

POSTS AND TELECOMMUNICATIONS INSTITUTE OF TECHNOLOGY

**INFORMATION TECHNOLOGY DEPARTMENT**

**Icon

Description automatically generated with medium confidence**

**BASIC INTERNSHIP**

**Caro game using artificial intelligence**

**Instructor: PhD. Nguyen Duy Phuong**

**Student: Pham Truong Giang**

**Student code: B20DCCN208**

**Class: E20CQCN01-B**

***Hanoi, December 2023***

**Table of contents**

[**Preface 2**](#_c9p3gmgrt4kp)

[**Chapter 1: Abstract 3**](#_xwuk5h212qwr)

[**Chapter 2: Introduction 4**](#_dd6qkltsto6r)

[**Chapter 3: Related Work 5**](#_xiaqd5rab4gb)

[1. Evolving Tic-Tac-Toe Playing Algorithms Using Co-Evolution, Interactive Fitness and Genetic Programming by Helia Mohammadi, Nigel PA Browne, Anastasios N. Venetsanopoulos, and Marcus V. dos Santos 5](#_tnqvxjne07dr)

[2. Reinforcement Learning: Playing Tic-Tac-Toe by ocelyn Ho, Jeffrey Huang, Benjamin Chang, Allison Liu and Zoe Liu 5](#_1pfg0sd4fzur)

[3. Tic-Tac-Toe Learning Using Artificial Neural Networks by Abubakarsidiq Makame Rajab, Nadhira Haji Hassan and Abdul-Rahimsidiq Rajab 5](#_kj48pr7mi83e)

[**Chapter 4: Methodology 7**](#_o27s4edm1v8p)

[Main.Java: 9](#_mabymk59vqw8)

[CaroView.Java 9](#_b7ero0s1tc1o)

[CaroModel.Java 13](#_fltvebzidaky)

[CaroMain.Java 26](#_290icpygpfyj)

[**Chapter 5: Conclusion 31**](#_sz5ab3en3mcc)

[**Chapter 6: Reference 32**](#_x0d47l1q6ls9)

[**Acknowledgments 33**](#_kv3yb6qcizxe)

# **Preface**

Dear Teacher PhD. Nguyen Duy Phuong

First of all, we would like to express our deep gratitude to our teacher, who supported us throughout our study and research process. This project "Caro Game Using Artificial Intelligence" is not only a part of the learning process but also a testament to the passion for programming and artificial intelligence that we have nurtured for a long time.

The project is not limited to just an entertaining game but is also an effort to apply theoretical knowledge into practice, especially in the field of artificial intelligence and algorithms. Through that, we hope to not only improve our programming skills but also expand our understanding and creativity.

In this report, we will present in detail the process from the initial idea, the challenges encountered, the approach and problem solving, to the final results achieved by the project. We hope that this report is not only evidence of progress in our academic journey but also a useful resource for those interested in programming and artificial intelligence.

Thank you very much for taking the time to review and evaluate our report.

# **Chapter 1: Abstract**

This report presents the development of an enhanced Caro game, titled "Caro Game Using Artificial Intelligence with Alpha Beta Algorithm in Java". The primary objective of this project is to integrate advanced artificial intelligence (AI) into the traditional Caro game, thereby elevating the gaming experience with a more challenging and strategic AI opponent. By utilizing the Alpha Beta pruning algorithm, this version of Caro not only improves the efficiency of decision-making processes but also provides an intellectually stimulating experience for players.

The project is developed using Java, leveraging the capabilities of Java Core for the game's logic and Java Swing for the user interface. This combination allows for a robust and responsive gaming environment. The implementation of the Alpha Beta algorithm plays a crucial role in enhancing the AI's performance, enabling it to make optimal moves while minimizing computational resources.

This report details the process from conceptualization to the final execution of the game, highlighting the challenges faced and the methodologies employed in overcoming them. The integration of AI in Caro serves as a testament to the potential of artificial intelligence in transforming traditional games, offering insights into the development of more advanced AI-based applications in the future.

# **Chapter 2: Introduction**

The realm of artificial intelligence (AI) has seen remarkable advancements in recent years, significantly impacting various sectors, including gaming. This report introduces an innovative project titled "Caro Game Using Artificial Intelligence with Alpha Beta Algorithm in Java," which aims to bridge the gap between traditional board games and modern AI technologies. Caro, also known as Gomoku or Five in a Row, is a well-known strategy game that offers an ideal platform for implementing and testing AI algorithms due to its simple rules yet deep strategic complexity.

The core of this project is the integration of the Alpha Beta pruning algorithm, a widely recognized method in AI for decision-making in two-player games. By applying this algorithm, the game's AI can forecast multiple moves ahead, making intelligent and challenging decisions that enhance the gameplay experience. The choice of Java as the development language, with its robust Java Core for backend logic and Java Swing for graphical user interface, provides a stable and efficient platform for game development.

This project not only aims to create an engaging and challenging Caro game for players but also serves as a practical application of AI principles in game development. The report will cover the conceptualization, design, implementation, and testing phases of the project, offering insights into the challenges faced and the innovative solutions employed. Through this endeavor, we aim to demonstrate the capabilities of AI in transforming traditional games and explore the potential of Java in creating sophisticated AI-driven applications.

# **Chapter 3: Related Work**

## **Evolving Tic-Tac-Toe Playing Algorithms Using Co-Evolution, Interactive Fitness and Genetic Programming by Helia Mohammadi, Nigel PA Browne, Anastasios N. Venetsanopoulos, and Marcus V. dos Santos**

In Helia Mohammadi's study, focusing on genetic algorithms for developing a perfect Tic-Tac-Toe strategy, offers a contrast to the project's use of the Alpha Beta algorithm. This comparison not only highlights different AI methodologies applied to game strategy but also underscores the evolution of AI techniques in gaming. Discussing Hochmuth's approach provides a broader understanding of AI's role in strategic game development and illustrates the diverse solutions to achieving unbeatable gameplay strategies.

