**Implementação do Quatro em Linha**

FCUP

Inteligência Artificial (CC2006) 2018/2019

Trabalho de Grupo AK

Eduardo Morgado – 201706894 – MIERSI

Simão Cardoso – 201604595 – MIERSI

Sónia Rocha – 201704679 – MIERSI

1. **Introdução:**
   1. **Jogos como Problemas de Pesquisa:**

Em termos matemáticos, podemos considerar qualquer **ambiente com múltiplos agentes**[[1]](#footnote-1)como um jogo, caso haja um impacto por cada agente no jogo **[1]**. De entre os vários ramos de jogos possíveis de estudar em IA, os mais comuns são **jogos determinísticos**[[2]](#footnote-2), geralmente são jogos onde o agente está limitado por um número finito de ações restringidas por regras, jogos onde o estado atual do jogo é relativamente fácil de representar **[1]**.

Sendo assim um jogo pode ser visto como um problema de pesquisa onde o espaço é de tal forma grande que são necessários novos algoritmos para os poder resolver (algoritmos que anteriormente, para problemas de pesquisa simples, eram algoritmos ótimos, como o A\*, tornam-se quase obsoletos para jogos, devido a necessidades maioritariamente de tempo).

Um jogo, pode então ser definido como um problema de pesquisa constituído por:

* : estado inicial do jogo;
* : define qual dos agentes deverá tomar ação;
* : Ações disponíveis para um dado agente, seguindo um conjunto de regras;
* : uma aplicação de uma dada ação a um estado do jogo;
* : método que testa a existência de estados terminais[[3]](#footnote-3) no jogo;
* : função que define um dado valor final para um estado terminal, quantifica o estado terminal para o tipo de jogo em causa;

O estado inicial, o conjunto de ações e a sua aplicação define a árvore de pesquisa para o jogo, onde os ramos são as ações aplicadas/ a aplicar e os nós os resultados das ações (onde a raiz é o estado inicial).

Durante as próximas secções deste relatório, iremos aprofundar mais o temas de pesquisa em jogos, bem como implementar métodos de pesquisa para o jogo dos quatro em linha.

* 1. **Escolhas Ótimas em Jogos:**

Em qualquer problema de pesquisa, o objetivo será, através de uma sequência de ações, atingir um estado pretendido, no caso de jogos, essa sequência tem como objetivo chegar a um estado um estado terminal correspondente à sua vitória.

Um jogo pode, como vimos anteriormente, ter múltiplos agentes, quanto maior o número de agentes, mais trabalhosa será a pesquisa da sequência ótima, como tal, iremos focar-nos apenas em jogos com dois jogadores/agentes.

Num jogo com dois agentes, estes têm como objetivo ganhar/derrotar o jogador oposto, sendo assim, um jogador, tenta **maximizar** as suas hipóteses de ganhar quando é a sua vez de jogar e ao mesmo tempo, **minimizar** as hipóteses do adversário ganhar, pelo que, a árvore de pesquisa do jogo é constituída por níveis alternantes de jogadores, ou seja, num dado nível d, todos os nós/estados são produzidos pelo jogador do nível d-1, e os nós do nível d+1, são produzidos pelo jogador do nível d, como a *Figura 1* representa.

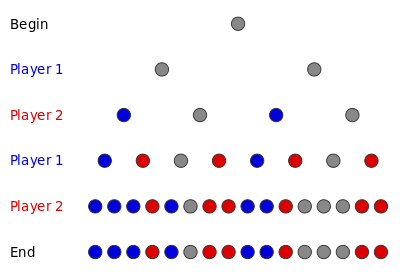


Figura 1-Exemplo de jogo com dois adversários (imagem retirada de [<https://en.wikipedia.org/wiki/Game\_tree#/media/File:Arbitrary-gametree-solved.svg>](%3chttps://en.wikipedia.org/wiki/Game_tree#/media/File:Arbitrary-gametree-solved.svg>))

Os algoritmos de pesquisa de solução ótima em jogos que iremos estudar são:

* Minmax sem Corte: Algoritmo recursivo que expande a árvore de jogo e, quando chega a um estado terminal, calcula a sua utilidade e retorna esse valor para os pais do nó terminal. Faz uma exploração completa da árvore em DFS **[2]**. Para uma árvore de profundidade *m* e ramificação *b* (número de ações legais), a sua complexidade temporal é **[2]**;
* Minmax com Corte Alpha-Beta: Este algoritmo surge da necessidade de melhorar o algoritmo anterior, ele opera da mesma forma que o Minmax, no entanto, é capaz de reduzir o espaço de busca em metade, eliminando partes da árvore **[3]**, obtendo uma complexidade ;
* Monte Carlo Tree Search: Método de simulação estatística que utiliza um parâmetro aleatório para simular soluções na árvore de pesquisa, capaz de lidar com árvores muito maiores e mais eficazmente que os algoritmos anteriores, uma vez que, apenas explora nós promissores na árvore.

Nas próximas secções iremos estudar com maior detalhe estes algoritmos.

