Connect Four

Trabalho Realizado Por:

Miguel Proença Fernandes Ramalho Amaro, up202106985

Miguel Filipe Miranda dos Santos, up202105289

Nuno dos Santos Moreira, up202104873

Índice

Introdução

Neste trabalho iremos desenvolver um programa que permita jogar "Connect Four" (ou "4-in-line", em português "4 em linha") na consola, bem como um agente capaz de jogar este jogo.

Este jogo, contrariamente ao levado a cabo no trabalho anterior, envolve 2 jogadores. Esta mudança reflete-se de dois modos: a introdução de um elemento de imprevisibilidade, devido ao não conhecimento da jogada do adversário; e a mudança do método de solução, que advém do aumento do espaço de busca e do ponto anteriormente mencionado.

Assim, de acordo com as suas características, cada jogo terá um melhor método de resolução. Geralmente, jogos com adversário são caracterizados por:

- Quantidade de informação sobre o jogo disponibilizada aos jogadores (perfeitos se ambos os jogadores têm toda a informação sobre o jogo, imperfeitos caso contrário);
- Existência de elementos de "sorte" (determinísticos se os resultados não dependem de fatores externos; indeterminísticos caso contrário).

Para resolver este tipo de problemas recorremos a busca adversarial, formulando o jogo como um problema de busca com as seguintes componentes: estado inicial; função para gerar os sucessores; teste de terminação; utility function; limite de profundidade.

Antes de iniciar o jogo atribui-se a um jogador o título de MAX (tem como objetivo maximizar a utility function e inicia o jogo) e ao outro jogador o título de MIN (tem como objetivo minimizar a utility function). Com estes dados, expande-se a árvore de jogadas até aos estados terminais, utiliza-se a utility function para avaliar a utilidade de cada estado e, por fim, realiza-se uma jogada de acordo com o algoritmo escolhido. Neste trabalho iremos abordar 3 algoritmos adversariais (Minimax, Alpha-Beta Pruning, Monte Carlo Tree Search).

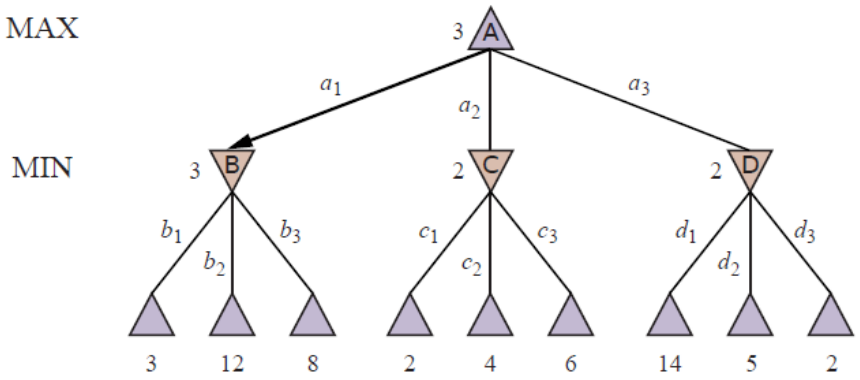
Minimax

Neste algoritmo MAX assume o pior cenário possível, ou seja, as suas jogadas são como que um plano de contingência, assumindo que o oponente joga de forma ótima.

Assim, após ter expandido a árvore de jogadas e atribuído um valor a cada estado final de acordo com a utility function, vai-se retornando o valor mais adequado a cada turno (MAX escolhe a jogada que mais maximiza o valor, MIN escolhe a que mais minimiza o valor) desde os estados finais até à raiz. Assim, torna-se possível escolher a jogada que parece oferecer um melhor resultado.

Este algoritmo recorre a depth-first search para explorar a árvore de jogadas. Sendo *d* a profundidade e *b* o número de movimentos possíveis em cada ponto (branching factor), então a complexidade espacial do Minimax é O(bd), enquanto que a complexidade temporal é $O(b^d)$.

Temos, na imagem que se segue, um exemplo que permite melhor visualizar este algoritmo.



