Electrical and Computer Systems Engineering Department

Artificial Intelligence ENCS3340

Project1 Report

Magnetic Cave Game

Prepared by:

Christina Saba ID: #1201255

Sahar Hmidat ID: #1202038

Instructor’s Name: Dr. Ismail Khater

Section No: 3

Date: 20 June 2023

Table of Contents

List of Figures

Introduction

This project is called Magnetic Cave Game which consistes of a board of 8*8 size and two player opposing each other and each one tries to win the game by placing its position in a certain condition. The game has three modes, first one where both players are manual and therefore the user is controlling the game and its result, the second mode is the first player manually played by the user and the second player is played automatically based on the minimax algorithm and heuristic used to find the best move, the final mode is the same as the second but the players are the opposite which means the first is automatic and the second is manual.

Program Implementation

We used java language to implement this game using Eclipse. First we started by innitiating the Start_Playing_Game class for the user to choose from the manual and the game in the class called Magnectic_Cave_Board which indicates every detail in the game, it has private character parameters the board which is a two dimensional array, currentPlayer initialized to player1, player1 given the value ■ and player2 given the value □.

```java
public class Start_Playing_Game { //this class is to print the menu choices and ask the user to enter the choice he/she wants

    public static void main(String[] args) {

        Magnetic_Cave_Board game = new Magnetic_Cave_Board();
        Scanner scanner = new Scanner(System.in);

        System.out.println("-------------------Magnetic Cave Game--------------------");
        System.out.println("-------------------User Menu Choices--------------------");
        System.out.println();
        System.out.println("1. Manual entry for both ■'s moves and □'s moves");
        System.out.println("2. Manual entry for ■'s moves and automatic moves for □");
        System.out.println("3. Manual entry for □'s moves and automatic moves for ■");
        System.out.print("Enter your choice from 1 to 3: ");
        int choice = scanner.nextInt();

        switch (choice) {
            case 1:
                game.manualMode(); // calling the manual players method
                break;
            case 2:
                game.manualAndAutoMode(); //calling the manual player1 and automatic player2 method
                break;
            case 3:
                game.AutoAndManualMode();//calling the manual player2 and automatic player1 method
                break;
            default:
                System.out.println("Not Accepted Choice, Exiting the game!!");
                break;
        }


        scanner.close();
    }
```

Figure 1: Start_Playing_Game Code

Since we used the command-line as an interface to print the board as given in the project description then each field is initialized to '-' which is empty.

```java
import java.util.ArrayList;
import java.util.Scanner;

public class Magnetic_Cave_Board {
    private char[][] board;
    private char currentPlayer;
    private char player1;
    private char player2;

    public Magnetic_Cave_Board() {
        board = new char[8][8];
        player1 = '■';
        player2 = '□';
        currentPlayer = player1;
        initializeBoard();
    }

    public void printBoard() {
        System.out.println("   A B C D E F G H");
        for (int row = 0; row < 8; row++) {
            System.out.print((row + 1) + "|");
            for (int col = 0; col < 8; col++) {
                System.out.print(board[row][col] + " ");
            }
            System.out.print("|" + (8 - row));
            System.out.println();
        }
        System.out.println("   A B C D E F G H");
    }

    private void initializeBoard() {
        for (int row = 0; row < 8; row++) {
            for (int col = 0; col < 8; col++) {
                board[row][col] = '-';
            }
        }
    }
```

Figure 2: Initialize the Game

Magnectic_Cave_Board class also contains each method used for playing the game, including makeMove which checks the validity of the move and places the currentPlayer in the accepted empty position. The validity depends on restrictions given which “a player can only place a brick on an empty cell of the cave provided that the brick is stacked directly on the left or right wall, or is stacked to the left or the right of another brick (of any color)”.

```
public boolean makeMove(int row, int column) { // this method checks if the position is valid and empty and places
                                                // the currentPlayer
    if (row < 0 || row >= 8 || column < 0 || column >= 8) {
        return false;
    }

    if (column != 0 && column != 7 && board[row][column - 1] == '-' && board[row][column + 1] == '-') {
        return false;
    }

    if (board[row][column] != '-') {
        return false;
    }

    board[row][column] = currentPlayer;
    return true;
}
```

