Inteligência Artificial

CC2006

Jogos Com Oponentes

Catarina Monteiro - 202105279

Lara Sousa - 202109782

Mariana Serrão - 202109927

2022 / 2023

Índice

Índice de Ilustrações

Índice de Tabelas

Índice de Gráficos

1. Introdução

Em Inteligência Artificial, os jogos mais comuns são de um tipo especializado, denominados de jogos determinísticos para dois jogadores, com turnos, soma-zero e informação perfeita (tal como o xadrez). Isto significa que são jogos sem elementos de aleatoriedade, que envolvem competição pura, com valores de utilidade iguais e opostos (quando um jogador ganha, o outro perde), e em que cada jogador conhece todos os eventos que ocorreram anteriormente.

Para investigadores de IA, a natureza abstrata destes jogos torna-os num assunto curioso para estudo. O estado de um jogo é fácil de representar e os agentes são normalmente restritos a um pequeno número de ações, cujos resultados são definidos por regras precisas.

Assim, jogos com oponentes necessitam da capacidade de tomar alguma de decisão e, mesmo calculando a decisão ótima, é inviável que a escolha feita seja a correta. Para além disso, jogos penalizam gravemente a ineficiência. Foram, então, elaboradas uma série de ideias interessantes sobre como fazer o melhor uso possível do tempo.

As primeiras abordagens utilizadas para projetar um programa de jogo são baseadas num algoritmo de pesquisa em árvore, como o minimax, combinado com uma função de avaliação de estado de jogo artesanal baseada em conhecimento especializado.

No entanto, para muitos jogos, como o Hex ou o Go, abordagens baseadas em minimax, com ou sem aprendizagem computacional, falham em superar um humano. Duas causas foram identificadas: o elevado número de ações possíveis em cada estado do jogo; ou a impossibilidade de identificar uma função de avaliação suficientemente eficiente. Uma abordagem alternativa para a resolução destes dois problemas é denominada de Monte Carlo Tree Search (MCTS).

2. Algoritmo Minimax

O Minimax é um algoritmo de tomada de decisão utilizado na teoria dos jogos e em inteligência artificial. O objetivo do algoritmo é determinar a melhor jogada possível para um jogador, num determinado jogo com oponentes, ou seja, envolvendo dois jogadores, onde um tenta maximizar a sua pontuação e outro tenta minimizá-la.

Este algoritmo calcula a decisão Minimax do estado atual e explora todas as jogadas possíveis, que podem ser feitas por ambos os jogadores, criando uma árvore de jogo representativa de todos os possíveis resultados do jogo. Adicionalmente, atribui um valor de utilidade a cada nó da árvore de jogo, correspondente à vantagem adquirida pelo jogador, no nó onde joga. O jogador maximizador escolhe a jogada com o valor mais alto de utilidade e, pelo contrário, o jogador minimizador escolhe a jogada com o valor mais baixo de utilidade. A computação usada pelo Minimax é uma recursiva simples dos valores mínimos de cada estado sucessor que percorre toda a árvore até chegar às folhas. À medida que a recursão se desenrola, os valores Minimax são copiados através da árvore.

No exemplo ilustrativo da computação do Minimax (Figura 1), a recursão inicia-se pelos três nós inferiores esquerdos onde, recorrendo à função utilidade, se descobre os valores dos referidos: 3, 12 e 8, respetivamente. De seguida, elege-se o mínimo dos três, isto é, o mínimo entre 3, 12 e 8 é 3, retornando-o como valor de backup do nó B. Repete-se todo o procedimento para os nós seguintes, ficando o nó C e o nó D com o valor de backup 2. No fim, obtém-se o valor de backup 3 para o nó raiz derivado do máximo entre os valores de backup obtidos para o nó B, C e D.

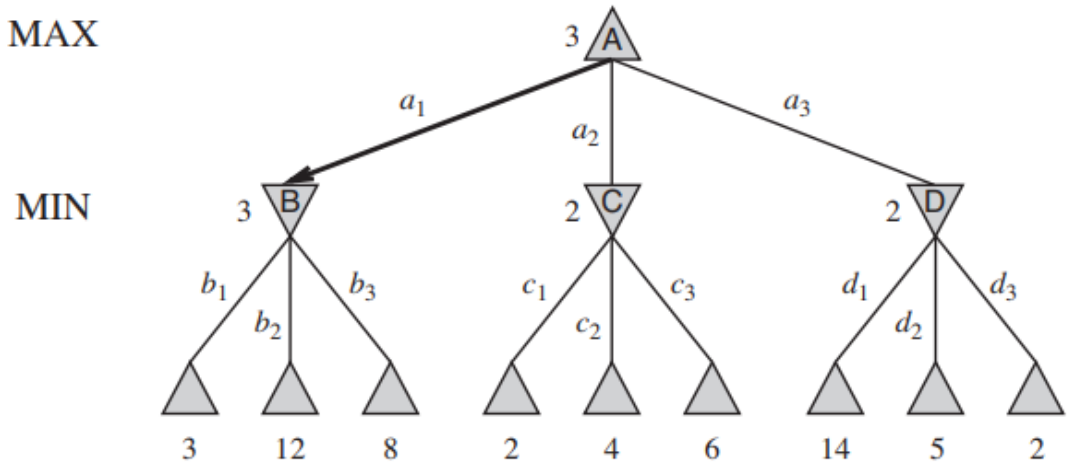


Figura 1 – Algoritmo Minimax

As complexidades temporal e espacial estão representadas na tabela a seguir, tendo em conta que m representa a profundidade máxima da árvore e b surge como os movimentos permitidos em cada ponto. Sabendo que o Minimax realiza uma exploração completa em profundidade da árvore do jogo, tem-se:

Tabela 1 - Complexidade do Minimax

Complexidade	Minimax que gera todas as ações de uma só vez	Minimax que gera ações, uma de cada vez
Temporal	O(b^m)	O(b^m)
Espacial	O(bm)	O(m)

O Minimax é praticável para inúmeros jogos, por exemplo o Jogo do Galo (Tic Tac Toe). Neste jogo, há três possibilidades de resultado: empate, ganha 'O' ou ganha 'X', onde para ganhar, têm de ficar todos em linha. Começamos por associar um número aos resultados possíveis, empate é 0, ganha 'O' é -1 e ganha 'X' é 1. Cada jogador tem um objetivo estabelecido: o max player ('X'), com o objetivo de maximizar a pontuação (preferencialmente 1 e, se não houver essa possibilidade, então 0); e o min player ('O'), cujo objetivo é minimizar o score (preferencialmente -1 e, se não for possível, 0).

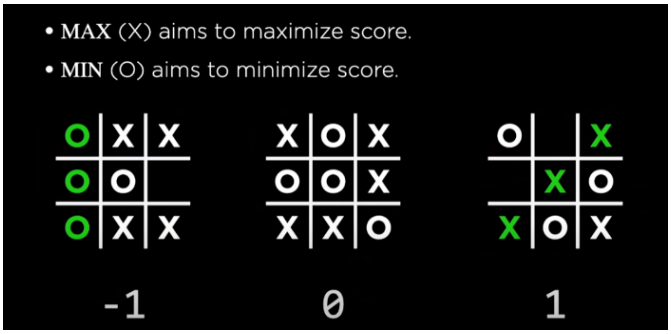


Figura 2 - Pontuação Jogo do Galo

Se os dois jogadores jogarem de forma otimizada, isto é, procurarem jogar sempre o melhor movimento possível, um computador, para determinar o valor, pensa sempre nas duas opções. Começando pelo min player ('O'), vê as opções de jogada e, a partir delas pensa no que o max player ('X') poderá fazer a seguir, de forma a escolher sempre o menor valor. A isto dá-se o nome de árvore de jogo, onde se explora todos os ramos possíveis, ou seja, todas as maneiras com que o jogo pode decorrer. O contrário acontece para o X.

Em resumo, o algoritmo Minimax considera todas as jogadas possíveis e depois, recursivamente, após fazer a melhor jogada possível, tenta saber qual a jogada que o oponente irá fazer em resposta e, sucessivamente, qual a jogada a fazer após a do oponente.

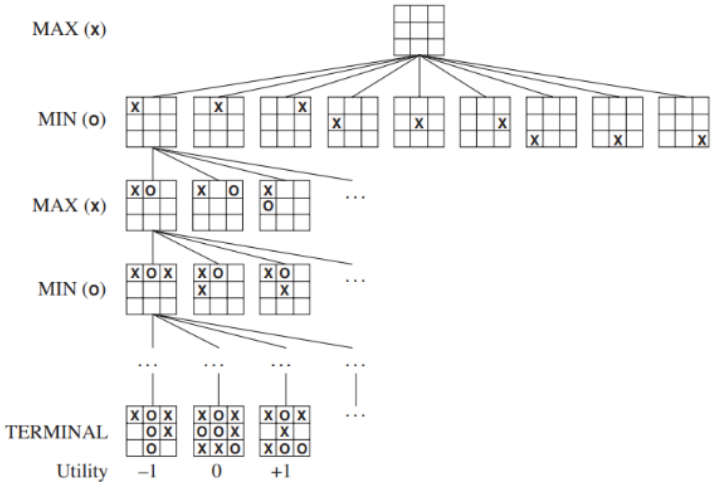


Figura 3 - Algoritmo Minimax no Jogo do Galo

No entanto surgem problemas: este algoritmo investiga todas as possibilidades de jogada, seguido de todas as possibilidades de jogada do oponente e, por conseguinte, as respostas possíveis, tudo recursivamente até se obter o fim do jogo. Pode levar demasiado tempo a explorar todas as jogadas possíveis e ser bastante demorado dependendo do quão complexo o jogo é. Por exemplo no xadrez, só depois dos primeiros 4 passos, tem 288000000000 jogadas possíveis e, se considerando um jogo complexo tem-se, em estimativa, 10^29000 jogadas possíveis, o que é demasiado para qualquer computador.

Assim, usar um Minimax que leve muito tempo, não será, portanto, tão eficaz na procura de todas as soluções num jogo de xadrez. No entanto não é impossível, uma vez que a 10 de fevereiro de 1996, o melhor jogador humano de xadrez, Garry Kasparov, foi derrotado pelo supercomputador da IBM, Deep Blue, no primeiro jogo de uma partida de seis jogos (os outros dois jogos dessa partida foram empate), ficando este jogo batizado como a primeira vez que um computador conseguiu vencer um humano num jogo de xadrez formal.

Figura 4 - Match Garry Kasparov vs Deep Blue

Deste modo, que mudanças se pode, então, fazer no algoritmo para que este seja mais rápido a procurar todas as possibilidades? Utilizar cortes Alfa-Beta seria uma boa escolha, tal como será destacado no capítulo 3.

Existem diversas modificações que podem ser aplicadas ao algoritmo Minimax. Uma delas é o Depth-limited minimax, onde consideramos uma função de avaliação (que estima a utilidade expectável do jogo para um determinado estado), onde, em vez de se procurar até ao final do jogo, procura apenas até um certo número de passos, levantando a pergunta “Neste nível de jogo quem é que aparenta ganhar?”. É necessário a criação de uma boa função de avaliação com capacidade para ser executada num jogo complexo e conseguir estimar o jogador potencialidade para ganhar.