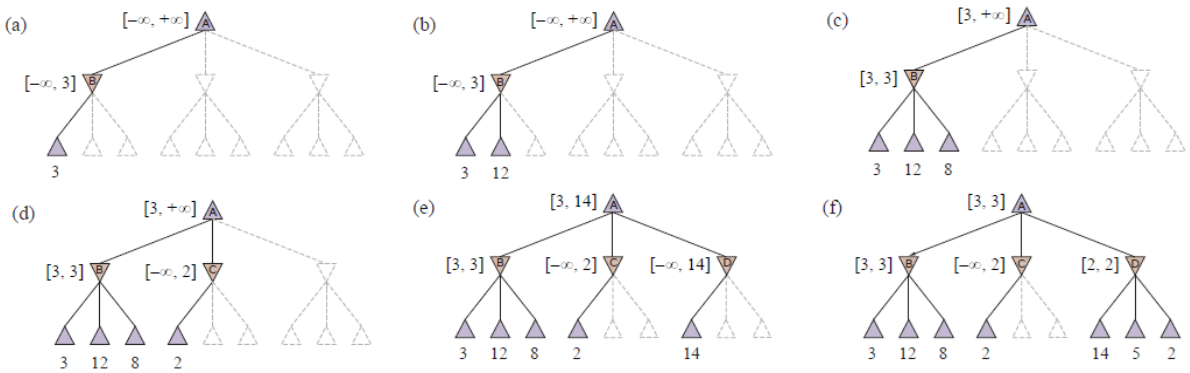
Alpha-Beta Pruning

O algoritmo Minimax, embora bom para alguns problemas, torna-se lento e encontra problemas quando utilizado em problemas com um grande espaço de busca. Assim, aplicam-se os cortes Alfa-Beta (Alpha-Beta Pruning) no Minimax de modo a ultrapassar estas dificuldades.

Este algoritmo baseia-se no facto de que alguns ramos do jogo não interferem no resultado final do jogo se o adversário jogar de forma ótima, pelo que podem ser ignorados/cortados. Assim, num nível de máximo (vez de MAX), diz-se alpha o melhor valor para MAX encontrado até agora no caminho atual, e qualquer valor encontrado que seja menor que este deverá levar ao corte do novo ramo. Num nível de mínimo (vez de MIN), diz-se beta o melhor valor para MIN encontrado até agora no caminho atual e qualquer valor encontrado que seja maior que este deverá levar ao corte do novo ramo.

Do mesmo modo que o Minimax, este algoritmo também recorre a depth-first search para explorar a árvore de jogadas. Assim, com *d* profundidade e *b* número de movimentos possíveis em cada ponto (branching factor), a complexidade espacial do Alpha-Beta Pruning é O(bd), enquanto a complexidade espacial é, na prática, $O(b^{d/2})$, mas poderá ser, no pior caso, $O(b^{d})$.

Temos, na imagem que se segue, um exemplo que permite melhor visualizar este algoritmo.



Monte Carlo Tree Search

Para problemas que não é prático o uso dos algoritmos anteriores, é comum o uso do Monte Carlo Tree Search.

Este método, contrariamente aos previamente analisados, não avalia os estados de acordo com uma evaluation function, mas sim através de uma série de simulações (“playout” ou “rollout”) de jogos completos a partir do estado, calculado uma utilidade média. Todavia, de modo à escolha do estado seguinte não ser totalmente aleatória, pode-se usar uma heurística local (“playout policy”) de modo a inclinar as escolhas para as mais indicadas. O número de “playouts” é definido e, ao longo do algoritmo, vão-se registando o número de vitórias que cada nó pode levar, criando uma probabilidade. O algoritmo passa pelos seguintes passos:

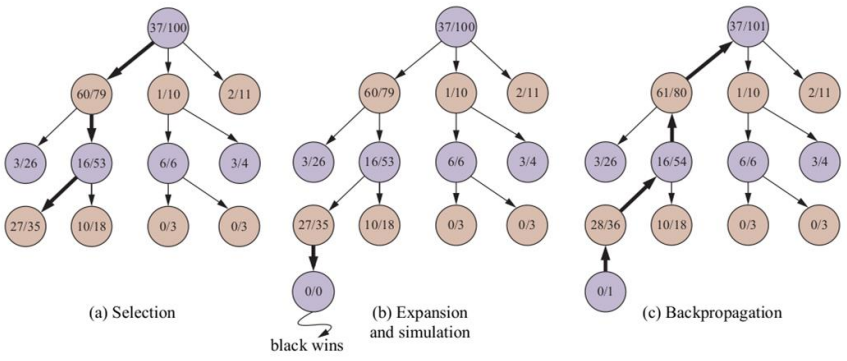
→Seleção. Iniciando na raiz da árvore, escolher uma jogada que leve a um novo estado, e repetir até se atingir uma folha da árvore.

→Expansão. Aumentar a árvore, gerando um filho do nó selecionado.

