**Logotipo

Descrição gerada automaticamenteUNIVERSIDADE FEDERAL DA GRANDE DOURADOS**

**Faculdade de Ciências Exatas e Tecnologia**

**Curso de Engenharia De Computação - UFGD**

**Maykoll Rocha**

**Documentação do Jogo Competitivo usando**

**Algoritmo MaxMin com poda Alpha-Beta**

**Maykoll Rocha**

**Documentação do Jogo Competitivo usando**

**Algoritmo MaxMin com poda Alpha-Beta**

Trabalho I de Inteligência Artificial, construção de um jogo competitivo em uma linguagem de preferência do discente que apresento o algoritmo de MaxMin com poda AlphaBeta. Este projeto tem como visão concluir a matéria de Inteligência Artificial sendo uma matéria obrigatória do meu curso.

Resumo:

Dentro das normas avaliativas da disciplina de Inteligência Artificial, ministrada pelo professor Willian Paraguassu Amorim no curso de Sistemas de Informação, foi solicitado o desenvolvimento de dois trabalhos. O primeiro consiste na criação de um jogo com dois modos de jogo: Humano x Humano e Humano x Máquina, sendo o último com diferentes níveis de dificuldade. O aluno tem liberdade para escolher o jogo e a linguagem de programação, desde que seja apresentada uma interface gráfica.

Escolhi desenvolver o jogo da velha utilizando Python, pela sua facilidade de uso e suporte nativo para interfaces gráficas, sem necessidade de instalar bibliotecas adicionais. O desenvolvimento inicial foi feito no Colaboratory, sem interface gráfica, para facilitar testes e correções rápidas. Posteriormente, migrei o projeto para o Visual Studio Code para implementar a interface gráfica.

Sumário

[1.Introdução 5](#_Toc177476044)

[2. Requisitos 6](#_Toc177476045)

[3. Arquitetura do Sistema 7](#_Toc177476046)

[3.1 Diagrama de arquitetura 7](#_Toc177476047)

[3.2 Componentes Principais 7](#_Toc177476048)

[4. Estrutura do Código 9](#_Toc177476049)

[4.1 Descrição dos arquivos e pastas 9](#_Toc177476050)

[4.2 Principais Funções ou Classes 10](#_Toc177476051)

[5. Algoritmos Implementados 12](#_Toc177476052)

[5.1 Descrição dos Algoritmos 12](#_Toc177476053)

[5.2 Complexidade 13](#_Toc177476054)

[6. Instruções de Instalação 15](#_Toc177476055)

[7. Interface Gráfica 16](#_Toc177476056)

[8. Considerações sobre Performance 19](#_Toc177476057)

[9. Referências 23](#_Toc177476058)

# 1.Introdução

O presente projeto tem como principal objetivo a implementação de um jogo da velha em Python, focado na integração de uma Inteligência Artificial (IA) para aprimorar a experiência de jogo. O jogo segue as regras clássicas: um tabuleiro de 3x3, composto por nove espaços vazios, onde dois jogadores, representados pelos símbolos "X" e "O", alternam suas jogadas. A vitória é alcançada pelo primeiro jogador que alinhar três símbolos idênticos em sequência, seja em linha, coluna ou diagonal. Caso todas as nove posições sejam ocupadas sem que nenhum jogador consiga alinhar três símbolos, o resultado é declarado como empate.

Além de proporcionar a jogabilidade tradicional, o projeto também visa destacar o comportamento da IA durante a partida. Para esse fim, foi implementada uma funcionalidade de debug, que pode ser ativada diretamente pela interface gráfica, permitindo ao usuário acompanhar em tempo real as decisões tomadas pela IA. A interface gráfica, desenvolvida para ser intuitiva e funcional, será discutida em detalhes mais adiante, com explicações sobre seus recursos e formas de utilização.

Na seção dedicada à IA, será apresentada sua concepção teórica, juntamente com a implementação do algoritmo Minimax e sua otimização por meio da poda Alfa-Beta. Serão discutidas também as heurísticas utilizadas para definir as melhores jogadas e a complexidade computacional do algoritmo. O código contém comentários explicativos em cada método, além de breves descrições dos atributos, com o intuito de facilitar a compreensão do fluxo do programa.

O projeto foi desenvolvido utilizando a linguagem Python, versão 3.10.11, garantindo compatibilidade com bibliotecas modernas e desempenho eficiente para o objetivo proposto.

