**A flag on a shield

AI-generated content may be incorrect.**

**Analysis of Algorithms**

**24CSCI01I**

Group no. 37

Presented to DR. Mostafa & TA. Lobna

|  |  |
| --- | --- |
| Seif Ahmed Amin | 230598 |
| Seif Allah Mohamed | 229879 |
| Ahmed Hany | 236295 |
| Amr Galal | 235355 |

**Game Tree**

**Description:**

This is a 2-player turn-based strategy game built with SFML and C++. The board is a 4x4 grid where Green tokens (Player A) try to reach the rightmost column, while Red tokens (Player B) aim to reach the bottom row. The first player to get 3 tokens to their target zone wins the game.

**Players:**

* Player A (Green) – Controlled by the user.
* Player B (Red) – Controlled by the bot.

**Gameplay Rules:**

1. Green tokens move right.
2. Red tokens move down.
3. Players take turns moving one token.

**Tokens can:**

* Move one step forward (default).
* Jump two steps if blocked.

**The game supports:**

* Undo (reverts one move).
* Restart (resets the board).
* Visuals and scores.

**Pseudocode:**

START:

INCLUDE graphics library

INCLUDE standard libraries

DEFINE BOARD\_SIZE = 4

DEFINE CELL\_SIZE = 100

DEFINE WINDOW\_WIDTH = BOARD\_SIZE \* CELL\_SIZE + 200

DEFINE WINDOW\_HEIGHT = BOARD\_SIZE \* CELL\_SIZE + 150

DEFINE STRUCT Token

int row, col

bool reachedGoal = false

DEFINE STRUCT GameState

list of Token playerA

list of Token playerB

bool isPlayerATurn

bool gameOver

CLASS TokenGame

PRIVATE:

list of Token playerA, playerB

bool isPlayerATurn = true

bool gameOver = false

bool winnerAnnounced = false

stack of GameState gameHistory

bool canUndo = false

graphics shapes: tokenShape, boardBg, targetZoneA, targetZoneB

font and texts: font, turnText, scoreText, gameOverText

buttons: restartButton, undoButton

CONSTRUCTOR:

CALL resetGame()

SETUP graphics shapes, text, buttons

LOAD font

METHOD saveState:

CREATE GameState with current game state

PUSH state onto gameHistory

SET canUndo = true

END METHOD

METHOD undoMove:

IF gameHistory is empty THEN RETURN false

POP last state

RESTORE state to playerA, playerB, isPlayerATurn, gameOver

SET winnerAnnounced = false

SET canUndo depending on whether history is empty

RETURN true

END METHOD

METHOD isOccupied(row, col):

FOR EACH token IN playerA:

IF token.row == row AND token.col == col AND token has NOT reached the goal:

RETURN true

FOR EACH token IN playerB:

IF token.row == row AND token.col == col AND token has NOT reached the goal:

RETURN true

RETURN false

END METHOD

METHOD makeMove(tokenIndex):

CALL saveState()

IF it's Player A's turn THEN:

token ← playerA[tokenIndex]

IF token is already at the last column (goal position) THEN:

mark token as reachedGoal

switch turn to Player B

RETURN

IF the cell to the right is NOT occupied THEN:

move token one cell to the right

ELSE IF two cells to the right are within board AND not occupied THEN:

move token two cells to the right (jump)

ELSE IF diagonal lower-right cell is within board AND not occupied THEN:

move token diagonally (down and right)

IF token is now at the last column THEN:

mark token as reachedGoal

aFinished ← count how many Player A tokens have reached goal

bFinished ← count how many Player B tokens have reached goal

IF aFinished OR bFinished equals (BOARD\_SIZE - 1) THEN:

gameOver ← true

ELSE:

switch turn to the other player

END METHOD

METHOD makeBotMove:

IF it's player A's turn OR game is over THEN RETURN

CALL saveState()

FOR each token in playerB:

TRY move down by 1 or 2 if possible, mark reachedGoal if at last row, switch turn, CALL checkGameOver(), RETURN

IF no move possible, switch to player A's turn

METHOD checkGameOver:

COUNT playerA and playerB tokens in goal

IF any count == BOARD\_SIZE - 1 THEN SET gameOver

METHOD handleClick(x, y):

IF Restart button clicked THEN CALL resetGame()

ELSE IF Undo button clicked AND canUndo THEN CALL undoMove()

ELSE IF gameOver THEN RETURN

CONVERT x, y to row, col

VALIDATE position

FOR each token in playerA:

