Faculty of Engineering and Technology

Electrical and Computer Engineering Department

ARTIFICIAL INTELLIGENCE – ENCS3340

Project # 1

Magnetic Cave

Prepared by:

Osaid Hamza 1200875

Mohammed Owda 1200089

Instructor: Dr. Yazan Abu Farha

Section: 2

Date: 20/6/2023

Birzeit

Introduction:

This project presents a game developed using React, a popular JavaScript library for building user interfaces. The game is inspired by board games such as chess or checkers, wherein two players (Player 1 and Player 2) take turns to make moves. Players can be either humans or the computer.

The board is a grid comprising 64 squares, much like a traditional chessboard. At the start of the game, only a limited number of squares are valid moves. After each turn, new valid moves are determined based on the player's move. The game ends when there are no more valid moves, or when a player creates a sequence of their own tokens in a row (horizontally, vertically, or diagonally).

This game also includes audio feedback to enrich the user experience. Audio effects are played when a player makes a move, attempts an invalid move, or when the game ends.

Before the game begins, the players have the option to choose whether they want to play against a friend or against the computer. If the option to play against the computer is chosen, the player can then choose to play as either Black or White.

The game uses React's built-in hooks to manage state, and the Concurrent Mode feature for a smooth user experience. The AI for the computer player is implemented using a minimax algorithm, a popular AI algorithm used in two-player turn-based games.

Table of Contents

Getting Started:

To run the program, follow these steps:

1. Download and install Visual Studio Code (VS Code), a popular code editor, if you don't already have it. You can download it from https://code.visualstudio.com/.

2. Once you have VS Code installed, open it and navigate to the Extensions view by clicking on the Extensions view icon on the Sidebar (it looks like four small squares).

3. In the Extensions view, search for "Live Server" and install this extension. Live Server is a simple development http server with live reload capability.

4. After installing the Live Server extension, open your project in VS Code by selecting "File" -> "Open Folder" and navigating to the location of your project.

5. Once the project is open, locate the **index.html** file in the file explorer. Right-click on **index.html** and select "Open with Live Server". This will start the live server and open your project in your default web browser.

Alternatively, you can try it from out site: https://lucent-lebkuchen-a93af8.netlify.app/

Main Functions:

1. **Square({ value, onSquareClick })**: This function represents a square component on the game board. It receives the **value** prop, which represents the current value of the square, and the **onSquareClick** prop, which is a callback function to handle the square click event. It renders a button element with a div inside, and the CSS classes of the div are determined by the **value**. When the square is clicked, it triggers the **onSquareClick** function.

2. **openNewMove(j, nextSquares)**: This function is responsible for determining the next open move based on the current move **j**. It checks the position of the current move on the game board and determines the next open move by considering the adjacent squares. If the next square is empty, it adds it to the **ValidMoves** set. It also removes the current move from the **ValidMoves** set. Additionally, it plays a sound effect using the **move** audio.

3. **Board({ player1Turn, squares, onPlay })**: This function represents the game board component. It receives the **player1Turn** prop, which indicates if it's player

1's turn, the **squares** prop, which represents the current state of the game board, and the **onPlay** prop, which is a callback function to handle the game board updates. It renders the game board UI and handles the click event on squares. If a valid move is made, it updates the **squares** array, sets the next player's turn, calls the **openNewMove** function, plays a sound effect, and triggers the **onPlay** callback with the updated **squares**.

4. **handleClick(i)**: This function is called when a square is clicked by a player. It receives the index **i** of the clicked square. It first checks if the move is valid by verifying if it exists in the **ValidMoves** set and if there is no winner or if the square has not been played yet. If the move is valid, it updates the **squares** array, updates the **PlayedSquares** array, changes the player's turn, calls the **openNewMove** function, plays a sound effect, and triggers the **onPlay** callback with the updated **squares**. If the move is invalid, it plays a sound effect indicating an invalid move.

