Báo cáo đồ án game caro nxn

# Mục lục

[1 Mục lục 1](#_Toc185209921)

[2 Giới thiệu tổng quan 2](#_Toc185209922)

[3 Nguyên nhân thực hiện Project này? 5](#_Toc185209923)

[4 Các tính năng chính của game Caro nxn 6](#_Toc185209924)

[5 Các cấu trúc dữ liệu, giải thuật và thư viện được dùng. 8](#_Toc185209925)

[5.1 Menu game 8](#_Toc185209926)

[5.1.1 Cấu trúc dữ liệu 8](#_Toc185209927)

[5.1.2 Giải thuật 8](#_Toc185209928)

[5.1.3 Thư viện 9](#_Toc185209929)

[5.2 Game 9](#_Toc185209930)

[5.2.1 Cấu trúc dữ liệu 9](#_Toc185209931)

[5.2.2 Giải thuật 10](#_Toc185209932)

[5.2.3 Thư viện 11](#_Toc185209933)

[5.3 GameBot 12](#_Toc185209934)

[6 Chi tiết cấu trúc dữ liệu, giải thuật, thư viện được sử dụng trong GameBot 12](#_Toc185209935)

[6.1 CheckWin 12](#_Toc185209936)

[6.1.1 Cấu trúc dữ liệu 12](#_Toc185209937)

[6.1.2 Giải thuật 12](#_Toc185209938)

[6.1.3 Thư viện 13](#_Toc185209939)

[6.1.4 Mả giả(Pseudo code) 13](#_Toc185209940)

[6.1.5 Độ phức tạp 15](#_Toc185209941)

[6.2 CheckDraw 15](#_Toc185209942)

[6.2.1 Cấu trúc dữ liệu 15](#_Toc185209943)

[6.2.2 Giải thuật 15](#_Toc185209944)

[6.2.3 Thư viện 16](#_Toc185209945)

[6.2.4 Mả giả(Pseudo code) 16](#_Toc185209946)

[6.2.5 Độ phức tạp 16](#_Toc185209947)

[6.3 undoMove 16](#_Toc185209948)

[6.3.1 Cấu trúc dữ liệu 16](#_Toc185209949)

[6.3.2 Giải thuật 17](#_Toc185209950)

[6.3.3 Thư viện được dùng 18](#_Toc185209951)

[6.3.4 Pseudo code (Mã giả) 19](#_Toc185209952)

[6.3.5 Độ phức tạp 20](#_Toc185209953)

[6.4 handleMouseClick 20](#_Toc185209954)

[6.4.1 Cấu trúc dữ liệu 20](#_Toc185209955)

[6.4.2 Giải thuật 21](#_Toc185209956)

[6.4.3 Thư viện 22](#_Toc185209957)

[6.4.4 Mã giả (Pseudo Code) 23](#_Toc185209958)

[6.4.5 Độ phức tạp 24](#_Toc185209959)

[6.5 botMove 25](#_Toc185209960)

[6.5.1 Cấu trúc dữ liệu 25](#_Toc185209961)

[6.5.2 Giải thuật 25](#_Toc185209962)

[6.5.3 Thư viện 27](#_Toc185209963)

[6.5.4 Mã giả (Pseudo Code) 27](#_Toc185209964)

[6.5.5 Độ phức tạp 29](#_Toc185209965)

[6.6 easyBotMove 29](#_Toc185209966)

[6.6.1 Cấu trúc dữ liệu 29](#_Toc185209967)

[6.6.2 Giải thuật 30](#_Toc185209968)

[6.6.3 Thư viện 30](#_Toc185209969)

[6.6.4 Mả giả(Pseudo code) 31](#_Toc185209970)

[6.6.5 Độ phức tạp 31](#_Toc185209971)

[6.6.6 Các hàm hỗ trợ easyBotMove 32](#_Toc185209972)

[6.7 mediumBotMove 41](#_Toc185209973)

[6.7.1 Cấu trúc dữ liệu 41](#_Toc185209974)

[6.7.2 Giải thuật 41](#_Toc185209975)

[6.7.3 Thư viện 41](#_Toc185209976)

[6.7.4 Mả giả(Pseudo code) 42](#_Toc185209977)

[6.7.5 Độ phức tạp 42](#_Toc185209978)

[6.7.6 Các hàm phụ trợ mediumBotMove 43](#_Toc185209979)

[6.8 hardBotMove 70](#_Toc185209980)

[6.8.1 Cấu trúc dữ liệu 70](#_Toc185209981)

[6.8.2 Giải thuật 70](#_Toc185209982)

[6.8.3 Thư viện 71](#_Toc185209983)

[6.8.4 Mả giả(Pseudo code) 71](#_Toc185209984)

[6.8.5 Độ phức tạp 72](#_Toc185209985)

[6.8.6 Các hàm hỗ trợ cho HardBotMove 73](#_Toc185209986)

[7 Cách chơi 102](#_Toc185209987)

[8 Thư viện hỗ trợ, cách cài thư viện thêm để thực thi đồ án 102](#_Toc185209988)

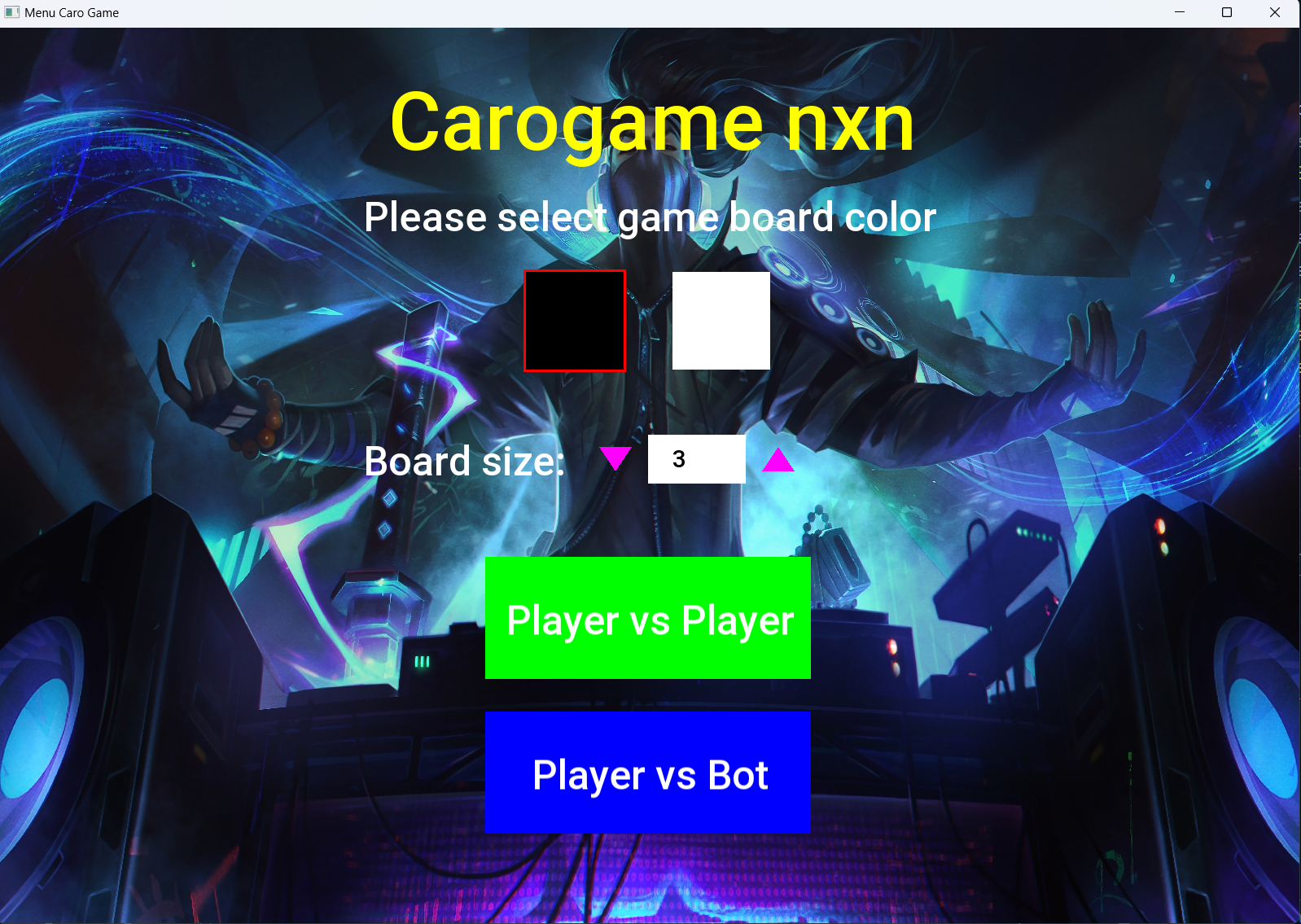
[9 Nguồn kham khảo chính 104](#_Toc185209989)

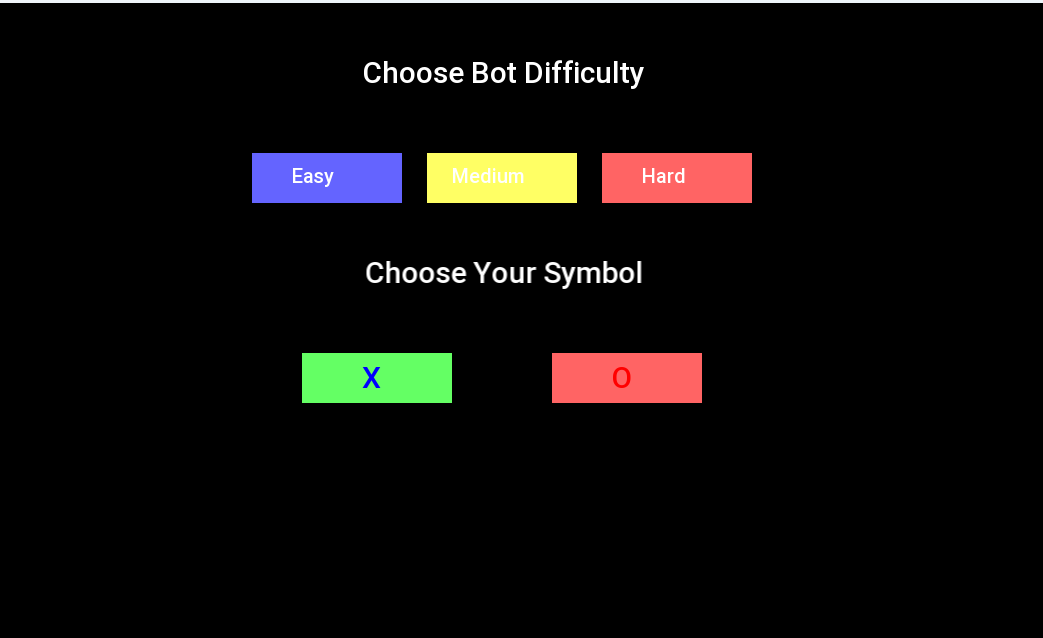
# Giới thiệu tổng quan

Game caro là một trò chơi chiến thuật cổ điển. Trò chơi thường được chơi trên bàn cờ kẻ ô vuông, với mục tiêu là xếp đủ một số lượng ký hiệu liên tiếp theo hàng ngang, dọc, hoặc chéo để dành chiến thắng.

Trong phiên bản này, Game Caro nxn được phát triển bằng ngôn ngữ C++ và sử dụng với thư viện SFML để xây dựng giao diện đồ họa. Cho phép người chơi tùy chỉnh:

* Màu bàn cờ: Chọn giữa bàn cờ đen hoặc trắng.
* Kích cỡ bàn cờ: Tùy chỉnh từ 3x3, 5x5 đến 30x30.
* Chế độ chơi:
  + Người với người: Hai người chơi sẽ lần lượt thực hiện các nước đi.
  + Người với máy: Người chơi lựa chọn mức độ khó của bot (Dễ, Vừa, Khó), chọn ký hiệu X hoặc O, và người chơi sẽ đi trước trong lượt đầu tiên.





# Nguyên nhân thực hiện Project này?

* Mục tiêu học thuật: Phát triển dự án nhằm rèn luyện kỹ năng lập trình, sử dụng cấu trúc dữ liệu và giải thuật trong phát triển game.
* Ứng dụng thực tế: Xây dựng một trò chơi gắn liền với tuổi thơ, một trò chơi chiến thuật đơn giản nhưng thú vị. Cung cấp trải nghiệm tương tác và giải trí, đồng thời thử thách tư duy logic của người chơi.
* Thực hành thuật toán AI: Tích hợp trí tuệ nhân tạo(AI) mức độ dễ với các thuật toán Minimax kết hợp với Alpha-Beta Pruning để xây dựng một đối thủ ảo thông minh.

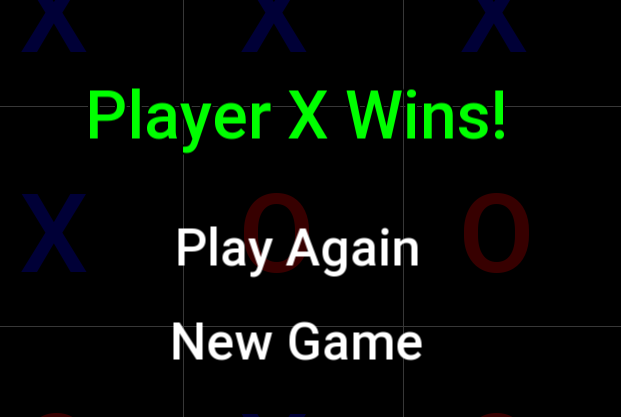
# Các tính năng chính của game Caro nxn

* Menu game:
  + Chọn màu bàn cờ.
  + Chọn kích cỡ bàn cờ.
  + Chọn chế độ chơi:
    - PvP(2 người chơi) -> Game
    - PvB(chơi vơi bot) ->GameBot
  + Truyền dữ liệu(màu bàn cờ, kích cỡ bàn cờ) cho Game hoặc GameBot.

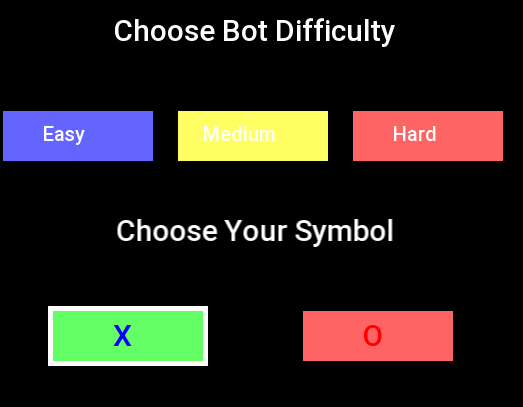




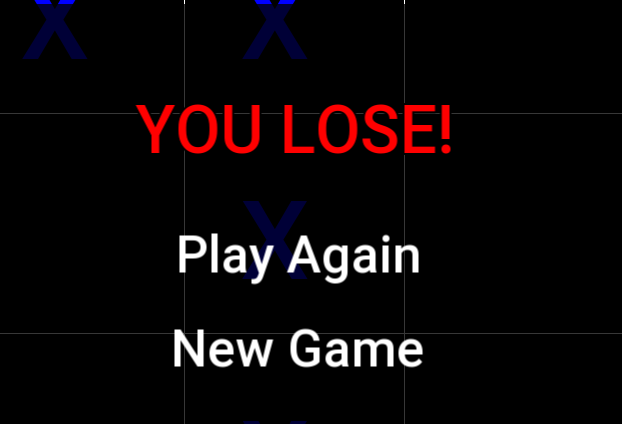
* Game:
  + Hiển thị bàn cờ với màu bàn cờ, kích cỡ màu cờ từ Menu game.
  + Hỗ trợ chế độ chơi 2 người.
  + Hiển thị thông báo chiến thắng và cung cấp tùy chọn chơi lại hoặc bắt đầu ván mới(->Menu game).



* GameBot:
  + Hiển thị chọn chế độ khó của bot, chọn nước đi.



* + Hỗ trợ chế độ chơi vơi Bot tùy độ khó, và tính năng Undo.
  + Hiện thị thông báo chiến thắng tùy chọn chơi lại hoặc bắt đầu ván mới(->Menu game).



# Các cấu trúc dữ liệu, giải thuật và thư viện được dùng.

## Menu game

### Cấu trúc dữ liệu

* bool blackSelected: Xác định màu bàn cờ được chọn (đen hoặc trắng).
* int boardSize: Kích thước bàn cờ (n x n).
* sf::RenderWindow window: Cửa sổ hiển thị giao diện.
* sf::Font font: Phông chữ sử dụng trong giao diện.
* sf::Texture backgroundTexture: Kết cấu hình nền.
* sf::Sprite backgroundSprite: Hình nền được hiển thị.
* sf::RectangleShape BlackSquare, whiteSquare: Các ô vuông chọn màu bàn cờ.
* sf::RectangleShape sizeBox: Ô vuông hiển thị kích thước bàn cờ.
* sf::ConvexShape downTriangle, upTriangle: Các nút tam giác tăng/giảm kích thước bàn cờ.
* sf::RectangleShape PvPButton, PvBButton: Nút chọn chế độ chơi (PvP, PvB).
* sf::Text: Chứa các dòng chữ hiển thị (tiêu đề, hướng dẫn, nút bấm).

### Giải thuật

* Căn chỉnh giao diện: Tính toán vị trí trung tâm cho các thành phần (button, text).
* Tải tài nguyên: Kiểm tra và tải phông chữ, hình nền (có xử lý lỗi khi thất bại).
* Căn chỉnh kích thước hình nền: Tỷ lệ giữa kích thước cửa sổ và hình nền.
* Xử lý sự kiện (processEvents):
  + Kiểm tra sự kiện đóng cửa sổ hoặc nhấn chuột.
  + Gọi hàm handleMouseClick() để xử lý hành động nhấn chuột.
* Cập nhật trạng thái giao diện (update):
  + Đặt viền đỏ cho ô màu được chọn (BlackSquare hoặc whiteSquare).
  + Loại bỏ viền của ô không được chọn.
* Vẽ giao diện (render):
  + Vẽ các thành phần giao diện như background, text, các nút, hình vuông, và số kích thước bàn cờ.
  + Sử dụng window.display() để cập nhật giao diện hiển thị.
* Xử lý nhấn chuột (handleMouseClick):
  + Kiểm tra vị trí nhấn chuột để:
  + Chọn màu bàn cờ (đen hoặc trắng).
  + Tăng/giảm kích thước bàn cờ (giới hạn từ 3 đến 100).
  + Kích hoạt nút "Player vs Player" hoặc "Player vs Bot" để bắt đầu trò chơi với chế độ tương ứng.
  + Đóng cửa sổ khi một chế độ chơi được chọn.

### Thư viện

* SFML/Graphics.hpp: Vẽ giao diện đồ họa (ô vuông, nút bấm, hình nền, text).
* SFML/Window.hpp: Tạo cửa sổ và xử lý sự kiện.
* SFML/System.hpp: Hỗ trợ các đối tượng liên quan đến SFML.
* stdexcept: Xử lý lỗi, đặc biệt khi tải tài nguyên không thành công.
* iostream: Hiển thị thông báo lỗi khi không tải được ảnh nền.
* string: Quản lý chuỗi ký tự (ví dụ: std::to\_string trong render() để hiển thị kích thước bàn cờ).

