

Autor(a): 026

Jogo da Velha

Documentação referente ao jogo da velha feita como parte do processo seletivo do projeto de extensão da Crossbots 2025.2.

Sumário

1	Introdução	2
2	Compilação e Execução	3
2.1	Pré-requisitos	3
2.2	Comandos do Makefile	3
2.3	Compilação Rápida	4
2.4	Descrição dos Diretórios	4
2.5	Interface do Jogo	4
2.6	Fluxo de Execução	4
2.7	Arquivos de Cabeçalho	5
3	Algoritmo Minimax	5
4	Diagrama de Classes	6
4.1	Descrição das Classes	7
5	Resultados e Testes	7
6	Conclusão	8
6.1	Trabalhos Futuros	8

1 Introdução

Este documento apresenta o desenvolvimento da solução para o desafio "Jogo da Velha", proposto como parte da primeira etapa do processo seletivo da equipe Crossbots 2025.2. O projeto consiste em uma aplicação de software desenvolvida na linguagem C++ e orientação a objetos (OOP), que implementa o clássico Jogo da Velha, no qual o Jogador 2 é controlado por um *bot*.

Primeiramente, o Jogo da Velha é disputado por dois jogadores em uma matriz 3x3. O primeiro usa "X" e o segundo, "O". Vence quem alinhar três símbolos iguais na horizontal, vertical ou diagonal.

Neste problema, o Jogador 2 será um *bot* que foi implementado utilizando o algoritmo *Minimax*. O programa exibe o tabuleiro vazio, recebe a jogada do Jogador 1 e, em seguida, o *bot* faz automaticamente a melhor jogada para vencer ou bloquear o adversário. Após cada lance, o tabuleiro atualizado é mostrado.

```
===== JOGO DA VELHA =====
Use coordenadas de 1 a 3 para linha e coluna

=== NOVA PARTIDA INICIADA ===
Jogadores:
- Jogador 1 (X)
- Bot (O)

  1  2  3
  ---
1 |  |  |  |
  ---
2 |  |  |  |
  ---
3 |  |  |  |
  ---

Jogador 1, eh sua vez! (Simbolo: X)
Digite a linha (1-3): |
```

Figura 1: Início de uma partida.

```
Jogador 1, eh sua vez! (Simbolo: X)
Digite a linha (1-3): 1
Digite a coluna (1-3): 1
Jogada realizada com sucesso na posicao (1, 1)!

  1  2  3
  ---
1 | X |  |  |
  ---
2 |  |  |  |
  ---
3 |  |  |  |
  ---

Bot esta pensando...
Bot jogou na posicao (2, 2)

  1  2  3
  ---
1 | X |  |  |
  ---
2 |  | O |  |
  ---
3 |  |  |  |
  ---

Jogador 1, eh sua vez! (Simbolo: X)
Digite a linha (1-3): |
```

Figura 2: Jogo em andamento .

O jogo termina quando alguém vence ou todas as casas são preenchidas, resultando em um empate. No fim, é exibida uma mensagem indicando o vencedor ou o empate.

```
Bot jogou na posicao (3, 2)

  1  2  3
-----
1 | X | O | X |
-----
2 |   | O |   |
-----
3 |   | O | X |
-----

=== RESULTADO FINAL ===
VITORIA DO Bot (O)!
Obrigado por jogar!
```

Figura 3: Finalização de um partida com a vitória do *bot*.

Utilizou-se o paradigma da Programação Orientada a Objetos (OOP) com o objetivo de promover maior coesão e menor acoplamento entre as classes, além de facilitar futuras melhorias e atualizações no sistema.

2 Compilação e Execução

2.1 Pré-requisitos

- Compilador C++ compatível com C++11 (g++, clang++, MSVC)
- Make (GNU Make ou compatível)
- Sistema operacional: Windows, Linux ou macOS

2.2 Comandos do Makefile

```
make          # Compilação básica
make build-run # Compilação e execução
make debug    # Versão com símbolos de depuração
make release  # Versão otimizada
make run      # Executar após compilação
make clean    # Limpar arquivos compilados
make rebuild  # Recompilar do zero
make check    # Verificar sintaxe
make count    # Contar linhas de código
make info     # Informações do projeto
make help     # Ajuda
```

2.3 Compilação Rápida

`make`

`make build-run`

`make clean`

2.4 Descrição dos Diretórios

- `src/`: Implementação (.cpp)
- `include/`: Cabeçalhos (.h)
- `doc/`: Documentação
- `Diagramas/`: Diagramas de classe
- `imagens/`: Imagens inseridas para a documentação

2.5 Interface do Jogo

- Tela de boas-vindas com regras
- Tabuleiro colorido:
 - X: BOT
 - O: Jogador
- Entrada: linha (1-3) e coluna (1-3)
- Resultado: vencedor ou empate

2.6 Fluxo de Execução

Esta seção descreve a sequência lógica de execução do programa principal. Logo, o arquivo `main.cpp` inicializa o jogo, cria as entidades e inicia a partida:

1. Exibe instruções iniciais ao usuário.
2. Cria uma instância da classe `Game`.
3. Cria um jogador humano (`Player`) e um bot (`Bot`).
4. Adiciona ambos à lista de entidades do jogo.
5. Chama o método `playGame()`, que executa o loop principal até o fim da partida.

