Universidade Federal de Ouro Preto
Instituto de Ciências Exatas e Aplicadas
Departamento de Computação e Sistemas

Relatório do TP01 - Jogo do conecta-4

Aluno(s): Francis Lucas de Sá - 19.1.8100
Guilherme Ribeiro Figueiredo - 19.1.8024

Julho
2023

Universidade Federal de Ouro Preto
Instituto de Ciências Exatas e Aplicadas
Departamento de Computação e Sistemas

Relatório

Relatório do trabalho 01 do jogo conecta-4.

Alunos: Francis Lucas de Sá - 19.1.8100
Guilherme Ribeiro Figueiredo - 19.1.8024
Professor: Talles Henrique de Medeiros

Julho
2023

Sumário

1 Resumo

Neste trabalho, apresentamos o desenvolvimento de um agente de Inteligência Artificial (IA) capaz de jogar o jogo conecta-4, utilizando o algoritmo da poda alfa-beta e a heurística de janelas deslizantes. O objetivo é criar um agente que consiga vencer o adversário humano, explorando as melhores jogadas possíveis, descartando as ramificações desnecessárias da árvore de busca e tomando boas decisões mesmo sem explorar todo o tabuleiro.

Para isso, partimos do código original disponibilizado pelo professor, que continha o jogo básico e algumas funções auxiliares. A seguir, descrevemos as principais modificações e implementações realizadas no código, bem como os resultados obtidos e as dificuldades encontradas.

2 Plano de desenvolvimento

A primeira modificação feita no código original disponibilizado pelo professor foi a correção da orientação das peças no tabuleiro, originalmente o jogo ocorria de cima para baixo, e agora ele está no sentido correto de baixo para cima. Posteriormente foi implementado de forma paralela o algoritmo da poda alfa-beta e a heurística de janelas deslizantes.

2.1 Orientação do jogo

A estrutura da matriz em python que representa o tabuleiro do jogo é ordenada de cima para baixo, da esquerda para a direita, ou seja, em um tabuleiro 6x7 a linha 0 é que vai estar no topo e a linha 5 é a que está no fundo (lembrando que as posições de vetores em python começam em 0 e vão até n - 1). Tendo isso em mente, o código original na figura 1, e mantendo o exemplo de um tabuleiro 6x7, a função valid location original verifica se a posição na linha (5) e coluna Y que está no fundo já está preenchida com alguma peça (1 ou 2) e caso não esteja retorna verdadeiro, indicando que é possível soltar uma peça nessa coluna. Já a função drop piece original percorre as linhas do tabuleiro de cima para baixo, até encontrar um espaço vazio, então ela posiciona uma peça na linha X e coluna escolhida pelo jogador da vez.

```python
def valid_location(board, column):
    return board[ROWS - 1][column] == 0

# ----------------------------------------
def drop_piece(board, column, piece):
    for r in range(ROWS):
        if board[r][column] == 0:
            board[r][column] = piece
            return
```

Figura 1: Código original.

Para alterar a ordenação do jogo, modificamos a função valid location para que ela verificasse a linha 0 ao invés da última, e também adaptamos a função drop piece para que ela verificasse as linhas de baixo para cima. Essas simples modificações foram o suficiente para alterar a ordem do jogo, fazendo com que ele fique mais próximo do conecta4 real. O código modificado pode ser encontrado na figura 2 abaixo.

```python
def valid_location(board, column):
    return board[0][column] == 0  # Tem que verificar a linha 0 e não a última


# --------------------------------------------------------------------------------
def drop_piece(board, column, piece):
    for r in range(ROWS - 1, -1, -1):  # Troquei a ordem da lista, para começar de baixo
        if board[r][column] == 0:
            board[r][column] = piece
            return
```

Figura 2: Código modificado.

2.2 Heurística

A Heurística implementada no código foi a de sliding windows (janelas deslizantes) e sua inspiração veio das citações em aula do professor e das camadas de redes convolucionais. A ideia por trás das janelas deslizantes é verificar apenas uma parte do tabuleiro por vez e pular as partes que não tem informações úteis, diminuindo assim o processamento necessário se comparado com uma implementação que verifica todo o tabuleiro.

