Universidade Federal de Minas Gerais

DCC642 - Introdução à Inteligência Artificial (2025/2)
TP2: Busca Competitiva

Raphael Henrique Braga Leivas - 2020028101

1 Introdução

Neste trabalho, algoritmos de busca competitiva são implementados em Python para competirem em um jogo de Ligue-4. Diferentes algoritmos são implementados e sua performance é comparada experimentalmente com base em diferentes métricas.

Os algoritmos Minimax, Minimax com poda Alfa-Beta e Iterative Deepening são algoritmos usadas em jogos de dois jogadores para escolher o melhor lance possível. O Minimax simula jogadas futuras alternando turnos de maximizar e minimizar para avaliar estados do jogo e selecionar o que leva ao melhor resultado garantido, usando uma heurísticad e avaliação do estado atual do tabuleiro.

A poda Alfa-Beta é uma otimização do Minimax que reduz o número de nós analisados ao descartar ramos da árvore de decisão que não podem influenciar o resultado final, mantendo a mesma qualidade da decisão, porém com maior eficiência.

Já o Iterative Deepening combina a precisão do Minimax com o limite de tempo, realizando buscas sucessivas com profundidades crescentes e aproveitando resultados anteriores para melhorar a ordem de exploração dos nós, permitindo encontrar a melhor jogada possível dentro do tempo disponível.

As implementações dos algoritmos nesse trabalho são baseadas nas implementações de (Russell and Norvig 2020).

2 Objetivos

Os objetivos principais do trabalho são:

- Implementar em Python os algoritmos Minimax, Minimax com poda Alfa-Beta, Iterative Deepening com uma função heurística de avaliação do estado atual do tabuleiro;
- Comparar as performances dos algoritmos com base nas seguintes métricas: taxa de vitória, tempo médio por jogada, média de estados visitados

3 Metodologia

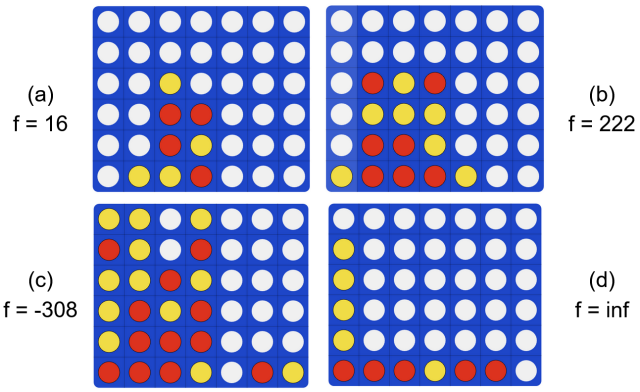
3.1 Heurística de Avaliação

O primeiro passo é definir a heurística de avaliação. Definimos uma heurística da seguinte forma:

- Para cada sequência de 4 posições na horizontal, vertical ou diagonal, adicionamos um valor em uma ordem de grandeza: sequência de duas peças soma-se 10, sequência de três peças soma-se 100, quatro, 1000.
- O valor atribuído varia no sentido positivo indicando que o estado é favorável ao jogar 2, e no sentido negativo para o jogador 1. Assim, o jogador P1 assume o papel de minimizar e o P2 de maximizar.
- As células do centro são mais importantes no jogo, e portanto são multiplicadas por um fator 6.

Dessa forma, a heurística avalia os tabuleiros como mostra a Figura 1.

Figura 1: Exemplos de avaliação do estado atual do jogo com a heurística definida.



Na Figura 1 (a), o jogo parece bem empatado, com ambos jogadores sem ameaças iminentes e com controle central disputado. Assim, a avaliação é próxima de zero: $f(n) = 16$. Em (b), o jogador amarelo tem sequências de 3 peças sem bloqueio e bom controle central, de modo que $f(n) = 222$, um valor positivo elevado. Em (c) temos o contrário de (b), logo a heurística é um valor elevado negativo: $f(n) = -308$.