## Game

### Cấu trúc dữ liệu

* Bàn cờ (std::vector<std::vector<char>> board):
  + Mảng 2D chứa các giá trị 'X', 'O', hoặc ' ' để biểu diễn trạng thái các ô trên bàn cờ.
* Các dòng lưới (std::vector<sf::RectangleShape> gridLines):
  + Mảng lưu trữ các đối tượng sf::RectangleShape dùng để vẽ các đường lưới chia bàn cờ.
* Ký hiệu người chơi (std::vector<sf::Text> markers):
  + Lưu trữ các đối tượng văn bản sf::Text để hiển thị ký hiệu 'X' và 'O' trên bàn cờ.
* Các biến trạng thái:
  + char currentPlayer: Lưu trữ ký hiệu của người chơi hiện tại ( 'X' hoặc 'O').
  + bool gameOver: Cờ hiệu cho biết trò chơi có kết thúc hay không.
  + bool gameDraw: Cờ hiệu xác định nếu trò chơi kết thúc hòa.
  + sf::VertexArray winLine: Lưu trữ đường thắng, nếu có.
* Cài đặt font (sf::Font font):
  + Lưu trữ đối tượng font để sử dụng cho các văn bản hiển thị trên màn hình (như tên người chơi, thông báo chiến thắng).
* Vị trí các nút (Play Again, New Game):
  + sf::Text playAgainText, newGameText, winnerText: Các đối tượng văn bản cho các nút và thông báo chiến thắng.

### Giải thuật

* Khởi tạo và thiết lập game:
  + Game::Game(int boardSize, sf::Color boardColor, MenuCarogame &menu): Khởi tạo trò chơi, cài đặt kích thước bàn cờ, màu sắc bàn cờ, và khởi tạo cửa sổ hiển thị.
* Quản lý sự kiện:
  + processEvents(): Xử lý các sự kiện như đóng cửa sổ, nhấn chuột vào các nút "Play Again" và "New Game".
  + handleMouseClick(const sf::Vector2i &mousePosition): Xử lý nhấn chuột vào các ô trên bàn cờ, kiểm tra nếu người chơi có thể đánh vào ô đó. Nếu không phải là ô đã có dấu hoặc trò chơi đã kết thúc, nó sẽ thay đổi lượt chơi.
* Vẽ bàn cờ và các đối tượng:
  + render(): Xử lý việc vẽ lại bàn cờ, các ô trên bàn, các dấu X và O, và đường thắng nếu có. Cập nhật mỗi lần vẽ lại khi có thay đổi.
* Kiểm tra kết quả:
  + checkDraw(): Kiểm tra xem bàn cờ có đầy đủ các dấu X và O chưa, nếu có thì kết thúc trò chơi với kết quả hòa.
  + checkWin(int row, int col, char player): Kiểm tra nếu có người chơi nào thắng sau một nước đi. Kiểm tra theo 4 hướng: ngang, dọc, chéo trái phải và chéo phải trái.
* Màn hình kết quả:
  + showWinScreen(const std::string &winner): Hiển thị màn hình chiến thắng hoặc hòa, với các nút "Play Again" và "New Game".
* Reset game:
  + resetGame(): Đặt lại trạng thái bàn cờ, xóa các ký hiệu đã vẽ và khởi tạo lại trò chơi.
* Quay lại menu:
  + openMenu(): Đóng cửa sổ trò chơi hiện tại và mở lại menu chính.

### Thư viện

* sf::RectangleShape: Dùng để vẽ các ô trên bàn cờ và các đường lưới.
* sf::Text: Dùng để vẽ các ký tự (X hoặc O) và các thông báo (kết quả trận đấu).
* sf::Color: Dùng để xác định màu sắc cho các đối tượng (như màu ô, màu chữ).
* sf::Event: Dùng để theo dõi các sự kiện từ người dùng (nhấn chuột, đóng cửa sổ).
* sf::Font: Dùng để tải font chữ từ file.
* sf::RenderWindow: Dùng để tạo cửa sổ trò chơi và quản lý việc vẽ các đối tượng trong cửa sổ.
* sf::VertexArray: Dùng để vẽ các đường thẳng (đường thắng).
* <stdexcept>: Dùng để xử lý lỗi khi không tải được font.
* <iostream>: Dùng để in thông báo lỗi.

## GameBot

Cấu trúc dữ liệu, giải thuật và thư viện trong GameBot có một số phần giống với Game, nên sẽ không được nêu lại.

# Chi tiết cấu trúc dữ liệu, giải thuật, thư viện được sử dụng trong GameBot

## CheckWin

### Cấu trúc dữ liệu

* boardSize: Một biến số nguyên xác định kích thước bàn cờ (nxn).
* board: Một mảng 2 chiều kích thước boardSize x boardSize, lưu trạng thái các ô trên bàn cờ, có thể chứa các ký tự như 'X', 'O', hoặc dấu trống (ví dụ: ' ').
* row, col: Các chỉ số vị trí hiện tại trên bàn cờ mà người chơi hoặc bot đang kiểm tra.

### Giải thuật

* checkDirection: Hàm phụ trách kiểm tra một hướng di chuyển (theo hàng, cột hoặc đường chéo). Hàm sẽ đếm số lượng ký tự liên tiếp theo hướng đó từ vị trí (row, col). Nếu có đủ số lượng ký tự liên tiếp (>= winCondition, có thể là 3 hoặc 5 tùy kích thước bàn cờ), hàm trả về true.
  + Duyệt hai hướng: một theo chiều dương (tăng chỉ số) và một theo chiều âm (giảm chỉ số).
  + Mỗi lần di chuyển, hàm kiểm tra nếu vị trí trong bảng còn hợp lệ (không ra ngoài phạm vi) và ký tự tại vị trí đó có giống với ký tự của người chơi (player) không.
* winCondition: Biến này xác định điều kiện thắng:
  + Nếu boardSize < 4, điều kiện thắng là 3 ký tự liên tiếp.
  + Nếu boardSize > 4, điều kiện thắng là 5 ký tự liên tiếp.
* Điều kiện thắng: Kiểm tra 4 hướng: hàng (1, 0), cột (0, 1), đường chéo (1, 1) và đường chéo ngược (1, -1). Nếu tìm thấy 5 ký tự liên tiếp (hoặc 3 tùy thuộc vào kích thước bàn cờ), hàm trả về true.
* Trạng thái game kết thúc: Nếu không phải kiểm tra của bot (biến botCheck là false), thì biến gameOver được đánh dấu là true.

### Thư viện

### Mả giả(Pseudo code)

Function checkWin(row, col, player, botCheck):

winCondition = 3 if boardSize <= 4 else 5

Function checkDirection(dr, dc):

count = 1

// Check in one direction (dr, dc)

For step from 1 to winCondition - 1:

nr = row + step \* dr

nc = col + step \* dc

If nr >= 0 and nr < boardSize and nc >= 0 and nc < boardSize:

If board[nr][nc] == player:

count += 1

Else:

Break

Else:

Break

// Check in the opposite direction (-dr, -dc)

For step from 1 to winCondition - 1:

nr = row - step \* dr

nc = col - step \* dc

If nr >= 0 and nr < boardSize and nc >= 0 and nc < boardSize:

If board[nr][nc] == player:

count += 1

Else:

Break

Else:

Break

Return count >= winCondition

If checkDirection(1, 0) or checkDirection(0, 1) or checkDirection(1, 1) or checkDirection(1, -1):

If not botCheck:

gameOver = True

Return True

Return False

### Độ phức tạp

* Không gian
  + O(1): Hàm không sử dụng bộ nhớ phụ thuộc vào kích thước bảng, chỉ sử dụng một vài biến đơn giản, vì vậy độ phức tạp không gian là hằng số O(1).
* Thời gian
  + O(n): Với mỗi hướng (4 hướng), hàm kiểm tra tối đa winCondition bước đi (tối đa 5 bước nếu boardSize > 4), nên độ phức tạp tổng cộng cho mỗi hướng là O(n) với n là kích thước bàn cờ.

## CheckDraw

### Cấu trúc dữ liệu

* boardSize: Một biến số nguyên xác định kích thước bàn cờ (nxn).
* board: Một mảng 2 chiều kích thước boardSize x boardSize, lưu trạng thái các ô trên bàn cờ, có thể chứa các ký tự như 'X', 'O', hoặc dấu trống (ví dụ: ' ').

### Giải thuật

* Duyệt toàn bộ bàn cờ: Hàm sử dụng hai vòng lặp lồng nhau để duyệt qua tất cả các ô trên bàn cờ (từ hàng r = 0 đến r < boardSize và từ cột c = 0 đến c < boardSize).
* Kiểm tra ô trống: Nếu tại bất kỳ ô nào, ký tự là ' ' (trống), hàm sẽ trả về false, chỉ ra rằng bàn cờ vẫn còn ô trống và trò chơi chưa hòa.
* Trả về kết quả hòa: Nếu không tìm thấy ô trống nào, hàm sẽ trả về true, chỉ ra rằng bàn cờ đã đầy và không có ô trống, do đó trò chơi hòa.

### Thư viện

### Mả giả(Pseudo code)

Function checkDraw():

For each row r from 0 to boardSize - 1:

For each column c from 0 to boardSize - 1:

If board[r][c] == ' ':

Return false

Return true

### Độ phức tạp

* Không gian
  + O(n^2): Hàm duyệt qua toàn bộ bàn cờ, với mỗi vòng lặp đi qua n hàng và n cột, nên độ phức tạp thời gian là O(n^2).
* Thời gian
  + O(1): Hàm không sử dụng bộ nhớ phụ thuộc vào kích thước bảng, chỉ sử dụng một vài biến đơn giản, vì vậy độ phức tạp không gian là hằng số O(1).

## undoMove

### Cấu trúc dữ liệu

* stack<Move> moveHistory
  + Dùng để lưu lịch sử các bước đi (dạng LIFO - Last In, First Out).
  + Mỗi bước đi là một Move (giả định là struct hoặc class) với các thuộc tính như:
    - row: hàng của nước đi.
    - col: cột của nước đi.
  + Phục vụ thao tác "undo" bằng cách lấy nước đi cuối cùng và xóa nó khỏi lịch sử.
  + Ưu điểm:
    - Phù hợp với logic "undo" vì chỉ cần thao tác trên phần tử cuối cùng.
    - Truy cập và xóa phần tử cuối cùng có độ phức tạp O(1).
* std::vector<std::vector<char>> board
  + Lưu trạng thái của bàn cờ dưới dạng ma trận 2D.
  + Mỗi phần tử của ma trận đại diện cho một ô, có thể chứa:
    - ' ' (trống),
    - 'X' (nước đi của người chơi),
    - 'O' (nước đi của bot).
  + Phục vụ xóa nước đi bằng cách đặt lại giá trị của ô tương ứng thành ' '
* std::vector markers
  + Lưu các "marker" (ký hiệu hoặc điểm đánh dấu) trên bàn cờ, phục vụ hiển thị giao diện.
  + Sử dụng pop\_back() để xóa marker cuối cùng, đồng bộ với trạng thái bàn cờ.
* Các biến trạng thái
  + bool gameOver: Xác định trò chơi đã kết thúc hay chưa.
  + bool playerWinner và bool botCheck: Theo dõi trạng thái thắng/thua của người chơi và bot.
  + bool isUndoing: Cờ báo hiệu rằng đang thực hiện thao tác "undo".

### Giải thuật

* Kiểm tra điều kiện
  + Kiểm tra if (!moveHistory.empty())
    - Nếu moveHistory trống, không có bước đi nào để "undo", thoát hàm.
* Lùi bước đi cuối cùng
  + Lấy nước đi cuối cùng
    - Move lastMove = moveHistory.top(); lấy nước đi trên cùng.
    - moveHistory.pop(); xóa nước đi này khỏi stack.
  + Cập nhật bàn cờ
    - board[lastMove.row][lastMove.col] = ' '; xóa nước đi trên bàn cờ.
  + Cập nhật giao diện
    - markers.pop\_back(); xóa marker cuối cùng, đồng bộ với trạng thái bàn cờ.
* Lùi thêm 1 bước
  + Tương tự như trên, để xóa nước đi của bot rồi xóa nước đi của người chơi.
* Cập nhật trạng thái trò chơi
  + gameOver = false: Trò chơi chưa kết thúc.
  + playerWinner = false: Không có người chơi nào thắng.
  + botCheck = false: Bot không trong trạng thái kiểm tra điều kiện thắng.
* Vẽ lại giao diện
  + Gọi hàm render() để vẽ lại bàn cờ và các marker sau khi thay đổi.

### Thư viện được dùng

* <stack>
  + Cấu trúc Stack: Phù hợp để lưu lịch sử các bước đi theo dạng LIFO, dễ dàng thực hiện thao tác "undo".
* <vector>
  + Dùng làm ma trận lưu trạng thái bàn cờ (board) và danh sách marker (markers).
  + Hỗ trợ truy cập nhanh và thao tác xóa phần tử cuối cùng với độ phức tạp O(1).

### Pseudo code (Mã giả)

function undoMove():

// Kiểm tra nếu có lịch sử bước đi

if moveHistory is not empty:

// Lấy nước đi cuối cùng

lastMove = moveHistory.top()

moveHistory.pop()

// Xóa nước đi trên bàn cờ

board[lastMove.row][lastMove.col] = ' '

// Xóa marker tương ứng

markers.pop\_back()

// Lặp lại cho bước đi trước đó (nước đi của bot)

if moveHistory is not empty:

lastMove = moveHistory.top()

moveHistory.pop()

// Xóa nước đi trên bàn cờ

board[lastMove.row][lastMove.col] = ' '

// Xóa marker tương ứng

markers.pop\_back()

// Cập nhật trạng thái trò chơi

gameOver = false

playerWinner = false

botCheck = false

// Vẽ lại giao diện

render()

isUndoing = false

### Độ phức tạp

* Không gian
  + Phụ thuộc vào số lượng nước đi lưu trữ trong moveHistory (O(N), với N là số nước đi).
* Thời gian
  + Các thao tác chính (pop, cập nhật board, cập nhật markers, gọi render) đều có độ phức tạp O(1) ngoại trừ render (phụ thuộc vào số ô cờ).

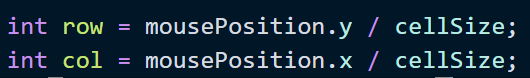
## handleMouseClick

### Cấu trúc dữ liệu

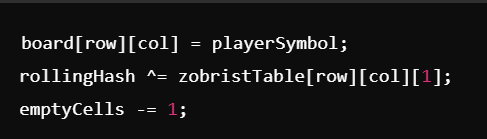
* board
  + Kiểu dữ liệu: std::vector<std::vector<char>>
  + Ý nghĩa: Lưu trạng thái của người chơi trên bàn cờ ('X', hoặc 'O').
  + Cập nhật ô tương ứng với nước đi được cập nhật ngay khi người chơi thực hiện bước đi.
* zobristTable (bảng Zobrist)
  + Kiểu dữ liệu: Mảng 3 chiều (zobristTable[row][col][value]).
  + Ý nghĩa: Bảng băm Zobrist để hỗ trợ tính Rolling Hash.
    - Rolling Hash là kỹ thuật để nhanh chóng cập nhật giá trị băm của bàn cờ khi có thay đổi nhỏ.
    - Ứng dụng: So sánh trạng thái bàn cờ hoặc lưu trữ trong bảng tra cứu cho thuật toán Minimax.
* moveHistory (lịch sử nước đi)
  + Kiểu dữ liệu: std::stack<Move>.
  + Ý nghĩa: Lưu lịch sử các nước đi của người chơi để hỗ trợ chức năng "undo".
* markers (vector chứa các đối tượng đồ họa)
  + Kiểu dữ liệu: std::vector<sf::Text>
  + Ý nghĩa: Lưu các ký hiệu (X, O) trên bàn cờ để hiển thị trực quan với SFML.
* Các biến trạng thái
  + boardSize: Kích thước bàn cờ (nxn).
  + cellSize: Kích thước mỗi ô vuông trên bàn cờ (dùng để tính tọa độ).
  + gameOver: Xác định trạng thái kết thúc của trò chơi.
  + playerSymbol: Ký hiệu của người chơi hiện tại (X hoặc O).
  + turn: Xác định lượt đi (1 cho người chơi, -1 cho bot).
  + botCheck: Theo dõi trạng thái kiểm tra điều kiện thắng của bot.
  + nextBound: Vùng tối ưu để bot đánh (giới hạn không gian tìm kiếm).

### Giải thuật

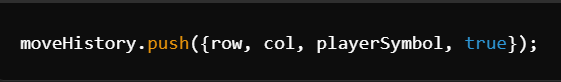
* Xác định tọa độ nhấp chuột.



* Kiểm tra điều kiện hợp lệ
  + Gọi hàm isValidHard để kiểm tra tính hợp lệ nâng cao:
    - Nước đi phải nằm trong biên bàn cờ.
    - Ô phải trống.
    - Trò chơi chưa kết thúc.
* Cập nhật trạng thái bàn cờ
  + Cập nhật giá trị bàn cờ và Zobrist Hash



* + Lưu trạng thái nước đi



* + Hiển thị đồ họa nước đi
    - Tạo đối tượng sf::Text và thêm vào markers.
* Kiểm tra điều kiện thắng hoặc hòa
  + Gọi hàm checkWin để kiểm tra người chơi thắng.
  + Gọi checkDraw để kiểm tra bàn cờ đầy (kết thúc hòa).
* Bot thực hiện nước đi
  + Nếu không có kết thúc (thắng/hòa), gọi botMove() để bot thực hiện nước đi tiếp theo.

### Thư viện

* <SFML/Graphics.hpp>
  + sf::Vector2i: Lưu tọa độ chuột.
  + sf::Text: Tạo các ký hiệu (X, O) và hiển thị trên màn hình.
  + sf::Font: Dùng để đặt font chữ cho ký hiệu.
  + Ứng dụng: Tạo và hiển thị giao diện người chơi.
* <stack>
  + Sử dụng std::stack để quản lý lịch sử nước đi.
* <vector>
  + Sử dụng std::vector để lưu trạng thái bàn cờ và các đối tượng đồ họa.