As janelas deslizantes implementadas aqui tem um tamanho de 4x4 e percorrem o tabuleiro de baixo para cima, da esquerda para a direita, pulando para a direita sempre que a linha 0 da janela não possuir nenhuma peça, indicando que não há necessidade de calcular a pontuação de janelas acima, pois elas estão vazias.

O código das janelas deslizantes foi dividido partes, a primeira (slidding windows, figura 3) cria as janelas e as percorre, pulando as janelas que não trarão informações úteis e retornando a pontuação do estado atual do tabuleiro.

A segunda parte é a window score, ela recebe como parametros a janela e qual é a peça do jogador que está sendo avaliado, então ela avalia a pontuação total daquela janela com base nas condições: 100 pontos para 4 peças do jogador e 0 do adversário, 5 pontos para 3 peças do jogador e 1 espaço vazio, 2 pontos para 2 peças do jogador e 2 espaços vazios, e -4 pontos para 3 peças do jogador e 1 do adversário. As condições são verificadas em todas as linhas, colunas, diagonal principal e secundária da janela. As condições de pontuação e valores foram baseadas no projeto [Galli, 2019].

```python
def sliding_windows(board, piece):
    score = 0

    for c in range(0, COLUMNS - WINDOW_SIZE + 1, STRIDE):
        for r in range(ROWS - WINDOW_SIZE, -1, -STRIDE):
            window = board[r:r + WINDOW_SIZE, c:c + WINDOW_SIZE]
            if np.count_nonzero(window[0, :] == 0) == WINDOW_SIZE:
                break
            score += window_score(window, piece)

    return score
```

Figura 3: Código que cria as janelas e retorna a pontuação de cada uma.

```python
def window_score(window, piece):  # Baseado no conceito das sliding windows das CNNs
    score = 0  # Vamos definir a pontuação de acordo com a quantidade de peças na janela
    adversary = 1  # Por padrão o adversario será o jogador

    conditions = [
        ((4, 0), 100),
        ((3, 1), 5),
        ((2, 2), 2),
        ((3, 1), - 4)
    ]

    if adversary == piece:
        adversary = 2  # Caso a peça seja do jogador, setamos o adversario para a IA (2)

    score += horizontal_score(window, piece, adversary, conditions)
    score += vertical_score(window, piece, adversary, conditions)
    score += diagonal_score(window, piece, adversary, conditions)

    # print(f'Score: {score} ')
    return score
```

Figura 4: Trecho do código que faz a avaliação da janela.

2.3 Poda Alfa-Beta

A poda alfa-beta é uma técnica utilizada em algoritmos de busca em árvores de decisão ou em outros problemas que possam ser modelados dessa forma, como o jogo de xadrez, por exemplo. O objetivo da poda alfa-beta é reduzir o número de nós que precisam ser explorados em uma árvore de decisão, permitindo uma busca mais eficiente.

O algoritmo de poda alfa-beta funciona considerando dois valores adicionais para cada nó na árvore: alfa e beta. O valor alfa representa o melhor valor atualmente conhecido para o jogador maximizador (geralmente o jogador humano) e o valor beta representa o melhor valor atualmente conhecido para o jogador minimizador (geralmente o jogador adversário ou a IA).

A ideia básica é que, à medida que o algoritmo explora a árvore de decisão, ele realiza uma busca em profundidade, avaliando os nós e atualizando os valores α e β conforme necessário. Durante a busca, quando um nó é avaliado e descobre-se que ele não afetará o resultado final (ou seja, não contribuirá para a escolha da melhor jogada), ocorre a poda. Para isso, existem algumas metodologias para melhorar ainda mais essa poda, com os algoritmos "Fail-Soft"e "Fail-Hard".

```
ab (p,alpha,beta)
position p; int alpha,beta;
{
    int m,d;
    if p is a leaf return (staticvalue(p));
    let Pl ... Pd be the successors of p;
    m = -inf
    for ( i = 1;i< = d ; i + +){
        m = max(m,- ab(Pi,-beta,-m));
        if(m > = beta) return(m);
    }
    return (m);
}
```

Figura 5: Pseudocódigo do Fail-Soft para β.

