HTML:

1) game_menu.html:

```
<!DOCTYPE html>
<html>
<head>
	<title>Go Game Menu</title>
	<link rel="stylesheet" href="./css/game_menu.css">
	<link rel="icon" type="image/png" href="./png/001-go.png"/>
</head>
<body>
	<h1>Go Game Menu</h1>
	<div class="container">
		<button class="btn" onclick="startGame('human')">Human vs Human</button>
		<button class="btn" onclick="startGame('bot')">Human vs Bot</button>
		<button class="btn" onclick="showRules()">Rules and Instructions</button>
	</div>

	<div id="rules-modal" class="modal" >
		<div class="modal-content">

			<span class="close" onclick="hideRules()">&times;</span>

			<h2><b>How to Play GO:</b></h2>

			<ul>
				<li>The objective of the game is for players to use the stones to form
territories by occupying the vacant areas on the board.</li><br>
				<li>At the beginning of the game, players should consider placing the
stones near handicap markers, usually located in the corners of the board. This way, the player is at
an advantage of gaining corner positions that help gain territory and are easy to defend.</li><br>
				<li>Players should only play stones at the edge of the board as they
are as easy to capture. Typically, the corner only needs two stones captured while side requires
three stones. The open area, however, requires players to seize for stones.</li><br>
				<li>If looking to occupy an open area, consider building off a stable
structure. You get to protect your stones and create a broader base for subsequent moves.</li><br>
				<li>Avoid placing your stones close to your opponent’s. You don’t
want to allow them to gain more considerable influence when you are chasing stones.</li><br>
				<li>Avoid placing your stones on your opponent’s territory. You are
only providing them with free stones for capture. This strategy works when you are confident of
capturing his stones.</li>
			</ul>

			<br><br>
```

```
                              <h2><b>Go Game Rules and Instructions:</b></h2>

                              <ul>
                                        <li>Go is a board game for two players, played on a 19x19 grid
board.</li><br>
                                        <li>The goal of the game is to control more territory than your
opponent by placing stones on the board.</li><br>
                                        <li>Players take turns placing stones on the board, with Black playing
first.</li><br>
                                        <li>Stones are black or white, and are placed on the intersections of
the board lines.</li><br>
                                        <li>Stones can be captured by surrounding them on all sides with
your opponent's stones, which removes them from the board.</li><br>
                                        <li>A liberty is an empty intersection adjacent to a stone.</li><br>
                                        <li>Stones have liberties equal to the number of adjacent empty
intersections.</li><br>
                                        <li>Stones or groups of stones that have no liberties left are removed
from the board as captured stones.</li><br>
                                        <li>The ko rule states that a player cannot capture back immediately
after their stone or group of stones has been captured. This is to prevent an endless cycle of
captures, where two players capture each other's stones back and forth repeatedly.</li>
                              </ul>


                    </div>

          </div>

          <script src="js/script.js"></script>
</body>
</html>
```

2) **human_game.html:**

```
<!DOCTYPE html>
<html>
<head>
          <meta http-equiv="content-type" content="text/html; charset=utf-8" />
          <title>Go Board</title>
          <link rel="stylesheet" href="css/style.css" />
          <link rel="icon" type="image/png" href="./png/001-go.png"/>

          <script type="text/javascript" src="js/go.js"></script>
          <script type="text/javascript" src="js/play1.js"></script>
          <script type="text/javascript" src="js/config.js"></script>
```

```
	<script type="text/javascript" src="js/script.js"></script>

</head>

<body>

	<div style="">
		<canvas id="weiqi" width="600" height="600"></canvas>
		<canvas id="path" width="600" height="600"></canvas>
	</div>
	<div class="parent">

		<div class="child"><h3 style="margin-left: 10px;">White</h3><h3
id="white-score" style="margin-left: 273px;">0</h3></div>

		<div class="child-1"><h3 id="black-score" style="margin-left: 8px;">0</h3><h3
style="margin-left: 270px; float: right;">Black</h3></div>

	</div>

	<div class="buttons">
		<button id="move_show">Show no</button>
		<button id="pass" data-no="0" onclick="">Pass</button>
		<button id="resign" type="button">Resign</button>
	</div>

	<!-- The Modal -->
	<div id="myModal" class="modal">
		<!-- Modal content -->
		<div class="modal-content">
		<span class="close">&times;</span>
		<p>Game Over! Thanks for playing.</p>
		</div>
	</div>


</body>
</html>
```

3) human_vs_bot.html:

```
<!DOCTYPE html>
<html>
```

```
<head>
        <meta http-equiv="content-type" content="text/html; charset=utf-8" />
        <title>Go Board</title>
        <link rel="stylesheet" href="css/style.css" />
        <link rel="icon" type="image/png" href="./png/001-go.png"/>

        <script type="text/javascript" src="js/go.js"></script>
        <script type="text/javascript" src="js/playbot.js"></script>
        <script type="text/javascript" src="js/config.js"></script>

</head>

<body>

        <div style="">
                <canvas id="weiqi" width="600" height="600"></canvas>
                <canvas id="path" width="600" height="600"></canvas>
        </div>

        <div class="parent">

                <div class="child"><h3 style="margin-left: 10px;">Bot</h3><h3 id="white-score"
style="margin-left: 273px;">0</h3></div>

                <div class="child-1"><h3 id="black-score" style="margin-left: 8px;">0</h3><h3
style="margin-left: 260px; float: right;">Human</h3></div>

        </div>

        <div class="buttons">
                <button id="move_show">Show no</button>
                <!-- <button id="pass" data-no="0" onclick="">Pass</button> -->
                <button id="resign" type="button">Resign</button>
        </div>


</body>
</html>
```

JAVASCRIPT:

1) **config.js:**

```
var move_show_flag = false;
function deal_button() {
	var move_show_button = document.getElementById("move_show");
	if (move_show_button) {
		move_show_button.onclick = function() {
			//alert(move_show_button);
			if (move_show_flag) {
				move_show_button.innerHTML="Show no";
				move_show_flag = false;
			} else {
				move_show_button.innerHTML="Showing";
				move_show_flag = true;
			}
			showPan();
		}
	}
}
addLoadEvent(deal_button);
```

2) **go.js:**

```
function grid(cxt) {
	// the first point is (30, 30)
	for (var i = 0; i < 9; i++) {
		cxt.beginPath();
		cxt.moveTo(0+59,   (i+1)*60);
		cxt.lineTo(600-59, (i+1)*60);
		cxt.stroke();
	}
	for (var i = 0; i < 9; i++) {
		cxt.beginPath();
		cxt.moveTo((i+1)*60,   0+59);
		cxt.lineTo((i+1)*60, 600-59);
		cxt.stroke();
	}

}
function ninePoints(cxt) {
	var np = new Array(
		[180,180],[300,180],[420,180],
		[180,300],[300,300],[420,300],
		[180,420],[300,420],[420,420]
```

```
        );

        for (var i = 0; i < np.length; i++) {
                //circle
                cxt.beginPath();
                cxt.arc(np[i][0],np[i][1],6,0,2*Math.PI,false);
                cxt.fillStyle="black";
                cxt.fill();
        }
}


var move_count = 0;
function mousedownHandler(e) {
        var x, y;
        if (e.offsetX || e.offsetX == 0) {
                x = e.offsetX; //- imageView.offsetLeft;
                y = e.offsetY; //- imageView.offsetTop;
        }
        if (x < 60-50 || x > 600-60)
                return;
        if (y < 60-50 || y > 600-60)
                return;

        var xok = false;
        var yok = false;
        var x_;
        var y_;
        for (var i = 1; i <= 9; i++) {
                if (x > i*60-30 && x < i*60+30) {
                        x = i*60;
                        xok = true;
                        x_ = i - 1;
                }
                if (y > i*60-30 && y < i*60+30) {
                        y = i*60;
                        yok = true;
                        y_ = i - 1;
                }
        }
        if (!xok || !yok)
                return;

        play(x_, y_, move_count);
        showPan();
        // now we put the new stone on the board
        /*
        move_count ++;
        var c = document.getElementById("weiqi");
        var cxt = c.getContext("2d");
        cxt.beginPath();
        cxt.arc(x,y,15,0,2*Math.PI,false);
        if (move_count % 2 == 1)
                cxt.fillStyle="black";
        else
                cxt.fillStyle="white";
        cxt.fill();
        */
}

function mousemoveHandler(e) {
        var x, y;
        if (e.offsetX || e.offsetX == 0) {
                x = e.offsetX ;//- imageView.offsetLeft;
                y = e.offsetY ;//- imageView.offsetTop;
        }
        if (x < 60-20 || x > 600-60)
                return;
        if (y < 60-20 || y > 600-60)
                return;

        var xok = false;
        var yok = false;
        for (var i = 1; i <= 9; i++) {
                if (x > i*60-20 && x < i*60+20) {
                        x = i*60;
                        xok = true;
                }
                if (y > i*60-20 && y < i*60+20) {
                        y = i*60;
                        yok = true;
                }
        }
        if (!xok || !yok)
                return;

        var c = document.getElementById("path");
        var cxt = c.getContext("2d");
        // clear the path
        cxt.clearRect(0,0,600,600);

        // put a new Gray stone
        cxt.beginPath();
        cxt.arc(x,y,15,0,2*Math.PI,false);
        cxt.fillStyle="gray";
        cxt.fill();

        cxt.beginPath();
        cxt.arc(x,y,30,0,2*Math.PI,false);
        if (move_count % 2 == 0)
                cxt.fillStyle="black";
        else
                cxt.fillStyle="#e0e0e0";
        cxt.fill();
}

function mouseoutHandler(e) {
        var c = document.getElementById("path");
        var cxt = c.getContext("2d");
        cxt.clearRect(0,0,600,600);
}

function initBoard() {
        var c_path = document.getElementById("path");
        c_path.addEventListener('mousedown', mousedownHandler, false);
