UNIVERSIDADE FEDERAL DE MATO GROSSO

CURSO DE ENGENHARIA DE COMPUTAÇÃO

Bruno Fernandes de Lima Santos

Relatório de atividade prática 2: Jogo Lig4 com algoritmo Minimax

Cuiabá, 2017

Bruno Fernandes de Lima Santos

Relatório de atividade prática 2: Jogo Lig4 com algoritmo Minimax

Relatório técnico referente à atividade prática de número 2, da disciplina de Inteligência Artificial, no curso de Engenharia de Computação, na Universidade Federal do Mato Grosso.

Prof. Dr. Raoni F. S. Teixeira

Cuiabá, 2017

RESUMO

O documento apresenta uma descrição acerca da elaboração de um agente racional para o jogo Lig 4, implementado utilizando o algoritmo Minimax com poda alfa-beta e corte por profundidade, bem como de sua função heurística. Também apresenta resultados referentes a alguns experimentos utilizando o agente construído, com objetivo de testar sua eficácia.

Palavras-chave: Minimax. Alfa-beta. Agente. Heurística. CPU.

SUMÁRIO

1 INTRODUÇÃO

Nos mais diversos jogos eletrônicos há a implementação de jogadores virtuais capazes de simular jogadores reais. É comum, por exemplo, em jogos eletrônicos de xadrez, ter-se uma função onde o usuário do programa pode jogar "contra a máquina". Este recurso também está presente em grande parte dos jogos de videogames onde tem-se uma disputa entre dois jogadores.

Este relatório apresenta uma descrição detalhada sobre construção e execução de um agente racional capaz de jogar o jogo Lig4[1]. O agente foi programado utilizando o algoritmo Minimax, utilizado em problemas de busca competitiva. Outras técnicas foram utilizadas para "polir" o algoritmo do Minimax, devido ao seu elevado tempo de execução: a poda alfa-beta (alpha-beta pruning) e a busca em corte (cutoff search). Estas técnicas são vitais para utilização do algoritmo. Isto porque o Minimax possui performance exponencial no pior caso: $O(b^n)$. Para o caso do jogo Lig4, tem-se a complexidade $O(7^n)$.

Junto à busca em corte (de profundidade), é necessária a implementação de uma função de avaliação capaz de avaliar um estado qualquer (terminal ou não) da árvore do Minimax e atribuí-lo uma nota. Esta nota é uma estimativa do quanto aquele estado está próximo da vitória, sob a vista de um determinado jogador. Este tipo de função é chamada de função heurística, e sua construção também será tratada neste documento.

Todo o programa construído nesta atividade, incluindo os algoritmos que implementam o agente e o próprio Minimax, foi programado na linguagem Octave/MATLAB.

1 Para informações referentes ao jogo Lig4, consulte: https://pt.wikipedia.org/wiki/Lig_4. Acesso em: 01 de mar. 2017.

2 CONSTRUÇÃO DO AGENTE

Neste capítulo serão descritos os algoritmos / funções que implementam o Minimax alfa-beta e a função de avaliação utilizada na busca em corte.

2.1 A função heurística

A função heurística é simples, podendo ser descrita matematicamente em vez de exibição de seu algoritmo. Denotada por $h : (B, p) \mapsto \mathbb{R}$, a função recebe um tabuleiro B e um jogador p, e atribui uma nota àquele estado, da perspectiva de p.

$$h(B, p) = \sigma_{\mathrm{w}}(B, p) - \sigma_{\mathrm{w}}(B, p') \tag{2.1}$$

Onde p' é o jogador adversário de p. Já a função $\sigma_{\mathrm{w}} : (B, p) \mapsto \mathbb{R}$ calcula a soma ponderada de streaks feita pelo jogador p no tabuleiro B. É definida como:

$$\sigma_{\mathrm{w}}(B, p) = \mathbf{w} \cdot \mathbf{f}(B, p) \tag{2.2}$$

A função $\mathbf{f}(B, p) : (B, p) \mapsto \mathbb{N}^4$ devolve um vetor contendo, em cada uma de suas posições, a soma de streaks de tamanhos, respectivamente, 1, 2, 3 e 4, realizados pelo jogador p no tabuleiro B. O vetor $\mathbf{w} = (w_1, w_2, w_3, w_4) \in \mathbb{R}^4$ contém os "pesos" atribuídos aos streaks de tamanho 1, 2, 3 e 4. Obviamente, seus valores devem ser tais que $w_1 < w_2 < w_3 < w_4$. Para este problema foi definido $\mathbf{w} = (0, 1, 10, 100)$ (o experimento em 3.1 consiste na escolha do vetor).