Em conclusão, o Minimax é um método eficaz na tomada de decisões, em situações competitivas em que um jogador tenta superar outro.

3. Corte Alfa-Beta

Os cortes Alfa-Beta surgem como uma técnica complementar do algoritmo Minimax que visa melhorar a eficiência da procura em espaços de estados, em jogos com oponentes e com informações perfeitas. Indicada, por exemplo, para jogos como xadrez, damas e Go.

Esta técnica procede eliminando ramos específicos da árvore de pesquisa que não precisam de ser explorados, uma vez que, existem partes da árvore que podem ser cortadas, sem que o resultado da procura seja afetado. A procura inicia-se na raiz da árvore, a partir da qual o algoritmo avalia, para cada nó filho, o seu valor de acordo com uma função heurística, dependente da lógica do jogo.

Alfa representa o menor valor possível que o jogador Maximizador pode alcançar e Beta o maior valor que o jogador Minimizador pode alcançar. Quando o algoritmo percebe que o valor do nó atual não se encontra dentro do intervalo formado por Alfa e Beta, ou seja, não é necessário continuar a explorar os seus filhos, dá-se o corte desses ramos que se tornam então irrelevantes para a solução do problema.

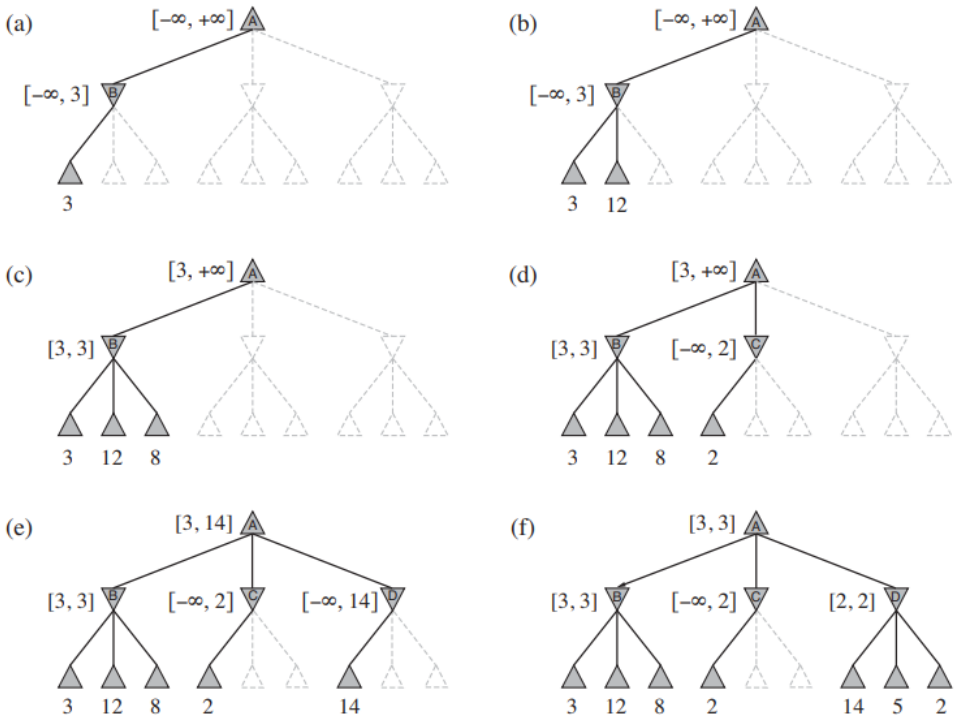


Figura 5 - Minimax com Cortes Alfa-Beta

A figura 5 representa as etapas da decisão ótima para a árvore de jogo onde, em cada alínea, surge também o intervalo de valores possíveis para cada nó. Na alínea a), é atribuído ao nó B (nó MIN) um valor de no máximo 3, visto que, a primeira folha abaixo de B tem o valor 3. Este valor não é alterado em b), pois, como a segunda folha em baixo de B tem um valor de 12, o MIN evitaria esse movimento. Pode-se inferir em c) que o valor da raiz é pelo menos 3, uma vez que MAX tem uma escolha de valor 3 na raiz e já se visitou todos os estados sucessores de B.

De seguida, em d), retrata-se um exemplo do corte Alfa-Beta, no qual, inicialmente é atribuído ao nó C, que é um nó MIN, o valor máximo de 2, derivado de a primeira folha abaixo de C ter o valor 2. No entanto, como B vale 3, MAX nunca escolheria C, pelo que, não faria sentido olhar para os outros estados sucessores de C.

Em e) não se efetua cortes, pois D vale no máximo 14, que é ainda a melhor alternativa para o MAX do que o 3 do nó anterior, então é necessário continuar a explorar os estados sucessores de D.

A partir deste passo foram impostos limites em todos os sucessores da raiz, atribuindo um valor máximo de 14 à raiz. Com a continuação da exploração, f), D fica com o valor 2, já que o segundo sucessor de D vale 5, o que obriga a continuar até ao terceiro sucessor de valor 2. Finalmente, a raiz fica com o valor de exatamente 3, já que o MAX faz a decisão de mover para B, que engloba o valor máximo entre B, C e D.

Com este exemplo, inferiu-se o caso geral para os cortes Alfa-Beta, ilustrado na figura 6. Se para o Player m for melhor que n, então nunca se chegará, no jogo, a escolher n.

