UNIVERSIDADE FEDERAL DE MATO GROSSO
CAMPUS DE VÁRZEA GRANDE
INSTITUTO DE ENGENHARIA

Inteligência Artificial – 2017/2
Prof. Raoni F. S. Teixeira

Atividade Prática 2: Busca Competitiva
Discente: Ricardo Gonçalves de Aguiar

1 Introdução

Esta atividade foi desenvolvida na disciplina de inteligência artificial. O principal objetivo da atividade foi implementar um algoritmo de busca competitiva para o jogo Lig-4 no MATLAB.

A resolução do jogo Lig 4 pode ser vista como encontrar o melhor caminho em uma árvore de decisão onde cada nó é uma posição. Em cada nó, o jogador deve escolher um movimento levando a uma das possíveis próximas posições.

2 Resumo

Para implementar o agente para o jogo Lig 4 foi utilizado o algoritmo Minimax com poda alfa-beta, bem como sua função heurística. Também apresenta resultados referentes a alguns testes utilizando o agente construído, com objetivo de testar sua eficiência.

O minimax implementado como uma busca em profundidade tem que avaliar todos os filhos antes de poder avaliar um nó, logo, para avaliar a raiz tem que avaliar a árvore inteira. Isto pode ser extremamente custoso, principalmente se o fator de ramificação for muito alto. O método utilizado para não examinar todos os nós foi a poda alfa-beta.

A poda alfa-beta inicialmente, a árvore de jogo é percorrida em ordem de profundidade. A cada nó que não seja folha, é armazenado um valor, sendo:

- ☞ α **MAX**, é o máximo valor encontrado até então nos descendentes dos nós MAX;
- ☞ β **MIN**, é o mínimo valor encontrado até então nos descendentes dos nós MIN.

Se é nó MAX: se valor-alfa-de(nó) $\geq$ valor-beta-ancestral;
ENTÃO poda os demais ramos do nó MAX(poda beta).

Se é nó MIN: se valor-beta-de(nó) $\leq$ valor-alfa-ancestral;
ENTÃO poda os demais ramos do nó MIN(poda alfa).

Utilizou-se como base para implementação do Minimax com poda alfa-beta o seguinte pseudocódigo.

```
function alphabeta(node, depth, α, β, maximizingPlayer)
     if depth = 0 or node is a terminal node
         return the heuristic value of node
     if maximizingPlayer
         v := -∞
         for each child of node
             v := max(v, alphabeta(child, depth - 1, α, β, FALSE))
             α := max(α, v)
             if β ≤ α
                 break (* β cut-off *)
         return v
     else
         v := +∞
         for each child of node
             v := min(v, alphabeta(child, depth - 1, α, β, TRUE))
             β := min(β, v)
             if β ≤ α
                 break (* α cut-off *)
         return v
```

Figura 1

No pseudocódigo acima temos os seguintes parâmetros: **node** é o nó da árvore de estado, **depth** a profundidade máxima de descida na árvore e **maximizingPlayer** o jogador que efetuará a jogada na rodada.

Figura 1: Russell, Stuart J.; Norvig, Peter (2003), Artificial Intelligence: A Modern Approach (2nd ed.).

A chamada inicial para o pseudocódigo acima:

$$\boxed{\textbf{alphabeta(origin, depth, -}\infty\textbf{, +}\infty\textbf{, TRUE)}}$$

O algoritmo inicia como **maximizingPlayer** TRUE, pois a primeira jogada é sempre de maximização. Alfa e Beta são iniciado como tal, pois para que a primeira jogada do nó inicial(**origin**) sejam definidos alfa e beta adequados, armazenando-os em cada nó interno da árvore de busca.

A função heurística utilizada recebe um estado '**n**' e o jogador que estará realizando a jogada. Assim, a função heurística é denotada por:

$$h(n, p) = \sigma_v(n, p_1) - \sigma_v(n, p_2) \tag{1}$$

Onde $'p_1'$ é jogador, $'p_2'$ o adversário e 'n' o nó estado.
A função $\sigma_v(n, p)$ pondera as streaks feita pelo jogador 'p' no estado 'n'. Assim, a função é denotada por:

$$\sigma_v(n, p) = scalarproduct(v, findstreak(n, p, i)) \tag{2}$$

Sendo:

- **v** vetor de pesos para as streaks 1,2,3 e 4, respectivamente;
- **findstreak(n,p,i)** quantidade de streaks, sendo i=1,2,3 e 4.