2.2 Implementação do Minimax

Utilizou-se como base para implementação do Minimax com poda alfa-beta e busca em corte em MATLAB o seguinte pseudocódigo.

Algoritmo 1: Minimax alfa-beta com corte de profundidade.

```
function MINIMAX(B, p, d, α, β, is_max)
    if TERMINAL_TEST(B)
        return EVAL(B, p) (* função heurística *)
    if is_max
        v ← −∞
        for each child b of B
            v ← MAX(v, MINIMAX(b, p, d − 1, α, β, false))
            α ← MAX(β, v)
            if β ⩽ α return v (* poda alfa-beta *)
    else
        v ← +∞
        for each child b of B
            v ← MIN(v, MINIMAX(b, p, d − 1, α, β, true))
            β ← MIN(β, v)
            if β ⩽ α return v (* poda alfa-beta *)
    return v
```

Acima temos os seguintes parâmetros da função: B é uma configuração do tabuleiro, que pode ser entendida como um nó da árvore de estados, p é o jogador ativo na rodada e d é a profundidade máxima de descida na árvore (para busca em corte), decrementado a cada recursão. O parâmetro *is_max* é utilizado para indicar se uma determinada jogada (execução do algoritmo) é de maximização ou de minimização. Note que a função EVAL(B, p) no algoritmo é a heurística definida na seção 2.1.

A chamada para o algoritmo acima, dados um tabuleiro inicial B (inicialmente zerado), um jogador inicial p e uma profundidade máxima d, consiste na seguinte linha de execução.

Algoritmo 2: Primeira chamada para o Minimax.

MINIMAX(B, p, d, $-\infty$, $+\infty$, true)

O parâmetro *is_max* inicia-se com verdadeiro pois a primeira jogada é sempre de maximização. Alfa e beta são iniciados como tal para que na primeira jogada (válida) subsequente do nó inicial B sejam configurados alfa e beta adequados.

3 EXPERIMENTOS

Neste capítulo serão listados e descritos os experimentos realizados com o programa implementado, enquanto seus resultados serão discutidos no capítulo 4.

Três experimentos foram executados. Em todos, o agente racional implementado no trabalho é colocado em confronto contra outro agente racional, distribuído pelo professor da disciplina, para fins de testes.

Para simplificação de futuras explicações, define-se os seguintes nomes para os dois agentes citados:

- CPU 1: Agente distribuído pelo professor da disciplina para os testes;
- CPU 2: Agente construído e implementado neste trabalho.

Vale ressaltar que os agentes utilizam diferentes funções heurísticas cada um. Neste caso, CPU 2 utiliza a heurística definida na seção 2.1. Ambos, contudo, utilizam a mesma estratégia do Minimax com poda alfa-beta e busca em corte.

3.1 Exp. 1: Definindo o vetor de parâmetros da função heurística

Antes de mais nada, é preciso definir quais os melhores valores para o vetor $\mathbf{w}$ (equação 2.2) da função heurística do agente CPU 2, a fim de obter-se uma boa métrica para a função.

A seguinte amostra A (equação 3.1) contendo 10 diferentes valores para $\mathbf{w} = (w_1, w_2, w_3, w_4)$ foi submetida a testes. Cada coluna da matriz é um vetor candidato para $\mathbf{w}$. Tais valores foram escolhidos de forma arbitrária, sob a condição necessária de $w_1 < w_2 < w_3 < w_4$,

$$A = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 10^{-4} & 10^{-2} & 10^{-3} \\ 25 & 50 & 100 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 50 & 100 & 10^3 & 100 & 10 & 10 & 5 & 50 & 100 & 100 \\ 100 & 200 & 10^4 & 10^4 & 20 & 100 & 10 & 10^3 & 10^4 & 10^4 \end{pmatrix} \tag{3.1}$$

O critério de avaliação do melhor **w** de A foi o desempenho de confrontos diretos entre os dois agentes (CPU 1 e CPU 2): Para cada **w**, foram realizados 7 confrontos, cujas profundidades do Minimax, para ambos agentes, variaram de 1 a 7. O parâmetro **w** com melhor desempenho (vitórias, empates e derrotas) foi escolhido.