Figure 3: makeMove function

After assigning each position of the players there is a function that checks if one of them is a winner which is called isPlayerWinner. It checks if there are 5 consecutive bricks of any player in a row or column or diagonal based on a nested for loop and if statements checking the existance of the player.

```
public boolean isPlayerWinner(char player) { // this method is to check the rows, columns, or diagonal if there are
                                             // five block of one of the players to win
    for (int row = 0; row < 8; row++) { // check the rows and count for 5 columns
        for (int col = 0; col < 4; col++) {
            if (board[row][col] == player && board[row][col + 1] == player && board[row][col + 2] == player
                    && board[row][col + 3] == player && board[row][col + 4] == player) {
                return true;
            }
        }
    }
    for (int row = 0; row < 4; row++) { // check the columns and count for 5 rows
        for (int col = 0; col < 8; col++) {
            if (board[row][col] == player && board[row + 1][col] == player && board[row + 2][col] == player
                    && board[row + 3][col] == player && board[row + 4][col] == player) {
                return true;
            }
        }
    }
    for (int row = 0; row < 4; row++) { // check the columns and count the blocks from the top right to the left
        for (int col = 4; col < 8; col++) {
            if (board[row][col] == player && board[row + 1][col - 1] == player && board[row + 2][col - 2] == player
                    && board[row + 3][col - 3] == player && board[row + 4][col - 4] == player) {
                return true;
            }
        }
    }
    for (int row = 0; row < 4; row++) { // check the columns and count the blocks from the top left to the right
        for (int col = 0; col < 4; col++) {
            if (board[row][col] == player && board[row + 1][col + 1] == player && board[row + 2][col + 2] == player
                    && board[row + 3][col + 3] == player && board[row + 4][col + 4] == player) {
                return true;
            }
        }
    }

    return false; // if no one wins then return false
}
```

Figure 4: isPlayerWinner Function

The code has a manualMode function which satisfies the first choice for both players manually played and is contained in a while loop checking if the board is full indicating the tie, or player1 wins, or player2 wins. The user enters a value like A1 or H1 and the function parts the value in column and row by subtracting the column of 'A' character to get the column value and subtracting the row of 1 if player1 or 8 if player2 then switches between the players until termination.

```
public void manualMode() { // this method is for the first choice which alternate between both players
                           // each is manually played
    Scanner scanner1 = new Scanner(System.in);
    String entry;
    int row, column;
    while (!isBoardFull() && !isPlayerWinner(player1) && !isPlayerWinner(player2)) {
        printBoard();

        System.out.print("Player " + currentPlayer + ", enter your position (eg: A1 or H1): ");
        entry = scanner1.next().toUpperCase();

        column = entry.charAt(0) - 'A'; // calculate column number for player1 and player2

        if (currentPlayer == player1) {
            row = entry.charAt(1) - '1'; // calculate row for player1 which is from the left
        } else {
            row = '8' - entry.charAt(1); // calculate row for player2 which is from the right
        }

        if (!makeMove(row, column)) {
            System.out.println("Not Accepted Value, Try Again!!");
            continue;
        }
        switchPlayers(); //switch between players to alternate between them
    }

    printBoard();
    // keep checking if one of the players wins or a tie to terminate the game
    // accordingly
    if (isPlayerWinner(player1)) {
        System.out.println("Player ■ wins!");
    } else if (isPlayerWinner(player2)) {
        System.out.println("Player □ wins!");
    } else {
        System.out.println("It's a tie!");
    }

    scanner1.close();
}
```

Figure 5: manualMode Function

There is the manualAndAutoMode function which indicates the second choice and AutoAndManualMode for the third choice both of them call the functions onePlayerManualMode and makeAIMove for each.

```
public void manualAndAutoMode() { // this method is for the manual player1 and automatic player2 which alternates
                                  // between them according to the player
    while (!isBoardFull() && !isPlayerWinner(player1) && !isPlayerWinner(player2)) {
        printBoard();
        if (currentPlayer == player1) {
            onePlayerManualMode(); // manual for player1
        } else {
            makeAIMove(4, currentPlayer); // automatic for player2
        }
    }

    printBoard();

    if (isPlayerWinner(player1)) {
        System.out.println("Player ■ wins!");
    } else if (isPlayerWinner(player2)) {
        System.out.println("Player □ wins!");
    } else {
        System.out.println("It's a tie!");
    }
}
```