# 2. Requisitos

Para o desenvolvimento e execução do projeto, são necessárias as seguintes dependências e configurações de software:

* **Python 3.10.11**: Versão da linguagem de programação utilizada para o desenvolvimento do jogo.
* **Tkinter**: Biblioteca padrão do Python utilizada para a criação da interface gráfica.
* **Random**: Biblioteca do Python usada para as jogadas aleatórias do modo fácil da IA.
* **Time**: Utilizada para adicionar delays e melhorar a experiência visual durante a jogabilidade.
* **Math:** Biblioteca utilizada para fazer alguns cálculos complexos que tem saídas melhores utilizando a biblioteca.
* **Secrets:** Biblioteca utilizada para gerar números e buscar valores aleatória o mais aleatórios possível.

Essas bibliotecas são instaladas automaticamente junto com o Python, exceto algumas versões específicas de outras bibliotecas, que podem ser instaladas por meio do gerenciador de pacotes pip se necessário. Porém como não foi utilizado nenhuma versão específica não será necessário uso desse artifício. Não tem retenção por parte de SO as biblioteca é presente tanto em Linux quanto em Windows e levando em conta ambas provavelmente MAC OS também rodará.

# 3. Arquitetura do Sistema

## 3.1 Diagrama de arquitetura

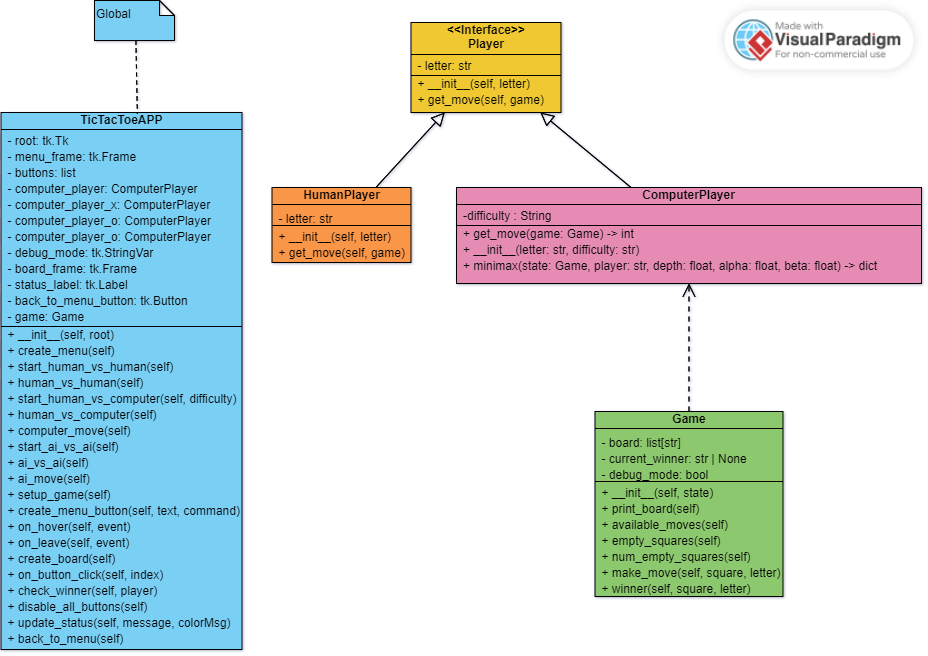


Imagem 1 – Diagrama de Classe UML

## 3.2 Componentes Principais

O código é estruturado em quatro classes principais e uma interface, cada uma desempenhando um papel essencial para o funcionamento do jogo da velha. A seguir, descrevemos a coordenação dessas classes no sistema:

A classe mais importante é a Game, que é responsável por gerenciar toda a lógica de negócios e regras do jogo, servindo como o back-end do sistema. Ela realiza a conexão direta com a interface gráfica e coordena o fluxo de dados entre os jogadores (humanos ou IA), as jogadas e o estado do tabuleiro. Entre suas principais responsabilidades estão:

**Controle do Tabuleiro**: A Game mantém o estado atual do tabuleiro e verifica as condições de vitória ou empate.

**Gestão de Turnos**: de quem é a vez, alternando entre o jogador humano e o computador (IA).

**Conexão com a Interface**: responsável por enviar as atualizações necessárias para a interface gráfica, garantindo que as jogadas sejam refletidas visualmente.

A Game também interage com as classes que representam os jogadores e a IA, recebendo as jogadas e atualizando o estado do jogo conforme necessário.

A classe que implementa a parte do front-end é a TicTacToeAPP. Ela faz uso da biblioteca Tkinter para criar a interface gráfica (GUI) que os usuários veem e com a qual interagem. Esta classe é responsável por:

**Criação da Interface Visual**: Constrói a janela principal do jogo, desenhando o tabuleiro, os botões de interação e exibindo as mensagens de estado (como "Vitória", "Empate" ou "Próximo Jogador").

