Inteligência Artificial (CC2006) 2021/2022

Relatório de Inteligência Artificial

Beatriz Sameiro da Rocha – 202006133

Beatriz Soares Pereira - 202007190

João Fernandes Azevedo - 202008367

1. Introdução

Segundo a teoria da matemática, jogos são ambientes multiagentes, nos quais cada agente precisa considerar o impacto das ações de cada agente e como elas afetam o outro, independentemente se estes são cooperativos ou competitivos. Contrariamente à procura de busca, existe uma imprevisibilidade entre estes agentes que podem introduzir casualidades no processo de resolução de problemas. Desta forma, estes ambientes competitivos, nos quais os objetivos dos agentes estão em conflito, dão origem a problemas de busca adversária.

Em Inteligência artificial, os jogos mais comuns são os jogos determinísticos, de tomada de turnos e de soma zero de informação perfeita (exemplo: xadrez, jogo do galo, 4 em linhas, etc.) para dois jogadores. Insto é, aqueles em ambientes determinísticos totalmente observáveis, nos quais dois agentes agem alternadamente e nos quais os valores de utilidade no final do jogo são sempre iguais e opostos. Por exemplo, se um jogador ganha, o outro jogador necessariamente perde. Esta oposição entre as funções de utilidade dos agentes torna a situação adversa.

1.1. Técnicas necessárias para obter a melhor solução e elementos de busca do jogo

Começamos com uma definição da melhor escolha de movimento e um algoritmo para encontrá-lo. Para selecionar este algoritmo é necessário escolher a melhor solução em tempo limitado. Para isso, podemos utilizar as seguintes técnicas:

- Pruning: permite ignorar partes da árvore de busca que não fazem diferença para a escolha final
- Funções de avaliação heurística: permitem aproximar a verdadeira utilidade de um estado sem fazer uma busca completa

Consideramos jogos com dois jogadores, a quem chamamos de MAX e MIN. MAX move -se primeiro, e em seguida cada um deles move-se em turnos até o jogo acabar. No final do jogo, são atribuídos pontos ao vencedor enquanto o perdedor é penalizado. Um jogo pode ser formalmente definido com os seguintes elementos:

- Estado inicial: indica como o jogo é configurado no início
- Jogador: define qual jogador tem o turno atual para poder mover
- Ações: retorna um conjunto de movimentos válidos num estado
- Resultado: resultado de um movimento
- Teste de terminação: verifica se o jogo acabou ou não. Os estados em que o jogo termina são denominados de estados de terminação
- Função utilidade (utility function): também conhecida como objective function e payoff function, define o valor numérico final para um jogo que termina no estado termina. Os valores podem ser -1 se o jogador perder, 0 caso houver um empate ou +1 se o jogador vencer. Alguns jogos possuem maior variedade de resultados possíveis. Um jogo de soma zero é definido como aquele em que o pagamento total para todos os jogadores é o mesmo para todas as instâncias do jogo.

Vamos perceber a funcionalidade dos elementos e da árvore de jogo através da Figura 1 onde se demonstra um exemplo de parte de uma árvore de jogo, uma árvore onde os nós são estados de jogo e as arestas são movimentos, configurada para o popular jogo do galo.

