**משחק איקס עיגול מורחב – מסמך SRS**

**אלעד ארמוני 206157323 | אופק בג'רנו 318435195 | בר שניידר 208183061 | יוני בנינו 318645686**

**לינקים (יש ללחוץ על האייקונים)**   
מצגת PPT סרטון הצגת המצגת GitHub README קישור למשחק סרטון הצגת המשחק

[תמונה שמכילה אומנות קליפיפם, סמל, צללית

התיאור נוצר באופן אוטומטי](https://github.com/EladArmoni/Tic-Tac-Toe)[תמונה שמכילה לוגו, גרפיקה, סמל, גופן

התיאור נוצר באופן אוטומטי](https://www.canva.com/design/DAGL8LkWOmM/g4n7Lr7yPUdn0x0wUnTDUA/edit?utm_content=DAGL8LkWOmM&utm_campaign=designshare&utm_medium=link2&utm_source=sharebutton)[תמונה שמכילה שרטוט, סמל, גופן, אומנות קליפיפם

התיאור נוצר באופן אוטומטי](https://www.youtube.com/watch?v=HIGbxwxMcqM)[](https://github.com/EladArmoni/Tic-Tac-Toe/blob/main/README.md)[תמונה שמכילה סמל, עיצוב

התיאור נוצר באופן אוטומטי](https://tic-tac-toe-iota-two-32.vercel.app/)[תמונה שמכילה לוגו, סמל, אדום, גרפיקה

התיאור נוצר באופן אוטומטי](https://youtu.be/KPMyYc7iJF4)

**סיכום כללי**  
משחק "איקס עיגול מורחב" הוא משחק איקס עיגול, המשוחק על לוח 3X3 עם תוספת של 3 משבצות נוספות מסביב ללוח במקומות רנדומליים. המשחק מכיל אפשרויות נוספות של הטלת פצצות לאורך שורה או עמודה בלוח ומשתמש באלגוריתם בינה מלאכותית alpha beta minimax לחישוב הצעדים הטובים ביותר.

**הבעיות והפתרונות במשחק**במשחק זה היו שתי בעיות עיקריות:

1. הטלת הפצצות בלוח עם משבצות נוספות מסביב:
   * הבעיה: הלוח בנוי כך שיש 3 משבצות רנדומליות מסביב ללוח המשחק, ולכן יש לבצע חישוב מיוחד כדי לדעת היכן השחקן הטיל פצצה ובאילו משבצות יש למחוק את הסימונים.
   * הפתרון: יצרנו לוח בגודל 5X5 שבו הלוח הפנימי המוצג לשחקן הוא בגודל 3X3 והשורות והעמודות העוטפות את הלוח הקטן הן בלתי נראות למשתמש. מתוך השורות והעמודות העוטפות, הגרלנו בצורה רנדומלית 3 משבצות. המשבצות הנראות לשחקן מסומנות במטריצה על ידי המספר 0, והמשבצות הלא נראות מסומנות על ידי 2-. במהלך המשחק, אנחנו מתייחסים רק למשבצות המסומנות ב-0. מאחר וישנה מטריצה של 5X5 מאחורי הקלעים, קל ונוח להטיל פצצות לאורך שורה או עמודה מסוימת.
2. שימוש באלגוריתם בינה מלאכותית Alpha-Beta Minimax בצעדי המחשב:
   * הבעיה: יצירת סוכן חכם (המחשב) שמחשב את כל הצעדים האפשריים של היריב (עד לעומק מסוים) באמצעות אלגוריתם בינה מלאכותית זה, וכך יודע איפה כדאי לו לשים את הסימון שלו כדי לנצח את היריב. הקושי היה לבחור את העומק היעיל כך שהאלגוריתם יעבוד בצורה מהירה ויעילה, ויוכל לנצח את המשחק. היה צורך להתחשב במשבצות הנוספות מסביב ללוח במהלך הפעולה של הסוכן החכם וגם בהטלת הפצצות כצעד שיכול להוביל אולי לניצחון.
   * הפתרון: הגבלנו את האלגוריתם לעומק של 3 צעדים, והוספנו התחשבות במהלכים הכוללים הטלת פצצות. בצורה זו, האלגוריתם גם מהיר וגם יעיל בניצחון במשחק.