→ Simulação. Executar um “playout” a partir do filho que foi gerado, escolhendo jogadas para cada jogador de acordo com o “playout policy” (não memorizadas na árvore).

→ Back-Propagation. Utilizar o resultado da simulação para atualizar os nós até à raiz.

Temos, na imagem que se segue, um exemplo que permite melhor visualizar este algoritmo.



Connect 4

Connect Four (ou “4-in-line”, em português “4 em linha”) é um jogo para 2 jogadores (cada jogador com peças de uma cor) que decorre numa grade de dimensões 6x7, cujo objetivo de cada jogador é alinhar 4 das suas peças.

O jogo inicia com o primeiro jogador a colocar uma das suas peças em qualquer coluna, na fila mais abaixo de todas. O segundo jogador pode colocar a sua peça noutra coluna, ou na mesma que o seu oponente, ficando com a peça na fila diretamente acima desta (cada coluna só pode ter 6 peças). O turno muda e a situação repete-se, até se dar um empate (todas as colunas chegam ao máximo de peças possíveis) ou uma vitória (um jogador conseguiu alinhar 4 das suas peças, seja na diagonal, horizontal ou diagonal).

No que toca à escolhe da função de avaliação para a implementação deste jogo, seguimos os seguintes valores (positivos para X MAX, negativos para O MIN): uma vitória corresponde a 512 pontos, três peças alinhadas correspondem a 50 pontos, duas peças alinhadas correspondem a 10 pontos, e uma peça corresponde a 1 ponto.

Implementação dos Algoritmos

A linguagem escolhida para a realização deste trabalho foi a linguagem Python, tendo apenas sido utilizadas listas e arrays para a implementação dos algoritmos.

No que toca ao código, primeiro foram criadas diversas funções compartidas pelos diversos algoritmos, sendo estas: uma função que cria o tabuleiro do jogo; uma função que faz uma jogada nesse mesmo tabuleiro; uma função que indica todas as jogadas possíveis de; e um conjunto de funções que diz se um certo jogo já acabou e quem foi o vencedor.

As duas versões do algoritmo Minimax foram relativamente simples de implementar, sendo que a função de avaliação utilizada foi a mencionada no capítulo anterior (página 7, terceiro parágrafo).

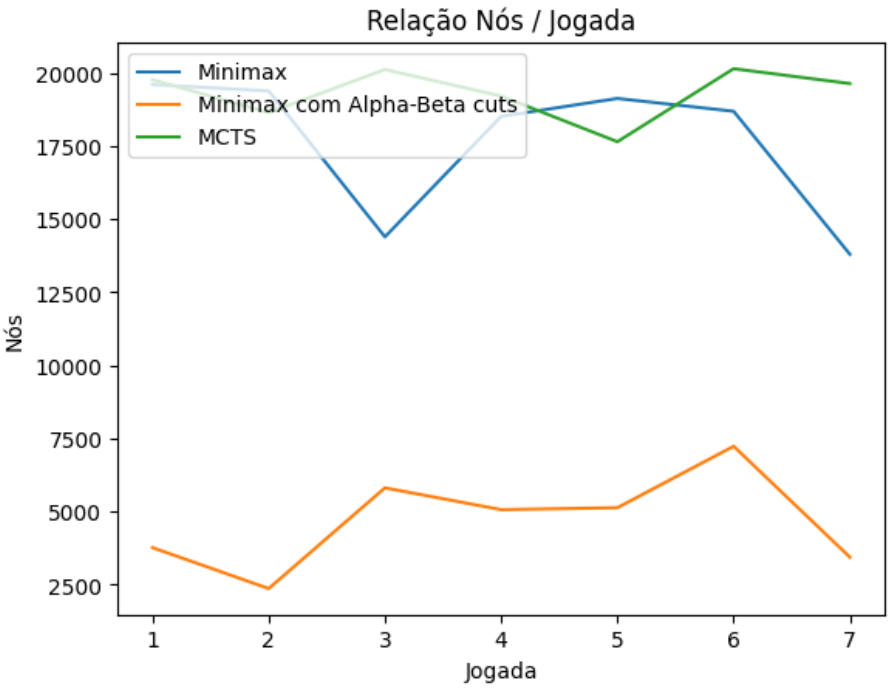
Foi imposto um limite de profundidade 5, dado ser a maior profundidade que o algoritmo Minimax consegue correr num tempo aceitável, para ambos os algoritmos, e para assim facilitar a comparação. Como se pode ver pelas tabelas a seguir, o Minimax com Alfa-Beta Cuts expande um número bem menor de nodes comparado ao Minimax.

