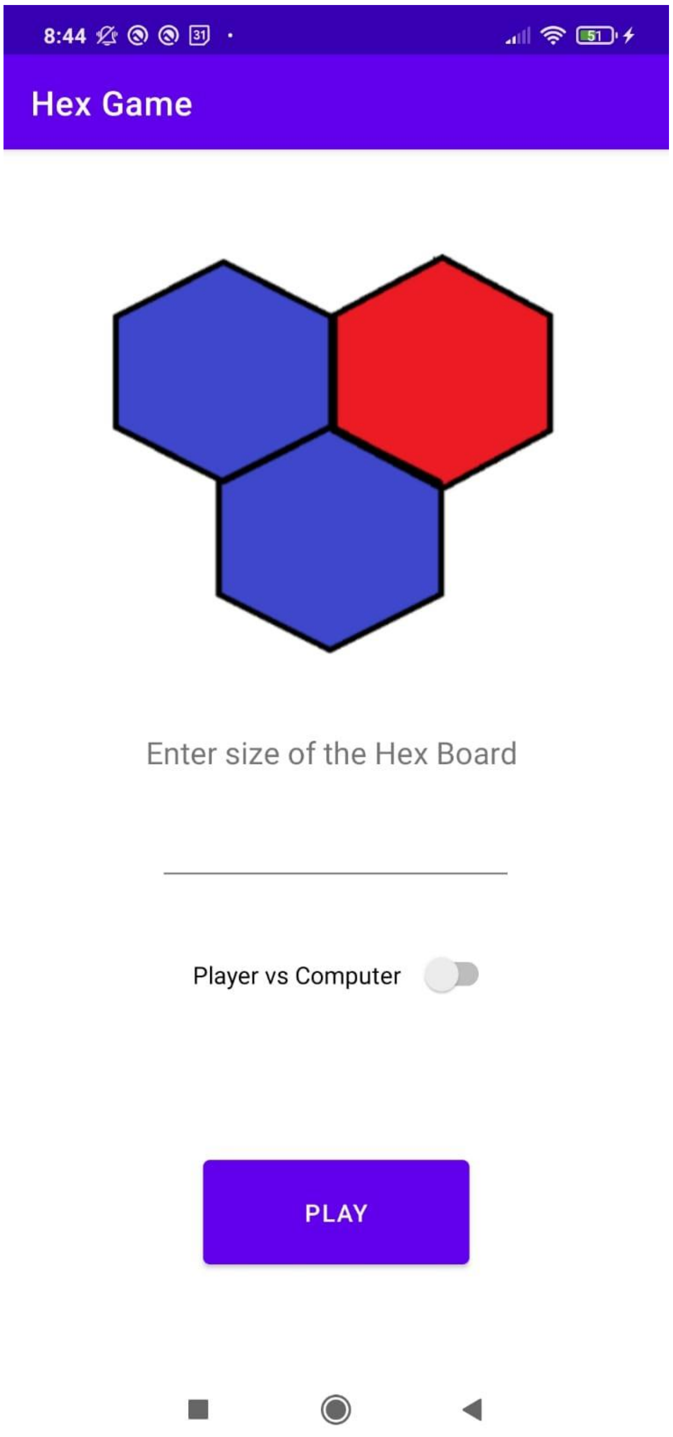
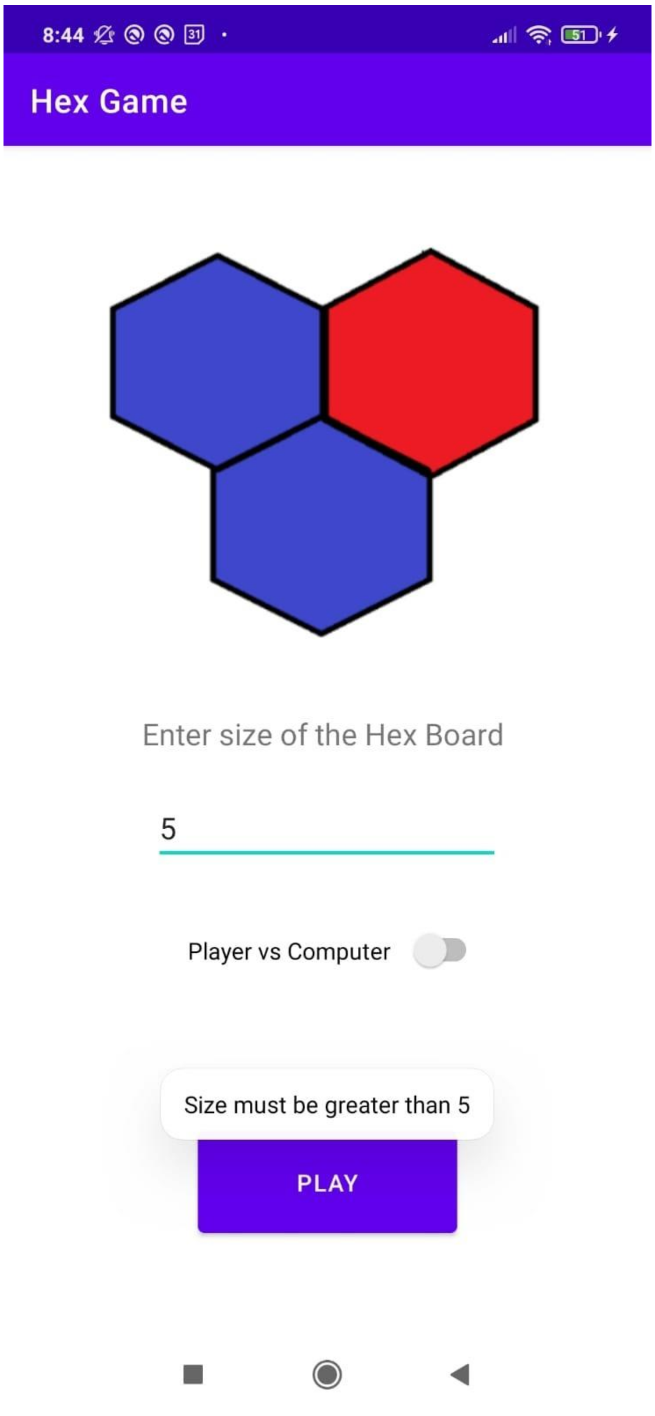
GIT Department of Computer Engineering
CSE 222/505 - Spring 2021
Winter Project Report

Mustafa Karakaş
1801042627

1. GAME SCREENSHOTS



This is the first screen we encounter when we open the game.