**קוד המשחק**

import React, { useState, useEffect } from "react";

import styled, { keyframes } from "styled-components";

import { FontAwesomeIcon } from "@fortawesome/react-fontawesome";

import { faBomb } from "@fortawesome/free-solid-svg-icons";

import { flash } from "react-animations";

import Player from "./Player";

// Styled components for the application

const AppContainer = styled.div`

  display: flex;

  flex-direction: column;

  align-items: center;

  justify-content: center;

  height: 100vh;

  background-color:#4d04a5;

  background: url("/bg3.jpeg");

      background-size:     cover;

    background-repeat:   no-repeat;

    background-position: center center;

`;

const BoardContainer = styled.div`

  display: flex;

  flex-direction: column;

  align-items: center;

  margin-top: 20px;

`;

const BombControlsContainer = styled.div`

  margin-top: 20px;

`;

const BombIcon = styled(FontAwesomeIcon)`

  cursor: pointer;

  margin: 0 10px;

  font-size: 24px;

  &:hover {

    color: ${({ type }) => (type === "row" ? "#ff3bad7f" : "#3bffd57b")};

  }

  &.disabled {

    pointer-events: none;

    color: #d0d0d0;

  }

`;

const GameBoard = styled.div`

  display: grid;

  grid-template-columns: repeat(5, 80px);

`;

const CellButton = styled.button`

  width: 80px;

  height: 80px;

  font-size: 38px;

  font-weight: 700;

  display: flex;

  align-items: center;

  justify-content: center;

  color: ${({ value }) =>

    value === 1 ? "white" : value === -1 ? "white" : "white"};

  background-color: ${({ rowIndex, colIndex, selectedRow, selectedCol }) =>

    rowIndex === selectedRow ? "#320066" : colIndex === selectedCol ? "#320066": "#320066"};

  background: ${({ rowIndex, colIndex, selectedRow, selectedCol }) =>

    rowIndex === selectedRow

      ? "#320066 url('/bomb.png') no-repeat center/contain"

      : colIndex === selectedCol

      ? "#320066 url('/bomb.png') no-repeat center/contain"

      : "#320066"};

  border: 5px solid #73f2fa;

  transition: background-color 0.3s ease;

  visibility: ${({ value }) => (value === -2 ? "hidden" : "visible")};

  &:hover {

    background-color: ${({ rowIndex, colIndex, selectedRow, selectedCol }) =>

      rowIndex === selectedRow || colIndex === selectedCol

        ? "#e0e0e0"

        : "#e0e0e0"};

  }

  &:disabled {

    background-color: #d0d0d0;

    cursor: not-allowed;

  }

`;

const bombAnimation = keyframes`${flash}`;

const NewGameButton = styled.button`

  font-size: 16px;

  padding: 10px 20px;

  margin-top: 20px;

  background-color: #fad969; /\* Green background \*/

  border: none;

  color: #1f1f1f; /\* White text \*/

  font-weight:600;

  text-align: center;

  text-decoration: none;

  display: inline-block;

  transition-duration: 0.4s;

  cursor: pointer;

  border-radius: 8px;

  &:hover {

    background-color: black; /\* Darker green on hover \*/

    color: #fff; /\* Darker green on hover \*/

  }

`;

const App = () => {

  const initialPlayerBombs = { row: 1, col: 1 }; // Initial number of bombs for player

  const initialAiBombs = { row: 1, col: 1 }; // Initial number of bombs for AI

  const [board, setBoard] = useState(null); // Board state

  const [bombs, setBombs] = useState(null); // Bombs state

  const [conditions, setConditions] = useState([]); // Win conditions

  const [playerBombs, setPlayerBombs] = useState(initialPlayerBombs); // Player bombs count

  const [aiBombs, setAiBombs] = useState(initialAiBombs); // AI bombs count

  const [turn, setTurn] = useState("Player"); // Current turn

  const [message, setMessage] = useState("Player's Turn"); // Game message

  const [selectedRow, setSelectedRow] = useState(-1); // Selected row for bomb

  const [selectedCol, setSelectedCol] = useState(-1); // Selected column for bomb

  const [winner, setWinner] = useState(null); // Winner state

  const [rowsSet, setRowsSet] = useState([]); // Rows with available cells

  const [colsSet, setColsSet] = useState([]); // Columns with available cells

  useEffect(() => {

    newGame();