3.2 Exp. 2: CPU vs CPU: Profundidades iguais

Uma vez definido **w**, ambos os agentes são então colocados em confronto novamente. Desta vez, são configurados profundidades do Minimax variando de 1 a 10 para ambos os agentes, totalizando 10 confrontos. O experimento foi dividido em duas partes.

3.2.1 Parte I: Jogador inicial: CPU 1

Na parte I o agente que inicia jogando é o CPU 1, que correspondente ao agente distribuído para testes.

3.2.2 Parte II: Jogador inicial: CPU 2

Na parte II inverte-se o jogador inicial na parte I. Assim, CPU 2 inicia jogando. Os resultados entre partes I e II são comparados.

3.3 Exp. 3: CPU vs CPU: Profundidades variadas

Os agentes novamente são colocados em confronto. Níveis de profundidades são agora diferentes para os dois, variando de 1 até 7. Cada combinação possível de profundidades é testada para ambos, totalizando 49 partidas. Objetiva-se, aqui, ter uma ideia do quão é relevante o nível de profundidade para a qualidade do agente racional. Novamente, o experimento é dividido em duas partes.

3.3.1 Parte I: Jogador inicial: CPU 1

Analogamente ao experimento 2, na parte I o jogador inicial é o CPU 1.

3.3.2 Parte I: Jogador inicial: CPU 2

Na parte II o jogador inicial é o CPU 2. Os resultados entre partes I e II são comparados e discutidos, em seguida.

4 RESULTADOS E DISCUSSÕES

4.1 Exp. 1: Definindo o vetor de parâmetros da função heurística

Conforme descrito anteriormente, uma amostra de 10 vetores de parâmetros **w** foi testada em confrontos diretos entre os dois agentes. Os resultados do desempenho de CPU 2 nos confrontos estão tabelados abaixo.

Tabela 1: Resultado do experimento para CPU 2 vs CPU 1.

Vetor w	Nº de jogos	Vitórias	Empates	Derrotas
(1, 25, 50, 100)	7	4	0	3
(1, 50, 100, 200)	7	4	0	3
(1, 100, 10^3, 10^4)	7	6	0	1
(0, 1, 100, 10^4)	7	6	0	1
(0, 1, 10, 20)	7	6	0	1
(0, 1, 10, 100)	7	6	0	1
(0, 1, 5, 10)	7	6	0	1
(10^{-4}, 1, 50, 10^3)	7	6	0	1
(10^{-2}, 1, 100, 10^4)	7	5	1	1
(10^{-3}, 1, 100, 10^4)	7	4	1	2

A tabela exibe diversos bons candidatos, com 6 vitórias e 1 derrota. Por uma questão de notação, o vetor $\mathbf{w} = (0, 1, 10, 100)$ foi escolhido como parâmetro da heurística.

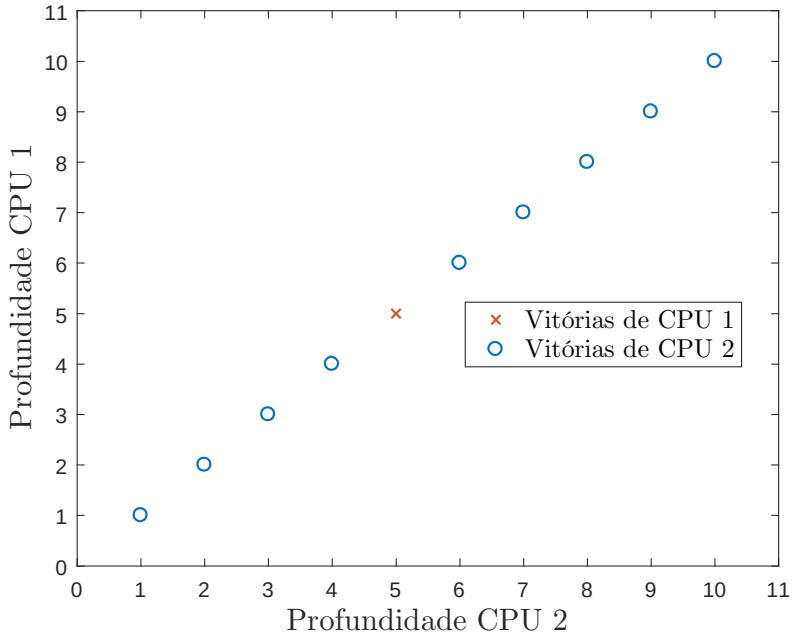
Obviamente poderiam ser feitos mais testes, contendo uma amostra maior de vetores de parâmetros, para um melhor resultado. Entretanto, o tempo de execução do Minimax para profundidades elevadas (acima de 6), mesmo com a poda alfa-beta, é bastante elevado, o que torna inviável testes com uma amostra muito grande. Ainda assim, um vetor de parâmetros que, em 7 jogos, guia o agente ao resultado de 6 vitórias e apenas 1 derrota, aparenta ser um bom parâmetro.

