Modelagem do jogo Ultimate Tic-Tac-Toe utilizando grafos Minimax com alpha-beta pruning

Fagner M. Dias[1], João S. Belau[1], Patrícia P. Cruz[1]

[1]Instituto Metrópole Digital - Universidade Federal do Rio Grande do Norte (UFRN) - RN - Brasil

Abstract: This article aims describe the modeling for a graph-problem and the possible solutions or approximations. We're dealing, more specifically, of a decision tree, popularly known in the literature as Minimax-Tree. In this example, we use a game called Ultimate Tic-Tac-Toe, a variation of the widely known Tic-Tac-Toe.

The algorithm should be able of take the best possible decisions for each existent scenario during the game execution. This paper still includes Alpha-Beta Pruning, a optimization of the traditional algorithm that enhances its performance using a "pruning", removing branches with low score and avoiding the checking of useless cases.

Resumo: O presente artigo tem o objetivo de descrever a modelagem para um problema em grafos bem como sua solução ótima ou aproximada. Estamos tratando, mais especificamente, da árvore de tomada de decisões, conhecida na literatura como Árvore Minimax. Neste exemplo, utilizaremos um jogo chamado Ultimate Tic-Tac-Toe, que é uma variante do famoso "Jogo da Velha", popularmente conhecido no Brasil.'

O algoritmo deverá ser capaz de tomar as melhores decisões possíveis a partir de cada cenário existente durante a execução do jogo. Este artigo contempla a melhoria conhecida na literatura como Alpha-Beta Pruning, que se trata de uma otimização do algoritmo tradicional que melhora seu desempenho a partir de uma "podagem" feita em seus ramos de menor pontuação, tornando desnecessária a verificação de muitos dos nós da árvore.

1.Introdução

Este estudo tem por objetivo a compreensão e modelagem em grafos para a solução de um jogo chamado Ultimate Tic Tac Toe (Ultra Jogo da Velha). Mais do que obviamente, este jogo é baseado no famoso Tic Tac Toe (jogo da velha).

Descobrimos sobre este jogo enquanto desenvolvíamos nossa primeira pesquisa para o primeiro Workshop da disciplina de grafos, onde falamos sobre a aplicação de árvores minimax, seguindo as definições de (*Minimax*, 2016), para otimização de jogadas em Jogo da Velha.

Esta modificação do famoso jogo de Soma-zero, que pode ser visto em (*Soma-zero*), consiste em um tabuleiro 3x3 em que cada posição possui outro tabuleiro 3x3. Esta modificação adiciona uma complexidade maior ao jogo.

2. Regras do Jogo

Demostraremos as regras utilizando algumas imagens a seguir, onde o quadrado azul representa o primeiro jogador e o vermelho, o segundo.

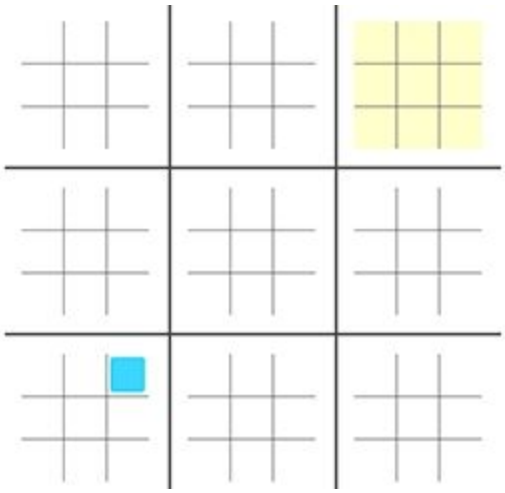

Figura 1. Jogada inicial.

Ao invés de 9 jogadas iniciais possíveis, o primeiro jogador pode jogar em qualquer um dos 81 quadrados possiveis. Porém, a partir da primeira jogada, todas as outras se basearão na posição do adversário na jogada anterior.