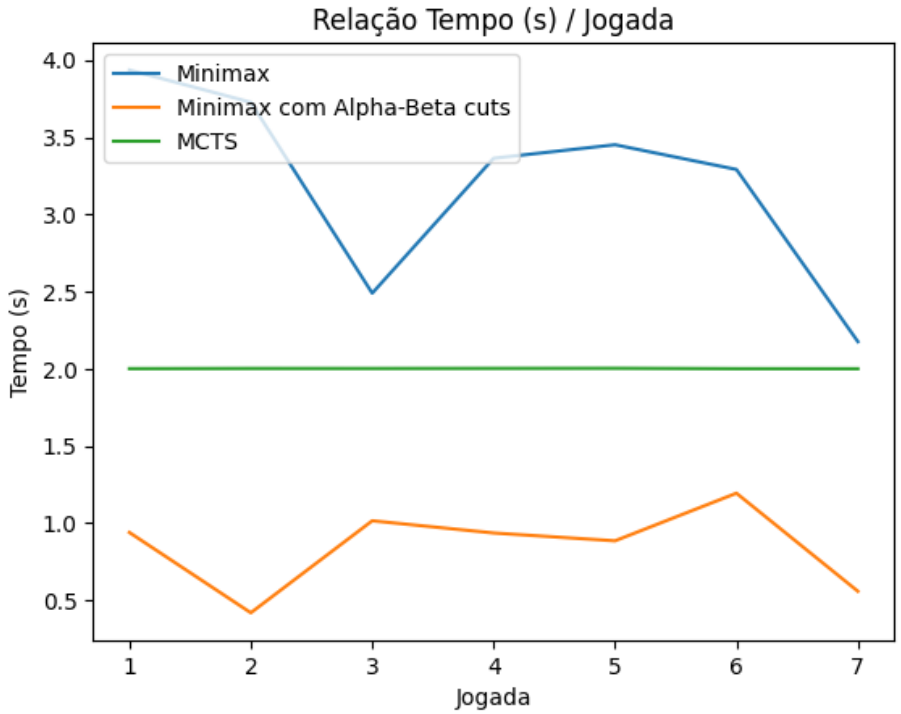
Já o MCTS, mesmo que não se tivesse de implementar uma função de avaliação, foi mais complicado de implementar. Primeiro teve-se de criar uma classe MTCS node que tinha como atributos o estado do tabuleiro, o state anterior a este, a jogada que deu origem a esse state, os states que surgem a partir deste node, o número de vezes que o node foi visitado, o número de vezes que o jogador venceu seguindo esse node, o jogador que vai jogar e as ações possíveis que não foram testadas a partir deste node.

De seguida teve-se de criar uma função para cada um dos passos do MTCS (seleção, expansão, simulação e Back-Propagation) por fim teve-se de criar um grupo de funções que escolhiam a jogada que tinha melhor pontuação de acordo com Upper Confidence Bound for Trees.

Este algoritmo expande quase o mesmo número de nós que o algoritmo Minimax com profundidade 5.

Temos, de seguida, duas tabelas que melhor permitem visualizar estas comparações entre algoritmos:





Conclusão e Comentários

No geral se tivermos em conta os dois algoritmos (o Minimax normal não conta para esta discussão dado que o Alfa-Beta Cuts é uma versão melhorada deste), o algoritmo que teve melhor performance foi o Minimax com alfa beta cuts, dado que para qualquer estado do jogo este algoritmo jogava sempre de maneira boa (mas não ótima dado que havia momentos que ele podia vencer, mas decide não fazer essa jogada ao contrario do MTCS que jogava maioritariamente de maneira ótima mas havia momentos que dava o jogo a perder, por exemplo, não impedia o adversário de vencer, isto é devido á natureza aleatória do MCTS).

Bibliografia

1. https://www.analyticsvidhya.com/blog/2019/01/monte-carlo-tree-search-introduction-algorithm-deepmind-alphago/

2. https://www.deepmind.com/research/highlighted-research/alphago

3. https://en.wikipedia.org/wiki/Connect_Four

4. https://moodle.up.pt/pluginfile.php/194243/mod_resource/content/1/ebin.pub_artificial-intelligence-a-modern-approach-global-edition-4nbsped-9780134610993-1292401133-9781292401133-9781292401171.pdf