O seguinte pseudocódigo representa a implementação da função heurística.

Algorithm 1 Função heurística `(n,p1,p2)`

$v \leftarrow [v_1, v_2, v_3, v_4]$ ▷ vetor pesos
for i:1 **do** 4
  $f_1[i] \leftarrow findstreak(n, p_1, i)$ ▷ vetor com total de streaks
  $f_2[i] \leftarrow findstreak(n, p_2, i)$
end for
$s_1 \leftarrow scalarproduct(v, f_1)$ ▷ produto escalar entre os vetores
$s_2 \leftarrow scalarproduct(v, f_2)$
$return(s_1 - s_2)$

3 Resultados e Discussões

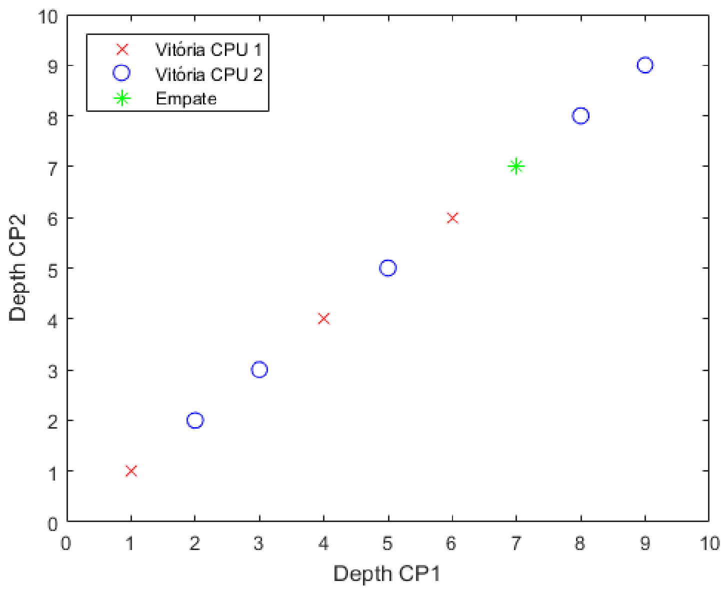
Para analisar o desempenho do agente implementado nesse trabalho, ele é colocado em confronto com o agente disponibilizado para realização dos testes. Assim, os agentes são denotados por:

- **CPU 1:** Agente disponibilizado para os testes;
- **CPU 2:** Agente implementado nesse trabalho.

A CPU 1 utiliza a função heurística implementado nesse trabalho. Já a CPU 2 utiliza a função heurística disponibilizada pelo professor.

Antes de realizarmos os testes, precisamos definir um vetor peso $v = (v_1, v_2, v_3, v_4)$, sendo $v_1 < v_2 < v_3 < v_4$. Para os testes foi escolhido o seguinte vetor $v = (1, 10, 100, 1000)$, sendo ele padrão para todos os testes realizados.

O teste que iremos realizar propõem analisar o desempenho da CPU 2 em diferentes níveis de profundidade. Assim, iremos colocar a CPU 2 em confronto direito com a CPU 1, a fim de demonstrar a eficiência do agente.



Os resultados apresentados no gráfico acima são satisfatório, pois para fazer uma boa função heurística é levado em conta a quantidade de característica do problema que está sendo analisado. É interessante ressaltar que na prática nem sempre uma boa heurística implica em bons resultados, pois para resultar sempre em bons resultados, há necessidade de que o agente sempre avalie qual será o próximo movimento baseado na melhor previsão dada pela heurística.

Quando é a vez da CPU 1, ela deseja escolher o melhor movimento possível que maximizará sua pontuação. Mas, em seguida, a CPU 2 tentará maximizar sua pontuação, minimizando assim o seu.