  }, []);

  useEffect(() => {

    if (board != null) {

      checkGameResult();

    }

  }, [board]);

  // We use -2 for unavailable cells, 0 for available cells - cause we add 3 random cell around the board so this cells will be with 0 value in new game

  async function createBoard() {

    const initialBoard = [

      [-2, -2, -2, -2, -2],

      [-2, 0, 0, 0, -2],

      [-2, 0, 0, 0, -2],

      [-2, 0, 0, 0, -2],

      [-2, -2, -2, -2, -2],

    ];

    const uniqueCells = new Set();

    while (uniqueCells.size < 3) {

      const row = Math.floor(Math.random() \* 5);

      const col = Math.floor(Math.random() \* 5);

      // Ensure the cell is currently -2 and not already chosen

      if (initialBoard[row][col] === -2) {

        initialBoard[row][col] = 0; // Change -2 to 0

        uniqueCells.add(`${row},${col}`);

      }

    }

    // Filter out rows that contain only -2 values

    let rows = [];

    initialBoard.filter((row, index) => {

      if (row.some((cell) => cell !== -2)) rows.push(index);

    });

    setRowsSet(rows);

    // Transpose the filtered board to filter columns

    const transpose = (matrix) =>

      matrix[0].map((col, i) => matrix.map((row) => row[i]));

    const transposedBoard = transpose(initialBoard);

    // Filter out columns that contain only -2 values

    let cols = [];

    transposedBoard.filter((col, index) => {

      if (col.some((cell) => cell !== -2)) cols.push(index);

    });

    setColsSet(cols);

    setBoard(initialBoard);

    setBombs(initialBoard);

    return initialBoard;

  }

  // Function to compute new conditions based on the updated board

  async function computeNewConditions(currBoard) {

    const size = currBoard.length;

    const newConditions = [];

    // Check rows for conditions

    for (let row = 0; row < size; row++) {

      for (let i = 0; i <= size - 3; i++) {

        // Check if the current cells and the next two cells are valid

        if (

          currBoard[row][i] === 0 && // Current cell

          currBoard[row][i + 1] === 0 && // Next cell

          currBoard[row][i + 2] === 0 // Next next cell

        ) {

          newConditions.push([

            [row, i],

            [row, i + 1],

            [row, i + 2],

          ]);

        }

      }

    }

    // Check columns for conditions

    for (let col = 0; col < size; col++) {

      for (let i = 0; i <= size - 3; i++) {

        // Check if the current cells and the next two cells are valid

        if (

          currBoard[i][col] === 0 && // Current cell

          currBoard[i + 1][col] === 0 && // Next cell

          currBoard[i + 2][col] === 0 // Next next cell

        ) {

          newConditions.push([

            [i, col],

            [i + 1, col],

            [i + 2, col],

          ]);

        }

      }

    }

    // Check diagonals for conditions

    for (let i = 0; i <= size - 3; i++) {

      for (let j = 0; j <= size - 3; j++) {

        // Check diagonal from top-left to bottom-right

        if (

          currBoard[i][j] === 0 && // Current cell

          currBoard[i + 1][j + 1] === 0 && // Next cell

          currBoard[i + 2][j + 2] === 0 // Next next cell

        ) {

          newConditions.push([

            [i, j],

            [i + 1, j + 1],

            [i + 2, j + 2],

          ]);

        }

        // Check diagonal from top-right to bottom-left

        if (

          currBoard[i][j + 2] === 0 && // Current cell

          currBoard[i + 1][j + 1] === 0 && // Next cell

          currBoard[i + 2][j] === 0 // Next next cell

        ) {

          newConditions.push([

            [i, j + 2],

            [i + 1, j + 1],

            [i + 2, j],

          ]);