Hex Game

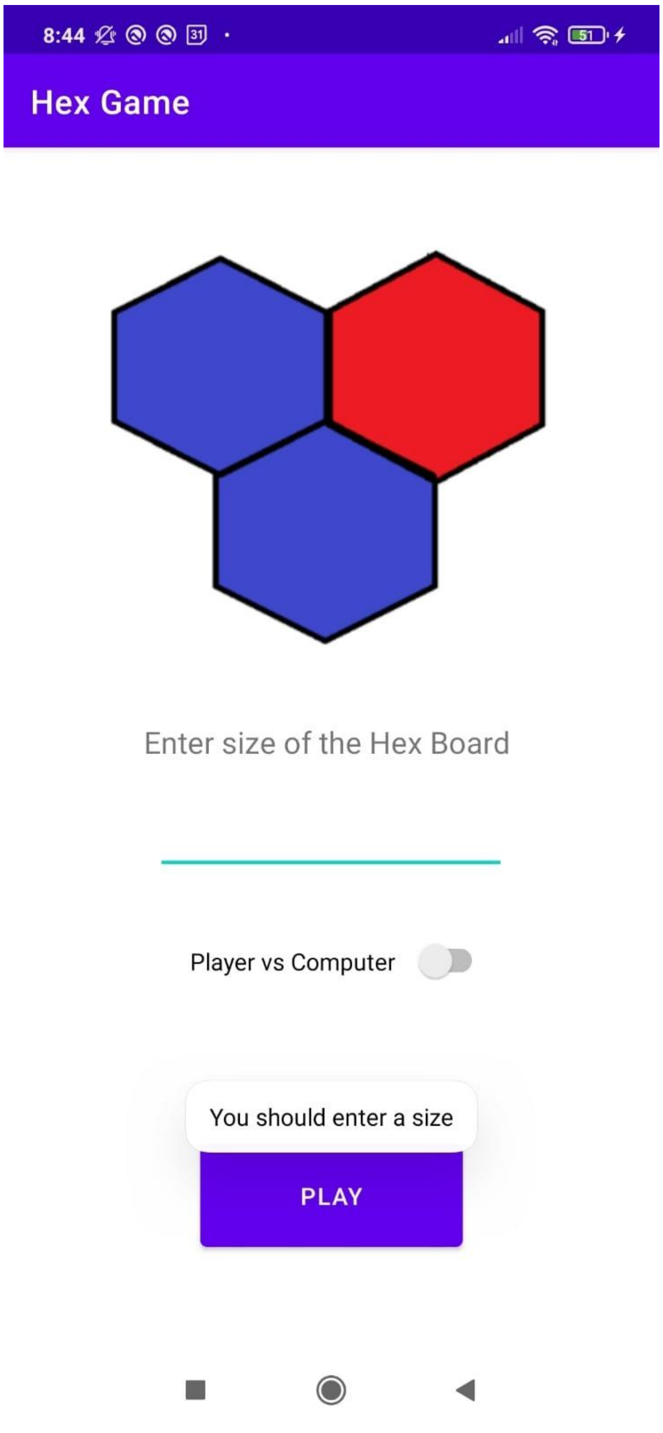
Enter size of the Hex Board

5

Player vs Computer

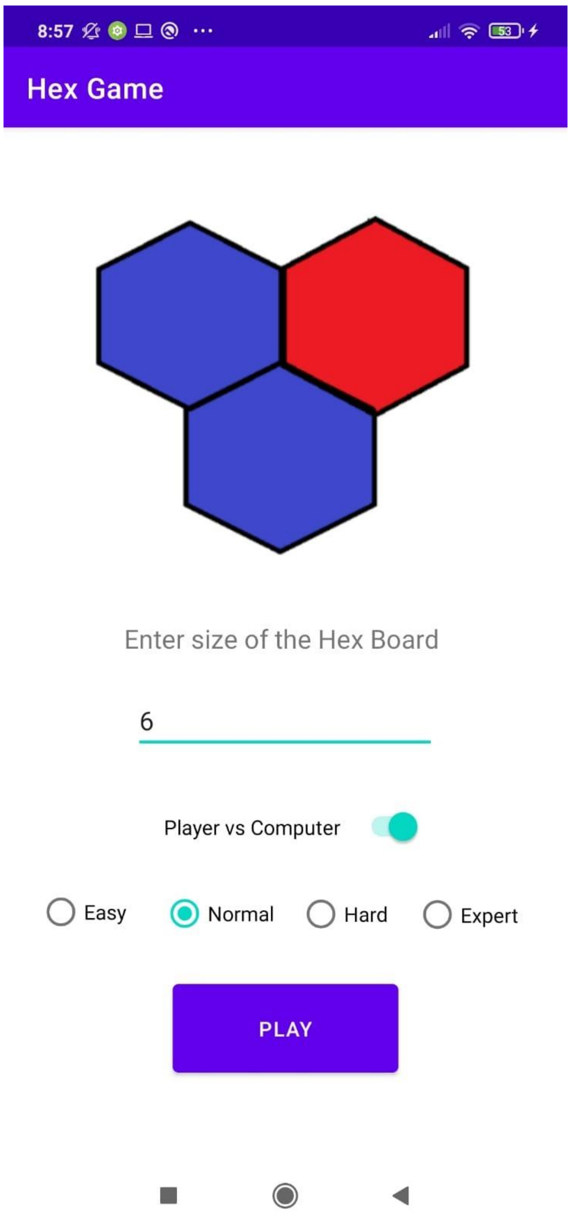
Size must be greater than 5

If the size is not greater than 5 it shows a warning message.



You should enter a size

It also shows message if the size is not provided.



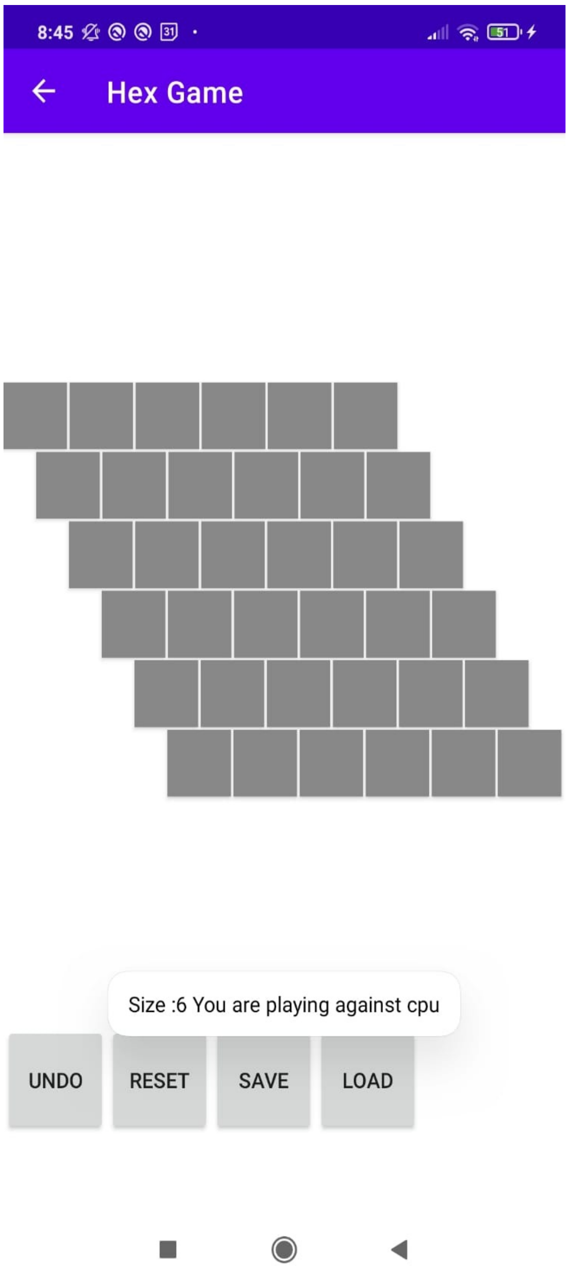
Enter size of the Hex Board

6

Player vs Computer [✓ on]

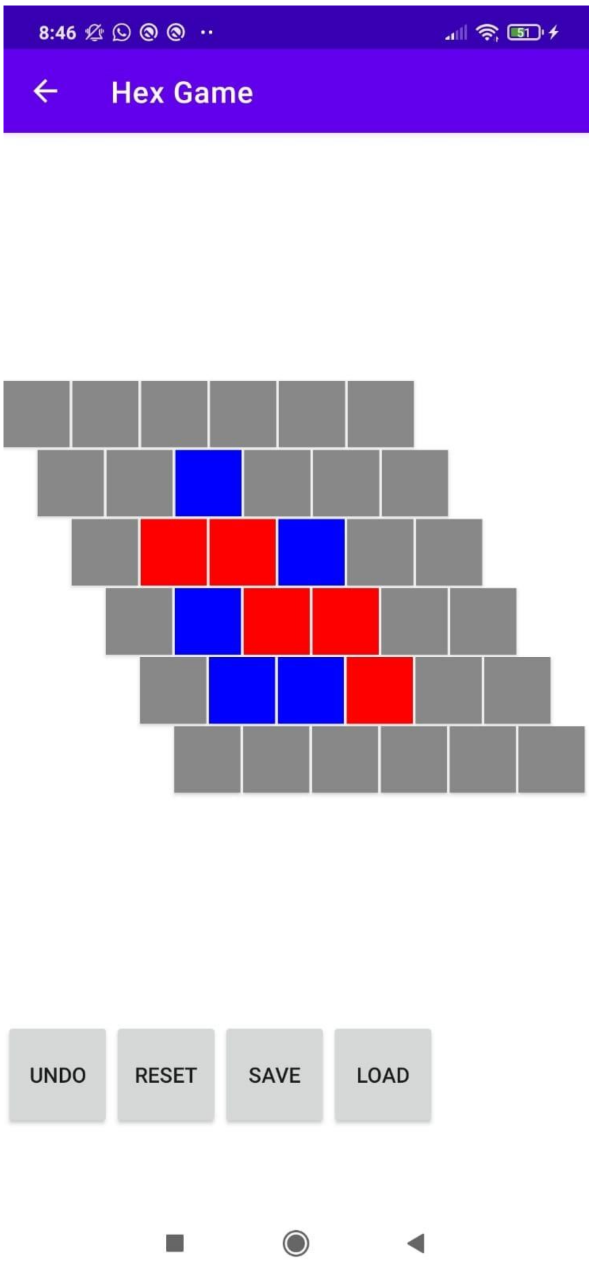
Easy [✓ Normal] Hard Expert

When we open the Player vs Computer switch it shows the level of the Computer. We should select one of them. Easy is selected automatically.