Nesse caso temos que levar em consideração que estamos lidando com duas heurísticas distintas, sendo a do agente(CP1) diferente da CPU 2. Logo, como as heurísticas são distintas, ambas levam em conta características diferente do problema. A CPU 1 obteve resultados melhores em pequenas profundidades. Logo, a CPU 2 obteve melhores resultados em grandes profundidas. Assim, um bom agente para Minimax depende de quantos estados na árvore é possível analisar, ou seja, seu nível de profundidade.

Lig-4 pode ser vencido na maior parte do tempo. Se você é um especialista, quase sempre vai ganhar, a não ser que seu oponente também jogue otimamente também, caso em que o jogo deverá ser decidido a favor do jogador que começou. Podemos assumir que estamos diante de um jogador que não admite erros, assim há apenas uma forma de sair vencedor do jogo, e a regra é bastante simples: é necessário jogar a primeira peça na coluna central. Para que a CPU 2 inicie jogando na coluna central foram realizados alguns ajustes na função heurística.

Algorithm 2 Função heurística 2 `(n,p1,p2)`

$v \leftarrow [v_1, v_2, v_3, v_4]$ ▷ vetor pesos
for i:1 **do** 4
    $f_1[i] \leftarrow findstreak(n, p_1, i)$ ▷ vetor com total de streaks
    $f_2[i] \leftarrow findstreak(n, p_2, i)$
end for
$s_1 \leftarrow scalarproduct(v, f_1)$ ▷ produto escalar entre os vetores
$s_2 \leftarrow scalarproduct(v, f_2)$
$cp_1 \leftarrow columnplayer(n, p1)$
$cp_2 \leftarrow columnplayer(n, p2)$
$return((s_1 - s_2) + (cp_1 - cp_2))$

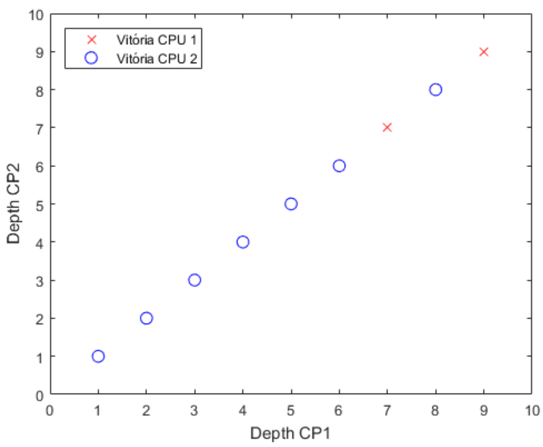
Na função heurística acima, foi acrescentado a função columnplayer, que consiste em ponderar as colunas do tabuleiro. Assim, a partir da nova heurística a CPU 2 sempre inicializará jogando na coluna central do tabuleiro. O pseudocódigo abaixo representa a implementação da função columnplayer.

Algorithm 3 Columnplayer `(n,p)`

```
peso ← [c1, c2, c3, c4, c5, c6, c7]        ▷ vetor pesos p/ cada coluna
col ← find(n == p)        ▷ retorna a posição(coluna) que o player jogou
for i:1 do 7
    s1[i] ← sum(col == i)        ▷ Soma a qt de jogadas do jogador 'p' na coluna 'i'
end for
sp ← dot(s1, peso)        ▷ produto escalar entre os vetores
return(sp)
```

Antes de realizarmos os testes para nova função heurística, precisamos definir um vetor $peso = (p_1, p_2, p_3, p_4, p_5, p_6, p_7)$ da função Columnplayer, sendo $p_1 < p_2 < p_3 < p_4 > p_5 > p_6 > p_7$. Para os testes foi escolhido o seguinte vetor $peso = (-2, -1, 0, 1, 0, -1, -2)$, sendo ele padrão para todos os testes realizados. O vetor peso foi escolhido com base nos testes realizados em Connect four perfect solver[1].

Novamente, iremos analisar o desempenho da CPU 2 em diferentes níveis de profundidade, mas agora utilizaremos a função heurística 2 descrita acima. Assim, iremos colocar a CPU 2 em confronto direito com a CPU 1, a fim de demonstrar a eficiência do agente novamente. Observe o gráfico abaixo.



[1] `http://connect4.gamesolver.org/?pos=`

Como era de se esperar, os resultados apresentados no gráfico acima são satisfatórios. Logo, a CPU 2 obteve uma vantagem ligeiramente maior.