        c_path.addEventListener('mousemove', mousemoveHandler, false);
        c_path.addEventListener('mouseout', mouseoutHandler, false);

        var c_weiqi = document.getElementById("weiqi");
        var cxt = c_weiqi.getContext("2d");
        cxt.fillStyle = "silver";
        cxt.fillRect(0,0,600,600);

        grid(cxt);
        ninePoints(cxt);

        showPan();
}

function addLoadEvent(func) {
        var oldonload = window.onload;
```

```
	if (typeof window.onload != 'function') {
		window.onload = func;
	} else {
		window.onload = function() {
			oldonload();
			func();
		}
	}
}
//window.addEventListener("load", initBoard, true);
addLoadEvent(initBoard);
```

3) **play1.js:**

```
/* some global values */
var pan = new Array(
	[0,0,0,0,0,0,0,0,0],
	[0,0,0,0,0,0,0,0,0],
	[0,0,0,0,0,0,0,0,0],
	[0,0,0,0,0,0,0,0,0],
	[0,0,0,0,0,0,0,0,0],
	[0,0,0,0,0,0,0,0,0],
	[0,0,0,0,0,0,0,0,0],
	[0,0,0,0,0,0,0,0,0],
	[0,0,0,0,0,0,0,0,0]


);
var shadow = new Array(
	[0,0,0,0,0,0,0,0,0],
	[0,0,0,0,0,0,0,0,0],
	[0,0,0,0,0,0,0,0,0],
	[0,0,0,0,0,0,0,0,0],
	[0,0,0,0,0,0,0,0,0],
	[0,0,0,0,0,0,0,0,0],
	[0,0,0,0,0,0,0,0,0],
	[0,0,0,0,0,0,0,0,0],
	[0,0,0,0,0,0,0,0,0]
);
var jie = new Array();
var move_record = new Array();
var passes = [];
```

```
function showPan() {
        var c = document.getElementById("weiqi");
        var cxt = c.getContext("2d");
        cxt.strokeStyle="black";


        cxt.clearRect(0,0,600,600);
        cxt.fillStyle = "#dcb35c";
        cxt.fillRect(0,0,600,600);
        grid(cxt);
        ninePoints(cxt);


        for (var i = 0; i < 9; i++) {
                for (var j = 0; j < 9; j++) {
                        if (pan[i][j] === 1) { //black
                                var rg = cxt.createRadialGradient((i+1)*30-3, (j+1)*30-3, 1,
(i+1)*30-4, (j+1)*30-4, 11);
                                rg.addColorStop(1, /*"black"*/"#202020");
                                rg.addColorStop(0, "gray");
                                cxt.beginPath();
                                cxt.arc((i+1)*60, (j+1)*60,30,0,2*Math.PI,false);
                                //cxt.fillStyle="black";
                                cxt.fillStyle=rg;
                                cxt.fill();

                        }
                        else if (pan[i][j] === 2) { //white
                                var rg = cxt.createRadialGradient((i+1)*30-3, (j+1)*30-3, 1,
(i+1)*30-4, (j+1)*30-4, 11);
                                rg.addColorStop(1, /*"lightgray"*/"#e0e0e0");
                                rg.addColorStop(0, "white");
                                cxt.beginPath();
                                cxt.arc((i+1)*60, (j+1)*60,30,0,2*Math.PI,false);
                                //cxt.fillStyle="white";
                                cxt.fillStyle=rg;
                                cxt.fill();
                        }
                        else if (pan[i][j] === 7) { // fill color
                                cxt.beginPath();
                                cxt.arc((i+1)*30, (j+1)*30,15,0,2*Math.PI,false);
                                cxt.fillStyle="red";
                                cxt.fill();
                        }
```

```
				}
			}

			if (move_show_flag) {
				for (var m = 0; m < move_record.length-1; m++) {

					if (pan[move_record[m][0]][move_record[m][1]] === 0)
						continue;


					var repeat_move_flag = false;
					for (var j = m+1; j < move_record.length; j++) {
						if (move_record[m][0] === move_record[j][0] &&
								move_record[m][1] === move_record[j][1]) {
							repeat_move_flag = true;
							break;
						}
					}
					if (repeat_move_flag)
						continue;


					if (move_record[m][2] % 2 === 1) { //black
						cxt.fillStyle="white";
					} else {
						cxt.fillStyle="black";
					}
					cxt.font="bold 18px sans-serif";
					if (move_record[m][2] > 99) {
						cxt.font="bold 16px sans-serif";
					}
					cxt.font="bold 16px sans-serif";
					cxt.textAlign="center";
					var move_msg = move_record[m][2].toString();
					//cxt.fillText(move_msg, (i+1)*30, (j+1)*30+6);
					cxt.fillText(move_msg, (move_record[m][0]+1)*60,
(move_record[m][1]+1)*60+6);

				}
			}

			if (move_record.length > 0) {
				cxt.fillStyle = "red";
				var newest_move = move_record.length-1;
```

```
                cxt.fillRect(
                        (move_record[newest_move][0]+1)*60-5,
                        (move_record[newest_move][1]+1)*60-5,
                        10, 10
                );
        }
}

function play(row, col) {
        if (row < 0 || row > 9 || col < 0 || col > 9) {
                alert("index error....");
                return;
        }

        if (pan[row][col] != 0) {

                return;
        }

        var can_down = false;

        var color = 2;
        if (move_count % 2 === 0) {
                color = 1;
        }

        if (!have_air(row, col)) {
                if (have_my_people(row, col)) {
                        make_shadow();


                        flood_fill(row, col, color);
                        if (fill_block_have_air(row, col, color)) {
                                can_down = true;
                                var dead_body = new Array();
                                can_eat(row, col, color, dead_body);
                                clean_dead_body(dead_body);
                        } else {
                                var dead_body = new Array();
                                var cret = can_eat(row, col, color, dead_body);
                                clean_dead_body(dead_body);

                                if (cret) {
                                        can_down = true;
                        } else {
                            alert("No liberty. Cannot place the stone!");
                        }
                    }
                } else {
                    var dead_body = new Array();
                    var cret = can_eat(row, col, color, dead_body);

                    if (cret) {
                        if (!is_jie(row, col, dead_body)) {
                            clean_dead_body(dead_body);
                            can_down = true;
                        } else {
                            alert("KO,you cannot place the stone.Please play another
move!");
                        }
                    }
                }
            } else {
                can_down = true;
                var dead_body = new Array();
                can_eat(row, col, color, dead_body);
                clean_dead_body(dead_body);
            }
            if (can_down) {
                stone_down(row, col);
            }
            var blacks = 0, whites = 0;
            for(var sizex = 0; sizex < 9; sizex++)
            {
                for(var sizey = 0; sizey < 9; sizey++)
                {
                    if(pan[sizex][sizey]==1)
                    blacks++;
                    else if(pan[sizex][sizey]==2)
                    whites++;
                }
                sizey = 0;


            }

            console.log(blacks,whites);
```

```
const colors = {
        WHITE: 2,
        BLACK: 1
 }

const direction = {
        TOP: 1,
        RIGHT: 2,
        BOTTOM: 3,
        LEFT: 4
 }

 const checkCurrentPoints = (newValues, boardSize) => {
        let whitePoints = 0;
        let blackPoints = 0;

        for (let y = 0; y < boardSize; y++) {
          const row = newValues[y];
          for (let x = 0; x < boardSize; x++) {
                const val = row[x];
                const currentColorVal = checkCurrentFieldPoint(newValues, val, y, x,
boardSize)
                if (currentColorVal === 1) {
                  whitePoints++
                } else if (currentColorVal === 2) {
                  blackPoints++
                }
          }
        }

        return [whitePoints, blackPoints]
 }
 /**
  * Determine for what color the point for this field should be assigned
  */
 const checkCurrentFieldPoint = (newValues, val, yVal, xVal, boardSize) => {
        if (val === 1) {
          return 1;
        } else if (val === 2) {
          return 2;
        } else if (val === 0) {
          let surroundedColorCheck;
          // Check top
          if (yVal != 0) {
                if (newValues[yVal-1][xVal] === 0) {
                  return 0;
                }
                surroundedColorCheck = newValues[yVal-1][xVal]
          }
          // Check right
          else if (xVal !== boardSize-1) {
                if (newValues[yVal][xVal+1] !== surroundedColorCheck) {
                  return 0;
                }
          }
          // Check bottom
          else if (yVal !== boardSize-1) {
                if (newValues[yVal+1][xVal] !== surroundedColorCheck) {
                  return 0;
                }
          }
          // Check left
          else if (xVal != 0) {
                if (newValues[yVal][xVal-1] !== surroundedColorCheck) {
                  return 0;
                }
          }

          // Pass checking and return color by which is surrounded
          return surroundedColorCheck;
        }
}

 const isFieldSurroundedByNothing = (newValues, yVal, xVal, boardSize) => {
        // Check top
        if (yVal != 0) {
          if (newValues[yVal-1][xVal] !== 0) {
                return false;
          }
        }
        // Check right
        if (xVal !== boardSize-1) {
          if (newValues[yVal][xVal+1] !== 0) {
                return false;
          }
        }
```

```
          // Check bottom
          if (yVal !== boardSize-1) {
            if (newValues[yVal+1][xVal] !== 0) {
                      return false;
            }
          }
          // Check left
          if (xVal != 0) {
            if (newValues[yVal][xVal-1] !== 0) {
                      return false;
            }
          }

          return true;
}

const isGameFinished = (newValues, boardSize) => {
          for (let y = 0; y < boardSize; y++) {
            const row = newValues[y];
            for (let x = 0; x < boardSize; x++) {
                      const val = row[x];
                      if (!isFieldDetermined(newValues, val, y, x, boardSize)) {
                        return false;
                      }
            }
          }
          return true;
}

/**
 * Check if field has inserted stone or around are just one color stones
 */
 const isFieldDetermined = (newValues, val, yVal, xVal, boardSize) => {