IF it's player A's turn AND clicked on a valid token THEN CALL makeMove(i)

METHOD resetGame:

CLEAR playerA and playerB lists

INITIALIZE playerA tokens in column 0

INITIALIZE playerB tokens in row 0

RESET flags and game history

PRINT game reset message

METHOD draw(window):

CLEAR window

DRAW board, target zones, and grid lines

DRAW playerA tokens in green

DRAW playerB tokens in red

UPDATE and DRAW current turn text

COUNT scores and DISPLAY them

DRAW Restart and Undo buttons with text

IF gameOver:

DRAW semi-transparent overlay

DISPLAY winner text depending on score

ANNOUNCE winner in console (only once)

METHOD findPathToGoal(token, visited, path):

IF token has reached goal column/row based on player THEN ADD to path, RETURN true

MARK current cell as visited

ADD current position to path

DEFINE movement directions based on player

FOR each direction:

COMPUTE new position

IF inside bounds, not visited, and not occupied THEN

CREATE next token at new position

IF recursive call to findPathToGoal(next, ...) is true THEN RETURN true

BACKTRACK (remove last step), RETURN false

METHOD testBacktrackingForFirstToken:

IF it's player A's turn AND tokens exist:

INIT visited and path

IF path found using findPathToGoal THEN

PRINT path to console

ELSE PRINT "No path found"

MAIN:

CREATE game window

PRINT game instructions

CREATE TokenGame object

WHILE window is open:

POLL events:

IF window closed THEN close

IF mouse pressed THEN pass coordinates to game.handleClick

CALL game.makeBotMove()

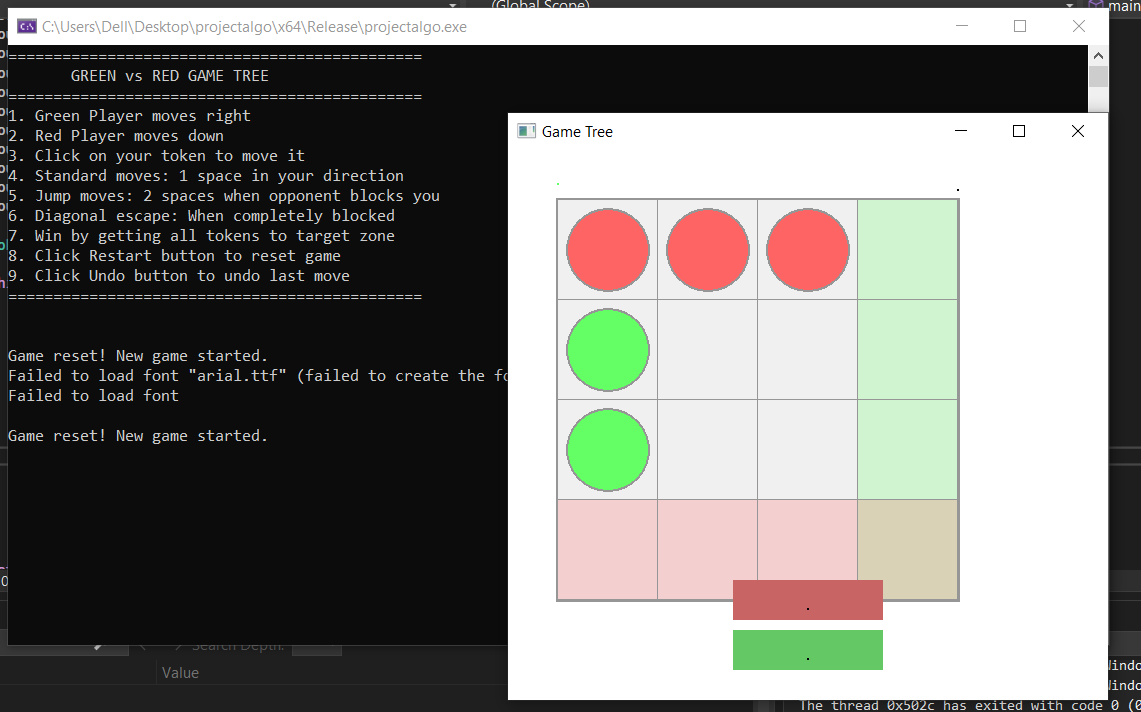
CALL game.draw(window)