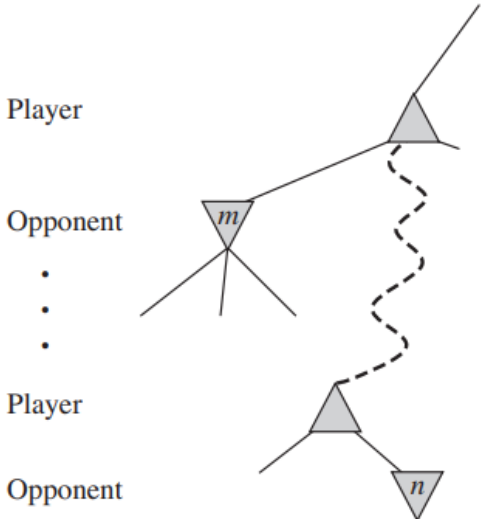
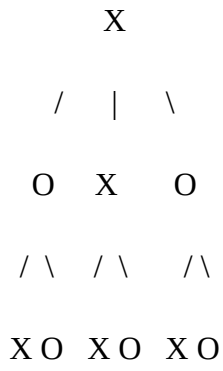


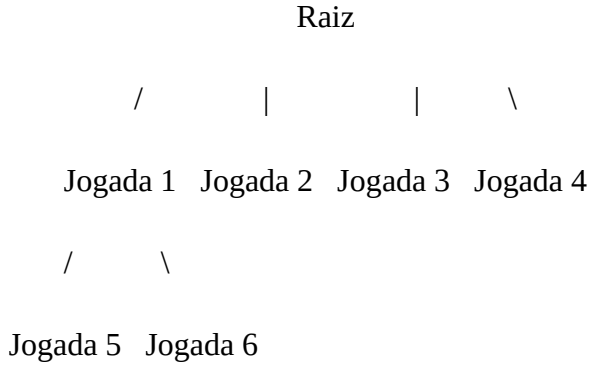
Figura 6 - Cortes Alfa-Beta

Considere-se o Jogo do Galo com profundidade 2, onde ‘X’ é o jogador Maximizador e ‘O’ é o jogador Minimizador.



Supondo que o valor da função heurística para ‘O’ é 0 e para ‘X’ é 1, o algoritmo começa por avaliar o primeiro nó ‘X’, seguido do primeiro nó filho ‘O’, e atualiza Beta para 0. Em seguida, o algoritmo avalia o segundo nó filho ‘X’ e atualiza Alfa para 1. Como Beta é menor que Alfa, o algoritmo pode fazer um corte neste ramo da árvore, pois não afeta o resultado da procura.

Considere-se, agora, um Jogo de Xadrez com uma profundidade 2, onde a raiz representa a posição atual no tabuleiro, e cada nó filho representa uma jogada possível que pode ser feita a partir dessa posição. Supondo que a função heurística do jogo de xadrez retorna um valor numérico que representa a avaliação da posição atual do jogo, o algoritmo de procura avalia cada nó filho da raiz, atualizando os valores Alfa e Beta.



Supondo que na jogada 1 o algoritmo encontra uma jogada que leva a uma posição avaliada em 10 pela função heurística, e na jogada 2 encontra uma jogada que leva a uma posição avaliada em 5, o valor de Alfa é atualizado para 10, que representa a maior pontuação encontrada até agora para o jogador Maximizador.

Ao avaliar o nó filho Jogada 3, o algoritmo percebe que o valor retornado pela função heurística para esta jogada é 3, que é menor do que o valor de Alfa. Assim, não é necessário explorar mais nós filhos da Jogada 3, já que estes não podem melhorar a pontuação do jogador Maximizador.

Na Jogada 4, o algoritmo encontra uma jogada que leva a uma posição avaliada em 8. Como o valor de Beta ainda não foi atualizado, o algoritmo atualiza o valor de Beta para 8, que representa a menor pontuação encontrada até agora para o jogador Minimizador.

Finalmente, ao avaliar os nós filhos de Jogada 5, o minimax encontra uma posição avaliada em 7, que é menor do que o valor de Alfa (10), e, portanto, não é necessário continuar a explorar os filhos na Jogada 5. No entanto, ao avaliar os filhos na Jogada 6, o algoritmo encontra uma posição avaliada em 11, que é maior que o valor de Beta (8). Assim, corta-se o ramo que se encontra abaixo da Jogada 6, já que este não afeta o resultado da procura.

De um modo geral, o algoritmo Minimax com Cortes Alfa-Beta é muito eficiente em jogos com muitas possibilidades de jogadas e estados, permitindo que os jogadores tomem decisões de forma rápida e eficiente. Reduz, significativamente, o tempo de execução do algoritmo de procura, tornando possível avaliar muitos mais nós da árvore num espaço de tempo limitado. Assim, surge como uma boa solução para o problema levantado no capítulo 2.

4. Árvore de Pesquisa Monte Carlo (MCTS)

O Monte Carlo Search Tree (MCTS) é um algoritmo de pesquisa utilizado em inteligência artificial para a tomada de decisões em jogos e outras aplicações similares. Utiliza simulações aleatórias para avaliar a qualidade de uma jogada num determinado estado de jogo.

O MCTS começa com um estado inicial do jogo e constrói uma árvore de pesquisa através de simulações de jogadas. Cada nó da árvore representa um estado do jogo e cada aresta representa uma jogada possível. O algoritmo avalia os nós da árvore através de um processo de simulação de jogadas chamado de "rollout". Durante este processo, o algoritmo realiza uma sequência aleatória de jogadas a partir do nó em questão e avalia o resultado da simulação. Este resultado é utilizado para atualizar as informações de qualidade dos nós da árvore.

A partir da árvore de pesquisa construída, o MCTS pode selecionar a jogada mais promissora para o jogador atual. A escolha é feita através de um processo chamado "seleção", que procura os nós da árvore que parecem mais promissores, de acordo com as informações de qualidade que foram atualizadas durante as simulações.