          // Check if is filled with any stone
          if (val !== 0) {
            return true;
          }
          return isFieldSurroundedByJustOneColor(newValues, yVal, xVal, boardSize)
}

 const isFieldSurroundedByJustOneColor = (newValues, yVal, xVal, boardSize) => {
          let isNeighborChecked = false;
          let isWhiteNeighbor = false;
          let isBlackNeighbor = false;
```

```
// Check top
if (yVal != 0) {
  if (newValues[yVal-1][xVal] === 0) {
          return false;
  } else if (newValues[yVal-1][xVal] === 1) {
          isWhiteNeighbor = true;
  } else {
          isBlackNeighbor = true;
  }
  isNeighborChecked = true;
}
// Check right
if (xVal !== boardSize-1) {
  if (newValues[yVal][xVal+1] === 0) {
          return false;
  } else if (newValues[yVal][xVal+1] === 1) {
          if (isNeighborChecked && isBlackNeighbor) {
            return false;
          }
          isWhiteNeighbor = true;
  } else {
          if (isNeighborChecked && isWhiteNeighbor) {
            return false;
          }
          isBlackNeighbor = true;
  }
  isNeighborChecked = true;
}
// Check bottom
if (yVal !== boardSize-1) {
  if (newValues[yVal+1][xVal] === 0) {
          return false;
  } else if (newValues[yVal+1][xVal] === 1) {
          if (isNeighborChecked && isBlackNeighbor) {
            return false;
          }
          isWhiteNeighbor = true;
  } else {
          if (isNeighborChecked && isWhiteNeighbor) {
            return false;
          }
          isBlackNeighbor = true;
  }
  isNeighborChecked = true;
```

```
			}
			// Check left
			if (xVal != 0) {
			  if (newValues[yVal][xVal-1] === 0) {
				return false;
			  } else if (newValues[yVal][xVal-1] === 1) {
				if (isNeighborChecked && isBlackNeighbor) {
				  return false;
				}
				// isWhiteNeighbor = true; // ALGORITHM OPTIMIZATION - not used
assignment
			  } else {
				if (isNeighborChecked && isWhiteNeighbor) {
				  return false;
				}
				// isBlackNeighbor = true; // ALGORITHM OPTIMIZATION - not used
assignment
			  }
			  // isNeighborChecked = true; // ALGORITHM OPTIMIZATION - not used
assignment
			}
			// If anywhere around there is no empty fields - say it is determined
			return true;
		}

		/**
		 * Evaluate how many stones each user have, including surrounded empty fileds
		 * USE IT JUST WHEN GAME IS FINISHED - it is optimized for this case and
otherwise it may return wrong results
		 */
		 const checkFinalPoints = (newValues, boardSize) => {
			let whitePoints = 0;
			let blackPoints = 0;

			for (let y = 0; y < boardSize; y++) {
			  const row = newValues[y];
			  for (let x = 0; x < boardSize; x++) {
				const val = row[x];
				const finalColorVal = checkFinalFieldPoint(newValues, val, y, x,
boardSize)
				if (finalColorVal === 1) {
				  whitePoints++
				} else {
				  blackPoints++
                    }
                }
            }

            return [whitePoints, blackPoints]
        }
        /**
         * Determine for what color the point for this field should be assigned
         */
        const checkFinalFieldPoint = (newValues, val, yVal, xVal, boardSize) => {
            if (val === 1) {
              return 1;
            } else if (val === 2) {
              return 2;
            } else if (val === 0) {
              // Check top
              if (yVal != 0) {
                    return newValues[yVal-1][xVal] === 1 ? 1 : 2
              }
              // Check right
              else if (xVal !== boardSize-1) {
                    return newValues[yVal][xVal+1] === 1 ? 1 : 2
              }
              // Check bottom
              else if (yVal !== boardSize-1) {
                    return newValues[yVal+1][xVal] === 1 ? 1 : 2
              }
              // Check left
              else if (xVal != 0) {
                    return newValues[yVal][xVal-1] === 1 ? 1 : 2
              }
            }
        }
        const evaluateBoard = (newValues, boardSize) => {
            let valuesWereChanged = false;
            let valuesAreChanged = false;
            // Opponent values might be surrounded in more then one place, thus need to be
calculated as much time as possible
            do {
              valuesAreChanged = false;
              // Iterate over the whole board
              for (let y = 0; y < boardSize; y++) {
                    const row = newValues[y];
                    let isChanged = false;
```

```
                        for (let x = 0; x < boardSize; x++) {
                          const val = row[x];
                          if (val != 0) {
                                  let [currentValuesChanged, evaluatedValues] =
evaluateCurrentValue(newValues, val, y, x, boardSize)
                                  isChanged = currentValuesChanged;
                                  // console.log(currentValuesChanged)
                                  if (currentValuesChanged) {
                                    valuesWereChanged = true;
                                    valuesAreChanged = true;
                                    newValues = evaluatedValues;
                                    // return [true, evaluatedValues] // temp....
                                    break;
                                  }
                          }
                        }
                        if (isChanged) {
                          break;
                        }
                }
            } while (valuesAreChanged)
            return [valuesWereChanged, newValues]
        }

        /**
         * Check if the current stone in given place is surrounded by opponent's stones
         */
        const evaluateCurrentValue = (newValues, val, yVal, xVal, boardSize) => {
            const opponentColor = val === 1 ? 2 : 1;
            let valuesAreChanged = false;
            // let checkedValues = [
            //   [0,0,0,0,0,0,0,0,0],
            //   [0,0,0,0,0,0,0,0,0],
            //   [0,0,0,0,0,0,0,0,0],
            //   [0,0,0,0,0,0,0,0,0],
            //   [0,0,0,0,0,0,0,0,0],
            //   [0,0,0,0,0,0,0,0,0],
            //   [0,0,0,0,0,0,0,0,0],
            //   [0,0,0,0,0,0,0,0,0],
            //   [0,0,0,0,0,0,0,0,0],
            // ]
            // Chack top
            if (yVal != 0) {
              if (newValues[yVal-1][xVal] === opponentColor) {
                        // console.log(`p 1 val:${val} yVal:${yVal} xVal:${xVal}`)
                        let [isSorrounded, obtainedValues] = evaluateOpponentValue(newValues,
val, yVal-1, xVal, direction.BOTTOM, boardSize)
                        if (isSorrounded) {
                          valuesAreChanged = true;
                          // console.log('pass 1')
                          newValues = obtainedValues;
                        }
                  }
            }
            // Check right
            if (xVal !== boardSize-1) {
                  if (newValues[yVal][xVal+1] === opponentColor) {
                        // console.log(`p 2 val:${val} yVal:${yVal} xVal:${xVal}`)
                        let [isSorrounded, obtainedValues] = evaluateOpponentValue(newValues,
val, yVal, xVal+1, direction.LEFT, boardSize)
                        if (isSorrounded) {
                          valuesAreChanged = true;
                          // console.log('pass 2')
                          newValues = obtainedValues;
                        }
                  }
            }
            // Check bottom
            if (yVal !== boardSize-1) {
                  if (newValues[yVal+1][xVal] === opponentColor) {
                        // console.log(`p 3 val:${val} yVal:${yVal} xVal:${xVal}`)
                        let [isSorrounded, obtainedValues] = evaluateOpponentValue(newValues,
val, yVal+1, xVal, direction.TOP, boardSize)
                        if (isSorrounded) {
                          valuesAreChanged = true;
                          // console.log('pass 3')
                          newValues = obtainedValues;
                        }
                  }
            }
            // Check left
            if (xVal != 0) {
                  if (newValues[yVal][xVal-1] === opponentColor) {
                        // console.log(`p 4 val:${val} yVal:${yVal} xVal:${xVal}`)
                        let [isSorrounded, obtainedValues] = evaluateOpponentValue(newValues,
val, yVal, xVal-1, direction.RIGHT, boardSize)
                        if (isSorrounded) {
                          valuesAreChanged = true;
```

```
                                // console.log('pass 4')
                                newValues = obtainedValues;
                              }
                         }
                       }
                       // Resluting Values after current value is placed
                       return [valuesAreChanged, newValues];
              }

              // If it will be refactored, probably possible to remove prevDirection and use just
checkedValues
              const evaluateOpponentValue = (newValues, val, yVal, xVal, prevDirection, boardSize,
checkedValuesMap) => {
                       if (!checkedValuesMap)
                         checkedValuesMap = Array.from({length: boardSize}, () =>
Array(boardSize).fill(0));
                       // let checkedValuesMap = [
                       //   [0,0,0,0,0,0,0,0,0],
                       //   [0,0,0,0,0,0,0,0,0],
                       //   [0,0,0,0,0,0,0,0,0],
                       //   [0,0,0,0,0,0,0,0,0],
                       //   [0,0,0,0,0,0,0,0,0],
                       //   [0,0,0,0,0,0,0,0,0],
                       //   [0,0,0,0,0,0,0,0,0],
                       //   [0,0,0,0,0,0,0,0,0],
                       //   [0,0,0,0,0,0,0,0,0],
                       // ]//checkedValues.map(arr => arr.slice());
                       checkedValuesMap[yVal][xVal] = 1;

                       const opponentColor = val === 1 ? 2 : 1;
                       // console.log(`eov val:${val} yVal:${yVal} xVal:${xVal}
prevDirect:${prevDirection}`)
                       let evaluatedValues = newValues.map(arr => arr.slice());

                       // Chack top
                       if (yVal != 0 && prevDirection !== direction.TOP &&
checkedValuesMap[yVal-1][xVal] != 1) {
                         // console.log(`eov val:${val} yVal:${yVal} xVal:${xVal}
prevDirect:${prevDirection} 1`)
                         if (evaluatedValues[yVal-1][xVal] === opponentColor) {