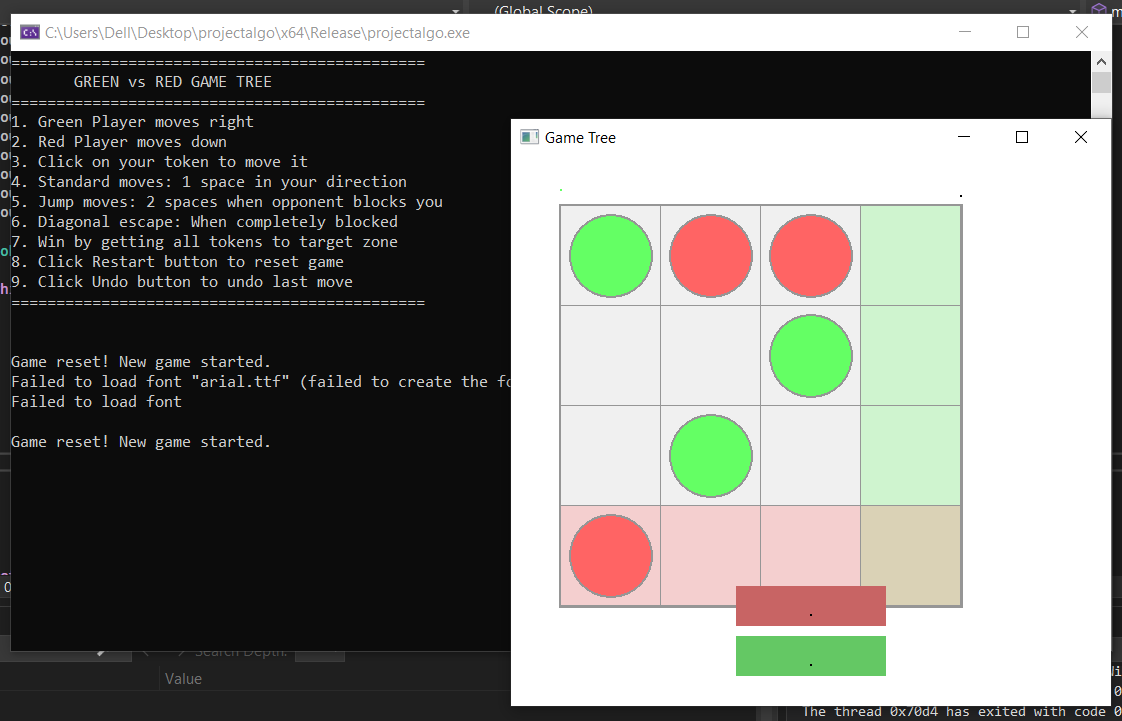
DISPLAY window

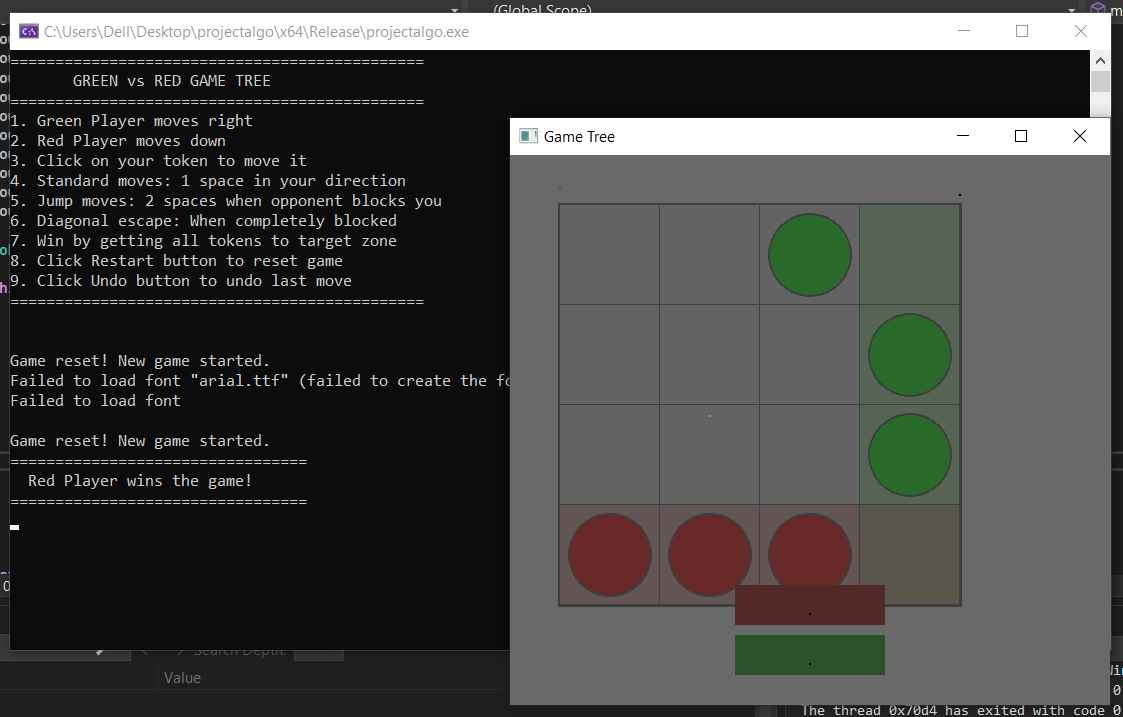
**Explanation of Code:**

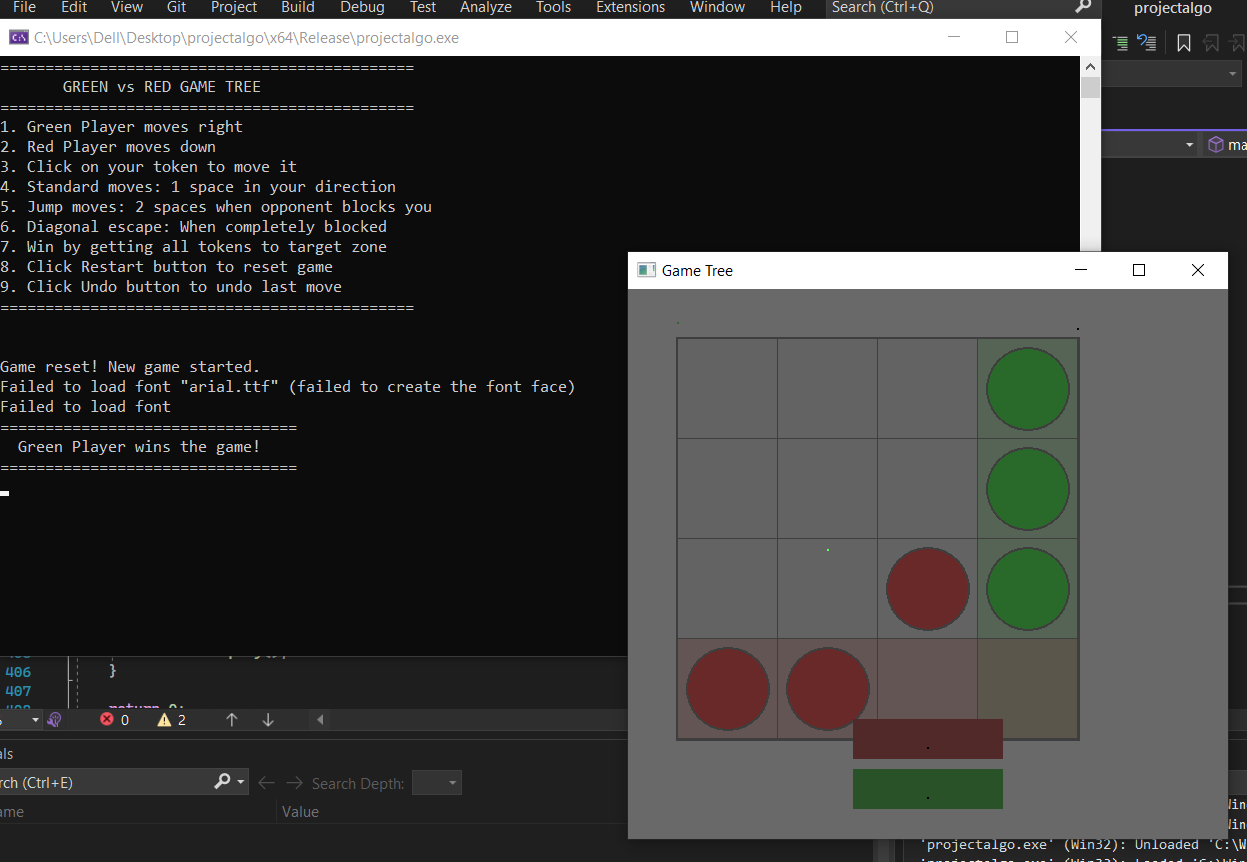
The provided code implements a backtracking algorithm to find a valid path for the token from the starting point to the goal chosen for the grid of 4x4, for example, Player B's bot. The first step is defining a 2-dimensional vector visited to view what cells have already been traversed on the grid so that the algorithm does not hinder itself from using the cell again and running into an infinite loop. A different path vector stores the coordinates generated for the current point of exploration. The pivotal function, “findPathToGoal”, which takes in the current location (row, col) as well as the destination represented by the target goal location (goalRow, goalCol): If the present position matches the goal, the position is appended to the current path and it returns true showing that valid path is found. Otherwise, it marks the cell as visited and seeks more movement based on this array of defined directions: one step to the right, two steps to the right. Each valid step that is within limits and has landed on an unvisited cell will again call the function recursively. If one of these calls returns true, so will this function call, thereby preserving that particular path. If there is no direction towards the goal in those possible directions, it will backtrack by removing the last position from the path vector and return false. In this way, the bot can explore the paths that it could take and backtrack when it finds the dead ends, eventually giving it a valid path from the starting point to the desired goal, if such a path exists.