Por fim, no caso de vitória de um jogador como ocorre em (d), a função retorna $\pm\infty$.

3.2 Agente Minimax

O primeiro agente implementado é o Minimax. O jogador vermelho assume o papel de minimizar e o amarelo de maximizar. O Algoritmo 1 mostra o pseudocódigo para o Minimax usado no projeto. Os experimentos comparam a performance do minimax para diferentes valores de profundidade $depth$, bem como o tempo de execução.

Algorithm 1: Minimax.

```
function MINIMAX(state, depth, maxPlayer)
    if depth = 0 or ISTERMINAL(state) then
        return EVALUATE(state)
    end if
    if maxPlayer then
        maxEval ← −∞
        for all child in SUCCESSORS(state) do
            eval ← MINIMAX(child, depth − 1, false)
            maxEval ← max(maxEval, eval)
        end for
        return maxEval
    else
        minEval ← +∞
        for all child in SUCCESSORS(state) do
            eval ← MINIMAX(child, depth − 1, true)
            minEval ← min(minEval, eval)
        end for
        return minEval
    end if
end function
```

3.3 Agente Minimax com Poda Alfa Beta

Para adicionar a Poda Alfa Beta no Algoritmo 1, basta adicionar a seguinte condicional no algoritmo dentro do loop dos sucessores:

Algorithm 2: Minimax com Poda Alfa Beta.

```
if α ≥ β then
    break
end if
```

Note que α e β agora são argumentos passados para a função Minimax. O número de nós expandidos com a poda é comparado com o Minimax sem pode, de modo a verificar experimentalmente o impacto da poda na execução do algoritmo.

3.4 Iterative Deepening

Para implementar o Iterative Deepening, basta chamar o Minimax várias vezes incrementando a profundidade máxima e usar o melhor lance encontrado entre todos as buscas, como mostra o Algoritmo 3. Usamos a poda Alfa-Beta dado que o Minimax é chamado várias vezes, de modo a maximizar o número de chamadas dentro do limite de tempo estipulado.

Algorithm 3: Iterative Deepening com Minimax

```
Require: Initial state s, maximum depth D, player p
Ensure: Best move found
bestMove ← null
for d = 1 to D do
    move ← Minimax(s, d, −∞, +∞, true, p)
    if move ≠ null then
        bestMove ← move
    end if
end for
return bestMove
```

3.5 Experimentos

Os seguintes experimentos serão realizados:

- Minimax vs Aleatório
- Alfa-Beta vs Minimax (sem poda)
- Iterative Deepening vs Alfa-Beta
- IA do Aluno vs Jogador Humano

Os experimentos são realizados com o seguinte procedimento:

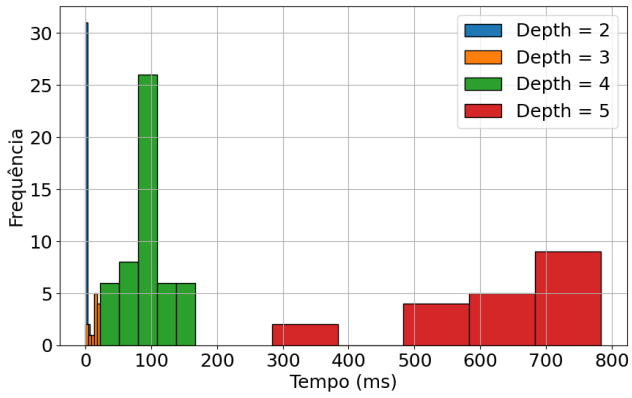
1. Realiza 3 jogos entre o Minimax e o oponente para cada nível de profundidade;
2. Salva os tempos por jogada e número de nós expandidos em cada jogada em um csv;
3. Analisa os dados a posteriori com `matplotlib` e extrai as conclusões.

4 Resultados

4.1 Minimax vs Aleatório

A Figura 2 mostra os tempos por lance para diferentes profundidades do Minimax, e A Figura 3 número de nós expandidos com diferentes profundidades configuradas. Como esperado, o algoritmo gasta mais tempo e expande mais nós para profundidades maiores.