Por exemplo, na figura 1, o jogador azul jogou na posição superior direita do tabuleiro interno. Com isso, o jogador vermelho será obrigado a fazer a sua jogada em algum lugar dentro do tabuleiro externo superior direito (marcado em amarelo).

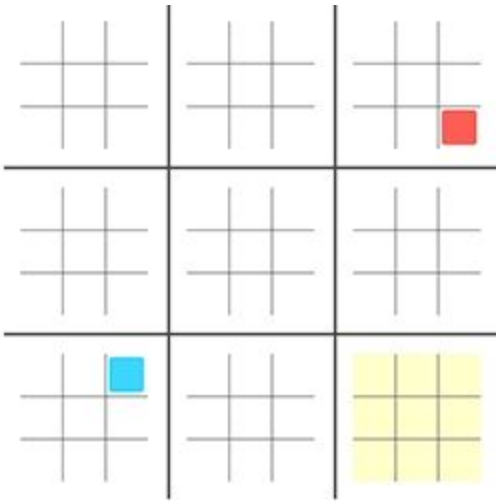

Figura 2. Segunda Jogada.

Seguindo o exemplo da jogada passada, na figura 2, o jogador vermelho fez a sua jogada na posição inferior direita do tabuleiro interno, forçando o azul a jogar na posição inferior direita no tabuleiro externo (figura 3).

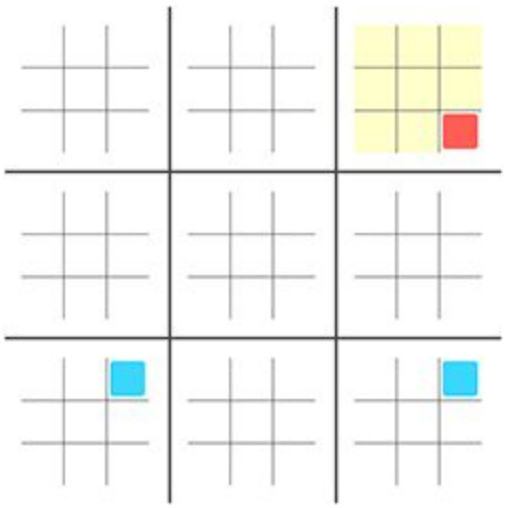

Figura 3. Terceira jogada.

Mesmo parecendo complicado, o jogo ainda é um jogo da velha, ou seja, é necessário um dos jogadores preencher 3 posições seguidas(vertical, diagonal, horizontal) do tabuleiro externo que ele ganha.

Para poder marcar uma posição do tabuleiro externo como “seu”, o jogador precisa ganhar o jogo interno dessa posição, e esta ficará marcada, no caso da figura 4, com a cor do jogador que a ganhou.

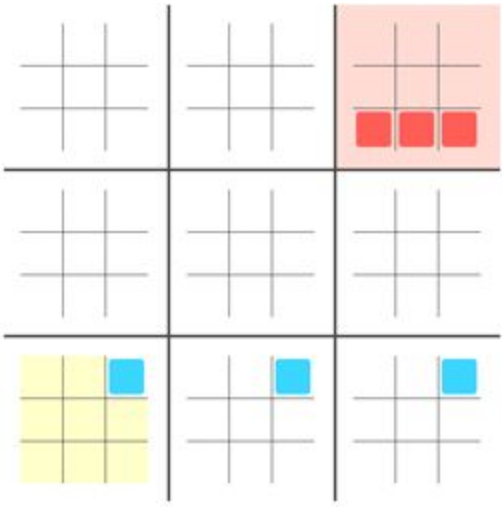

Figura 4. Ganhando um jogo interno.

Da mesma forma, quando um jogo interno dá velha, a posição do tabuleiro externo correspondente à ela inutilizada, pois nenhum jogador conseguiu vencê-la.