2.7 Arquivos de Cabeçalho

- `Entity.h`: Define a classe base `Entity`, que representa qualquer jogador (humano ou bot). Contém atributos de nome, símbolo, posição e pontuação.
- `Player.h`: Declara a classe `Player`, derivada de `Entity`, responsável por receber as jogadas do usuário.
- `Bot.h`: Declara a classe `Bot`, também derivada de `Entity`, implementando a lógica de decisão automática via algoritmo Minimax.
- `Board.h`: Define a classe `Board`, que armazena o tabuleiro, verifica vitórias, empates e imprime o estado do jogo.
- `Game.h`: Implementa a classe `Game`, que gerencia o fluxo completo da partida, alternando entre jogadores e verificando o término.
- `Minimax.h`: Contém uma implementação independente e genérica do algoritmo Minimax, incluindo a estrutura `Move`.
- `Ui.h`: Define macros para estilização de texto (cores e negrito) no terminal.

3 Algoritmo Minimax

O Minimax se trata de um algoritmo que maximiza a chance do *bot* ganhar e minimiza as chances de seu oponente vencer, por isso o nome “mini-max”. Para tanto, é utilizada uma busca em profundidade (*Depth-First Search* - *DFS*) para explorar recursivamente todas as possibilidades de jogadas futuras a partir do estado atual do jogo.

A cada nó terminal da árvore de busca (um estado de vitória, derrota ou empate), uma pontuação é atribuída para avaliar o resultado:

- +10: para uma vitória do bot (o jogador maximizador).
- -10: para uma vitória do jogador humano (o jogador minimizador).
- 0: para um empate.

Esses valores são arbitrários; a Figura 4 ilustra o mesmo conceito utilizando 1, -1 e 0.

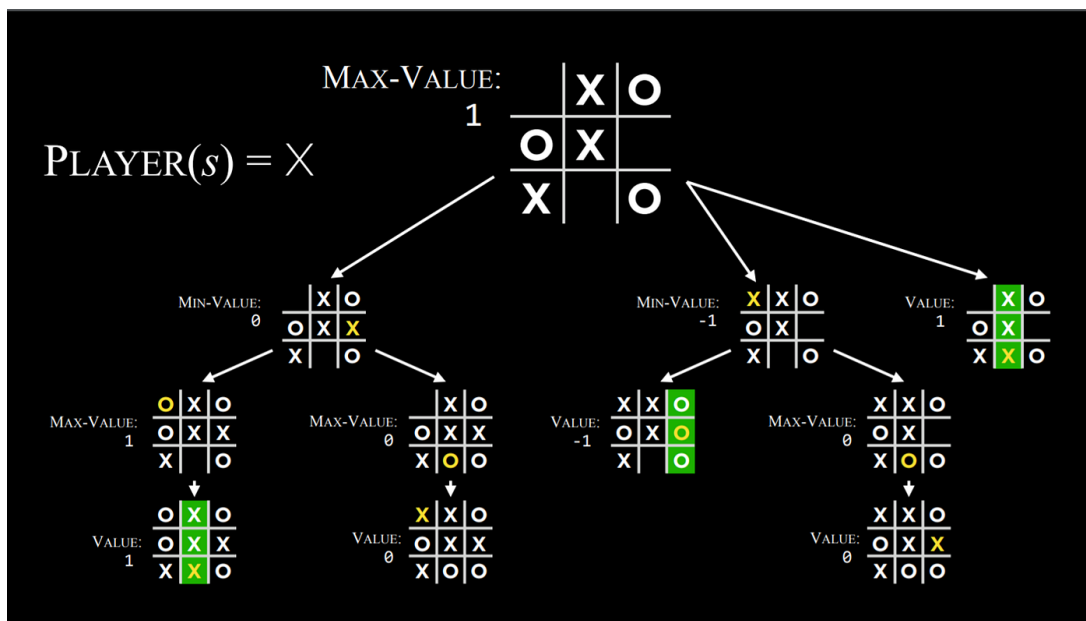


Figura 4: Funcionamento do algoritmo Minimax. Fonte: [1]

Esses valores são então propagados de volta pela árvore. Nos turnos do bot (níveis MAX), ele escolhe o movimento que leva ao maior valor. Nos turnos do humano (níveis MIN), ele assume que o oponente fará o mesmo, escolhendo o movimento que leva ao menor valor. A Figura 4 demonstra visualmente este processo de decisão.