1. **Pesquisa em Jogos com Adversários:**

Nesta secção iremos ver com mais pormenor os métodos de pesquisa para jogos com dois jogadores implementados para este trabalho.

* 1. **Minmax:**

Para um dado jogo, o nosso objetivo, o objetivo do algoritmo, é ganhar o jogo, para isso, este em cada vez que joga escolhe a jogada que tem maior potencialidade de levar a uma vitória, no entanto, não basta calcular as suas jogadas mas também, “prever” a jogada do adversário, uma vez que essa jogada tem carácter subjetivo e como tal imprevisível, o algoritmo considera a jogada do adversário como sendo a **jogada que irá causar mais dano à pontuação do computador** (sendo esta uma jogada ótima) e, considerando a máquina como sendo um jogador ótimo[[4]](#footnote-4), o algoritmo define uma árvore de decisão onde cada nível irá, alternadamente, procurar pelo maior valor (quando estamos em nível max, quando é uma jogada do computador) e pelo menor valor (quando em nível min, quando a jogada é do adversário) de utilidade de todos os estados, ou seja, quando em um estado max, este prefere escolher um nó com maior utilidade e quando num estado min, este prefere o nó com menor utilidade[[5]](#footnote-5) **[2-4]**.

É importante referir que a utilidade de um nó, é a pontuação que o computador (o jogador que chama o algoritmo minmax) tem, ou seja, quando é a vez do computador (max) é normal que, este sendo um jogador ótimo, escolha o nó/movimento que lhe maximize a pontuação, enquanto que, quando é a vez do adversário jogar[[6]](#footnote-6) este por sua vez, sendo um jogador ótimo, irá escolher o nó que minimize a pontuação do computador. Caso, na realidade, o adversário não for um jogador ótimo, como, no algoritmo consideramos sempre as situações do adversário que causassem maior dado, o resultado do computador ainda será melhor **[2]**.

Podemos então definir um método geral para o valor de minmax quando aplicado a um dado nó *s* tal como demonstrado na Figura *2*.

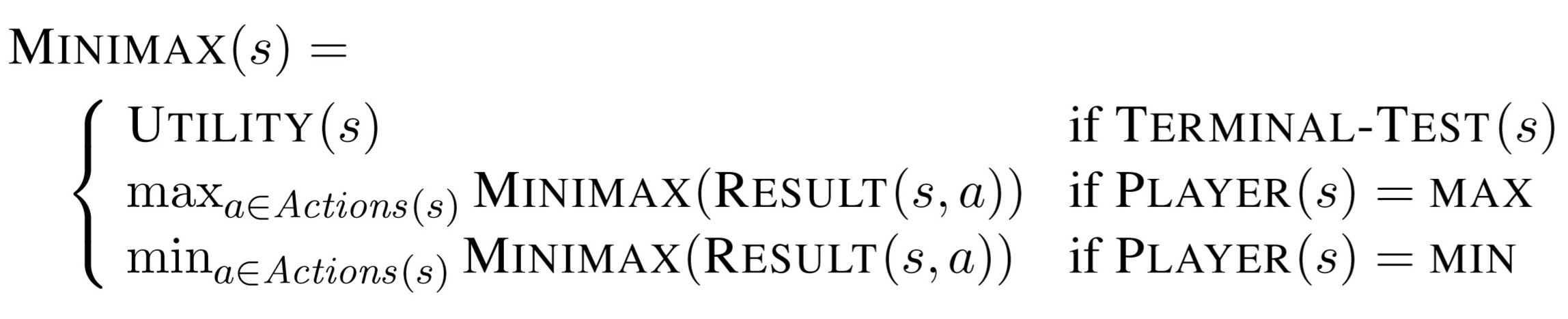


Figura 2-Método geral de Minmax (imagem retirada de **[2]**)

Este método é um método recursivo, como tal, desce a árvore até encontrar um estado terminal e depois os valores de utilidade desse estado são calculados e passados para os seus pais por backtrack. Uma vez que, o algoritmo é para ser aplicado numa situação de jogo, devem existir limitações de profundidade de forma que o algoritmo produza resposta em tempo útil, sendo assim, num jogo competitivo, para que o computador escolha sempre a melhor jogada no melhor tempo possível este algoritmo não é recomendado.

Considerando *m* como a profundidade máxima de pesquisa para o algoritmo e *b* as ações permitidas em cada estado (o grau de ramificação) a complexidade temporal deste algoritmo é de e a sua complexidade espacial pode ser de caso todas as ações sejam geradas ao mesmo tempo ou caso apenas as ações de um dado caminho sejam geradas, sendo assim, devido à sua complexidade exponencial, para problemas de pesquisa/jogos com um espaço de pesquisa considerável ou para jogos competitivos, onde o tempo de jogada é um influenciador, este algoritmo não é prático, sendo necessário encontrar/aplicar um algoritmo diferente.

A *Figura* *3* apresenta uma árvore de pesquisa para um dado jogo com dois jogadores, a seguir iremos analisar a árvore.