O método utilizado foi o "Fail-Soft", uma variante do algoritmo Alpha-Beta, introduzida por John P. Fishburn, em 1984, no artigo "Another optimization of alpha-beta search", que permite uma busca eficiente em árvores de decisão com a capacidade de retornar um valor aproximado mesmo quando ocorre um estouro de tempo ou recursos durante a busca.

```python
if maximizing_player:
    value = -np.Inf
    column = np.random.choice(valid_locations)
    for col in valid_locations:
        temp_board = board.copy()
        drop_piece(temp_board, col, 2)

        STATES_EXPLORED += 1

        new_score = minimax_alpha_beta(temp_board, depth - 1, False, ALPHA, BETA)[1]
        if new_score > value:
            value = new_score
            column = col

        ############################################
        ALPHA = max(ALPHA, value)
        if ALPHA >= BETA:
            break
        ############################################

    return column, value
```

Figura 6: Implementação do Fail-soft para o Maximizing.

```python
else:  # minimizing player
    value = np.Inf
    column = np.random.choice(valid_locations)
    for col in valid_locations:
        temp_board = board.copy()
        drop_piece(temp_board, col, 1)

        STATES_EXPLORED += 1

        new_score = minimax_alpha_beta(temp_board, depth - 1, True, ALPHA, BETA)[1]
        if new_score < value:
            value = new_score
            column = col

        ############################################
        BETA = min(BETA, value)
        if ALPHA >= BETA:
            break
        ############################################

    return column, value
```

Figura 7: Implementação do Fail-soft para o Minimizing.

A atualização do valor de α ou β acontece antes da poda, permitindo um armazenamento mais preciso de valores

3 Metodologia

Para realizar os testes e avaliar o desempenho dos algoritmos desenvolvidos, definimos 2 testes de validação (um com um agente aleatório vs IA e o segundo com uma IA vs IA), 4 tamanhos de tabuleiro (6x7, 10x11, 12x13, 14x15) foram testados em 4 cenários dos mesmos, o primeiro cenário (cenário 1) é o caso base utilizando o minimax original e sem um sistema de heurística; O segundo (cenário 2) consiste em utilizar o minimax com a poda alpha-beta ainda sem a heurística; O terceiro (cenário 3) consiste em utilizar o minimax e a heurística; O quarto (cenário 4) é utilizando a minimax com poda alpha-beta e a heurística.

As métricas utilizadas na avaliação foram a quantidade de estados explorados e o tempo gasto pelas jogadas da IA. Os resultados serão abordados no próximo capítulo.

4 Análise dos Resultados

Com a intenção de obter testes mais regulares, as experiências foram realizadas em uma máquina virtual pela plataforma do google Colab. Para verificar a eficiência do algoritmo, foram testados diferentes tamanhos de tabuleiro, como o 6x7, 10x11, 12x13 e 14x15. Sendo o 14x15 o último tamanho testável em tempo tolerável.

A seguir, serão apresentadas tabelas com a comparação dos números de estados explorados em cinco partidas, comparando cada algoritmo e em seguida, o tempo correspondente às mesmas partidas.

4.1 Agente aleatório

O agente será um valor pseudo-aleatório a cada jogada. Não possui estratégia nem heurística.

Para um tabuleiro 6x7

Cada alfa-beta teve uma derrota.

Número de estados

Tabela 1: Estados explorados

Algoritmo	Partida1	Partida2	Partida3	Partida4	Partida5
Minimax	23408	10262	20552	19316	16909
Minimax+alfa-beta	24670	25814	10262	10507	10507
Minimax+Eval(s)	51	13	13	13	13
Minimax+Eval(s)+alfa-beta	13	13	33	17	13

Tempo gasto

Tabela 2: Tempo gasto

Algoritmo	Partida1	Partida2	Partida3	Partida4	Partida5
Minimax	3.059802	1.289378	2.765436	2.572438	2.1996
Minimax+alfa-beta	3.558124	3.437732	1.246346	1.289066	1.293015
Minimax+Eval(s)	0.010726	0.002689	0.002744	0.002701	0.002826
Minimax+Eval(s)+alfa-beta	0.003055	0.002747	0.007777	0.004766	0.002811

Para um tabuleiro 10x11

Cada alfa-beta teve uma derrota.

