Universidade Federal de Santa Catarina

# Minimax com Poda $\alpha$ $\beta$ Versão Final

INE5430 - Inteligência Artificial

*Alunos*
Andrei Donati
Ígor Assis Rocha Yamamoto
Luis Felipe Pelison

*Professor*
Elder Rizzon Santos

Setembro de 2017

# Sumário

# 1 Problema

Este trabalho tem como objetivo o desenvolvimento de um jogo denominado "5 em Linha", também conhecido como *Gomoku*. A solução que será desenvolvida deve possuir Inteligência Artificial (I.A.), para competir com um jogador humano, aplicando o algoritmo *MiniMax* com podas $\alpha$ e $\beta$, um estilo de busca adversária. O jogo consiste de um tabuleiro $15x15$, onde o objetivo é formar uma sequência de 5 peças, em qualquer direção.

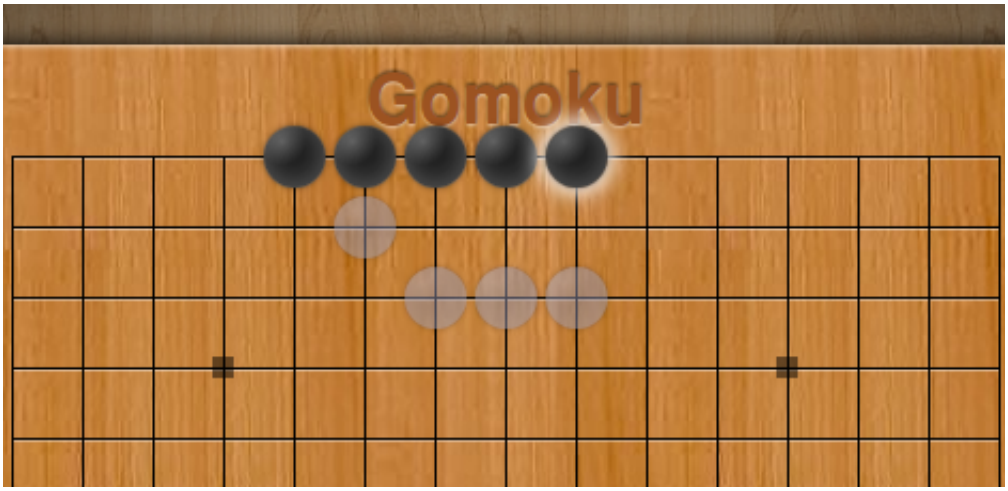


Figura 1: Exemplo de um *Gomoku* onde as peças pretas venceram. Fonte do jogo: http://gomoku.yjyao.com/

Alguns requisitos do projeto são:

- Um estudo sobre possibilidades de cálculo para a utilidade e heurística;
- Explicações sobre a estrutura de dados escolhida, limitações do projeto e os problemas enfrentados;
- Desenvolver e explicar o papel das principais classes e métodos;
- Explicar a técnica utilizada para detecção de fim de jogo e quando ela é aplicada;
- Explicar e desenvolver a detecção de sequencias de peças;
- O desenvolvimento de uma interface gráfica, onde podem jogar humanos e computador (selecionando quem irá iniciar a partida).

Nessa etapa final do projeto, foi desenvolvido a matemática e as técnicas de cálculo da utilidade e heurística, as estratégias e a estrutura de dados que modela os estados do jogo, informando as principais classes e métodos, otimizações implementadas, a detecção de sequencias de peças, a detecção de fim de jogo, o código-fonte utilizado, os problemas e limitações da solução e ideias para melhorias futuras.

# 2 Definição matemática da utilidade e heurística

Na Inteligência Artificial, algumas técnicas são utilizadas em conjunto para a realização da busca de estados. O cálculo de funções como utilidade e heurística são essenciais para o funcionamento do sistema de I.A., já que são elas que guiam o algoritmo em busca da melhor jogada, e consequentemente da vitória. Para a resolução do problema proposto, será usado o algoritmo *MiniMax* com poda $\alpha$ $\beta$. Esse é um algoritmo que requer a busca em profundidade. Para tal, inicialmente foi realizado um trabalho experimental a fim de decidir alguns parâmetros gerais para a heurística. Após algumas conclusões, segue as principais que serão aplicadas neste trabalho:

- Priorizar jogadas na região central do tabuleiro;
- Levar em consideração o número de peças que compõem uma sequência;
- Diferenciar sequências que têm aberturas maiores (não estão trancadas por peças adversárias ou fim do tabuleiro);
- Considerar sequências que não podem chegar a 5 peças, pois estão próximos do final do tabuleiro, por exemplo;
- Considerar o aumento do número de profundidade como algo ruim.

