Faculty of Engineering & Technology

Electrical & Computer Engineering Department

Artificial Intelligence project #1

Magnetic cave problem

Prepared by:

Jana Herzallah 1201139

Lana Thabit 1200071

Instructor: Dr.Yazan Abufarha

Section: 2

Date: 16/6/2023

1. Program description (how to run it, what the main functions and data structures are):

Our project idea is to develop a game called the Magnetic Cave Game. The game is implemented using three classes. In the first class, named "Game," we handle the main logic and flow of the game. It includes methods for initializing the game board, displaying the board, making moves, checking for wins or ties, and switching players. The class also provides three different modes for the players to choose from.

In the "Game" class, we have a static method called "minu()" that serves as the entry point of the game. It displays a menu of modes for the players to choose from and accepts user input to determine the game mode. Depending on the chosen mode, it calls the respective methods in the same class or switches to the second and third classes for automatic moves.

The "Game" class also includes a method called "case1()" that corresponds to the first game mode, where both players make manual moves. Inside this method, the game board is initialized, and the instructions for playing the game are displayed. It uses a loop to keep the game running until a win or tie condition is met. The players are prompted to enter their moves, and the game checks the validity of the moves, updates the board, and determines the game outcome. After the game ends, the players have the option to play again or exit.

The second class, named "mode2," handles the second game mode, where one player makes manual moves (■) and the other player's moves (□) are automatically determined. Similarly, the third class, named "mode3," handles the third game mode, where the roles are reversed, and the manual moves are made by (□) while (■) moves automatically.

Mode 2 and mode3 classes implement a game loop for a board game where the player (manual or automatic) takes turns making moves on the board. The goal of the game is to get five consecutive bricks in a row, column, or diagonal.

Here's a summary of each class:

mode2:

Represents a game mode where the player with the brick '■' (manual) plays against the player with the brick '□' (automatic).

The game loop alternates between the manual player and the automatic player until there is a win or a tie.

The manual player takes input from the user for their moves.

The automatic player uses the minimax algorithm with alpha beta bruning to determine its moves.

mode3:

Represents a game mode where the player with the brick '□' (manual) plays against the player with the brick '■' (automatic).

The game loop follows a similar structure to mode2, but with reversed player turns.

Both classes share similar methods for initializing the board, getting player moves, making moves, checking win conditions, evaluating the board, and printing the board.

Function **case1()** in the **Game** class represents the logic and functionality for Mode 1: Manual moves for both players. Here's a description of the function and the data structures used:

Function Name: **case1()** Functionality: This function allows two players to take turns making manual moves on the game board until a win condition or tie is reached. It displays the game board, prompts the current player for their move (row and column), validates the move, checks for a win or tie, switches the player, and repeats the process until the game ends.

-Data Structures:

board (2D char array): Represents the game board where the moves are placed. It is initialized with a size of 8x8 and contains characters '-' to indicate empty positions.

-Main Functionality Steps:

1. Initializes the game board and sets the current player to '■'.
2. Displays the game mode instructions and initial empty game board.
3. Enters a loop to continue the game until it ends.
4. Inside the loop:

- Displays the current player.
- Prompts the current player for their move by accepting row and column numbers.
- Validates the move using the **makeMove()** method, which checks if the move is within the bounds and follows the game rules.
- If the move is invalid, an error message is displayed, and the loop continues.
- If the move is valid:
- Checks if the current player has won using the **checkWin()** method, which examines the game board for winning configurations.
- If a win condition is met, displays a message indicating the winning player and breaks out of the loop.

- Checks for a tie condition using the **checkTie()** method, which verifies if the game board is full and no player has won.
- If a tie condition is met, displays a tie message and breaks out of the loop.
- Switches the current player for the next turn using the **switchPlayer()** method.
- Displays an empty line to separate each player's turn.

5. After the game ends, displays the final game board.
6. Prompts the user if they want to play again.

- If the user enters "yes," the function continues by calling **minu()** (menu) method to restart the game.
- If the user enters "no," the application exits.
- If the user enters an invalid input, they are prompted again.

Mode2 & Mode 3:

1. Data Structures:

- **private static final int BOARD_SIZE = 8**: Defines the size of the game board.
- **private static final char EMPTY_CELL = '-';**: Represents an empty cell on the board.
- **private static final char PLAYER_BRICK = '■';**: Represents the player's game piece.
- **private static final char COMPUTER_BRICK = '□';**: Represents the computer's game piece.
- **Private static char[][] board = new char[BOARD_SIZE][BOARD_SIZE];**: Represents the game board as a 2D character array.