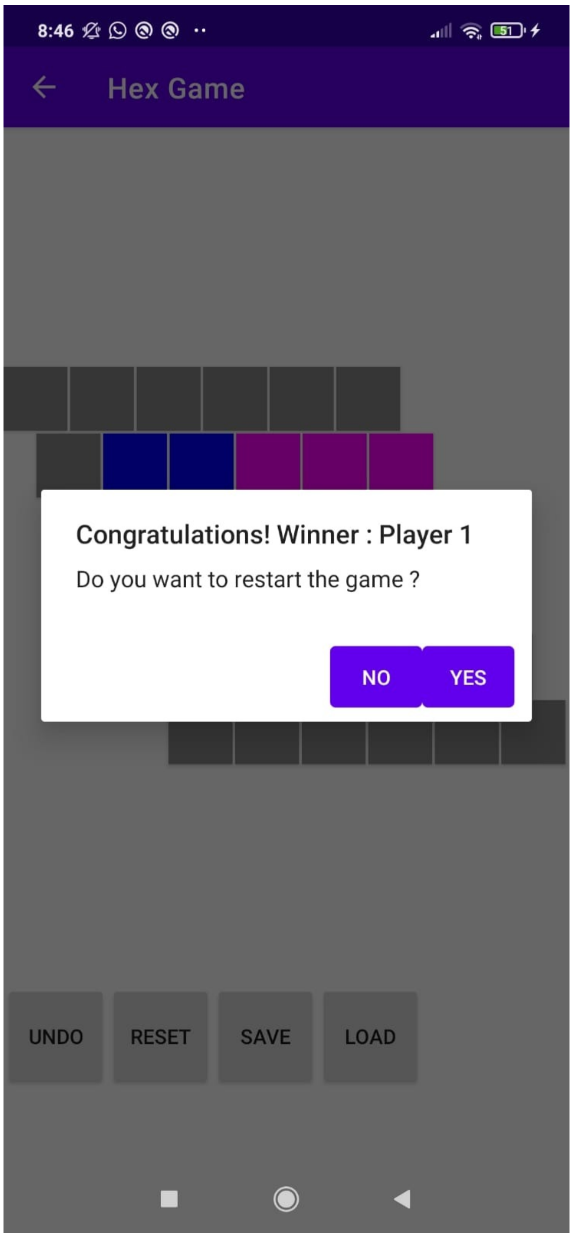
Size :6 You are playing against cpu

Table is created. Red should go horizontaly. Blue should go vertically. Red starts first. I did not implement the features ; “Who starts first” and “Change the color”.
And there is a message which shows the user what is the size and who you are playing against.



Game continues...

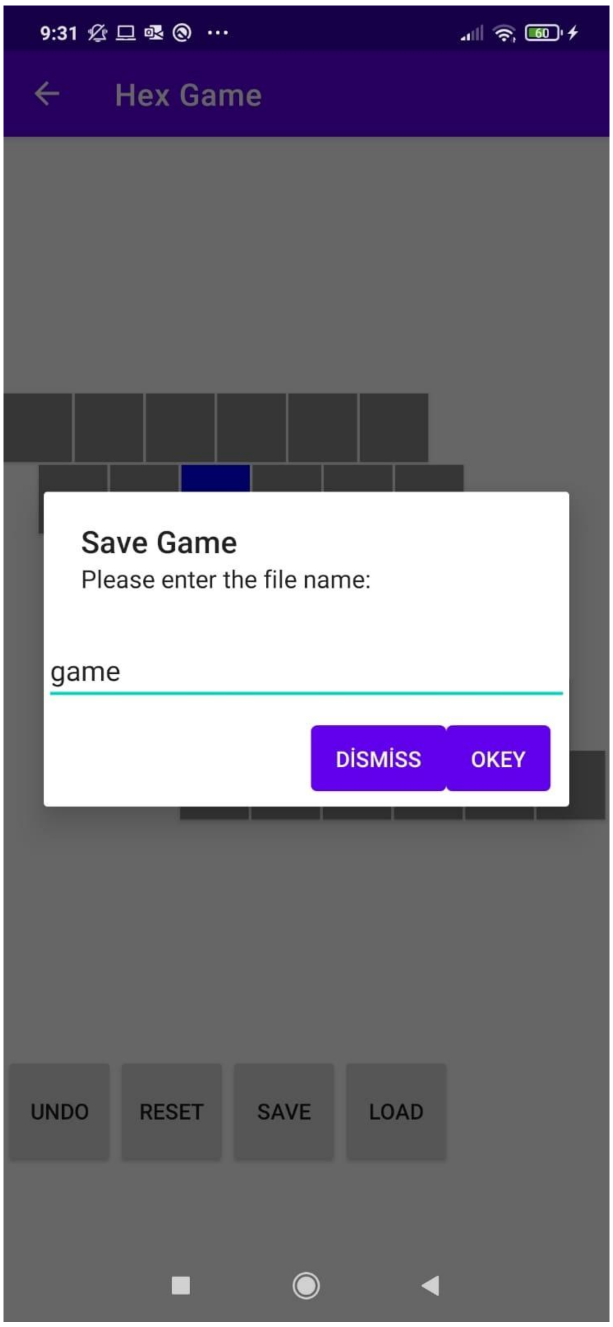
When the user pressed to “UNDO” button it removes the player’s last move and the move till to that move



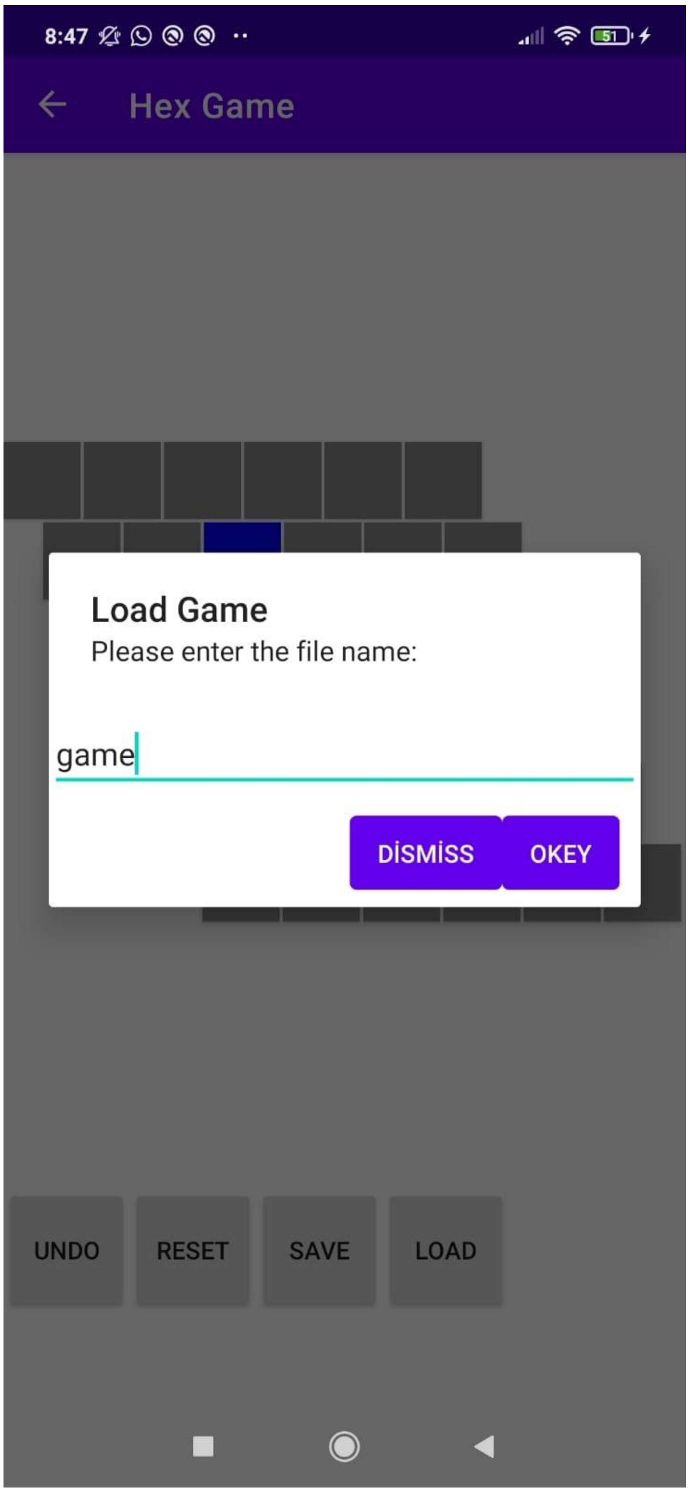
Congratulations! Winner : Player 1

Do you want to restart the game ?