**Outputs:**

****

****

****

****

**Functions Analysis:**

**1)**

void saveState() {

        GameState state;

        state.playerA = playerA;

        state.playerB = playerB;

        state.isPlayerATurn = isPlayerATurn;

        state.gameOver = gameOver;

        gameHistory.push(state);

        canUndo = true;

    }

* The time complexity of this function is equal O(n) because the n is the size of the vector and the state.PlayerA and state.playerB is copying the vector.
* State.playerA and state.playerB the time complexity for each is O(n) so if we added the together it will be : O(n+n) which is O(n)
* Space complexity is O(n)

**2)**

 bool undoMove() {

        if (gameHistory.empty()) return false;

        GameState previousState = gameHistory.top();

        gameHistory.pop();

        playerA = previousState.playerA;

        playerB = previousState.playerB;

        isPlayerATurn = previousState.isPlayerATurn;

        gameOver = previousState.gameOver;

        winnerAnnounced = false;

        canUndo = !gameHistory.empty();

        return true;

    }

* gameHistory.top() and gameHistory.pop() are both O(1).
* Assigning playerA and playerB from previousState (just like in saveState()) so the time complexity is O(n**)** for both.
* All other assignments are O(1).
* The time complexity is equal to O(n)
* Space complexity is O(n)

**3)**

 bool isOccupied(int row, int col) {

        for (auto& token : playerA)

      if (token.row == row && token.col == col && !token.reachedGoal) return true;

        for (auto& token : playerB)

            if (token.row == row && token.col == col && !token.reachedGoal) return true;

        return false;

    }

First for loop:

* The loop iterates through all the elements of playerA. Since playerA.size() = n, this loop runs n times.
* Each iteration performs a comparison and condition checks, which is O(1).
* So, the time complexity of this loop is O(n).

Second for loop:

* Similarly, the second loop iterates through all the elements of playerB. Since playerB.size() = n, this loop also runs n times.
* The condition checks inside the loop are O(1).
* So, the time complexity of this loop is also O(n).
* and we added them because they are independent and not nested so no need to multiply them .
* the time complexity for each is O(n) so if we added them together it will be : O(n+n) which is O(n)
* Space complexity is O(1) as it doesn’t create any additional data structures

**4)**

void makeMove(int tokenIndex) {

        saveState();

        if (isPlayerATurn) {

            auto& token = playerA[tokenIndex];

            if (token.col == BOARD\_SIZE - 1) {

                token.reachedGoal = true;

                isPlayerATurn = !isPlayerATurn;

                return;

            }

            if (!isOccupied(token.row, token.col + 1)) {

                token.col++;

            }

            else if (token.col + 2 < BOARD\_SIZE && !isOccupied(token.row, token.col + 2)) {

                token.col += 2;

            }

          else if (token.row + 1 < BOARD\_SIZE && !isOccupied(token.row +1, token.col + 1)) {

                token.row++;

                token.col++;

            }

            if (token.col == BOARD\_SIZE - 1) {

                token.reachedGoal = true;

            }

        }

        int aFinished = count\_if(playerA.begin(), playerA.end(), [](const Token& t) { return t.reachedGoal; });

        int bFinished = count\_if(playerB.begin(), playerB.end(), [](const Token& t) { return t.reachedGoal; });

        if (aFinished == BOARD\_SIZE - 1 || bFinished == BOARD\_SIZE - 1) {

            gameOver = true;

        }

        else {

            isPlayerATurn = !isPlayerATurn;