        }

      }

    }

    setConditions(newConditions);

  }

  // Initialize the Tic-Tac-Toe board and bombs

  async function newGame() {

    setSelectedRow(-1);

    setSelectedCol(-1);

    const currBoard = await createBoard();

    await computeNewConditions(currBoard);

    setPlayerBombs(initialPlayerBombs);

    setAiBombs(initialAiBombs);

    setTurn("Player");

    setMessage("Player's Turn");

    setWinner(null); // Reset winner state

    enableBoard();

  }

  //update the board with the current cells and states

  const updateBoard = (board, bombs, playerBombs, aiBombs) => {

    setBoard([...board]);

    setBombs([...bombs]);

    setPlayerBombs({ ...playerBombs });

    setAiBombs({ ...aiBombs });

  };

  //handle player click on cell and update the board

  const handleCellClick = (row, col) => {

    if (turn === "Player" && board[row][col] === 0 && bombs[row][col] === 0) {

      setMove(board, row, col, 1);

      updateBoard(board, bombs, playerBombs, aiBombs);

      if (!gameWon(board) && !boardFull()) {

        setTurn("AI");

        setMessage("AI's Turn");

        setTimeout(() => handleAIMove(), 500);

      } else {

        checkGameResult();

      }

    }

  };

  //handle AI move with the AI algorithm

  const handleAIMove = () => {

    const newBoard = [...board];

    const newBombs = [...bombs];

    const newAiBombs = { ...aiBombs };

    const newPlayerBombs = { ...playerBombs };

    //AI use abminimax algorithm to find the best step to do

    const result = abminimax(

      newBoard,

      newBombs,

      3, // Depth of the minimax algorithm

      -Infinity, // Alpha value for alpha-beta pruning

      Infinity, // Beta value for alpha-beta pruning

      -1, // Player value (-1 for AI, 1 for Player)

      newPlayerBombs, // Number of bombs available to the player

      newAiBombs // Number of bombs available to the AI

    );

    // Check if the AI move is valid

    if (result[0] !== -1 && result[1] !== -1) {

      // Check if the result includes a bomb move

      if (result[3]) {

        const [bombType, index] = result[3];

        // Detonate the bomb based on the type (row or column) and index

        detonateBomb(newBoard, newBombs, index, bombType);

        // Update the AI bomb count based on the type of bomb used

        bombType === "col"

          ? setAiBombs({ ...aiBombs, col: aiBombs["col"] - 1 })

          : setAiBombs({ ...aiBombs, row: aiBombs["row"] - 1 });

      } else {

        // Make the move on the board for the AI

        setMove(newBoard, result[0], result[1], -1);

      }

      // Check if the game has been won or if the board is full

      if (!gameWon(newBoard) && !boardFull(newBoard)) {

        // If the game is not over, switch turn back to the player

        setTurn("Player");

        setMessage("Player's Turn");

      } else {

        // If the game is over, check the game result

        checkGameResult(newBoard);

      }

    }

  };

  const rowbombbutton = document.getElementById("rowBombButton")

  const colbombbutton = document.getElementById("colBombButton")

  useEffect(function(){

    if(rowbombbutton && colbombbutton){

        // Disable row bomb button if no row is selected

        if(selectedRow === -1){

          document.getElementById("rowBombButton").classList.add("disabled");

        }

        // Enable row bomb button if a row is selected and player has bombs left

        else if (selectedRow !== -1 && playerBombs.row > 0){

          document.getElementById("rowBombButton").classList.remove("disabled");

        }

        // Disable column bomb button if no column is selected

        if(selectedCol === -1){

          document.getElementById("colBombButton").classList.add("disabled");

        }

        // Enable column bomb button if a column is selected and player has bombs left

        else if (selectedCol !== -1 && playerBombs.col > 0){

          document.getElementById("colBombButton").classList.remove("disabled");