5. **calculateWinner(PlayedSquares, player1Turn)**: This function determines if there is a winner in the game. It receives the **PlayedSquares** array, which represents the squares played by the players, and the **player1Turn** variable, which indicates if it's player 1's turn. It checks for winning combinations horizontally, vertically, and diagonally. If a winning combination is found, it returns the winner's name. If there are no empty squares left, it returns "Draw". Otherwise, it returns **null**.

6. **generateBoard()**: This function initializes the game board and starts the game by rendering the **App** component. It uses the **createRoot** function from the **react-dom/client** module to create the root element and render the **App** component inside it.

7. **chooseBlackOrWhite()**: This function allows the user to choose between playing as black or white. It replaces the current content of the root element with two buttons, "I'm Black" and "I'm White". When the buttons are clicked, it updates the **players** array accordingly and calls the **generateBoard** function to start the game.

8. **OneOrTwoPlayers()**: This function presents the user with the option to play against friends or against the computer. It replaces the current content of the root element with two buttons, "Play vs Friends" and "Play vs Computer". When the buttons are clicked, it either calls the **generateBoard** function to start a two-player game or calls the **chooseBlackOrWhite** function to allow the user to choose black or white against the computer.

9. evaluatePotentialWins(squares, player, opponent): This function evaluates the potential wins for a given player on the game board. It takes three parameters: squares (representing the current state of the game board), player (representing the player's color), and opponent (representing the opponent's color). The function

iterates over the winning combinations on the board and analyzes the squares in each combination. It assigns scores based on the number of squares occupied by the player, opponent, or empty squares. Higher scores indicate a better potential win for the player. The function returns the calculated score.

10. minimax(squares, depth, isMaximizing, player, opponent): This function implements the minimax algorithm to determine the best move for the computer player. It takes five parameters: squares (representing the current state of the game board), depth (representing the depth of the recursive minimax search), isMaximizing (indicating whether the current turn is for the maximizing player), player (representing the computer player's color), and opponent (representing the opponent's color). The function recursively explores the game tree by simulating all possible moves and assigns scores to each possible outcome. It alternates between maximizing and minimizing the score based on the player's turn. The function returns the best score achievable for the computer player.

Data Structures:

1. **ValidMoves**: This is a **Set** data structure that keeps track of the valid moves on the game board. It is initialized with the valid starting moves for the game. The set is updated throughout the game as new moves are made and invalid moves are removed. It ensures that the computer player and the human player only select valid moves during their turns.

2. **PlayedSquares**: This is an **Array** of size 64 that represents the state of the game board. Each element in the array corresponds to a square on the board, and its value can be **null** (indicating an empty square), **"Black"** (indicating a square occupied by the black player), or **"White"** (indicating a square occupied by the white player). The **PlayedSquares** array keeps track of the moves made by both players during the game.

3. **players**: This is an **Array** that stores the names of the players. By default, it is initialized with two players: **"player1"** and **"player2"**. The order of players determines whose turn it is during the game. It can be customized to accommodate different player names or configurations.

4. **PlayerTurn**: This is a variable that keeps track of the current player's turn. It is initialized with the first player in the **players** array. It is updated after each move to switch turns between players.

5. **ColorPlayer1** and **ColorPlayer2**: These variables store the colors assigned to each player. By default, **ColorPlayer1** is set to **"Black"** and **ColorPlayer2** is set to **"White"**. These colors are used to represent the players' moves on the game board.

Heuristic:

Our Heuristic function is **evaluatePotentialWins.**

1. The **evaluatePotentialWins** function examines various winning combinations on the game board to assess the potential win or advantage for the AI player.
2. It considers different patterns, such as horizontal, vertical, and diagonal alignments, where the AI player can potentially form a winning sequence of moves.
3. The function assigns scores to different game states based on the presence of AI player's moves, opponent's moves, and empty squares within the winning combinations.
4. The scores are determined as follows:
    - If the AI player has five consecutive moves in a winning combination, the score is set to 100, indicating a guaranteed win for the AI player.
    - If the opponent has five consecutive moves in a winning combination, the score is set to -90, indicating a strong disadvantage for the AI player.
    - The presence of four AI player's moves with one empty square in a winning combination suggests a high potential for the AI player to win, resulting in a score of 50.
    - Similarly, the presence of four opponent's moves with one empty square in a winning combination indicates a high potential for the opponent to win, resulting in a score of -60.
    - Three AI player's moves with two empty squares in a winning combination suggests a moderate potential for the AI player to form a winning sequence, resulting in a score of 10.