        }

    }

* saveState() so the time complexity of this function(as calculated before) is O(n)
* isOccupied() this function is called three time which the time complexity of it is O(n).
* playerA.begin(), playerA.end() the complexity of them is O(n+n) which is O(n)
* playerB.begin(), playerB.end() the complexity of them is O(n+n) which is O(n)
* So the time complexity of this function will be equal to O(n+n+n+n) which is O(n)
* Space complexity is O(n)

**5)**

void makeBotMove() {

        if (isPlayerATurn || gameOver) return;

        saveState();

        for (auto& token : playerB) {

            if (token.reachedGoal) continue;

            if (token.row + 1 < BOARD\_SIZE && !isOccupied(token.row + 1, token.col)) {

                token.row++;

                if (token.row == BOARD\_SIZE - 1) token.reachedGoal = true;

                isPlayerATurn = true;

                checkGameOver();

                return;

            }

            if (token.row + 2 < BOARD\_SIZE && !isOccupied(token.row + 2, token.col)) {

                token.row += 2;

                if (token.row == BOARD\_SIZE - 1) token.reachedGoal = true;

                isPlayerATurn = true;

                checkGameOver();

                return;

            }

        }

* saveState() so the time complexity of this function (as calculated before) is O(n)
* the time complexity of the first loop is O(n^2) because there is a function call of function isOccupied() and its time complexity is O(n) and the loop itself isO(n) so that means we multiply them together.
* the time complexity of the first loop is O(n^2) because there is a function call of function isOccupied() and its time complexity is O(n) and the loop itself isO(n) so that means we multiply them together.
* The time complexity of checkGameOver() is O(n) (as calculated ).
* So the time complexity of this function is O(n^2)
* Space complexity is O(n)

**6)**

void checkGameOver() {

        int aFinished = count\_if(playerA.begin(), playerA.end(), [](const Token& t) { return t.reachedGoal; });

        int bFinished = count\_if(playerB.begin(), playerB.end(), [](const Token& t) { return t.reachedGoal; });

        if (aFinished == BOARD\_SIZE - 1 || bFinished == BOARD\_SIZE - 1) {

            gameOver = true;

        }

    }

* playerA.begin(), playerA.end() the complexity of them is O(n+n) which is O(n)
* playerB.begin(), playerB.end() the complexity of them is O(n+n) which is O(n).
* So the time complexity of this function is O(n).
* Space complexity is O(1)

**7)**

void handleClick(int x, int y) {

        if (x >= WINDOW\_WIDTH / 2 - 75 && x <= WINDOW\_WIDTH / 2 + 75 &&

            y >= WINDOW\_HEIGHT - 70 && y <= WINDOW\_HEIGHT - 30) {

            resetGame();

            return;

        }

        if (canUndo && x >= WINDOW\_WIDTH / 2 - 75 && x <= WINDOW\_WIDTH / + 75 &&

            y >= WINDOW\_HEIGHT - 120 && y <= WINDOW\_HEIGHT - 80) {

            undoMove();

          return;

        }

        if (gameOver) return;

        int col = (x - 50) / CELL\_SIZE;

        int row = (y - 50) / CELL\_SIZE;

        if (col < 0 || col >= BOARD\_SIZE || row < 0 || row >= BOARD\_SIZE) return;

        for (int i = 0; i < playerA.size(); i++) {

            if (isPlayerATurn && playerA[i].row == row && playerA[i].col == col && !playerA[i].reachedGoal) {

                makeMove(i);

                return;

            }

        }

    }

* The time complexity of resetGame() is O(n)(as calculated).
* The time complexity of undoMove() is O(n)(as calculated before).
* The time complexity of the for loop is O(n) and inside the loop there is a function call inside the loop which is makeMove(i) and the time complexity of it is O(n) (as calculated before). So, we are going to multiply them together.
* So, the time complexity of this function is O(n^2)
* Space complexity is O(n)

**8)**

void resetGame() {

playerA.clear();

playerB.clear();

for (int i = 0; i < BOARD\_SIZE - 1; i++) {

playerA.push\_back({ i, 0 });

playerB.push\_back({ 0, i });

}

isPlayerATurn = true;

gameOver = false;

winnerAnnounced = false;

while (!gameHistory.empty()) gameHistory.pop();

canUndo = false;

cout << "\nGame reset! New game started.\n";

}

* Time complexity of playerA.clear() & playerB.clear() is O(n) in which n is the size of vectors A and B.
* The for loop has a T(n)= 1+(n+1)+n where n is the board size.
* Time complexity of for loop is O(n)
* The while loop runs for n times in which n is the no. of elements in gameHistory
* Time complexity of while loop is O(n)
* Time complexity of the function is O(n+n+n) = O(n)
* Space complexity is O(n)