                              let [isSorrounded, obtainedValues, checkedValuesObtained] =
evaluateOpponentValue(evaluatedValues, val, yVal-1, xVal, direction.BOTTOM, boardSize,
checkedValuesMap.map(arr => arr.slice()))
                              if (!isSorrounded) {
```

```
                                          return [false, newValues];
                                        }
                                        evaluatedValues = obtainedValues;
                                        checkedValuesMap = checkedValuesObtained;
                              } else if (evaluatedValues[yVal-1][xVal] === 0) {
                                        return [false, newValues];
                              }
                            }
                            // Check right
                            if (xVal !== boardSize-1 && prevDirection !== direction.RIGHT &&
checkedValuesMap[yVal][xVal+1] != 1) {
                              // console.log(`eov val:${val} yVal:${yVal} xVal:${xVal}
prevDirect:${prevDirection} 2`)
                              if (evaluatedValues[yVal][xVal+1] === opponentColor) {
                                        let [isSorrounded, obtainedValues, checkedValuesObtained] =
evaluateOpponentValue(evaluatedValues, val, yVal, xVal+1, direction.LEFT, boardSize,
checkedValuesMap.map(arr => arr.slice()))
                                        if (!isSorrounded) {
                                          return [false, newValues];
                                        }
                                        evaluatedValues = obtainedValues;
                                        checkedValuesMap = checkedValuesObtained;
                              } else if (evaluatedValues[yVal][xVal+1] === 0) {
                                        return [false, newValues];
                              }
                            }
                            // Check bottom
                            if (yVal !== boardSize-1 && prevDirection !== direction.BOTTOM &&
checkedValuesMap[yVal+1][xVal] != 1) {
                              // console.log(`eov val:${val} yVal:${yVal} xVal:${xVal}
prevDirect:${prevDirection} 3`)
                              if (evaluatedValues[yVal+1][xVal] === opponentColor) {
                                        let [isSorrounded, obtainedValues, checkedValuesObtained] =
evaluateOpponentValue(evaluatedValues, val, yVal+1, xVal, direction.TOP, boardSize,
checkedValuesMap.map(arr => arr.slice()))
                                        if (!isSorrounded) {
                                          return [false, newValues];
                                        }
                                        evaluatedValues = obtainedValues;
                                        checkedValuesMap = checkedValuesObtained;
                              } else if (evaluatedValues[yVal+1][xVal] === 0) {
                                        return [false, newValues];
                              }
                            }
```

```
                    // Check left
                    if (xVal != 0 && prevDirection !== direction.LEFT &&
checkedValuesMap[yVal][xVal-1] != 1) {
                     // console.log(`eov val:${val} yVal:${yVal} xVal:${xVal}
prevDirect:${prevDirection} 4`)
                     if (evaluatedValues[yVal][xVal-1] === opponentColor) {
                         let [isSorrounded, obtainedValues, checkedValuesObtained] =
evaluateOpponentValue(evaluatedValues, val, yVal, xVal-1, direction.RIGHT, boardSize,
checkedValuesMap.map(arr => arr.slice()))
                         if (!isSorrounded) {
                           return [false, newValues];
                         }
                         evaluatedValues = obtainedValues;
                         checkedValuesMap = checkedValuesObtained;
                     } else if (evaluatedValues[yVal][xVal-1] === 0) {
                         return [false, newValues];
                     }
                    }
                    // If opponent is surrounded
                    evaluatedValues[yVal][xVal] = 0;
                    // console.log(evaluatedValues)
                    return [true, evaluatedValues, checkedValuesMap];
            }

          var pass_button = false;
          // var blackpass = false;
          // var whitepass = false;
          // const blackpasses = [];
          // const whitepasses = [];
          var pass = document.getElementById("pass");

               pass.onclick = function() {
                    //alert(move_show_button);
                    pass_button = true
                    if (pass_button) {
                         // pass.innerHTML="show no";
                         console.log(whited);
                         console.log(blacked);
                         // blackpasses.push(move_count);
                         if(move_count - passes[passes.length - 1] == 1){
                              // alert("Two Consecutive Passes!!!!!!!!");
                              if(blacked > whited){
                                   alert("Black has won the game");
document.getElementById('path').style.pointerEvents = 'none';
                                        }
                                        else if(whited > blacked){
                                                alert("White has won the game");

document.getElementById('path').style.pointerEvents = 'none';
                                        }
                                        else{
                                                alert("The game has ended in a tie as the scores
are level");
                                        }

                                        // move_count++;
                                }

                                else{

                                        passes.push(move_count);
                                        move_count++;
                                        pass_button = false;
                                        console.log(passes);

                                }
                        }
                }

        var resign = document.getElementById("resign");
                resign.onclick = function() {

                        document.getElementById('path').style.pointerEvents = 'none';

                        if(move_count%2==0){
                                alert("Black has resigned, White has won the game");
                        }
                        else{
                                alert("White has resigned, Black has won the game");
                        }

                }
}


// Implementing KO rule
```

```
function is_jie(row, col, dead_body) {
        if (dead_body.length === 1) {
                for (var i = 0; i < jie.length; i++) {
                        //If it meets (there is coordinates, and move_count is the first hand)

                        if (        jie[i][0] === dead_body[0][0] &&
                                        jie[i][1] === dead_body[0][1] &&
                                        jie[i][2] === move_count) {
                                return true;
                        }
                }
                //Add the record table
                jie.push([row, col, move_count+1]);
                return false;
        }
        return false;
}


function can_eat(row, col, color, dead_body) {
        var ret = false;
        var anti_color = 2;
        if (color === 2)
                anti_color = 1;

        if (row+1 <= 9-1 && pan[row+1][col] === anti_color) {
                make_shadow();
                shadow[row][col] = color;
                flood_fill(row+1, col, anti_color);
                if (!anti_fill_block_have_air(anti_color)) {


                        var rret = record_dead_body(dead_body);
                        ret = ret || rret;
                }

        }
        if (row-1 >= 0 && pan[row-1][col] === anti_color) {
                make_shadow();
                shadow[row][col] = color;
                flood_fill(row-1, col, anti_color);
                if (!anti_fill_block_have_air(anti_color)) {
                        var rret = record_dead_body(dead_body);
                        ret = ret || rret;
                }

        }
        if (col+1 <= 9-1 && pan[row][col+1] === anti_color) {
                make_shadow();
                shadow[row][col] = color;
                flood_fill(row, col+1, anti_color);
                if (!anti_fill_block_have_air(anti_color)) {
                        var rret = record_dead_body(dead_body);
                        ret = ret || rret;
                }

        }
        if (col-1 >= 0 && pan[row][col-1] === anti_color) {
                make_shadow();
                shadow[row][col] = color;
                flood_fill(row, col-1, anti_color);
                if (!anti_fill_block_have_air(anti_color)) {
                        var rret = record_dead_body(dead_body);
                        ret = ret || rret;
                }

        }
        return ret;
}

function record_dead_body(db) {
        var ret = false;
        for (var row = 0; row < shadow.length; row++) {
                for (var col = 0; col < shadow[row].length; col++) {
                        if (shadow[row][col] === 7) {
                                db.push([row, col]);
                                ret = true; // it's true have dead body
                                //alert("DEAD: "+(row).toString()+","+col.toString());
                        }
                }
        }
        return ret;
}
var whited = 0, blacked = 0;
function clean_dead_body(db) {

        for (var i = 0; i < db.length; i++) {
                if(pan[db[i][0]][db[i][1]] == 2)
                {
                        pan[db[i][0]][db[i][1]] = 0;
                   blacked = blacked + 1;
                }
                if(pan[db[i][0]][db[i][1]] == 1)
                {
                        pan[db[i][0]][db[i][1]] = 0;
                   whited = whited + 1;
                }
                //alert("OUT: "+(db[i][0]).toString()+","+(db[i][1]).toString());
        }

        document.getElementById("white-score").innerHTML = whited;
        document.getElementById("black-score").innerHTML = blacked;
        console.log(whited);
        console.log(blacked);
}

/* Is it free around the filling area? */
function fill_block_have_air(row, col, color) {
        for (var i = 0; i < pan.length; i++) {