A implementação desta lógica está contida nos métodos privados `minimax()` e `evaluateBoard()` da classe Bot, garantindo que a IA sempre selecione o movimento que leva ao melhor resultado possível.

Todavia, o algoritmo Minimax é ineficiente quando se trata de tabuleiros grandes devido à sua complexidade de tempo exponencial, $O(b^m)$, onde b é o fator de ramificação e m é a profundidade máxima da árvore. No caso do Jogo da Velha, por se tratar de uma matriz pequena 3×3 , isso não se tornou um problema prático, mas é uma limitação a ser considerada em projetos de maior escala.

4 Diagrama de Classes

Foi realizado o seguinte diagrama de classes utilizando o *software Mermaid* [2]:

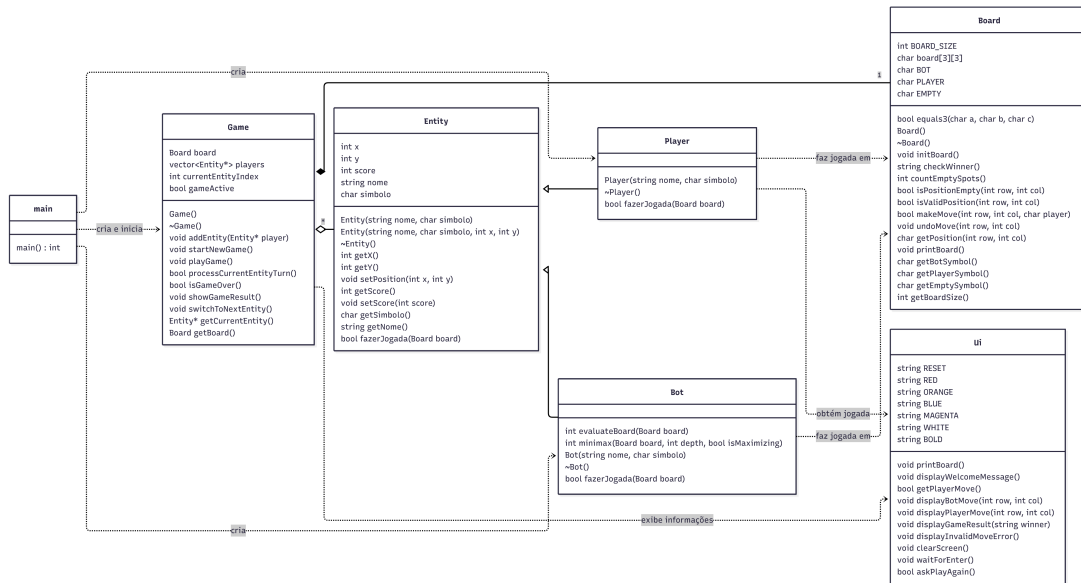


Figura 5: Diagrama de classes.

4.1 Descrição das Classes

- **Entity**: Classe base abstrata que representa uma entidade genérica (jogador ou bot), armazenando nome, símbolo, posição e pontuação.
- **Player**: Herda de Entity. Representa o jogador humano e gerencia a entrada via teclado.
- **Bot**: Herda de Entity. Implementa a jogada automática utilizando o algoritmo Minimax.
- **Board**: Responsável pelo tabuleiro do jogo, controle das posições, verificação de vitórias e exibição.
- **Game**: Controla o fluxo da partida, alternando os turnos entre as entidades e verificando o término do jogo.

5 Resultados e Testes

Foram realizados testes de jogabilidade para validar o comportamento do algoritmo Minimax e a alternância correta de turnos.

- O *bot* sempre evita derrotas imediatas.
- Em tabuleiros simétricos, o *bot* escolhe jogadas centrais.
- O sistema detecta corretamente vitórias horizontais, verticais e diagonais.
- Empates são tratados quando todas as posições estão preenchidas.

6 Conclusão

O projeto atendeu ao objetivo proposto, implementando um Jogo da Velha funcional com inteligência artificial baseada em Minimax. O uso de orientação a objetos facilitou a modularização e manutenção do código.

6.1 Trabalhos Futuros

- Adicionar níveis de dificuldade ajustando a profundidade do Minimax.
- Implementar uma interface gráfica (por exemplo, SFML, Qt).
- Armazenar pontuações e histórico de partidas.
- Adicionar melhorias ao algoritmo Minimax que diminuam custo de tempo de execução.

Referências

- [1] Muhammad Hamza Hassaan. *Mini-Max Algorithm in Artificial Intelligence*. Acesso em: 10 out. 2025. 2025. url: <https://medium.com/@mhhassaan.1/mini-max-algorithm-in-artificial-intelligence-e0a0da694b3b>.
- [2] Mermaid Chart Team. *Mermaid Chart*. Acesso em: 10 out. 2025. 2025. url: <https://www.mermaidchart.com>.