Número de estados

Tabela 3: Estados explorados

Algoritmo	Partida1	Partida2	Partida3	Partida4	Partida5
Minimax	185811	60654	61743	92862	61743
Minimax+alfa-beta	201462	61743	61743	60654	171160
Minimax+Eval(s)	13	13	13	13	21
Minimax+Eval(s)+alfa-beta	17	13	17	17	24

Tempo gasto

Tabela 4: Tempo gasto

Algoritmo	Partida1	Partida2	Partida3	Partida4	Partida5
Minimax	73.970631	23.188415	23.78653	36.02637	23.91333
Minimax+alfa-beta	82.223703	24.457752	24.3908	23.587492	68.977299
Minimax+Eval(s)	0.007488	0.007388	0.008111	0.00753	0.012065
Minimax+Eval(s)+alfa-beta	0.009727	0.007457	0.010149	0.010243	0.014252

Para um tabuleiro 12x13

Apenas o alfa-beta eval foi derrotado 1 vez.

Número de estados

Tabela 5: Estados explorados

Algoritmo	Partida1	Partida2	Partida3	Partida4	Partida5
Minimax	115635	119353	148434	271869	242957
Minimax+alfa-beta	144248	115635	178152	147966	177190
Minimax+Eval(s)	13	13	13	13	25
Minimax+Eval(s)+alfa-beta	49	21	13	12	13

Tempo gasto

Tabela 6: Tempo gasto

Algoritmo	Partida1	Partida2	Partida3	Partida4	Partida5
Minimax	67.374388	69.99939	87.402717	159.641472	142.413449
Minimax+alfa-beta	82.895788	65.55728	103.312812	86.198643	103.039111
Minimax+Eval(s)	0.012323	0.011196	0.010828	0.010994	0.020666
Minimax+Eval(s)+alfa-beta	0.042404	0.0183	0.011675	0.010311	0.01161

Para um tabuleiro 14x15

Apenas o minimax eval foi derrotado 1 vez.

Número de estados

Tabela 7: Estados explorados

Algoritmo	Partida1	Partida2	Partida3	Partida4	Partida5
Minimax	769590	423600	315120	474915	585885
Minimax+alfa-beta	210195	207270	210195	210195	368505
Minimax+Eval(s)	13	13	13	32	13
Minimax+Eval(s)+alfa-beta	13	21	13	17	13

Tempo gasto

Tabela 8: Tempo gasto

Algoritmo	Partida1	Partida2	Partida3	Partida4	Partida5
Minimax	614.227443	336.4109	248.30685	377.8591	471.0443
Minimax+alfa-beta	167.303252	165.8766	167.4013	166.6445	291.2403
Minimax+Eval(s)	0.014626	0.014811	0.016892	0.036883	0.014685
Minimax+Eval(s)+alfa-beta	0.014656	0.024105	0.016362	0.020322	0.015189

4.2 Segundo tipo de teste

Para um tabuleiro 6x7

Jogador 1: 10 vitórias, Jogador 2: 3 vitórias, 5 empates.

Número de estados 1

Tabela 9: Tabela do nº de estados explorados pelo jogador 1

Algoritmo	Partida1	Partida2	Partida3	Partida4	Partida5
Minimax	17057	16127	16729	16567	16394
Minimax+alfa-beta	16127	16729	16824	16729	16394
Minimax+Eval(s)	34	46	36	46	46
Minimax+Eval(s)+alfa-beta	76	70	36	46	36

Número de estados 2

Tabela 10: Tabela do nº de estados explorados pelo jogador 2

Algoritmo	Partida1	Partida2	Partida3	Partida4	Partida5
Minimax	19080	16966	18729	18383	17846
Minimax+alfa-beta	16966	18729	18867	18729	17846
Minimax+Eval(s)	36	48	38	48	48
Minimax+Eval(s)+alfa-beta	78	72	38	48	38

Tempo 1

Tabela 11: Tabela do tempo do jogador 1

Algoritmo	Partida1	Partida2	Partida3	Partida4	Partida5
Minimax	1.351489	1.206428	1.322218	1.305438	1.269111
Minimax+alfa-beta	1.197923	1.299917	1.335186	1.336579	1.308966
Minimax+Eval(s)	0.005331	0.006392	0.004508	0.006608	0.006074
Minimax+Eval(s)+alfa-beta	0.010031	0.00945	0.004501	0.007	0.004547

Tempo 2

Tabela 12: Tabela do tempo do jogador 2

Algoritmo	Partida1	Partida2	Partida3	Partida4	Partida5
Minimax	1.218894	1.174307	1.197743	1.190758	1.187795
Minimax+alfa-beta	1.172064	1.171966	1.19254	1.210889	1.20223
Minimax+Eval(s)	0.00494	0.005881	0.004257	0.005815	0.005767
Minimax+Eval(s)+alfa-beta	0.009978	0.009431	0.004225	0.006701	0.004182

Para um tabuleiro 10x11

Jogador 1: 6 vitórias, Jogador 2: 7 vitórias, 7 empates.

