A close-up of a logo

Description automatically generated

***Analysis Of Algorithim***

***Group Project***

***Group 32***

|  |  |
| --- | --- |
| ***Zeyad Sherif Fayez*** | ***219350*** |
| ***Hussein Ayman*** | ***223696*** |
| ***SeifEldin ElAlami*** | ***236459*** |

**Game Tree Algorithm Overview**

The bot (Red player) automatically uses the **Minimax algorithm** to determine the best possible move. The Minimax algorithm recursively explores all possible future game states up to a fixed depth, alternating between maximizing (Red’s turn) and minimizing (Green’s turn) strategies.

**Algorithm 1: GET\_AI\_MOVE()**

**Purpose:**  
Finds the best move for the Red bot using the Minimax game tree.

**Description:**  
This function iterates over all Red tokens and their legal moves. For each simulated move, it calls the MINIMAX() function to evaluate the resulting game state. It then selects the move with the highest evaluation score.

**Pseudocode:**

function GET\_AI\_MOVE():

bestValue ← -∞

bestMove ← null

for each Red token R on the board:

for each legal move M of R:

Apply move M (simulate it)

moveValue ← MINIMAX(depth - 1, isMaximizing = false)

Undo move M

if moveValue > bestValue:

bestValue ← moveValue

bestMove ← M

return bestMove

**Algorithm 2: MINIMAX()**

**Purpose:**  
Recursively evaluates the game tree to provide the Red player with the best possible move based on score evaluation.

**Description:**  
This function simulates the game tree up to a fixed depth, alternating between the Red bot (maximizing) and the Green player (minimizing). The evaluation returns a score representing a win, loss, or draw. It chooses the optimal move by comparing the returned values.

**Pseudocode:**

function MINIMAX(depth, isMaximizingPlayer):

score ← evaluateBoard()

if score == 10 or score == -10:

return score // Terminal state: win or loss

if depth == 0:

return 0 // Reached depth limit

if isMaximizingPlayer: // Red player's turn

best ← -∞

for each Red token R:

for each legal move M of R:

Simulate move M

val ← MINIMAX(depth - 1, false)

Undo move M

best ← max(best, val)

return best

else: // Green player's turn

best ← +∞

for each Green token G:

for each legal move M of G:

Simulate move M

val ← MINIMAX(depth - 1, true)

Undo move M

best ← min(best, val)

return best

**Analysis of the implemented algorithm:**

### **Purpose:**

The **Minimax algorithm** is used to control the decision-making of the **Red player (the bot)** in this game. The goal is to help the bot figure out which move gives it the best chance of winning. It does this by looking ahead at all the possible moves it and the opponent could make, and then picking the one that leads to the best outcome. Even though the bot doesn’t learn or adapt like real AI, the algorithm gives it a way to act smart — by assuming both players will make the best choices they can. This helps the bot avoid careless moves, block the opponent when needed, and overall play in a way that feels more strategic and challenging for the human player.

**How It Works:**

The game uses **two main functions**: **getAIMove()** and **minimax().** These work together to simulate future moves, evaluate them, and pick the best one for the bot.

* **getAIMove()** loops through every Red token on the board. For each one, it checks if there's a legal move (moving right into an empty space). If there is, it simulates that move and calls minimax() to evaluate the result. After testing all moves, it chooses the one with the highest score.
* **minimax()** is a recursive function that simulates the rest of the game after a move is made. It switches between the bot (Red) trying to maximize the score and the human player (Green) trying to minimize it. It evaluates moves by checking win conditions and depth limits.

The algorithm uses a fixed depth (in this case, 3) to limit how far ahead it looks. This helps avoid performance issues, especially since each move leads to many possible future game states.

**Scoring and Evaluation:**

The scoring is pretty simple:

* If the Green player has no tokens left, the bot wins → score: **+10**
* If the Red player has no tokens left, the bot loses → score: **-10**
* Otherwise → score: **0**

This evaluation helps the algorithm decide which paths lead to a win, a loss, or a draw. Even though the scoring is basic, it’s enough to guide the bot toward smart choices during gameplay.