### Mã giả (Pseudo Code)

function handleMouseClick(mousePosition):

// Chuyển đổi tọa độ chuột thành hàng và cột

row = mousePosition.y / cellSize

col = mousePosition.x / cellSize

// Kiểm tra nước đi có hợp lệ với các điều kiện nâng cao

if isValidHard(row, col, true):

boardValue = evaluate(row, col, boardValue, -1, nextBound)

setState(row, col, -1)

updateBound(row, col, nextBound)

// Kiểm tra điều kiện cơ bản (trong biên, ô trống, game chưa kết thúc)

if row >= 0 && row < boardSize && col >= 0 && col < boardSize && board[row][col] == ' ' && !gameOver:

// Cập nhật trạng thái bàn cờ

board[row][col] = playerSymbol

rollingHash ^= zobristTable[row][col][1]

emptyCells -= 1

currentI = row

currentJ = col

setState(row, col, turn)

turn \*= -1

// Lưu lịch sử nước đi

moveHistory.push({row, col, playerSymbol, true})

// Hiển thị đồ họa

createTextMarker(row, col, playerSymbol)

// Kiểm tra thắng/thua/hòa

if checkWin(row, col, playerSymbol, botCheck):

playerWinner = true

gameOver = true

gameDraw = false

return

if checkDraw():

gameOver = true

gameDraw = true

return

// Bot thực hiện nước đi

botMove()

### Độ phức tạp

* Không gian
  + O(N²): Ma trận board và các dữ liệu liên quan (Zobrist Hash, trạng thái trò chơi).
  + O(H): Lịch sử nước đi (moveHistory), với H là số lượng bước đi tối đa.
* Thời gian
  + O(1): Các thao tác cơ bản (cập nhật board, rollingHash, markers).
  + O(k): Hàm checkWin kiểm tra điều kiện thắng với k là số ô cần kiểm tra (5 ô liên tiếp trong trò chơi Gomoku).

## botMove

### Cấu trúc dữ liệu

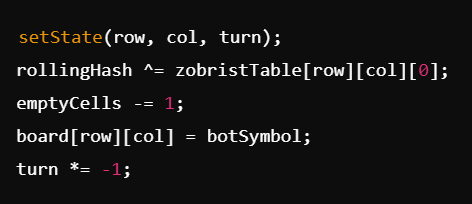
* Board
  + Ô tại (row, col) được gán ký hiệu của bot (botSymbol).
* zobristTable
  + Bảng Zobrist hỗ trợ Rolling Hash.



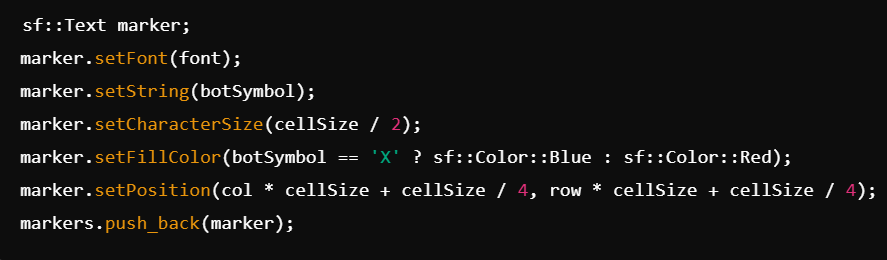
* + 0 là giá trị tương ứng với bot.
  + Rolling Hash được dùng để nhanh chóng so sánh trạng thái bàn cờ trong thuật toán Minimax.
* moveHistory
  + Lưu nước đi của bot.
* Markers
  + Phục vụ hàm render() vẽ bàn cờ.
* Biến trạng thái
  + botDifficulty: Mức độ khó của bot (EASY, MEDIUM, HARD).
  + emptyCells: Số ô trống còn lại trên bàn cờ.
  + turn: Xác định lượt chơi.
  + gameOver, gameDraw: Trạng thái kết thúc trò chơi.
  + botSymbol: Ký hiệu của bot (X hoặc O).

### Giải thuật

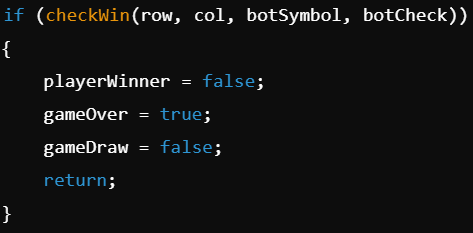
* Chọn nước đi cho bot:
  + Bot chọn vị trí (row, col) dựa trên mức độ khó:
    - EASY: Gọi easyBotMove() (có thể là nước đi ngẫu nhiên hoặc chiến lược đơn giản).
    - MEDIUM: Gọi mediumBotMove() (áp dụng thuật toán Minimax hoặc một chiến lược trung bình).
    - HARD: Gọi hardBotMove() (áp dụng Minimax với Alpha-Beta Pruning).
* Cập nhật trạng thái bàn cờ
  + Kiểm tra tính hợp lệ của nước đi
    - Phải nằm trong bàn cờ và ô phải trống.
  + Cập nhật trạng thái



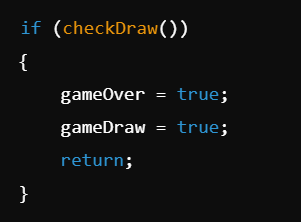
* Hiển thị đồ họa



* Kiểm tra điều kiện thắng



* Kiểm tra điều kiện hòa



### Thư viện

* SFML
  + sf::Vector2i: Lưu tọa độ nước đi của bot.
  + sf::Text: Hiển thị ký hiệu bot (X hoặc O) trên bàn cờ.
  + sf::Font: Dùng để thiết lập font chữ.
* STL
  + std::vector: Lưu trạng thái bàn cờ (board) và danh sách đồ họa (markers).
  + std::stack: Lưu lịch sử nước đi (moveHistory).

### Mã giả (Pseudo Code)

function botMove():

// Chọn vị trí nước đi dựa trên mức độ khó

if botDifficulty == "EASY":

botMovePos = easyBotMove()

else if botDifficulty == "MEDIUM":

botMovePos = mediumBotMove()

else if botDifficulty == "HARD":

botMovePos = hardBotMove()

row = botMovePos.y

col = botMovePos.x

// Kiểm tra tính hợp lệ của nước đi

if row >= 0 and row < boardSize and col >= 0 and col < boardSize and board[row][col] == ' ':

// Cập nhật trạng thái bàn cờ

setState(row, col, turn)

rollingHash ^= zobristTable[row][col][0]

emptyCells -= 1

board[row][col] = botSymbol

turn \*= -1

moveHistory.push({row, col, botSymbol, false})

// Hiển thị ký hiệu bot

createTextMarker(row, col, botSymbol)

// Kiểm tra bot thắng

if checkWin(row, col, botSymbol, botCheck):

playerWinner = false

gameOver = true

gameDraw = false

return

// Kiểm tra hòa

if checkDraw():

gameOver = true

gameDraw = true

return

### Độ phức tạp

* Không gian
  + O(N²): Ma trận board, bảng Zobrist, và các dữ liệu liên quan.
  + O(N): Lịch sử nước đi (moveHistory).
* Thời gian
  + Tùy thuộc vào mức độ khó.

## easyBotMove

### Cấu trúc dữ liệu

* Board
  + Lưu trạng thái bàn cờ.
* sf::Vector2i
  + Đại diện cho một cặp tọa độ (x, y) trên bàn cờ.
  + Hàm trả về một đối tượng Vector2i chứa tọa độ của nước đi được chọn.
* Hàm phụ
  + findBlockMove: Xác định nước đi cần chặn để ngăn người chơi chiến thắng.
  + findExtendMove: Tìm nước đi có lợi để bot mở rộng chuỗi của mình.
  + findSurroundMove: Chọn nước đi gần các bước của người chơi nhằm tạo áp lực hoặc chuẩn bị phòng thủ.

### Giải thuật

* Chặn người chơi:
  + Kiểm tra các vị trí có thể dẫn đến chiến thắng cho người chơi (sử dụng findBlockMove).
  + Trả về tọa độ của nước đi cần chặn nếu có.
* Mở rộng chuỗi:
  + Tìm các vị trí giúp bot kéo dài chuỗi ký hiệu của mình (sử dụng findExtendMove).
  + Nếu có nước đi, trả về tọa độ.
* Đánh xung quanh:
  + Xác định vị trí gần các bước đi trước của người chơi (sử dụng findSurroundMove).
  + Trả về nước đi nếu tìm thấy.
* Đánh ô trống bất kỳ:
  + Duyệt qua toàn bộ bàn cờ để tìm ô trống đầu tiên.
  + Trả về tọa độ của ô trống.
* Trường hợp không có nước đi:
  + Trả về (-1, -1) nếu không còn ô trống trên bàn cờ.

### Thư viện

* SFML: hỗ trợ đồ họa.

### Mả giả(Pseudo code)

function easyBotMove():

// Step 1: Tìm nước đi chặn người chơi

blockMove = findBlockMove()

if blockMove exists:

return blockMove

// Step 2: Tìm nước đi mở rộng chuỗi của bot

extendMove = findExtendMove()

if extendMove exists:

return extendMove

// Step 3: Tìm nước đi gần nước đi của người chơi

surroundMove = findSurroundMove()

if surroundMove exists:

return surroundMove

// Step 4: Đánh ô trống bất kỳ

for row in 0 to boardSize-1:

for col in 0 to boardSize-1:

if board[row][col] == ' ':

return (col, row)

// Không còn nước đi khả thi

return (-1, -1)

### Độ phức tạp

* Không gian
  + O(N2)+O(H): N2 cho ma trận board và bảng Zobrist, H cho lịch sử nước đi.
* Thời gian
  + O(N^2): Các bước đều kiểm tra toàn bộ bàn cờ.

### Các hàm hỗ trợ easyBotMove

#### findBlockMove

##### Cấu trúc dữ liệu

* winCondition: Số nước đi liên tiếp cần thiết để chiến thắng (ví dụ, 5 trong trò chơi Gomoku).
* isValid(): Hàm kiểm tra tính hợp lệ của một ô (nằm trong phạm vi bàn cờ).
* countDirection(): Hàm đếm số lượng nước đi liên tiếp theo một hướng cụ thể từ một điểm bắt đầu.

##### Giải thuật

* Duyệt qua từng ô trên bàn cờ
  + Nếu ô chứa nước đi của người chơi (board[row][col] == playerSymbol), kiểm tra các hướng xung quanh.
* Kiểm tra từng hướng:
  + Duyệt qua 4 hướng (ngang, dọc, chéo xuôi, chéo ngược).
  + Đếm số nước đi liên tiếp của người chơi trong một hướng bằng cách sử dụng countDirection().
* Kiểm tra các đầu chuỗi
  + Nếu có chuỗi đủ gần để chiến thắng (count == winCondition - 1 hoặc count == winCondition - 2), kiểm tra xem có ô trống ở đầu chuỗi hay không (ở cả hai hướng).
* Trả về nước đi chặn
  + Nếu phát hiện một chuỗi gần chiến thắng, trả về ô trống ở đầu chuỗi để chặn người chơi.
* Trả về sf::Vector2i(-1, -1) nếu không tìm thấy nước đi nào cần chặn.

##### Thư viện

* SFML: hỗ trợ giao diện.

##### Mả giả(Pseudo code)

function findBlockMove():

// Định nghĩa các hướng di chuyển: ngang, dọc, chéo

directions = [[0, 1], [1, 0], [1, 1], [1, -1]]

// Duyệt qua từng ô trên bàn cờ

for each cell (row, col) in board:

// Kiểm tra nếu ô này là nước đi của người chơi

if board[row][col] == playerSymbol:

// Kiểm tra các hướng di chuyển (ngang, dọc, chéo)

for each direction (dx, dy) in directions:

count = 1 // Khởi tạo số lần liên tiếp của người chơi

leftEmpty = rightEmpty = false // Biến kiểm tra ô trống ở hai phía

// Tính toán tọa độ của ô bên trái và bên phải theo hướng

leftX = col - dx, leftY = row - dy

rightX = col + dx, rightY = row + dy

// Đếm số nước đi liên tiếp của người chơi theo hai hướng

count += countDirection(row, col, dx, dy, playerSymbol)

count += countDirection(row, col, -dx, -dy, playerSymbol)

// Kiểm tra xem có ô trống ở phía bên trái hoặc bên phải không

if isValid(leftX, leftY) and board[leftY][leftX] == ' ':

leftEmpty = true

if isValid(rightX, rightY) and board[rightY][rightX] == ' ':

rightEmpty = true

// Nếu phát hiện một cơ hội chiến thắng, ngăn chặn bằng cách chặn đường đi

if count == winCondition - 1 or count == winCondition - 2:

if leftEmpty:

return (leftX, leftY) // Chặn đường ở bên trái

if rightEmpty:

return (rightX, rightY) // Chặn đường ở bên phải

return (-1, -1) // Không tìm thấy nước chặn nào

##### Độ phức tạp

* Không gian
  + O(1)
    - Không gian chỉ phụ thuộc vào các biến tạm thời (vị trí, hướng, và trạng thái trống), không thay đổi theo kích thước bàn cờ.
    - Do đó, không gian phức tạp là hằng số O(1).
* Thời gian
  + O(N^3)
    - Vòng lặp ngoài duyệt qua tất cả các ô trên bàn cờ, có độ phức tạp O(N^2).
    - Vòng lặp trong duyệt qua 4 hướng, và mỗi lần gọi hàm countDirection có độ phức tạp O(N) trong trường hợp tồi tệ nhất.
    - Tổng cộng, thời gian phức tạp là O(N^2) \* O(N) = O(N^3).

#### findExtendMove

##### Cấu trúc dữ liệu

* sf::Vector2i: Thư viện SFML được sử dụng để lưu trữ tọa độ 2 chiều (x, y) của một điểm trên bàn cờ. Đây là kiểu dữ liệu đặc biệt trong SFML, dùng để đại diện vị trí trên mặt phẳng 2D.
* board: Là bàn cờ dạng ma trận 2 chiều, mỗi ô có giá trị là ký tự (ví dụ: ' ', 'X', 'O', 'b'…).
* directions[4][2]: Mảng 2D chứa 4 hướng di chuyển: ngang, dọc, chéo xuôi và chéo ngược.

##### Giải thuật

* Đầu tiên, hàm duyệt qua tất cả các ô trên bàn cờ để tìm các ô là nước đi của bot (botSymbol).
* Sau đó, cho mỗi ô này, kiểm tra tất cả 4 hướng di chuyển (ngang, dọc, chéo).
* Hàm đếm số nước đi liên tiếp của bot bằng cách gọi countDirection cho mỗi hướng, sau đó kiểm tra ô tiếp theo theo hướng đó (nếu còn trống).
* Nếu có ô trống tiếp theo và là một nước đi hợp lệ, trả về tọa độ của ô đó.
* Nếu không tìm thấy nước đi mở rộng, trả về (-1, -1).

##### Thư viện

* SFML: hỗ trợ giao diện.

##### Mả giả(Pseudo code)

function findExtendMove():

directions = [[0, 1], [1, 0], [1, 1], [1, -1]] // Các hướng di chuyển: ngang, dọc, chéo xuôi, chéo ngược

for each row from 0 to boardSize - 1:

for each col from 0 to boardSize - 1:

if board[row][col] != botSymbol: // Kiểm tra nếu ô hiện tại không phải của bot

continue

for each direction in directions: // Kiểm tra các hướng

count = 1 // Khởi tạo đếm chuỗi liên tiếp của bot

nextX = col + direction[1] // Tính tọa độ ô tiếp theo theo hướng

nextY = row + direction[0]

// Đếm số lượng nước đi liên tiếp của bot trong cả hai hướng

count += countDirection(row, col, direction[0], direction[1], botSymbol)

// Kiểm tra xem ô tiếp theo có hợp lệ và trống không

if isValid(nextY, nextX) and board[nextY][nextX] == ' ':

return (nextX, nextY) // Trả về vị trí nước đi mở rộng

return (-1, -1) // Không tìm thấy nước đi mở rộng

##### Độ phức tạp

* Không gian
  + O(1): Sử dụng các biến tạm thời
* Thời gian
  + O(boardSize^3)
    - Vòng lặp ngoài duyệt qua mỗi ô trên bàn cờ: O(N^2).
    - Vòng lặp trong duyệt qua 4 hướng di chuyển, mỗi lần gọi countDirection có độ phức tạp O(N) trong trường hợp xấu nhất (nếu cần kiểm tra chiều dài bàn cờ).
    - Tổng độ phức tạp thời gian là: O(N^2 \* N) = O(N^3).

#### findSurroundMove

##### Cấu trúc dữ liệu

* sf::Vector2i: Đây là kiểu dữ liệu 2D của thư viện SFML, dùng để lưu trữ và trả về tọa độ của ô trên bàn cờ (vị trí x, y).
* directions[4][2]: Mảng chứa các hướng di chuyển trên bàn cờ (ngang, dọc, chéo xuôi và chéo ngược). Mỗi phần tử là một mảng 2 chiều [dx, dy] xác định hướng di chuyển.
* board: Là bàn cờ, một ma trận 2 chiều đại diện cho trạng thái các ô (mỗi ô có thể là trống, hoặc ký hiệu của người chơi hoặc bot).
* botSymbol: Ký hiệu của bot trên bàn cờ (ví dụ: 'X' hoặc 'O').

##### Giải thuật

* Duyệt qua các ô của bàn cờ: Đầu tiên, hàm kiểm tra tất cả các ô trên bàn cờ. Nếu ô đó không phải của bot (botSymbol), hàm tiếp tục kiểm tra ô tiếp theo.
* Kiểm tra các hướng: Sau khi tìm thấy ô của bot, hàm kiểm tra tất cả 4 hướng di chuyển: ngang, dọc, chéo xuôi và chéo ngược. Mỗi hướng được xác định bởi một cặp giá trị [dx, dy].
* Đếm chuỗi liên tiếp: Hàm countDirection được gọi để đếm số lượng nước đi liên tiếp của bot trong cả hai chiều (tiến và lùi) theo mỗi hướng.
* Kiểm tra nước mở rộng: Sau khi đếm được số lượng nước đi liên tiếp, hàm tính toán tọa độ của ô tiếp theo theo hướng di chuyển. Nếu ô đó hợp lệ (nằm trong bàn cờ) và trống, hàm trả về tọa độ đó như là một nước đi mở rộng của bot.
* Không tìm thấy nước mở rộng: Nếu không có nước mở rộng nào được tìm thấy, hàm trả về sf::Vector2i(-1, -1).

##### Thư viện

* SFML: hỗ trợ giao diện

##### Mả giả(Pseudo code)

function findExtendMove():

directions = [[0, 1], [1, 0], [1, 1], [1, -1]] // Các hướng di chuyển: ngang, dọc, chéo xuôi, chéo ngược

for each row from 0 to boardSize - 1:

for each col from 0 to boardSize - 1:

if board[row][col] != botSymbol: // Kiểm tra nếu ô hiện tại không phải của bot

continue

for each direction in directions: // Kiểm tra các hướng

count = 1 // Khởi tạo đếm chuỗi liên tiếp của bot

nextX = col + direction[1] // Tính tọa độ ô tiếp theo theo hướng

nextY = row + direction[0]

// Đếm số lượng nước đi liên tiếp của bot trong cả hai hướng

count += countDirection(row, col, direction[0], direction[1], botSymbol)

// Kiểm tra xem ô tiếp theo có hợp lệ và trống không

if isValid(nextY, nextX) and board[nextY][nextX] == ' ':

return (nextX, nextY) // Trả về vị trí nước đi mở rộng

return (-1, -1) // Không tìm thấy nước đi mở rộng

##### Độ phức tạp

* Không gian
* Không gian
  + O(1): Sử dụng các biến tạm thời
* Thời gian
  + O(boardSize^3)
    - Vòng lặp ngoài duyệt qua mỗi ô trên bàn cờ: O(N^2).