When the red player won the winning path becomes pink. And message box pops up.
If the player touches to the 'NO' button:
The table appears again and the command buttons can be used.
If the player touches to the 'YES' button:
Game restart itself with the given settings(size and vs Player or Computer)



To be able to save the game the player should enter a text name.
After the saving is completed user can continue playing the current game.

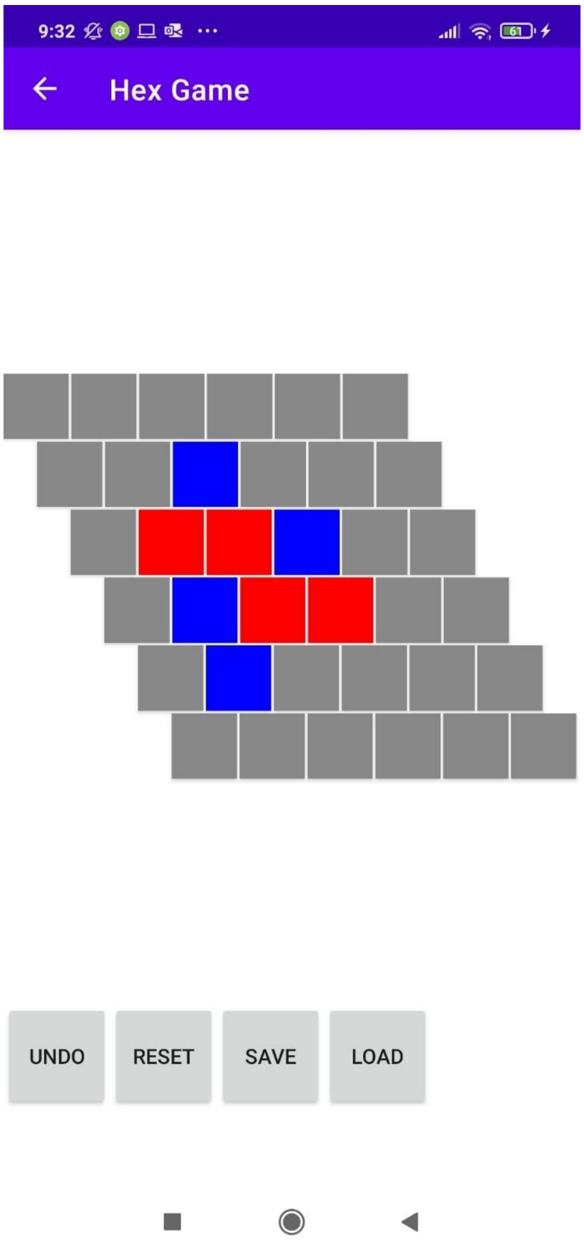


Load Game

Please enter the file name:

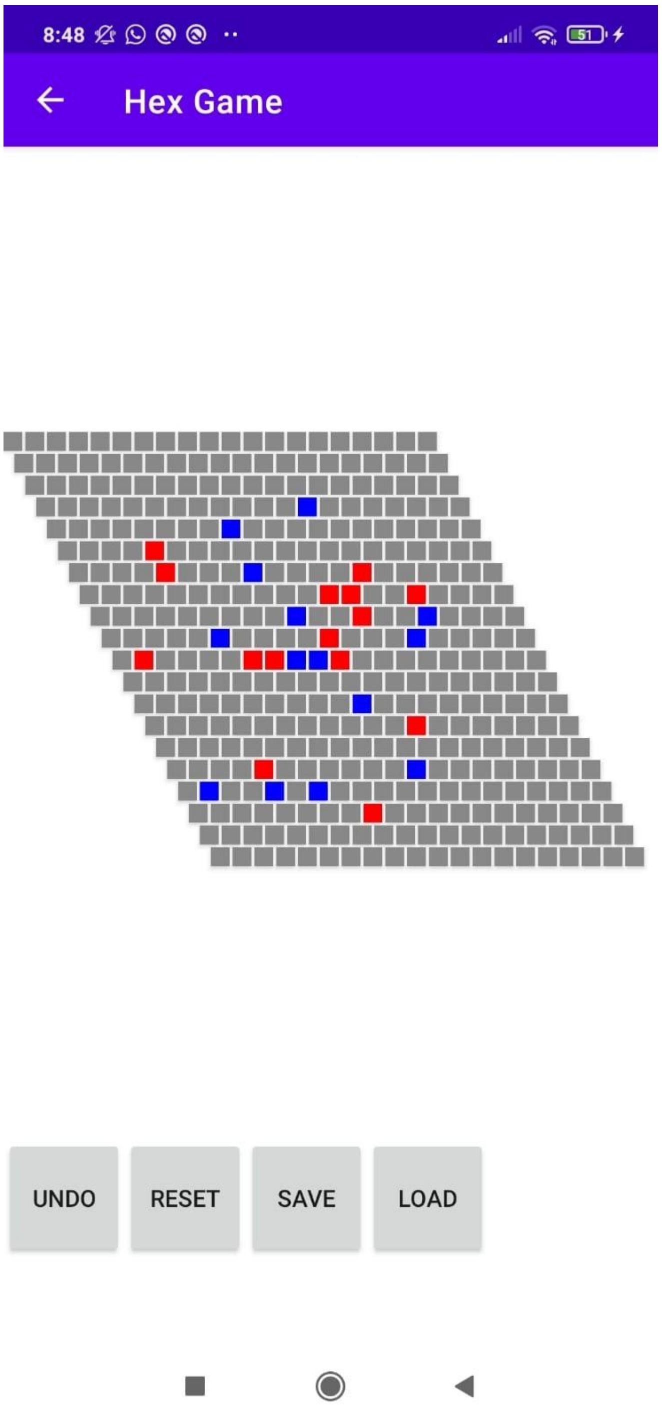
game

How to load game...



Game is loaded.

Example of 30x30 User vs User Game:



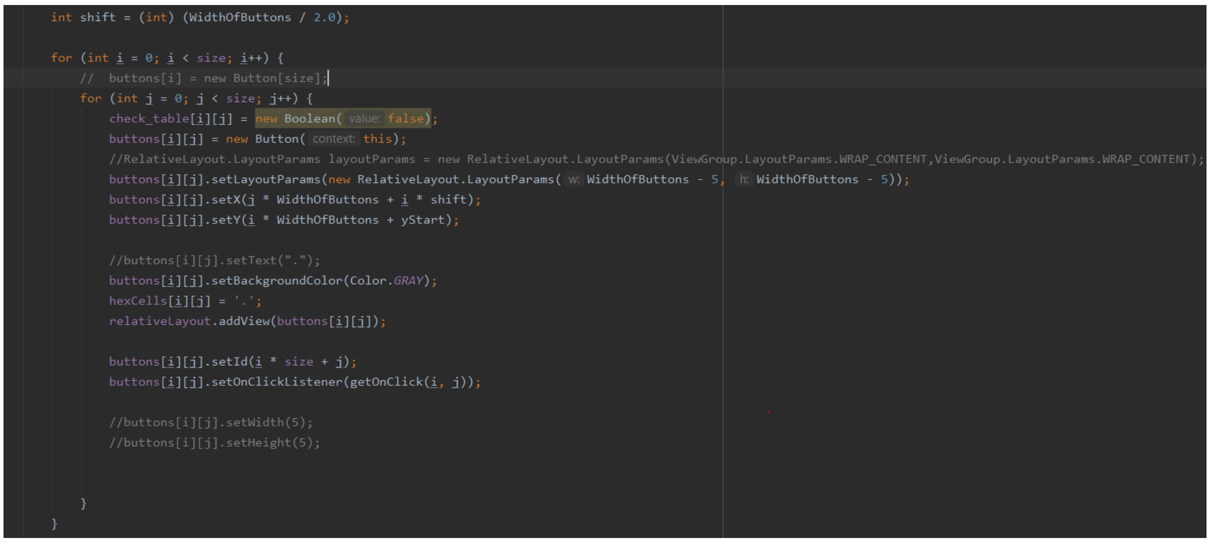
2. Algorithms

Game design is the same with the game we have built in the first term. Of course there is minimax algorithm additionally. We used open source code for the minimax algorithm. That is the link for the minimax algorithm:

https://github.com/kkmanos/hex-minimax/blob/master/src/board.c

When I apply the minimax algorithm to the my game , it was too much slow. Because of that, it goes through all the buttons and takes the max profit. I changed it a little bit and from now on , it only checks the neighbour buttons. Of course this has some disadvantages. It can not move as freely as it did but the first version is not playable. It was waiting too much.