        }

    }

  },[selectedCol,selectedRow,rowbombbutton,colbombbutton, playerBombs])

  const placeBomb = (type) => {

    if (type === "row" && playerBombs.row > 0) {

      // Detonate row bomb and update player bombs count

      detonateBomb(board, bombs, selectedRow, "row");

      setPlayerBombs({ ...playerBombs, row: playerBombs.row - 1 });

      if (playerBombs.row - 1 === 0) {

        document.getElementById("rowBombButton").classList.add("disabled");

      }

    } else if (type === "col" && playerBombs.col > 0) {

      // Detonate column bomb and update player bombs count

      detonateBomb(board, bombs, selectedCol, "col");

      setPlayerBombs({ ...playerBombs, col: playerBombs.col - 1 });

      if (playerBombs.col - 1 === 0) {

        document.getElementById("colBombButton").classList.add("disabled");

      }

    }

    // Switch turn to AI and update message

    setTurn("AI");

    setMessage("AI's Turn");

    setTimeout(() => handleAIMove(), 500);

  };

  // Check if either player has won the game

  const gameWon = (newBoard) => {

    return winningPlayer(newBoard, 1) || winningPlayer(newBoard, -1);

  };

  // Check if the board is full

  const boardFull = () => {

    return board.flat().every((cell) => cell !== 0);

  };

  const checkGameResult = () => {

    if (winningPlayer(board, 1)) {

      setMessage("Player has won!");

      setWinner("Player"); // Set winner state

      disableBoard();

    } else if (winningPlayer(board, -1)) {

      setMessage("AI has won!");

      setWinner("AI"); // Set winner state

      disableBoard();

    } else if (boardFull()) {

      setMessage("The game is a draw.");

      setWinner("Draw"); // Set winner state

      disableBoard();

    }

  };

  // Disable all cells on the board and bomb buttons

  const disableBoard = () => {

    const cells = document.querySelectorAll('button:not([value="new game"])'); // Exclude New Game button

    cells.forEach((cell) => {

      cell.disabled = true;

    });

    document.getElementById("rowBombButton").classList.add("disabled");

    document.getElementById("colBombButton").classList.add("disabled");

  };

  // Enable all cells on the board

  const enableBoard = () => {

    const cells = document.querySelectorAll('button:not([value="new game"])'); // Exclude New Game button

    cells.forEach((cell) => {

      cell.disabled = false;

    });

  };

  // Check if the given player has won based on conditions

  const winningPlayer = (newBoard, player) => {

    return conditions.some((condition) =>

      condition.every(([x, y]) => newBoard[x][y] === player)

    );

  };

  // Set the move for the given player on the board

  const setMove = (newBoard, x, y, player) => {

    newBoard[x][y] = player;

    setBoard(newBoard);

  };

  // Detonate row bomb and update board and bombs states

  const detonateBomb = (newBoard, newBombs, index, bomb\_type) => {

    if (bomb\_type === "row") {

      for (let i = 0; i < 5; i++) {

        if (newBoard[index][i] === -2) {

          newBoard[index][i] = -2;

          newBombs[index][i] = -2;

        } else {

          newBoard[index][i] = 0;

          newBombs[index][i] = 0;

        }

        addBombAnimation(index, i); // Add animation to the bomb

      }

    } else if (bomb\_type === "col") {

      // Detonate column bomb and update board and bombs states

      for (let i = 0; i < 5; i++) {

        if (newBoard[i][index] === -2) {

          newBoard[i][index] = -2;

          newBombs[i][index] = -2;

        } else {

          newBoard[i][index] = 0;

          newBombs[i][index] = 0;

        }

        addBombAnimation(i, index); // Add animation

      }

    }

    setBoard(newBoard);

    setBombs(newBombs);

    setSelectedCol(-1)

    setSelectedRow(-1)

  };

  // Add animation to cell

  const addBombAnimation = (row, col) => {

    const cell = document.getElementById(`cell-${row}-${col}`);

    if (cell) {

      cell.classList.add("bomb-animation");

      setTimeout(() => {

        cell.classList.remove("bomb-animation");