Outra regra importante é que quando um jogador faz sua jogada em uma posição do tabuleiro interno e sua posição correspondente no tabuleiro externo já teve seu jogo interno ganho (como a figura 4), o jogador seguinte poderá escolher qualquer posição do tabuleiro para realizar a sua jogada.

3.Análise das Bibliografias

Durante a busca por um algoritmo para a modelagem do nosso problema, descobrimos a árvore de tomada de decisões chamada de minimax.

Uma árvore Minimax é uma árvore de tomada de decisão que visa minimizar a perda máxima possível observando os possíveis estados seguintes. O algoritmo padrão da Minimax (também conhecida como naïve) nada mais é a versão mais simples da Minimax, onde todos os estados possíveis seguintes serão checados, de forma a ter certeza que cada decisão tomada irá resultar em um estado final ótimo conhecido.

Porém a árvore minimax pode ficar bastante grande dependendo do número de estados possíveis. O tempo de cálculo e de memória podem se tornar incomodos com o tempo. Existem métodos que melhoram consideravelmente o uso do minimax, um deles é conhecido na bibliografia como Alpha-Beta Pruning e pode ser visto em (*CS 161 recitation notes - Minimax with Alpha Beta pruning)*. Este método consiste, resumidamente, em uma melhoria do algoritmo

padrão que exclui da árvore caminhos que podem ser desconsiderados, de forma a não ser necessário o percorrimento de todos os caminhos possíveis.

Em nossa aplicação, temos um tabuleiro de 9 posições cada um com outras 9 posições. Essa grande quantidade de possibilidades torna necessária a implementação dessa melhoria, tendo em vista que o método naïve poderia facilmente sobrecarregar a memória e deixar o processamento bastante lento.

4. Minimax com Alpha-Beta Pruning

Alpha-Beta Pruning é uma melhoria da minimax desenvolvida por Alexander Brudno em 1963, como descrito em (*Alexander Brudno*, 2016). É também interessante citar o nome de Judea Perl, que provou sua otimização em cima do algoritmo em 1982.

Resumidamente, o algoritmo remove galhos da árvore min-max com o objetivo de acelerar o processo de cálculo. É como se o Alpha-Beta "limitasse" o algoritmo diretamente para o melhor e "mais promissor" galho da árvore. A otimização reduz efetivamente a profundidade da árvore para um valor muito próximo da metade do que um simples minimax se os nós são organizados em uma órdem ótima ou em uma órdem quase ótima.

Com um aproximado fator branching (número de filhos por nó) de b e uma profundidade de busca de d turnos, o número máximo de nós folhas (quando o movimento é não-ótimo) é $O(b^d)$, cuja demonstração pode ser verificada em (*T. Hajiaghayi*), ou seja, o mesmo que uma busca minimax. Se o movimento é ótimo, significando que os melhores movimentos são sempre encontrados primeiro, o número de nós folhas fica por volta de $O(b^{d/2})$, que também pode ser verificado em (*T. Hajiaghayi*).

O algoritmo de minimax com alpha-beta pruning consiste, basicamente, na mesma ideia da naïve minimax, porém com a utilização de fatores α e β.

O α está ligado aos nós max e o β ao min, onde, do mesmo jeito da minimax simples, o α será inicializado com o pior valor para max (-∞) e β com o pior valor para mim (+∞), porém cada nó apresentará valores para ambos α e β.

O código será executado como uma busca em profundidade onde os valores de α e β serão alterados com os valores das heurísticas dos nós terminais, porém com um fator importante: a condição da podação (pruning). A condição utilizada será $\alpha \geq \beta$.

Quando essa condição for atingida o pruning acontecerá e os ramos seguintes serão excluidos da busca, diminuindo assim o uso de memória, viabilizando melhores tempos de execução.

Demonstraremos a seguir, de forma mais clara, a execução do código da minimax com Alpha-Beta pruning utilizando as figuras com heurísticas hipotéticas.