## **Reinforcement Learning: Playing Tic-Tac-Toe by ocelyn Ho, Jeffrey Huang, Benjamin Chang, Allison Liu and Zoe Liu**

The document "Reinforcement Learning: Playing Tic-Tac-Toe" discusses using Q-learning, a reinforcement learning algorithm, to train a computer for playing Tic-Tac-Toe. The study explores the efficiency of Q-learning in decision-making and pattern recognition, aiming to improve upon previous studies that achieved less than a 50% win/tie rate. The approach involves setting an initial high randomness factor (epsilon value) and decreasing it over time, allowing the program to learn from a large number of game simulations. The project played 300,000 games, eventually achieving a win/tie rate of about 90%. The research highlights the potential of Q-learning in enhancing AI performance in games and other practical fields.

## **Tic-Tac-Toe Learning Using Artificial Neural Networks by Abubakarsidiq Makame Rajab, Nadhira Haji Hassan and Abdul-Rahimsidiq Rajab**

The document "Designing and Implementation of Tic-Tac-Toe-4X4 based Artificial Intelligence using Python Programming" details the creation of a 4x4 Tic-Tac-Toe game using Python. The project's goal is to develop an AI capable of playing Tic-Tac-Toe, implementing strategies to win or tie. The AI uses the standard Minimax algorithm, modified with rules for optimal gameplay. Strategies include attempting to win, avoiding a loss, making strategic or random moves. The AI's decision-making utilizes a lightweight decision tree, making gameplay fast, balanced, and enjoyable. The study also includes a comparative analysis of the winning strategies in chess and demonstrates Tic-Tac-Toe strategies using symbolic model checking tools.

# **Chapter 4: Methodology**

**Topic:**Game Caro uses the alpha beta pruning algorithm.

**Rule:**The chess board has 12x12 cells, each turn the player will play

**Technology:**JavaCore, Java Swing

**Fundamentals of Alpha-Beta Pruning**

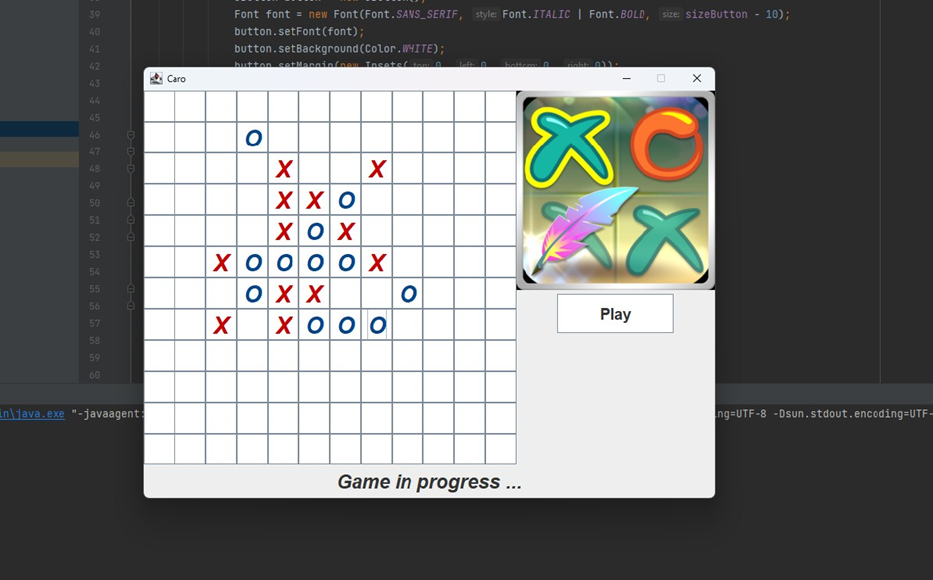
* **Minimax Algorithm:**
  + Minimax is a game tree search algorithm, where each node represents a state of the game. Each turn alternates between "min" (opponent, trying to reduce your score) and "max" (current player, trying to increase his score).
  + The algorithm considers all possibilities and chooses the optimal move based on the assumption that the opponent also plays optimally.
* **Alpha and Beta:**
  + Alpha is the best score a player can currently secure, starting at -Infinity.
  + Beta is the best score the opponent can guarantee, starting at +Infinity.
* **Pruning:**
  + During traversal of the game tree, if a move has a score worse than the best score the opponent (in this case, the bot) can guarantee, it is removed from consideration (pruned). This reduces the number of moves that need to be considered, increasing the efficiency of the algorithm.

**Application in Project “Caro game using artificial intelligence”**

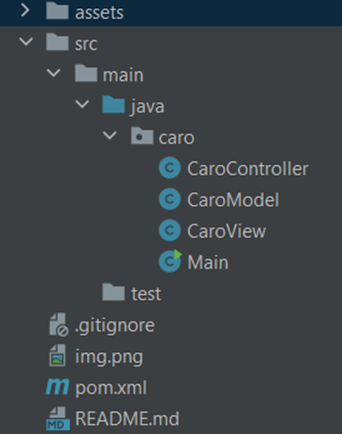
**Alpha-Beta Pruning algorithm is used to optimize move finding for AI (bot):**

* Find a Move for the Bot: When the bot is about to make a move, the algorithm considers possible moves and uses Alpha-Beta Pruning to choose the most optimal move. This includes evaluating moves based on both the bot's ability to win and prevent the opponent (player) from winning.
* Evaluating Game State: The algorithm also evaluates the game state after each move, determining whether it is a winner, a loser, or the game continues.
* Performance: Using Alpha-Beta Pruning increases the speed of the AI ​​by reducing the number of game states that need to be considered.

**Display:**

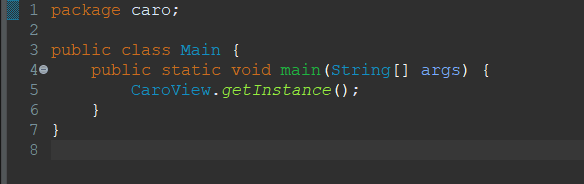


**Directory structure:**

****

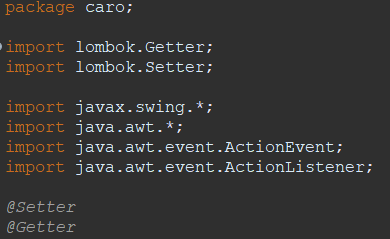
## Main.Java:

**Class Main:** Contains the Main function of the program and will call the instance of ClassView



## CaroView.Java

**CaroView** is a class in Java, inheriting from JFrame, which means it is a window of the application. This class uses Lombok to automatically generate getters and setters, and manage the user interface for the Caro game.