O MCTS usa a fórmula Upper Confidence Bound (UCB), que equilibra a exploração e explotação, na etapa de seleção para percorrer a árvore. A fórmula UCB calcula o valor de um nó adicionando uma penalidade à média do valor estimado do nó e uma recompensa baseada na incerteza do valor estimado. Essa penalidade incentiva a exploração de nós que ainda não foram muito visitados, enquanto a recompensa incentiva a explotação de nós com altos valores estimados. A fórmula UCB é definida como:

$$UCB_i = V_i + C * \sqrt{\ln(N) / n_i}$$

onde UCB_i é o valor UCB do nó i, V_i é a média de vitórias no nó i, C é uma constante que ajusta o trade-off entre exploração e explotação, N é o número total de simulações realizadas até o momento e n_i é o número de vezes que o nó i foi visitado. Quanto maior o valor de UCB_i, mais interessante é o nó i para ser visitado.

O processo de seleção é seguido por uma etapa chamada "expansão", em que um novo nó é adicionado à árvore para representar uma nova jogada possível. Depois disso,

é efetuado o passo “simulação”, que simula diferentes jogadas até atingir um determinado limite de tempo ou profundidade.

Após a simulação, o resto da árvore deve ser atualizado. É então realizado o processo “retropropagação”, onde é retropropagado do novo nó até o nó raiz. Durante o processo, o número de simulações armazenado em cada nó é incrementado. Além disso, se os resultados da simulação do novo nó resultarem numa vitória, o número de vitórias também é incrementado.

O método de Monte Carlo é utilizado em diversos jogos que envolvam incerteza ou aleatoriedade, incluindo jogos de tabuleiro e jogos de cartas, como o Go, Xadrez e Poker, bem como em outras aplicações, como o planeamento de rotas em veículos autónomos e em robótica.

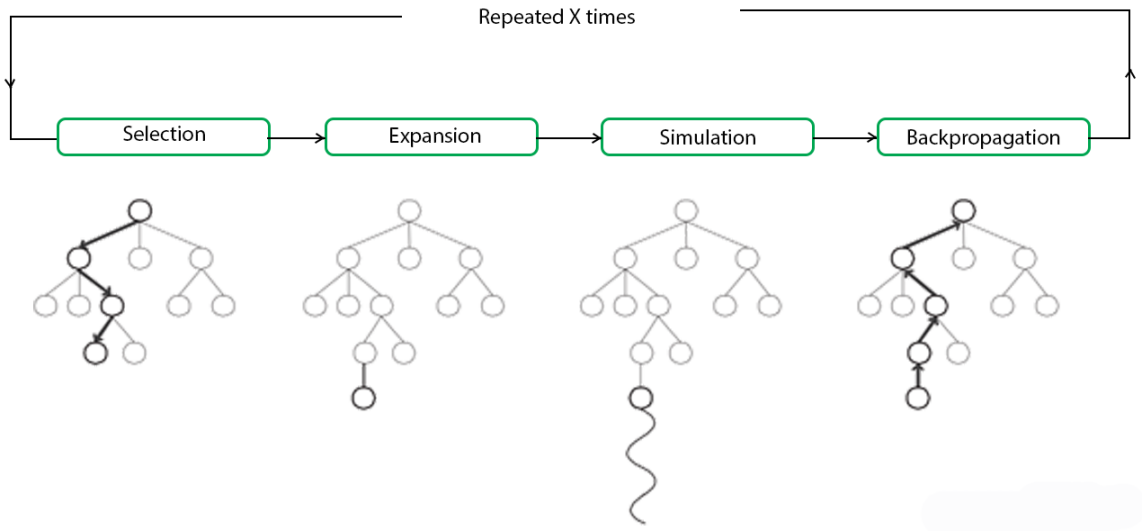


Figura 7 - Fases da MCTS

A complexidade temporal do MCTS é O(C*N), onde C é o número de simulações por nó e N o número de nós. Para muitas implementações, o número de simulações é fixo e, portanto, a complexidade temporal é linear, O(N). No entanto, quando o número de simulações é variável, a complexidade temporal pode ser exponencial, ou seja, O(C^N).

A complexidade espacial é O(N), uma vez que a maioria das implementações armazenam a árvore na memória, e a quantidade de memória necessária para armazenar cada nó é constante. No entanto, em algumas implementações, pode haver uma sobrecarga de memória para armazenar informações adicionais, como avaliações heurísticas., tornando a complexidade espacial maior do que O(N).

5. Quatro em linha

Quatro em Linha, ou Connect Four, trata-se de um jogo de estratégia para dois jogadores numa grade com 7 colunas e 6 linhas, totalizando 42 espaços. Cada jogador está associado a uma cor e, por sua vez, joga um disco da sua cor numa coluna à sua escolha. O disco cai, então, para a célula vazia mais abaixo da coluna. O primeiro jogador a fazer um alinhamento de quatro discos consecutivos ganha, seja verticalmente, horizontalmente ou diagonalmente. Se o tabuleiro estiver completamente cheio sem qualquer alinhamento formado, então o jogo termina empatado.

Trata-se de um jogo para dois jogadores com informações perfeitas para ambos os lados, o que significa que nada está escondido de ninguém. O Quatro em Linha também pertence à classificação de jogo adversário de soma-zero, uma vez que a vantagem de um jogador é a desvantagem do seu oponente.

Uma medida da complexidade do jogo Connect Four é o número de posições possíveis no tabuleiro do jogo. Para o clássico jogo, existem 4.531.985.219.092 posições para todos os tabuleiros preenchidos com 0 a 42 peças.