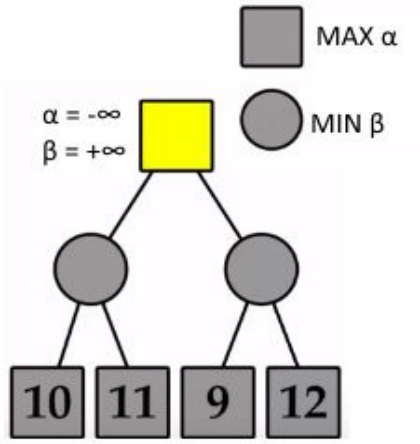

Figura 8. Inicio α-β Pruning

Vamos seguir os passos com base no nó max(α), em amarelo, que quer maximizar seus resultados, com base na figura 8.

De início a raíz(max) recebe os valores $\alpha = -\infty$ e $\beta = +\infty$, que são os piores valores para α e β.

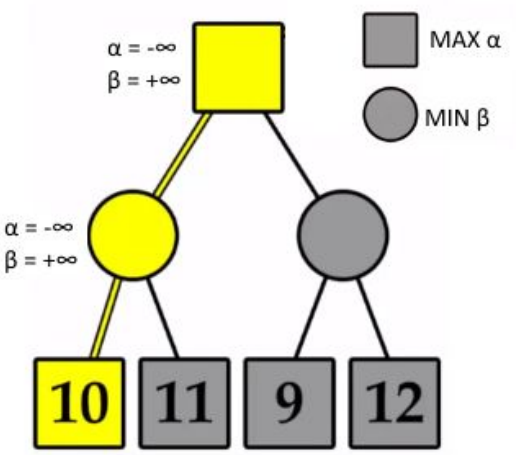

Figura 9. Continuação α-β Pruning

Como o algoritmo segue uma busca em profundidade, ele irá descer, repassando o valor de max para seus filhos, de forma a colocar os valores $\alpha = -\infty$ e $\beta = +\infty$ no nó intermediário(min),

Quando chegar no nó terminal cujo valor de heurística é 10, este valor será retornado para o nó antecessor.

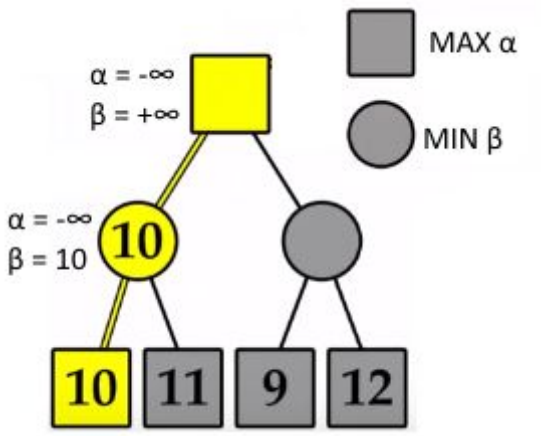

Figura 10. Continuação α-β Pruning

Voltando recursivamente, como o nó antecessor é min, ou seja, ele altera o apenas o valor de β, que no caso será o mínimo entre +∞ e 10, como mostrado na figura 10.

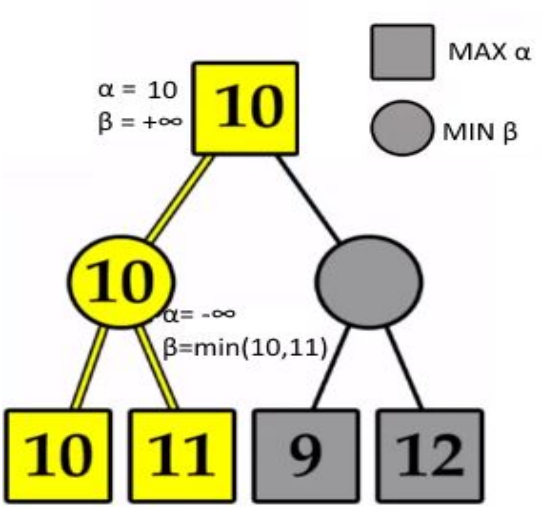

Figura 11. Continuação α-β Pruning

Antes de descer para o filho esquerdo, será testada a condição de pruning($\alpha \geq \beta$). Como 10 não é maior que +∞, então o código continua para o próximo filho cuja heurística é 11.