**Tratamento de Eventos**: Captura e gerencia os cliques dos jogadores nos botões do tabuleiro, enviando essas informações para a classe Game, que processa as jogadas.

**Atualização Visual**: a cada jogada, o tabuleiro é atualizado visualmente para refletir as mudanças feitas pelo jogador humano ou pela IA.

Essa classe garante uma experiência de usuário fluida, exibindo o progresso do jogo e permitindo a interação intuitiva.

As classes ComputadorPlayer e HumanoPlayer são responsáveis por representar os diferentes tipos de jogadores no jogo. Ambas são derivadas de uma interface comum chamada Player, que define os métodos e comportamentos essenciais que todo jogador deve implementar. A interface Player serve como uma classe abstrata que garante que tanto os jogadores humanos quanto a IA sigam uma estrutura consistente.

**Interface Player**: define métodos abstratos que são implementados pelas classes ComputadorPlayer e HumanoPlayer, como o método get\_move(), que retorna a jogada que cada jogador deseja fazer.

**Classe HumanoPlayer**: representa o jogador humano e captura suas interações (cliques) na interface gráfica, passando-as para a classe Game. Essa classe utiliza a interface gráfica para obter a jogada escolhida pelo usuário.

**Classe ComputadorPlayer**: representa a IA do jogo, utilizando o algoritmo Minimax com poda Alfa-Beta para calcular a melhor jogada. Esta classe também implementa o método get\_move(), mas ao contrário do jogador humano, ela utiliza o estado do tabuleiro e a heurística do algoritmo para decidir sua ação.

Essas classes são a ponte entre os inputs (humanos ou computacionais) e o gerenciamento centralizado da lógica do jogo feito pela Game.

# 4. Estrutura do Código

Nessa sessão será apresentado para gente a descrição dos arquivos e pastas explicação da organização do código e as principais funções ou classes com breve explicação de cada função ou classe importante.

## 4.1 Descrição dos arquivos e pastas

O código do projeto está organizado de maneira a separar as responsabilidades e garantir que cada parte do sistema seja modular e fácil de manter. A seguir, a descrição da organização dos principais arquivos e pastas:

* **/ (Diretório Raiz)**: A pasta raiz do projeto contém todos os arquivos principais necessários para a execução do jogo da velha. Ela inclui o código-fonte e qualquer script de inicialização do jogo.
  + **main.py**: Arquivo principal responsável pela execução do jogo. Ele inicia a interface gráfica e conecta as classes principal do sistema. Este arquivo é o ponto de entrada do programa.
  + **/imgs/ (Pasta de Recursos)**: Contém os recursos visuais do jogo, como ícones, imagens do tabuleiro e fontes personalizadas utilizadas pela interface gráfica.
    - **icon.png**: Ícone usado na janela do jogo.
  + **/packs/ (Código-fonte)**: Contém os arquivos de código-fonte que implementam a lógica do jogo, a inteligência artificial e a interface gráfica.
    - **/\_\_pycache\_\_/**: armazenar arquivos compilados em bytecode para acelerar o tempo de carregamento do Python.
    - **TicTacToeApp**.py: Implementa a interface gráfica utilizando a biblioteca Tkinter. Este arquivo contém a classe TicTacToeAPP, que gerencia a interação visual com o usuário e desenha o tabuleiro e os controles.
    - **Game.py**: Contém a classe Game, que gerencia a lógica central do jogo. Ele controla o estado do tabuleiro, verifica condições de vitória/empate e coordena a interação entre os jogadores e a interface gráfica.
    - ComputadorPlayer**.py**: Contém a implementação da classe ComputadorPlayer, responsável por calcular as jogadas da IA usando o algoritmo Minimax com poda Alfa-Beta.
    - Player**.py**: Define as classes Player (interface), HumanoPlayer e ComputadorPlayer. Essas classes representam os jogadores (humanos ou IA) e são responsáveis por tomar decisões de jogada.
    - HumanoPlayer**.py**: Contem a implementação de como é pego a jogada dos humanos durante a partida.
    - GlobalVars.py: esse arquivo não carrega classes é usada somente para fins gráficos ela salva algumas alterações de cores, botões e fundos e depois envia para a interface principal.
  + **README.md**: Arquivo de documentação básica do projeto, que explica como instalar as dependências, rodar o jogo e o que cada componente do sistema faz.