Foi resolvido pela primeira vez em 1988 por James Dow Allen e independentemente por Victor Allis, que descrevem estratégias vencedoras nas suas análises. No período das soluções iniciais para o Quatro em Linha, a análise de força bruta não era considerada viável devido à complexidade do jogo e à tecnologia de computador disponível na época.

Desde então, o Quatro em Linha foi resolvido precisamente com este método, força bruta. Os algoritmos de inteligência artificial capazes de resolver fortemente o Connect Four são o minimax, com otimizações que incluem corte alfa-beta, ordem de movimento e tabelas de transposição, assim como a árvore de pesquisa Monte Carlo.

A conclusão resolvida para o Quatro em Linha é que o primeiro jogador é o que pode vencer. Com um jogo perfeito, o primeiro jogador pode forçar uma vitória, na 41º jogada ou antes, começando na coluna do meio. O jogo é um empate teórico quando o primeiro jogador começa nas colunas adjacentes ao centro. Começando com as quatro colunas externas, o primeiro jogador permite que o segundo jogador force uma vitória.

5.1. Algoritmo Minimax

O algoritmo minimax aplicado ao jogo Quatro em Linha avalia todas as jogadas possíveis, constrói uma árvore de jogadas e escolhe a melhor jogada para o jogador que está a jogar e a pior para o adversário. Este processo é realizado em diferentes níveis na árvore, onde um nível representa o jogador atual e o nível seguinte representa o jogador adversário. Cada nó da árvore é avaliado com um valor de utilidade e, no final, a jogada com a maior pontuação é selecionada.

A implementação é feita em três funções principais: a minimax, a máximo e a mínimo.

A função minimax é chamada para encontrar a melhor jogada possível. Por sua vez, chama a função máximo para o jogador atual e retorna a jogada correspondente.

A função máximo é a que representa o jogador atual e encontra a jogada que leva ao melhor resultado possível para o jogador. Para isso, percorre todas as jogadas possíveis, cria cópias do tabuleiro e chama a função mínimo para cada uma dessas jogadas. Por fim retorna a jogada correspondente com a maior pontuação.

A função mínimo representa o adversário e encontra a jogada que leva ao pior resultado possível para o jogador atual. Para fazer isso, percorre todas as jogadas possíveis, cria cópias do tabuleiro e chama a função máximo para cada uma dessas jogadas. Ao contrário da função máximo, retorna a jogada com a menor pontuação.

Estas funções utilizam outras funções auxiliares para verificar se o jogo acabou, contar pontos (função de avaliação), copiar o tabuleiro e realizar movimentos no tabuleiro. A dificuldade do jogo é controlada pela variável dificuldade, que representa a profundidade máxima da árvore de jogadas que o algoritmo deve considerar.

5.2. Corte Alfa-Beta

O algoritmo minimax com cortes alfa-beta é uma otimização do algoritmo minimax convencional. A ideia básica é que se corta os ramos da árvore de jogadas que não necessitam de ser explorados, reduzindo assim o número de avaliações de jogadas que precisam ser feitas.

No código implementado, isto é feito com a função maximo_alphabeta e minimo_alphabeta. O alfa representa a melhor escolha encontrada até o momento para o jogador atual, enquanto o beta representa a melhor escolha encontrada até o momento para o jogador adversário. Inicialmente, o alfa é definido como o pior resultado possível (menos infinito) e o beta como o melhor resultado possível (mais infinito).

Durante a pesquisa, quando um nó é expandido, se o valor desse nó for menor do que o alfa, então todos os nós filhos desse nó podem ser ignorados, uma vez que o adversário nunca escolheria essa jogada. Por outro lado, se o valor desse nó for maior do que o beta, então todos os nós filhos desse nó também podem ser ignorados, sendo que o jogador atual nunca escolheria essa jogada.

Na função maximo_alphabeta, se o valor do filho atual que está sendo avaliado for maior que o valor máximo, então o valor máximo é atualizado para o valor atual. Em seguida, alfa é atualizado para o máximo valor entre alfa e o valor máximo guardado anteriormente. Se alfa for maior ou igual a beta, o ciclo é interrompido, uma vez que todos os filhos podem ser ignorados.

O mesmo processo é repetido na função minimo_alphabeta, mas com a diferença de que o beta é atualizado em vez do alfa.

Desta forma, o algoritmo minimax com cortes alfa-beta pode reduzir significativamente o número de avaliações de jogadas necessárias para encontrar a melhor jogada possível.

5.3. Árvore de Pesquisa Monte Carlo (MCTS)

Para adaptar a Árvore de Pesquisa Monte Carlo ao jogo dos Quatro em Linha, começou-se com a definição da classe Node, que representa um nó na árvore de pesquisa. Cada nó armazena informações sobre um determinado estado do jogo, incluindo a sua pontuação / quantidade de vitórias (Q), número de simulações (N) e os seus filhos (outros nós). Também contém a função UCB (Upper Confidence Bound), que é a fórmula para determinar o melhor nó a ser explorado na fase de seleção do algoritmo.

Em seguida, construiu-se a classe MCTS, que é a classe principal do algoritmo. Esta armazena informações sobre o estado do jogo, como a raiz da árvore de busca (self.raiz) e o número de simulações a serem realizadas (self.num_rollouts). Além disso, contém as quatro fases principais do algoritmo: seleção, expansão, simulação e retropropagação.

A função de seleção escolhe o melhor nó a ser explorado a partir da raiz da árvore de busca. A função de expansão adiciona novos nós (representando novas jogadas possíveis) à árvore de busca. A função de simulação realiza uma jogada aleatória a partir de um determinado nó, simulando o resto do jogo.