2. Functionalities:

- **public mode2() , public mode3()**: The class constructor that initializes the game board and constants.
- **public static void execute()**: Executes the game mode logic.
- **private static void initializeBoard()**: Initializes the game board with empty cells.
- **private static void printBoard()**: Prints the current state of the game board.
- **private static int[] getPlayerMove(Scanner scanner, char brick)**: Prompts the player to enter their move and returns the row and column indices.
- **private static void makeMove(int row, int col, char brick)**: Places a game piece (brick) on the specified row and column of the board.
- **private static int[] minimax(int depth, boolean maximizingPlayer, int alpha, int beta)**: Implements the minimax algorithm to find the best move for the computer player. It evaluates the board state recursively and returns the optimal row, column, and score for the current player.
- **private static int evaluateBoard()**: Evaluates the current state of the game board and returns a score based on the number of consecutive game pieces for each player.
- **private static int getConsecutiveBricksCount(char brick)**: Counts the number of consecutive game pieces in each direction (horizontal, vertical, diagonal) for a given player.
- **private static boolean checkWinCondition(char brick)**: Checks if a player has won the game by having five consecutive game pieces in a row, column, or diagonal.
- **private static boolean isBoardFull()**: Checks if the game board is completely filled with game pieces.
- **private static boolean isValidMove(int row, int col)**: Checks if a move (specified by row and column) is valid, i.e., if the cell is empty and has at least one neighboring occupied cell.
- **private static int[] getComputerMove()**: Gets the computer player's move by calling the **minimax** function with depth 4 and the appropriate parameters. It also checks if the manual player is about to win and blocks their move if necessary.

2. heuristic description and justification.

```
Private static int evaluateBoard2() {

    int computerScore = getConsecutiveBricksCount(COMPUTER_BRICK);
    int playerScore = getConsecutiveBricksCount(PLAYER_BRICK);

    if (computerScore >= 5) {
        return Integer.MAX_VALUE;
    } else if (playerScore >= 5) {
        return Integer.MIN_VALUE;
    }

    int computerTwo = getConsecutiveBricksCount2(COMPUTER_BRICK, 2);
    int playerTwo = getConsecutiveBricksCount2(PLAYER_BRICK, 2);

    int computerThree = getConsecutiveBricksCount2(COMPUTER_BRICK, 3);
    int playerThree = getConsecutiveBricksCount2(PLAYER_BRICK, 3);

    int finalcomputerScore = 10 * computerThree + 5 * computerTwo;
    int finalplayerScore = 10 * playerThree + 5 * playerTwo;

    return finalcomputerScore - finalplayerScore;
}
```

That relies on a function named getconsecutivebricks for a determined length as shown in the project

This is the heuristic we built to help the algorithm to choose the best move and one way that we used to justify it was to play against the AI into so many times and noticing that the AI always tends to win or at least blocks the manual player from winning.

where it calculates a score indicating the desirability of the current game state for the computer player. Here's a breakdown of the code:

1. The function starts by calculating the number of consecutive bricks for the computer player (**computerScore**) and the player (**playerScore**). The **getConsecutiveBricksCount()** function is called to determine the number of consecutive bricks for each player.
2. If the computer player has five or more consecutive bricks, the function returns **Integer.MAX_VALUE**, indicating that this state is highly desirable for the computer player (a winning state).
3. If the player has five or more consecutive bricks, the function returns **Integer.MIN_VALUE**, indicating that this state is highly undesirable for the computer player (a losing state).
4. Next, the function calculates the number of consecutive bricks for the computer player in groups of two (**computerTwo**) and three (**computerThree**). Similarly, it calculates the same values for the player (**playerTwo** and **playerThree**).
5. The final scores for the computer player and the player are determined by assigning weights to the counts of consecutive bricks. In this case, each group of three bricks contributes 10 points to the computer player's score, and each group of two bricks contributes 5 points. The scores are calculated as follows:

- **finalcomputerScore = 10 * computerThree + 5 * computerTwo**
- **finalplayerScore = 10 * playerThree + 5 * playerTwo**

6. The function returns the difference between the computer player's final score and the player's final score. This difference represents the evaluation score for the current board state. A positive value indicates a favorable state for the computer player, while a negative value indicates a favorable state for the player.