Voltando recursivamente, min recebe a heurística anterior, mas como 11 é maior que 10, e β sempre quer o menor valor possível, então β continua como 10.

Como o min não tem mais filhos, o código volta recursivamente e o valor de β dele(como é min retorna β) é repassado para a raiz. Sabendo que a raiz é um nó máximo, o valor recebido irá alterar o valor de α, sendo passado o maior valor entre -∞ e 10, no caso, o 10. Podendo ser verificado na figura 11.

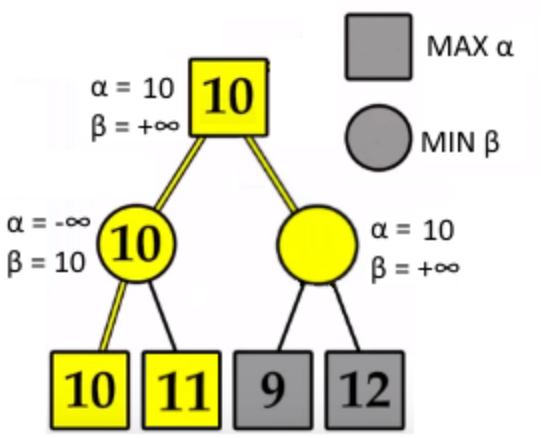

Figura 12. Continuação α-β Pruning

Seguindo para a figura 12, podemos verificar que como não é o caso da condição de pruning (α ≥ β), o algoritmo continua para o ramo direito da árvore, onde os valores de α e β serão repassados para o filho direito.

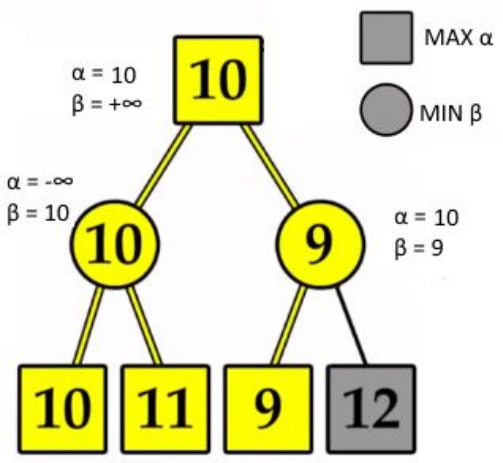

Figura 13. Continuação α-β Pruning

Continuando a demonstração do código, sabemos que não é o caso da condição de pruning ($\alpha \geq \beta$). Com isso o algoritmo continua para o ramo esquerdo, seguindo para o nó terminal com valor de heurística 9 que será retornado ao seu antecessor.

Como o valor foi retornado para um nó min, ele irá alterar o valor de β para 9, já que 9 é menor que $+\infty$, como observado na figura 13.

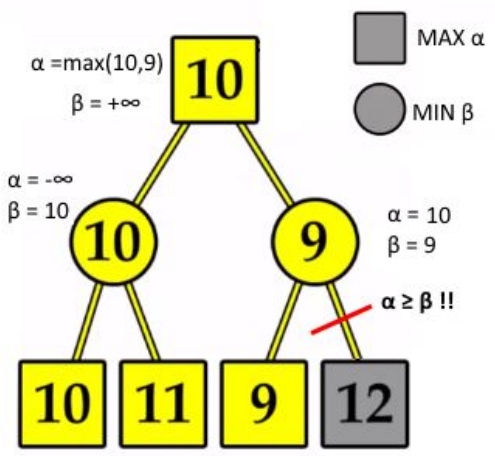

Figura 14. Continuação α-β Pruning

A partir da figura 14, podemos observar que, nesse caso, a condição de pruning foi alcançada ($\alpha \geq \beta$) e , portanto, não será necessário continuar para os possíveis filhos de min ainda não verificados, já que, para a minimax, saber que aquele ramo já garante um valor bom é o suficiente.