**Turn members** (Member Variables):

CaroController caroController: Controller object, obtained via CaroController.getInstance().



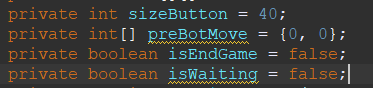
JButton[][] buttons: 2-dimensional array of buttons, used to display and manage cells on the chessboard.



JLabel labelNotification: Label displaying game status notifications.



Other variables like sizeButton, preBotMove, isEndGame, isWaiting to manage the state and configuration of the game.



**Constructor private CaroView():**

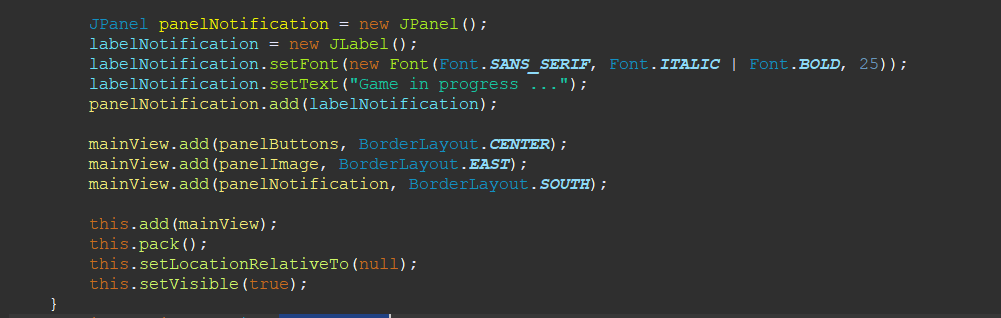
Set the basic configuration of the window (title, size, ...).

Initialize interface components such as button panel (JButton), image panel, notification panel.

Set events for buttons, including the game start event and the event when a player clicks on a box on the board.

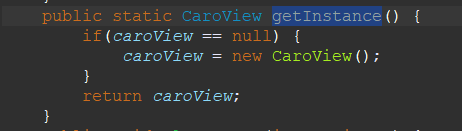
****

****

****

**getInstance() method:**

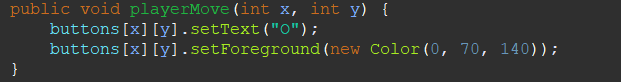
This method follows the Singleton design pattern, ensuring only one instance of CaroView is created.



**PlayerMove(int x, int y) and botMove(int x, int y) methods:**

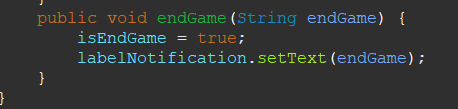
Updates the interface when a player or bot makes a move.

Set the button text and color to correspond to the move position.



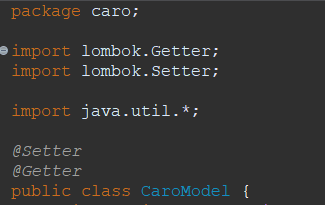
**EndGame(String endGame) method:**

Update the isEndGame state and display the end-game message.



## CaroModel.Java

**CaroModel** is the Caro game logic and state management model class. This class uses Lombok to automatically generate getters and setters.



**Member Variables:**

transpositionTable: Hash table that stores game states for reuse in computation.

maxDepth: Maximum depth in the search algorithm.

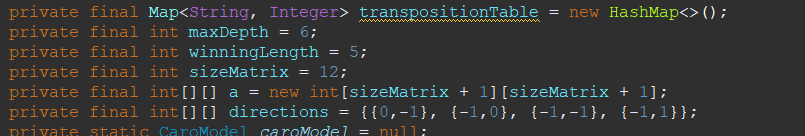
winningLength: Length needed to win (usually 5 in Caro).

sizeMatrix: Size of the chess board.

a[][]: Table representing the current state of the chessboard.

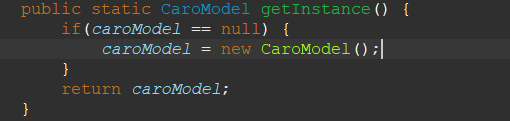
directions: Directions to check in the chessboard.

caroModel: Singleton representation of CaroModel.



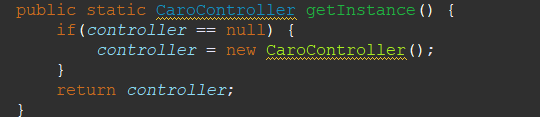
**getInstance() method:**

Following the Singleton pattern, there is guaranteed to be only one instance of CaroModel.



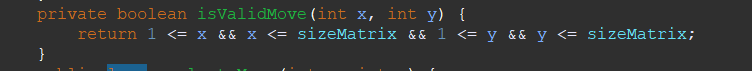
**readTranspositionTable() method:**

Expected to read data from CaroController.java, but not implemented yet.



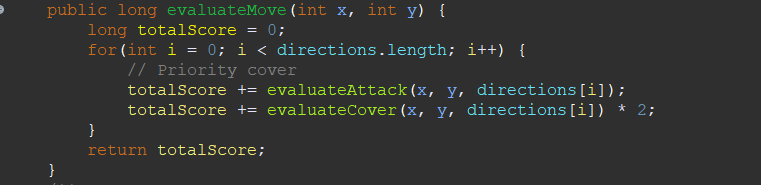
**isValidMove(int x, int y) method:**

* Check if a move is legal on the board.
* Parameters:
  + int x and int y: These are the coordinates of the move on the chessboard, with x being the row index and y being the column index.
* Logic of Function:
  + 1 <= x && x <= sizeMatrix: Checks if row index x is between 1 and sizeMatrix. sizeMatrix is ​​the size of the chessboard, so this condition ensures that x does not exceed the size of the chessboard.
  + 1 <= y && y <= sizeMatrix: Similarly, check if the y column index is within the allowable range of the chessboard.
* Result:
  + The function returns true if both row and column index conditions are satisfied, that is, move (x, y) is within the scope of the chessboard.
  + Conversely, if either index exceeds the size of the board, the function returns false, indicating that this is not a valid move.