##### Vòng lặp trong duyệt qua 4 hướng di chuyển, mỗi lần gọi countDirection có độ phức tạp O(N) trong trường hợp xấu nhất (nếu cần kiểm tra chiều dài bàn cờ).

* + - Tổng độ phức tạp thời gian là: O(N^2 \* N) = O(N^3).

#### findSurroundMove

##### Cấu trúc dữ liệu

* sf::Vector2i: Là kiểu dữ liệu 2D của thư viện SFML, dùng để lưu trữ tọa độ x và y (vị trí ô trên bàn cờ).
* offsets[8][2]: Mảng chứa các offset (điều chỉnh) để kiểm tra 8 ô xung quanh một ô đã được chọn. Bao gồm các hướng: trên trái, trên, trên phải, trái, phải, dưới trái, dưới, dưới phải.
* board: Bàn cờ, lưu trạng thái của các ô trên bàn.
* playerSymbol: Ký hiệu của người chơi (ví dụ: 'X' hoặc 'O').

##### Giải thuật

* Duyệt qua bàn cờ: Hàm duyệt qua tất cả các ô trên bàn cờ. Nếu ô chứa ký hiệu của người chơi (playerSymbol), hàm tiếp tục kiểm tra các ô xung quanh.
* Kiểm tra xung quanh: Với mỗi ô của người chơi, hàm kiểm tra 8 ô xung quanh theo các offset có sẵn trong mảng offsets. Các hướng này bao gồm các hướng chéo, ngang và dọc.
* Kiểm tra ô trống: Nếu ô xung quanh là hợp lệ (nằm trong bàn cờ) và là ô trống (chứa ký hiệu ' '), hàm trả về tọa độ của ô đó như là nước đi tiếp theo.
* Không tìm thấy nước đi: Nếu không có nước đi nào xung quanh, hàm trả về sf::Vector2i(-1, -1).

##### Thư viện

* SFML: hỗ trợ giao diện.

##### Mả giả(Pseudo code)

function findSurroundMove():

offsets = [[-1, -1], [-1, 0], [-1, 1], [0, -1], [0, 1], [1, -1], [1, 0], [1, 1]] // Các hướng xung quanh ô hiện tại

for each row from 0 to boardSize - 1:

for each col from 0 to boardSize - 1:

if board[row][col] == playerSymbol: // Kiểm tra nếu ô hiện tại là nước đi của người chơi

for each offset in offsets: // Kiểm tra các hướng xung quanh

x = col + offset[1] // Tính tọa độ của ô xung quanh theo offset

y = row + offset[0]

// Kiểm tra nếu ô xung quanh hợp lệ và trống

if isValid(y, x) and board[y][x] == ' ':

return (x, y) // Trả về vị trí ô trống xung quanh ô của người chơi

return (-1, -1) // Không tìm thấy nước đi xung quanh

##### Độ phức tạp

* Không gian
  + O(1): Sử dụng các biến tạm thời.
* Thời gian
  + O(N^2)
    - Vòng lặp ngoài duyệt qua tất cả các ô trên bàn cờ: O(N^2).
    - Vòng lặp trong duyệt qua 8 hướng xung quanh mỗi ô: O(1).
    - Tổng độ phức tạp thời gian là: O(N^2).

## mediumBotMove

### Cấu trúc dữ liệu

* board[row][col]: Ma trận 2D biểu diễn bàn cờ (kích thước boardSize x boardSize).
* sf::Vector2i: Cặp tọa độ (x, y) được sử dụng để trả về nước đi hợp lệ.
* Mảng directions[4][2]: Lưu các hướng chính (ngang, dọc, chéo xuôi, chéo ngược) để kiểm tra chuỗi liên tiếp.
* Các hàm phụ trợ:
  + countDirection: Đếm số quân liên tiếp của một bên theo hướng xác định.
  + checkOpenEnds: Xác định số đầu chuỗi mở.
  + detectFourBlocked: Phát hiện chuỗi 4 bị chặn của người chơi.
  + findBlockingMove: Tìm tọa độ để chặn/mở rộng chuỗi dựa trên hướng.

### Giải thuật

* Mở rộng chuỗi để tạo cơ hội thắng (getBestExtendMove).
* Chặn người chơi nếu họ đang có thế nguy hiểm (getBestBlockingMove).
* Đánh xung quanh nước đi của người chơi (findSurroundMove).
* Đánh một ô trống bất kỳ nếu không có lựa chọn nào khả thi.

### Thư viện

* SFML: hỗ trợ giao hiện.

### Mả giả(Pseudo code)

function mediumBotMove():

extendMove = getBestExtendMove()

if extendMove is valid:

return extendMove

blockMove = getBestBlockingMove()

if blockMove is valid:

return blockMove

surroundMove = findSurroundMove()

if surroundMove is valid:

return surroundMove

for each cell in board:

if cell is empty:

return cell position

return (-1, -1) // No valid move found

### Độ phức tạp

* Không gian
  + O(1): sử dụng các biến tạm thời.
* Thời gian
  + Tùy thuộc vào các hàm phụ trợ

### Các hàm phụ trợ mediumBotMove

#### getBestExtendMove

##### Cấu trúc dữ liệu

* board[row][col]: Ma trận 2D biểu diễn trạng thái bàn cờ.
* directions[4][2]: Các hướng chính (ngang, dọc, chéo xuôi, chéo ngược) để kiểm tra chuỗi liên tiếp.
* Các hàm hỗ trợ:
  + countDirection: Đếm số quân liên tiếp của bot trong một hướng.
  + checkOpenEnds: Xác định số đầu chuỗi mở cho một chuỗi quân.
  + findBlockingMove: Tìm vị trí tốt nhất để chặn hoặc mở rộng chuỗi.
  + detectFourBlocked: Phát hiện chuỗi 4 quân liên tiếp của đối thủ cần chặn.

##### Giải thuật

* Xác định các chuỗi quân hiện tại của bot.
* Kiểm tra các nước đi tiềm năng dựa trên số quân liên tiếp (countDirection) và trạng thái "mở" ở hai đầu chuỗi (checkOpenEnds).
* Ưu tiên các nước đi quan trọng:
  + Chặn đối thủ: Nếu đối thủ có chuỗi 4 quân liên tiếp, bot phải chặn ngay.
  + Tạo chuỗi thắng: Nếu bot có chuỗi 4 quân liên tiếp và có ít nhất một đầu chuỗi mở, đánh để thắng.
  + Mở rộng chuỗi: Nếu bot có chuỗi 3 quân liên tiếp và cả hai đầu chuỗi mở, đánh để tăng chuỗi.

##### Thư viện

* SFML: hỗ trợ giao diện.

##### Mả giả(Pseudo code)

function getBestExtendMove():

directions = [(0, 1), (1, 0), (1, 1), (1, -1)] # Các hướng cần kiểm tra

for row from 0 to boardSize - 1:

for col from 0 to boardSize - 1:

if board[row][col] != botSymbol:

continue # Chỉ kiểm tra các ô chứa quân của bot

for dir in directions:

# Đếm số quân liên tiếp của bot trong hướng dir

count = countDirection(row, col, dir[0], dir[1], botSymbol) + 1

# 1. Ưu tiên chặn chuỗi 4 của đối thủ

blockMove = detectFourBlocked()

if blockMove is valid:

return blockMove

# 2. Bot có 4 quân liên tiếp, đánh để thắng

if count == 4 and checkOpenEnds(row, col, dir, count) >= 1:

return findBlockingMove(row, col, dir, count)

# 3. Bot có 3 quân liên tiếp với hai đầu chuỗi mở

if count == 3 and checkOpenEnds(row, col, dir, count) == 2:

return findBlockingMove(row, col, dir, count)

return (-1, -1) # Không tìm thấy nước đi phù hợp

##### Độ phức tạp

* Không gian
  + O(1): sử dụng các biến tạm thời
* Thời gian
  + O(N^2⋅maxDepth).
    - Duyệt qua toàn bộ bàn cờ O(N^2).
    - Với mỗi ô, kiểm tra 4 hướng, độ dài tối đa của 1 chuỗi: maxDeapth.

#### getBestBlockingMove

Ý tưởng: hàm getBestBlockingMove được thiết kế để xác định nước đi ưu tiên chặn các thế nguy hiểm nhất của người chơi. Bot sẽ lần lượt kiểm tra từng kiểu thế trận nguy hiểm theo mức độ cao nhất đến thấp, và thực hiện nước đi đầu tiên phát hiện được. Các bước chính của giải thuật:

* Xử lý đặc biệt với bàn cờ nhỏ (< 5x5):
  + Ưu tiên chặn chuỗi 2 liên tiếp của người chơi (detectTwoConsecutive) vì đây là điều kiện nguy hiểm trên bàn cờ nhỏ.
* Kiểm tra theo thứ tự ưu tiên các thế nguy hiểm của người chơi:
  + detectAlmostFive: Phát hiện chuỗi gần thắng (4 quân liên tiếp, không bị chặn cả hai đầu).
  + Nếu tìm thấy, thực hiện ngay để chặn người chơi chiến thắng.
  + detectFourBlocked: Chuỗi 4 quân bị chặn một đầu, nhưng đầu còn lại có thể hoàn thành 5 quân.
  + detectThreeOpen: Chuỗi 3 quân liên tiếp với cả hai đầu chuỗi mở.
  + detectDoubleThree: Hai chuỗi 3 quân mở có thể kết hợp, dẫn đến nguy cơ chiến thắng.
  + detectFourThree: Một chuỗi 4 quân liên tiếp và một chuỗi 3 quân mở, nguy hiểm cao.
  + detectTwoFourParallel: Hai chuỗi 4 quân song song có tiềm năng kết hợp để chiến thắng.
  + detectGapPair: Hai quân bị ngắt quãng bởi một ô trống nhưng có tiềm năng phát triển.
  + detectLOrTShape: Các hình dạng đặc biệt (L hoặc T) với nhiều hướng phát triển.
  + detectOpenTwo: Chuỗi 2 quân liên tiếp với cả hai đầu mở.
* Trả về nước đi đầu tiên phát hiện được:
  + Với mỗi kiểu thế nguy hiểm, nếu hàm phát hiện nước đi phù hợp (khác (-1,-1)), trả về ngay lập tức.
  + Nếu tất cả các kiểu thế được kiểm tra nhưng không tìm thấy, trả về(-1,-1), nghĩa là không có nước đi cần chặn.

##### detectTwoConsecutive

###### Cấu trúc dữ liệu

* board: Ma trận boardSize x boardSize lưu trạng thái của bàn cờ. Mỗi ô lưu ký hiệu (playerSymbol, botSymbol, hoặc rỗng).
* directions: Mảng lưu các hướng kiểm tra (ngang, dọc, chéo xuôi, chéo ngược) dưới dạng vector 2D (dx, dy).

###### Giải thuật

* Tìm kiếm trên bàn cờ (O(n²)):
  + Duyệt qua tất cả các ô trên bàn cờ bằng hai vòng lặp row, col.
  + Bỏ qua các ô không thuộc ký hiệu của người chơi (playerSymbol).
* Kiểm tra hướng cụ thể (countDirection & checkOpenEnds):
  + Với mỗi ô, kiểm tra 4 hướng.
  + Tính số ký hiệu liên tiếp (countDirection) và kiểm tra xem chuỗi có "mở" ở một đầu hay không (checkOpenEnds).
  + Nếu thỏa mãn chuỗi 2 ký hiệu liên tiếp và có ít nhất một đầu mở, trả về nước đi chặn (findBlockingMove).

###### Thư viện

* SFML: hỗ trợ giao diện

###### Mả giả

FUNCTION detectTwoConsecutive():

directions ← {{0, 1}, {1, 0}, {1, 1}, {1, -1}} // Các hướng: ngang, dọc, chéo xuôi, chéo ngược

FOR row FROM 0 TO boardSize - 1:

FOR col FROM 0 TO boardSize - 1:

IF board[row][col] ≠ playerSymbol:

CONTINUE // Bỏ qua ô không phải của đối thủ

FOR dir IN directions:

count ← countDirection(row, col, dir[0], dir[1], playerSymbol) + 1

IF count = 2 AND checkOpenEnds(row, col, dir, count) = 1:

RETURN findBlockingMove(row, col, dir, count) // Trả về tọa độ nước chặn

RETURN sf::Vector2i(-1, -1) // Không tìm thấy chuỗi 2 liên tiếp

###### Độ phức tạp

* Không gian
  + O(n²): quét bàn cờ.
* Thời gian
  + O(n²): Duyệt qua mỗi ô, mỗi ô kiểm tra 4 hướng.

##### detectAlmostFive

###### Cấu trúc dữ liệu

* Board: Ma trận board[boardSize][boardSize] (kích thước n x n), lưu trữ trạng thái của các ô trên bàn cờ.
* directions: Mảng 2D lưu các hướng kiểm tra (ngang, dọc, chéo xuôi, chéo ngược), kích thước 4x2.

###### Giải thuật

* Duyệt bàn cờ: Duyệt từng ô trong ma trận, kiểm tra nếu ô là của đối thủ (dựa trên playerSymbol).
* Kiểm tra chuỗi gần thắng: Với mỗi ô của đối thủ, kiểm tra các hướng để tìm chuỗi có 4 ký tự liên tiếp và có 2 đầu mở (không bị chặn).
* Tính toán nước chặn: Nếu tìm thấy chuỗi gần thắng, tính toán và trả về nước đi chặn.

###### Thư viện

* SFML: hỗ trợ giao diện.

###### Mả giả(Pseudo code)

FUNCTION detectAlmostFive():

directions ← {{0, 1}, {1, 0}, {1, 1}, {1, -1}} // Các hướng: ngang, dọc, chéo xuôi, chéo ngược

FOR row FROM 0 TO boardSize - 1:

FOR col FROM 0 TO boardSize - 1:

IF board[row][col] ≠ playerSymbol:

CONTINUE // Bỏ qua ô không phải của đối thủ

FOR dir IN directions:

count ← countDirection(row, col, dir[0], dir[1], playerSymbol) + 1

IF count = 4 AND checkOpenEnds(row, col, dir, count) = 2:

RETURN findBlockingMove(row, col, dir, count) // Trả về nước chặn

RETURN sf::Vector2i(-1, -1) // Không tìm thấy thế gần thắng

###### Độ phức tạp

* Không gian
  + O(n²): Quét toàn bộ bàn cờ.
* Thời gian
  + O(n²): Duyệt toàn bộ bàn cờ và kiểm tra từng ô.

##### detectFourBlocked

###### Cấu trúc dữ liệu

* Board: Ma trận board[boardSize][boardSize], lưu trữ trạng thái các ô trên bàn cờ.
* directions: Mảng 2D gồm các hướng kiểm tra (ngang, dọc, chéo xuôi, chéo ngược).

###### Giải thuật

* Duyệt qua bàn cờ: Duyệt từng ô trong ma trận. Nếu ô là của đối thủ (playerSymbol), kiểm tra các hướng xem có chuỗi 4 quân liên tiếp bị chặn một đầu.
* Kiểm tra chuỗi 4 bị chặn: Nếu chuỗi 4 quân liên tiếp bị chặn một đầu (bởi quân của đối thủ hoặc biên), trả về nước đi chặn.
* Tính toán nước chặn: Tìm và trả về vị trí nước đi có thể chặn chuỗi đối thủ.

###### Thư viện

* SFML: Sử dụng sf::Vector2i để lưu trữ tọa độ trả về của nước đi chặn.

###### Mả giả(Pseudo code)

FUNCTION detectFourBlocked():

directions ← {{0, 1}, {1, 0}, {1, 1}, {1, -1}} // Các hướng kiểm tra

FOR row FROM 0 TO boardSize - 1:

FOR col FROM 0 TO boardSize - 1:

IF board[row][col] ≠ playerSymbol:

CONTINUE // Bỏ qua ô không thuộc đối thủ

FOR dir IN directions:

count ← countDirection(row, col, dir[0], dir[1], playerSymbol) + 1

IF count = 4 AND checkOpenEnds(row, col, dir, count) = 1:

dangerous ← true

RETURN findBlockingMove(row, col, dir, count) // Trả về nước đi chặn

RETURN sf::Vector2i(-1, -1) // Không tìm thấy thế 4 bị chặn

###### Độ phức tạp

* Không gian
  + O(n²): Quét toàn bộ bàn cờ.
* Thời gian
  + O(n²): Duyệt toàn bộ bàn cờ và kiểm tra từng ô.

##### detectThreeOpen

###### Cấu trúc dữ liệu

* Board: Ma trận board[boardSize][boardSize], lưu trữ trạng thái các ô trên bàn cờ.
* directions: Mảng 2D gồm các hướng kiểm tra (ngang, dọc, chéo xuôi, chéo ngược).

###### Giải thuật

* Duyệt qua bàn cờ: Duyệt từng ô trong ma trận. Nếu ô là của đối thủ (playerSymbol), kiểm tra các hướng để tìm chuỗi 3 quân liên tiếp và hai đầu đều mở.
* Kiểm tra chuỗi 3 quân liên tiếp: Nếu tìm thấy chuỗi 3 quân của đối thủ và cả hai đầu đều mở (không bị chặn), trả về nước đi chặn.
* Tính toán nước chặn: Tìm và trả về vị trí nước đi có thể chặn chuỗi đối thủ.

###### Thư viện

* SFML: Sử dụng sf::Vector2i để lưu trữ tọa độ trả về của nước đi chặn.

###### Mả giả

FUNCTION detectThreeOpen():

directions ← {{0, 1}, {1, 0}, {1, 1}, {1, -1}} // Các hướng kiểm tra

FOR row FROM 0 TO boardSize - 1:

FOR col FROM 0 TO boardSize - 1:

IF board[row][col] ≠ playerSymbol:

CONTINUE // Bỏ qua ô không thuộc đối thủ

FOR dir IN directions:

count ← countDirection(row, col, dir[0], dir[1], playerSymbol) + 1

IF count = 3 AND checkOpenEnds(row, col, dir, count) = 2:

RETURN findBlockingMove(row, col, dir, count) // Trả về nước đi chặn

RETURN sf::Vector2i(-1, -1) // Không tìm thấy thế 3 mở

###### Độ phức tạp

* Không gian
  + O(n²): Quét toàn bộ bàn cờ.
* Thời gian
  + O(n²): Duyệt toàn bộ bàn cờ và kiểm tra từng ô.

##### detectThreeOpen

###### Cấu trúc dữ liệu

* Board: Ma trận board[boardSize][boardSize] lưu trữ trạng thái các ô trên bàn cờ (có thể là quân của người chơi hoặc đối thủ).
* directions: Mảng 2D chứa 4 hướng kiểm tra (ngang, dọc, chéo xuôi, chéo ngược).