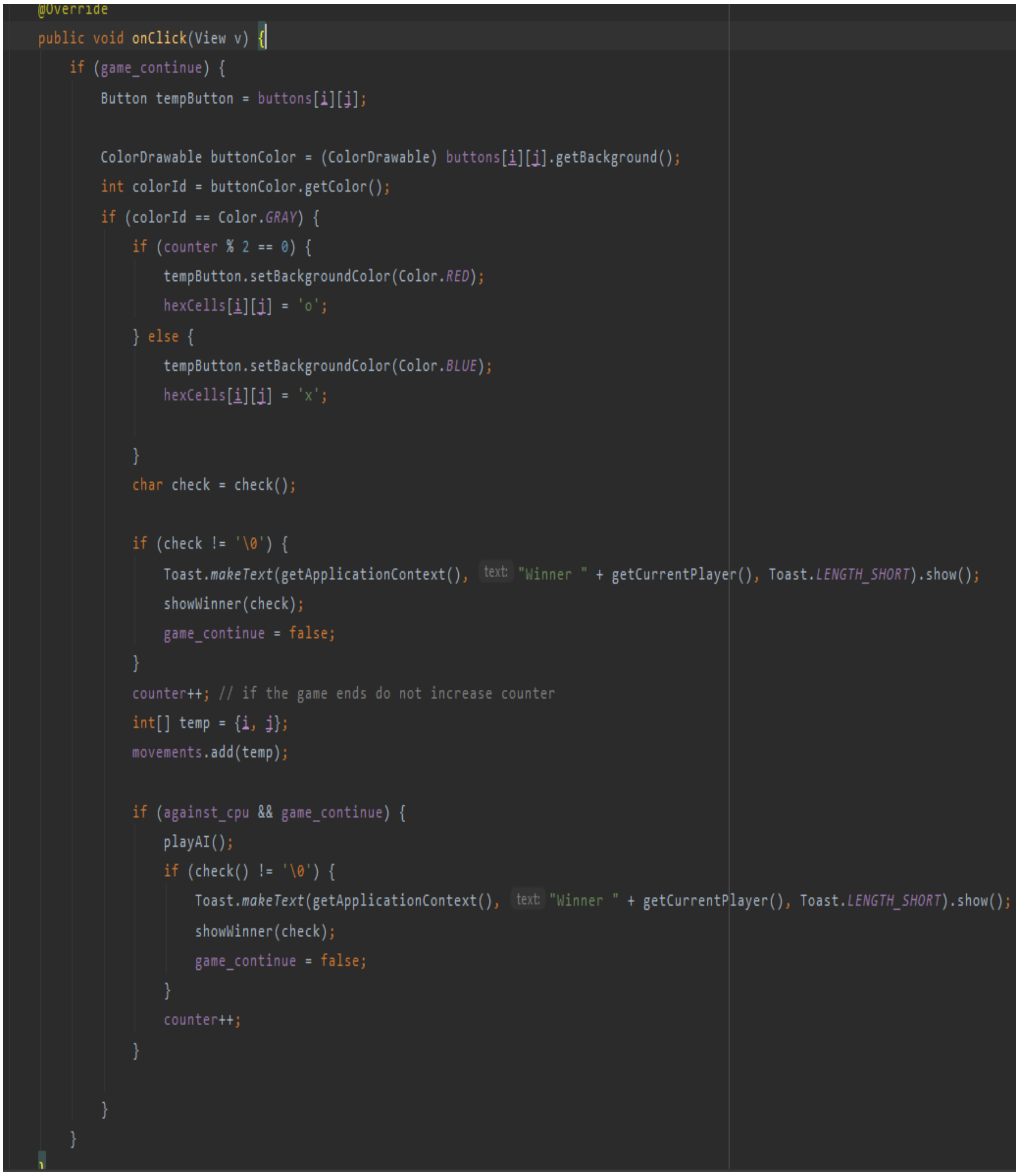
That's how I show buttons:



```
int shift = (int) (WidthOfButtons / 2.0);

for (int i = 0; i < size; i++) {
    //  buttons[i] = new Button[size];
    for (int j = 0; j < size; j++) {
        check_table[i][j] = new Boolean( value: false);
        buttons[i][j] = new Button( context: this);
        //RelativeLayout.LayoutParams layoutParams = new RelativeLayout.LayoutParams(ViewGroup.LayoutParams.WRAP_CONTENT,ViewGroup.LayoutParams.WRAP_CONTENT);
        buttons[i][j].setLayoutParams(new RelativeLayout.LayoutParams( w: WidthOfButtons - 5,  h: WidthOfButtons - 5));
        buttons[i][j].setX(j * WidthOfButtons + i * shift);
        buttons[i][j].setY(i * WidthOfButtons + yStart);

        //buttons[i][j].setText(".");
        buttons[i][j].setBackgroundColor(Color.GRAY);
        hexCells[i][j] = '.';
        relativeLayout.addView(buttons[i][j]);

        buttons[i][j].setId(i * size + j);
        buttons[i][j].setOnClickListener(getOnClick(i, j));

        //buttons[i][j].setWidth(5);
        //buttons[i][j].setHeight(5);


    }
}
```

To be able to make table portable to different platforms , I needed to reach size of the screen and I tried to fill it efficiently as I can.

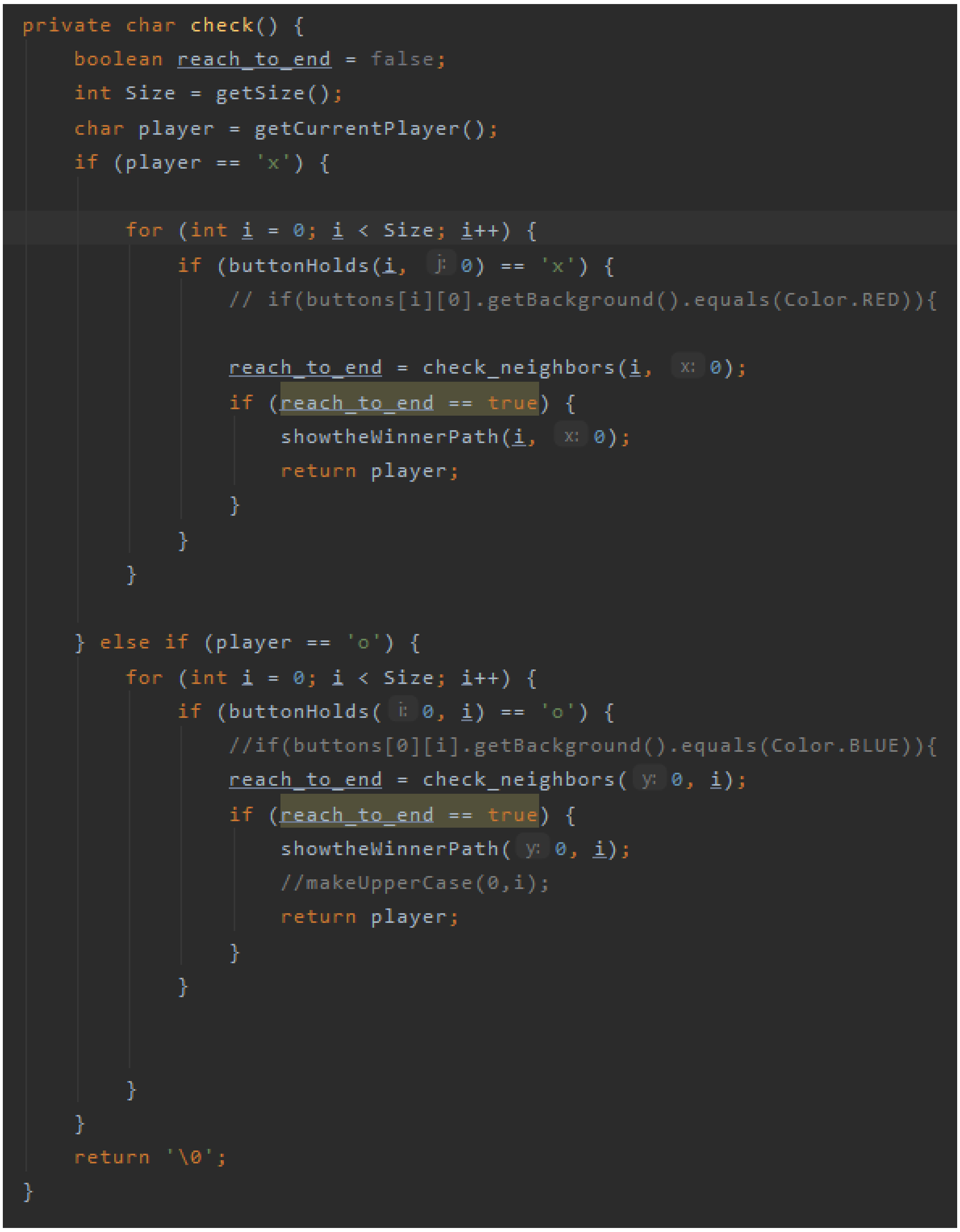
Game loop:



```
@Override
public void onClick(View v) {
    if (game_continue) {
        Button tempButton = buttons[i][j];

        ColorDrawable buttonColor = (ColorDrawable) buttons[i][j].getBackground();
        int colorId = buttonColor.getColor();
        if (colorId == Color.GRAY) {
            if (counter % 2 == 0) {
                tempButton.setBackgroundColor(Color.RED);
                hexCells[i][j] = 'o';
            } else {
                tempButton.setBackgroundColor(Color.BLUE);
                hexCells[i][j] = 'x';

            }
            char check = check();

            if (check != '\0') {
                Toast.makeText(getApplicationContext(),  text: "Winner " + getCurrentPlayer(), Toast.LENGTH_SHORT).show();
                showWinner(check);
                game_continue = false;
            }
            counter++; // if the game ends do not increase counter
            int[] temp = {i, j};
            movements.add(temp);

            if (against_cpu && game_continue) {
                playAI();
                if (check() != '\0') {
                    Toast.makeText(getApplicationContext(),  text: "Winner " + getCurrentPlayer(), Toast.LENGTH_SHORT).show();
                    showWinner(check);
                    game_continue = false;
                }
                counter++;
            }

        }
    }
}
```