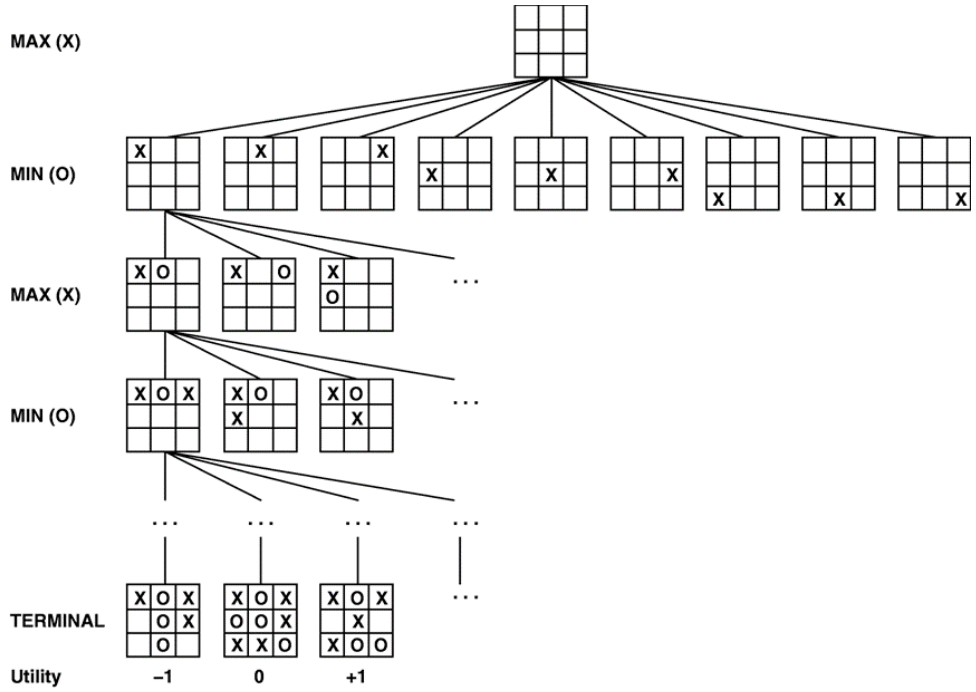


Figura 1 – Exemplo de parte de uma árvore de jogo para o jogo do galo

O nó superior é o estado inicial e MAX move-se primeiro, colocando um X em um quadrado vazio, tendo estes nove movimentos possíveis. O jogo alterna entre MAX colocando uma cruz (X) e MIN colocando um zero (O) até chegarmos aos nós folha correspondentes aos estados terminais, de modo que um jogador tenha três em uma linha (existe um vencedor) ou todos os quadrados são preenchidos (houve um empate). O número em cada nó folha indica o valor da utilidade do estado terminal do ponto de vista do MAX; valores altos são considerados bons para MAX e maus para MIN. Mas, independentemente do tamanho da árvore do jogo, é trabalho do MAX procurar uma boa jogada.

1.2. Tipos de algoritmos na pesquisa adversa

Numa busca normal, seguimos uma sequência de ações para atingir o objetivo ou terminar o jogo de forma otimizada. Mas em uma busca adversária, o resultado depende dos jogadores que decidirão o resultado do jogo. Além disso, é evidente que a solução para o estado objetivo será uma solução ótima porque o jogador tentará vencer o jogo pelo caminho mais curto e em tempo limitado. Existem os seguintes tipos de busca adversária:

- Algoritmo Minimax
- Alfa-beta pruning

2.Minimax

O Algoritmo minimax é usado principalmente para jogos de dois jogadores. Este tem como objetivo dar uma jogada ótima assumindo que o jogador adversário também vai jogar otimamente.

No minimax o jogador, que é ajudado, é conhecido como maximizador, pois o algoritmo tenta maximizar as possibilidades de vitoria deste jogador, e o seu adversário é o minimizador, pois o objetivo do algoritmo é minimizar as suas possibilidades de ganhar o jogo (minimizando assim as suas chances de vencer) (1).

No minimax cada tabuleiro tem um valor associado, se o maximizador estiver em vantagem o valor deste tabuleiro tenderá para ser positivo, se o contrário acontecer então o valor vai tender para o negativo, estes valores são calculados por heurísticas, únicas para cada jogo. Assim o minimax avalia todos os caminhos que pode ir tendo em conta também, que o outro jogador irá jogar de forma ótima, assim quando percorre a arvore terá em consideração, que o minimizador vai escolher o menor valor, usando isto para ir no caminho com o maior valor para a jogado do minimizador.