Com isso, o valor de β do min direito é retornado para raiz , que é um nó max. Porém esse valor não irá para α, já que o maior valor entre 10 e 9 é 10.

Note que não foi necessário a verificação de um dos ramos para chegar a uma conclusão com relação ao valor dessa árvore. Essa exclusão que torna a minimax mais eficiente e viável.

Segue abaixo o pseudocódigo da minimax com alpha-beta pruning:

```
Function alphabeta(node, depth, a, b, maximizingPlayer){

    //caso base
    if depth = 0 or node is a terminal node{
        return the heuristic value of node
        }

    if maximizingPlayer is TRUE{

        for each child of the node{

            a = max ( a , alphabeta(child, depth -1, a, b, FALSE))

            if a >= b
                break // podar o resto da árvore
        }

        return a
    }

    else if maximizingPlayer is FALSE{

        for each child of the node{
            b = min( b, alphabeta(child, depth -1, a, b, TRUE))

            if a >= b
                break //podar o resto da árvore
        }

        return b
    }
}
```

Figura 15. Pseudcódigo da Minimax com alpha-beta pruning

5.Heurísticas

Levando em consideração o custo computacional da função minimax caso ela tivesse que percorrer a árvore inteiro em busca de valores de fim de jogo, decidimos utilizar uma heurística para otimização do algoritmo.

Com as pesquisas, encontramos uma heurística bastante interessante para o Ultimate Tic-Tac-Toe que irá facilitar bastante a podação da minimax com alpha-beta pruning.

Nos nós terminais, cada umas das 8 linhas válidas(3 linhas horizontais, 3 linhas horizontais e 2 linhas diagonais) serão avaliadas de acordo com a quantidade de 'X's (ou 'O's) existentes, e um valor era atribuído às mesmas da seguinte forma:

- 200 (ou -200) para cada tabuleiro interno que leve a vitória de X (ou O);
- 100 (ou -100) para cada linha com três 'X's (ou 'O's);
- 10 (ou -10) para cada linha com dois 'X's (ou 'O's) e um espaço vazio;
- 1 (ou -1) para cada linha com um 'X's (ou 'O's)e dois espaços vazios;
- 0 em outros casos.

A aplicação dessa heurística garante um funcionamento do algoritmo eficiente e com menor custo computacional, porém não garante 100% de vitória da máquina, já que não necessariamente o algoritmo irá escolher o caminho ótimo, dando chances ao usuário de ganhar.

5.Modelagem em Grafos

Após aprender como funciona o jogo e estudar o algoritmo da Minimax com alpha-beta pruning, é possível visualizar e modelar o problema em grafos.

No caso da Ultimate Tic-Tac-Toe os estados dos tabuleiros internos serão os nós do grafo e as arestas serão as transições entre os estados. Os nós receberão valores heurísticos de acordo com o estado do tabuleiro interno, seguindo o modelo explicado no tópico de Heurísticas.

Nesse grafo será aplicado o algoritmo da Minimax com alpha-beta pruning e algoritmos auxiliares para manter a lógica das regras do jogo, e com base nas heurísticas, a árvore será podada de forma eficiente.

7.Criação do Código

De início, partimos do código do jogo da velha normal que produzimos para o workshop e o expandimos mantendo a base da minimax naïve e a jogabilidade em turnos, onde o usuário será o X e a máquina o O.

Em um segundo momento, criamos a classe Board, responsável pelos tabuleiros internos. Esta classe possui métodos e atributos que servem para controle e checagens que ajudam o método heurístico adotado auxiliando a Alpha-Beta, pois, apesar dela ser uma melhoria para a Minimax, o número de possibilidades é imenso e carece de heurísticas para acelerar o processo de cálculo.