**Method evaluateMove(int x, int y):**The evaluateMove function evaluates the total score of a move on the Caro board, calculated based on the attack and defense strategies in each direction. This helps determine the tactical value of the move, playing an important role in deciding the next move of the bot or player.

* Parameters:
  + int x, int y: Coordinates of moves on the chessboard.
* Variable totalScore:
  + Initialized with the value 0, totalScore will store the total score of the move.
* for loop:
  + Iterate through each direction in the directions array. The directions array contains the directions that the function will consider to evaluate the move.
* Score calculation:
  + evaluateAttack(x, y, directions[i]): Call the evaluateAttack method with the coordinates of the move and a specific direction from the directions array. This method returns a score based on the move's attack potential.
  + evaluateCover(x, y, directions[i]) \* 2: Similarly, evaluateCover evaluates the score based on the defensive ability of the move. This score is doubled (multiplied by 2) to increase the priority of the defensive strategy.
* Return score:
  + Finally, the function returns the totalScore, which is the sum of the calculated attack and defense scores for each direction.



**evaluateAttack method:**The evaluateAttack function evaluates the attack value of a move at (x, y) based on the number of consecutive pieces of the same type and the surrounding space in a specific direction. This helps determine the attack strategy for the Caro game, especially important in deciding moves for the AI ​​or player.

* Variable initialization:
  + countPointSame: Count the number of consecutive pieces of the same type.
  + countPointEmptyV1 and countPointEmptyV2: Count the number of empty cells in two directions.
  + directionX and directionY: Determine the test direction (taken from directionModel).
* Check in two directions:
  + The function uses a while loop to check the cells around the (x, y) position in the specified direction.
  + If consecutive pieces of the same type are found (a[X][Y] == a[x][y]), increase countPointSame.
  + If you encounter empty cells, count the number of consecutive empty cells (countPointEmptyV1 or countPointEmptyV2).
* Score rating:
  + Calculate the total number of empty cells: countPointEmpty = countPointEmptyV1 + countPointEmptyV2.
  + Evaluate the attack value based on the number of consecutive pieces of the same type and the number of empty cells. Cases:
    - If the number of pieces of the same type is close enough to win (countPointSame >= winningLength - 1), return the maximum or minimum value depending on the type of piece.
    - If the required number of troops is close and there is enough free space, return a high score value.
    - In other cases, returns scores based on the number of pieces of the same type and spaces.

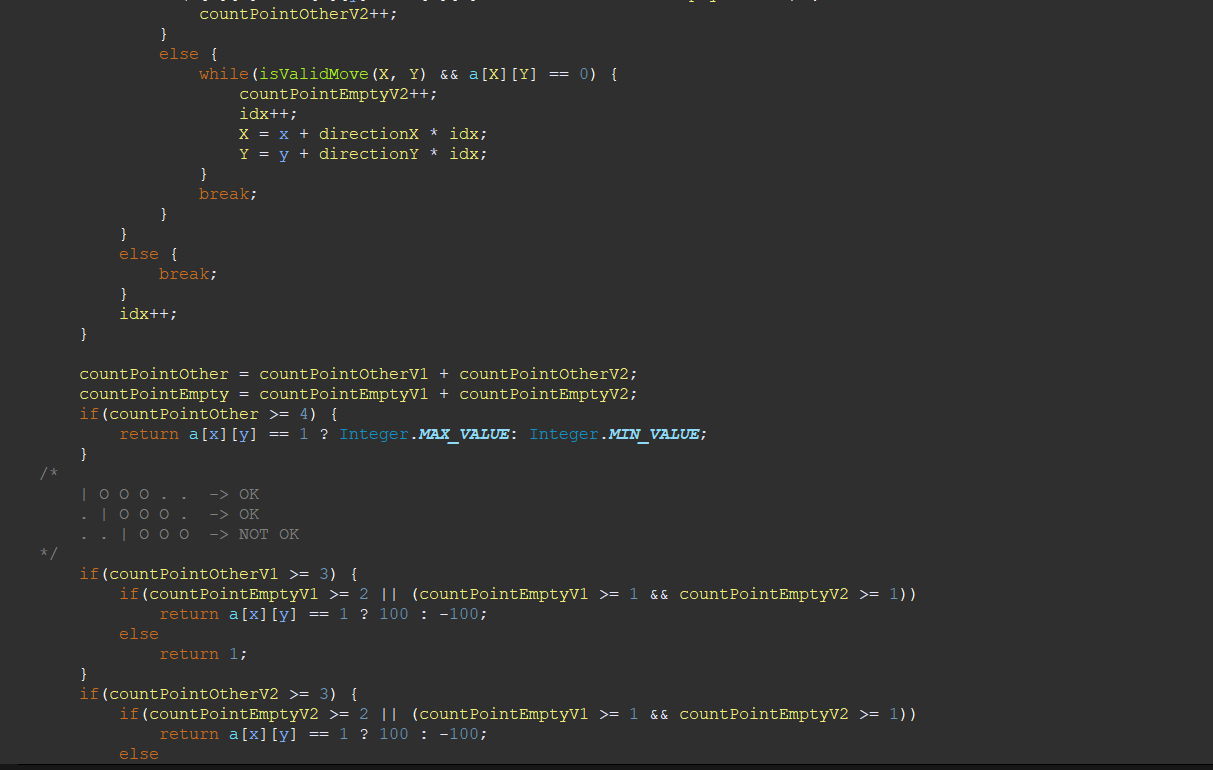




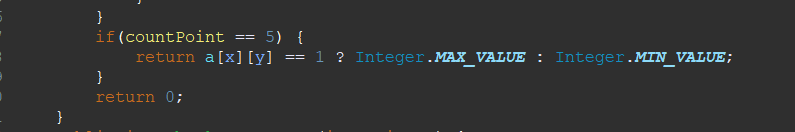
**evaluateCover method:** The evaluateCover function evaluates the defensive value of a move at (x, y) based on the number of consecutive enemy units and surrounding space in a specific direction. This helps determine the defensive strategy for the Caro game, important in deciding moves for the AI ​​or player.