O valor associado a cada tabuleiro é chamado de valor máximo e é dado por

$$vi = \max_{ai} \min_{a-i} \quad vi(ai, a-i) \text{(2)}$$

Vi é o valor da função

i é o índice do maximizador

-i é o índice do minimizador

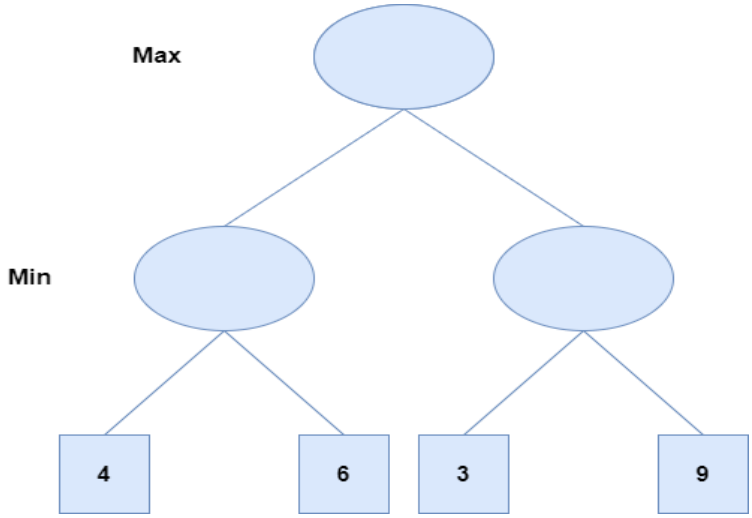
ai ação feita pelo maximizador

a-i ação feita pelo minimizador

Para o cálculo do valor máximo, são vistas todas as ações possíveis para o minimizador e é determinada a pior combinação (a que dá ao maximizador o menor valor) e assim é decido que jogada vai trazer ao maximizador o menor valor mais alto possível.

O valor minimax ,de onde este algoritmo ganha o seu nome, é o menor valor que o minimizador pode forçar o maximizador a receber, sem ter noção das jogadas do maximizador. O seu cálculo e semelhante ao do valor máximo, apenas mudando a ordem dos operadores max e min.

Vamos perceber melhor com esta arvore de exemplo:



Neste exemplo como minimax é um algoritmo que usa backtracking , por isso tenta todos os movimentos e depois retrocede e toma uma decisão.(3)

Assim:

- Quando o maximizador vai para a esquerda, assim o minimizador vai poder escolher entre 4 e 6, e este provavelmente escolherá o 4.
- Se maximizador seguir pela direita o minimizador vai poder escolher entre 3 e 9 e provavelmente escolhera o 3.
- Assim como o 4 > 3 o algoritmo escolherá seguir pela esquerda.

3.Alpha-Beta Prunning

O Alpha-Beta Prunning é considerado uma otimização do algoritmo falado em cima, o minimax.

Neste novo algoritmo o pior cenário a resolução será igual minimax e no melhor cenário conseguirá chegar ao melhor caminho podando os outros caminhos.

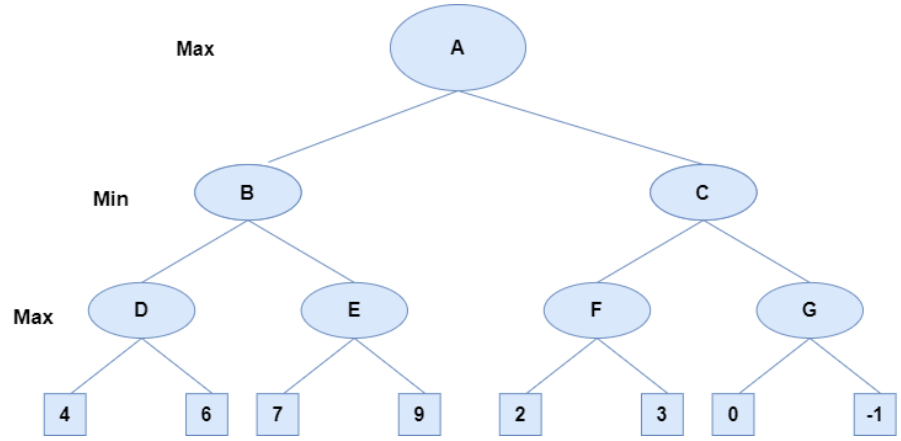
No Alpha-Beta Prunning temos 2 parâmetros, em vez de um, denominados por alpha e beta.

Alpha : é usado e mudado apenas pelo maximizador, e representa o maior valor encontrado até agora, o seu valor inicial é de -∞.

Beta: é usado e mudado apenas pelo minimizador, e que representa o menor valor encontrado até agora, o seu valor inicial é ∞.

Ambos os valores assumem o valor do nó filho enquanto o verdadeiro valor é usado recursivamente (4).

Assim pra descobrir que ramos são desnecessários este usa a condição alpha >= beta.