Foi feita uma classe tabuleiro que armazena 9 boards (tabuleiros internos) e possui métodos para a verificação das condições de vitória e empate.

Nas regras do jogo, somos obrigados à marcar no board referente à posição da última jogada, isto gera excessões nas quais tabuleiros podem ser invalidados caso estejam vencidos por alguém ou empatados (isto é, sem posições disponíveis para jogar), então se fez necessária uma função auxiliar que retorna o tabuleiro com as maiores condições de vitória em determinado momento. Essa função retorna qual board possui a maior probabilidade de sucesso a partir de condições pré determinadas. Ela funciona varrendo os boards válidos e salvando em um atributo do proprio board um valor correspondente ao jogador que precisa jogar naquele tabuleiro e

retorna o maior ou menor valor (dependendo do jogador da rodada). Dentro da Alpha-Beta, portanto, a escolha do póximo board a ser jogado acontece por cada posição disponível do board atual determinado pela jogada anterior ou, em caso da jogada anterior enviar para um local invalido, pela função heurística auxiliar. É importante salientar que, como qualquer heurística, ela não gera um resultado ótimo e sim aproximado, pois visualiza apenas o melhor tabuleiro à ser jogado naquele determinado momento, ignorando a próxima jogada. Ela é executada justamente para evitar que a minimax precise percorrer todos os nós possíveis para descobrir qual é o melhor tabuleiro a ser jogado, causando uma grande redução do esforço computacional do algoritmo.

Utilizamos um array como estrutura para armazenar os boards e um array para armazenar as posições de cada um deles. Dessa forma, pudemos utilizar a própria pilha de recursão como uma estrutura auxiliar. O algoritmo de Minimax portanto analisa, partindo de um determinado estado até uma profundidade N qual a melhor jogada naquela ocasião.

8.Complexidade do Algoritmo

Com um aproximado fator branching (número de filhos por nó) de b e uma profundidade de busca de d turnos, sabemos que a complexidade da minimax com alpha-beta pruning é $O(b^{d/2})$, porém para cada execução do algoritmo é chamada a função heurística auxiliar, cuja complexidade é O(b), visto que os if atuam em tempo contante, inclusive no método check_heuristic(). Com isso, a complexidade do algoritmo Alpha-Beta implementado é $O(b^{d/2+1})$.

9. Conclusão

O algoritmo de Alpha-Beta pruning causa uma melhoria bastante perceptível no tempo de execução de uma minimax. Em um jogo como Ultimate-Tic-Tac-Toe, onde as possibilidades chegam à $5.8 * 10^{120}$, o tempo que levaria para calcular todas elas seria imenso e seria impossível manter um jogo "fluido". A alpha-beta, apesar de ser responsável pelo corte de diversos ramos, não é suficiente para o cálculo rápido dos casos do jogo, por isso é importante o uso de um limite de profundidade na árvore e uma heurística para casos de escolhas mais complexas.

Durante a execução, podemos perceber que quando marcamos a profundidade da minimax para valores um pouco maiores, o tempo que leva para a finalização dos calculos aumenta bastante. Por isso, configuramos o sistema para que o algoritmo se aprofunde mais quando o numero de posições marcadas for maior, desta forma, podemos garantir um resultado satisfatório em um tempo relativamente curto. Ainda assim, algumas vezes é notório um tempo maior para a conclusão da tomada de decisão. Isto se deve pela quantidade de estados a serem verificados que cresce exponencialmente de acordo com a profundidade da árvore minimax.

Bibliografias

Ultimate Tic Tac Toe. Available at: https://pclub.in/site/sites/default/files/Ultimate_Tic_Tac_Toe_Doc.pdf

Minimax (2016) in *Wikipedia*. Available at: https://en.wikipedia.org/wiki/Minimax .