Com base nisso, juntamos as ideias inicias e colocamos em um formato matemático, para depois implementar no algoritmo. Para isso, dividimos em conjuntos as sequências possíveis e suas aberturas. Por exemplo, um conjunto é:

- "$Quartet\_2Opens$" - Significa que é uma sequência de 4 peças, por isso $Quartet$, com 2 aberturas possíveis ($2Opens$), ou seja, um "vazio"de cada lado da sequência.

Após isso, ordenamos todos os conjuntos, de forma que o mais prioritário ficasse em primeiro. Então, demos valores matemáticos (pesos), que irão multiplicar pela quantidade de vezes que aquela sequência aparece. No final, montamos uma função matemática que recebe o estado e calcula sua função heurística ou utilidade.

Os conjuntos e seus valores numéricos ficaram dessa forma:

- "Quintet": 200000
- "Quartet_2Opens": 120000

- "Quartet_1Open": 50000
- "Triplet_2Opens": 30000
- "Triplet_1Open": 15000
- "ProbQuartet_2Opens": 7000
- "ProbQuartet_1Open": 3000
- "Double_2Opens: 1000
- "Double_1Open": 400
- "ProbTriplet_2Opens": 100
- "ProbTriplet_1Open": 40

Onde 'Quintet' corresponde a uma sequência de 5 peças, 'Quartet': 4 peças, 'Triplet': 3 peças, 'Double': 2 peças e 'Prob' corresponde a uma sequência com um vazio no meio, por exemplo, 'ProbQuartet' é uma sequência de 2 peças + um vazio + 1 peça. Todas essas peças sendo de um jogador só. Os valores foram decididos de forma que as melhores jogadas tenham maior valor, em qualquer circunstância, para que o algoritmo sempre escolha pela melhor/maior.

Com isso realizado, também definimos uma matriz "máscara"que dará maiores pesos para jogadas na região central do tabuleiro. Dessa forma, a heurística tenderá o algoritmo por jogadas nessa região. Assim, a matriz que será multiplicada, elemento a elemento, pela matriz de estado, será a seguinte matriz M:

```
[[0, 0, 0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0, 0, 0],
[0, 0, 0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0, 0, 0],
[0, 0, 0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0, 0, 0],
[0, 0, 0, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0, 0, 0],
[0, 0, 0, 0.2, 0.4, 0.4, 0.4, 0.4, 0.4, 0.4, 0.4, 0.2, 0, 0, 0],
[0, 0, 0, 0.2, 0.4, 0.6, 0.6, 0.6, 0.6, 0.6, 0.4, 0.2, 0, 0, 0],
[0, 0, 0, 0.2, 0.4, 0.6, 0.8, 0.8, 0.8, 0.6, 0.4, 0.2, 0, 0, 0],
[0, 0, 0, 0.2, 0.4, 0.6, 0.8, 1.0, 0.8, 0.6, 0.4, 0.2, 0, 0, 0],
[0, 0, 0, 0.2, 0.4, 0.6, 0.8, 0.8, 0.8, 0.6, 0.4, 0.2, 0, 0, 0],
[0, 0, 0, 0.2, 0.4, 0.6, 0.6, 0.6, 0.6, 0.6, 0.4, 0.2, 0, 0, 0],
[0, 0, 0, 0.2, 0.4, 0.4, 0.4, 0.4, 0.4, 0.4, 0.4, 0.2, 0, 0, 0],
[0, 0, 0, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0, 0, 0],
[0, 0, 0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0, 0, 0],
[0, 0, 0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0, 0, 0],
[0, 0, 0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0, 0, 0]]
```

Juntando tudo isso, a nossa função heurística final ficou da seguinte maneira:

$Heuristic = (NumberOfQuintet*200000+NumberOfQuartet\_2Opens*120000+NumberOfQuartet\_1Open*50000+NumberOfTriplet\_2Opens"*30000+NumberOfTriplet\_1Open*15000+NumberOfProbQuartet\_2Opens*7000+NumberOfProbQuartet\_1Open*3000+NumberOfDouble\_2Opens*1000+NumberOfDouble\_1Open*400+NumberOfProbTriplet\_2Opens*100+NumberOfProbTriplet\_1Open*40)+SumElements(StateMatrix.*M)$

Onde StateMatrix representa a entrada da função, sendo a matriz do estado. O operador ".*"representa a multiplicação elemento a elemento, e não a multiplicação tradicional de matrizes. E, por fim, o "SumElements"representa o somatório de todos os elementos da matriz resultado.