Figure 6: ManualandAutoMode Function

The called method is makeAIMove for the automatic player (anyone of them) and initiates the bestrow and bestcolumn to -1 then creates an arraylist to al the possible valid moves and calls the minimax function with the parameters board, depth, alpha, beta, isMaximizingPlayer. Based on the values returned it compares the minimax value with the maxValue and if larger then switch the values to the current position of the currentPlayer and if the values are not -1 then the makeMove function is called to place the player.

```
public void makeAIMove(int depth, char player) { // this method calls the minimax method and compares the scores and
                                                 // chooses the best score and places the player in that position
    int maxValue = Integer.MIN_VALUE;
    int bestRow = -1;
    int bestCol = -1;
    ArrayList<int[]> validMoves = getValidMoves();
    for (int[] move : validMoves) {
        int row = move[0];
        int col = move[1];
        board[row][col] = currentPlayer;
        switchPlayers();
        int Minimax = minimax(board, depth, Integer.MIN_VALUE, Integer.MAX_VALUE, false);
        board[row][col] = '-';
        switchPlayers();
        if (Minimax > maxValue) {
            maxValue = Minimax;
            bestRow = row;
            bestCol = col;
        }
    }

    if (bestRow != -1 && bestCol != -1) {
        makeMove(bestRow, bestCol);
    }
    if (isPlayerWinner(player) == true) {
        printBoard();
        System.out.println("Automatic" + player + "wins!");
    }

    else if (isBoardFull()) {
        printBoard();
        System.out.println("It's a draw!");
    }

    switchPlayers();

}
```

Figure 7: makeAIMove Function

The minimax function checks if the depth is 0 which is the base for the recursive calls to call the evaluateBoard function, if not, it takes the depth value, alpha, beta and determines if the player is maximizing or minimizing, if maximizing is true then inside the loop which moves between the valid moves the minimax function is called resursive then maximizing is set to false to evaluate the opponent's moves and predict the worst move then the functions alternated between maximizing and minimizing players till optimal solution is found. It returns the maxValue for the maximizing player and minValue for minimizing player.

When the depth is 0 then the maximum depth is found as in each step the depth is decremented and the recursion should stop, it also checks if any player has won the game or tie to call the evaluateBoard function.

```
public int minimax(char[][] board, int depth, int alpha, int beta, boolean isMaximizingPlayer) {
    if (depth == 0 || isPlayerWinner(player1) || isPlayerWinner(player2) || isBoardFull()) {
        return evaluateBoard();
    }
    if (isMaximizingPlayer) { //maximizing player
        int maxValue = Integer.MIN_VALUE;
        ArrayList<int[]> validMoves = getValidMoves();
        for (int[] move : validMoves) {
            int row = move[0];
            int col = move[1];
            board[row][col] = currentPlayer;
            switchPlayers();
            int Minimax = minimax(board, depth - 1, alpha, beta, false);
            board[row][col] = '-';
            switchPlayers();
            maxValue = Math.max(maxValue, Minimax);
            alpha = Math.max(alpha, Minimax);
            if (beta <= alpha) {
                break;
            }
        }
        return maxValue;
    } else { //minimizing player
        int minValue = Integer.MAX_VALUE;
        ArrayList<int[]> validMoves = getValidMoves();
        for (int[] move : validMoves) {
            int row = move[0];
            int col = move[1];
            board[row][col] = currentPlayer;
            switchPlayers();
            int Minimax = minimax(board, depth - 1, alpha, beta, true);
            board[row][col] = '-';
            switchPlayers();
            minValue = Math.min(minValue, Minimax);
            beta = Math.min(beta, Minimax);
            if (beta <= alpha) {
                break;
            }
```

Figure 8:Minimax Function

The getValidMoves function returns an arraylist type which contains all the valid moves the player can take. makeMove function is used to check the constraints given and accordingly stores the rows and columns available in an array which is added to the arraylist.

```
public ArrayList<int[]> getValidMoves() { // this method gathers all possible moves which are valid in the
                                          // board and saves them in an arrayList
    ArrayList<int[]> validMoves = new ArrayList<>();
    for (int row = 0; row < 8; row++) {
        for (int col = 0; col < 8; col++) {
            if (makeMove(row, col)) {
                validMoves.add(new int[] { row, col });
                board[row][col] = '-';
            }
        }
    }
    return validMoves;
}
```

Figure 9: getValidMove Function

The last function is the evaluateBoard function which is called by the minimax function, it begins with a score set to 0, determines if the opponent is player1 or player2, based on the value of the currentPlayer.

It starts with the rows and makes nested for loops to iterate on the rows, columns, or diagonals.