4.2 Exp. 2: CPU vs CPU: Profundidades iguais

4.2.1 Parte I: Jogador inicial: CPU 1

Dez jogos foram executados entre CPU 1 e CPU 2 e os resultados estão representados no gráfico abaixo. Vale novamente lembrar que CPU 1 é o agente distribuído pelo professor e CPU 2 é o agente implementado neste trabalho.

Gráfico 1: Resultados do experimento 2 parte I.



Os resultados ressaltam a qualidade da função heurística implementada em 2.1. Basicamente, há dois meios possíveis para melhorar a performance do agente racional: configurar um nível maior de profundidade máxima do Minimax e utilizar uma boa função heurística. A qualidade de uma função heurística depende de suas características (ou features), isto é, de quantas características do problema estão

sendo levadas em conta no processo de atribuição de nota àquele estado, e de como estas estão sendo avaliadas.

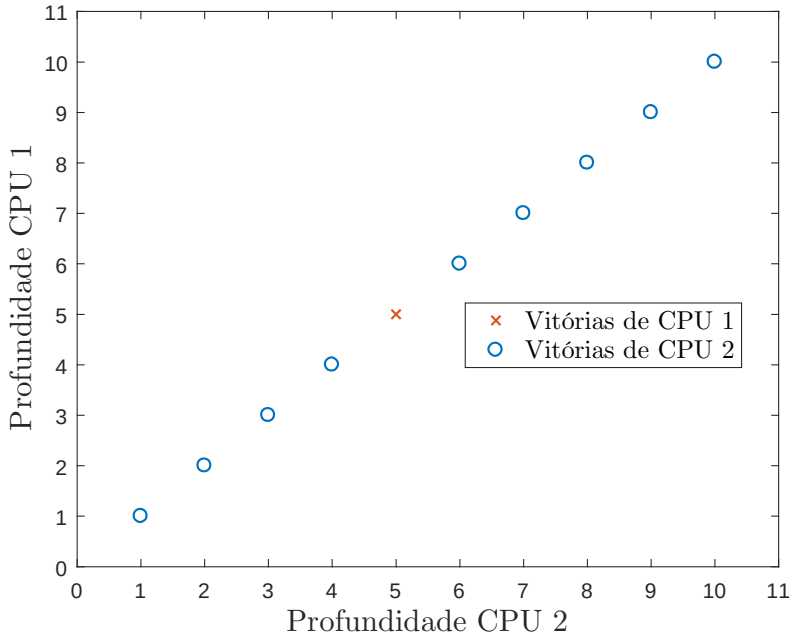
Uma importante *feature* desta função heurística é que esta descarta a contagem de qualquer streak impossível de ser feito, isto é, um streak obstruído pelos limites do tabuleiro ou por uma peça do adversário de tal forma que seja impossível completá-lo com 4 peças. Com isto, temos uma precisão maior na avaliação de um estado.

Entretanto, nem sempre uma boa heurística implica em resultados ótimos com o Minimax. Isto porque o Minimax avalia qual será o próximo movimento do agente baseando-se numa previsão, isto é, na condição de que este agente tome sempre o estado de melhor nota (avaliação de sua própria função heurística) enquanto seu oponente tome sempre o estado que minimize os ganhos do primeiro agente. Na prática, esta previsão nem sempre é correta, pois o oponente pode ser um jogador humano ou um outro agente racional que utilize uma métrica completamente diferente para sua função heurística, fazendo que o oponente tome caminhos diferentes dos previstos pelo agente chamador do Minimax. Assim, a construção de um bom agente racional para o Minimax depende, principalmente, de quantos estados na árvore podem ser analisados, isto é, do seu nível de profundidade.