Número de estados 1

Tabela 13: Tabela do nº de estados do jogador 1

Algoritmo	Partida1	Partida2	Partida3	Partida4	Partida5
Minimax	224980	222518	222518	221768	223696
Minimax+alfa-beta	224980	220798	223696	221768	223696
Minimax+Eval(s)	118	98	60	60	60
Minimax+Eval(s)+alfa-beta	128	98	60	58	78

Número de estados 2

Tabela 14: Tabela do nº de estados do jogador 2

Algoritmo	Partida1	Partida2	Partida3	Partida4	Partida5
Minimax	236054	232278	232278	229610	233607
Minimax+alfa-beta	236054	225909	233607	229610	233607
Minimax+Eval(s)	120	100	62	62	62
Minimax+Eval(s)+alfa-beta	130	100	62	60	80

Tempo 1

Tabela 15: Tabela do tempo do jogador 1

Algoritmo	Partida1	Partida2	Partida3	Partida4	Partida5
Minimax	48.916048	48.623741	48.642216	48.039691	48.689386
Minimax+alfa-beta	48.911084	47.113684	48.762564	48.122312	48.634398
Minimax+Eval(s)	0.042007	0.032195	0.018444	0.018599	0.025154
Minimax+Eval(s)+alfa-beta	0.042888	0.032141	0.019457	0.01849	0.018396

Tempo 2

Tabela 16: Tabela do tempo do jogador 2

Algoritmo	Partida1	Partida2	Partida3	Partida4	Partida5
Minimax	47.026755	46.777917	46.983605	46.643466	47.143972
Minimax+alfa-beta	47.223258	46.228371	46.762824	46.861948	46.848654
Minimax+Eval(s)	0.041203	0.03147	0.018115	0.018239	0.024445
Minimax+Eval(s)+alfa-beta	0.042208	0.031187	0.018802	0.017923	0.017773

Para um tabuleiro 12x13

Jogador 1: 8 vitórias, Jogador 2: 10 vitórias, 2 empates.

Número de estados 1

Tabela 17: Tabela do nº de estados do jogador 1

Algoritmo	Partida1	Partida2	Partida3	Partida4	Partida5
Minimax	586703	587962	497795	587962	584797
Minimax+alfa-beta	584797	583457	589779	589779	584797
Minimax+Eval(s)	72	94	118	72	70
Minimax+Eval(s)+alfa-beta	154	142	142	72	94

Número de estados 2

Tabela 18: Tabela do nº de estados do jogador 2

Algoritmo	Partida1	Partida2	Partida3	Partida4	Partida5
Minimax	604794	606337	520194	606337	606068
Minimax+alfa-beta	606068	593640	610648	610648	606068
Minimax+Eval(s)	74	96	120	74	72
Minimax+Eval(s)+alfa-beta	156	144	144	74	96

Tempo 1

Tabela 19: Tabela do tempo do jogador 1

Algoritmo	Partida1	Partida2	Partida3	Partida4	Partida5
Minimax	187.3794	187.4348	153.7352	187.7325	185.3373
Minimax+alfa-beta	187.3794	187.4348	153.7352	187.7325	185.3373
Minimax+Eval(s)	0.031499	0.043447	0.057589	0.031985	0.031756
Minimax+Eval(s)+alfa-beta	0.070815	0.068809	0.066793	0.031624	0.043303

Tempo 2

Tabela 20: Tabela do tempo do jogador 2

Algoritmo	Partida1	Partida2	Partida3	Partida4	Partida5
Minimax	182.9885	183.215	147.7091	183.1593	181.0753
Minimax+alfa-beta	181.2233	181.8246	182.5226	183.1307	181.4990
Minimax+Eval(s)	0.030628	0.042685	0.057137	0.031071	0.03056
Minimax+Eval(s)+alfa-beta	0.069188	0.067526	0.065805	0.030763	0.042223

Para um tabuleiro 14x15

Jogador 1: 4 vitórias, Jogador 2: 16 vitórias, 0 empates.