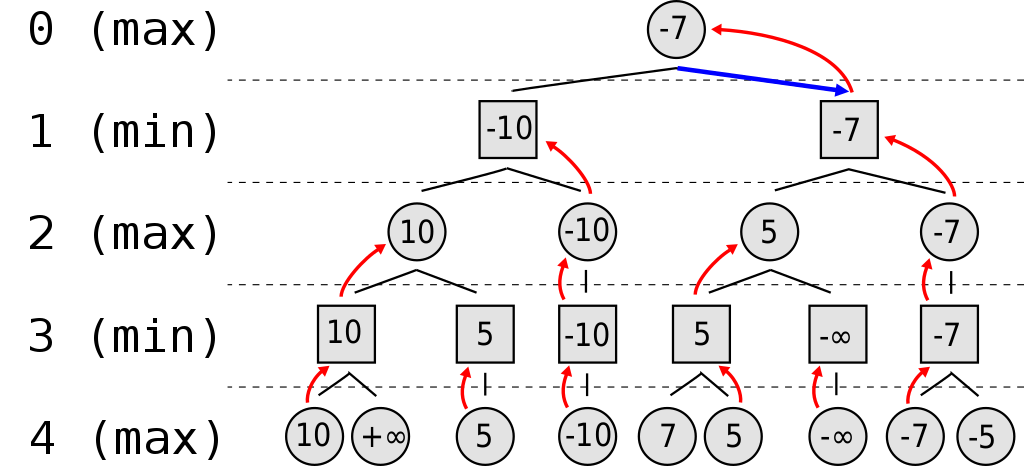


Figura 3-Uma árvore de pesquisa minmax para um dado jogo (imagem retirada de **[4])**

Inicialmente, o algoritmo é chamado pelo computador (max) e o nó raiz ainda não tem valor, durante a execução as fazes min e max serão chamadas alternadamente, de seguida, é expandido o 1º nó do nível 1, como ainda não é nó terminal, ou ainda não atingiu um limite de profundidade, continuamos a pesquisa em profundidade até atingir o 1º nó do nível 4, este nó ou é um nó terminal ou atinge um limite de profundidade, como tal, a sua utilidade é calculada (10), exploramos agora os outros filhos do nó pai (1º nó nível 3) e, após a exploração, como este é um nó min, iremos, de todos os seus filhos, escolher o nó com menor valor, neste caso, 10, assim, o 1º nó do nível 3 fica com valor 10. Continuando a pesquisa, após todos os filhos do 1º nó do nível 2 terem sido expandidos e os seus valores escolhidos, como este nó está num nível max, irá, de todos os seus filhos, escolher o filho com maior valor, ficando este agora com valor 10. A pesquisa irá continuar até não existirem mais nós a explorar (todos os filhos da raiz foram explorados) nesta altura, podemos agora escolher o valor da raiz, sendo esse, o valor/nó que o computador irá escolher quando fizer a sua jogada.

A *Figura 4* apresenta o pseudocódigo para este algoritmo.

**Minmax(root, max\_depth):**

1 retorna max(root, 0, max\_depth)

**max(node, depth, max\_depth):**

1 Se depth==max\_depth ou Termina(node) então retorna Valor(node)

2 Node max\_node

3 Para cada node\_filho fazer

4 Se min(node\_filho,depth+1,max\_depth) > Valor(max\_node) então

5 max\_node <-node\_filho

6 retorna max\_node

**min(node, depth, max\_depth):**

1 Se depth==max\_depth retorna Valor(node)

2 Node min\_node

3 Para cada node\_filho fazer

4 Se mxn(node\_filho,depth+1,max\_depth) < Valor(min\_node) então

5 min\_node <-node\_filho

6 retorna min\_node

Figura 4-Pseudocódigo Minmax

* 1. **Minmax corte Alpha-Beta:**

Como vimos anteriormente, o método minmax, tem uma complexidade temporal exponencial, limitando a aplicabilidade deste método, como tal, este novo método surge como uma tentativa de **diminuir o número de nós explorados** pelo minmax na árvore de pesquisa **[5]**. Quando este algoritmo é aplicado à árvore de pesquisa, retorna a mesma solução que o algoritmo minmax, cortando/ignorando ramos[[7]](#footnote-7) que não influenciam a decisão final **[3]**.

A partir de um dado nó, caso, no caminho a ser explorado, tenhamos encontrado um valor melhor, este nó nunca será alcançado **[3,5]**. Como tal, irão ser guardadas duas variáveis *alpha*, e *beta*, (daí o nome do algoritmo), *alpha* irá guardar o maior/melhor valor que foi encontrado no caminho explorado para max e *beta* irá guardar o menor valor encontrado no cominho explorado para min **[3,5]**.

Este algoritmo vai atualizando os valores de *apha* e *beta* durante a pesquisa, caso encontre, para um estado max ou min um valor pior que *alfa* ou *beta*, respetivamente, esse caminho que produz estes valores é cortado **[3]**.

Através dos cortes, este algoritmo tem a potencialidade de reduzir a árvore de pesquisa em metade, ou seja, tem na mesma uma complexidade espacial de ou (dependendo de como os nós são gerados, todos de uma só vez ou apenas os nós de um determinado caminho), no entanto, a sua complexidade pode ser no melhor dos casos ) (no pior a sua complexidade é igual à do minmax), ou seja, a pesquisa pode 2x mais aprofundada com o mesmo tempo de computação **[5]**.