## 4.2 Principais Funções ou Classes

**1. Classe Game**

**Descrição**: A classe principal responsável pela lógica do jogo da velha. Ela gerencia o estado do jogo, verifica as condições de vitória ou empate e coordena as interações entre os jogadores e a interface gráfica.

**Funções Principais**:

* **\_\_init\_\_()**: Inicializa o estado do tabuleiro e define os jogadores. Reseta as variáveis para um novo jogo.
* **make\_move(square, letter)**: Processa a jogada de um jogador em uma posição específica no tabuleiro, verificando se é uma jogada válida.
* **winner()**: Verifica se há um vencedor, retornando o jogador que venceu ou indicando empate.

**2. Classe TicTacToeAPP**

**Descrição**: Responsável por criar e gerenciar a interface gráfica (front-end) do jogo, utilizando a biblioteca Tkinter. Ela exibe o tabuleiro, captura as interações dos jogadores e atualiza visualmente o estado do jogo.

**Funções Principais**:

* **\_\_init\_\_()**: Inicializa a janela principal e pré-set de parâmetros.
* **create\_menu()**: faz a criação do menu e adicionando os eventos da interface.
* setup\_game(): faz a destruição de todas as janelas anteriores para a inicialização da tela principal.
* **create\_board()**: Desenha o tabuleiro na interface com os botões representando as células.
* **update\_board()**: Atualiza o visual do tabuleiro com base nas jogadas feitas.
* **back\_to\_menu()**: Retorna para o menu principal após concluir o jogo.

**3. Classe Player (Interface)**

**Descrição**: Interface que define as funções básicas que todo jogador (humano ou IA) deve implementar.

**Funções Principais**:

* **get\_move()**: Método abstrato que será implementado nas classes derivadas para retornar a jogada de um jogador.

**4. Classe HumanoPlayer (Herda de Player)**

**Descrição**: Representa o jogador humano. Captura as interações da interface gráfica e retorna a jogada escolhida pelo usuário.

**Funções Principais**:

* **get\_move()**: Captura a jogada do jogador humano com base no clique no tabuleiro.

**5. Classe ComputadorPlayer (Herda de Player)**

**Descrição**: Implementa o jogador controlado pelo computador, utilizando o algoritmo Minimax com poda Alfa-Beta para escolher a melhor jogada.

**Funções Principais**:

* **get\_move()**: Retorna a jogada calculada pela IA utilizando o algoritmo Minimax.
* **minimax()**: Implementa o algoritmo Minimax para avaliar as possíveis jogadas e escolher a mais vantajosa para o computador.

# 5. Algoritmos Implementados

## 5.1 Descrição dos Algoritmos

O algoritmo Minimax é uma técnica fundamental utilizada em jogos de dois jogadores, como o jogo da velha, para determinar a melhor jogada possível. A ideia central do Minimax é simular todas as possíveis jogadas que podem ocorrer a partir de um estado atual do jogo e escolher a jogada que maximiza a vantagem do jogador enquanto minimiza a vantagem do adversário.

O algoritmo constrói uma árvore de decisão onde cada nó representa um estado do jogo e cada aresta representa uma jogada. Em cada nó folha da árvore (ou seja, em estados finais ou estados com profundidade máxima definida), uma função de avaliação determina o valor do estado. O algoritmo propaga os valores da folha para cima na árvore. No nível do jogador máximo, o valor de cada nó é o máximo dos valores de seus filhos. No nível do jogador mínimo, o valor é o mínimo dos valores de seus filhos. No final, o algoritmo escolhe a jogada correspondente ao valor máximo no nível da árvore onde o jogador atual está.

A poda alfa-beta é uma otimização do algoritmo Minimax que reduz o número de nós que precisam ser avaliados na árvore de decisão. Essa técnica é usada para podar (ou cortar) ramos da árvore que não afetam a decisão final, economizando tempo de computação.

A poda alfa-beta utiliza dois parâmetros, alfa e beta. Alfa é o melhor valor que o jogador maximizado pode garantir até o momento, e beta é o melhor valor que o jogador minimizado pode garantir até o momento. Durante a busca na árvore de decisão, a poda alfa-beta compara os valores alfa e beta e decide se deve continuar explorando um ramo. Se o valor mínimo possível de um ramo (beta) é menor ou igual ao valor máximo possível (alfa), o ramo é podado. Com a poda alfa-beta, a complexidade do algoritmo Minimax pode ser reduzida para O(bd/2), onde b é o fator de ramificação e d a profundidade da árvore. Isso é muito mais eficiente do que a complexidade original de O(bd) sem a poda.

A heurística de avaliação é uma função que atribui um valor a um estado do jogo para ajudar a determinar a qualidade do estado sem ter que simular o jogo até o final.