Além da função heurística, também temos a função utilidade. Essa, será aplicada em estados finais, para avaliar a pontuação de um nó folha. A diferença dessa função para a heurística é que ela deve resultar um valor maior e também não terá a matriz de pesos para a região central. Assim, a função utilidade ficará da seguinte forma:

$Utility = 2*(NumberOfQuintet*200000+NumberOfQuartet\_2Opens*120000+NumberOfQuartet\_1Open"*50000+NumberOfTriplet\_2Opens"*30000+NumberOfTriplet\_1Open*15000+NumberOfProbQuartet\_2Opens*7000+NumberOfProbQuartet\_1Open*3000+NumberOfDouble\_2Opens*$

$1000 + NumberOfDouble\_1Open * 400 + NumberOfProbTriplet\_2Opens * 100 + NumberOfProbTriplet\_1Open * 40)$

# 3 Estratégias e Otimizações

O capítulo anterior trouxe como foi pensado e calculado os valores para as funções heurística e utilidade. Assim, ele não deixa de ser uma parte da nossa estratégia para a resolução do problema.

Seguindo, então, com o desmembramento de nossa estratégia, procuraremos deixar mais claro nessa parte do documento como foi realizado a implementação dos estados do tabuleiro na linguagem de programação, o papel das principais classes e métodos que utilizamos, algumas limitações, problemas encontrados e também otimizações realizadas

## 3.1 Estratégias Iniciais

Como primeira estratégia, escolhemos a linguagem de programação mais familiar para os integrantes do grupo, Python. Com isso em mãos, realizamos algumas pesquisas sobre como esse tipo de abordagem de inteligência artificial em jogos é utilizada por outras pessoas no mundo, seja na academia ou como *hobbie*. Achamos dois artigos que falavam sobre ([1] e [2]), além de muitas pessoas interessadas no desenvolvimento do mesmo jogo na internet. Isso nos deu alguma experiência para tomarmos outras decisões.

Decidimos, também, escolher uma abordagem de orientação a objetos. Assim, iremos relatar as principais classes agora. O código-fonte do projeto pode ser observado no link a seguir: *https://github.com/igoryamamoto/ine5430-gomoku*

## 3.2 Tabuleiro

Para os estados do jogo/tabuleiro, implementamos um *array* de 2 dimensões, uma matriz. Essa matriz é $15x15$ visivelmente, mas também é $17x17$ no cálculo da heurística, pois inserimos as bordas como sendo o carácter '2' nas linhas 0 e 16 e colunas 0 e 16. Além disso, a parte central, $15x15$, começa zerada, ou seja, com todos os elementos iguais a '0'. O '0' nos indica que não há peças ali, é vazio. Depois, será inserida peça por peça, sendo a peça do jogador 1 igual a '1' e a peça do jogador 2 igual a '-1'.