**9)**

void draw(RenderWindow& window) {

window.clear(Color::White);

window.draw(boardBg);

window.draw(targetZoneA);

window.draw(targetZoneB);

for (int i = 0; i <= BOARD\_SIZE; i++) {

Vertex hLine[] = {

Vertex(Vector2f(50.f, 50.f + i \* CELL\_SIZE), Color(150, 150, 150)),

Vertex(Vector2f(50.f + BOARD\_SIZE \* CELL\_SIZE, 50.f + i \* CELL\_SIZE), Color(150, 150, 150))

};

window.draw(hLine, 2, Lines);

Vertex vLine[] = {

Vertex(Vector2f(50.f + i \* CELL\_SIZE, 50.f), Color(150, 150, 150)),

Vertex(Vector2f(50.f + i \* CELL\_SIZE, 50.f + BOARD\_SIZE \* CELL\_SIZE), Color(150, 150, 150))

};

window.draw(vLine, 2, Lines);

}

for (auto& token : playerA) {

tokenShape.setFillColor(Color(100, 255, 100));

tokenShape.setPosition(50 + token.col \* CELL\_SIZE + CELL\_SIZE / 10.f,

50 + token.row \* CELL\_SIZE + CELL\_SIZE / 10.f);

window.draw(tokenShape);

}

for (auto& token : playerB) {

tokenShape.setFillColor(Color(255, 100, 100));

tokenShape.setOutlineColor(Color(150, 150, 150));

tokenShape.setPosition(50 + token.col \* CELL\_SIZE + CELL\_SIZE / 10.f,

50 + token.row \* CELL\_SIZE + CELL\_SIZE / 10.f);

window.draw(tokenShape);

}

turnText.setString(isPlayerATurn ? "Player A's Turn (Green)" : "Player B's Turn (Red)");

turnText.setFillColor(isPlayerATurn ? Color(100, 255, 100) : Color(200, 50, 50));

window.draw(turnText);

int aScore = count\_if(playerA.begin(), playerA.end(), [](const Token& t) { return t.reachedGoal; });

int bScore = count\_if(playerB.begin(), playerB.end(), [](const Token& t) { return t.reachedGoal; });

scoreText.setString("Green: " + to\_string(aScore) + " - Red: " + to\_string(bScore));

scoreText.setFillColor(Color::Black);

window.draw(scoreText);

window.draw(restartButton);

Text restartText("Restart", font, 24);

restartText.setPosition(WINDOW\_WIDTH / 2 - restartText.getLocalBounds().width / 2, WINDOW\_HEIGHT - 65);

restartText.setFillColor(Color::Black);

window.draw(restartText);

window.draw(undoButton);

Text undoText("Undo Move", font, 24);

undoText.setPosition(WINDOW\_WIDTH / 2 - undoText.getLocalBounds().width / 2, WINDOW\_HEIGHT - 115);

undoText.setFillColor(Color::Black);

window.draw(undoText);

if (gameOver) {

RectangleShape overlay(Vector2f(WINDOW\_WIDTH, WINDOW\_HEIGHT));

overlay.setFillColor(Color(0, 0, 0, 150));

window.draw(overlay);

if (aScore == BOARD\_SIZE - 1) {

gameOverText.setString("Green Player Wins!");

gameOverText.setFillColor(Color(100, 255, 100));

}

else {

gameOverText.setString("Red Player Wins!");

gameOverText.setFillColor(Color(255, 100, 100));

}

if (!winnerAnnounced) {

cout << "=================================\n";

cout << (aScore == BOARD\_SIZE - 1 ? " Green Player wins the game!\n" : " Red Player wins the game!\n");

cout << "=================================\n";

winnerAnnounced = true;

}

window.draw(gameOverText);

}

}

* The function starts with static drawing instructions that run for 1 time each.
* Time complexity of static drawing instructions is O(1).
* First for loop runs n + 1 times.
* The body of the loop includes two instructions for drawing which run 1 time each.
* T(n) = 1+1+(n+1), so the time complexity is O(n).
* Second and third for loops that draw token for player A and B both run n+1 times.
* Time complexity of second and third loops is O(n).
* Time complexity of the function is O(n+n+1) = O(n).
* Space complexity is 1, although it loops on vectors of n size but no new memory is allocated.