Para perceber melhor o algoritmo vamos perceber como é que ele percorre a arvore acima(5):

- O maximizador que escolherá ir para B primeiro, depois o minimizador escolherá o D primeiro, depois vai pela esquerda e encontra o valor de 4, assim o beta=+INF e o alpha=4, avaliando o nó da direita vê o valor de 6, assim como 6>4 o alpha passa a ser 6 assim D retorna o valor de 6 para o B.
- No B o beta assume o valor de 6 enquanto o alpha mantém-se -INF, e passa esses valores para o E. Aqui vai se olhar para o filho da esquerda que é o 7 por isso o alpha assume este valor. Como alpha>=beta o programa para e retorna 7 para B. Neste caso o algoritmo não vai olhar para o nó da direita de E poupando tempo de computação.
- Agora B tem o valor de 5 pois 5<6.
- Usando a mesma logica para o caminho em C, onde os valores são de alpha=5 e Beta=+INF, este vai seguir e passar estes valores para F, na esquerda este vê o número 2 e como 5>2 o alpha mantem-se 5, na direita este encontra o valor de 3, mantendo o alpha 5, como 3>2 o F retorna 3 para o C onde o Beta passa a assumir este valor.
- Como agora em C o alpha>=beta o programa para e não avalia G.
- Assim C assume o valor de 2.
- Quando B e C retornam, respetivamente, 5 e 2, para o A este vai escolher o maior entre estes número que, neste caso é o 5.
- **Logo o programa conclui que 5 é o valor ótimo para o maximizador.**

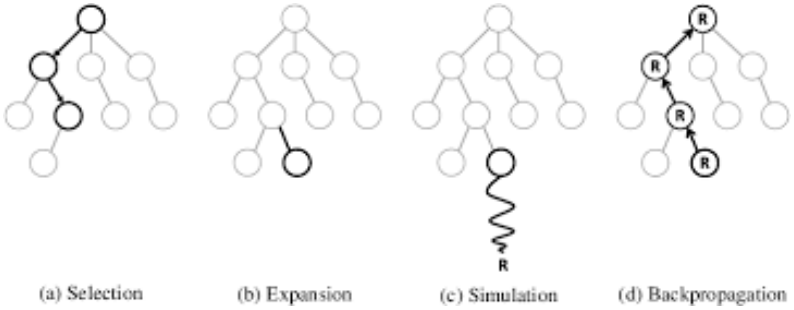
4.Monte Carlo Tree Search

O Monte carlo tree search(MCTS) é um algoritmo de procura que usa procura heurística e probabilística para chegar ao objetivo de forma mais rápida e eficaz, este combina busca em árvore com machine learning.

Quando um algoritmo segue o melhor caminho, este pode não ser a solução ótima, para estes casos o MCTS avalia periodicamente as outras opções que não as ótima. Este faz isto durante a fase de aprendizagem, executando-os, no lugar da solução ótima atual, processo conhecido como exploration-exploitation trade-off.

Neste processo a exploration continua a explorar diferentes caminhos da arvore que não o ótimo atual, podendo encontrar um ainda mais ótimo, isto pode ficar não eficiente em situações de grande escala, pra isso existe o processo de exploitation, este mantem-se num único caminho que tem o maior valor estimado (6).

O mcts é caracterizado por 4 paços:



- **Seleção:**

Na seleção o algoritmo escolhe um caminho a seguir, normalmente o com maior valor de nó.

Os valores são normalmente calculados pela Na expansão o algoritmo adiciona um nó filho ao ramo que foi selecionado. fórmula UCT (Upper Confidence Bound applied to Trees 1):

$$UCT\,Value = \frac{xi}{ni} + C \cdot \sqrt{\frac{\ln \quad T}{ni}}$$

Com xi = pontuação do nó, ni = playouts do nó, T = playouts do nó pai e C = 2, C e uma constante escolhida empiricamente

- **Expansão:**

Na expansão o algoritmo adiciona um nó filho ao ramo que foi selecionado.

- **Simulação:**

Na simulação o algoritmo escolhe aleatoriamente o caminho a seguir até chegar ao objetivo.

- **Back Propagation:**

Na “back propagation”, depois de se chegar ao valor do novo nó, é atualizado o valor desse ramo até ao nó inicial.