## 3.3 Estrutura e Principais Classes

O Código foi dividido em 3 arquivos: *main.py*, *heuristic.py* e *minimax.py*. O primeiro, *main.py*, trata-se do principal arquivo para o programa ser executado. Nele, estão presentes a classe controladora do jogo, chamada de *Game* e *State*, e os elementos da interface gráfica, os quais explicaremos na próxima secção. Sobre a classe controladora do jogo, ela possui vários atributos e métodos. Vamos explicar os principais para o seu entendimento. Como atributos, a classe *Game* possui o atual valor para a heurística, o valor da heurística da melhor jogada (que será encontrada pelo algoritmo de busca *Minimax*), os próximos movimentos (lista de 2 elementos, um valor para $x$ e outro para $y$), o estado atual, que é instanciado da outra classe que temos, além dos necessários para a interface gráfica. Na classe *State*, a segunda mais importante, temos o jogador que está jogando (sendo 1 para o primeiro e -1 para o segundo) e o *board* ou tabuleiro. Além disso, essa classe fica responsável por fazer a ligação com os outros dois arquivos, calculando a heurística da jogada, as possíveis jogadas (os filhos), a atualização do tabuleiro e a verificação de estados folhas. Para explicar melhor os métodos, iremos entrar no fluxo do jogo, assim, ver passo-a-passo como o programa se comporta por baixo dos panos, entrando em cada método mais importante. Observe que, no código que está em anexo e disponível no *http://github.com/igoryamamoto/ine5430-gomoku*, pode ser visto alguns comentários indicando a sequência dos passos, cujo estão dentro do padrão $|1st.|$. Então, no início de tudo, a classe *Game* monta a interface gráfica com o método *createButtons* e outros atributos do *Tkinter*. Após, ele verifica quem serão os jogadores, se são humanos ou não e se querem ser o primeiro ou o segundo. Caso não sejam humanos, o principal método que agirá será o chamado de *PCMove*. Esse é responsável pelas jogadas da inteligência artificial. Caso o jogador seja humano, um evento de clique no botão da interface gerará a ação do método *ClickLeft*. Essas duas funções/métodos são responsáveis pelas jogadas, tanto da IA quanto do humano. Elas precisam jogar, ou seja, marcar um botão com uma cor, e verificar se o jogo acabou o não. Isso é feito com o método *is_gameover* que herda a propriedade da classe *State*, chamada de *is_terminal*, a qual vai em outro arquivo, o *heuristic.py*, que possui o método *hasWinnerSeq*. Esse método verifica se o padrão 'xxxxx' aconteceu. Como funciona o algoritmo e o método para verificar os padrões que ocorrem, falaremos mais adiante.

## 3.4 Interface Gráfica

Para a interface gráfica, optamos por uma biblioteca consolidada em *Python*, chamada *Tkinter*. Nela, o jogador já escolhe logo que abre a in-

terface, se deseja iniciar o jogo ou não. Ela está presente no arquivo *main.py*

Optamos por uma apresentação com botões, representando os espaços. Esses botões são marcados por -1 e 1, dependendo do jogador, ou 0 se não tiver nenhum acionado.

O Jogador 1 sempre será de cor preta e o -1 de cor branca.

Na interface, existe a opção de sair do jogo: Quit or recomeçar: Reset.

No início de cada jogo, existem mensagens que perguntam quem o jogador quer ser e se ele é humano. Assim, as possibilidades são: Humano x Humano, Humano x Máquina, Máquina x Máquina, podendo intercalar entre jogador 1 ou jogador -1.

## 3.5 Otimizações realizadas

Quanto às otimizações, utilizamos algumas técnicas:

- Fazer a busca do algoritmo *MiniMax* em estados com um raio de 3 peças distantes das peças presentes no tabuleiro, em vez de pesquisar no tabuleiro inteiro.
- Implementar a geração dos filhos de forma a priorizar os estados com peças no meio do tabuleiro.
- Depois de encontrar uma sequência de peças, saltar algumas casas, a fim de procurar por outro padrão não sobreposto ao anterior.
- Começar a analisar em uma profundidade maior de jogadas a partir da oitava peça jogada.

Porém, mesmo assim o desempenho não foi satisfatório, como contaremos na última seção. Mas, as otimizações foram de grande ajuda.

Outros adicionais que aplicamos foram: o modo humano x humano, IA x IA e uma interface agradável, mesmo sendo herdada do Tk/tcl. O modo IA x IA é representado na figura 2 abaixo.