No entanto, é importante destacar que, na simulação, o número de jogadas realizadas é limitado de acordo com a dificuldade escolhida pelo jogador. Isto é feito para evitar que o algoritmo gaste muito tempo a simular jogos muito longos e, assim, tornar a escolha da jogada mais rápida. Ou seja, se a dificuldade escolhida for fácil, a simulação é limitada a um número menor de jogadas em relação à dificuldade média ou difícil.

Por fim, a função de retropropagação atualiza as pontuações dos nós da árvore de busca com base no resultado da simulação. A classe MCTS apresenta também uma função para determinar a melhor jogada possível a partir da raiz da árvore de busca (self.best_move).

5.4. Resultados

Foram testadas as três estratégias, Minimax, Minimax com Cortes Alfa-Beta e Árvore de Pesquisa Monte Carlo, no modo de jogo Computador vs. Computador, na dificuldade 1, na qual se fez todas as possibilidades de jogo entre estas. Na seguinte tabela é possível verificar o tempo gasto para cada jogo, a quantidade de memória do CPU e RAM utilizada, assim como o resultado (qual foi o vencedor, 1 ou 2) e a quantidade de jogadas efetuadas.

Tabela 2 - Computador vs. Computador

Jogador 1	Jogador 2	Tempo (seg)	Espaço CPU (%)	Espaço RAM (%)	Vencedor	Nº de Jogadas
Minimax	Minimax	1,27	2,4	80,9	1	7
	Cortes Alfa-Beta	1,07	3,4	88	1	7
	Monte Carlo	1,21	1,3	87,2	2	30
Cortes Alfa-Beta	Minimax	1,05	2	81,8	1	7
	Cortes Alfa-Beta	1,07	3,1	82	1	7
	Monte Carlo	1,12	1,9	81,5	1	23
Monte Carlo	Minimax	1,06	1,5	81,5	1	7
	Cortes Alfa-Beta	1,04	2,3	81,5	1	7
	Monte Carlo	1,21	2,3	82	2	32

O gráfico abaixo demostra a variação do tempo, tendo em conta a jogada realizada.

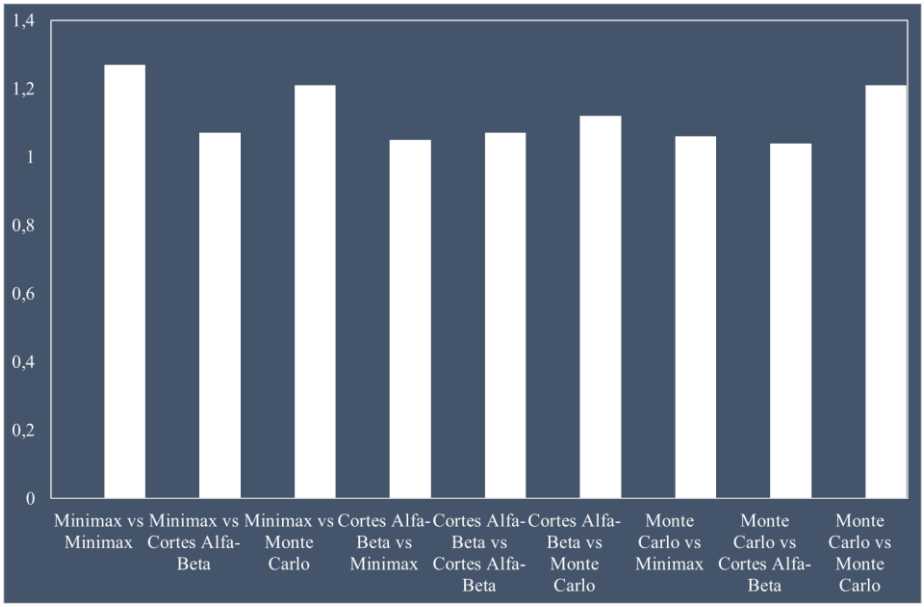


Gráfico 1 - Tempo de Execução da Jogada

Por outro lado, foi também realizado um gráfico onde se pode analisar a quantidade de jogadas efetuadas em cada jogo.