5. Quatro em linha

5.1. Descrição do jogo e regras

O 4 em linha ou Connect Four, é um jogo de estratégia que pertence a uma série de jogos populares. O jogo desenrola-se num tabuleiro de 7x6, ou seja, 7 colunas e 6 linhas, podendo cada coluna conter no máximo 6 peças e o jogo inicia-se com um tabuleiro vazio. Ambos os jogadores jogam as suas peças no tabuleiro, uma peça por jogada. É jogado usando 42 peças, cada jogador com peças coloridas diferentes (geralmente 21 fichas para um jogador e 21 fichas para o outro jogador). Um movimento consiste em um jogador deixar cair uma de suas fichas na coluna de sua escolha. Quando uma peça é jogada numa coluna, esta cai sempre, até atingir o fundo da coluna ou uma peça já colocada. O objetivo deste jogo é colocar quatro peças numa linha contínua vertical, horizontal ou diagonalmente. O jogo termina quando uma das condições seguintes se verificar:

- Um dos jogadores coloca quatro ou mais peças numa linha contínua vertical, horizontal ou diagonalmente, ou seja, esse jogador vence.
- Todas as casas do tabuleiro estão ocupadas e nenhum jogador satisfaz a condição anterior de vitória. Neste caso o jogo termina em empate.

O tamanho do tabuleiro do 4 em linha usualmente mais usado é 7 colunas por 6 linhas. Contudo, já foram criadas variações deste jogo tanto em tamanho como 5×4, 6×5, 8×7, 9×7, 10×7, 8×8, como em peças de jogo, por exemplo o jogador ter de colocar 5 ou mais peças na vertical, horizontal ou diagonal para vencer, e como em regras do jogo.

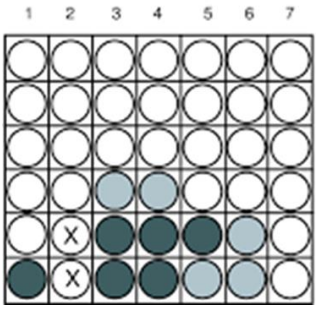


Figura 2 – Exemplo de um jogo de 4 em linha

Na Figura 2 está exposto um exemplo de uma partida do jogo 4 em linha. Neste caso, é visível que o jogador portador das fichas pretas é o vencedor. Independentemente de quem seja o seguinte a jogar: caso fosse a vez do jogador que possui as peças brancas e este decidisse colocar a sua ficha na coluna 2 para impedir o outro jogador de vencer, este poderia por a sua peça em cima dessa peça branca e igualmente vencer. Caso fosse a vez do jogador que tem as peças pretas este automaticamente ganharia se pusesse uma ficha na coluna 2.

5.2. Funções de avaliação

Uma função de avaliação, também denominada de função de avaliação heurística, retorna uma estimativa da utilidade esperada do jogo a partir de uma dada posição. Estas foram introduzidas devido à quantidade de tempo dos movimentos gerida pelos algoritmos. O algoritmo minimax gera todo o espaço de busca do jogo, enquanto alfa-beta nos permite ignorar grandes partes dele. Contudo, o algoritmo alfa-beta ainda tem que pesquisar o caminho todo até aos estados terminais qua ainda se revela ser uma porção do espaço de busca. Assim, essa profundidade geralmente não é prática porque os movimentos devem ser feitos em uma quantidade razoável de tempo. Em vez disso, os programas devem interromper a pesquisa mais cedo e aplicar uma função de avaliação heurística aos estados na pesquisa, transformando efetivamente nós não terminais em folhas terminais. Em outras palavras, altera-se minimax ou alfa-beta substituindo a função de utilidade por uma função de avaliação heurística e substitui-se o teste terminal por um teste de corte que decide quando aplicar a função de avaliação.

Para o este jogo em específico é usada uma heurística na qual se da uma pontuação a todos os segmentos de 4 peças. Essa pontuação é dada por:

- Se existem peças dos dois jogadores no mesmo segmento: resultado = 0
- Se existe apenas uma peça do computador: resultado = 1
- Se existem 2 peças do computador: resultado = 10
- Se existem 3 peças do computador: resultado = 50
- Se existe apenas uma peça do jogador: resultado = -1
- Se existem 2 peças do jogador: resultado = -10
- Se existem 3 peças do jogador: resultado = -50

Se num qualquer segmento forem encontradas 4 peças do computador então quer dizer que o computador ganhou e o resultado da heurística é 512, se forem encontradas 4 peças do jogador então o resultado da heurística é -512.