There are four loops based on each position, the first is the row detection, the player iterates on the number of the rows for 0 to 7 and checks columns of size 5 making sure that the game does not fall to index-out-of-bound error and therefore increases the currentPlayer count and opponent count. If the currentPlayer has 5 in a row then return maximum integer so that the player wins, if the opponent has 5 in a row then return minimum in which the opponent wins. If no one wins at that time then add the square of the player count to the score and subtract from it the square of the opponent count.

```
public int evaluateBoard() {
    int score = 0;
    char opponentPlayer; // finding the currentPlayer and opponent to determine the score accordingly
    if (currentPlayer == player1) {
        opponentPlayer = player2;
    } else {
        opponentPlayer = player1;
    }
    // Evaluate rows
    for (int row = 0; row < 8; row++) {
        for (int col = 0; col < 4; col++) {
            int playerCount = 0;
            int opponentCount = 0;
            for (int i = col; i < col + 5; i++) {
                if (board[row][i] == currentPlayer) {
                    playerCount++;
                } else if (board[row][i] == opponentPlayer) {
                    opponentCount++;
                }
            }
            if (playerCount == 5) {
                return Integer.MAX_VALUE; // Current player wins the game
            } else if (opponentCount == 5) {
                return Integer.MIN_VALUE; // Opponent player wins the game
            }
            score += Math.pow(playerCount, 2); //return the score if no one won yet
            score -= Math.pow(opponentCount, 2);
        }
    }
```

Figure 10: Evaluate Rows

The second is the column detection, the player iterates on the number of the columns 0 to 7 and checks rows of size 4 making sure that the game does not fall to index-out-of-bound error and therefore increases the currentPlayer count and opponent count. If the currentPlayer has 5 in a column then return maximum integer so that the player wins, if the opponent has 5 in a column then return minimum in which the opponent wins. If no one wins at that time then add the square of the player count to the score and subtract from it the square of the opponent count.

```
for (int col = 0; col < 8; col++) {
    for (int row = 0; row < 4; row++) {
        int currentplayerCount = 0;
        int opponentCount = 0;
        for (int i = row; i < row + 5; i++) {
            if (board[i][col] == currentPlayer) {
                currentplayerCount++;
            } else if (board[i][col] == opponentPlayer) {
                opponentCount++;
            }
        }
        if (currentplayerCount == 5) {
            return Integer.MAX_VALUE;
        } else if (opponentCount == 5) {
            return Integer.MIN_VALUE;
        }
        score += Math.pow(currentplayerCount, 2);
        score -= Math.pow(opponentCount, 2);
    }
}
```

Figure 11: Evaluate Columns

The third is the diagonal detection from top left to right, the player iterates on outerloop of the number of the rows 0 to 3, in which there are 5 bricks in the same diagonal and if greater then the diagonals can extend greater than the size of the board making sure that the game does not fall to index-out-of-bound error, and innerloop of columns of size 3 as well. Then another for loop to iterate along five positions and increasing the currentplayercount and opponent count if they exist there. If the currentPlayer has 5 in a diagonal then return maximum integer so that the currentplayer wins, if the opponent has 5 in a diagonal then return minimum in which the opponent wins. If no one wins at that time then add the square of the player count to the score and subtract from it the square of the opponent count.

```
for (int row = 0; row < 4; row++) {
    for (int col = 0; col < 4; col++) {
        int currentplayerCount = 0;
        int opponentCount = 0;
        for (int i = 0; i < 5; i++) {
            if (board[row + i][col + i] == currentPlayer) {
                currentplayerCount++;
            } else if (board[row + i][col + i] == opponentPlayer) {
                opponentCount++;
            }
        }
        if (currentplayerCount == 5) {
            return Integer.MAX_VALUE;
        } else if (opponentCount == 5) {
            return Integer.MIN_VALUE;
        }
        score += Math.pow(currentplayerCount, 2);
        score -= Math.pow(opponentCount, 2);
    }
}
```

Figure 12: Evaluate diagonal top left to right

The fourth is the diagonal detection from top right to left which has the same technique as the above.

```
for (int row = 0; row < 4; row++) {
    for (int col = 4; col < 8; col++) {
        int currentplayerCount = 0;
        int opponentCount = 0;
        for (int i = 0; i < 5; i++) {
            if (board[row + i][col - i] == currentPlayer) {
                currentplayerCount++;
            } else if (board[row + i][col - i] == opponentPlayer) {
                opponentCount++;
            }
        }
        if (currentplayerCount == 5) {
            return Integer.MAX_VALUE;
        } else if (opponentCount == 5) {
            return Integer.MIN_VALUE;
        }
        score += Math.pow(currentplayerCount, 2);
        score -= Math.pow(opponentCount, 2);
    }
}

return score;
```

Figure 13: Evaluate Diagonal top right to left

Then the function returns the evaluated score and it is higher if the opponent has more cells in more desired positions and lower scores if the current player has control over.