A heurística considera se há um vencedor atual. Se o jogador atual é o vencedor, o estado é avaliado positivamente. Se o adversário é o vencedor, o estado é avaliado negativamente.

A heurística avalia estados baseados em jogadas que estão próximas de levar à vitória ou de bloquear a vitória do adversário. Jogadas próximas de vitória são avaliadas positivamente, enquanto jogadas que precisam ser bloqueadas são avaliadas negativamente. A função heurística atribui uma pontuação para possíveis movimentos, ajudando a IA a priorizar jogadas que maximizam suas chances de ganhar ou minimizar as chances do adversário.

## 5.2 Complexidade

O algoritmo Minimax avalia todas as possíveis jogadas e suas consequências, criando uma árvore de decisão completa. A complexidade do Minimax depende da profundidade da árvore de decisão e do fator de ramificação (o número de possíveis jogadas em cada estado).

O tempo de execução do Minimax é exponencial em relação à profundidade da árvore de decisão. Para um jogo de tabuleiro como o jogo da velha, a complexidade é O(bd), onde b é o fator de ramificação (número de movimentos possíveis em cada estado) e d é a profundidade da árvore. No caso do jogo da velha, o fator de ramificação máximo é 9 (todos os espaços vazios) e a profundidade máxima é 9, resultando em uma complexidade máxima de O(99). O espaço de memória utilizado também é exponencial, pois o algoritmo precisa armazenar todos os nós da árvore de decisão. Isso pode se tornar um problema significativo para jogos com árvores de decisão grandes.

Já com a poda alfa-beta é uma otimização significativa para o algoritmo Minimax. Ela reduz o número de nós avaliados ao podar ramos que não precisam ser explorados.

Com a poda alfa-beta, a complexidade do Minimax é reduzida para O(bb/2). Isso ocorre porque a poda alfa-beta pode cortar metade dos ramos da árvore em média, reduzindo significativamente o número de estados que precisam ser avaliados. A complexidade espacial é similar à do Minimax, pois a poda alfa-beta ainda precisa armazenar todos os nós da árvore de decisão, mas a quantidade de nós é reduzida em média.

A heurística de avaliação é usada para avaliar a qualidade de um estado do jogo sem ter que simular todas as possíveis jogadas até o final.

A avaliação de um estado usando uma heurística é geralmente muito rápida, com complexidade O(1), pois envolve apenas a contagem de peças e a verificação de padrões no tabuleiro. A heurística não requer armazenamento adicional significativo além das variáveis utilizadas para contar e avaliar o estado atual, resultando em uma complexidade espacial mínima.

A combinação do Minimax com poda alfa-beta torna o algoritmo mais eficiente e viável para jogos como o jogo da velha, onde a árvore de decisão pode ser grande. A heurística de avaliação também ajuda a reduzir a complexidade ao evitar a necessidade de simular todas as possíveis jogadas até o final do jogo. No entanto, para jogos mais complexos com árvores de decisão maiores, a eficiência computacional pode se tornar um desafio, e técnicas adicionais ou simplificações podem ser necessárias para garantir uma jogabilidade fluida.

Analisando a recursividade no algoritmo Minimax com e sem poda alfa-beta, observamos que a poda alfa-beta reduz significativamente o número de nodos verificados. Sem a poda, o algoritmo explora um grande número de possíveis jogadas e ramificações, resultando em uma complexidade muito alta. Com a poda alfa-beta, a eficiência melhora consideravelmente, percorrendo cerca de 10% do que seria percorrido sem a otimização.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Teste de buscas | | | | |
| Espaço da jogada | X | Com poda | Sem poda | Dif. De nodos |
| sup.esq | 2788 | 59705 | 4,67% |
| sup.cent | 3187 | 63905 | 4,99% |
| sup.dir | 3648 | 59705 | 6,11% |
| cent.esq | 3848 | 63905 | 6,02% |
| cent.cent | 2459 | 55505 | 4,43% |
| cent.dir | 3957 | 63905 | 6,19% |
| inf.esq | 4173 | 59705 | 6,99% |
| inf.cent | 5631 | 63905 | 8,81% |
| sup.dir | 4511 | 59705 | 7,56% |

Tabela 1 – Testes de Busca

A análise das jogadas iniciais no jogo da velha revela que as posições nas bordas e nas quinas afetam a profundidade da análise. Jogar nas quinas geralmente oferece uma visão mais restrita do jogo, pois há menos caminhos de vitória disponíveis. Por outro lado, jogadas nas bordas e no centro podem resultar em uma análise mais extensa devido ao maior número de possíveis vitórias e espaços de jogo.