6.Algoritmos aplicados ao 4 em linha:

6.1. Implementação do jogo:

A Implementação do jogo foi feita através de uma classe chamada de Board. Esta classe tem como atributos 2 arrays bidimensionais(matrizes) de 6 colunas por 7 linhas de booleanos. A primeiro matriz tem o nome de “value” e a segunda de “visited” assim, para uma dada posição da matriz, se “visited” dessa posição é falso (0) então ainda não foi lá colocada uma peça de jogo, se “visited” é verdadeiro (1) então já foi lá colocada uma peça de jogo então, se “value” for falso quer dizer que o jogador colocou lá uma peça e se “value” for verdadeiro quer dizer que o computador colocou lá uma peça.

A classe tem 2 construtores e vários outros métodos:

- Board(): construtor publico, cria um novo tabuleiro vazio (matriz “visited” só tem 0).
- Board(bool nValues[6][7], bool nVisited[6][7]): construtor privado, cria um novo tabuleiro a partir de duas matrizes já existentes.
- Board* makeMove(int pos, bool who): método que cria um novo tabuleiro a partir do qual o método foi chamado com uma peça extra na coluna “pos” com o valor “who” (1 para computador e 0 para o utilizador)

- Boad* playerMove(): devolve o resultado do método “makeMove(pos,0)” com “pos” a ser input dado pelo utilizador.
- Board* computerMove(int pos): devolve o resultado do método “makeMove(pos,1)”, com “pos” a ser o parâmetro do método.
- void printBoad(): escreve o conteúdo das matrizes “visited” e “values” para o standard output de uma maneira legível por qualquer utilizador, quando “visited” é 0 é escrito um espaço em branco, quando “visited” é 1 e “value” é 1 é escrito um ‘X’ e quando “visited” é 1 e “value” é 0 é escrito um ‘O’.
- bool tie(): retorna verdadeiro apenas se o tabuleiro através do qual foi chamado é um empate, para isso verifica a linha do topo da matriz “visited” e se todos os seus elementos forem 1 então é empate.
- Short heuristic(): calcula a avaliação do tabuleiro a partir do qual foi chamado, para isso utilizamos o conceito de “sliding window”, por exemplo, há 4 segmentos na horizontal na primeira linha do tabuleiro, sendo que esses 4 segmentos tem muitos elementos comuns entre eles então para calcular o valor deles todos começamos por ver os constituintes do primeiro segmento depois, para o segundo segmento, basta ver qual o próximo valor (pois este é parte do segundo segmento mas não do primeiro) e retirar o valor do ultimo constituinte (pois este fazia parte do primeiro segmento mas não faz parte do segmento)

6.2. Tipos de algoritmos na pesquisa adversa

Connect four é um jogo no qual há no máximo 7 jogadas possíveis a fazer, sendo que o número de jogadas possíveis diminui apenas quando uma das colunas esta completamente cheia, o que só acontece depois de 6 jogadas, no mínimo.

Por isso, a arvore cresce exponencialmente com a altura (7^altura), então para os algoritmos minmax e alpha beta, que vêm todas as possíveis configurações a partir de um tabuleiro inicial, vão ter complexidade temporal exponencial, O(7^altura).

Por isso é necessário limitar o crescimento da arvore a uma certa altura.

Para o minmax é aconselhado um limite até 5 (inclusive) e para o alpha beta é aconselhado um limite até 9 (inclusive). Quando os limites escolhidos são maiores que os aconselhados o algoritmo começa a demorar mais do que 1 segundo para fazer uma jogada.

Para alem disso, numa tentativa de tentar aumentar a quantidade de tabuleiros podados pelo algoritmo alpha beta decidimos implementar uma ordem de desenvolvimento dos filhos feita especialmente para o jogo Connect four, então desenvolvemos primeiro as colunas centrais e só depois as colunas laterias.

Porem o algoritmo mcts tem sempre uma resposta, independente do número de iterações que são feitas, e a otimalidade da resposta dada aumenta consoante o número de iterações. Então decidimos limitar o tempo de execução do algoritmo em 0.75 s.