4.2.2 Parte II: Jogador inicial: CPU 2

O mesmo procedimento da parte I foi repetido, alterando-se o jogador inicial para o CPU 2. Os resultados deste experimento estão representados no gráfico abaixo.

Gráfico 2: Resultados do experimento 2 parte 2.



Note que não houve nenhuma alteração de resultados entre as partes I e II do experimento. A partir deste resultado poderia concluir-se que a escolha do jogador inicial pouco interfere no resultado do Minimax. Entretanto, a segunda parte do terceiro experimento mostrará que isto não é verdade (seção 4.3.2).

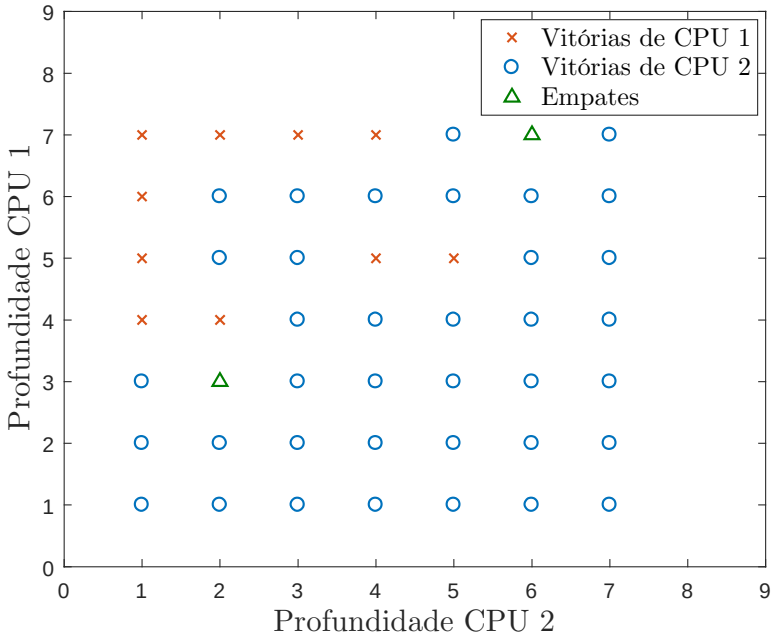
4.3 Exp. 3: CPU vs CPU: Profundidades variadas

4.3.1 Parte I: Jogador inicial: CPU 1

Desta vez foram testados diferentes níveis de profundidade para cada um dos agentes. CPU 1 é o jogador inicial.

Tome os resultados do Gráfico 1. Mais ainda do que uma boa função de avaliação, é importante notar o quanto uma maior profundidade do Minimax influencia na performance de um agente.

Gráfico 3: Resultados do experimento 3 parte I.

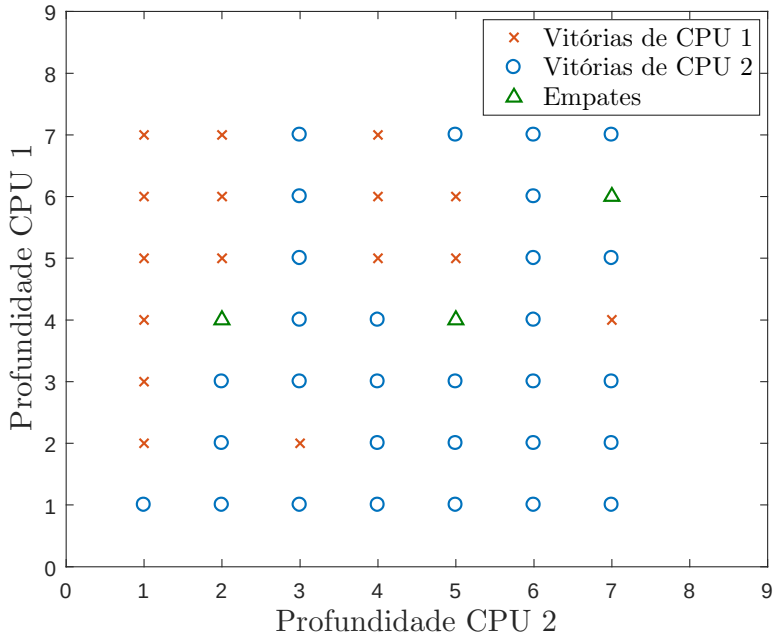


Alguns resultados curiosos podem ser observados quando CPU 1 tem profundidade 5: CPU 2 vence o jogo quando tem profundidades 2 e 3, e perde o jogo quando tem profundidades maiores 4 e 5. Em teoria, um agente é melhor quando possui um nível maior de profundidade para descer em sua árvore de estados com o Minimax. Na prática, isto nem sempre acontece, conforme já foi explicado no último parágrafo da seção 4.2.1.