Figura 2: Histograma de tempos gasto por lance para o Minimax para diferentes profundidades.



Podemos tomar a média e o desvio padrão para cada uma das medições acima, e junto com a taxa de vitórias para cada profundidade, obtemos os resultados da Tabela 1.

Figura 3: Histograma de nós expandidos por lance para o Minimax para diferentes profundidades.

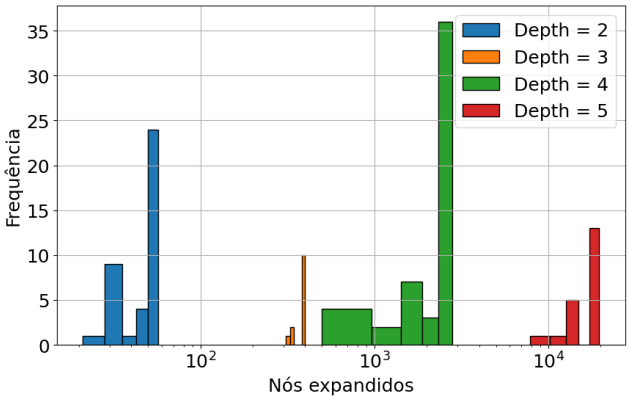


Tabela 1: Resultados finais do experimento Minimax vs Aleatório.

Depth	Vitórias (%)	Tempo (ms)	Nós
2	0	3.6 ± 4.9	48.2 ± 12
3	100	14.1 ± 7.4	384 ± 30
4	100	92.5 ± 33.2	2357 ± 695
5	100	618 ± 130	16769 ± 2354

4.2 Alfa-beta vs Minimax

Repetindo o mesmo procedimento do primeiro experimento, mas dessa vez usando o minimax com a pode alta-beta vs o minimax, obtemos os resultados exibidos na Tabela 2

Tabela 2: Resultados finais do experimento Minimax Alfa-Beta vs Minimax.

Depth	Vitórias (%)	Tempo (ms)	Nós
2	0	1.7 ± 2.7	31 ± 7.96
3	100	6.4 ± 3.8	125.6 ± 52.2
4	0	10.1 ± 5.9	265 ± 107.7
5	100	83.2 ± 66.3	2366.4 ± 1916

Comparando as Tabelas 1 e 2, vemos que a poda causou uma redução significativa nos tempos e números de nós expandidos a cada lance, em particular para maiores profundidades. A Figura 4 sumariza as diferenças entre os tempos e números de nós visitados para ambos algoritmos, destacando a redução com a poda alfa-beta.

4.3 Iterative Deepening vs Alfa-beta

Para o tempo limite de 1000 ms, o Iterative Deepening consegue atingir a profundidade 6. Com 2000 ms, atinge a profundidade 7. As vezes estourava o limite e usava o lance aleatório de fallback ao tentar atingir a pronfundidade 7 com 2000 ms, de modo que a performance com esse limite foi menor que a com 1000 ms. A Tabela 3 sumariza os resultados obtidos nesse experimento.

Figura 4: Diferenças entre os tempos e números de nós expandidos pela profundidade entre os algoritmos.