Apesar deste algoritmo poder reduzir o espaço de busca do minmax, terá sempre que fazer uma pesquisa completa (até estados terminais ou atingir um limite de profundidade) para alguma porção da árvore, ou seja, nunca perdemos a complexidade exponencial. Mais uma vez, este método não é prático para obter soluções num tempo razoável, como tal, continua a ser necessário a existência de um algoritmo mais eficaz tanto em memória como em tempo.

A *Figura 5* apresenta o pseudo código do minmax alfa-beta.

**MinmaxAlphaBeta(root, max\_depth):**

1 retorna maxAlpha(root, 0, max\_depth,+,-)

**maxAlpha(node, depth, max\_depth, alpha,beta):**

1 Se depth==max\_depth ou Terminal(node) então retorna Valor(node)

2 Node max\_node

3 Para cada node\_filho fazer

4 Se MinMax\_min(node\_filho,depth+1,max\_depth) > Valor(max\_node) então

5 max\_node <-node\_filho

6 Se Valor(max\_node) >= beta então retorna max\_node

7 Se Valor(max\_node) < alfa então alpha <- Valor(max\_node)

8 retorna max\_node

**minAlpha(node, depth, max\_depth, alpha,beta):**

1 Se depth==max\_depth ou Terminal(node) então retorna Valor(node)

2 Node min\_node

3 Para cada node\_filho fazer

4 Se MinMax\_max(node\_filho,depth+1,max\_depth) > Valor(min\_node) então

5 min\_node <-node\_filho

6 Se Valor(min\_node) >= beta então retorna min\_node

7 Se Valor(min\_node) < alfa então alpha <- Valor(min\_node)

8 retorna min\_node

Figura 5-Pseudocódigo de Minmax corte Alpha-Beta

**2.3. Monte Carlo Tree Search:**

O método de Monte Carlo é um método para escolha de decisões ótimas em problemas, usualmente utilizado para jogo combinatórios **[7]**. Analisando os nós/movimentos mais promissores, combinando simulações aleatórias **[6-7]**, podendo ser visto como um método numérico universal para resolver problemas através de amostragem aleatória, este algoritmo **vai melhorando/aprendendo durante o tempo de execução**.

O algoritmo vai construindo uma árvore de pesquisa, onde, os nós são os seus átomos, sendo formados, consoante o resultado de um certo número de simulações de jogadas **[8]**. Este algoritmo pode ser dividido em 4 fases, todas estas a serem executadas em fila, até o método atingir um dado limite de recursos[[8]](#footnote-8).

As 4 fases são:

1. **Seleção/Selection**: A partir de um dado nó raiz da árvore, será escolhido recursivamente o melhor dos filhos de cada nó, enquanto esse nó não for filho **[6,7]**. Os nós ótimos (os escolhidos) são escolhidos com base numa função de avaliação **[7,8]**. Para garantir que a seleção dos nós é feita de forma justa[[9]](#footnote-9) este método utiliza o valor de UCB (*Upper Confidence Bound*), representado na *Figura 6*, este valor irá, ao longo da pesquisa, permitir que o algoritmo “aprenda” quais os nós têm uma maior prioridade de escolha. O valor de é o número de vitórias do nó para o i-ésimo movimento, é o número de visitas para o nó no i-ésimo movimento, e é o número de visitas ao pai do nó a ser analisado, *c* é o parâmetro de exploração[[10]](#footnote-10), geralmente, é considerado como sendo o seu valor **[6]**. A parcela à esquerda é alta para nós com uma média de vitórias alta e a parcela à direita é alta para nós com poucas visitas **[6]**, desta forma é possível balançar os nós com muitas vitórias e nós pouco visitados

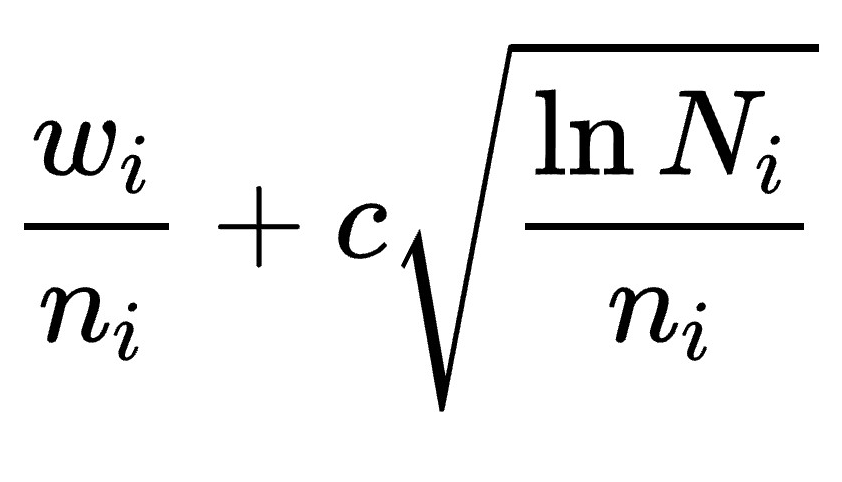


Figura 6- UCB (imagem retirada de **[6]**)