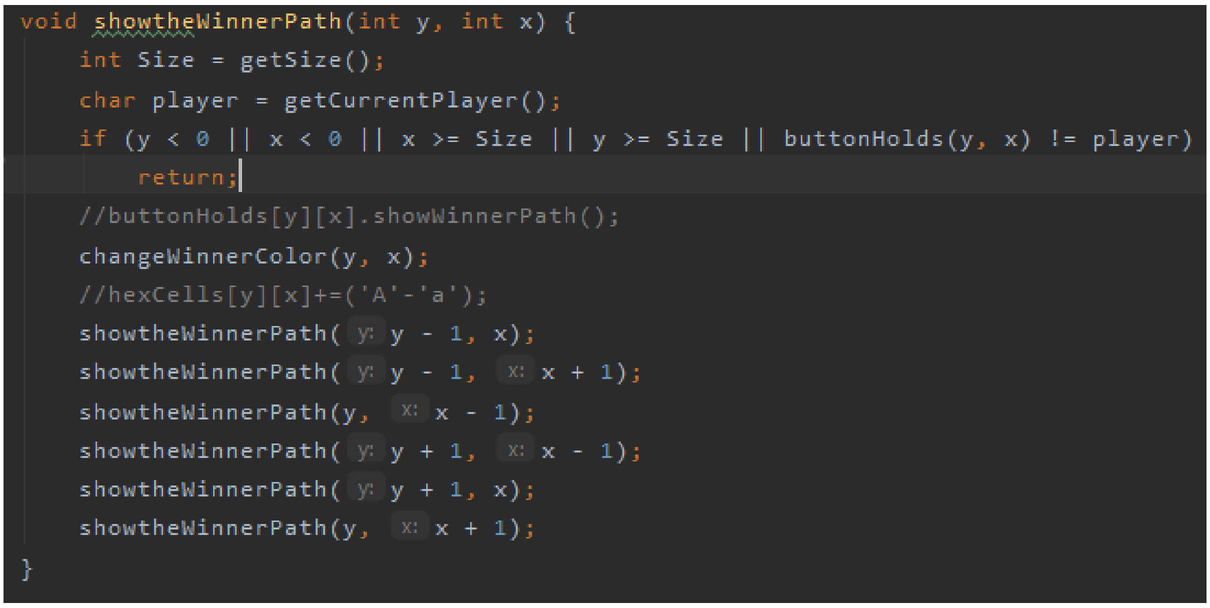
After every moves , checks if the game ended or not. If it does , breaks the loop. And it saves all moves so that ‘undo’ can work.

Check method:



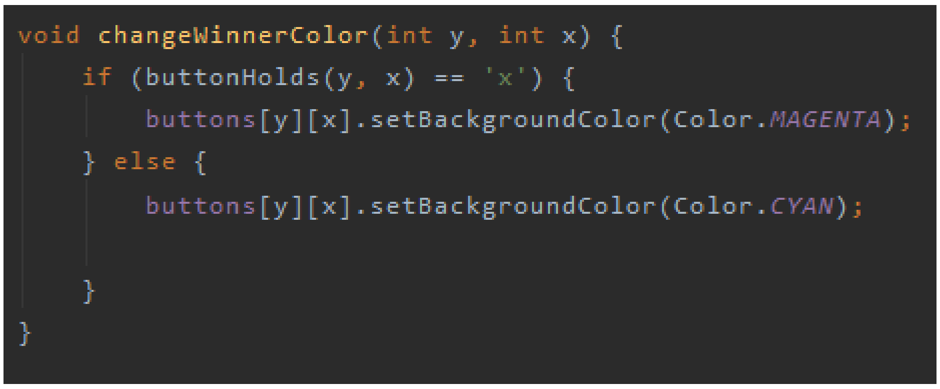
```
private char check() {
    boolean reach_to_end = false;
    int Size = getSize();
    char player = getCurrentPlayer();
    if (player == 'x') {

        for (int i = 0; i < Size; i++) {
            if (buttonHolds(i, j: 0) == 'x') {
                // if(buttons[i][0].getBackground().equals(Color.RED)){

                reach_to_end = check_neighbors(i, x: 0);
                if (reach_to_end == true) {
                    showtheWinnerPath(i, x: 0);
                    return player;
                }
            }
        }

    } else if (player == 'o') {
        for (int i = 0; i < Size; i++) {
            if (buttonHolds( i: 0, i) == 'o') {
                //if(buttons[0][i].getBackground().equals(Color.BLUE)){
                reach_to_end = check_neighbors( y: 0, i);
                if (reach_to_end == true) {
                    showtheWinnerPath( y: 0, i);
                    //makeUpperCase(0,i);
                    return player;
                }
            }


        }
    }
    return '\0';
}
```

Almost equal to the old check method. For red it starts from the left buttons and checks if it reaches to the right. For the blue is starts from the top and checks if it reaches to the bottom of the table.