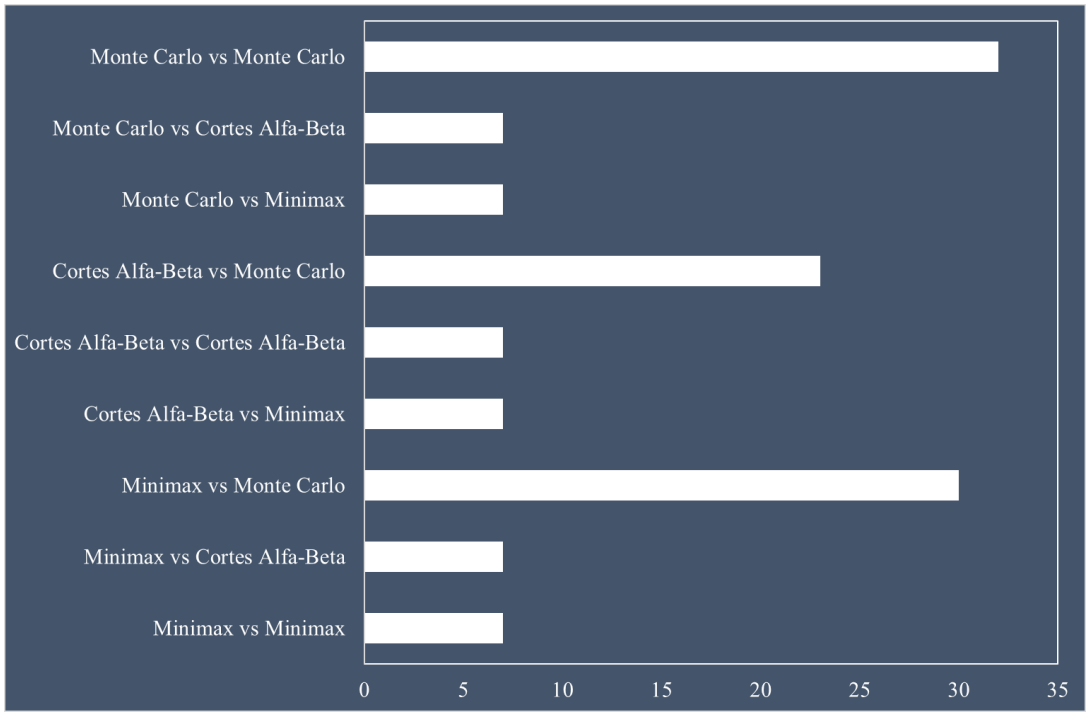


Gráfico 2 - Quantidade de Jogadas Efetuadas

Por fim, realizou-se também uma tabela onde se compara, para a jogada Minimax vs. Minimax, a alteração dos dados consoante a dificuldade do jogador.

Tabela 3 - Jogada Minimax nas Diferentes Dificuldades

Dificuldade 1	Dificuldade 2	Tempo (seg)	Espaço CPU (%)	Espaço RAM (%)	Vencedor	Nº de Jogadas
Fácil	Fácil	1,27	2,4	80,9	1	7
	Intermédio	1,1	1,5	80,3	1	7
	Difícil	1,35	4,6	81,4	1	7
Intermédio	Fácil	1,1	1,8	78,8	1	7
	Intermédio	1,4	1,7	78,5	1	7
	Difícil	1,3	2,3	78,7	1	7
Difícil	Fácil	1,5	2,4	78,5	2	10
	Intermédio	1,5	4,1	80,2	2	10
	Difícil	2,4	3,3	78,6	2	26

6. Comentários Finais e Conclusões

Após uma análise detalhada dos resultados acima enumerado chegou-se a várias conclusões.

Nos modos de jogo que não envolvem um jogador humano encontraram-se algumas disparidades, como por exemplo no número de jogadas necessárias para terminar o jogo: nos casos em que um dos jogadores utiliza o algoritmo Monte Carlo, o número de jogadas aumenta consideravelmente. Também este algoritmo revela o maior tempo de execução.

Analisando a tabela 2, conseguimos ter uma visão global das diferenças entre métodos. Na grande maioria das vezes, o primeiro a jogar é quem vence a partida, não se encontrando nenhum padrão influenciado pelo algoritmo a uso. Conseguimos também realizar um rank quanto aos algoritmos baseado na quantidade de vezes que cada algoritmo ganhou/perdeu:

Monte Carlo (em 4 jogos ganhou 3 deles)

Cortes Alfa-Beta (em 4 jogos ganhou 1 deles)

Minimax (em 4 jogos ganhou 1 deles)

Após esta análise conseguimos concluir que existem algumas incoerências nos resultados, como por exemplo o facto do algoritmo Cortes Alfa-Beta não ter um melhor desempenho do que o algoritmo Minimax. Também o resultado dos níveis de dificuldade não foi o pretendido, ou seja, em todos os casos em que dois níveis de dificuldade se cruzavam, sendo um deles o nível “difícil”, este perdia sempre. Algo semelhante se sucede quando os níveis “intermédio” e “fácil” se confrontam.

Em suma, podemos afirmar que existem algumas falhas na implementação do código, no entanto, consideramos que o balanço final foi positivo sendo o jogo dinâmico e interativo. Uma das possíveis falhas pode ter tido a ver com a profundidade a ser usada, isto é, os níveis de dificuldade poderiam ter sido implementados de modo a pesquisarem até um ponto mais fundo na árvore.

7. Referências Bibliográficas

Artificial Intelligence: Foundations of Computational Agents Book by Alan Mackworth and David Lynton Poole (2010).

Artificial Intelligence: A Modern Approach, 3rd US ed. by Stuart Russell and Peter Norvig (2009).

Medium (11/10/2019). *How AI Decides in a Two-Player Game by Rean Neil Luces.* Consulta realizada a 17 de março de 2023, a partir de *https://medium.datadriveninvestor.com/how-ai-decides-in-a-two-player-game-a51bc21b7fe7*

Quentin Cohen-Solala (12/10/2021). *Learning to Play Two-Player Perfect-Information Games without Knowledge.* Consulta realizada a 17 de março de 2023, a partir de https://arxiv.org/pdf/2008.01188.pdf

Luke Kim, Hormazd Godrej, Chi Trung Nguyen (2019). *Applying Machine Learning to Connect Four*. Consulta realizada a 17 de março de 2023, a partir de http://cs229.stanford.edu/proj2019aut/data/assignment_308832_raw/26646701.pdf

Wikipedia (01/03/2023). *Connect Four*. Consulta realizada a 17 de março de 2023, a partir de https://en.wikipedia.org/wiki/Connect_Four

GeeksforGeeks (05/07/2022). *ML | Monte Carlo Tree Search (MCTS)*. Consulta realizada a 16 de abril de 2023, a partir de https://www.geeksforgeeks.org/ml-monte-carlo-tree-search-mcts/

CS50 (31/12/2020). *CS50 2020 - Artificial Intelligence*. Consulta realizada a 16 de abril de 2023, a partir de https://www.youtube.com/watch?v=eey91kzfOZs

Wikipedia (28/10/2021). *Match Garry Kasparov vs Deep Blue*. Consulta realizada a 16 de abril a partir de https://pt.wikipedia.org/wiki/Match_Garry_Kasparov_vs_Deep_Blue