                for (var j = 0; j < pan[i].length; j++) {
                        if (i !== row || j !== col) {
                                if (shadow[i][j] === 7 && pan[i][j] !== color) {
                                        return true; // This block is free, but it can be downloaded
                                }
                        }
                }
        }

        return false;
}

function anti_fill_block_have_air(color) {
        for (var i = 0; i < pan.length; i++) {
                for (var j = 0; j < pan[i].length; j++) {
                        if (shadow[i][j] === 7 && pan[i][j] !== color) {
                                return true; // live
                        }
                }
        }

        return false; //die
}
```

```
/*Creating a copy of the current state of the game board*/
function make_shadow() {
	for (var i = 0; i < pan.length; i++) {
		for (var j = 0; j < pan[i].length; j++) {
			shadow[i][j] = pan[i][j];
		}
	}
}

function shadow_to_pan() {
	for (var i = 0; i < pan.length; i++) {
		for (var j = 0; j < pan[i].length; j++) {
			pan[i][j] = shadow[i][j];
		}
	}
}

/* Flood filling, only the shadow board */
function flood_fill(row, col, color) {
	if (row < 0 || row > 9-1 || col < 0 || col > 9-1)
		return;

	var anti_color = 2;
	if (color === 2)
		anti_color = 1;

	if (shadow[row][col] !== anti_color && shadow[row][col] !== 7) {
		shadow[row][col] = 7; // Indicates that it has been filled
		flood_fill(row+1, col, color);
		flood_fill(row-1, col, color);
		flood_fill(row, col+1, color);
		flood_fill(row, col-1, color);
	}
}

/* Checking if there are any liberties around the stone for different scenarios*/
function have_air(row, col) {
	if (row > 0 && row < 9-1 && col > 0 && row < 9-1) {
		if (	pan[row+1][col] !== 0 &&
				pan[row-1][col] !== 0 &&
				pan[row][col+1] !== 0 &&
				pan[row][col-1] !== 0 ) {
			//alert("have no air");
```

```
                return false;
            } else {
                //alert("have air");
                return true;
            }
        } else if (row === 0 && col > 0 && col < 9-1) { // side
            if (    pan[row+1][col] !== 0 &&
                        pan[row][col+1] !== 0 &&
                        pan[row][col-1] !== 0 ) {
                //alert("have no air");
                return false;
            } else {
                //alert("have air");
                return true;
            }
        } else if (row === 9-1 && col > 0 && col < 9-1) {
            if (    pan[row-1][col] !== 0 &&
                        pan[row][col+1] !== 0 &&
                        pan[row][col-1] !== 0 ) {
                return false;
            } else {
                return true;
            }
        } else if (col === 0 && row > 0 && row < 9-1) {
            if (    pan[row][col+1] !== 0 &&
                        pan[row+1][col] !== 0 &&
                        pan[row-1][col] !== 0 ) {
                return false;
            } else {
                return true;
            }
        } else if (col === 9-1 && row > 0 && row < 9-1) {
            if (    pan[row][col-1] !== 0 &&
                        pan[row+1][col] !== 0 &&
                        pan[row-1][col] !== 0 ) {
                return false;
            } else {
                return true;
            }
        } else if (row === 0 && col === 0) {
            if (    pan[row][col+1] !== 0 &&
                        pan[row+1][col] !== 0) {
                return false;
            } else {
```

```
                        return true;
                }
        } else if (row === 0 && col === 9-1) {
                if (    pan[row][col-1] !== 0 &&
                                pan[row+1][col] !== 0) {
                        return false;
                } else {
                        return true;
                }
        } else if (row === 9-1 && col === 0) {
                if (    pan[row][col+1] !== 0 &&
                                pan[row-1][col] !== 0) {
                        return false;
                } else {
                        return true;
                }
        } else if (row === 9-1 && col === 9-1) {
                if (    pan[row][col-1] !== 0 &&
                                pan[row-1][col] !== 0) {
                        return false;
                } else {
                        return true;
                }
        }
}

/* Checking if stones near the current stone are the same color or not */
function have_my_people(row, col) {
        if (row > 0 && row < 9
                -1 && col > 0 && row < 9-1) {
                if (move_count % 2 === 0) { //Before placing a stone,it is white
                        if (    pan[row+1][col] === 1 ||
                                        pan[row-1][col] === 1 ||
                                        pan[row][col+1] === 1 ||
                                        pan[row][col-1] === 1 ) {
                                //alert("have my people");
                                return true;
                        }
                } else {
                        if (    pan[row+1][col] === 2 ||
                                        pan[row-1][col] === 2 ||
                                        pan[row][col+1] === 2 ||
                                        pan[row][col-1] === 2 ) {
                                //alert("have my people");
                        return true;
                    }
                }
            } else if (row === 0 && col > 0 && col < 9-1) { // side
                if (move_count % 2 === 0) {
                    if (    pan[row+1][col] === 1 ||
                            pan[row][col+1] === 1 ||
                            pan[row][col-1] === 1 ) {
                        //alert("have my people");
                        return true;
                    }
                } else {
                    if (    pan[row+1][col] === 2 ||
                            pan[row][col+1] === 2 ||
                            pan[row][col-1] === 2 ) {
                        //alert("have my people");
                        return true;
                    }
                }
            } else if (row === 9-1 && col > 0 && col < 9-1) { //side
                if (move_count % 2 === 0) {
                    if (    pan[row-1][col] === 1 ||
                            pan[row][col+1] === 1 ||
                            pan[row][col-1] === 1 ) {
                        //alert("have my people");
                        return true;
                    }
                } else {
                    if (    pan[row-1][col] === 2 ||
                            pan[row][col+1] === 2 ||
                            pan[row][col-1] === 2 ) {
                        //alert("have my people");
                        return true;
                    }
                }
            } else if (col === 9-1 && row > 0 && row < 9-1) {
                if (move_count % 2 === 0) {
                    if (    pan[row+1][col] === 1 ||
                            pan[row-1][col] === 1 ||
                            pan[row][col-1] === 1 ) {
                        //alert("have my people");
                        return true;
                    }
                } else {
```

```
            if (    pan[row+1][col] === 2 ||
                        pan[row-1][col] === 2 ||
                        pan[row][col-1] === 2 ) {
                //alert("have my people");
                return true;
            }
        }
    } else if (col === 0 && row > 0 && row < 9-1) {
        if (move_count % 2 === 0) {
            if (    pan[row+1][col] === 1 ||
                        pan[row-1][col] === 1 ||
                        pan[row][col+1] === 1 ) {
                //alert("have my people");
                return true;
            }
        } else {
            if (    pan[row+1][col] === 2 ||
                        pan[row-1][col] === 2 ||
                        pan[row][col+1] === 2 ) {
                //alert("have my people");
                return true;
            }
        }
    } else if (row === 0 && col === 0) { // Corner
        if (move_count % 2 === 0) {
            if (    pan[row+1][col] === 1 ||
                        pan[row][col+1] === 1 ) {
                //alert("have my people");
                return true;
            }
        } else {
            if (    pan[row+1][col] === 2 ||
                        pan[row][col+1] === 2 ) {
                //alert("have my people");
                return true;
            }
        }
    } else if (row === 0 && col === 9-1) {
        if (move_count % 2 === 0) {
            if (    pan[row+1][col] === 1 ||
                        pan[row][col-1] === 1 ) {
                //alert("have my people");
                return true;
            }
```

```
			} else {
				if (	pan[row+1][col] === 2 ||
						pan[row][col-1] === 2 ) {
					//alert("have my people");
					return true;
				}
			}
		} else if (row === 9-1 && col === 0) {
			if (move_count % 2 === 0) {
				if (	pan[row-1][col] === 1 ||
						pan[row][col+1] === 1 ) {
					//alert("have my people");
					return true;
				}
			} else {
				if (	pan[row-1][col] === 2 ||
						pan[row][col+1] === 2 ) {
					//alert("have my people");
					return true;
				}
			}
		} else if (row === 9-1 && col === 9-1) {
			if (move_count % 2 === 0) {
				if (	pan[row-1][col] === 1 ||
						pan[row][col-1] === 1 ) {
					//alert("have my people");
					return true;
				}
			} else {
				if (	pan[row-1][col] === 2 ||
						pan[row][col-1] === 2 ) {
					//alert("have my people");
					return true;
				}
			}
		}

		return false;
	}


	function stone_down(row, col) {
		if (move_count % 2 === 0) {
			pan[row][col] = 1;
```

```
        } else {
                pan[row][col] = 2;
        }
        move_count ++;
        move_record.push([row, col, move_count]);
}
```

4) **playbot.js:**

```
/*
        description: Realize the move logic of Go (including the handling of picking and robbing)
*/

/* some global values */
var pan = new Array(
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0]
);

var shadow = new Array(
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0]
);

var jie = new Array();
var move_record = new Array();
var blackpasses = [];
var whitepasses = [];
```

```
// creating the stones using HTML canvas
function showPan() {
        var c = document.getElementById("weiqi");