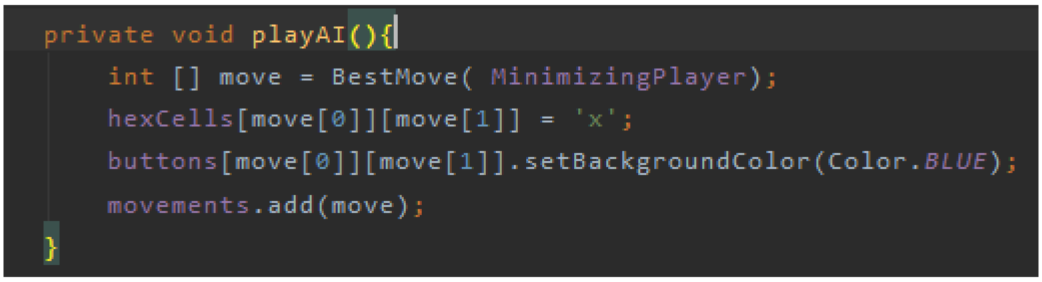
```java
void showtheWinnerPath(int y, int x) {
    int Size = getSize();
    char player = getCurrentPlayer();
    if (y < 0 || x < 0 || x >= Size || y >= Size || buttonHolds(y, x) != player)
        return;
    //buttonHolds[y][x].showWinnerPath();
    changeWinnerColor(y, x);
    //hexCells[y][x]+=('A'-'a');
    showtheWinnerPath( y: y - 1, x);
    showtheWinnerPath( y: y - 1,  x: x + 1);
    showtheWinnerPath(y,  x: x - 1);
    showtheWinnerPath( y: y + 1,  x: x - 1);
    showtheWinnerPath( y: y + 1, x);
    showtheWinnerPath(y,  x: x + 1);
}
```

When the game ends , this method shows the winner path as its name implies.
And this method change the color of the buttons which belongs to the winner path:



```java
void changeWinnerColor(int y, int x) {
    if (buttonHolds(y, x) == 'x') {
        buttons[y][x].setBackgroundColor(Color.MAGENTA);
    } else {
        buttons[y][x].setBackgroundColor(Color.CYAN);

    }
}
```

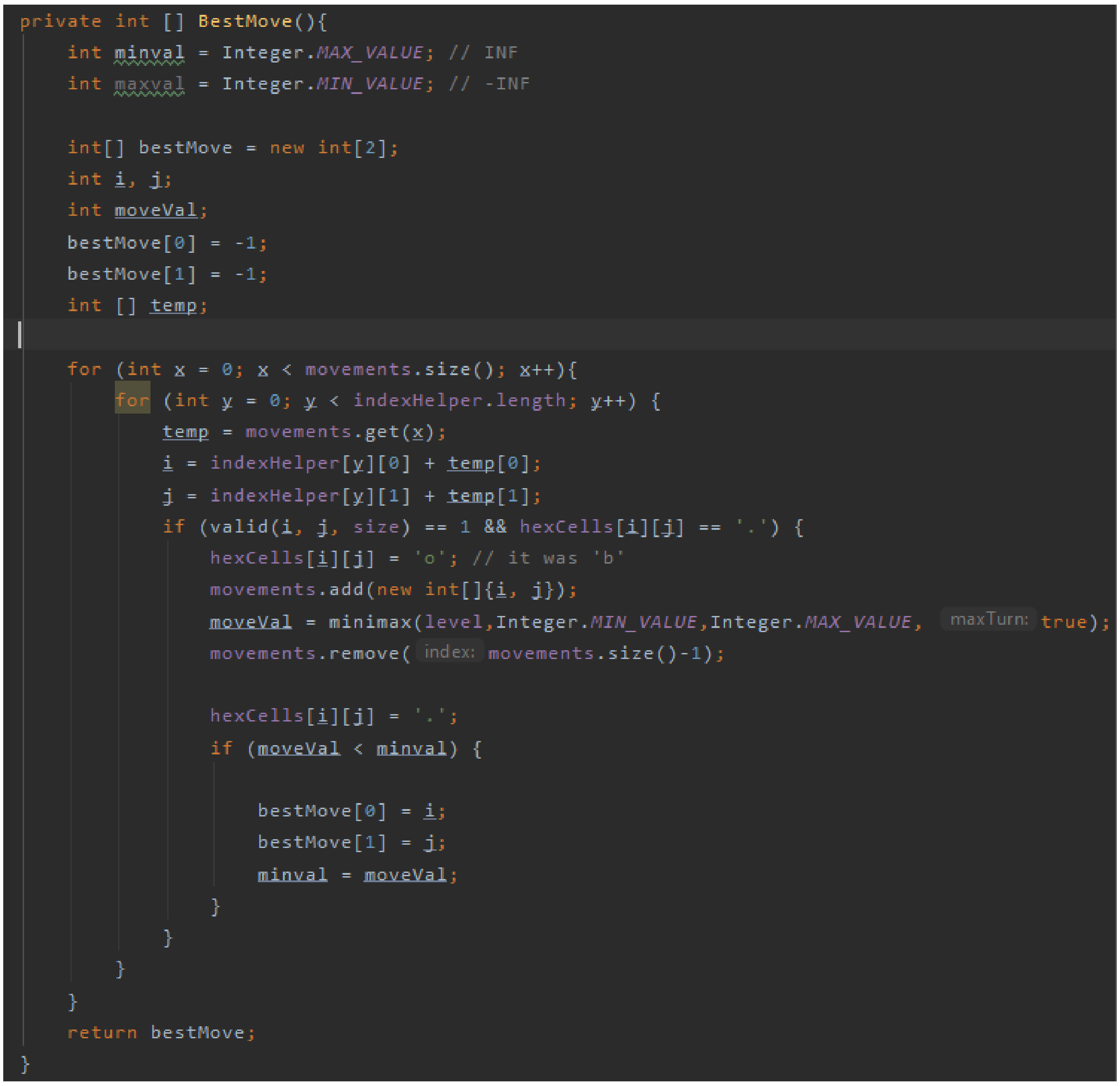
These are the functions to make computer create a move:



```java
  private void playAI(){
      int [] move = BestMove( MinimizingPlayer);
      hexCells[move[0]][move[1]] = 'x';
      buttons[move[0]][move[1]].setBackgroundColor(Color.BLUE);
      movements.add(move);
  }
```

This function takes the ‘Best Move’ ,according to the minimax algortihm, and makes it.

The BestMove Method:

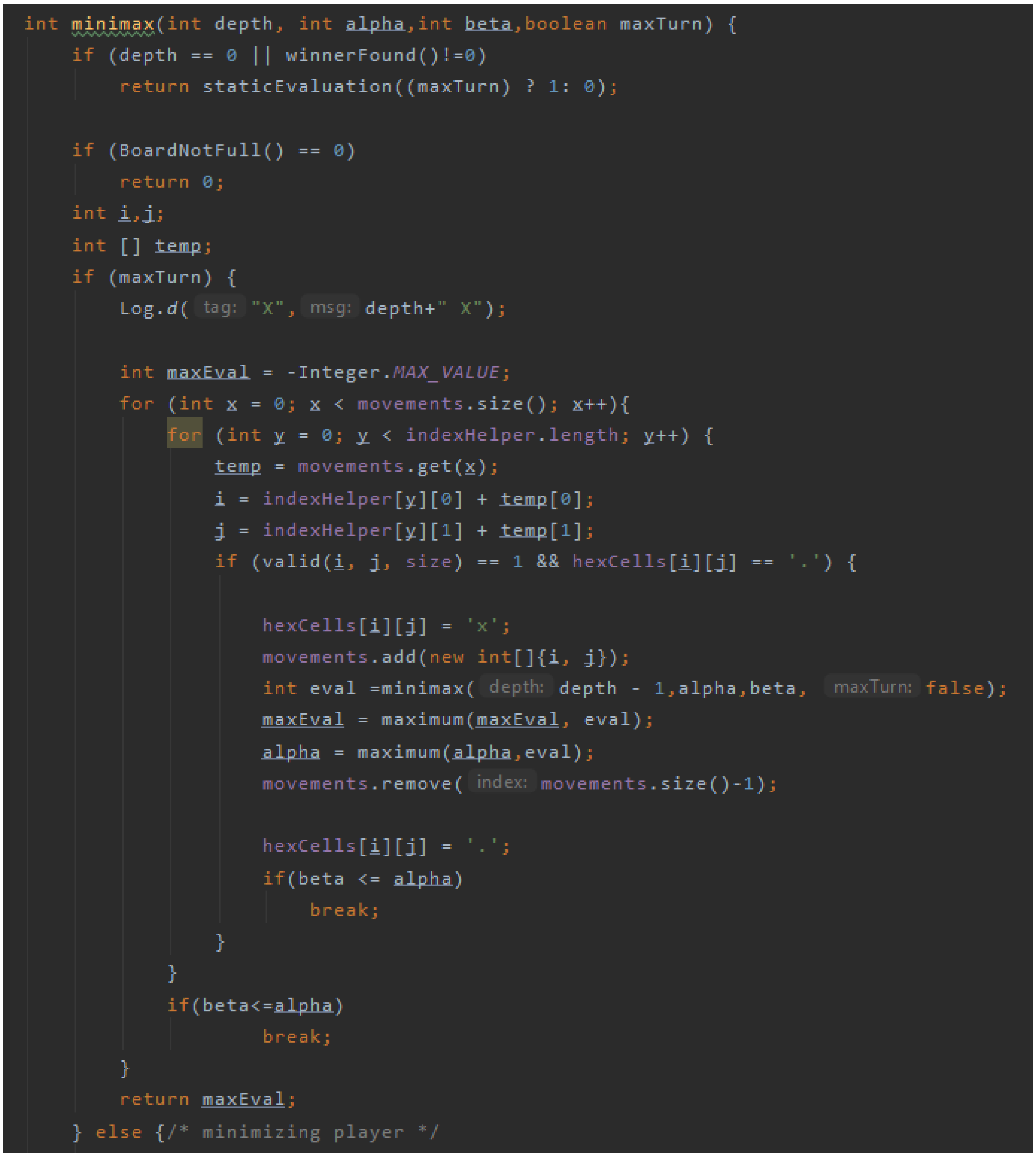


```java
private int [] BestMove(){
    int minval = Integer.MAX_VALUE; // INF
    int maxval = Integer.MIN_VALUE; // -INF

    int[] bestMove = new int[2];
    int i, j;
    int moveVal;
    bestMove[0] = -1;
    bestMove[1] = -1;
    int [] temp;

    for (int x = 0; x < movements.size(); x++){
        for (int y = 0; y < indexHelper.length; y++) {
            temp = movements.get(x);
            i = indexHelper[y][0] + temp[0];
            j = indexHelper[y][1] + temp[1];
            if (valid(i, j, size) == 1 && hexCells[i][j] == '.') {
                hexCells[i][j] = 'o'; // it was 'b'
                movements.add(new int[]{i, j});
                moveVal = minimax(level,Integer.MIN_VALUE,Integer.MAX_VALUE,  maxTurn: true);
                movements.remove( index: movements.size()-1);

                hexCells[i][j] = '.';
                if (moveVal < minval) {

                    bestMove[0] = i;
                    bestMove[1] = j;
                    minval = moveVal;
                }
            }
        }
    }
    return bestMove;
}
```