###### Giải thuật

* Duyệt qua bàn cờ: Duyệt từng ô trong ma trận, nếu ô đó chứa quân của đối thủ (playerSymbol), tiếp tục kiểm tra các hướng.
* Kiểm tra chuỗi 3 quân liên tiếp: Nếu tại ô đó và trong một trong 4 hướng, tìm thấy chuỗi 3 quân liên tiếp của đối thủ, kiểm tra xem cả hai đầu của chuỗi có bị chặn hay không.
* Tính toán nước chặn: Nếu phát hiện được chuỗi 3 quân liên tiếp và cả hai đầu đều mở, hàm trả về nước đi có thể chặn chuỗi đối thủ.

###### Thư viện

* SFML: Sử dụng sf::Vector2i để lưu trữ và trả về tọa độ nước đi chặn.

###### Mả giả(Pseudo code)

FUNCTION detectThreeOpen():

directions ← {{0, 1}, {1, 0}, {1, 1}, {1, -1}} // Các hướng kiểm tra

FOR row FROM 0 TO boardSize - 1:

FOR col FROM 0 TO boardSize - 1:

IF board[row][col] ≠ playerSymbol:

CONTINUE // Bỏ qua ô không thuộc đối thủ

FOR dir IN directions:

count ← countDirection(row, col, dir[0], dir[1], playerSymbol) + 1

IF count = 3 AND checkOpenEnds(row, col, dir, count) = 2:

RETURN findBlockingMove(row, col, dir, count) // Trả về nước đi chặn

RETURN sf::Vector2i(-1, -1) // Không tìm thấy thế 3 mở

###### Độ phức tạp

* Không gian
  + O(n²): Quét toàn bộ bàn cờ.
* Thời gian
  + O(n²): Duyệt toàn bộ bàn cờ và kiểm tra từng ô.

##### detectDoubleThree

###### Cấu trúc dữ liệu

* Board: Ma trận board[boardSize][boardSize], lưu trữ trạng thái các ô trên bàn cờ.
* directions: Mảng 2D chứa 4 hướng kiểm tra (ngang, dọc, chéo xuôi, chéo ngược).
* potentialMoves: Vector lưu trữ các nước đi có thể chặn chuỗi 3 quân liên tiếp không bị chặn hai đầu.

###### Giải thuật

* Duyệt qua bàn cờ: Duyệt từng ô trên bàn cờ. Nếu ô đó chứa quân của đối thủ (playerSymbol), kiểm tra các hướng để tìm chuỗi 3 quân liên tiếp.
* Kiểm tra chuỗi 3 không bị chặn: Kiểm tra chuỗi 3 quân không bị chặn từ cả hai đầu. Nếu có, tìm nước đi có thể chặn chuỗi này và thêm vào danh sách potentialMoves.
* Kiểm tra 3-3 kép: Nếu danh sách potentialMoves có ít nhất 2 nước đi, trả về nước đi đầu tiên từ danh sách đó.

###### Thư viện

* SFML: Sử dụng sf::Vector2i để lưu trữ và trả về tọa độ nước đi chặn.

###### Mả giả

FUNCTION detectDoubleThree():

directions ← {{0, 1}, {1, 0}, {1, 1}, {1, -1}} // Các hướng kiểm tra

FOR row FROM 0 TO boardSize - 1:

FOR col FROM 0 TO boardSize - 1:

IF board[row][col] ≠ playerSymbol:

CONTINUE // Bỏ qua ô không thuộc đối thủ

potentialMoves ← [] // Khởi tạo danh sách các nước đi tiềm năng

FOR dir IN directions:

count ← countDirection(row, col, dir[0], dir[1], playerSymbol) + 1

IF count = 3 AND checkOpenEnds(row, col, dir, count) = 2:

blockMove ← findBlockingMove(row, col, dir, count)

IF blockMove IS VALID:

ADD blockMove TO potentialMoves

IF length of potentialMoves ≥ 2:

RETURN potentialMoves[0] // Trả về nước đi đầu tiên trong danh sách

RETURN sf::Vector2i(-1, -1) // Không có thế 3-3 kép

###### Độ phức tạp

* Không gian
  + O(n²): Quét toàn bộ bàn cờ.
* Thời gian
  + O(n²): Duyệt toàn bộ bàn cờ và kiểm tra từng ô.

##### detectDoubleThree

###### Cấu trúc dữ liệu

* Board: Ma trận board[boardSize][boardSize], lưu trữ trạng thái các ô trên bàn cờ.
* directions: Mảng 2D chứa 4 hướng kiểm tra (ngang, dọc, chéo xuôi, chéo ngược).
* potentialMoves: Vector lưu trữ các nước đi có thể chặn chuỗi 3 quân liên tiếp không bị chặn hai đầu.

###### Giải thuật

* Duyệt qua bàn cờ: Duyệt từng ô trên bàn cờ. Nếu ô đó chứa quân của đối thủ (playerSymbol), kiểm tra các hướng để tìm chuỗi 3 quân liên tiếp.
* Kiểm tra chuỗi 3 không bị chặn: Kiểm tra chuỗi 3 quân không bị chặn từ cả hai đầu. Nếu có, tìm nước đi có thể chặn chuỗi này và thêm vào danh sách potentialMoves.
* Kiểm tra 3-3 kép: Nếu danh sách potentialMoves có ít nhất 2 nước đi, trả về nước đi đầu tiên từ danh sách đó.

###### Thư viện

* SFML: Sử dụng sf::Vector2i để lưu trữ và trả về tọa độ nước đi chặn.

###### Mả giả(Pseudo code)

FUNCTION detectDoubleThree():

directions ← {{0, 1}, {1, 0}, {1, 1}, {1, -1}} // Các hướng kiểm tra

FOR row FROM 0 TO boardSize - 1:

FOR col FROM 0 TO boardSize - 1:

IF board[row][col] ≠ playerSymbol:

CONTINUE // Bỏ qua ô không thuộc đối thủ

potentialMoves ← [] // Khởi tạo danh sách các nước đi tiềm năng

FOR dir IN directions:

count ← countDirection(row, col, dir[0], dir[1], playerSymbol) + 1

IF count = 3 AND checkOpenEnds(row, col, dir, count) = 2:

blockMove ← findBlockingMove(row, col, dir, count)

IF blockMove IS VALID:

ADD blockMove TO potentialMoves

IF length of potentialMoves ≥ 2:

RETURN potentialMoves[0] // Trả về nước đi đầu tiên trong danh sách

RETURN sf::Vector2i(-1, -1) // Không có thế 3-3 kép

###### Độ phức tạp

* Không gian
  + O(n²): Quét toàn bộ bàn cờ.
* Thời gian
  + O(n²): Duyệt toàn bộ bàn cờ và kiểm tra từng ô.

##### detectFourThree

###### Cấu trúc dữ liệu

* Board: Ma trận board[boardSize][boardSize] chứa trạng thái của bàn cờ.
* directions: Mảng 2D gồm 4 hướng kiểm tra (ngang, dọc, chéo xuôi, chéo ngược).
* Các biến:
  + countFour: Đếm số quân của đối thủ trong chuỗi 4 theo hướng.
  + countThree: Đếm số quân của đối thủ trong chuỗi 3 theo hướng ngược lại.

###### Giải thuật

* Duyệt qua bàn cờ: Duyệt từng ô trên bàn cờ. Nếu ô đó chứa quân của đối thủ (playerSymbol), kiểm tra các hướng để tìm chuỗi 4 và chuỗi 3.
* Đếm chuỗi 4 và 3: Đếm số quân của đối thủ trong chuỗi 4 và 3 theo hướng hiện tại và ngược lại.
* Kiểm tra điều kiện:
  + Ưu tiên chặn chuỗi 4 quân trước.
  + Nếu không thể chặn chuỗi 4, chặn chuỗi 3.
* Trả về nước đi: Nếu tìm thấy nước đi hợp lệ để chặn, trả về tọa độ nước đi đó.

###### Thư viện

* SFML: Sử dụng sf::Vector2i để lưu trữ tọa độ nước đi cần chặn.

###### Mả giả(Pseudo code)

FUNCTION detectFourThree():

directions ← {{0, 1}, {1, 0}, {1, 1}, {1, -1}} // Các hướng kiểm tra

FOR row FROM 0 TO boardSize - 1:

FOR col FROM 0 TO boardSize - 1:

IF board[row][col] ≠ playerSymbol:

CONTINUE // Bỏ qua ô không thuộc đối thủ

FOR dir IN directions:

countFour ← countDirection(row, col, dir[0], dir[1], playerSymbol) + 1

countThree ← countDirection(row, col, -dir[0], -dir[1], playerSymbol) + 1

IF countFour = 4 AND checkOpenEnds(row, col, dir, countFour) = 1 AND countThree = 3 AND checkOpenEnds(row, col, dir, countThree) = 2:

blockMoveFour ← findBlockingMove(row, col, dir, countFour)

IF blockMoveFour IS VALID:

RETURN blockMoveFour // Trả về nước đi chặn chuỗi 4

blockMoveThree ← findBlockingMove(row, col, dir, countThree)

IF blockMoveThree IS VALID:

RETURN blockMoveThree // Trả về nước đi chặn chuỗi 3

RETURN sf::Vector2i(-1, -1) // Không có thế 4-3

###### Độ phức tạp

* Không gian
  + O(n²): Quét toàn bộ bàn cờ.
* Thời gian
  + O(n²): Duyệt toàn bộ bàn cờ và kiểm tra từng ô.

##### detectTwoFourParallel

###### Cấu trúc dữ liệu

* Board: Ma trận board[boardSize][boardSize] chứa trạng thái bàn cờ.
* directions: Mảng 2D gồm 4 hướng kiểm tra (ngang, dọc, chéo xuôi, chéo ngược).
* Các biến:
  + fourCount: Đếm số chuỗi 4 quân bị chặn.
  + blockingMoves: Lưu các nước đi có thể chặn.
  + blockMove: Lưu nước đi chặn cuối cùng.

###### Giải thuật

* Duyệt qua bàn cờ: Duyệt từng ô trên bàn cờ. Nếu ô chứa quân của đối thủ (playerSymbol), tiếp tục kiểm tra các hướng.
* Đếm chuỗi 4 bị chặn:
  + Kiểm tra chuỗi 4 trong mỗi hướng.
  + Nếu tìm thấy chuỗi 4 bị chặn một đầu, lưu vị trí chặn vào blockingMoves.
* Kiểm tra chuỗi 4 song song:
  + Nếu có ít nhất 2 chuỗi 4 song song, ưu tiên chặn tại vị trí có thể phá cả hai chuỗi.
  + Nếu không có nước chặn chung, trả về nước đầu tiên trong danh sách blockingMoves.
* Trả về nước đi: Nếu tìm được nước đi hợp lệ để chặn, trả về tọa độ của nước đi đó.

###### Thư viện

* SFML: Sử dụng sf::Vector2i để lưu trữ tọa độ nước đi cần chặn.

###### Mả giả

FUNCTION detectTwoFourParallel():

directions ← {{0, 1}, {1, 0}, {1, 1}, {1, -1}} // Các hướng kiểm tra

FOR row FROM 0 TO boardSize - 1:

FOR col FROM 0 TO boardSize - 1:

IF board[row][col] ≠ playerSymbol:

CONTINUE // Bỏ qua ô không thuộc đối thủ

fourCount ← 0

blockingMoves ← [] // Danh sách các nước đi có thể chặn

FOR dir IN directions:

count ← countDirection(row, col, dir[0], dir[1], playerSymbol) + 1

IF count = 4 AND checkOpenEnds(row, col, dir, count) = 1:

potentialBlock ← findBlockingMove(row, col, dir, count)

IF potentialBlock IS VALID:

fourCount ← fourCount + 1

blockingMoves.append(potentialBlock)

IF fourCount ≥ 2:

FOR move IN blockingMoves:

IF move appears more than once in blockingMoves:

RETURN move // Chặn chung tại vị trí có thể phá cả 2 chuỗi

RETURN blockingMoves[0] // Nếu không có chặn chung, trả về nước đầu tiên

RETURN sf::Vector2i(-1, -1) // Không tìm thấy thế 2 chuỗi 4 song song

###### Độ phức tạp

* Không gian
  + O(n²): Quét toàn bộ bàn cờ.
* Thời gian
  + O(n³):
    - Duyệt qua tất cả các ô trên bàn cờ → O(n²).
    - Kiểm tra chuỗi quân liên tiếp trong một hướng → O(n)

##### detectGapPair

###### Cấu trúc dữ liệu

* Biến directions[4][2]: Mảng 2D chứa các hướng kiểm tra (ngang, dọc, chéo xuôi, chéo ngược).
* Biến board: Mảng 2D đại diện cho bàn cờ, mỗi ô có thể chứa ký tự 'X', 'O' hoặc dấu cách ' '.
* Biến playerSymbol: Biến chứa ký hiệu của quân của đối thủ.
* Biến sf::Vector2i: Dùng để lưu tọa độ (row, col) của ô trống cần chặn.

###### Giải thuật

* Vòng lặp ngoài: Duyệt qua tất cả các ô trên bàn cờ.
* Điều kiện board[row][col] != playerSymbol: Bỏ qua ô không phải của đối thủ.
* Vòng lặp trong: Duyệt qua 4 hướng (ngang, dọc, chéo xuôi, chéo ngược).
  + isGapPresent(row, col, dir): Kiểm tra xem có một khoảng cách trống giữa 2 quân đối thủ trong hướng này không.
  + Nếu có, tính toán vị trí của ô trống giữa 2 quân (sử dụng dir để xác định ô trống).
  + isValid(gapRow, gapCol): Kiểm tra ô trống có hợp lệ hay không (nằm trong phạm vi bàn cờ).
  + Nếu ô trống hợp lệ, trả về tọa độ ô trống.

###### Thư viện

* sf::Vector2i: Được sử dụng để lưu trữ và trả về tọa độ của ô trống giữa cặp quân đối thủ.

###### Mả giả(Pseudo code)

sf::Vector2i detectGapPair() {

// Duyệt qua tất cả các ô trên bàn cờ

for (row = 0 to boardSize) {

for (col = 0 to boardSize) {

if (board[row][col] != playerSymbol) continue; // Bỏ qua ô không phải của đối thủ

// Kiểm tra theo 4 hướng

for (dir in directions) {

if (isGapPresent(row, col, dir)) {

// Xác định vị trí ô trống giữa cặp quân

gapRow = row + dir[0];

gapCol = col + dir[1];

if (isValid(gapRow, gapCol) && board[gapRow][gapCol] == ' ') {

return sf::Vector2i(gapRow, gapCol); // Trả về vị trí chặn

}

}

}

}

}

return sf::Vector2i(-1, -1); // Không tìm thấy cặp quân có khoảng cách 1 ô

}

###### Độ phức tạp

* Không gian
  + O(1) — Dùng một số biến tạm và không sử dụng bộ nhớ phụ thuộc vào kích thước bàn cờ.
* Thời gian
  + O(n²) — Cả hai vòng lặp ngoài và trong đều duyệt qua tất cả các ô trên bàn cờ (với kích thước bàn cờ n x n).

##### detectLOrTShape

###### Cấu trúc dữ liệu

* Biến directions[4][2]: Mảng 2D chứa các hướng kiểm tra (ngang, dọc, chéo xuôi, chéo ngược).
* Biến board: Mảng 2D đại diện cho bàn cờ, mỗi ô có thể chứa ký tự 'X', 'O' hoặc dấu cách ' '.
* Biến playerSymbol: Biến chứa ký hiệu của quân đối thủ (X hoặc O).
* Biến sf::Vector2i: Dùng để lưu tọa độ của nước đi chặn.

###### Giải thuật

* Vòng lặp ngoài: Duyệt qua tất cả các ô trên bàn cờ.
* Điều kiện board[row][col] != playerSymbol: Bỏ qua ô không thuộc quân đối thủ.
* Vòng lặp trong: Duyệt qua 4 hướng (ngang, dọc, chéo xuôi, chéo ngược).
  + Tính toán dir1 và dir2: Lựa chọn 2 hướng vuông góc với nhau để kiểm tra (ví dụ, ngang và dọc, ngang và chéo, v.v.).
  + isLShape(row, col, dir1, dir2): Kiểm tra xem có hình "L" hoặc "T" tại vị trí (row, col) trong 2 hướng này không.
  + Đếm số lượng quân trong mỗi hướng: Dùng countDirection để xác định số quân trong mỗi hướng và tính tổng.
  + findBlockingMove(row, col, dir1, count1): Tìm và trả về nước đi chặn trong hướng 1 nếu có.
  + findBlockingMove(row, col, dir2, count2): Tìm và trả về nước đi chặn trong hướng 2 nếu có.
* Nếu không tìm thấy bất kỳ hình dạng "L" hoặc "T" nào, trả về sf::Vector2i(-1, -1).

###### Thư viện

* sf::Vector2i: Lưu trữ và trả về tọa độ của nước đi chặn.

###### Mả giả(Pseudo code)

sf::Vector2i detectLOrTShape() {

// Duyệt qua tất cả các ô trên bàn cờ

for (row = 0 to boardSize) {

for (col = 0 to boardSize) {

if (board[row][col] != playerSymbol) continue; // Bỏ qua ô không phải quân đối thủ

// Duyệt qua 4 hướng và kiểm tra hình L hoặc T

for (i = 0 to 4) {

dir1 = directions[i];

dir2 = directions[(i + 1) % 4]; // Hướng vuông góc với dir1

if (isLShape(row, col, dir1, dir2)) {

// Đếm quân trong 2 hướng

count1 = countDirection(row, col, dir1[0], dir1[1], playerSymbol) + 1;

count2 = countDirection(row, col, dir2[0], dir2[1], playerSymbol) + 1;

// Tìm nước chặn trong các hướng

blockMove = findBlockingMove(row, col, dir1, count1);

if (blockMove.x != -1 && blockMove.y != -1) return blockMove;

blockMove = findBlockingMove(row, col, dir2, count2);

if (blockMove.x != -1 && blockMove.y != -1) return blockMove;

}

}

}

}

return sf::Vector2i(-1, -1); // Không tìm thấy hình L hoặc T

}

###### Độ phức tạp

* Không gian
  + O(1) — Dùng một số biến tạm và không sử dụng bộ nhớ phụ thuộc vào kích thước bàn cờ.
* Thời gian
  + O(n²) — Cả hai vòng lặp ngoài và trong đều duyệt qua tất cả các ô trên bàn cờ (với kích thước bàn cờ n x n).

##### detectOpenTwo

###### Cấu trúc dữ liệu

* Biến directions[4][2]: Mảng 2D chứa các hướng kiểm tra (ngang, dọc, chéo xuôi, chéo ngược).
* Biến board: Mảng 2D đại diện cho bàn cờ, mỗi ô có thể chứa ký tự 'X', 'O' hoặc dấu cách ' '.
* Biến playerSymbol: Biến chứa ký hiệu của quân đối thủ (X hoặc O).
* Biến sf::Vector2i: Dùng để lưu tọa độ của nước đi chặn.

