.

5.1 Min-Max e Alfa-Beta sem limitação de profundidade

Quando o jogador escolhe qualquer um dos algoritmos normal (sem ser limitada), ou seja, percorre a árvore toda, a função utilidade é a seguinte:

```
1.  int utility(State, Level) // State é estado atual e Level é o nível de profundidade
2.  {
3.    if(State == PC) // Se for um estado terminal e o PC ganha
4.      return 15 - Level;
5.    else if(State == Player) // Se for um estado terminal e o utilizador ganha
6.      return -(15 - Level);
7.    else if(State == Draw) // Se for um estado terminal e é empate
8.      return 0;
9.    return -979; // Jogo por acabar, -979 pois é impossível de atingir e para haver uma
    grande distinção dos outros casos
10. }
```

Esta função utilidade, recebe a matriz em causa e o nível de profundidade da árvore, se for um estado terminal (empate, vitoria ou derrota) do computador e retorna 0 para empate, um número positivo para vitoria e um número negativo para derrota. Se repararmos o tamanho máximo da árvore será 9, caso seja o computador a jogar, e no caso de vitória, retornar 15 menos o nível da árvore irá reduzir o número de casos de empate, melhorando a pesquisa. O mesmo acontece, em caso de derrota e retorna o negativo de 15 menos o nível da árvore.

O número 15, na verdade poderia ser outro número maior do que 9, para que não exista a hipótese de haver números menores ou iguais a 0, caso contrario poderia retornar números correspondentes a empate e vitoria. Escolhemos 15 para que haja uma grande distinção entre os valores.

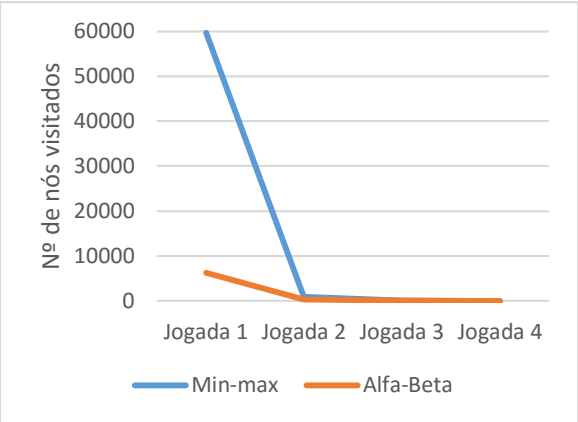
Optamos por alterar os valores de vitória e derrota, pelo facto de que à medida que descemos na árvore, os valores de derrota são mais altos e os de vitória mais baixos, isto para que ao realizar o mínimo, opte pela melhor jogada do utilizador, que é a que está mais perto da raiz da árvore e que tem o valor mais baixo, e ao realizar o máximo, opte pela melhor jogada que pode fazer, que é a que está mais perto da raiz da árvore e que tem o valor mais alto. Isto de ter em consideração a melhor jogada de ambos (computador e jogador), torna o nosso algoritmo mais eficiente, pois reduz os casos de empate.

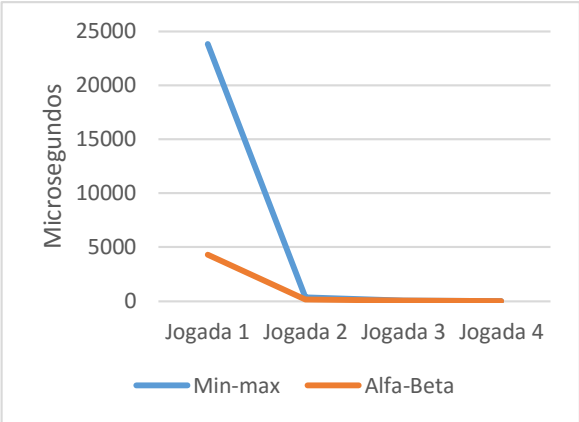
Neste caso, sem limitação em profundidade, apesar de o utilizador não notar diferença no tempo de execução, porque o jogo é basicamente um tabuleiro 3x3 (relativamente pequeno), se aumentássemos o tamanho do mesmo, o utilizador já notava uma grande diferença na resposta por parte do computador. Contudo existe uma grande diferença, no qual, tiramos capturas de ecrã para comprovar a diferença, Figura 1 o caso do Min-Max e a Figura 2 o caso do Alfa-Beta.