        var cxt = c.getContext("2d");
        cxt.strokeStyle="black";

        /* Empty, redraw lines, etc. */
        cxt.clearRect(0,0,600,600);
        cxt.fillStyle = "#dcb35c";
        cxt.fillRect(0,0,600,600);
        grid(cxt);
        ninePoints(cxt);


        for (var i = 0; i < 9; i++) {
                for (var j = 0; j < 9; j++) {
                        if (pan[i][j] === 1) { //black
                                var rg = cxt.createRadialGradient((i+1)*30-3, (j+1)*30-3, 1,
(i+1)*30-4, (j+1)*30-4, 11);
                                rg.addColorStop(1, /*"black"*/"#202020");
                                rg.addColorStop(0, "gray");
                                cxt.beginPath();
                                cxt.arc((i+1)*60, (j+1)*60,30,0,2*Math.PI,false);

                                //cxt.fillStyle="black";
                                cxt.fillStyle=rg;
                                cxt.fill();
                        }
                        else if (pan[i][j] === 2) { //white
                                var rg = cxt.createRadialGradient((i+1)*30-3, (j+1)*30-3, 1,
(i+1)*30-4, (j+1)*30-4, 11);
                                rg.addColorStop(1, /*"lightgray"*/"#e0e0e0");
                                rg.addColorStop(0, "white");
                                cxt.beginPath();
                                cxt.arc((i+1)*60, (j+1)*60,30,0,2*Math.PI,false);

                                //cxt.fillStyle="white";
                                cxt.fillStyle=rg;
                                cxt.fill();
                        }
                        else if (pan[i][j] === 7) { // fill color
                                cxt.beginPath();
                                cxt.arc((i+1)*30, (j+1)*30,15,0,2*Math.PI,false);
                                cxt.fillStyle="red";
```

```
                    cxt.fill();
                }
            }
        }

        // Display hand number
        if (move_show_flag) {
            for (var m = 0; m < move_record.length-1; m++) { //The most recent hand/move is
marked with red square

                // First judge whether the piece is still on the board
                if (pan[move_record[m][0]][move_record[m][1]] === 0)
                    continue;

                // Displaying only the recent move on the canvas
                var repeat_move_flag = false;
                for (var j = m+1; j < move_record.length; j++) {
                    if (move_record[m][0] === move_record[j][0] &&
                            move_record[m][1] === move_record[j][1]) {
                        repeat_move_flag = true;
                        break;
                    }
                }
                if (repeat_move_flag)
                    continue;

                // Drawing the text(move no.) on the stone
                if (move_record[m][2] % 2 === 1) { //black
                    cxt.fillStyle="white";
                } else {
                    cxt.fillStyle="black";
                }
                cxt.font="bold 18px sans-serif";
                if (move_record[m][2] > 99) {
                    cxt.font="bold 16px sans-serif";
                }
                cxt.font="bold 16px sans-serif";
                cxt.textAlign="center";
                var move_msg = move_record[m][2].toString();
                //cxt.fillText(move_msg, (i+1)*30, (j+1)*30+6);
                cxt.fillText(move_msg, (move_record[m][0]+1)*60,
(move_record[m][1]+1)*60+6);

            }
        }
        // Highlighting most recent move with red square
        if (move_record.length > 0) {
                cxt.fillStyle = "red";
                var newest_move = move_record.length-1;
                cxt.fillRect(
                        (move_record[newest_move][0]+1)*60-5,
                        (move_record[newest_move][1]+1)*60-5,
                        10, 10
                );
        }
}

function play(row, col) {
        if (row < 0 || row > 9 || col < 0 || col > 9) {
                alert("index error....");
                return;
        }
        // Checking if stone is already present on the coordinates
        if (pan[row][col] != 0) {
                return;
        }

        var can_down = false; // Can the player make a move
        var color = 2; // white
        if (move_count % 2 === 0) { // Setting up the color for the next move
                color = 1;
        }

        if (!have_air(row, col)) {
                if (have_my_people(row, col)) {
                        make_shadow();


                        flood_fill(row, col, color);
                        if (fill_block_have_air(row, col, color)) {
                                can_down = true;
                                var dead_body = new Array();
                                can_eat(row, col, color, dead_body);
                                clean_dead_body(dead_body);
                        } else {
                                var dead_body = new Array();
                                var cret = can_eat(row, col, color, dead_body);
                                clean_dead_body(dead_body);
                        if (cret) {
                            can_down = true;
                        } else {
                            alert("No liberty. Cannot place the stone!!");
                        }
                    }
                } else {
                    var dead_body = new Array();
                    var cret = can_eat(row, col, color, dead_body);

                    // KO rule
                    if (cret) {
                        // Checking KO rule
                        if (!is_jie(row, col, dead_body)) {
                            // removing the captured stones
                            clean_dead_body(dead_body);
                            can_down = true;
                        } else {
                            alert("KO,you cannot place the stone.Please play another
move!");
                        }
                    }
                }
            } else {
                can_down = true;
                var dead_body = new Array();
                can_eat(row, col, color, dead_body);
                clean_dead_body(dead_body);
            }
            if (can_down) {
                stone_down(row, col);
            }
            var blacks = 0, whites = 0;
            for(var sizex = 0; sizex < 9; sizex++)
            {
                for(var sizey = 0; sizey < 9; sizey++)
                {
                    if(pan[sizex][sizey]==1)
                    blacks++;
                    else if(pan[sizex][sizey]==2)
                    whites++;
                }
                sizey = 0;
```

```
        }

        console.log(blacks,whites);

        const colors = {
                WHITE: 2,
                BLACK: 1
         }

        const direction = {
                TOP: 1,
                RIGHT: 2,
                BOTTOM: 3,
                LEFT: 4
         }

         const checkCurrentPoints = (newValues, boardSize) => { //newValues = Current board
state
                let whitePoints = 0;
                let blackPoints = 0;

                for (let y = 0; y < boardSize; y++) {
                  const row = newValues[y];
                  for (let x = 0; x < boardSize; x++) {
                        const val = row[x];
                        const currentColorVal = checkCurrentFieldPoint(newValues, val, y, x,
boardSize)
                        // field occupied by white stones if currentColorVal = 1
                        if (currentColorVal === 1) {
                          whitePoints++
                        } else if (currentColorVal === 2) {
                          blackPoints++
                        }
                  }
                }

                return [whitePoints, blackPoints]
         }
         /**
          * Determine for what color the point for this field should be assigned
          */
         const checkCurrentFieldPoint = (newValues, val, yVal, xVal, boardSize) => {
                if (val === 1) {
                  return 1;
                } else if (val === 2) {
                  return 2;
                } else if (val === 0) {
                  let surroundedColorCheck;
                  // Check top
                  if (yVal != 0) {
                        if (newValues[yVal-1][xVal] === 0) {
                          return 0;
                        }
                        surroundedColorCheck = newValues[yVal-1][xVal]
                  }
                  // Check right
                  else if (xVal !== boardSize-1) {
                        if (newValues[yVal][xVal+1] !== surroundedColorCheck) {
                          return 0;
                        }
                  }
                  // Check bottom
                  else if (yVal !== boardSize-1) {
                        if (newValues[yVal+1][xVal] !== surroundedColorCheck) {
                          return 0;
                        }
                  }
                  // Check left
                  else if (xVal != 0) {
                        if (newValues[yVal][xVal-1] !== surroundedColorCheck) {
                          return 0;
                        }
                  }

                  // Pass checking and return color by which is surrounded
                  return surroundedColorCheck;
                }
         }

          const isFieldSurroundedByNothing = (newValues, yVal, xVal, boardSize) => {
                // Check top
                if (yVal != 0) {
                  if (newValues[yVal-1][xVal] !== 0) {
                        return false;
                  }
                }
                // Check right
                if (xVal !== boardSize-1) {
                  if (newValues[yVal][xVal+1] !== 0) {
                            return false;
                  }
                }
                // Check bottom
                if (yVal !== boardSize-1) {
                  if (newValues[yVal+1][xVal] !== 0) {
                            return false;
                  }
                }
                // Check left
                if (xVal != 0) {
                  if (newValues[yVal][xVal-1] !== 0) {
                            return false;
                  }
                }

                return true;
         }

         const isGameFinished = (newValues, boardSize) => {
                for (let y = 0; y < boardSize; y++) {