1. **Expansão/Expansion**: Após a fase de seleção (chegamos a um nó folha), podemos agora criar um novo nó para ser estudado, nesta altura, um nó filho do nó produzido pela fase de seleção é **aleatoriamente gerado** e **adicionado à árvore de pesquisa**;
2. **Simulação/Simulation**: Nesta fase, a partir do nó gerado na fase anterior, irá ser simulado um jogo completo, ou seja, enquanto não chegarmos a um estado terminal, iremos fazer jogadas aleatórias, é importante referir que durante esta fase, **nenhum nó é adicionado à árvore de pesquisa**, desta forma, a complexidade espacial deste método, é minimizada, uma vez que, apenas iremos guardar respostas concretas.
3. **Retorno/BackPropagation**: Quando, durante a fase de simulação, chegamos a um nó terminal, devemos aplicar uma função de avaliação/pontuação e a árvore de pesquisa atualizada com este novo valor, sendo assim o processo de *backpropagation* é iniciado, e todos os nós do caminho do nó criado na faze de expansão até à raiz da árvore são atualizados, as suas visitas incrementadas e, se o estado terminal obtido pela simulação for uma vitória para o computador, os seus valores de vitórias também serão incrementados.

A *Figura 7* apresenta um esquema destas fases.

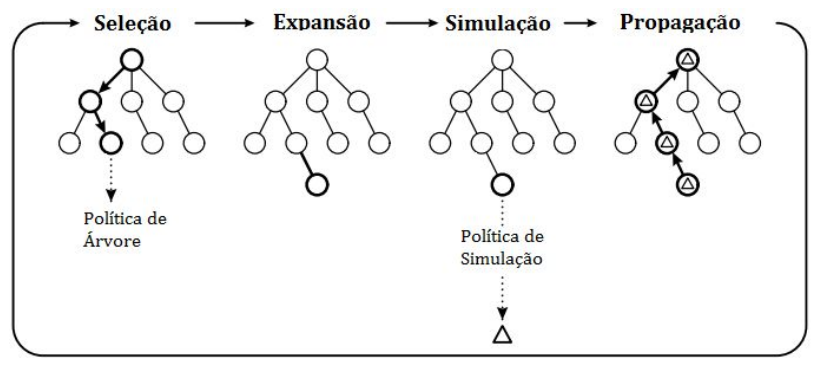


Figura 7-Fases do Monte Carlo

A *Figura* 8 apresenta o pseudocódigo para este método.

**MonteCarloTreeSearch (root, limit):**

1 Enquanto recursos < limit fazer

2 no\_selecionado <- selação()

2 no\_expandido <- expanção(no\_selecionado)

3 resultado <- simulação(no\_expandido)

4 backpropagate(resultado,no\_expandido)

**seleção():**

1 retorna seleção(root)

**Seleção(no):**

1 Para cada no\_filho fazer

2 ucb <-

3 Se ucb > max\_ucb então

4 max\_ucb <- ucb

5 melhor\_no <- no\_filho

6 retorna melhor\_no

**expanção(no)**

1 no\_filho <- FilhoAleatório(no)

2 AdicionaÁrvore(no\_filho)

3 retorna no\_filho

**Simulação(no)**

1 Enquanto no não é terminal fazer

2 no <- JogadaAleatória(no)

3 retorna no

**BackPropagate(valor,no)**

1 Enquanto Pai(no) não for Pai(root) fazer

2 Vitórias(no) <- Vitórias(no)+valor

3 IncrementaVisitas(no);

4 no <- Pai(no)

Figura 8-Pseudocódigo Monte Carlo

A complexidade deste método varia, consoante as implementações, numa implementação com pesquisas paralelas[[11]](#footnote-11) a sua complexidade temporal e espacial será, respetivamente e , onde *L* é o limite de recursos imposto n método, *m* o número de nós na árvore de pesquisa (nós expandidos, os simulados não são considerados), *k*, o número de pesquisas paralelas e *C* o número de *cores CPU* utilizados pelo método, isto para um método com vários *threads* **[9]**. No entanto, para a nossa implementação, não utilizamos mais do que um *core* a executar o método de Monte Carlo, sendo assim a nossa complexidade temporal e espacial será, respetivamente e , onde *m* são os nós na árvore de pesquisa e *L* o limite de recursos imposto.

Este algoritmo é vantajoso, uma vez que, é relativamente fácil de implementar, pode ser aplicado eficazmente sem um conhecimento muito extenso do problema[[12]](#footnote-12), escolhendo os movimentos e aprendendo com eles através de jogadas aleatórias, uma vez que, guarda a árvore de nós expandidos, esta pode ser reutilizada em iterações **[8]**.

No entanto, uma vez que, existe um grande número de nós e estes **não serem visitados um número suficiente de vezes** para que o algoritmo entenda o seu valor para o jogo, poderá **escolher um caminho que não leve à vitória**, para que este método sejam eficiente na escolha de uma jogada, deve ser fornecido um número elevado de iterações **[8]**.