      }, 1000); // Duration of animation

    }

  };

  // Heuristic function to evaluate the board

  const heuristic = (newBoard) => {

    if (winningPlayer(newBoard, 1)) {

      return 1;

    } else if (winningPlayer(newBoard, -1)) {

      return -1;

    } else {

      return 0;

    }

  };

  // Alpha-beta minimax algorithm to determine the best move

  const abminimax = (

    newBoard, // Current state of the board

    newBombs, // Current state of the bombs

    depth, // Depth of the minimax algorithm

    alpha, // Alpha value for alpha-beta pruning

    beta,  // Beta value for alpha-beta pruning

    player, // Current player (-1 for AI, 1 for Player)

    newPlayerBombs, // Number of bombs available to the player

    newAiBombs // Number of bombs available to the AI

  ) => {

    let row = -1;

    let col = -1;

    let bombMove = null;

    // Base case: if depth is 0 or the game is won

    if (depth === 0 || gameWon(newBoard)) {

      return [row, col, heuristic(newBoard)];

    } else {

      for (let cell of blanks(newBoard)) {

        setMove(newBoard, cell[0], cell[1], player); // Make the move

        let score = abminimax(

          newBoard,

          newBombs,

          depth - 1,

          alpha,

          beta,

          -player,

          newPlayerBombs,

          newAiBombs

        )[2]; // Recursively call minimax

        newBoard[cell[0]][cell[1]] = 0; // Undo the move

        if (player === 1) { // Maximizing player

          if (score > alpha) {

            alpha = score;

            row = cell[0];

            col = cell[1];

          }

        } else { // Minimizing player

          if (score < beta) {

            beta = score;

            row = cell[0];

            col = cell[1];

          }

        }

        if (alpha >= beta) break; // Alpha-beta pruning

      }

      // Consider using bombs for the player

      if (player === 1 && newPlayerBombs.row > 0) {

        for (let i = 0; i < 5; i++) {

          if (

            newBombs[i].some((bomb, j) => bomb === 0 && newBoard[i][j] !== 0)

          ) {

            const tempRow = [...newBoard[i]];

            detonateBomb(newBoard, newBombs, i, "row"); // Detonate row bomb

            let score = abminimax(

              newBoard,

              newBombs,

              depth - 1,

              alpha,

              beta,

              -player,

              { row: newPlayerBombs.row - 1, col: newPlayerBombs.col },

              newAiBombs

            )[2];

            newBoard[i] = tempRow; // Undo the bomb

            if (score > alpha) {

              alpha = score;

              bombMove = ["row", i];

              if (alpha >= beta) break;

            }

          }

        }

      }

      if (player === 1 && newPlayerBombs.col > 0) {

        for (let j = 0; j < 5; j++) {

          if (newBombs.some((row, i) => row[j] === 0 && newBoard[i][j] !== 0)) {

            const tempCol = newBoard.map((row) => row[j]);

            detonateBomb(newBoard, newBombs, j, "col"); // Detonate column bomb

            let score = abminimax(

              newBoard,

              newBombs,

              depth - 1,

              alpha,

              beta,

              -player,

              { row: newPlayerBombs.row, col: newPlayerBombs.col - 1 },

              newAiBombs

            )[2];

            for (let i = 0; i < 5; i++) newBoard[i][j] = tempCol[i]; // Undo the bomb

            if (score > alpha) {

              alpha = score;

              bombMove = ["col", j];

              if (alpha >= beta) break;

            }

          }

        }

      }

      // Consider using bombs for the AI

      if (player === -1 && newAiBombs.row > 0) {

        for (let i = 0; i < 5; i++) {

          if (

            newBombs[i].some((bomb, j) => bomb !== -2 && newBoard[i][j] !== 0)

          ) {

            const tempRow = [...newBoard[i]];

            detonateBomb(newBoard, newBombs, i, "row");

            let score = abminimax(

              newBoard,

              newBombs,

              depth - 1,

              alpha,

              beta,

              -player,

              newPlayerBombs,

              { row: newAiBombs.row - 1, col: newAiBombs.col }

            )[2];

            newBoard[i] = tempRow;

            if (score < beta) {

              beta = score;

              bombMove = ["row", i];

              if (alpha >= beta) break;