Número de estados 1

Tabela 21: Tabela do nº de estados do jogador 1

Algoritmo	Partida1	Partida2	Partida3	Partida4	Partida5
Minimax	1352915	1071926	1352915	1350526	1352915
Minimax + alfa-beta	1344579	1344579	1351234	1351234	1071926
Minimax+Eval(s)	138	84	166	84	84
Minimax+Eval(s) + alfa-beta	98	84	84	84	84

Número de estados 2

Tabela 22: Tabela do nº de estados do jogador 2

Algoritmo	Partida1	Partida2	Partida3	Partida4	Partida5
Minimax	1389079	1111847	1389079	1381436	1389079
Minimax + alfa-beta	1363261	1363261	1382171	1382171	1111847
Minimax+Eval(s)	140	86	168	86	86
Minimax+Eval(s) + alfa-beta	100	86	86	86	86

Tempo 1

Tabela 23: Tabela do tempo do jogador 1

Algoritmo	Partida1	Partida2	Partida3	Partida4	Partida5
Minimax	593.6893	453.1479	593.0365	596.7628	593.1792
Minimax + alfa-beta	586.8325	586.2814	594.0535	592.8356	450.8861
Minimax+Eval(s)	0.087557	0.049977	0.105813	0.049508	0.04975
Minimax+Eval(s) + alfa-beta	0.054918	0.049411	0.049623	0.049474	0.049512

Tempo 2

Tabela 24: Tabela do tempo do jogador 2

Algoritmo	Partida1	Partida2	Partida3	Partida4	Partida5
Minimax	583.3407	440.6540	584.3515	585.9702	583.8673
Minimax + alfa-beta	582.6660	580.6331	584.4621	582.8606	438.8960
Minimax+Eval(s)	0.08593	0.050642	0.1047	0.048556	0.048454
Minimax+Eval(s) + alfa-beta	0.05415	0.048759	0.048517	0.04844	0.048319

5 Conclusão

Conforme a análise detalhada das tabelas anteriores, podemos concluir que o algoritmo que apresentou o melhor resultado geral foi o Minimax + Eval(s), destacando a eficiência da heurística empregada, seguido do Minimax + Eval(s) + alfa-beta, demonstrando que caso uma heurística eficiente seja escolhida a poda pode nem sempre ser necessária. Porém, conforme o tabuleiro aumentou, o valor de Minimax + Eval(s) e Minimax + Eval(s) + alfa-beta foi se aproximando, com a primeira opção apresentando maior vantagem em tabuleiros menores.

Em suma, os resultados apontam para a importância da escolha da heurística, uma vez que ela e o tamanho do tabuleiro influênciam diretamente o desempenho geral dos algoritmos.

Bibliografia

Carolus, Jeroen W.T. (2006). *Alpha-Beta with Sibling Prediction Pruning in Chess*. https://homepages.cwi.nl/~paulk/theses/Carolus.pdf. Acessado em 29 de junho de 2023.

cmdepi (2021). *Alpha-beta pruning: Fail-hard VS. Fail-soft*. https://stackoverflow.com/questions/70050677/alpha-beta-pruning-fail-hard-vs-fail-soft. Acessado em 28 de junho de 2023.

Fishburn, John P. (1983). *Another optimization of alpha-beta search*. https://dl.acm.org/doi/pdf/10.1145/1056623.1056628. Acessado em 29 de junho de 2023.

Galli, Keith (2019). *Connect4-Python*. https://github.com/KeithGalli/Connect4-Python. Acessado em 28 de junho de 2023.

Hsu, Tsan-sheng (2012). *Alpha-Beta Pruning: Algorithm and Analysis*. https://homepage.iis.sinica.edu.tw/~tshsu/tcg/2012/slides/slide7.pdf. Acessado em 28 de junho de 2023.

Urbański, Mariusz (2011). "Logic and Cognition: the Faces of Psychologism". Em: *Logic and Logical Philosophy* 20, pp. 175–185.

Anexo

Link do repositório do trabalho: https://github.com/Uelfol/trabalho