                  const row = newValues[y];
                  for (let x = 0; x < boardSize; x++) {
                            const val = row[x];
                            if (!isFieldDetermined(newValues, val, y, x, boardSize)) {
                              return false;
                            }
                  }
                }
                return true;
         }

         /**
          * Check if field has inserted stone or around are just one color stones
          */
          const isFieldDetermined = (newValues, val, yVal, xVal, boardSize) => {
                // Check if is filled with any stone
                if (val !== 0) {
                  return true;
                }
                return isFieldSurroundedByJustOneColor(newValues, yVal, xVal, boardSize)
         }
```

```
const isFieldSurroundedByJustOneColor = (newValues, yVal, xVal, boardSize) => {
        let isNeighborChecked = false;
        let isWhiteNeighbor = false;
        let isBlackNeighbor = false;
        // Check top
        if (yVal != 0) {
          if (newValues[yVal-1][xVal] === 0) {
                return false;
          } else if (newValues[yVal-1][xVal] === 1) {
                isWhiteNeighbor = true;
          } else {
                isBlackNeighbor = true;
          }
          isNeighborChecked = true;
        }
        // Check right
        if (xVal !== boardSize-1) {
          if (newValues[yVal][xVal+1] === 0) {
                return false;
          } else if (newValues[yVal][xVal+1] === 1) {
                if (isNeighborChecked && isBlackNeighbor) {
                  return false;
                }
                isWhiteNeighbor = true;
          } else {
                if (isNeighborChecked && isWhiteNeighbor) {
                  return false;
                }
                isBlackNeighbor = true;
          }
          isNeighborChecked = true;
        }
        // Check bottom
        if (yVal !== boardSize-1) {
          if (newValues[yVal+1][xVal] === 0) {
                return false;
          } else if (newValues[yVal+1][xVal] === 1) {
                if (isNeighborChecked && isBlackNeighbor) {
                  return false;
                }
                isWhiteNeighbor = true;
          } else {
                if (isNeighborChecked && isWhiteNeighbor) {
                  return false;
                }
                isBlackNeighbor = true;
           }
           isNeighborChecked = true;
          }
          // Check left
          if (xVal != 0) {
           if (newValues[yVal][xVal-1] === 0) {
                return false;
           } else if (newValues[yVal][xVal-1] === 1) {
                if (isNeighborChecked && isBlackNeighbor) {
                  return false;
                }
                // isWhiteNeighbor = true; // ALGORITHM OPTIMIZATION - not used
assignment
           } else {
                if (isNeighborChecked && isWhiteNeighbor) {
                  return false;
                }
                // isBlackNeighbor = true; // ALGORITHM OPTIMIZATION - not used
assignment
           }
           // isNeighborChecked = true; // ALGORITHM OPTIMIZATION - not used
assignment
          }
          // If anywhere around there is no empty fields - say it is determined
          return true;
     }

     /**
      * Evaluate how many stones each user have, including surrounded empty fileds
      * USE IT JUST WHEN GAME IS FINISHED - it is optimized for this case and
otherwise it may return wrong results
      */
      const checkFinalPoints = (newValues, boardSize) => {
          let whitePoints = 0;
          let blackPoints = 0;

          for (let y = 0; y < boardSize; y++) {
           const row = newValues[y];
           for (let x = 0; x < boardSize; x++) {
                const val = row[x];
                const finalColorVal = checkFinalFieldPoint(newValues, val, y, x,
boardSize)
```

```
                    if (finalColorVal === 1) {
                      whitePoints++
                    } else {
                      blackPoints++
                    }
                  }
                }

                return [whitePoints, blackPoints]
        }
        /**
         * Determine for what color the point for this field should be assigned
         */
        const checkFinalFieldPoint = (newValues, val, yVal, xVal, boardSize) => {
                if (val === 1) {
                  return 1;
                } else if (val === 2) {
                  return 2;
                } else if (val === 0) {
                  // Check top
                  if (yVal != 0) {
                        return newValues[yVal-1][xVal] === 1 ? 1 : 2
                  }
                  // Check right
                  else if (xVal !== boardSize-1) {
                        return newValues[yVal][xVal+1] === 1 ? 1 : 2
                  }
                  // Check bottom
                  else if (yVal !== boardSize-1) {
                        return newValues[yVal+1][xVal] === 1 ? 1 : 2
                  }
                  // Check left
                  else if (xVal != 0) {
                        return newValues[yVal][xVal-1] === 1 ? 1 : 2
                  }
                }
        }
        const evaluateBoard = (newValues, boardSize) => {
                let valuesWereChanged = false;
                let valuesAreChanged = false;
                // Opponent values might be surrounded in more then one place, thus need to be
calculated as much time as possible
                do {
                  valuesAreChanged = false;
                // Iterate over the whole board
                for (let y = 0; y < boardSize; y++) {
                        const row = newValues[y];
                        let isChanged = false;
                        for (let x = 0; x < boardSize; x++) {
                          const val = row[x];
                          if (val != 0) {
                                  let [currentValuesChanged, evaluatedValues] =
evaluateCurrentValue(newValues, val, y, x, boardSize)
                                  isChanged = currentValuesChanged;
                                  // console.log(currentValuesChanged)
                                  if (currentValuesChanged) {
                                    valuesWereChanged = true;
                                    valuesAreChanged = true;
                                    newValues = evaluatedValues;
                                    // return [true, evaluatedValues] // temp....
                                    break;
                                  }
                          }
                        }
                        if (isChanged) {
                          break;
                        }
                }
              } while (valuesAreChanged)
              return [valuesWereChanged, newValues]
      }

      /**
       * Check if the current stone in given place is surrounded by opponent's stones
       */
      const evaluateCurrentValue = (newValues, val, yVal, xVal, boardSize) => {
              const opponentColor = val === 1 ? 2 : 1;
              let valuesAreChanged = false;
              // let checkedValues = [
              //   [0,0,0,0,0,0,0,0,0],
              //   [0,0,0,0,0,0,0,0,0],
              //   [0,0,0,0,0,0,0,0,0],
              //   [0,0,0,0,0,0,0,0,0],
              //   [0,0,0,0,0,0,0,0,0],
              //   [0,0,0,0,0,0,0,0,0],
              //   [0,0,0,0,0,0,0,0,0],
              //   [0,0,0,0,0,0,0,0,0],
              //   [0,0,0,0,0,0,0,0,0],
            // ]
            // Check top
            if (yVal != 0) {
              if (newValues[yVal-1][xVal] === opponentColor) {
                    // console.log(`p 1 val:${val} yVal:${yVal} xVal:${xVal}`)
                    let [isSorrounded, obtainedValues] = evaluateOpponentValue(newValues,
val, yVal-1, xVal, direction.BOTTOM, boardSize)
                    if (isSorrounded) {
                      valuesAreChanged = true;
                      // console.log('pass 1')
                      newValues = obtainedValues;
                    }
              }
            }
            // Check right
            if (xVal !== boardSize-1) {
              if (newValues[yVal][xVal+1] === opponentColor) {
                    // console.log(`p 2 val:${val} yVal:${yVal} xVal:${xVal}`)
                    let [isSorrounded, obtainedValues] = evaluateOpponentValue(newValues,
val, yVal, xVal+1, direction.LEFT, boardSize)
                    if (isSorrounded) {
                      valuesAreChanged = true;
                      // console.log('pass 2')
                      newValues = obtainedValues;
                    }
              }
            }
            // Check bottom
            if (yVal !== boardSize-1) {
              if (newValues[yVal+1][xVal] === opponentColor) {
                    // console.log(`p 3 val:${val} yVal:${yVal} xVal:${xVal}`)
                    let [isSorrounded, obtainedValues] = evaluateOpponentValue(newValues,
val, yVal+1, xVal, direction.TOP, boardSize)
                    if (isSorrounded) {
                      valuesAreChanged = true;
                      // console.log('pass 3')
                      newValues = obtainedValues;
                    }
              }
            }
            // Check left
            if (xVal != 0) {
              if (newValues[yVal][xVal-1] === opponentColor) {
                    // console.log(`p 4 val:${val} yVal:${yVal} xVal:${xVal}`)
```

```
                        let [isSorrounded, obtainedValues] = evaluateOpponentValue(newValues,
val, yVal, xVal-1, direction.RIGHT, boardSize)
                        if (isSorrounded) {
                          valuesAreChanged = true;
                          // console.log('pass 4')
                          newValues = obtainedValues;
                        }
                  }
                }
                // Resluting Values after current value is placed
                return [valuesAreChanged, newValues];
        }

        // If it will be refactored, probably possible to remove prevDirection and use just
checkedValues
        const evaluateOpponentValue = (newValues, val, yVal, xVal, prevDirection, boardSize,
checkedValuesMap) => {
                if (!checkedValuesMap)
                  checkedValuesMap = Array.from({length: boardSize}, () =>
Array(boardSize).fill(0));