Quando o centro é removido da análise, as possibilidades de vitória são reduzidas, e a análise se torna mais rápida, pois menos nodos precisam ser explorados. Assim, a poda alfa-beta não só reduz a quantidade de verificações, mas também ajusta a profundidade da análise com base na posição inicial da jogada, otimizando o desempenho geral do algoritmo.

# 6. Instruções de Instalação

Para fazer as instalações dos códigos siga os seguintes passos:

1. Iníce colando o repositório para sua pasta de desejo:

```bash

git clone https://github.com/MaykollRocha/JogoDaVelhaMinMax.git

cd JogoDaVelhaMinMax

```

1. Entrar na pasta virtual ou instale os requerimentos:

Entrar na pasta virtual:

```bash

.\MaxMinVenv\Scripts\Activate.ps1

```

Ou instalar os imports necessários:

```bash

pip install -r requirements.txt

```

3. Execute o jogo:

```bash

python main.py

```

# 7. Interface Gráfica

Com uma interface de 2 variações meu código é construído sendo a Menu e a interface de jogo. No código a parte de interface é bem lapidada não deixando vazar erro já que sua função é bem simples.

A primeira interface consta de 5 botões mais um rádio:

Interface gráfica do usuário, Texto, Aplicativo, chat ou mensagem de texto

Descrição gerada automaticamente

Figura 2 – Menu

O menu conta com seis opções de interação:

1. **Humano vs Humano**: Modo de jogo para dois jogadores locais, onde cada um faz suas jogadas alternadamente até que ocorra uma vitória ou empate.
2. **Fácil**: Modo de jogo entre Humano e Computador, onde o computador faz jogadas aleatórias, sem utilizar o algoritmo Minimax.
3. **Médio**: O computador joga com uma profundidade de busca de 2 níveis na árvore de decisões, o que permite uma análise mais estratégica, mas limitada.
4. **Difícil**: A Inteligência Artificial (IA) joga com profundidade de busca infinita, explorando todas as possibilidades até encontrar a melhor solução.
5. **IA vs IA**: Duas IA jogam entre si no nível de dificuldade "Difícil". A primeira jogada é escolhida aleatoriamente.
6. **Modo Debug**: Um botão de rádio que pode ser ativado ou desativado. Quando ativo, a depuração é exibida no terminal, detalhando as jogadas e a análise da IA.

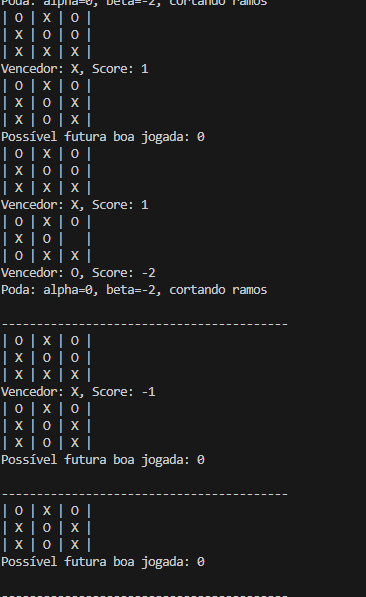
Desenho de palavras cruzadas

Descrição gerada automaticamente com confiança baixajogo da velha


Figura 3 – Jogadas e Tabuleiro

No tabuleiro, cada casa em branco é representada por um botão, somando um total de 9 botões. Cada jogador faz sua jogada clicando em uma casa, marcando o botão com a sua letra correspondente (X ou O). Após cada jogada, o tabuleiro é atualizado e o turno muda para o próximo jogador. Há também um botão "Voltar ao Menu", que encerra a partida atual e retorna ao menu principal.

Sobre o debug ele é terminal logo quando ativo ele primeiramente só mostra no modo IA vs IA ou Humano vs Computador, qualquer um dos 3 modos, nos outros modos não no é apresentado nada no temrinal.

  
figura 5 – Debug

O modo debug exibe, no terminal, informações detalhadas sobre as decisões da IA, disponíveis apenas nos modos "Humano vs Computador" (em qualquer dificuldade) e "IA vs IA". As informações incluem:

* O estado atual da árvore de decisões, exibindo os nós folha.
* O status do jogo nos nós folha, como vitória, derrota ou empate.
* O valor da heurística para cada possível jogada.
* As jogadas avaliadas e suas respectivas pontuações.

# 8. Considerações sobre Performance

* **Eficiência do código**: Potenciais limitações de performance ou otimizações realizadas.
* **Uso de memória e tempo de execução**: Análise do uso de recursos.