Figura 1, Min-Max

Figura 2, Alfa-Beta

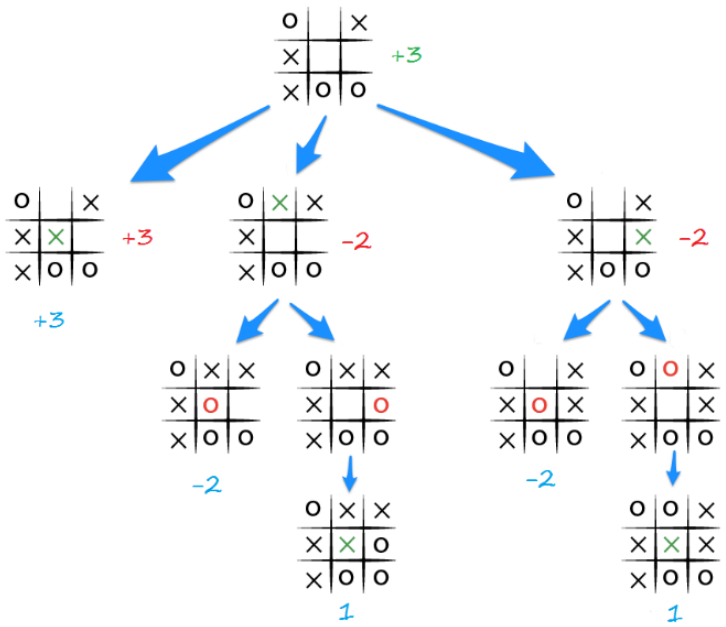
Ambos os algoritmos, funcionam em menos de 1 segundo, pelo que o utilizador não nota a diferença, como já referido, mas por exemplo no primeiro caso, o Min-Max visita 59 704 nós em 0,02 segundos enquanto o Alfa-Beta percorre 6 304 nós em 0,004 segundos.





Descrevendo os gráficos, que correspondem às jogadas do exemplo acima mostrado, o da direita traça duas curvas, relacionando o número de nós visitados por cada um dos algoritmos, com o das jogadas correspondentes, já o da esquerda relaciona as duas curvas com o tempo, em microssegundos, que os algoritmos demoram a executar cada jogada.

Com uma analise dos gráficos, observamos uma grande diferença entre os dois algoritmos, principalmente na primeira jogada, pois é onde o tabuleiro está menos preenchido, havendo mais jogadas possíveis, o que significa que o algoritmo percorre uma arvore de busca de elevadas dimensões. Contudo, nota-se uma grande diferença nos dois algoritmos, sendo as provas bem visíveis, provando assim que o Alfa-Beta otimiza bastante o Min-Max.



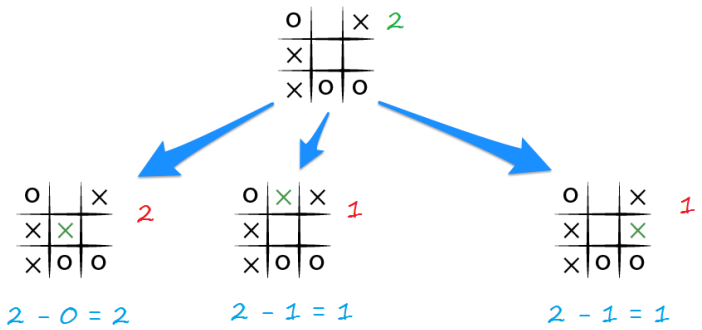
5.2 Min-Max e Alfa-Beta com limitação de profundidade

Quando o jogador escolhe qualquer um dos algoritmos com limite de profundidade, a função utilidade é a seguinte:

```
1.  int utilityLimit(State) // State é estado atual
2.  {
3.    int count = 0;
4.    for(int i = 0; i < 8; i++) // Percorre as linhas, colunas e diagonais possíveis de
    vitória/derrota 8 = 3linhas + 3colunas + 2diagonais
5.      {
6.        if(State[i] != empty && State[i] == possible) // Se a linha/coluna/diagonal for
    diferente de vazia e se não tiver os dois oponentes
7.        {
8.          if(State[i] == PC) // Se o PC pode ganhar nessa linha/coluna/diagonal
9.            count ++;
10.         else // Se o utilizador pode ganhar nessa linha/coluna/diagonal
11.           count --;
12.       }
13.     }
14.   return count;
15. }
```

Esta função utilidade, recebe a matriz em causa, onde percorre as linhas, colunas e diagonais do tabuleiro e verifica quais as que o computador pode preencher e ganhar, incrementando 1 (considerando caso de vitória), no contador iniciado no início da função, e verifica quais as que são possíveis para o utilizador ganhar, decrementando 1 (considerando caso de derrota). Contudo, numa linha, coluna ou diagonal vazia, quer o utilizador quer o computador pode ganhar, ou seja, no código é desnecessário incrementar e decrementar logo de seguida no mesmo caso, pois simplifica e torna mais eficiente a função. Por fim, retorna o valor final do contador, um rácio entre as linhas, colunas e diagonais possíveis para o computador ganhar e para o utilizador ganhar.