Sendo assim, de todos os métodos estudados, o de Monte Carlo é o método mais eficaz em memória e tempo de execução que os outros métodos (para os outros métodos as suas complexidades temporais eram exponenciais), no entanto, este método **não garante que o computador ganhe sempre o jogo**.

**3. O problema dos Quatro em Linha (Connect Four):**

O jogo dos quatro em linha é um jogo de dois jogadores (X,O), definido numa matriz 6x7, com 42 células inicialmente livres. O jogo é jogado utilizando 42 peças, 21 peças X para um jogados e 21 peças O para o jogador adversário. Os dois jogadores vão jogando alternadamente até um deles ganhar ou empatarem. Um movimento/ação de um jogador, consiste em colocar uma peça numa coluna até que esta, ou atinga o final da coluna, ou uma outra peça nessa mesma coluna. Um jogador ganha, quando conseguir produzir uma linha com 4 peças suas contiguas, em qualquer direção (vertical, horizontal ou diagonal).

Para o nosso problema a implementar, um dos jogadores, O, será um dos algoritmos de pesquisa descritos anteriormente.

Para este problema, é necessário implementar uma função de avaliação/utilidade que funciona da seguinte forma (note que O, representa o computador):

* Se estamos num estado terminal:
  + Se o estado for empate, utilidade tem valor 0;
  + Se O (AI) ganha, utilidade tem valor +512;
  + Se X (jogador) ganha, utilidade tem valor -512;
* Caso contrário para todas as sequências de 4 células em qualquer direção (horizontal, vertical e diagonal) fazer:
  + utilidade += 50, caso existam 3 Os na sequência e nenhum X;
  + utilidade +=10, caso existam 3 Os na sequência e nenhum X;
  + utilidade +=1, caso exista apenas um O e nenhum X na sequência;
  + utilidade +=0, caso não exista nenhuma peça de um qualquer jogador na sequência, ou estão peças alternadas (XOXO,OXOX,…);
  + utilidade -=1, caso não existam peças de O na sequência mas 1 de X;
  + utilidade -=10, caso existam 2 Xs e nenhum O na sequência;
  + utilidade -=50, caso existam 3 Xs e nenhum O na sequência;

**Nota:** Neste cenário, no final de todas as sequências ainda é atribuído um valor bónus à utilidade, +16, caso o jogador desta jogada seja O e -16 caso seja X;

**Nota:** Esta função de avaliação garante que funciona, no entanto, **não favorece vitórias rápidas**, ou seja, se O pode ganhar em 20 movimentos ou 10, não existe um incentivo para que este escolha a jogada mais rápida[[13]](#footnote-13).

**4. Descrição de Implementação:**

Para esta implementação de resolução do problema utilizamos como linguagem o Java, dos fatores que levaram a esta escolha, estão a experiência anterior com esta linguagem, o facto de ser uma linguagem de programação orientada a objetos, facilitando na abstração do problema e da representação da tabela, bem como a existência de várias bibliotecas com as estruturas de dados que utilizamos, facilitando assim a implementação do problema. Quando comparada a linguagens como C, a nossa escolha apresenta mais vantagens, no entanto, uma outra linguagem que poderia ter sido utilizada, seria Python, acabando por não ser escolhida.

**4.1. Estruturas de Dados Escolhidas**

Nesta implementação, utilizamos 2 estruturas de dados implementadas em bibliotecas do java para enfileirar configurações de tabelas descendentes LinkedList e ArrayList (apenas utilizada no método de Monte Carlo).

A LinkedList foi utilizada para armazenar os nós descentes de um nó produzidos por aplicação de operadores a um nó (para esta implementação, consideramos um nó como sendo uma configuração da tabela). Como, apenas a utilizamos para inserir/remover nós (não há necessidade de consultar a lista) e, como na inserção, estamos a adicionar um elemento ao topo da lista (addFirst) e na remoção, a retirar a cabeça da lista (removeFirst()), ambas estas complexidades são o que não influencia a complexidade (temporal) do programa, como a lista só é gerada quando chamada, não aumenta a complexidade dos programas.

A ArrayList apenas é utilizada para o método Monte Carlo, uma vez que, como temos que gerar filhos aleatoriamente, se estivéssemos a utilizar LinkedList, quando removêssemos um nó aleatório, a operação custaria , enquanto que, com a ArrayList, apenas custa .