* Variable initialization:
  + countPointOtherV1, countPointOtherV2: Count the number of enemy troops consecutively in two directions.
  + countPointEmptyV1, countPointEmptyV2: Count the number of empty cells in two directions.
  + directionX, directionY: Determine the test direction.
* Check in two directions:
  + Use a while loop to check surrounding cells in two directions (negative and positive).
  + If consecutive enemy units are found and there are no previous empty cells, increase countPointOtherV1 or countPointOtherV2.
  + Count the number of consecutive empty cells (countPointEmptyV1 or countPointEmptyV2).
* Calculate the total number of enemy troops and empty cells:
  + countPointOther = countPointOtherV1 + countPointOtherV2
  + countPointEmpty = countPointEmptyV1 + countPointEmptyV2
* Score evaluation:
  + Calculate score based on the number of enemy pieces and empty spaces around the move.
  + In specific cases:
    - If the number of surrounding enemies is large, return the maximum or minimum value.
    - Evaluate different situations based on the number of enemies and empty cells, returning the corresponding score.



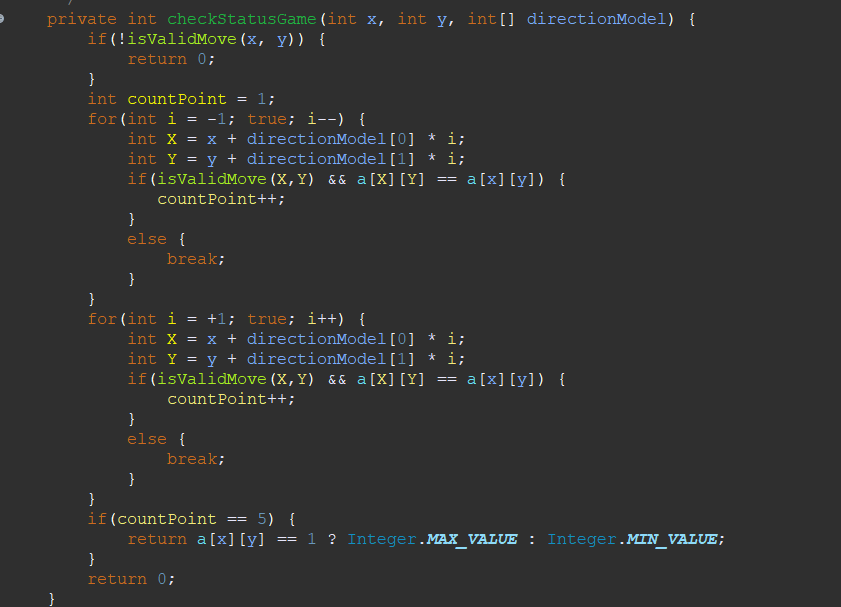






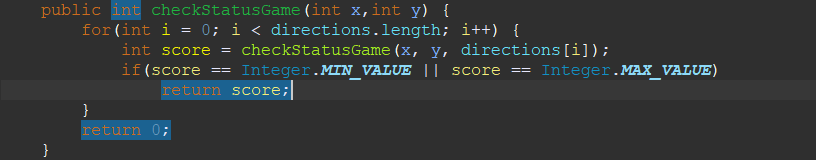
**Method checkStatusGame(int x, int y, int[] directionModel):**The checkStatusGame(int x, int y) function checks the game state after each move by checking if there is a winning sequence of pieces in any direction. If a win or loss is detected, the function returns the corresponding value; otherwise, the game continues. This is an important part in determining the state of the Caro game.

* for loop:
  + The function uses a loop to check the game state in each direction in the directions array. The directions array contains the directions that the function will consider to check the game state.
* Check game status:
  + Call the function checkStatusGame(x, y, directions[i]) with the current position of the move and a specific direction from the directions array.
  + The function checkStatusGame(x, y, directions[i]) checks to see if there is a string of consecutive pieces long enough to win in that direction.
* Handling the result:
  + If the function checkStatusGame(x, y, directions[i]) returns Integer.MIN\_VALUE or Integer.MAX\_VALUE, this means the move at (x, y) resulted in a win or loss (depending on the value) .
  + In this case, the function checkStatusGame(int x, int y) will return the corresponding value (win or lose).
* Returns results:
  + If no winning or losing states are detected in any direction, the function returns 0, indicating that the game continues.

****

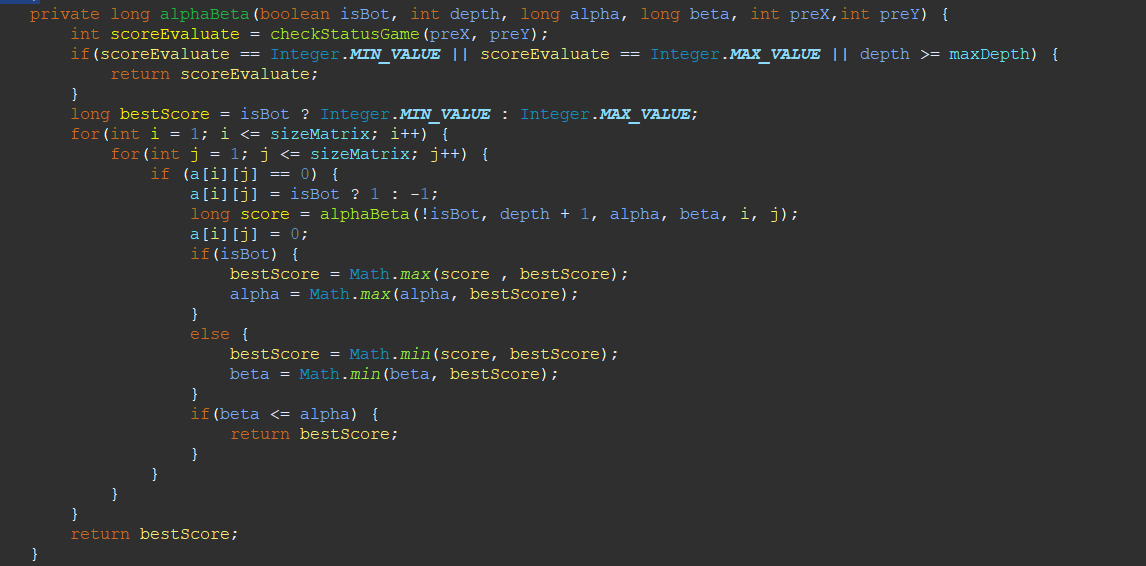
**Method checkStatusGame(int x, int y):**The checkStatusGame(int x, int y) function plays an important role in determining the state of the Caro game after each move, checking if any winning or losing states occur based on the current position. of moves and possible directions on the chessboard.