Running the Program

By running the program this figure shows up for the user to choose among the three choices available.

```
------------------Magnetic Cave Game-------------------
------------------User Menu Choices--------------------

1. Manual entry for both ■'s moves and □'s moves
2. Manual entry for ■'s moves and automatic moves for □
3. Manual entry for □'s moves and automatic moves for ■
Enter your choice from 1 to 3:
```

Figure 14: Run the Program

After choosing the first one for example, the user has the ability to manually play both players. Here both players can enter the position of each one accordingly, and based on the restrictions applied no player can violate them and is required to insert a position again.

```
Player ■, enter your position (eg: A1 or H1): A1
   A B C D E F G H
1|■ - - - - - - - |8
2|- - - - - - - - |7
3|- - - - - - - - |6
4|- - - - - - - - |5
5|- - - - - - - - |4
6|- - - - - - - - |3
7|- - - - - - - - |2
8|- - - - - - - - |1
   A B C D E F G H
Player □, enter your position (eg: A1 or H1): H1
   A B C D E F G H
1|■ - - - - - - - |8
2|- - - - - - - - |7
3|- - - - - - - - |6
4|- - - - - - - - |5
5|- - - - - - - - |4
6|- - - - - - - - |3
7|- - - - - - - - |2
8|- - - - - - - □ |1
   A B C D E F G H
Player ■, enter your position (eg: A1 or H1):
```

Figure 15: Player1 and Player2 Manually

Here player1 entered a wrong position which is B2 and there is no player in the left A2 so the game asks him to insert a position again and goes like this until a correct position is placed then alternates to the other player.

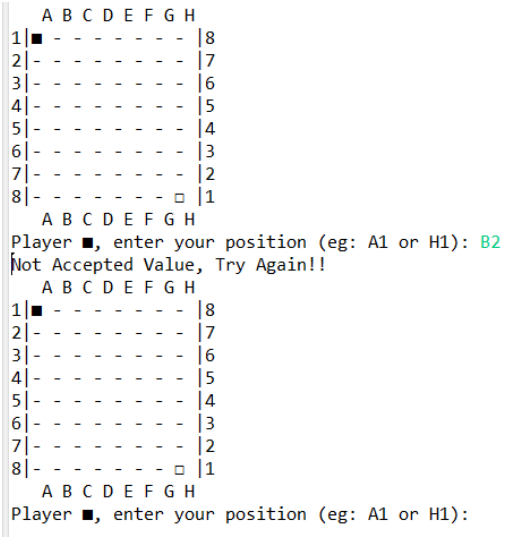

Figure 16: Player1 wrong Position

In this figure below, player1 wins the game by placing 5 bricks in the same row.

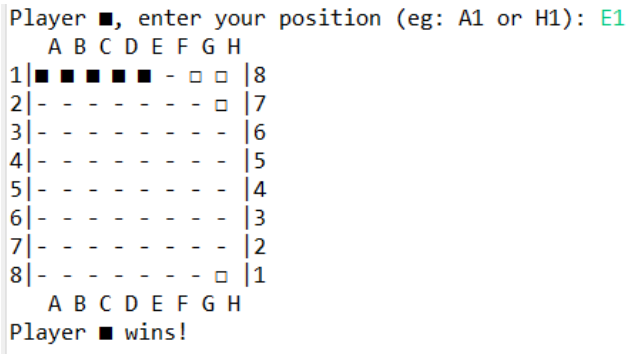

Figure 17: Player1 wins 5 bricks in a row

In this figure below, player1 wins the game by placing 5 bricks in the same column.

```
Player □, enter your position (eg: A1 or H1): H5
  A B C D E F G H
1|- - - - - - - ■ |8
2|■ ■ - - - - - - |7
3|■ - - - - - - - |6
4|■ - - - - - - □ |5
5|- - - - - - - □ |4
6|- - - - - - - □ |3
7|- - - - - - - □ |2
8|- - - - - - - □ |1
  A B C D E F G H
Player □ wins!
```