**4.2. Estrutura do código**

**4.2.1. Uma Classe Play**

Para podermos aplicar os métodos de Minmax e alpha-beta, é necessário na fase de backtrack subir o valor de utilidade e saber qual o movimento (linha/coluna) que conduz a uma vitória , para isso utilizamos uma classe *Play* para podermos guardar as jogadas, a *Figura 9* apresenta a interface *Play*

class Play{ //classe para armazenar as jogadas

int row,col; //valores das posições, linha coluna, do movimento guardado

int utility; //utilidade da jogada, apenas calculada

Play(int row,int col, int utility); //construtor da jogada para backtrack completo

Play(int row, int col)//construtor de jogada para pesquisa

int getRow(); //retorna a linha da jogada

int getCol(); //retorna a coluna da jogada

int getUtility(); //retorna a utilidade da jogada

void setRow(int r); //define a linha da jogada

void setCol(int c); //define a coluna da jogada

void setUtility(int u); //define a utilidade da jogada

}

Figura 9-Classe Play

**4.2.2. Uma Classe Table**

Independentemente do método de pesquisa que irá ser utilizado, terá sempre que existir uma estrutura de dados geral que represente o jogo, uma classe **Table** igual para todas as implementações. A classe deverá ter capacidades de manipular e gerar outras tabelas(descendentes), de calcular a utilidade da jogada, de guardar qual o jogador que a formou e se é um estado terminal. A *Figura 10* apresenta a interface da classe.

class Table{ //classe para armazenar uma configuração do jogo

char player; //guarda qual o jogador gerou esta tabela, ‘X’ ou ‘O’

char table[][]; //matrix representando a tabela de jogo, inicialmente todas a posições a ‘-‘

char champion; //guarda qual o jogador ganhou o jogo, caso table seja terminal

Play play; //valores das posições, linha coluna, da célula vazia na tabela

Table(); //construtor inicial do jogo, todas as células vazias

Table(Table copy); //construtor de tabela

Play getPlay(); //retorna a jogada para construção da configuração atual

char getPlayer(); //retorna o jogador que produziu a configuração atual

char getChampion(); //retorna o jogador que ganhou o jogo, caso terminal

char[][] getTable(); //retorna uma tabela independente de configuração

int getLastFreeRowPosition(int col); //dada uma coluna, retorna a primeira célula livre

boolean iscolumnFull(int col); //dada uma coluna, retorna true se não existirem mais células livres numa dada coluna

boolean hasWin(); //verifica se a tabela é um estado terminal

boolean isGameOver(); //verifica se o jogo terminou

boolean setChampion(char player); //atualiza o jogador que ganhou o jogo

LinkedList<Table> descendents(); //gera a lista de tabelas descendentes, para os 7 movimentos possíveis (7 colunas)

void makeNewPay(int colm,char player); //dado um jogador e uma coluna, atualiza a tabela, adiciona uma célula do jogador na coluna col (criando uma nova Play) e atualiza o jogador que produziu table

int getUtility(); //função de avaliação, retorna o valor da tabela, é suportado por 4 métodos

int utilHorizontal(); //cálculo de utilidade horizontal

int utilVertical(); //cálculo de utilidade vertical

int utilDiagonalRight(); //cálculo de utilidade para diagonal decrescente

int utilDiagonalLeft(); //cálculo de utilidade para diagonal crescente

}

Figura 10-Classe Table

**4.2.3. Uma Classe MinMaxNoPrun**

Para a nossa implementação, construímos um único objeto desta classe, fornecendo a máxima profundidade para o método, esse objeto irá, em cada jogada real do computador chamar a implementação do minmax, neste método, serão camados alternadamente os métodos de min e max. A *Figura 11* apresenta a interface desta classe.

class MinMaxNoPrun{ //classe para o algoritmo minmax

int max\_depth\_prun; //guarda o limite máximo de profundidade da pesquisa

MinMaxNoPrun(int m\_depth); //construtor inicial do algoritmo

Play minMax(Table tb); //dada uma tabela, chama max e retorna a jogada(Play) obtida por max

Play max(Table tb, int depth);

Play min(Tablet tb, int depth);

}

Figura 11-Classe MinMaxNoPrun

**4.2.4. Uma Classe MinMaxAlphaBeta**

Para a nossa implementação, construímos um único objeto desta classe, fornecendo a máxima profundidade para o método, esse objeto irá, em cada jogada real do computador chamar a implementação do alfabeta, irá chamar alpha, em cada iteração de alpha, será chamado o minmaxNoPrun para max. A *Figura 12* apresenta a interface desta classe.

class MinMaxAlphaBeta{ //classe para o algoritmo minmax

int max\_depth\_prun; //guarda o limite máximo de profundidade da pesquisa

MinMaxNoPrun minimax; //cria um objeto da classe MinMaxNoPrun ode depois, irá chamar max em alpha

MinMaxAlphaBeta(int m\_depth); //construtor inicial do algoritmo

Play alphaBeta(Table tb); //dada uma tabela, chama max e retorna a jogada(Play) obtida por max

Play max(Tablet tb, int depth, int alpha, int beta);

}

Figura 12-Classe MinMaxAlphaBeta

**4.2.4. Uma Classe Node e MonteCarloTreeSearch**

Para a nossa implementação, construímos um único objeto desta classe, fornecendo a máxima profundidade para o método, esse objeto irá, em cada jogada real do computador chamar a implementação do alfabeta, irá chamar alpha, em cada iteração de alpha, será chamado o minmaxNoPrun para max. A *Figura 12* apresenta a interface desta classe.