* for loop:
  + Iterate through the directions array, each element representing a specific direction on the board (e.g. horizontal, vertical, diagonal).
* Check Game Status:
  + checkStatusGame(x, y, directions[i]): Call the function to check the game status at location (x, y) in directions[i].
  + The checkStatusGame(x, y, directions[i]) function checks to see if there is a long enough sequence of pieces in that direction to produce a win.
* Handling the result:
  + If the test result is Integer.MIN\_VALUE or Integer.MAX\_VALUE, this implies that a winning or losing state has occurred.
    - Integer.MIN\_VALUE can represent a losing state.
    - Integer.MAX\_VALUE can represent a winning state.
  + In this case, the function will return that value, indicating the result of the game.

****

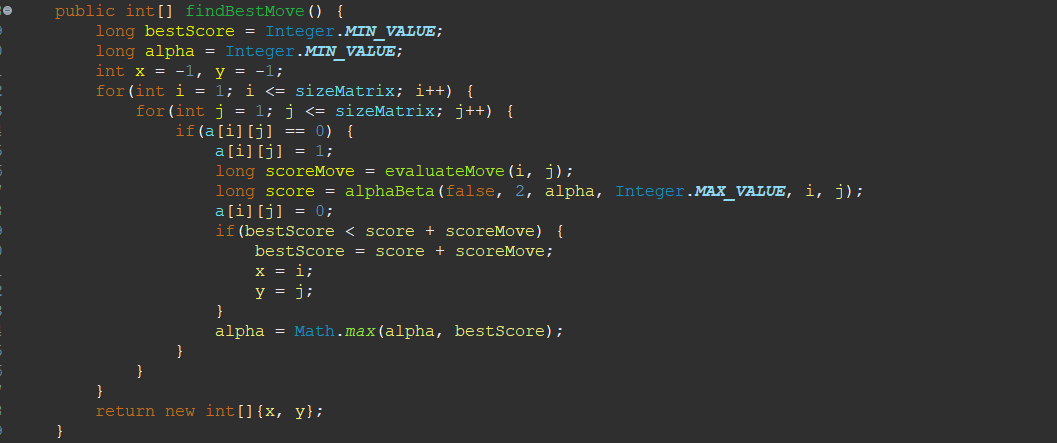
**Phương thức alphaBeta(boolean isBot, int depth, long alpha, long beta, int preX,int preY):** The alphaBeta function is an important part of determining the strategy for the AI ​​in the game Caro, helping to optimize the search for the best move through pruning inefficient branches in the game tree.

* Check the Stop Condition:
  + checkStatusGame(preX, preY): Check the game status at position preX, preY.
  + If the game ends or maxDepth is reached, return the score.
* Initialize bestScore:
  + For bots, bestScore starts from Integer.MIN\_VALUE and conversely, Integer.MAX\_VALUE for players.
* Browse Chess Board:
  + Check each square on the chessboard. If that cell is empty (a[i][j] == 0), try the move there.
* Alpha-Beta Recursion:
  + Set the value in that box depending on the bot or player's turn.
  + Recursively call alphaBeta with the next pass and increase the depth.
  + Then restore the value of that cell (backtracking).
* Update bestScore, alpha and beta:
  + Update bestScore with highest (for bots) or lowest (for players) scores.
  + Updated alpha and beta limits to prune unnecessary branches, optimizing the search process.
* Alpha-Beta Pruning:
  + If beta <= alpha, prune that branch and return bestScore.
* Returns bestScore:
  + After going through all possibilities, return the best bestScore found.

****

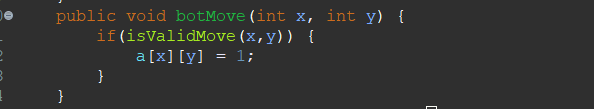
**findBestMove() method**: The findBestMove function is the heart of the AI ​​algorithm in the Caro game, helping AI find the most optimal move based on current assessment and predicting future consequences through the Alpha-Beta Pruning algorithm.

* Initialization:
  + bestScore: The current best score, starting from the minimum value.
  + alpha: Lower limit for Alpha-Beta Pruning.
  + x, y: Position of the best move (undetermined, default is -1).
* Browse Chess Board:
  + The nested for loop iterates over each square on the chessboard.
* Find Possible Moves:
  + Check if cell (i, j) is empty (a[i][j] == 0), then place a piece there (assume AI's piece).
* Move Rating:
  + evaluateMove(i, j): Evaluate the move based on the current strategy.
  + alphaBeta(false, 2, alpha, Integer.MAX\_VALUE, i, j): Use the Alpha-Beta algorithm to predict and evaluate the best move for the opponent after this move by AI.
* Optimal Moves Update:
  + If the total score of the current move and future prediction is greater than the current best score, update bestScore and save the position of the move.
* Update alpha:
  + Updates the alpha value for use in subsequent calls to alphaBeta, allowing for efficient pruning.
* Returns Best Move Position:
  + After going through all the empty cells, return the position (x, y) of the best move the AI ​​can make.



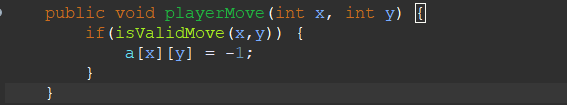
**botMove(int x, int y) method:**The botMove function manages the placement of the bot's pieces on the Caro chessboard. It only makes a move if the position is valid, i.e. there are no pieces there and are within the range of the board. This is a basic but important part of the AI's move processing in the game.