**1. Classe Player**

* **Eficiência**: A classe Player é bastante básica e não possui operações complexas, pois atua principalmente como uma classe base para jogadores. Não há preocupações significativas de desempenho nesta classe.

**2. Classe HumanPlayer**

* **Eficiência**: O método get\_move() solicita ao usuário para inserir um movimento e valida se o movimento é válido. A principal preocupação com a eficiência aqui é o tempo de execução associado ao uso do input() e à validação do movimento. No entanto, como o input() é uma operação de entrada/saída, o impacto no desempenho é mínimo e depende mais da interação do usuário do que da eficiência do código.

**3. Classe ComputerPlayer**

* **Eficiência**: A eficiência desta classe pode ser impactada principalmente pelo algoritmo Minimax, especialmente para o nível de dificuldade "hard".
  + **Nível Fácil**: Utiliza a função random.choice(), que é eficiente para a escolha aleatória de movimentos.
  + **Nível Médio**: Utiliza o algoritmo Minimax com uma profundidade fixa de 2, o que pode aumentar o tempo de execução em comparação com a escolha aleatória, mas ainda é gerenciável.
  + **Nível Difícil**: Utiliza o algoritmo Minimax com poda alfa-beta. O algoritmo Minimax tem uma complexidade exponencial no número de movimentos possíveis, mas a poda alfa-beta pode reduzir significativamente o número de nós explorados, melhorando a eficiência. No entanto, o desempenho pode ainda ser afetado pelo número de movimentos possíveis e pela profundidade de busca.
  + **Otimizações**:
    - **Poda Alfa-Beta**: A poda alfa-beta reduz o número de nós explorados na árvore de decisão, melhorando a eficiência do Minimax. Isso é especialmente útil em jogos com muitas possibilidades de movimento.
    - **Heurísticas**: O uso de heurísticas para avaliar o estado do jogo pode ajudar a melhorar a eficiência ao fornecer uma avaliação mais rápida dos estados do tabuleiro sem explorar todas as possibilidades.

**4. Classe Game**

* **Eficiência**: O código da classe Game é relativamente eficiente para um jogo da velha, com algumas operações que podem ser otimizadas.
  + **Método print\_board()**: Imprime o tabuleiro, o que é uma operação simples com um tempo de execução constante O(1).
  + **Método available\_moves()**: Percorre o tabuleiro para encontrar espaços vazios, com um tempo de execução O(n), onde n é o número de espaços no tabuleiro (n=9). Isso é eficiente dado o tamanho fixo do tabuleiro.
  + **Método empty\_squares()**: Verifica se há espaços vazios no tabuleiro, com um tempo de execução O(n).
  + **Método num\_empty\_squares()**: Conta o número de espaços vazios, com um tempo de execução O(n).
  + **Método make\_move()**: Faz uma jogada e verifica se há um vencedor, com um tempo de execução O(1) para a jogada e O(n) para verificar a vitória.
  + **Método winner()**: Verifica se a jogada resultou em vitória, o que pode ser otimizado para reduzir o número de verificações. No entanto, a verificação das linhas, colunas e diagonais é eficiente para o tamanho fixo do tabuleiro.

**5**. Classe TicTocToeApp

* **Método \_\_init\_\_**:
  + **Eficiência**: A inicialização da janela principal e o carregamento do ícone são operações básicas e eficientes. O uso de Image.open() e ImageTk.PhotoImage() para o ícone pode ser um pouco pesado se o ícone for muito grande, mas é geralmente eficiente para a maioria dos tamanhos de imagem.
  + **Otimizações**: A janela é configurada para não ser redimensionada, o que pode melhorar a eficiência da interface ao evitar recalculações de layout desnecessárias.

**Menu e Botões**

* **Método create\_menu**:
  + **Eficiência**: A criação do menu e dos botões é eficiente e usa layout em grade para organizar os widgets. A criação dos botões e labels é realizada com eficiência e com ações associadas que são diretas.
  + **Otimizações**: O uso de tk.Radiobutton para o modo debug e lambda para configurar as dificuldades evita a criação de funções separadas para cada nível de dificuldade.

**Modo de Jogo**

* **Métodos start\_human\_vs\_human, start\_human\_vs\_computer, start\_ai\_vs\_ai**:
  + **Eficiência**: Os métodos que iniciam diferentes modos de jogo são eficientes, destruindo o menu atual e configurando o jogo apropriado. A configuração do jogo é feita apenas uma vez por modo.
  + **Otimizações**: O método setup\_game() é utilizado para inicializar o tabuleiro e configurar o jogo, evitando duplicação de código.