Minmax:

O algoritmo foi implementado utilizando 2 funções:

- A primeira "**int minMax(Board* root, int depth)**" é a função base, esta tem como parâmetros um pointer para um objeto da classe Board e um inteiro que representa a altura máxima da arvore a avaliar, esta devolve um inteiro que representa a coluna onde o computador vai jogar.
- A segunda função "**int auxMM(Board* root, bool who, int depth)**" é a função recursiva que tem os mesmos parâmetros que a função minmax, mais um booleano que indica quem esta a jogar. Porém esta função devolve a pontuação, mínima ou máxima dependendo do conteúdo do booleano, dos seus descendentes.

Assim, é inicialmente chamada a função "minMax" dando lhe como parâmetros o tabuleiro em que o computador deve jogar e o limite de altura escolhido. Depois, para cada descendente do tabuleiro original a função "auxMM" é chamada (com o booleano sempre verdadeiro, ou seja, é o computador a jogar). É então na função "auxMM" que o algoritmo conhecido por minimax é realizado sendo que são retornados 7 (ou menos) valores à função "minMax" e depois essa função vai devolver a coluna que leva ao maior valor.

Alpha Beta:

A implementação deste algoritmo utiliza, também, 2 funções:

- A primeira "**int alphaBeta(Board* root, int depth)**", tem os mesmos parâmetros que a função "minMax" anteriormente referida e devolve, também um inteiro que representa a coluna onde o computador vai jogar.
- A segunda função "**int auxAB(Board* root, bool who, int alpha, int beta, int depth)**", tem 2 parâmetros a mais em relação à função "auxMM" esses dois parâmetros são os valores usados para podar ramos da arvore que não tem importância para o resultado. Tirando os parâmetros alpha e beta, a função "auxAB" comporta-se da mesma maneira que a função "auxMM" exceto que a ordem de desenvolvimento dos filhos de um dado tabuleiro é {3, 2, 4, 1, 5,0 ,6}.

Estas duas funções tem a mesma relação que as funções "minMax" e "auxMM" tem, ou seja, a função "alphaBeta" chama a função "auxAB" para os descendentes do tabuleiro original. O algoritmo é então executado e os melhores valores são devolvidos, e então a função "alphaBeta" escolhe o melhor dos valores devolvidos.

MCTS:

A implementação do algoritmo MCTS começa pela definição de uma classe chamada Node, nesta classe são guardados a pontuação, o número de "playouts", um "pointer" para o tabuleiro que lhe esta associado e um "array" de 7 "pointers" para outros Nodes que são os filhos deste node.

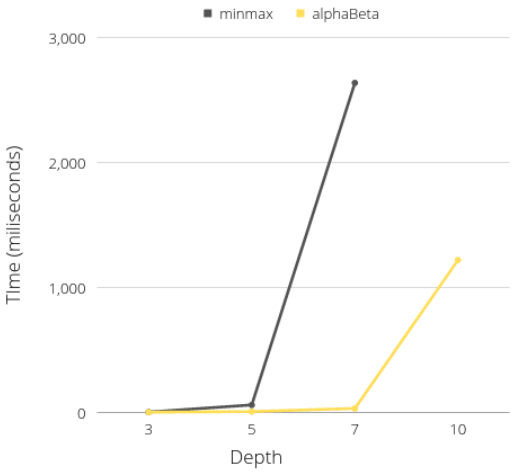
Esta classe tem também acesso a um construtor (que inicia um Node vazio) e a um método UCT que calcula o valor UCT de cada no, com C = 2.

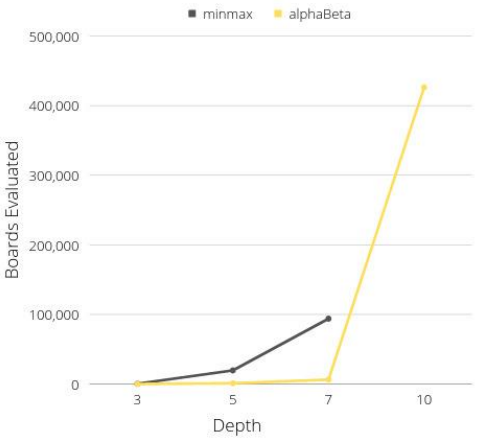
O algoritmo é iniciado pela função MCTS, é nesta função que se encontra o "loop" principal (enquanto não passaram 0.75 segundos continuar a fazer iterações).