It focuses on evaluating the board based on the number and arrangement of consecutive bricks for both the computer player and the manual player. It assigns higher scores to states where the computer player has more consecutive bricks, specifically giving more weight to groups of three bricks compared to groups of two bricks. This evaluation function aims to guide the computer player towards moves that increase the likelihood of achieving consecutive bricks while discouraging moves that benefit the player.

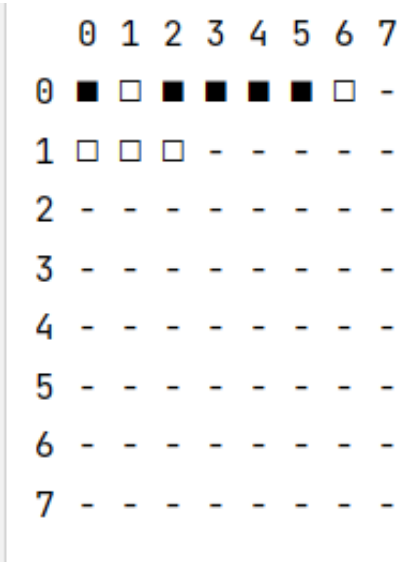
We also added a twist within the computer move function so that we can always ensure that its winning, the twist was is to always get values from the heuristic but if the manual has 4 consecutives in a row or a column or in a diagonal way, always swap the move with highest score got from the heuristic with the move that will block the player from winning.

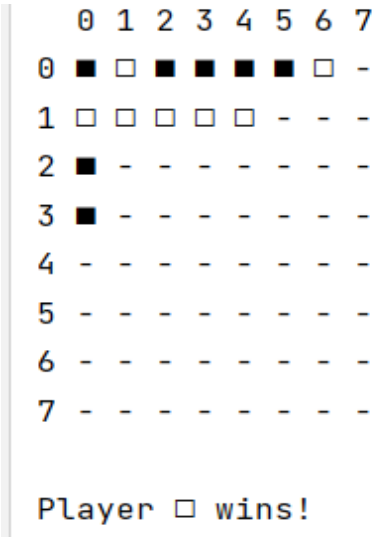
3. Results description and explanation at the tournament

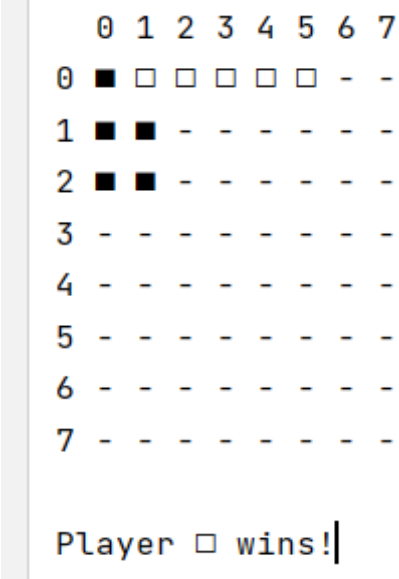
we think that we should win this tournament because of what we mentioned above that we added a special function that compares the highest score for the possible moves (calculated to depth 4) with the move that will ensure that the manual will never win and the time for our AI program takes to calculate the next move is less that 1 second which is considered amazing!

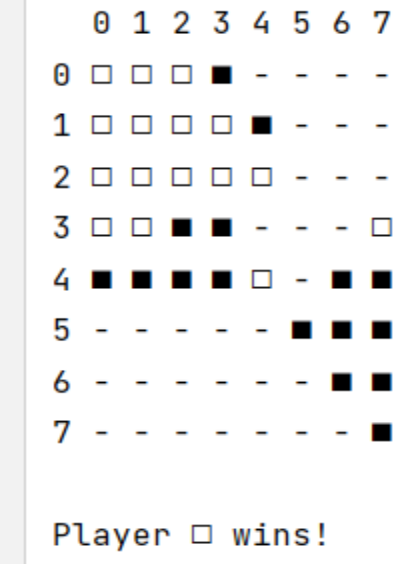
Here are some photos of the AI blocking the manual from winning:

Ps: the computer is the black bricks owner while the human has the whiones:









As we can see the manual player never has the chance to win and the computer either wins (mainly when the opponent is not blocking the computers way when its playing) or it at least doesn't allow the manual human to win which is totally what desired in this project.