            }

          }

        }

      }

      if (player === -1 && newAiBombs.col > 0) {

        for (let j = 0; j < 5; j++) {

          if (

            newBombs.some((row, i) => row[j] !== -2 && newBoard[i][j] !== 0)

          ) {

            const tempCol = newBoard.map((row) => row[j]);

            detonateBomb(newBoard, newBombs, j, "col");

            let score = abminimax(

              newBoard,

              newBombs,

              depth - 1,

              alpha,

              beta,

              -player,

              newPlayerBombs,

              { row: newAiBombs.row, col: newAiBombs.col - 1 }

            )[2];

            for (let i = 0; i < 5; i++) newBoard[i][j] = tempCol[i];

            if (score < beta) {

              beta = score;

              bombMove = ["col", j];

              if (alpha >= beta) break;

            }

          }

        }

      }

      return player === 1 ? [row, col, alpha] : [row, col, beta, bombMove];

    }

  };

  // Function to find all blank cells on the board

  const blanks = (newBoard) => {

    const moves = [];

    for (let i = 0; i < newBoard.length; i++) {

      for (let j = 0; j < newBoard[0].length; j++) {

        if (newBoard[i][j] === 0) {

          moves.push([i, j]);

        }

      }

    }

    return moves;

  };

  return (

    <>

    <Player player={"AI Agent"} rowbombs={aiBombs?.row} colbombs={aiBombs?.col} side={"left"}/>

    <Player player={"Player"} rowbombs={playerBombs?.row} colbombs={playerBombs?.col} side={"right"} />

    <AppContainer>

      <BoardContainer>

      <div className="turn">{message}</div>

        <GameBoard>

          {board !== null &&

            board.length === 5 &&

            board.map((row, rowIndex) =>

              row.map((cell, colIndex) => (

                <CellButton

                  key={`${rowIndex}-${colIndex}`}

                  onClick={() => {

                    handleCellClick(rowIndex, colIndex);

                  }}

                  value={cell}

                  rowIndex={rowIndex}

                  colIndex={colIndex}

                  selectedRow={selectedRow}

                  selectedCol={selectedCol}

                  id={`cell-${rowIndex}-${colIndex}`}

                  className={

                    cell === -2

                      ? "hidden"

                      : bombs[rowIndex][colIndex] === 1

                      ? "bomb"

                      : ""

                  }

                  disabled={winner !== null}

                >

                  {cell === 1 && "X"}

                  {cell === -1 && "O"}

                  {bombs[rowIndex][colIndex] === 1}

                </CellButton>

              ))

            )}

        </GameBoard>

        <BombControlsContainer>

          <div>

            <select

              id="rowBombSelect"

              value={selectedRow}

              onChange={(e) => setSelectedRow(parseInt(e.target.value))}

              disabled={winner !== null}

            >

            <option value={-1}  > Choose Row</option>

              {rowsSet.map((index) => (

                <option key={index} value={index}>

                  Row {index + rowsSet.includes(0)}

                </option>

              ))}

            </select>

            <BombIcon

              id="rowBombButton"

              icon={faBomb}

              onClick={() => placeBomb("row")}

              type="row"

            />

          </div>

          <div>

            <select

              id="colBombSelect"

              value={selectedCol}

              onChange={(e) => setSelectedCol(parseInt(e.target.value))}

              disabled={winner !== null}

            >

                          <option value={-1} > Choose Column</option>

              {colsSet.map((index) => (<option key={index} value={index}>

                Column {index + colsSet.includes(0)}

              </option>))}

            </select>

            <BombIcon

              id="colBombButton"

              icon={faBomb}

              onClick={() => placeBomb("col")}

              type="col"

            />

          </div>

        </BombControlsContainer>

        <NewGameButton value="new game" onClick={() => newGame()}>

          New Game

        </NewGameButton>

      </BoardContainer>

    </AppContainer>

    </>

  );

};

export default App;