###### Giải thuật

* Vòng lặp ngoài: Duyệt qua tất cả các ô trên bàn cờ.
* Điều kiện board[row][col] != playerSymbol: Bỏ qua ô không thuộc quân đối thủ.
* Vòng lặp trong: Duyệt qua 4 hướng (ngang, dọc, chéo xuôi, chéo ngược).
  + countDirection(row, col, dir[0], dir[1], playerSymbol): Đếm số quân liên tiếp trong hướng dir (bao gồm ô hiện tại).
  + checkOpenEnds(row, col, dir, count): Kiểm tra xem chuỗi 2 quân có bị chặn ở cả hai đầu hay không. Nếu giá trị trả về là 2, có nghĩa là chuỗi này mở ở cả hai đầu.
  + findBlockingMove(row, col, dir, count): Nếu phát hiện chuỗi 2 quân mở, hàm này tìm.
* Nếu không tìm thấy chuỗi 2 quân mở, trả về sf::Vector2i(-1, -1).

###### Thư viện

* sf::Vector2i: Lưu trữ và trả về tọa độ của nước đi chặn.

###### Mả giả(Pseudo code)

sf::Vector2i detectOpenTwo() {

// Duyệt qua tất cả các ô trên bàn cờ

for (row = 0 to boardSize) {

for (col = 0 to boardSize) {

if (board[row][col] != playerSymbol) continue; // Bỏ qua ô không phải quân đối thủ

// Duyệt qua 4 hướng và kiểm tra chuỗi 2 quân mở

for (i = 0 to 4) {

dir = directions[i];

// Đếm quân trong hướng và kiểm tra chuỗi mở

count = countDirection(row, col, dir[0], dir[1], playerSymbol) + 1;

if (count == 2 && checkOpenEnds(row, col, dir, count) == 2) {

// Tìm nước đi chặn

return findBlockingMove(row, col, dir, count);

}

}

}

}

return sf::Vector2i(-1, -1); // Không có chuỗi 2 quân mở

}

###### Độ phức tạp

* Không gian
  + O(1) — Dùng một số biến tạm và không sử dụng bộ nhớ phụ thuộc vào kích thước bàn cờ.
* Thời gian
  + O(n²) — Cả hai vòng lặp ngoài và trong đều duyệt qua tất cả các ô trên bàn cờ (với kích thước bàn cờ n x n).

##### Hàm hỗ trợ bổ sung(findBlockingMove)

###### Cấu trúc dữ liệu

* sf::Vector2i: Dùng để lưu tọa độ của nước đi chặn.
* dir[2]: Mảng chứa hướng di chuyển (x, y) để xác định đầu của chuỗi.
* count: Số lượng quân liên tiếp trong chuỗi (tính từ ô (row, col)).

###### Giải thuật

* Hàm kiểm tra hai vị trí có thể chặn chuỗi quân:
  + Đầu trước: Kiểm tra ô phía trước theo hướng -dir. Nếu ô này hợp lệ (trống), trả về vị trí đó.
  + Đầu sau: Kiểm tra ô phía sau chuỗi quân tại vị trí row + count \* dir[0], col + count \* dir[1]. Nếu ô này hợp lệ (trống), trả về vị trí đó.
* Nếu không có ô hợp lệ nào, trả về sf::Vector2i(-1, -1).

###### Thư viện

* sf::Vector2i: Để lưu trữ và trả về các tọa độ của các nước đi.

###### Mả giả(Pseudo code)

function findBlockingMove(row, col, dir, count):

# Kiểm tra đầu trước (phía -dir)

prevRow = row - dir[0]

prevCol = col - dir[1]

# Kiểm tra nếu vị trí trước hợp lệ và trống

if isValid(prevRow, prevCol) and board[prevRow][prevCol] == ' ':

return (prevCol, prevRow) # Trả về vị trí hợp lệ (col, row)

# Kiểm tra đầu sau (phía +dir)

nextRow = row + count \* dir[0]

nextCol = col + count \* dir[1]

# Kiểm tra nếu vị trí sau hợp lệ và trống

if isValid(nextRow, nextCol) and board[nextRow][nextCol] == ' ':

return (nextCol, nextRow) # Trả về vị trí hợp lệ (col, row)

# Nếu không tìm thấy vị trí hợp lệ, trả về (-1, -1)

return (-1, -1)

###### Độ phức tạp

* Không gian
  + O(1): Dùng các biến tạm thời.
* Thời gian
  + O(1): Các phép kiểm tra đơn giản, không phụ thuộc vào bàn cờ.

##### Hàm hỗ trợ bổ sung(checkOpenEnds)

###### Cấu trúc dữ liệu

* board: Đây là một mảng 2 chiều (hoặc vector 2 chiều) đại diện cho bàn cờ, nơi mỗi ô có thể chứa các ký tự như 'X', 'O' hoặc ' ' (ô trống).
* dir: Mảng 2 phần tử (int[2]) dùng để lưu thông tin về hướng di chuyển (hướng kiểm tra).
* row và col: Chỉ số của ô cần kiểm tra.
* openEnds: Biến đếm số đầu mở (vị trí trống ở hai đầu của chuỗi quân cờ).

###### Giải thuật

* Hàm kiểm tra các đầu của một chuỗi có chiều dài length theo một hướng cho trước (được xác định bởi mảng dir).
* Kiểm tra hai đầu của chuỗi: đầu trước và đầu sau. Nếu một trong hai đầu là ô trống (' '), thì tăng biến đếm openEnds.
* Đầu trước được tính bằng cách trừ các giá trị từ row và col theo hướng đã cho.
* Đầu sau được tính bằng cách cộng các giá trị từ row và col với chiều dài length theo hướng.
* Cuối cùng, hàm trả về số lượng đầu mở (từ 0 đến 2).

###### Thư viện

* SFML: hỗ trợ giao diện.

###### Mả giả(Pseudo code)

function checkOpenEnds(row, col, dir, length):

openEnds = 0

# Kiểm tra đầu trước (phía -dir)

prevRow = row - dir[0]

prevCol = col - dir[1]

if isValid(prevRow, prevCol) and board[prevRow][prevCol] == ' ':

openEnds += 1

# Kiểm tra đầu sau (phía +dir)

nextRow = row + length \* dir[0]

nextCol = col + length \* dir[1]

if isValid(nextRow, nextCol) and board[nextRow][nextCol] == ' ':

openEnds += 1

return openEnds

###### Độ phức tạp

* Không gianả
  + O(1): Dùng các biến tạm thời.
* Thời gian
  + O(1): Các phép kiểm tra đơn giản, không phụ thuộc vào bàn cờ.

## hardBotMove

### Cấu trúc dữ liệu

* sf::Vector2i move: Một đối tượng lưu trữ vị trí nước đi của bot dưới dạng cặp tọa độ (x, y).
* int row, col: Các biến lưu trữ vị trí hàng và cột của nước đi mà bot sẽ chọn.
* currentI, currentJ: Các chỉ số hàng và cột hiện tại mà bot dự định di chuyển tới.
* nextBound: Một cấu trúc dữ liệu (thường là vector hoặc map) chứa các giới hạn (bound) của các nước đi tiềm năng của bot. Mỗi phần tử có dạng (vị trí, giá trị).
* boardValue: Một giá trị đại diện cho trạng thái hiện tại của bàn cờ (có thể là điểm số hoặc trạng thái đánh giá của bàn cờ).
* depth: Độ sâu tìm kiếm của thuật toán Alpha-Beta Pruning.
* alphaBetaPruning: Hàm thực hiện thuật toán Alpha-Beta Pruning để tối ưu hóa việc lựa chọn nước đi.

### Giải thuật

* Alpha-Beta Pruning:
  + Hàm gọi alphaBetaPruning(depth, boardValue, nextBound, INT\_MIN, INT\_MAX, true) để thực hiện tìm kiếm tối ưu với thuật toán Alpha-Beta Pruning.
  + Thuật toán này giúp bot đánh giá tất cả các nước đi tiềm năng mà không cần phải duyệt qua toàn bộ cây tìm kiếm. Dựa trên giá trị đánh giá của các nước đi, Alpha-Beta Pruning giúp cắt tỉa những nhánh không cần thiết, từ đó rút ngắn thời gian tìm kiếm.
* Kiểm tra tính hợp lệ của nước đi:
  + Sau khi thực hiện Alpha-Beta Pruning, bot kiểm tra tính hợp lệ của nước đi tại vị trí currentI, currentJ bằng cách gọi hàm isValidHard(currentI, currentJ, true). Nếu hợp lệ, bot sẽ chọn vị trí này làm nước đi tiếp theo và cập nhật giá trị của nextBound.
* Tìm kiếm nước đi thay thế:
  + Nếu nước đi tại currentI, currentJ không hợp lệ, bot sẽ tìm các vị trí thay thế có thể di chuyển đến bằng cách sắp xếp các phần tử trong nextBound theo giá trị giảm dần. Sau đó, bot chọn vị trí có giá trị cao nhất (thường là nước đi tối ưu nhất).
* Cập nhật lại các giá trị vị trí của bot:
  + Sau khi chọn nước đi, bot cập nhật lại các giá trị currentI và currentJ để ghi nhớ nước đi vừa chọn.
* Trả về nước đi:
  + Cuối cùng, hàm trả về vị trí (col, row) của nước đi, được lưu trong đối tượng move.

### Thư viện

* sf::Vector2i: Được sử dụng để lưu trữ và trả về vị trí nước đi dưới dạng tọa độ (x, y).
* vector và pair: Được sử dụng để lưu trữ và sắp xếp các giới hạn (bounds) cho nước đi của bot.
* sort: Dùng để sắp xếp các giá trị của nextBound theo giá trị giảm dần.

### Mả giả(Pseudo code)

Function hardBotMove():

Initialize move, row, col

// Perform Alpha-Beta Pruning

Call alphaBetaPruning(depth, boardValue, nextBound, INT\_MIN, INT\_MAX, true)

// Check if the move at currentI, currentJ is valid

If isValidHard(currentI, currentJ, true):

row = currentI

col = currentJ

Call updateBound(row, col, nextBound)

Else:

Call updateBound(currentI, currentJ, nextBound)

// Sort the possible moves by value in descending order

Sort nextBound by second value in descending order

// Choose the move with the highest value

(row, col) = nextBound[0].first

currentI = row

currentJ = col

// Return the chosen move as (col, row)

return (col, row)

### Độ phức tạp

* Không gian
  + nextBound: Cấu trúc nextBound lưu trữ tất cả các nước đi tiềm năng, do đó độ phức tạp không gian của hàm là O(m).
  + alphaBetaPruning: Tùy thuộc vào cách triển khai, nhưng nhìn chung, nó sẽ sử dụng không gian để lưu trữ cây tìm kiếm. Độ phức tạp không gian của Alpha-Beta Pruning thường là O(b \* d).
  + Tổng độ phức tạp không gian là O(m + b \* d).
* Thời gian
  + Alpha-Beta Pruning: Độ phức tạp của thuật toán Alpha-Beta Pruning là O(b^d), trong đó b là số nhánh tại mỗi cấp độ và d là độ sâu tìm kiếm. Tuy nhiên, nhờ pruning, độ phức tạp thực tế sẽ thấp hơn so với tìm kiếm cây đầy đủ.
  + Sắp xếp nextBound: Việc sắp xếp danh sách các nước đi (nextBound) có độ phức tạp O(m log m), trong đó m là số lượng các nước đi tiềm năng.
  + Tổng độ phức tạp thời gian là O(b^d + m log m).

### Các hàm hỗ trợ cho HardBotMove

#### create\_pattern\_dict

##### Cấu trúc dữ liệu

* map<vector<int>, int> patternDict:
  + patternDict là một map (từ điển), trong đó khóa là một vector<int> (mảng các số nguyên), và giá trị là một số nguyên int.
  + Mỗi khóa biểu diễn một mẫu (pattern) của các ký tự trên bàn cờ, và giá trị tương ứng là điểm số cho mẫu đó.
* x: Biến dùng để điều chỉnh các giá trị trong các mẫu. x có thể là -1 hoặc 1, dùng để đại diện cho các mẫu với 'X' và 'O'.

##### Giải thuật

* Duyệt qua giá trị của x:
  + Biến x bắt đầu từ -1 và tăng dần lên 1, theo bước nhảy 2, tức là sẽ lặp qua các giá trị x = -1 và x = 1. Giá trị x đại diện cho dấu của người chơi (có thể là 'X' hoặc 'O').
* Thêm các mẫu vào patternDict:
  + Hàm liên tục tạo ra các mẫu (patterns) có chiều dài từ 5 đến 7 phần tử. Mỗi mẫu được lưu trữ trong patternDict với điểm số tương ứng.
  + Các mẫu được tạo ra bằng cách kết hợp các giá trị của x (hoặc -x) và 0, với các mẫu có chiều dài khác nhau, như {x, x, x, x, x}, {0, x, x, x, 0}, {y, x, x, x, x, y}, vv.
  + Mỗi mẫu có một giá trị điểm khác nhau, dao động từ 1000000 đến 10, tùy thuộc vào mức độ quan trọng của mẫu đó trong trò chơi.
* Chạy qua tất cả các mẫu:
  + Các mẫu được tạo ra từ việc kết hợp x và y = -x (nếu x = 1, thì y = -1; nếu x = -1, thì y = 1), và mỗi mẫu này có một giá trị điểm riêng được gán vào map.
* Trả về patternDict:
  + Sau khi hoàn tất việc thêm tất cả các mẫu và giá trị tương ứng vào patternDict, hàm trả về patternDict này.

##### Thư viện

* map<vector<int>, int>: Cấu trúc dữ liệu chính để lưu trữ các mẫu và giá trị của chúng.
* vector<int>: Được sử dụng để đại diện cho một mẫu cụ thể với các giá trị kiểu int (tương ứng với các ký tự như 'X', 'O', hoặc 0).

##### Mả giả(Pseudo code)

Function create\_pattern\_dict():

Initialize patternDict as an empty map

Initialize x = -1

While x < 2:

y = -x

// Add predefined patterns to patternDict with appropriate values

patternDict[{x, x, x, x, x}] = 1000000 \* x

patternDict[{0, x, x, x, x, 0}] = 100000 \* x

patternDict[{0, x, x, x, 0, x, 0}] = 100000 \* x

// (Add more patterns...)

x += 2

Return patternDict

##### Độ phức tạp

* Không gian
  + patternDict chứa các mẫu có chiều dài khác nhau và giá trị tương ứng, số lượng mẫu là cố định (khoảng 30 mẫu cho mỗi giá trị x), vì vậy độ phức tạp không gian của hàm là O(1) (vì không có sự thay đổi đáng kể theo kích thước bàn cờ hoặc các yếu tố khác).
* Thời gian
  + Vòng lặp ngoài (while x < 2): Lặp qua 2 giá trị của x (tức là 2 lần), mỗi lần là x = -1 và x = 1.
  + Thêm các mẫu vào patternDict: Mỗi lần lặp sẽ thêm khoảng 30 mẫu vào patternDict. Tổng số mẫu được thêm vào trong mỗi vòng lặp là cố định, khoảng 30 mẫu.
  + Vì vậy, tổng độ phức tạp thời gian là O(30) = O(1), do số mẫu được tạo ra là cố định và không phụ thuộc vào kích thước của bàn cờ.

#### isValidHard

##### Cấu trúc dữ liệu

* IntBoard[i][j]: Biến này đại diện cho bàn cờ dưới dạng mảng hai chiều (mảng số nguyên) chứa các giá trị biểu thị trạng thái các ô trên bàn cờ.
  + Nếu IntBoard[i][j] == 0, ô này trống.
  + Nếu IntBoard[i][j] != 0, ô này đã được đánh dấu bởi người chơi hoặc bot.
* boardSize: Biến này đại diện cho kích thước của bàn cờ (kích thước n x n).

##### Giải thuật

* Kiểm tra chỉ số bàn cờ hợp lệ:
  + Kiểm tra xem chỉ số hàng (i) và cột (j) có nằm trong phạm vi bàn cờ hay không: i và j phải lớn hơn hoặc bằng 0 và nhỏ hơn boardSize. Nếu không, trả về false.
* Kiểm tra tính hợp lệ của ô:
  + Nếu state là true và ô IntBoard[i][j] không phải là 0 (tức là ô đã được đánh dấu bởi người chơi hoặc bot), hàm trả về false (ô không hợp lệ để đánh).
* Trả về kết quả:
  + Nếu không có vấn đề gì, hàm trả về true, nghĩa là ô (i, j) hợp lệ cho người chơi hoặc bot thực hiện nước đi.

##### Thư viện

##### Mả giả(Pseudo code)

Function isValidHard(i, j, state):

// Kiểm tra chỉ số hàng và cột hợp lệ

If i < 0 or i >= boardSize or j < 0 or j >= boardSize:

Return false

// Kiểm tra nếu state là true và ô đã được đánh

If state is true and IntBoard[i][j] != 0:

Return false

// Nếu không có vấn đề gì, ô hợp lệ

Return true

##### Độ phức tạp

* Không gian
  + Hàm chỉ sử dụng các biến số nguyên và không thay đổi kích thước của dữ liệu, vì vậy độ phức tạp không gian là O(1).
* Thời gian
  + Hàm chỉ thực hiện một số phép toán điều kiện và kiểm tra chỉ số, vì vậy độ phức tạp thời gian là O(1).

#### setState

##### Cấu trúc dữ liệu

* IntBoard[x][y]: Đây là mảng hai chiều (board) chứa các giá trị trạng thái của các ô trên bàn cờ. Mỗi ô có giá trị là:
  + -1: đại diện cho người chơi thứ nhất (ví dụ: 'X').
  + 1: đại diện cho người chơi thứ hai (ví dụ: 'O').
  + 0: đại diện cho ô trống.
* lastPlayed: Biến lưu trạng thái (giá trị) của người chơi đã thực hiện nước đi cuối cùng. Có thể có giá trị là -1, 0 hoặc 1.

##### Giải thuật

* Kiểm tra hợp lệ của giá trị state:
  + Sử dụng assert để đảm bảo rằng state chỉ có thể nhận một trong ba giá trị: -1, 0 hoặc 1. Nếu state không phải là một trong các giá trị này, chương trình sẽ bị dừng tại dòng assert.
* Cập nhật bảng cờ:
  + Gán giá trị state vào ô IntBoard[x][y] để cập nhật trạng thái của ô đó.
* Cập nhật trạng thái người chơi đã thực hiện nước đi:
  + Cập nhật biến lastPlayed với giá trị state để lưu lại trạng thái của người chơi vừa thực hiện nước đi.

##### Thư viện

* assert: Dùng để kiểm tra tính hợp lệ của giá trị state. Nếu giá trị không hợp lệ, chương trình sẽ dừng lại.

##### Mả giả(Pseudo code)

Function setState(x, y, state):

// Kiểm tra tính hợp lệ của state

Assert state == -1 or state == 0 or state == 1

// Cập nhật giá trị trong bảng

IntBoard[x][y] = state

// Cập nhật trạng thái người chơi đã thực hiện nước đi

lastPlayed = state

##### Độ phức tạp

* Không gian
  + Hàm chỉ sử dụng các biến tạm thời và không thay đổi cấu trúc dữ liệu lớn, vì vậy độ phức tạp không gian của hàm là O(1).
* Thời gian
  + Hàm thực hiện các phép gán và kiểm tra đơn giản, vì vậy độ phức tạp thời gian của hàm là O(1).