* Check Eligibility:
  + isValidMove(x, y): This function checks whether position (x, y) is an empty cell and within the limits of the chessboard.
* Make the Move:
  + If position (x, y) is valid, the function will place one of the bot's pieces at that position on the board. In this case, placing a piece is represented by assigning a[x][y] = 1. The value 1 can be understood as a symbol representing the bot's piece.



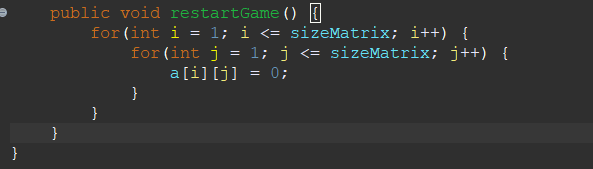
**Method playerMove(int x, int y):**The playerMove function manages the placement of the player's pieces on the Caro board. It only makes a move if the position is valid, i.e. there are no pieces there and are within the range of the board. This is a basic but important part of the player's move processing in the game.

* Check Eligibility:
  + isValidMove(x, y): This function checks whether position (x, y) is an empty cell and within the limits of the chessboard.
* Make the Move:
  + If position (x, y) is valid, the function places a player's piece at that position on the board. In this case, the placement of the pieces is represented by assigning a[x][y] = -1. The value -1 can be understood as a symbol representing a player's piece.



**restartGame() method:**The restartGame function performs the important task of resetting the chess board, helping prepare for a new game to begin. This is useful for playing multiple games in a row without recreating the entire game state from scratch.

* Nested Loops:
  + Two nested for loops iterate over each square on the chessboard. The outer loop iterates over each row (i), and the inner loop iterates over each column (j) in that row.
* Reset the Value of Each Cell:
  + In each square on the board (a[i][j]), the value is reset to 0. In the context of the game Caro, the value 0 is often used to represent an empty square, containing no pieces. of any player.
* Restart the Chessboard:
  + After completing the loop, all the squares on the chessboard will have no chess pieces left, meaning the chessboard is ready for a new game.



## CaroMain.Java

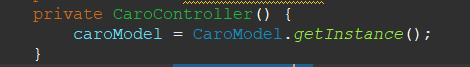
Declaration and Singleton Pattern:

* **private final CaroModel caroModel:** An instance of the CaroModel class, used to interact with the game state.
* **private static CaroController controller:** Static variable of the CaroController class itself, following the Singleton model.



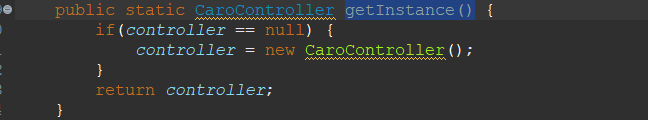
**CaroController() method:**The private CaroController() constructor in the CaroController class is an important part of the Singleton design pattern. It ensures that only a single instance of CaroController is created in the application. Initializing the caroModel in this constructor ensures that the CaroController always has access to the unique instance of the CaroModel, allowing it to manage the state and logic of the Caro game efficiently.

* Access Modifier - private:
  + The private keyword indicates that this constructor cannot be called from outside the CaroController class. This is part of the Singleton design pattern, which ensures that only a single instance of the CaroController class is created during the runtime of the application.
* Initialize caroModel:
  + In the constructor, the caroModel object is initialized. caroModel is an instance of the CaroModel class, obtained via the getInstance() method.
  + CaroModel.getInstance(): This method also follows the Singleton design pattern. It ensures that only a single instance of CaroModel exists. When calling getInstance(), if the CaroModel instance does not already exist, it will be created; if already exists, the existing instance will be returned.



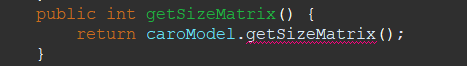
**getInstance() method:**The getInstance() method in the CaroController class ensures that only a single instance of this class is created and used in the entire application. This is useful in managing the overall state and actions of the game, ensuring consistency and avoiding the creation of unnecessary multiple instances of the same class.

* Singleton Design Pattern:
  + The Singleton design pattern ensures that only a single instance of a class is created during the runtime of the application.
* Static Method:
  + getInstance() is a static method, which means you can call it without an instance of the CaroController class.
* Check Show Current:
  + if(controller == null): Checks whether the controller static variable, an instance of the CaroController class, has been initialized or not.
  + If the controller has not been initialized (null), the method will create a new instance of CaroController.
* Initialize Instance:
  + controller = new CaroController(): Creates a new instance if it does not exist.
* Returns Express:
  + return controller: Returns the current instance of CaroController. If this instance already exists, it will not create a new one but return the existing instance.



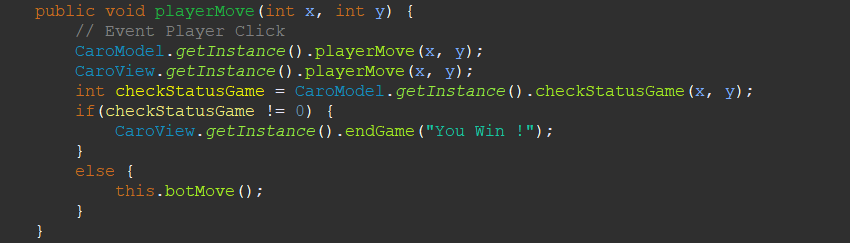
**getSizeMatrix() method:**The getSizeMatrix method in the CaroController class provides a way to access information about the size of the board from the CaroModel. This helps separate the game's data processing logic (in CaroModel) from the control and UI communication logic (in CaroController).

* Purpose of Method:
  + The getSizeMatrix() method is used to get the size of the Caro board. This size is usually the number of rows and columns in the chessboard.
* Return Value from CaroModel:
  + return caroModel.getSizeMatrix();: This method returns the size of the chessboard taken from the CaroModel class. CaroModel is the layer that manages the core data and logic of the game, including the size of the board, the state of the tiles, and the rules of the game.
* Interaction between Controller and Model:
  + The CaroController class acts as an intermediary between the user interface (view) and the game's data (model). The getSizeMatrix method is an example of this interaction: it allows other application components (such as the user interface) to get information about the board size without directly interacting with the CaroModel.