**Movimentos e Atualizações**

* **Métodos on\_button\_click, computer\_move, ai\_move**:
  + **Eficiência**: A gestão dos cliques e movimentos é eficiente. O método on\_button\_click atualiza o estado do jogo e chama métodos para verificar o vencedor. O uso de self.root.after() para agendar o movimento da IA é eficiente, mas pode haver um pequeno atraso perceptível de 500 ms entre as jogadas da IA.
  + **Otimizações**: O método self.root.after() é usado para movimentar a IA de forma assíncrona, o que melhora a responsividade da interface.

**Verificação e Status**

* **Métodos check\_winner, disable\_all\_buttons, update\_status**:
  + **Eficiência**: Verificar o vencedor e atualizar a interface é feito de forma eficiente com a verificação das condições de vitória e a atualização do status da interface gráfica.
  + **Otimizações**: O método disable\_all\_buttons desabilita todos os botões para evitar mais jogadas, e o método update\_status atualiza o status da interface com a mensagem e a cor apropriadas.

**Criação do Tabuleiro**

* **Método create\_board**:
  + **Eficiência**: A criação do tabuleiro com 9 botões é eficiente e os botões são configurados com comandos associados. O uso de tk.Button e grid() é eficiente para um tabuleiro de jogo da velha.
  + **Otimizações**: O método create\_board é modular e cria todos os componentes do tabuleiro de uma só vez. Isso ajuda a manter o código limpo e eficiente.

**Retorno ao Menu**

* **Método back\_to\_menu**:
  + **Eficiência**: Destruir o tabuleiro e os componentes associados é uma operação eficiente para limpar o estado atual do jogo e retornar ao menu principal. O código cuida para liberar as referências aos objetos para evitar vazamentos de memória.

**Uso de Memória e Tempo de Execução**

* **Classe Player**: Uso de memória é mínimo, apenas armazena uma letra. Tempo de execução é irrelevante para a operação de inicialização.
* **Classe HumanPlayer**: O uso de memória é baixo, armazenando apenas uma letra. O tempo de execução para get\_move() é dependente da interação com o usuário.
* **Classe ComputerPlayer**:
  + **Uso de Memória**: O uso de memória é mais significativo devido ao armazenamento do estado do tabuleiro e à complexidade do algoritmo Minimax.
  + **Tempo de Execução**: A eficiência pode variar bastante dependendo do nível de dificuldade. Para o nível "hard", a profundidade do Minimax e o número de possibilidades a serem exploradas podem aumentar o tempo de execução significativamente.
* **Classe Game**:
  + **Uso de Memória**: O uso de memória é proporcional ao tamanho do tabuleiro (9 espaços) e variáveis associadas.
  + **Tempo de Execução**: O tempo de execução é eficiente para o tamanho fixo do tabuleiro, com operações principais tendo complexidade O(n) ou O(1). No entanto, o desempenho pode ser afetado quando integrado com algoritmos complexos como o Minimax.
* **Classe TicTocToeApp**:
  + **Uso de Memória**:O uso de memória é eficiente para a interface gráfica de um jogo da velha. A memória é utilizada para armazenar o estado dos botões, o status do jogo, e as instâncias dos jogadores. A memória não deve ser um problema significativo devido ao tamanho relativamente pequeno da interface e ao número limitado de botões e widgets.
  + **Tempo de Execução**:O tempo de execução é geralmente rápido para as operações realizadas. A criação do menu e do tabuleiro é feita de forma eficiente. As atualizações do status e a resposta aos cliques são realizadas de maneira quase instantânea. O tempo de execução pode ser afetado pelas jogadas da IA devido à implementação do algoritmo Minimax e pela espera de 500 ms entre os movimentos da IA.

# 9. Referências

**1. PYTHON SOFTWARE FOUNDATION. Python Documentation. 2024. Disponível em: <https://docs.python.org/3/>. Acesso em: 17 set. 2024.0**

2. PYTHON SOFTWARE FOUNDATION. Tkinter — Python interface to Tcl/Tk. 2024. Disponível em: <https://docs.python.org/3/library/tkinter.html>. Acesso em: 17 set. 2024

3. PYTHON SOFTWARE FOUNDATION. math — Mathematical functions. 2024. Disponível em: <https://docs.python.org/3/library/math.html>. Acesso em: 17 set. 2024.

4. PYTHON PILLOW. Pillow (PIL Fork) Documentation. 2024. Disponível em: <https://python-pillow.org>. Acesso em: 17 set. 2024.