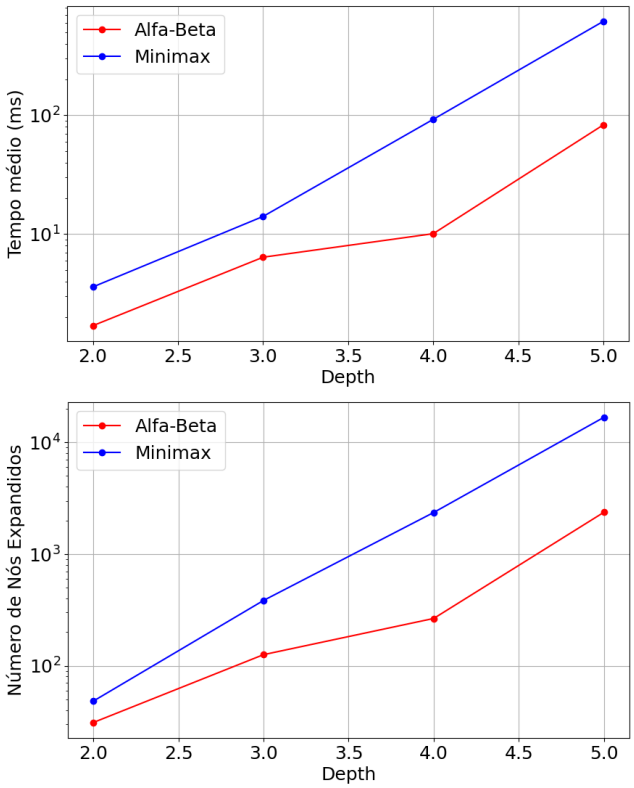


Tabela 3: Resultados finais do experimento Iterative Deepning vs Alfa-beta.

Limite (ms)	Vitórias (%)
1000	100
2000	0

4.4 Humano vs Alfa-beta

A melhor IA obtida foi o Minimax com poda Alfa-beta e profundidade 5. Ao jogar 5 partidas contra ela, obtive os resultados exibidos na Tabela 4.

Não consegui vencer a melhor IA em nenhuma das partidas. Quase sempre eu perdia por não ver alguma diagonal ameaçada. Uma outra observação interessante é que a IA sempre começa com a mesma jogada, exibida na Figura 5. Se eu não bloquear a primeira linha, ela sempre irá colocar nas colunas adjacentes e ameaçar colcoar 3 bolas em sequência com os extremos abertos, uma ameaça dupla que vence o jogo.

A jogada inicial da Figura 5 só ocorre em profundidade maior que 4. Para profundidades 2 e 3, ela começa em um dos cantos do tabuleiro.

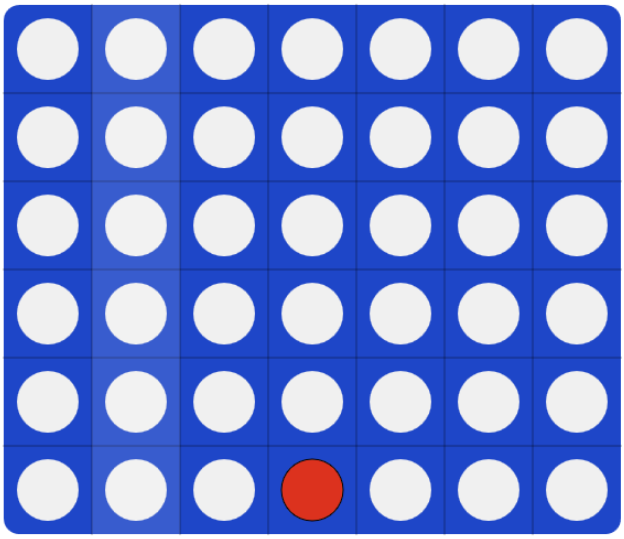
5 Conclusão

Tendo em vista os objetivos do trabalho, foi possível comparar experimentalmente a performance de três algoritmos

Tabela 4: Resultados finais das partidas humano versus melhor IA.

Partida	Vencedor	Número de Lances
1	IA	18
2	IA	14
3	IA	12
4	Empate	22
5	IA	17

Figura 5: Jogada inicial da IA em todas as partidas.



de busca competitiva vistos em sala de aula no jogo de Ligue-4, bem como os efeitos que a função heurística tem na execução do algoritmo Minimax.

Além disso, foi possível verificar como a poda Alfa-beta consegue reduzir substancialmente o tempo gasto por lance e o número de nós expandidos.

Referências

Russell, S.; and Norvig, P. 2020. *Artificial Intelligence: A Modern Approach*. Pearson, 4 edition. ISBN 978-0134610993.