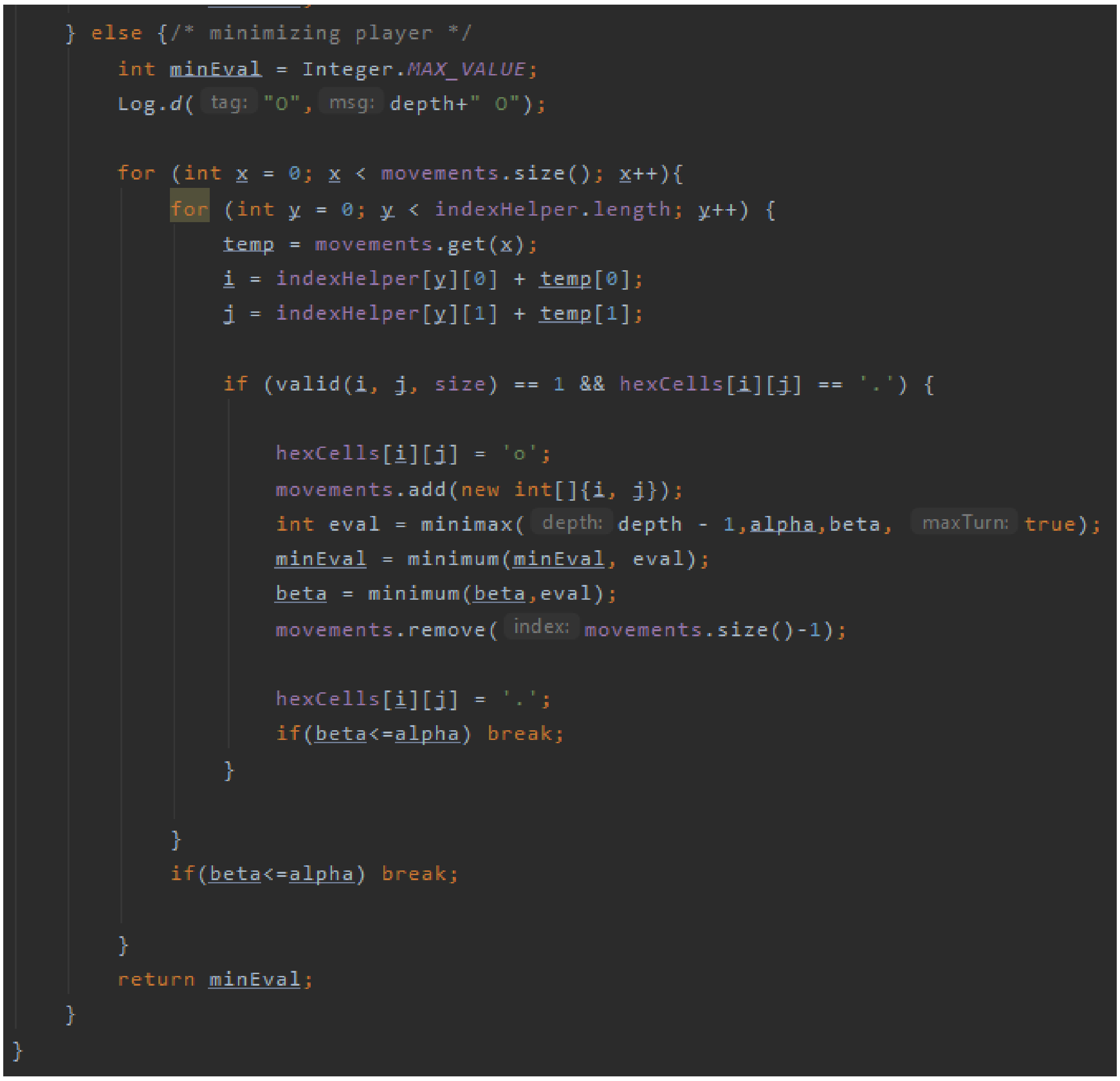
It calls the minimax algorithm for the every button which is neighbor of the buttons selected. Default best move method does it for the every button but as I mentioned earlier, that was very slow.

IndexHelper holds the x and y values which make current position to the reach every neighbor of it if these values are added to current x and y values.

Minimax Algorihm:



```
int minimax(int depth, int alpha,int beta,boolean maxTurn) {
    if (depth == 0 || winnerFound()!=0)
        return staticEvaluation((maxTurn) ? 1: 0);

    if (BoardNotFull() == 0)
        return 0;
    int i,j;
    int [] temp;
    if (maxTurn) {
        Log.d( tag: "X", msg: depth+" X");

        int maxEval = -Integer.MAX_VALUE;
        for (int x = 0; x < movements.size(); x++){
            for (int y = 0; y < indexHelper.length; y++) {
                temp = movements.get(x);
                i = indexHelper[y][0] + temp[0];
                j = indexHelper[y][1] + temp[1];
                if (valid(i, j, size) == 1 && hexCells[i][j] == '.') {

                    hexCells[i][j] = 'x';
                    movements.add(new int[]{i, j});
                    int eval =minimax( depth: depth - 1,alpha,beta,  maxTurn: false);
                    maxEval = maximum(maxEval, eval);
                    alpha = maximum(alpha,eval);
                    movements.remove( index: movements.size()-1);

                    hexCells[i][j] = '.';
                    if(beta <= alpha)
                        break;
                }
            }
            if(beta<=alpha)
                    break;
        }
        return maxEval;
    } else {/* minimizing player */
```



```
    } else {/* minimizing player */
        int minEval = Integer.MAX_VALUE;
        Log.d( tag: "O", msg: depth+" O");

        for (int x = 0; x < movements.size(); x++){
            for (int y = 0; y < indexHelper.length; y++) {
                temp = movements.get(x);
                i = indexHelper[y][0] + temp[0];
                j = indexHelper[y][1] + temp[1];

                if (valid(i, j, size) == 1 && hexCells[i][j] == '.') {

                    hexCells[i][j] = 'o';
                    movements.add(new int[]{i, j});
                    int eval = minimax( depth: depth - 1,alpha,beta,  maxTurn: true);
                    minEval = minimum(minEval, eval);
                    beta = minimum(beta,eval);
                    movements.remove( index: movements.size()-1);

                    hexCells[i][j] = '.';
                    if(beta<=alpha) break;
                }

            }
            if(beta<=alpha) break;

        }
        return minEval;
    }
}
```

It takes the depth (proportional to the level you selected) and alpha beta to be able to make function faster. And the maxturn value which shows that whose move is that.
Checks if the game ended or not if not continues. And for the every neighbor of the selected buttons it makes ‘depth’ number of moves and takes the best option according to the heuristic methods which also we took from the link provided.

3. Demo

   Here is the demo link :

   https://www.youtube.com/watch?v=MJD4q5WqTj4