T. Hajiaghayi, M. Available at: http://www.cs.umd.edu/~hajiagha/474GT15/Lecture12122013.pdf

Soma-zero (no date) in *Wikipedia*. Available at: https://pt.wikipedia.org/wiki/Soma-zero .

CS 161 recitation notes - Minimax with Alpha Beta pruning Available at: http://web.cs.ucla.edu/~rosen/161/notes/alphabeta.html .

CSCI Tutorials (2014) *CSCI 6350 artificial intelligence: Minimax and alpha-beta pruning Algorithms and Pseudocodes*. Available at: https://www.youtube.com/watch?v=J1GoI5WHBto .

Alexander Brudno (2016) in *Wikipedia*. Available at: https://en.wikipedia.org/wiki/Alexander_Brudno

ANEXOS

```
/*
* calcula a minimax com alphabeta pruning e seta o posmin na melhor posicao possivel para O
* param: boardnum:     numero do board atual
*        depth:        profundidade maxima que ira percorrer a alphabeta
*        a:            valor do alpha
*        b:            valor do beta
*        player:       numero do jogador
*        tab:          tabuleiro atual
*        profundidade: profundidade maxima, igual ao depth, mas que servira apenas como
verificador
* return: a melhor jogada calculada
*/
int alphabeta(int boardnum, int depth, int a, int b, int player, Tabuleiro tab, int
profundidade){
   int aux;
   //casos base
   if(tab.at(boardnum)->check_win() == 1){
       //se esse board fechado der a vitoria para o player X
       if (tab.check_win() == 1){
           return 200;
       }
       return 100;
   }else if (tab.at(boardnum)->check_win() == -1){
       //se esse board fechado der a vitoria para o player O
       if (tab.check_win() == -1){
           return -200;
       }
       return -100;
   }else if (tab.at(boardnum)->check_draw() == 1){
       return 0;
   }else if (depth == 0){
       return tab.at(boardnum)->heuristic;
   }
   if(!tab.at(boardnum)->valid){
       boardnum = findK(&tab, boardnum, player);
   }

   //player 1
   if(player == 1){
       for(int i = 1; i <= 9; i++){
           if(tab.at(boardnum)->board[i] == '_'){
               tab.at(boardnum)->board[i] = 'X';
               aux = alphabeta(i, depth -1, a, b, 2, tab, profundidade);
               tab.at(boardnum)->board[i] = '_';
               if(a < aux){a = aux;}
               if(a >= b){break;}
           }
       }
       return a;
   }
```

```
    //player 2
    if(player == 2){
        for(int i = 1; i <= 9; i++){
            if(tab.at(boardnum)->board[i] == '_'){
                tab.at(boardnum)->board[i] = 'O';
                aux = alphabeta(i, depth -1, a, b, 1, tab, profundidade);
                tab.at(boardnum)->board[i] = '_';
                if(b > aux){
                    b = aux;
                    if(depth == profundidade){
                        posmin = i;
                    }
                }
                if(a >= b){break;}
            }
        }
        return b;
    }
}
```

```
/*
* Procura um tabuleiro valido e retorna o numero dele
* param: tab: tabuleiro atual
*        k:   numero do board atual
*        player: Jogador da vez
* return: numero valido de tabuleiro
*/
int findK(Tabuleiro *tab, int k, int player){
    int tempk = k;
    int htemp;
    //Seta o maior/menor caso possível dependendo do jogador.
    if(player==2){
        htemp = 200;
    }
    else{
        htemp = -200;
    }

    for(int i=1; i<=9; i++){
        if(player==2){
            if(tab->at(i)->valid && tab->at(i)->check_heuristic() < htemp){
                htemp = tab->at(i)->check_heuristic();
                tempk = i;
            }
        }
        else{
            if(tab->at(i)->valid && tab->at(i)->check_heuristic() > htemp){
                htemp = tab->at(i)->check_heuristic();
                tempk = i;
            }
        }
    }
    return tempk;
}
```