**5. Resultados:**

**6. Comentários Finais**

**Nota Adicional:** Este relatório, bem como o programa utilizado para obtenção dos dados, pode ser acedido no repositório da equipa, <https://github.com/thejoblessducks/the-15-that-was-the-puzzle.git>, ou em <https://github.com/eamorgado/ProjetoIA-15Puzzle.git>

**7. Referências:**

**[1]** S.Russell, P. Norving, “Artificial Intelligence A Modern Approach”, 3rd Ed. , *5.1 Games*, *pp*. 161-163, 2009

**[2]** S.Russell, P. Norving, “Artificial Intelligence A Modern Approach”, 3rd Ed. , *5.2 Optimal Decisions in Games*, *pp*. 163-167, 2009

**[3]** S.Russell, P. Norving, “Artificial Intelligence A Modern Approach”, 3rd Ed. , *5.3 Alpha-Beta Prining*, *pp*. 167-169, 2009

**[4]** Wikiwand n.d., *MinMax*, consultado pela última vez a 27 de março de 2019, [<http://www.wikiwand.com/en/Minimax>](http://www.wikiwand.com/en/Minimax)

**[5]** Wikiwand n.d., *Alpha-bet pruning*, consultado pela última vez a 27 de março de 2019, [<http://www.wikiwand.com/en/Alpha–beta\_pruning>](http://www.wikiwand.com/en/Alpha–beta_pruning)

**[6]** Wikiwand n.d., *Monte Carlo tree search*, consultado pela última vez consultado a 27 de março de 2019, [<http://www.wikiwand.com/en/Monte\_Carlo\_tree\_search>](http://www.wikiwand.com/en/Monte_Carlo_tree_search)

**[7]** Monte Carlo Tree Searh n.d., *About*, consultado pela última vez a 28 de março de 2019, [<http://mcts.ai/about/index.html>](http://mcts.ai/about/index.html)

**[8]** GeeksforGeeks n.d., *Monte Carlo Tree Search (MTCS)*, consultado pela última vez a 28 de março de 2019, [<https://www.geeksforgeeks.org/ml-monte-carlo-tree-search-mcts/>](https://www.geeksforgeeks.org/ml-monte-carlo-tree-search-mcts/)

**[9]** Standford University n.d., *CME 323, Report*, Yifan Jin, Shaun Benjamin, pp 3, consultado pela última vez a 28 de março de 2019, [<http://stanford.edu/~rezab/classes/cme323/S15/projects/montecarlo\_search\_tree\_report.pdf>](http://stanford.edu/~rezab/classes/cme323/S15/projects/montecarlo_search_tree_report.pdf)

1. Espaço onde, para um determinado agente, para que este tome alguma decisão, deve considerar as ações de outros agentes no espaço bem como as suas consequências **[1]**, ou seja, para uma tomada de decisão é necessário o estudo de ações externas ao agente atual, estas ações externas/de outros agentes, introduzem imprevisibilidade. [↑](#footnote-ref-1)
2. Jogos onde o espaço de resultados pode ser completamente observado/estudado, onde existe uma alternância entre agentes e os valores de utilidade no final do jogo são simetricamente iguais para cada jogador. [↑](#footnote-ref-2)
3. Um estado terminal pode ser visto como um estado a partir do qual, independentemente das ações tomas pelos agentes, nos próximos passos, é sempre garantida a vitória/derrota de um agente, ou o empate. [↑](#footnote-ref-3)
4. Um jogador ótimo é um jogador que escolhe uma jogada que conduza à melhor pontuação. [↑](#footnote-ref-4)
5. Valores mínimos de utilidade favorecem o oponente, valores máximos de utilidade favorecem o computador. [↑](#footnote-ref-5)
6. Aqui, durante o algoritmo a vez do adversário é puramente hipotética, o algoritmo tenta prever uma jogada do adversário, considerando-o como um jogador ótimo. [↑](#footnote-ref-6)
7. Ramos podem ser cortados, pois nunca são visitados, caso o jogador adversário jogue de forma ótima. [↑](#footnote-ref-7)
8. É importante impor um limite para o método, caso contrário, nunca iria terminar, esse limite de recursos pode ser um limite temporal (de tempo de execução do método) ou um limite espacial (número de nós guardados na árvore). [↑](#footnote-ref-8)
9. Uma seleção justa, é aquela que confere a possibilidade de todos os nós virem a ser explorados, de forma a não focar muito a pesquisa, perdendo soluções. [↑](#footnote-ref-9)
10. A escolha do valor do parâmetro depende muito do problema a que o método de Monte Carlo está a ser aplicado. [↑](#footnote-ref-10)
11. Uma vez que, cada ramo da árvore é independente dos outros, numa máquina com vários *cores*, cada um poderá estar a explorar um ramo individual, tornando a pesquisa muito mais rápida [↑](#footnote-ref-11)
12. Necessita apenas de conhecer as regras do problemas e as condições terminais **[8]**. [↑](#footnote-ref-12)
13. Isto poderá ser facilmente corrigido nos algoritmos de pesquisa se, para além de estarmos a escolher a melhor jogada, tomar em atenção a profundidade da pesquisa. [↑](#footnote-ref-13)