- Three opponent's moves with two empty squares in a winning combination indicates a moderate potential for the opponent to form a winning sequence, resulting in a score of -9.
- The presence of two AI player's moves with three empty squares in a winning combination suggests a low potential for the AI player to form a winning sequence, resulting in a score of 2.
- Similarly, the presence of two opponent's moves with three empty squares in a winning combination indicates a low potential for the opponent to form a winning sequence, resulting in a score of -1.

5. The evaluation function calculates an overall score by summing up the individual scores for each winning combination.

Justification:

The heuristic function **evaluatePotentialWins** serves as a critical component of the AI player's decision-making process. It allows the AI player to evaluate the potential strength of different game states based on the presence of winning combinations. Here's a justification for the heuristic:

1. **Efficiency**: The heuristic provides a quick evaluation of game states without exploring the entire game tree, making it computationally efficient. It focuses on relevant winning combinations, which are crucial for the outcome of the game.
2. **Strategic Decision-making**: By assigning scores to different game states, the AI player can prioritize moves that have a higher potential for winning or providing an advantage. This allows the AI player to make strategic decisions and select moves that are likely to lead to a favorable outcome.
3. **Adaptability**: The heuristic can adapt to different game situations and dynamically adjust the scores based on the presence of AI player's moves, opponent's moves, and empty squares. It takes into account both offensive and defensive strategies by considering the potential wins for both players.
4. **Heuristic Guidance**: While the heuristic does not guarantee the optimal move in every situation, it provides valuable guidance by assessing the potential strength of game states. It helps the AI player to focus on promising moves that increase the chances of winning or avoiding potential losses.

Tournament:

The performance of my code can be analyzed based on the outcomes against opponents. The results can be attributed to various factors that influence the code's performance.
If my code emerged victorious against the opponent, the following factors might have contributed to your success:

1. **Strategic Decision-making**: my code implemented a well-thought-out strategy by analyzing the game board, anticipating the opponent's moves, and making optimal decisions based on that analysis. It considered potential winning combinations, evaluated moves, and selected the best possible move to outmaneuver the opponent and secure victory.

2. **Evaluation Function**: The evaluation function in my code played a crucial role in assessing the game state and determining the desirability of moves. By accurately capturing key aspects of the game, such as potential wins and threats, your code was able to guide its decision-making process and make winning moves to exploit the opponent's weaknesses effectively.

3. **Efficiency**: The efficiency of your code, including algorithmic optimizations and data structures, played a crucial role in its success. By efficiently performing computations and evaluations, your code gained an advantage over the opponent, especially in time-constrained tournaments.

However, if my code lost against the opponent, the following factors might have contributed to the outcome:

1. **Strong Opponent**: my opponent may have implemented a superior strategy, evaluation function, or search algorithm. They might have better analyzed the game board, identified winning moves, and exploited your code's weaknesses, resulting in your loss.

2. **Suboptimal Decisions**: my code might have made suboptimal decisions due to limitations in the evaluation function or the search algorithm. If my code failed to accurately evaluate the game state or explore the game tree deeply enough, it might have missed winning opportunities or fallen into unfavorable positions, leading to the loss.

3. **Insufficient Search Depth**: If my search algorithm had a limited depth, it might not have explored the game tree deeply enough to make optimal decisions. As a result, your code might have overlooked potential winning moves or failed to

anticipate the long-term consequences of certain moves, leading to a disadvantageous position and eventual loss.