#### countDirection

##### Cấu trúc dữ liệu

* IntBoard[x][y]: Mảng hai chiều lưu trữ trạng thái của các ô trên bàn cờ. Mỗi ô có giá trị:
  + -1: đại diện cho người chơi thứ nhất.
  + 1: đại diện cho người chơi thứ hai.
  + 0: ô trống.
* boardSize: Biến đại diện cho kích thước bàn cờ (n x n).

##### Giải thuật

* Khởi tạo biến đếm: count = 0 để đếm số ô liên tiếp có trạng thái giống với ô bắt đầu (state).
* Duyệt theo hướng:
  + Sử dụng các bước (xdir, ydir) để duyệt qua các ô trong một hướng cụ thể (theo phương ngang, dọc hoặc chéo).
  + xdir và ydir xác định hướng di chuyển:
  + Nếu xdir = 1 và ydir = 0, di chuyển sang phải.
  + Nếu xdir = 0 và ydir = 1, di chuyển xuống dưới.
  + Nếu xdir = 1 và ydir = 1, di chuyển theo chéo từ trên trái xuống dưới phải.
  + Nếu xdir = 1 và ydir = -1, di chuyển theo chéo từ dưới trái lên trên phải.
* Kiểm tra biên:
  + Kiểm tra xem ô mới có nằm trong phạm vi bàn cờ không (newX, newY phải nằm trong [0, boardSize-1]).
* Kiểm tra giá trị ô:
  + Nếu ô tại (newX, newY) có giá trị bằng state, tăng giá trị của count.
  + Nếu ô không có giá trị state, dừng vòng lặp.
* Trả về kết quả:
  + Trả về giá trị count, là số lượng ô liên tiếp có trạng thái giống với state trong hướng cho trước.

##### Thư viện

##### Mả giả(Pseudo code)

Function countDirection(x, y, xdir, ydir, state):

count = 0

For step = 1 to 4:

newX = x + xdir \* step

newY = y + ydir \* step

// Kiểm tra biên của bảng

If newX < 0 or newX >= boardSize or newY < 0 or newY >= boardSize:

Break

// Kiểm tra trạng thái ô

If IntBoard[newX][newY] == state:

count = count + 1

Else:

Break

Return count

##### Độ phức tạp

* Không gian
  + Hàm chỉ sử dụng các biến tạm thời, không thay đổi cấu trúc dữ liệu lớn, vì vậy độ phức tạp không gian của hàm là O(1).
* Thời gian
  + Vòng lặp duyệt từ 1 đến 4 lần (do điều kiện step <= 4), vì vậy độ phức tạp thời gian của hàm là O(1).

#### isFive

##### Cấu trúc dữ liệu

* directions[4][2][2]: Mảng ba chiều dùng để lưu trữ các hướng kiểm tra (ngang, dọc, chéo). Mỗi phần tử của mảng lưu hai hướng đối xứng:
  + Hướng dọc: từ trên xuống dưới và từ dưới lên trên.
  + Hướng ngang: từ trái qua phải và từ phải qua trái.
  + Hướng chéo lên: từ dưới trái lên trên phải và từ trên phải xuống dưới trái.
  + Hướng chéo xuống: từ trên trái xuống dưới phải và từ dưới phải lên trên trái.
* countDirection(i, j, xdir, ydir, state): Hàm đã được phân tích trước đó, đếm số lượng ô liên tiếp có giá trị state theo hướng (xdir, ydir) từ ô (i, j).

##### Giải thuật

* Khởi tạo các hướng kiểm tra:
  + Mảng directions chứa 4 hướng kiểm tra (ngang, dọc, và hai hướng chéo).
* Duyệt qua các hướng:
  + Với mỗi hướng, kiểm tra cả hai phía đối xứng (từ điểm hiện tại (i, j) đi về hai hướng).
* Đếm số lượng ô liên tiếp:
  + Sử dụng hàm countDirection để đếm số ô liên tiếp có giá trị state từ vị trí (i, j) trong từng hướng.
  + Cộng dồn vào axis\_count (đếm bao gồm cả ô (i, j)).
  + Nếu axis\_count đạt từ 5 ô trở lên, trả về true vì đã tìm được một chuỗi 5 ô liên tiếp của người chơi.
* Trả về kết quả:
  + Nếu không tìm được chuỗi 5 ô liên tiếp trong bất kỳ hướng nào, trả về false.

##### Thư viện

##### Mả giả(Pseudo code)

Function isFive(i, j, state):

directions = [

{{-1, 0}, {1, 0}}, // Hướng dọc

{{0, -1}, {0, 1}}, // Hướng ngang

{{-1, 1}, {1, -1}}, // Hướng chéo lên

{{-1, -1}, {1, 1}} // Hướng chéo xuống

]

For each direction in directions:

axis\_count = 1 // Đếm ô hiện tại (i, j)

For each direction part (xdir, ydir):

count = countDirection(i, j, xdir, ydir, state)

axis\_count += count

If axis\_count >= 5:

Return true

Return false

##### Độ phức tạp

* Không gian
  + Hàm sử dụng mảng directions có kích thước cố định và không sử dụng thêm bộ nhớ ngoài. Vì vậy, độ phức tạp không gian của hàm là O(1).
* Thời gian
  + Mỗi hướng kiểm tra có thể thực hiện tối đa 2 lần gọi hàm countDirection, mỗi hàm countDirection có độ phức tạp O(1). Vì vậy, độ phức tạp thời gian của hàm isFive là O(1), vì có tối đa 4 hướng và mỗi hướng chỉ cần kiểm tra 2 lần.

#### childNodes

##### Cấu trúc dữ liệu

* map<pair<int, int>, int>: bound là một map chứa các cặp tọa độ (x, y) và một giá trị integer tương ứng (có thể là điểm số, độ ưu tiên của các vị trí).
* vector<pair<pair<int, int>, int>> sortedNodes: Đây là vector chứa các cặp (vị trí, giá trị), được tạo ra từ bound để sắp xếp.
* vector<pair<int, int>> positions: Đây là vector lưu trữ các vị trí (x, y) đã được sắp xếp theo giá trị của chúng.

##### Giải thuật

* Chuyển đổi bound sang vector:
  + Đầu tiên, bound (map) được chuyển thành sortedNodes (vector) để có thể sắp xếp các phần tử theo giá trị.
* Sắp xếp theo giá trị:
  + Vector sortedNodes được sắp xếp giảm dần theo giá trị thứ hai của cặp (theo thứ tự giảm dần của giá trị trong bound).
* Trích xuất các vị trí:
  + Sau khi sắp xếp, các vị trí (x, y) (tức là phần tử đầu tiên của mỗi cặp trong sortedNodes) được thêm vào vector positions.
* Trả về danh sách vị trí:
  + Vector positions, chứa các vị trí đã được sắp xếp, được trả về.

##### Thư viện

* vector: Dùng để lưu trữ và thao tác với danh sách các vị trí và các cặp giá trị.
* map: Dùng để lưu trữ các cặp (vị trí, giá trị) với khả năng tự động sắp xếp theo khóa (ở đây là cặp (x, y)).
* sort: Sử dụng để sắp xếp vector theo giá trị giảm dần.

##### Mả giả(Pseudo code)

Function childNodes(bound):

Convert bound (map) to sortedNodes (vector of pairs)

Sort sortedNodes by the second element in descending order

Create an empty vector positions

For each node in sortedNodes:

Extract the position from node and add it to positions

Return positions

##### Độ phức tạp

* Không gian
  + Không gian sử dụng là O(n) để lưu trữ các cặp trong sortedNodes và các vị trí trong positions, với n là số phần tử trong bound.
* Thời gian
  + Chuyển đổi từ map sang vector: Thao tác này mất O(n), với n là số phần tử trong bound.
  + Sắp xếp vector: Sắp xếp vector sortedNodes có độ phức tạp O(n log n).
  + Duyệt qua sortedNodes để trích xuất các vị trí: Thao tác này có độ phức tạp O(n).
  + Tổng độ phức tạp thời gian là O(n log n), do bước sắp xếp chiếm phần lớn thời gian.

#### vupdateBound

##### Cấu trúc dữ liệu

* map<pair<int, int>, int>: bound là một map lưu trữ các cặp tọa độ (x, y) với giá trị integer tương ứng. Đây là cấu trúc để theo dõi các vị trí hợp lệ trong trò chơi mà bot có thể xem xét.
* pair<int, int> played: Biến này lưu tọa độ của ô vừa được chơi (new\_i, new\_j).
* vector<pair<int, int>> directions: Một danh sách các hướng (ngang, dọc và chéo) để bot có thể kiểm tra các ô xung quanh vị trí vừa được chơi.

##### Giải thuật

* Kiểm tra và xóa vị trí đã chơi khỏi bound:
  + Tạo cặp tọa độ played cho vị trí vừa được chơi.
  + Nếu vị trí này đã có trong bound, xóa nó khỏi map.
* Kiểm tra các ô xung quanh:
  + Lặp qua tất cả các hướng (ngang, dọc, chéo) được định nghĩa trong directions.
  + Với mỗi hướng, tính toán tọa độ của ô liền kề (new\_row, new\_col).
* Kiểm tra tính hợp lệ của ô xung quanh:
  + Dùng hàm isValidHard để kiểm tra xem ô liền kề có hợp lệ và chưa có trong bound không.
  + Nếu hợp lệ và chưa có trong bound, thêm ô vào bound với giá trị ban đầu là 0.

##### Thư viện

* map: Dùng để lưu trữ các vị trí (x, y) và giá trị đi kèm.
* vector: Dùng để lưu trữ danh sách các hướng mà bot cần kiểm tra.
* pair: Dùng để lưu trữ các tọa độ (x, y).

##### Mả giả(Pseudo code)

Function updateBound(new\_i, new\_j, bound):

Create a pair (new\_i, new\_j) for the played position

If played position exists in bound:

Remove played position from bound

Define directions as 8 possible directions (up, down, left, right, diagonals)

For each direction in directions:

Calculate new row and column based on direction

If (new\_row, new\_col) is valid and not in bound:

Add (new\_row, new\_col) to bound with value 0

##### Độ phức tạp

* Không gian
  + Không gian sử dụng là O(n) để lưu trữ các vị trí trong bound, với n là số phần tử trong map.
* Thời gian
  + Lặp qua các hướng: Có 8 hướng cần kiểm tra (4 hướng chính và 4 hướng chéo), nên vòng lặp này chạy 8 lần.
  + Kiểm tra tính hợp lệ và xóa/insert trong map: Mỗi lần kiểm tra tính hợp lệ hoặc xóa/insert trong bound có độ phức tạp O(log n) do map sử dụng cây nhị phân (có n là số phần tử trong bound).
  + Tổng độ phức tạp thời gian là O(8 log n) ≈ O(log n).

#### countPattern

##### Cấu trúc dữ liệu

* vector<int> pattern: Mẫu mẫu cần tìm trên bàn cờ, chứa các giá trị trạng thái ô (ví dụ: 1 cho người chơi, -1 cho bot).
* vector<pair<int, int>> directions: Lưu trữ các hướng tìm kiếm mẫu trên bàn cờ, gồm 4 hướng: ngang, dọc, chéo lên và chéo xuống.
* map<pair<int, int>, int> bound: Bảng lưu trữ các ô đã được kiểm tra, với điểm số được cập nhật cho mỗi ô dựa trên mẫu đã tìm thấy.
* int score: Điểm số dành cho mẫu được tìm thấy.
* int flag: Cờ chỉ định chiều hướng của mẫu (dành cho việc cập nhật giá trị trong bảng bound).

##### Giải thuật

* Khởi tạo: Tạo danh sách hướng tìm kiếm mẫu (directions), gồm 4 hướng căn bản: dọc, ngang, chéo lên và chéo xuống.
* Tính bước lùi: Tính toán steps\_back cho mỗi hướng tìm kiếm, xác định phạm vi cần kiểm tra dựa trên vị trí (i\_0, j\_0).
* Tìm mẫu: Duyệt qua từng hướng tìm kiếm, bắt đầu từ điểm (i\_0, j\_0), kiểm tra xem có khớp với mẫu không. Mỗi khi tìm thấy mẫu, tăng biến count và cập nhật bảng bound nếu có ô hợp lệ.
* Cập nhật bảng bound: Nếu một mẫu khớp được tìm thấy, các ô trong mẫu sẽ được thêm vào bảng bound và cập nhật giá trị của chúng bằng cách cộng thêm flag \* score.

##### Thư viện

* vector: Dùng để lưu trữ các hướng tìm kiếm và các vị trí ô trên bàn cờ.
* map: Sử dụng để theo dõi điểm số của các ô trong bàn cờ dựa trên các mẫu đã được tìm thấy.

##### Mả giả(Pseudo code)

Function countPattern(i\_0, j\_0, pattern, score, bound, flag):

Initialize directions = [(1, 0), (1, 1), (0, 1), (-1, 1)] // Các hướng tìm mẫu

Initialize count = 0 // Biến đếm số lượng mẫu

For each direction in directions:

Initialize steps\_back = 0 // Số bước cần lùi lại để kiểm tra mẫu

// Tính toán bước lùi cho các hướng dọc, ngang, chéo

If direction is horizontal or vertical:

steps\_back = Calculate steps based on position (i\_0, j\_0)

Else if direction is diagonal:

steps\_back = Calculate steps based on position (i\_0, j\_0)

Initialize i\_start = i\_0 - steps\_back \* direction[1]

Initialize j\_start = j\_0 - steps\_back \* direction[0]

z = 0 // Biến đếm bước đi

While z <= steps\_back:

Initialize i\_new = i\_start + z \* direction[1]

Initialize j\_new = j\_start + z \* direction[0]

Initialize index = 0 // Biến đếm chỉ số mẫu

Initialize remember = [] // Danh sách nhớ các vị trí hợp lệ

While index < length of pattern AND (i\_new, j\_new) is valid AND (i\_new, j\_new) matches pattern[index]:

If position (i\_new, j\_new) is valid to play:

Add (i\_new, j\_new) to remember list

Move to next position in the current direction

Increment index by 1

If index equals the length of pattern:

Increment count by 1

For each position in remember:

If position is not in bound:

Add position to bound with initial value 0

Update bound[position] by adding (flag \* score)

Skip the length of the pattern by setting z += index

Else:

Increment z by 1

Return count

##### Độ phức tạp

* Không gian
  + Không gian cần thiết là O(boardSize^2) để lưu trữ bảng IntBoard và bảng bound.
* Thời gian
  + Độ phức tạp chính của hàm này là O(boardSize^2), do phải kiểm tra tất cả các ô trên bàn cờ và duyệt qua các hướng để tìm mẫu. Mỗi hướng có thể kiểm tra tối đa 5 ô (do steps\_back không vượt quá 5).
  + Cộng thêm thời gian sắp xếp và cập nhật bảng bound, độ phức tạp tổng thể vẫn duy trì O(boardSize^2).

#### evaluate

##### Cấu trúc dữ liệu

* map<pair<int, int>, int> &bound: Đây là một bản đồ (map) lưu trữ các vị trí trên bàn cờ và giá trị tương ứng (được cập nhật trong các lượt đi). Đây là cấu trúc dữ liệu cho phép truy xuất nhanh các vị trí trên bàn cờ.
* IntBoard: Mảng 2 chiều đại diện cho bàn cờ, lưu trạng thái của các ô (0 = trống, 1 = người chơi, -1 = đối thủ).
* patternDict: Bản đồ (map) với các mẫu (patterns) và điểm số (score) tương ứng. Các mẫu này được sử dụng để kiểm tra số lượng và điểm số của các mẫu có trong bàn cờ.

##### Giải thuật

* Mục tiêu: Đánh giá giá trị của một lượt đi trên bàn cờ.
  + value\_before: Tính điểm của các mẫu trước khi thực hiện lượt đi.
  + value\_after: Tính điểm của các mẫu sau khi thực hiện lượt đi.
  + So sánh value\_before và value\_after để tính giá trị của lượt đi.
  + countPattern được gọi để đếm số lần xuất hiện của các mẫu trên bàn cờ, sử dụng các giá trị điểm có trong patternDict.
* Quá trình:
  + Lặp qua tất cả các mẫu trong patternDict.
  + Tính số lượng mẫu trước khi thay đổi bàn cờ (count\_before).
  + Cập nhật bàn cờ với lượt đi, tính số lượng mẫu sau khi thay đổi bàn cờ (count\_after).
  + Khôi phục lại trạng thái bàn cờ.
  + Tính giá trị thay đổi của lượt đi bằng value\_after - value\_before, cộng thêm giá trị ban đầu của bàn cờ.

##### Thư viện

* vector: Dùng để lưu trữ các hướng tìm kiếm và các vị trí ô trên bàn cờ.
* map: Sử dụng để theo dõi điểm số của các ô trong bàn cờ dựa trên các mẫu đã được tìm thấy.
* cmath: Cung cấp hàm abs() để tính giá trị tuyệt đối, sử dụng trong việc tính điểm và đánh giá bàn cờ.

##### Mả giả(Pseudo code)

function evaluate(new\_i, new\_j, board\_value, turn, bound):

value\_before = 0

value\_after = 0

for each pattern in patternDict:

patternVector = pattern.first

score = pattern.second

count\_before = countPattern(new\_i, new\_j, patternVector, absolute(score), bound, -1)

value\_before += count\_before \* score

IntBoard[new\_i][new\_j] = turn

count\_after = countPattern(new\_i, new\_j, patternVector, absolute(score), bound, 1)

value\_after += count\_after \* score

IntBoard[new\_i][new\_j] = 0

final\_value = board\_value + value\_after - value\_before

return final\_value

##### Độ phức tạp

* Không gian
  + Không gian O(n^2): Bàn cờ (IntBoard) có kích thước n x n (với n là kích thước bàn cờ). Bản đồ bound cũng lưu trữ các vị trí trên bàn cờ, cùng với các mẫu và điểm số trong patternDict có thể sử dụng bộ nhớ O(k) với k là số lượng mẫu.
* Thời gian
  + O(k \* m \* n): Trong đó:
    - k là số lượng mẫu trong patternDict.
    - m là chiều dài của mỗi mẫu.
    - n là số lượng ô trên bàn cờ.
    - Mỗi lượt gọi countPattern có độ phức tạp O(n) (vì kiểm tra qua từng ô), và việc lặp qua các mẫu trong patternDict tạo ra độ phức tạp tổng cộng O(k \* m \* n).
    - Các thao tác cập nhật và khôi phục bàn cờ chiếm O(1).