Figure 18: Player2 wins by placing 5 bricks in a column

In this figure below, player1 wins the game by placing 5 bricks in the same diagonal.

```
Player ■, enter your position (eg: A1 or H1): E5
  A B C D E F G H
1|■ □ □ - - - □ □ |8
2|■ ■ - - - - - ■ |7
3|■ ■ ■ □ - - - □ |6
4|■ ■ ■ ■ - - - □ |5
5|□ ■ ■ □ ■ - □ □ |4
6|- - - - - - □ □ |3
7|- - - - - - □ ■ |2
8|- - - - - - - - |1
  A B C D E F G H
Player ■ wins!
```

Figure 19: Player1 wins by placing 5 bricks in the same diagonal

If the second choice was selected then, player1 is manual and player2 is automatically played based on the minimax algorithm.

```
Player ■, enter your position (eg: A1 or H1): A1
  A B C D E F G H
1|■ - - - - - - - |8
2|- - - - - - - - |7
3|- - - - - - - - |6
4|- - - - - - - - |5
5|- - - - - - - - |4
6|- - - - - - - - |3
7|- - - - - - - - |2
8|- - - - - - - - |1
  A B C D E F G H
  A B C D E F G H
1|■ - - - - - - □ |8
2|- - - - - - - - |7
3|- - - - - - - - |6
4|- - - - - - - - |5
5|- - - - - - - - |4
6|- - - - - - - - |3
7|- - - - - - - - |2
8|- - - - - - - - |1
  A B C D E F G H
Player ■, enter your position (eg: A1 or H1):
```

Figure 20: Player1 manual and Player2 automatic

In the figure below, the second the manual player sets 4 brick in the first row, the automatic player blocks it and places its position.

```
Player ■, enter your position (eg: A1 or H1): D1
   A B C D E F G H
1|■ ■ ■ ■ - - - □ |8
2|□ - - - - - - - |7
3|- - - - - - - - |6
4|- - - - - - - - |5
5|- - - - - - - - |4
6|- - - - - - - - |3
7|□ - - - - - - - |2
8|■ - - - - - - □ |1
   A B C D E F G H
   A B C D E F G H
1|■ ■ ■ ■ □ - - □ |8
2|□ - - - - - - - |7
3|- - - - - - - - |6
4|- - - - - - - - |5
5|- - - - - - - - |4
6|- - - - - - - - |3
7|□ - - - - - - - |2
8|■ - - - - - - □ |1
   A B C D E F G H
Player ■, enter your position (eg: A1 or H1):
```

Figure 21: Player2 blocks Player1

If the third choice was selected then, player2 is manual and player1 is automatically played based on the minimax algorithm.

```
Player □, enter your position (eg: A1 or H1): H8
   A B C D E F G H
1|■ - - - - - - □ |8
2|- - - - - - - - |7
3|- - - - - - - - |6
4|- - - - - - - - |5
5|- - - - - - - - |4
6|- - - - - - - - |3
7|- - - - - - - - |2
8|- - - - - - - - |1
   A B C D E F G H
   A B C D E F G H
1|■ - - - - - ■ □ |8
2|- - - - - - - - |7
3|- - - - - - - - |6
4|- - - - - - - - |5
5|- - - - - - - - |4
6|- - - - - - - - |3
7|- - - - - - - - |2
8|- - - - - - - - |1
   A B C D E F G H
Player □, enter your position (eg: A1 or H1):
```

Figure 22: Player2 manual and Player1 automatic

In the figure below, the second the manual player sets 4 brick in the last column, the automatic player blocks it and places its position.

```
Player □, enter your position (eg: A1 or H1): H5
  A B C D E F G H
1|■ - - - - - ■ □ |8
2|- - - - - - - □ |7
3|- - - - - - - □ |6
4|- - - - - - - □ |5
5|- - - - - - - - |4
6|- - - - - - - - |3
7|- - - - - - - ■ |2
8|■ - - - - - - - |1
  A B C D E F G H
  A B C D E F G H
1|■ - - - - - ■ □ |8
2|- - - - - - - □ |7
3|- - - - - - - □ |6
4|- - - - - - - □ |5
5|- - - - - - - ■ |4
6|- - - - - - - - |3
7|- - - - - - - ■ |2
8|■ - - - - - - - |1
  A B C D E F G H
Player □, enter your position (eg: A1 or H1):
```

Figure 23: Player1 blocks Player2