**10)**

bool findPathToGoal(Token token, vector<vector<bool>>& visited, vector<pair<int, int>>& path) {

if ((isPlayerATurn && token.col == BOARD\_SIZE - 1) || (!isPlayerATurn && token.row == BOARD\_SIZE - 1)) {

path.emplace\_back(token.row, token.col);

return true;

}

visited[token.row][token.col] = true;

path.emplace\_back(token.row, token.col);

vector<pair<int, int>> directions;

if (isPlayerATurn) {

directions = { {0,1}, {0,2}, {1,1} };

}

else {

directions = { {1,0}, {2,0}, {1,1} };

}

for (auto& dir : directions) {

int newRow = token.row + dir.first;

int newCol = token.col + dir.second;

if (newRow >= 0 && newRow < BOARD\_SIZE && newCol >= 0 && newCol < BOARD\_SIZE &&

!isOccupied(newRow, newCol) && !visited[newRow][newCol]) {

Token next = { newRow, newCol };

if (findPathToGoal(next, visited, path)) return true;

}

}

path.pop\_back();

return false;

}

* The function starts by checking if the current token is at the goal position, if yes it marks the cell as visited and adds it to the path.
* Time complexity of these instructions is O(1).
* The function constructs a direction list based on player turn.
* Direction list construction takes is executed 1 time, so its time complexity is O(1).
* The function loops through at most 3 directions for possible moves, each call has a time complexity of O(1).
* If a valid move is found, the function returns true.
* If no move works, it performs backtracking by undoing the moves.
* Time complexity of backtracking is O(1).
* In the worst case, the function explores up to 3 paths per cell in an n × n table, T(n)= 3^(n^2).
* Time complexity of this recursive call is exponential: O(3^(n^2)).
* Space complexity is O(n^2)

**11)**

void testBacktrackingForFirstToken() {

if (isPlayerATurn && !playerA.empty()) {

vector<vector<bool>> visited(BOARD\_SIZE, vector<bool>(BOARD\_SIZE, false));

vector<pair<int, int>> path;

if (findPathToGoal(playerA[0], visited, path)) {

cout << "Path for Green token to reach goal:\n";

for (auto& p : path)

cout << "(" << p.first << "," << p.second << ") -> ";

cout << "Goal\n";

}

else {

cout << "No path found for Green token\n";

}

}

}

};

* A 2D array (visited) is initialized to keep track of visited positions on the board.
* Time complexity of initializing this array is O(n²), where n is the board size.
* A vector (path) is initialized to store the valid path if it is found.
* Time complexity of this initialization is O(1).
* The findPathToGoal() function is called in which its time complexity is O(3^(n^2)) (as calculated before).
* There is a for loop for checking if path is found or not.
* The path can be in worst case n^2 in length, so the time complexity of this code is O(n^2)
* Time Complexity of the function is O(3^(n^2)) as backtracking represents the upper bound.
* Space complexity is O(n^2)

**12)**

int main() {

RenderWindow window(VideoMode(WINDOW\_WIDTH, WINDOW\_HEIGHT), "Green vs Red Token Game");

cout << "==============================================\n";

cout << " GREEN vs RED GAME TREE \n";

cout << "==============================================\n";

cout << "1. Green Player moves right\n";

cout << "2. Red Player moves down\n";

cout << "3. Click on your token to move it\n";

cout << "4. Standard moves: 1 space in your direction\n";

cout << "5. Jump moves: 2 spaces when opponent blocks you\n";

cout << "6. Diagonal escape: When completely blocked\n";

cout << "7. Win by getting all tokens to target zone\n";

cout << "8. Click Restart button to reset game\n";

cout << "9. Click Undo button to undo last move\n";

cout << "==============================================\n\n";

TokenGame game;

while (window.isOpen()) {

Event event;

while (window.pollEvent(event)) {

if (event.type == Event::Closed)

window.close();

if (event.type == Event::MouseButtonPressed)

game.handleClick(event.mouseButton.x, event.mouseButton.y);

}

game.makeBotMove();

game.draw(window);

window.display();

}

return 0;

}

* The time complexity of the main loop (while (window.isOpen())) depends on how long the game runs, so it's O(t) where t is the run time of the game.
* Time complexity of handleClick() is O(n^2) (as calculate before).
* The Time complexity of makeBotMove() is O(n^2) (as calculate before).
* Time complexity of draw() is O(n) (as calculate before).
* In each iteration, the total time complexity is O(n^2 + n^2 +n) = O(n^2).
* Time complexity of the main function is O(t \* n^2), where t is the runtime of the game and n is the size of the game.
* Space complexity is O(n^2)