Cada iteração é iniciada por chamar a função "**double playout(Node* root, bool who)**", esta função vai recursivamente escolher o no que tem maior valor UCT. Quando um no não tem mais filhos então a função **"double develop(Node* root, bool who)"** é chamada, esta função vai pegar no nó selecionado e desenvolver os filhos dele, depois vai aleatoriamente escolher um dos filhos desenvolvidos e fazer uma simulação nesse filho. Depois o resultado da simulação vai ser propagado, aumentando o valor dos nos do jogador que ganhou. Essa propagação é feita na função "playout" pois esta e recursiva.

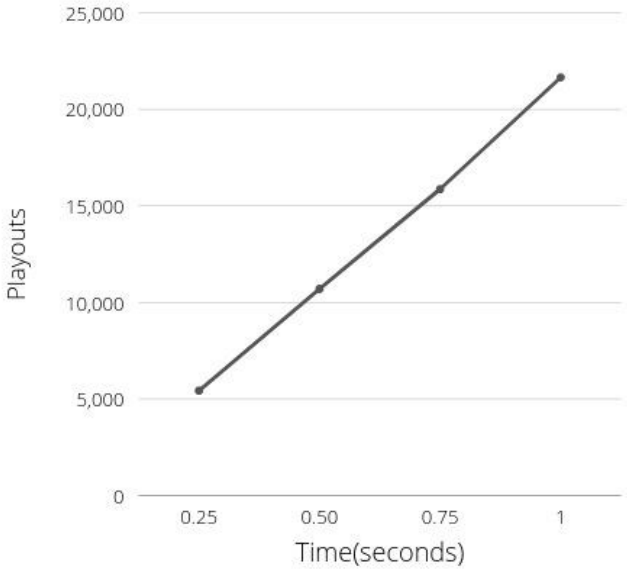
7. Estatisticas

7.1. Minmax e alpha-beta:





7.2. MCTS:



8.Comentários finais e conclusões:

Connect four é um jogo de 2 jogadores, jogado num tabuleiro 6 linhas por 7 colunas, onde os 2 jogadores poem uma peca alternadamente no tabuleiro, e a primeira pessoa a ter 4 pecas em vertical, horizontal ou diagonal ganha.

Existem vários algoritmos para encontrar a melhor jogada em jogos de 2 jogadores, dos quais os algoritmos minmax, alpha beta e monte carlo tree search(mcts). Os dois primeiros algoritmos são algoritmos que verificam todo o espaço de jogadas, enquanto o último algoritmo utiliza reinfored learning (fazer muitas simulações de jogo),

Por isso os dois primeiros algoritmos são bons quando um tabuleiro tem poucas jogadas a fazer num qualquer momento, ou seja, quando o grau de ramificação da arvore do jogo é baixo. Como o grau de ramificação do connect four é 7 (no máximo) a arvore torna-se muito grande o que faz. Isto faz com que o minimax só consiga ver até 6 ou 7 de jogadas, e que, para valores maiores, demore muito tempo. Mesmo o alfa beta, uma otimização do minimax, que tem uma performance bastante boa consegue ver apenas 10 ou 11 jogadas para a frente antes que a sua resposta demore mais de 2 segundos.

Por outro lado, o mcts é um algoritmo que não tem problemas quando o grau de ramificação e muito grande, isto deve se ao facto de que para selecionar um no para desenvolver descemos a arvore vendo apenas os filhos de um nó em cada nível, para alem disso ao fazer a simulação os movimentos são completamente a sorte(random). Com isso verificamos que o número de playouts efetuado é diretamente proporcional ao tempo de execução.

9.Referenciâs

(1) https://www.freecodecamp.org/news/minimax-algorithm-guide-how-to-create-an-unbeatable-ai/

(2) https://en.wikipedia.org/wiki/Minimax

(3) https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-4-alpha-beta-pruning/

(4) https://www.educative.io/edpresso/what-is-alpha-beta-pruning

(5) https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-4-alpha-beta-pruning/

(6) https://www.geeksforgeeks.org/ml-monte-carlo-tree-search-mcts/

(7) Chicago. Russell, Stuart J. (Stuart Jonathan). Artificial Intelligence: A Modern Approach.