**Strengths and Limitations:**

**Strengths:**

* Makes the bot feel smart and challenging without any learning involved.
* Helps avoid obvious mistakes like walking into traps or missing a win.
* Encourages both offensive and defensive play.

**Limitations:**

* The algorithm doesn’t actually "learn" from the player — it just simulates moves based on assumptions.
* The depth is limited to 3 levels, which means the bot can’t always see far ahead in complex situations.
* Performance might slow down if the board or rules get more complicated.

**Understanding the Game Setup;**

Before we analyze the algorithm, it’s important to understand how the game works:

* The board is a fixed 4x4 grid → size = **n = 4**
* Each player (Red and Green) starts with **4 tokens** placed on one edge of the board.
* The Red tokens move **right**, and Green tokens move **down**.
* At any time, each token may have up to **2 possible moves**:
  + A simple forward move
  + A jump over an opponent (if valid)

**Function: getAIMove():**

* This function runs once per turn for the bot.
* It loops through **all Red tokens** (max 4), and for each one:
  + It checks **legal moves** (up to 2 moves per token).
  + For each legal move, it simulates the move and calls minimax() with depth = 3.

So, in the worst case:

* Up to **4 Red tokens** × **2 moves each** = **8 calls to minimax()** at depth 3.

**Function: minimax(depth, isMaximizing):**

This is a recursive function. It simulates all possible move combinations up to a fixed depth (here, depth = 3). Let’s analyze how many recursive calls are made.

We define:

* **b** = average branching factor = number of possible moves per player per turn
* **d** = depth of the tree = how many turns ahead we simulate

In our case:

* Each player has at most **4 tokens**
* Each token can make up to **2 legal moves**
* So, maximum **b = 4 × 2 = 8**

This means each node in the game tree can have up to 8 child nodes.

The total number of nodes explored by the algorithm is:

A black background with white text

Description automatically generated

This is a geometric series, which simplifies to:

A black background with white text

Description automatically generated

If we only care about how the function grows as the board gets more complex, we use Big O notation and focus on the highest-order term:

A black background with white text

Description automatically generated

In our case:

* b=8
* d=3

So:

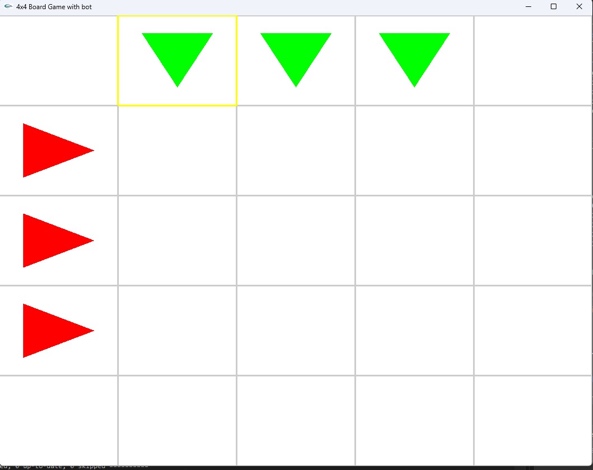
A black background with white text

Description automatically generated

This means, in the worst case, **512 recursive calls** may happen during a single bot decision.

**Analysis of the GUI:**

To build the GUI for the 5x5 board game using **OpenGL** and **GLUT**, start by initializing the window using **glutInit**, setting a display mode with RGB and double buffering, and defining a 500x500 pixel window. Next, create a **display()function** that clears the screen, draws the grid, and renders the tokens on each frame. The grid is drawn with black lines using **glBegin**(**GL\_LINES**) by dividing the window into five horizontal and five vertical segments. For token drawing, loop through the 2D board[5][5] array and use a circle function (built with **GL\_TRIANGLE**) to draw either a green or red circle at the center of each occupied cell. Coordinates are normalized from 0.0 to 1.0, so each cell is mapped by scaling indices accordingly and flipping the Y-axis for proper display. Finally, register the display function using **glutDisplayFunc**, and start the main loop with **glutMainLoop()** to keep the GUI responsive and updated.



A screenshot of a game

Description automatically generated