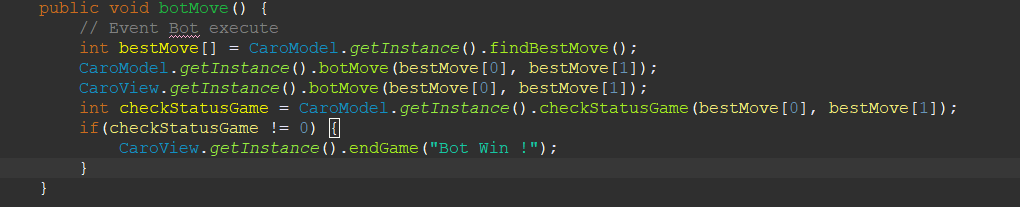
**Method playerMove(int x, int y):** The playerMove method in the CaroController class is the processing center for the player's moves. It updates the game state in the model, reflects the change in the user interface (view), and checks whether the move leads to the end of the game or not. If the player does not win, the function passes the turn to the bot to continue the game.

* Handling Player Moves:
  + CaroModel.getInstance().playerMove(x, y): Updates the game state in CaroModel to reflect the player's new move at position (x, y).
* User Interface Updates:
  + CaroView.getInstance().playerMove(x, y): Updates the user interface (view) to reflect the player's moves on the board.
* Check Game Status:
  + int checkStatusGame = CaroModel.getInstance().checkStatusGame(x, y): Check the game status after the player's move to see if it leads to a win, a loss, or the game continues.
* Handling the result:
  + If checkStatusGame is not equal to 0 (meaning the player wins), the function announces victory via CaroView.getInstance().endGame("You Win !").
  + Otherwise, if a win is not detected, the function calls this.botMove() to allow the bot to make the next move.



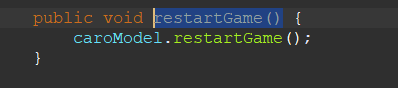
**botMove() method:**The botMove method in CaroController manages move selection and execution for the bot, updates the game state and user interface, and checks the outcome of the game after each bot move. If the bot wins, the function calls the endGame method to notify the game result.

* Find the Most Optimal Move for Bot:
  + int bestMove[] = CaroModel.getInstance().findBestMove();: Call the findBestMove() method from CaroModel to find the most optimal move for the bot. bestMove is an array containing the coordinates of the move (x, y).
* Execute Bot Moves:
  + CaroModel.getInstance().botMove(bestMove[0], bestMove[1]);: Updates the state of the board in CaroModel with the bot's new move.
  + CaroView.getInstance().botMove(bestMove[0], bestMove[1]);: Updates the user interface (CaroView) to reflect the bot's moves on the chessboard.
* Check Game Status After Bot Moves:
  + int checkStatusGame = CaroModel.getInstance().checkStatusGame(bestMove[0], bestMove[1]);: Check if the bot's move leads to victory or not.
* Handling the result:
  + If checkStatusGame is not equal to 0 (meaning the bot wins), then notify the bot of victory via CaroView.getInstance().endGame("Bot Win !").



restartGame() method: The restartGame method in CaroController is a simple yet effective way to restart the game, it does this by asking the CaroModel to reset the board's state. This ensures that all game rules and logic are handled centrally at the CaroModel, while the CaroController coordinates actions between the model and view.

* Purpose of Method:
  + The restartGame() method is used to restart the game, resetting the board to its original state.
* Call the restartGame Method in CaroModel:
  + caroModel.restartGame();: This method calls restartGame() from the caroModel object, which is an instance of the CaroModel class.
  + The CaroModel class manages the state and basic logic of the Caro game, including the state of the tiles on the board. When calling restartGame() in CaroModel, it will reset the state of all the cells on the board, preparing for a new game.



# **Chapter 5: Conclusion**

In conclusion, the "Caro Game Using Artificial Intelligence with Alpha Beta Algorithm in Java" project successfully demonstrated the integration of complex AI strategies into a traditional board game. The implementation of the Alpha Beta pruning algorithm significantly enhanced the game's challenge, providing a formidable AI opponent capable of strategic gameplay. The project utilized Java's robust programming capabilities to create an engaging user experience with a responsive interface. The journey through this development process not only highlighted the potential for sophisticated AI in board games but also opened avenues for future enhancements, including multiplayer modes and improved AI learning algorithms. The experience and knowledge gained lay a strong foundation for further exploration in the field of AI gaming.

# **Chapter 6: Reference**

1. Minimax with Alpha-Beta Pruning in Python - Mina Krivokuća -https://stackabuse.com/minimax-and-alpha-beta-pruning-in-python/
2. Finding optimal move in Tic-Tac-Toe using Minimax Algorithm in Game Theory - GeeksforGeeks - https://www.geeksforgeeks.org/finding-optimal-move-in-tic-tac-toe-using-minimax-algorithm-in-game-theory/
3. A Fast Parallel Alpha-Beta Algorithm for Tic TAC Toe Game - Amina Y. AlSallut, Hana H. Hejazi, Heba A. AbuGhali - November 2013
4. An Intuitive Implementation Of Alpha-Beta Pruning Using Tic-Tac-Toe - Asad Ali Jatoi, Sarvech Ali, Isma Farah Siddiqui - July 2020

# **Acknowledgments**

We would like to send our most sincere thanks to teacher for your dedicated guidance, extensive knowledge and continuous support throughout the process of implementing the project "Caro game using artificial intelligence". Thanks to teacher's guidance and support, we not only completed the report excellently but also learned a wealth of valuable knowledge, especially in the fields of artificial intelligence and Java programming.

Teacher were not only a guide but also a source of inspiration for us throughout the research and development process of the project. Your patience and dedication helped us overcome difficulties and challenges and develop a project that we are proud of.

We would like to express our deep gratitude and respect to the teacher. Hopefully we will have more opportunities to learn from you in the future.