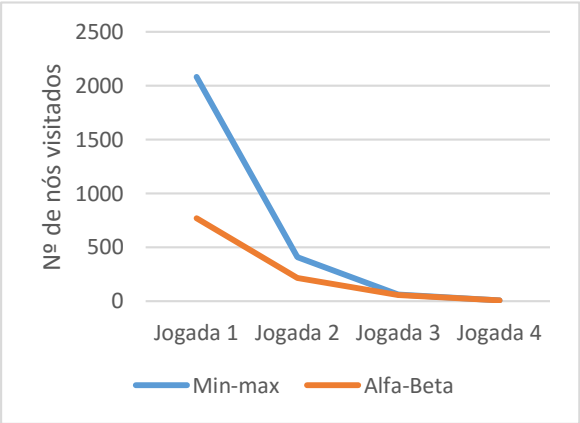
Nestes dois algoritmos, optamos por o limite de profundidade de 3 níveis, pois é o ideal para o computador poder gerir o jogo e não aprufundar muito a árvore, porém com este limite não é garantido que seja impossível que o utilizador ganhe ao computador.

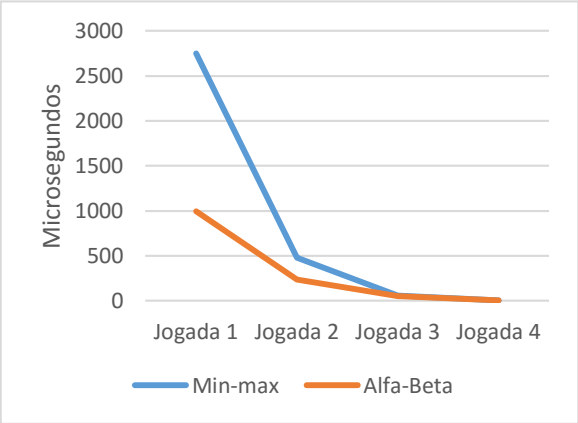


Neste caso, com limite de profundidade, apesar de o utilizador não notar diferença no tempo de execução dos algoritmos, como já explicado anteriormente, o computador responde mais rapido, pois não percorre a árvore até aos limites de vitótia, derrota ou empate. Contudo, esta diferença está comprovada nas seguintes capturas de ecrã, Figura 1 o caso do Min-Max e a Figura 2 o caso do Alfa-Beta.

Figura 3, Min-Max

Figura 4, Alfa-Beta





Ambos os algoritmos, funcionam em menos de 1 segundo, pelo que o utilizador não repara na diferença, como já referido, mas no entanto, no primeiro caso, o Min-Max visita 2 080 nós em 0,003 segundos enquanto o Alfa-Beta percorre 268 nós em 0,00009 segundos.

Após a analise dos gráficos, concluimos mais uma vez, que o Alfa-Beta é mais rápido e mais eficiente que o Min-Max.

5.3 Código e estruturas utilizadas

Inicialmente neste trabalho, guardávamos o tabuleiro numa matriz 3x3 e a árvore era toda percorrida e guardada numa stack. Contudo, era absurdo e desnecessário, e alteramos a estrutura do código, passamos a guardar o tabuleiro num array de tamanho 9, que nos simplificou bastante o código, gerávamos a primeira descendência chamávamos o Min-Max ou Alfa-Beta, recursivamente e guardávamos stack a primeira descendência, bem como o respetivo valor. Por fim, o programa final não guarda a descendência, apenas atualiza, caso seja favorável ao computador, e em caso de empate, vê qual dos descendentes tem um rácio maior entre vitórias e derrotas, desempatando assim.

Contudo, apenas utilizamos uma estrutura, que contem um array de char para guardar o tabuleiro, o respetivo valor que retorna do Min-Max ou Alfa-Beta, o rácio entre vitórias e derrotas (+1 e -1 respetivamente), e claro variáveis int, boolean, char e double.

6. Comentários Finais e Conclusões

Após a realização deste trabalho, onde pusemos em prática os conhecimentos teóricos dos algoritmos Minimax e Alfa-Beta, chegamos às seguintes conclusões:

- Comparando os dois algoritmos, o Minimax é aquele que expande um maior número de nós, especialmente na primeira jogada, onde a diferença entre o número de nós expandidos pelo Minimax e pelo Alfa-Beta é muito grande;
- Conclui-se então que o Minimax não é o mais eficiente;
- Em termos de tempo de resposta, o algoritmo Alfa-Beta é mais rápido, especialmente na primeira jogada, onde a diferença entre o tempo de resposta entre os algoritmos é muito elevada.
- Com base nos dados analisados anteriormente, concluimos então que o melhor algoritmo para o Jogo do Galo é o Alfa-Beta.

7. Bibliografia

- http://www.eurekit.ipb.pt/pdfs/5.pdf
- https://web.fe.up.pt/~eol/IA/IA0708/APONTAMENTOS/IA-MiniMaxeAlfa-Beta.pdf
- https://www.dcc.fc.up.pt/~ines/aulas/1516/SI/jogos.pdf