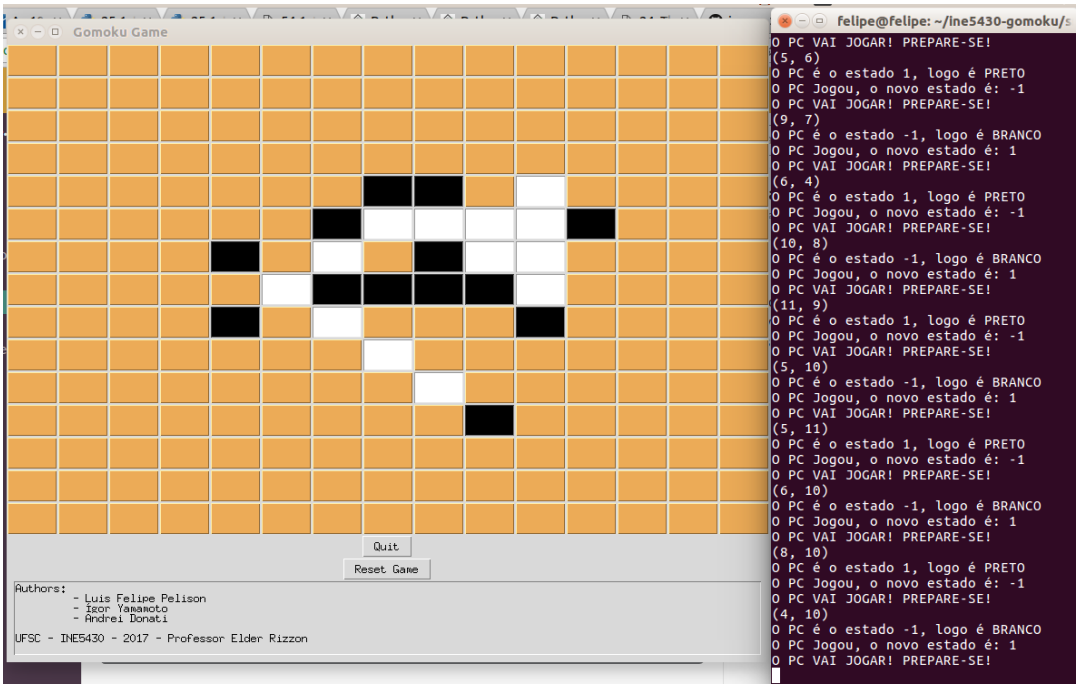


Figura 2: Duas I.A's jogando contra. O jogo estava no meio. Não sabemos quem ganhou.

# 4 Detecção de sequência de peças

Este capítulo será dedicado a tratar de como foi feita a detecção de sequências de peças de um mesmo jogador no tabuleiro. Ele é de extrema importância visto que toda a heurística e função utilidade se baseará nas sequências de peças formadas no tabuleiro. O algoritmo de detecção de sequência é muito similar, independente do número de peças em sequência que busca ser detectado e consiste em:

1. Produzir vetores de todas as linhas do tabuleiro
2. Produzir vetores de todas as colunas do tabuleiro
3. Produzir vetores de todas as diagonais maiores que 4 do tabuleiro
4. Substituir os valores dos vetores para que eles tenham "e"em um espaço *empty* (vazio, possível de receber uma peça), "x"no espaço com uma peça do jogador e "n"no espaço *not possible* (não pode receber uma peça, pois é borda ou uma peça do adversário)

5. Verificar se, em algum dos vetores produzidos, existe alguma sequência pré-definida de peças

As sequências pré-definidas citadas no último passo do algoritmo são sequências de peças que são associadas a valores determinados de heuristica. Um exemplo de sequência é quatro peças de um mesmo jogador com ambas as pontas livres; outro exemplo é uma sequência também de quatro peças de um mesmo jogador mas agora com apenas um dos lados livres.

Mesmo sendo um capítulo voltado para detecção de sequência, é importante destacar que como uma otimização foi adicionado na heurística, toda a sequência de peças deve observar, no mínimo, 5 posições no tabuleiro, mesmo se, por exemplo, a busca é para uma sequência de apenas três peças. Isso deve acontecer pois caso encontremos uma situação como a da imagem 3, que apesar da próxima jogada formar uma quádrupla, nunca será possível ganhar o jogo. Assim, a heurística desta sequência deve ser 0.

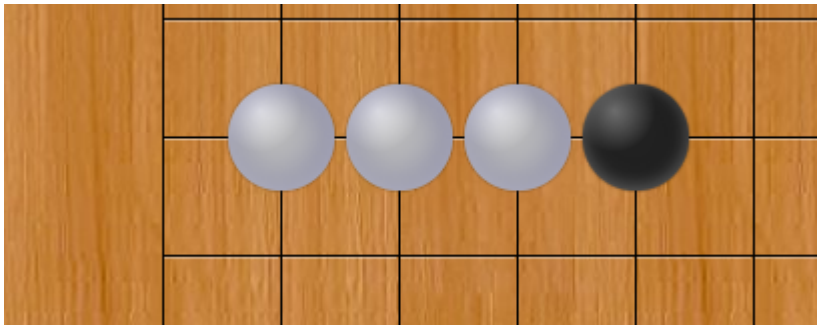

Figura 3: Situação onde a sequência de peças deve ter heurística 0.