#### initZobrist

##### Cấu trúc dữ liệu

* zobristTable: Đây là một bảng 3 chiều với kích thước boardSize x boardSize x 2. Mỗi ô lưu trữ một cặp giá trị uint64\_t, đại diện cho mã hóa Zobrist cho một ô trên bàn cờ. Các giá trị này được sử dụng trong thuật toán Zobrist hashing để theo dõi trạng thái bàn cờ trong các trò chơi như cờ vây hoặc caro.
  + boardSize: Kích thước bàn cờ (ví dụ, 15x15 hoặc 19x19).
  + Mảng 3 chiều zobristTable[i][j][0] và zobristTable[i][j][1] chứa các giá trị ngẫu nhiên cho hai người chơi hoặc hai trạng thái có thể của mỗi ô (ví dụ, người chơi X và O).

##### Giải thuật

* Khởi tạo bảng Zobrist: Mỗi ô trên bàn cờ được gán hai giá trị ngẫu nhiên 64-bit, được sinh ra từ việc kết hợp hai giá trị rand(). Cơ chế này giúp mã hóa trạng thái của bàn cờ thành một giá trị băm duy nhất, từ đó hỗ trợ việc kiểm tra các trạng thái đã gặp trong quá trình tìm kiếm hoặc khi so sánh các trạng thái.

##### Thư viện

* vector: Cấu trúc dữ liệu động của C++ dùng để lưu trữ bảng Zobrist.
* rand(): Hàm sinh số ngẫu nhiên chuẩn của C++, tạo ra các giá trị ngẫu nhiên.

##### Mả giả(Pseudo code)

function initZobrist():

// Khởi tạo bảng Zobrist rỗng có kích thước [boardSize][boardSize][2]

Create an empty 3D list `zobristTable` with dimensions [boardSize][boardSize][2]

// Duyệt qua từng ô trên bàn cờ

for i from 0 to boardSize - 1:

for j from 0 to boardSize - 1:

// Sinh hai giá trị ngẫu nhiên 64-bit cho mỗi ô

zobristTable[i][j][0] = generateRandom64BitValue()

zobristTable[i][j][1] = generateRandom64BitValue()

end function

function generateRandom64BitValue():

// Kết hợp hai giá trị ngẫu nhiên 32-bit thành một giá trị 64-bit

return (random32Bit() << 32) | random32Bit()

function random32Bit():

// Sinh một số ngẫu nhiên 32-bit

return random() // random() trả về giá trị 32-bit ngẫu nhiên

##### Độ phức tạp

* Không gian
  + O(boardSize^2): Bảng Zobrist yêu cầu không gian lưu trữ cho một bảng 3 chiều với kích thước boardSize x boardSize x 2.
* Thời gian
  + O(boardSize^2): Hai vòng lặp lồng nhau chạy qua từng ô trên bàn cờ, mỗi lần khởi tạo hai giá trị ngẫu nhiên. Thời gian thực hiện tỉ lệ thuận với tổng số ô trên bàn cờ.

#### updateTTable

##### Cấu trúc dữ liệu

* TTable: Là một bảng băm (hash table) lưu trữ thông tin về các trạng thái của bàn cờ. Mỗi phần tử trong bảng lưu trữ:
  + hash: Là giá trị băm của trạng thái bàn cờ, loại uint64\_t, đại diện cho một trạng thái duy nhất.
  + pair<int, int>: Mỗi trạng thái được lưu trữ dưới dạng một cặp giá trị (score, depth):
  + score: Điểm đánh giá của trạng thái đó, kiểu int.
  + depth: Độ sâu của trạng thái trong thuật toán tìm kiếm, kiểu int.

##### Giải thuật

* Cập nhật TTable: Khi hàm này được gọi, bảng băm TTable được cập nhật với một giá trị băm mới (hash) và một cặp giá trị (score, depth) tương ứng với trạng thái bàn cờ đó.

##### Thư viện

* map: Là cấu trúc dữ liệu của C++ dùng để tạo bảng băm, với khóa là uint64\_t (hash của trạng thái) và giá trị là một pair<int, int> (chứa điểm và độ sâu).
* pair: Cấu trúc dữ liệu của C++ để lưu trữ hai giá trị có thể khác kiểu dữ liệu với nhau, trong trường hợp này là score và depth.

##### Mả giả(Pseudo code)

function updateTTable(TTable, hash, score, depth):

// Cập nhật TTable với giá trị băm mới và thông tin điểm, độ sâu

TTable[hash] = (score, depth)

end function

##### Độ phức tạp

* Không gian
  + O(N): Không gian yêu cầu cho bảng băm TTable tỉ lệ thuận với số lượng trạng thái cần lưu trữ. Nếu có N trạng thái khác nhau được lưu trữ trong bảng, không gian là O(N), với mỗi phần tử có kích thước tương ứng với một pair<int, int>.
* Thời gian
  + O(1) trung bình: Vì map trong C++ thường sử dụng cấu trúc cây đỏ-đen, độ phức tạp của phép cập nhật (insert/update) là O(log N) trong trường hợp tệ nhất. Tuy nhiên, trong thực tế, với bảng băm có độ phân bổ tốt, nó có thể đạt được O(1) trong hầu hết các trường hợp.

#### checkResult

##### Cấu trúc dữ liệu

* optional<int>: Đây là kiểu trả về std::optional<int> trong C++, cho phép hàm trả về một giá trị có thể có hoặc không. Khi hàm không có kết quả (chưa xác định), nó trả về nullopt, còn khi có kết quả, trả về giá trị kiểu int.
  + lastPlayed: Biến lưu trữ trạng thái của người chơi hoặc bot, có thể là -1 (người chơi thắng), 1 (bot thắng), hoặc 0 (hòa).
  + emptyCells: Biến lưu trữ số ô trống trên bàn cờ.
  + currentI, currentJ: Vị trí ô vừa đánh (do bot hoặc người chơi thực hiện).

##### Giải thuật

* Kiểm tra kết quả trò chơi:
  + isFive(currentI, currentJ, lastPlayed): Kiểm tra nếu có một chuỗi 5 ký tự liên tiếp trên bàn cờ từ vị trí (currentI, currentJ) với người chơi hiện tại (bot hoặc người chơi).
  + Nếu có chuỗi 5 ký tự liên tiếp: Nếu lastPlayed == -1, người chơi thắng; nếu lastPlayed == 1, bot thắng.
  + Nếu không còn ô trống: Nếu số ô trống (emptyCells) bằng 0, trò chơi kết thúc với kết quả hòa.
  + Trường hợp khác: Trả về nullopt, nghĩa là trò chơi chưa kết thúc.

##### Thư viện

* optional: cho phép hàm trả về một giá trị có thể có hoặc không.

##### Mả giả(Pseudo code)

function checkResult():

if isFive(currentI, currentJ, lastPlayed) and (lastPlayed == -1 or lastPlayed == 1):

return lastPlayed // Trả về -1 (người thắng) hoặc 1 (bot thắng)

else if emptyCells <= 0:

return 0 // Hòa

else:

return nullopt // Trò chơi chưa kết thúc

end function

##### Độ phức tạp

* Không gian
  + O(1): Không gian sử dụng cho hàm này là hằng số, không phụ thuộc vào kích thước bàn cờ hoặc số lượng ô trống.
* Thời gian
  + O(1): Các phép kiểm tra như isFive() và kiểm tra emptyCells chỉ thực hiện các phép toán trực tiếp mà không cần duyệt qua toàn bộ bàn cờ. Mỗi phép kiểm tra là một thao tác đơn giản với độ phức tạp thời gian hằng số.

#### alphaBetaPruning

##### Cấu trúc dữ liệu

* TTable: Bảng tra cứu (Transposition Table), là một bảng băm lưu trữ các giá trị đã được tính toán trước đó với giá trị băm rollingHash là khóa và giá trị là một cặp (score, depth).
* bound: Một bảng băm lưu trữ thông tin về các nước đi hợp lệ và các giá trị gắn với mỗi ô trên bàn cờ.
* rollingHash: Giá trị băm hiện tại của bàn cờ, được cập nhật khi thay đổi trạng thái của bàn cờ.
* IntBoard: Bảng cờ (2D array) lưu trữ trạng thái các ô trên bàn cờ, với giá trị 1 cho bot và -1 cho người chơi.
* currentI, currentJ: Vị trí của nước đi hiện tại của bot.
* boardValue: Giá trị đánh giá của bàn cờ ở trạng thái hiện tại.
* nextBound: Thông tin về các giá trị bàn cờ sau khi thực hiện nước đi.

##### Giải thuật

* Alpha-Beta Pruning: Đây là một kỹ thuật tối ưu hóa cho thuật toán Minimax nhằm giảm số lượng nút phải kiểm tra trong cây tìm kiếm bằng cách loại bỏ các nhánh không cần thiết.
  + Điều kiện dừng: Nếu độ sâu depth đạt đến mức giới hạn hoặc trò chơi đã kết thúc (checkResult()), trả về giá trị đánh giá của bàn cờ hiện tại.
  + Bảng tra cứu: Kiểm tra xem trạng thái bàn cờ hiện tại đã được tính toán trước đó hay chưa trong TTable. Nếu có, trả về giá trị đã lưu.
  + MaximizingPlayer (Bot): Nếu lượt chơi của bot, tìm nước đi với giá trị tối đa.
  + MinimizingPlayer (Người chơi): Nếu lượt chơi của người chơi, tìm nước đi với giá trị tối thiểu.
  + Cắt tỉa Alpha-Beta: Dừng tìm kiếm các nhánh nếu beta <= alpha, tránh kiểm tra các nhánh không có khả năng ảnh hưởng đến kết quả cuối cùng.

##### Thư viện

* map: Dùng để lưu trữ bảng băm TTable và bound.
* pair: Dùng để lưu trữ các cặp giá trị, ví dụ như vị trí (i, j) của một ô trong bàn cờ.
* INT\_MIN, INT\_MAX: Các hằng số để khởi tạo giá trị ban đầu của max\_val và min\_val.

##### Mả giả(Pseudo code)

function alphaBetaPruning(depth, board\_value, bound, alpha, beta, maximizingPlayer):

// Kiểm tra điều kiện dừng

if depth <= 0 or checkResult() is not nullopt:

return board\_value

// Kiểm tra bảng tra cứu (TTable)

if TTable contains rollingHash and TTable[rollingHash].depth >= depth:

return TTable[rollingHash].score

if maximizingPlayer:

max\_val = -∞

children = childNodes(bound)

for each child in children:

i, j = child

if validMove(i, j):

new\_bound = bound

new\_val = evaluate(i, j, board\_value, 1, new\_bound)

makeMove(i, j, 1)

rollingHash ^= zobristTable[i][j][0]

updateBound(i, j, new\_bound)

eval = alphaBetaPruning(depth - 1, new\_val, new\_bound, alpha, beta, false)

if eval > max\_val:

max\_val = eval

if depth == this.depth:

updateBotMove(i, j, new\_bound)

alpha = max(alpha, eval)

undoMove(i, j, 1)

rollingHash ^= zobristTable[i][j][0]

if beta <= alpha:

break

updateTTable(TTable, rollingHash, max\_val, depth)

return max\_val

else:

min\_val = ∞

children = childNodes(bound)

for each child in children:

i, j = child

if validMove(i, j):

new\_bound = bound

new\_val = evaluate(i, j, board\_value, -1, new\_bound)

makeMove(i, j, -1)

rollingHash ^= zobristTable[i][j][1]

updateBound(i, j, new\_bound)

eval = alphaBetaPruning(depth - 1, new\_val, new\_bound, alpha, beta, true)

if eval < min\_val:

min\_val = eval

if depth == this.depth:

updatePlayerMove(i, j, new\_bound)

beta = min(beta, eval)

undoMove(i, j, -1)

rollingHash ^= zobristTable[i][j][1]

if beta <= alpha:

break

updateTTable(TTable, rollingHash, min\_val, depth)

return min\_val

end function

##### Độ phức tạp

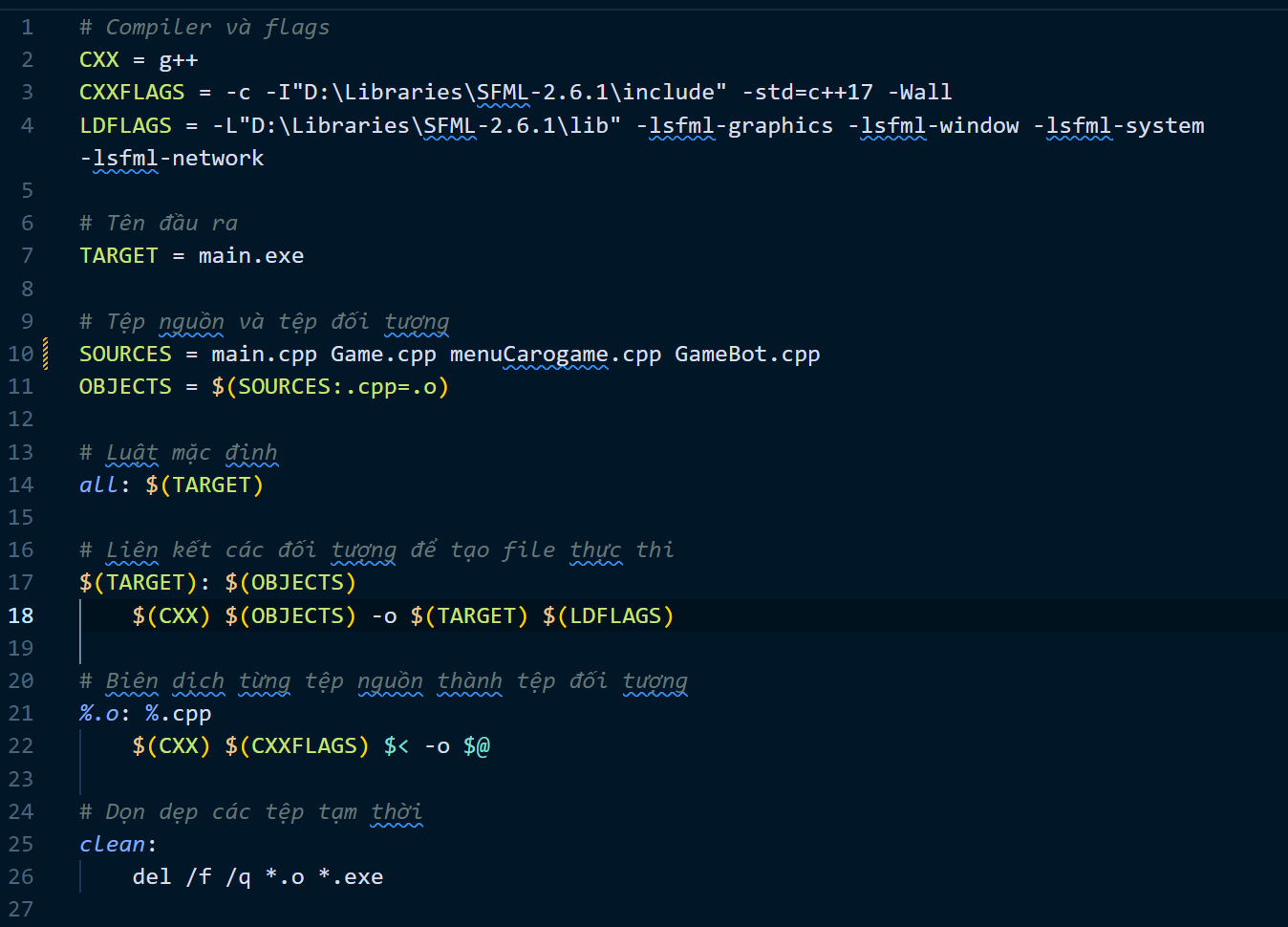
* Không gian
  + O(b^d): Cũng như thời gian, không gian cần thiết để lưu trữ cây tìm kiếm và bảng băm là O(b^d). Tuy nhiên, không gian thực tế có thể giảm nhờ vào bảng tra cứu (TTable) và cắt tỉa Alpha-Beta.
* Thời gian
  + O(b^d): Trong trường hợp không có cắt tỉa (alpha-beta pruning), độ phức tạp thời gian của thuật toán Minimax là O(b^d), trong đó b là bậc của cây (số nước đi hợp lệ ở mỗi trạng thái) và d là độ sâu của cây tìm kiếm. Cắt tỉa Alpha-Beta có thể giảm đáng kể độ phức tạp này trong thực tế.

# Cách chơi

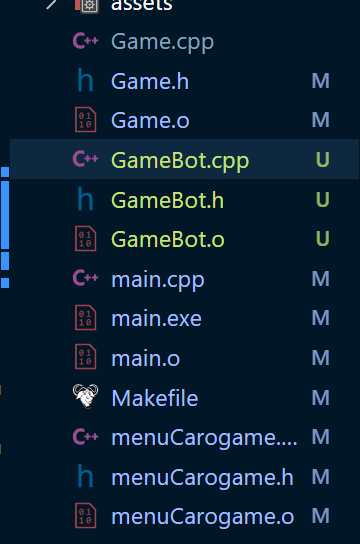
* Với bàn cờ 3x3, thì khi người chơi hay bot đạt được 3 nước liên tiếp sẽ chiến thắng.
* Với bàn cờ 5x5 trở lên, thì 5 nước liên tiếp sẽ chiến thắng.
* Khi bàn cờ đã không còn ô trống nào nữa và không có người chiến thắng, thì game hòa.
* Trò chơi hỗ trợ tính năng "Quay lại" trong chế độ người chơi với bot:
  + Thao tác: Nhấn Ctrl + Z để hoàn tác nước đi gần nhất.
  + Cơ chế hoạt động:
    - Khi thực hiện "Quay lại", nước đi cuối cùng của người chơi và bot sẽ được hoàn tác, trả bàn cờ về trạng thái trước đó.
    - Tính năng này có thể được sử dụng liên tục, ngay cả khi trận đấu đã kết thúc (thắng, thua, hoặc hòa), giúp người chơi tiếp tục trải nghiệm hoặc thử lại chiến thuật khác.

# Thư viện hỗ trợ, cách cài thư viện thêm để thực thi đồ án

* Thư viện hỗ trợ: SFML (Simple and Fast Multimedia Library).
  + Là thư viện lập trình đa phương tiện mã nguồn mở, được thiết kế để hỗ trợ phát triển các ứng dụng như trò chơi 2D, trình mô phỏng đồ họa và các phần mềm giao diện người dùng đa phương tiện.
  + Được viết bằng C++ và cung cấp các giao diện dễ sử dụng, đồng thời hỗ trợ việc phát triển trên nhiều nền tảng, bao gồm Windows, macOS và Linux.
* Cách sử dụng thư viện:
  + Gọi thư viện trong .cpp 
  + Lần lượt gọi g++ trong terminal
    - g++ -c main.cpp -I*<sfml-install-path>*/include
    - g++ main.o -o sfml-app -L*<sfml-install-path>*/lib -lsfml-graphics -lsfml-window -lsfml-system
    - ./sfml-app
* Có thể dùng công cụ Makefile như hình



* + Lần lượt gọi trong terminal
    - Make



* + - ./main để chạy game.

# Nguồn kham khảo chính

* [GomokuAI/source/Bot.py at main · hiamtrung103/GomokuAI](https://github.com/hiamtrung103/GomokuAI/blob/main/source/Bot.py)
* [ChatGPT](https://chatgpt.com/)