4.3.2 Parte II: Jogador inicial: CPU 2

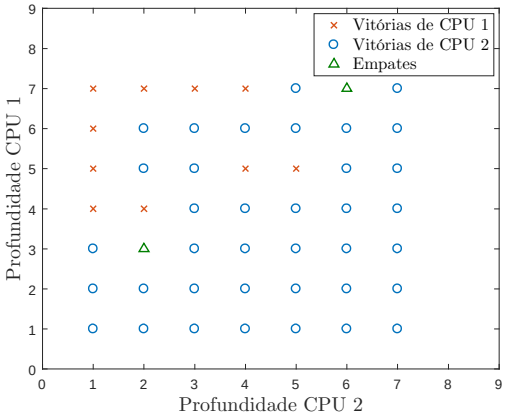
Alterando o jogador inicial para CPU 2 temos o seguinte resultado.

Gráfico 4: Resultados do experimento 3 parte II.

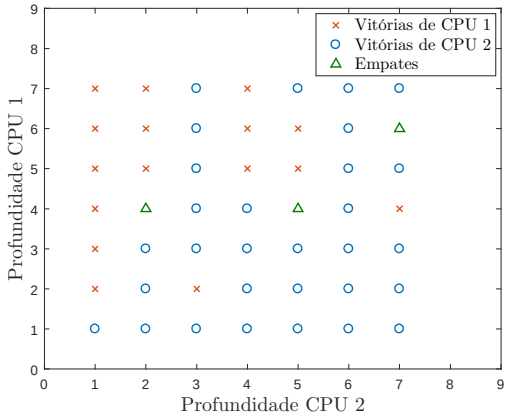


Comparando os resultados contidos em Gráfico 3 e Gráfico 4 (ver Figura 1) nota-se que os resultados são bem diferentes dos resultados obtidos na parte I. CPU 1 vence mais partidas, algumas inclusive quando há a vantagem de CPU 2 com maior nível de profundidade do Minimax.

Figura 1: Comparativo entre os resultados do experimento: Parte I (a), Parte II (b).



(a) Jogador inicial: CPU 1.



(b) Jogador inicial: CPU 2.

Claramente pelos resultados há uma ligeira desvantagem de quem começa jogando. Uma possível explicação para isto talvez seja a de que alguns jogos caem em estados que, inevitavelmente, levará um dos jogadores a vitória. Tome, por exemplo, a seguinte configuração de tabuleiro a seguir, onde o jogador inicial é o jogador 1.

Figura 2: Configuração de tabuleiro do exemplo dado.

1	2	2	1	1		1
2	1	2	2	2		2
1	2	1	1	2		1
2	2	1	2	1		1
1	1	2	2	2		2
2	1	1	1	2		1

O próximo a jogar é o jogador 1, e ele perderá o jogo na rodada seguinte. Analisando os resultados parciais (como configurações de tabuleiro em cada rodada de cada jogo executado) dos experimentos realizados até aqui, foi observado que este tipo de situação era mais comum de acontecer com o segundo jogador completando o streak de 4 peças (como na figura acima) do que o inverso (isto é, quando o jogador inicial vence o jogo, independente do que o segundo jogador faça).

5 CONCLUSÃO

Os resultados dos experimentos mostram que a construção de um bom agente racional em busca competitiva depende, além da implementação de uma boa função heurística, de um nível de profundidade da árvore de estados do Minimax. Este número deve ser grande o suficiente para que o agente possa prever, com mais precisão, quais movimentos levam-no para a vitória, mas também pequeno o suficiente para que o programa não se torne pesado (tanto em termos de memória quanto de processamento). Devido a isto que nenhuma execução do algoritmo acima de profundidade 10 foi realizada.