## 4.1 Detecção de sequências de quatro, três e duas peças

A detecção de todas essas sequências se dá de uma maneira bastante parecida. Após todos os vetores de linhas, colunas e diagonais serem produzidos, uma *string* representando o padrão da sequência de peças buscada, que está armazenada em um dicionário de dados, será procurada e contada. Para o caso de duas peças em sequência, temos quatro possibilidades de padrões, considerando a distinção entre peças consecutivas com e sem ambos os lados livres: ['eexxe'], ['exxee'], ['nxxeee'] e ['eeexxn'], lembrando que "e"representa um espaço *empty* (possível de receber uma peça), "x"representa um espaço com uma peça do jogador e "n"um espaço *not possible* (não pode receber uma peça). Assim como na busca por uma sequência de duas peças, a busca por uma sequência de três peças está focada em encontrar o padrão de *strings*: ['exxxe'], ['nxxxee'] e ['eexxxn'] e para os quartetos os padrões buscados são:

['nxxxxe'], ['exxxxn'] e ['exxxxe']. Há também outras sequências buscadas, como a de um provável trio (duas peças em sequência seguido de uma abertura e mais uma peça) e um provável quarteto (três peças em sequência, seguido de um espaço vazio e mais uma peça).

É importante destacar os motivos da decisão desta implementação de busca por *strings*. O primeiro motivo é a rapidez de busca, visto que a linguagem oferece bibliotecas que otimizam o trabalho deixando o sistema como um todo mais rápido. Em segundo lugar foi a facilidade de implementação, também por existir bibliotecas que já apresentam funções prontas.

Maiores detalhes da implementação estão dentro do código *heuristic.py*, do código no *github*.

### 4.2 Detecção de sequências de cinco peças e fim de jogo

A detecção de fim de jogo se dá de duas maneiras: detecção de cinco peças ou falta de lugares para jogar. A primeira opção se dá de uma maneira muito similar a detecção de sequência de 4 ou 3 peças, porém neste caso a *string* a ser buscada é ['xxxxx'].

## 5 Código-fonte

O código fonte está em anexo ao envio do relatório e também pode ser encontrado no repositório https://github.com/igoryamamoto/ine5430-gomoku

## 6 Limitações e Problemas da Solução

Algumas limitações no desenvolvimento do projeto e melhorias podem ser apontadas devido ao limite de tempo disponível. Entre elas podemos elencar:

- O algoritmo de detecção das sequências de peças que compoem o sistema de pontuação da função heurística deve ser otimizado a fim de reduzir o tempo computacional. Este cálculo configura-se em uma limitação na busca em camadas mais profundas por inviabilizar a prática do jogo em tempo hábil.
- Além de otimizações no algoritmo de detcção, poderíamos pensar em soluções para paralelizar a computação desta busca.

- O algoritmo *MiniMax* implementado permite a seleção do nível de profundidade da busca como um parametro, porém devido ao custo computacional elevado da implementação, a seleção de níveis de profundidades grandes é limitada.

# 7 Melhorias Futuras

O grupo não se satisfez com a solução apresentada, apesar de respeitar a maioria dos requisitos do projeto e também conseguir entregar o objetivo. Pensamos em melhorar na velocidade do jogo, implementando alguma técnica de programação paralela. Também, técnicas para jogadas mais rápidas podem ser implementadas no futuro, como escolher por uma possibilidade que seja muito boa em relação a uma média atual do valor da heurística dos filhos gerados até um momento mínimo. Outra ideia é procurar por soluções mais rápidas, tanto na implementação do algoritmo quanto da busca de padrões (principalmente.)

Outra melhoria seria uma interface ainda mais amigável, mostrando que o IA está calculando sua melhor jogada e atualizando na tela em qual nível de profundidade ela se encontra. Isso não foi realizado aqui pois limitamos a 2 níveis de profundidade (devido a demora do algoritmo) e falta de um conhecimento profundo da biblioteca de interface gráfica.

# Referências

[1] Kulikov J. Self-teaching Gomoku player using composite patterns with adaptive scores and the implemented playing framework. Tallinn University Of Technology.

[2] L.V. Allis, H.J. van den Herik, M.P.H. Huntjens. Go-Moku and Threat-Space Search. University of Limburg and Vrije Universiteit.