                // let checkedValuesMap = [
                //   [0,0,0,0,0,0,0,0,0],
                //   [0,0,0,0,0,0,0,0,0],
                //   [0,0,0,0,0,0,0,0,0],
                //   [0,0,0,0,0,0,0,0,0],
                //   [0,0,0,0,0,0,0,0,0],
                //   [0,0,0,0,0,0,0,0,0],
                //   [0,0,0,0,0,0,0,0,0],
                //   [0,0,0,0,0,0,0,0,0],
                //   [0,0,0,0,0,0,0,0,0],
                // ]//checkedValues.map(arr => arr.slice());
                checkedValuesMap[yVal][xVal] = 1;

                const opponentColor = val === 1 ? 2 : 1;
                // console.log(`eov val:${val} yVal:${yVal} xVal:${xVal}
prevDirect:${prevDirection}`)
                let evaluatedValues = newValues.map(arr => arr.slice());

                // Check top
                if (yVal != 0 && prevDirection !== direction.TOP &&
checkedValuesMap[yVal-1][xVal] != 1) {
                  // console.log(`eov val:${val} yVal:${yVal} xVal:${xVal}
prevDirect:${prevDirection} 1`)
                  if (evaluatedValues[yVal-1][xVal] === opponentColor) {
                        let [isSorrounded, obtainedValues, checkedValuesObtained] =
evaluateOpponentValue(evaluatedValues, val, yVal-1, xVal, direction.BOTTOM, boardSize,
checkedValuesMap.map(arr => arr.slice()))
                        if (!isSorrounded) {
                          return [false, newValues];
                        }
                        evaluatedValues = obtainedValues;
                        checkedValuesMap = checkedValuesObtained;
                  } else if (evaluatedValues[yVal-1][xVal] === 0) {
                        return [false, newValues];
                  }
                }
                // Check right
                if (xVal !== boardSize-1 && prevDirection !== direction.RIGHT &&
checkedValuesMap[yVal][xVal+1] != 1) {
                  // console.log(`eov val:${val} yVal:${yVal} xVal:${xVal}
prevDirect:${prevDirection} 2`)
                  if (evaluatedValues[yVal][xVal+1] === opponentColor) {
                        let [isSorrounded, obtainedValues, checkedValuesObtained] =
evaluateOpponentValue(evaluatedValues, val, yVal, xVal+1, direction.LEFT, boardSize,
checkedValuesMap.map(arr => arr.slice()))
                        if (!isSorrounded) {
                          return [false, newValues];
                        }
                        evaluatedValues = obtainedValues;
                        checkedValuesMap = checkedValuesObtained;
                  } else if (evaluatedValues[yVal][xVal+1] === 0) {
                        return [false, newValues];
                  }
                }
                // Check bottom
                if (yVal !== boardSize-1 && prevDirection !== direction.BOTTOM &&
checkedValuesMap[yVal+1][xVal] != 1) {
                  // console.log(`eov val:${val} yVal:${yVal} xVal:${xVal}
prevDirect:${prevDirection} 3`)
                  if (evaluatedValues[yVal+1][xVal] === opponentColor) {
                        let [isSorrounded, obtainedValues, checkedValuesObtained] =
evaluateOpponentValue(evaluatedValues, val, yVal+1, xVal, direction.TOP, boardSize,
checkedValuesMap.map(arr => arr.slice()))
                        if (!isSorrounded) {
                          return [false, newValues];
                        }
                        evaluatedValues = obtainedValues;
                        checkedValuesMap = checkedValuesObtained;
```

```
                } else if (evaluatedValues[yVal+1][xVal] === 0) {
                    return [false, newValues];
                }
            }
            // Check left
            if (xVal != 0 && prevDirection !== direction.LEFT &&
checkedValuesMap[yVal][xVal-1] != 1) {
                // console.log(`eov val:${val} yVal:${yVal} xVal:${xVal}
prevDirect:${prevDirection} 4`)
                if (evaluatedValues[yVal][xVal-1] === opponentColor) {
                    let [isSorrounded, obtainedValues, checkedValuesObtained] =
evaluateOpponentValue(evaluatedValues, val, yVal, xVal-1, direction.RIGHT, boardSize,
checkedValuesMap.map(arr => arr.slice()))
                    if (!isSorrounded) {
                      return [false, newValues];
                    }
                    evaluatedValues = obtainedValues;
                    checkedValuesMap = checkedValuesObtained;
                } else if (evaluatedValues[yVal][xVal-1] === 0) {
                    return [false, newValues];
                }
            }
            // If opponent is surrounded
            evaluatedValues[yVal][xVal] = 0;
            // console.log(evaluatedValues)
            return [true, evaluatedValues, checkedValuesMap];
        }

        const calculateMinMaxAlphaBetaPrunedMove = (currValues, playerColor, boardSize,
depth = 1) => {
            return calculateBestMove(currValues, playerColor, boardSize, depth * 2,
{outcome: Number.MIN_VALUE}, {outcome: Number.MAX_VALUE})
        }

        /**
         * It gives best (max) new values (worst for opponent)
         */
        const calculateBestMove = (currValues, playerColor, boardSize, depthIteration, alpha,
beta) => {
            const movesOutcomes = calculateMovesOutcomes(currValues, playerColor,
boardSize, depthIteration, alpha, beta)

            if (!(movesOutcomes instanceof Array)) {
              return movesOutcomes
```

```
		}


		const bestResult = movesOutcomes.reduce((prev, current) =>
		  (prev.outcome > current.outcome) ? prev : current
		);

		return bestResult;
	}
	/**
	 * It gives worst (min) new values (best for opponent)
	 */
	const calculateWorstCountermove = (currValues, playerColor, boardSize,
depthIteration, alpha, beta) => {
		const movesOutcomes = calculateMovesOutcomes(currValues, playerColor,
boardSize, depthIteration, alpha, beta)

		if (!(movesOutcomes instanceof Array)) {
		  return movesOutcomes
		}

		const worstCounterresult = movesOutcomes.reduce((prev, current) =>
		  (prev.outcome < current.outcome) ? prev : current
		);

		return worstCounterresult;
	}

	/**
	 * It calculates possible values with their outcomes for player
	 */
	const calculateMovesOutcomes = (currValues, playerColor, boardSize, depthIteration,
alpha, beta) => {
		const boardSqrtSize = boardSize * boardSize;
		let movesOutcomes = new Array(boardSqrtSize);

		let deeperAlpha = alpha
		let deeperBeta = beta

		for (let i = 0; i < boardSqrtSize; i++) {
		  const y = ~~(i / boardSize);
		  const x = i % boardSize;

		   // Calculating the outcome of making a move on this cell
```

```
                    const calculatedOutcome = calculateMoveMaxMinOutcome(currValues,
playerColor, currValues[y][x], y, x, boardSize, depthIteration, deeperAlpha, deeperBeta);
                    movesOutcomes[i] = calculatedOutcome

                    // console.log(calculatedOutcome)

                    // for max calculations for opponent
                    if (isEvenIteration(depthIteration)) {
                            // console.log(`EVEN iter${depthIteration}`)
                            if (beta.outcome <= calculatedOutcome.outcome) {
                              // console.log(`beta`)
                              /**
                               * The same as in odd iteration, but now we need to be sure here (in this
taken strategy) we will not have any outcome higher then it was in previously
                               * considered strategies. If we will have, then no need to consider
another outcomes, and so return beta (previous outcome from previous strategy).
                               */
                              return beta
                            }
                            if (deeperAlpha.outcome > calculatedOutcome.outcome) {
                              deeperAlpha = {outcome: calculatedOutcome.outcome, y:
calculatedOutcome.y, x: calculatedOutcome.x}
                            }
                    } else {
                            // console.log(`ODD iter${depthIteration}`)
                            // alpha = movesOutcomes[i].outcome > alpha ?
movesOutcomes[i].outcome : alpha
                            if (alpha.outcome >= calculatedOutcome.outcome) {
                              // console.log(`alpha`)
                              /**
                               * Do not check othet counter-player's outcomes, as there was previous
strategy alpha which already gave better or the same outcome and
                               * counter-player already will not give us better outcome (just may give
worse as it will be better for him/her)
                               *
                               * return just alpha (previous worst outcome from best strategy), as
strategy giving now calculated outcomes will not be considered at all, thus the whole
movesOutcomes now doesn't matter
                               */
                              return alpha
                            }
                            if (deeperBeta.outcome < calculatedOutcome.outcome) {
                              deeperBeta = {outcome: calculatedOutcome.outcome, y:
calculatedOutcome.y, x: calculatedOutcome.x}
```

```
                        }
                  }
                }

                return movesOutcomes
        }

        const calculateMoveMaxMinOutcome = (currValues, playerColor, val, y, x, boardSize,
depthIteration, alpha, beta) => {
                if (val === 0 && !isFieldSurroundedByNothing(currValues, y, x, boardSize) &&
!isFieldSurroundedByJustOneColor(currValues, y, x, boardSize)) {
                  return calculateMoveMaxOutcome(currValues, playerColor, y, x, boardSize,
depthIteration, alpha, beta)
                } else {
                  if (isEvenIteration(depthIteration)) {
                        return {outcome: -1000, y: y, x: x}
                  } else {
                        return {outcome: 1000, y: y, x: x}
                  }
                }
        }

        const calculateMoveMaxOutcome = (currValues, playerColor, y, x, boardSize,
depthIteration, alpha, beta) => {
                let newValues = currValues.map(arr => arr.slice())

                newValues[y][x] = playerColor

                // Evaluate new board
                let [currentValuesChanged, evaluatedValues] = evaluateBoard(newValues,
boardSize)
                if (currentValuesChanged) {
                  newValues = evaluatedValues
                }

                if (isGameFinished(newValues, boardSize)) {
                  /**
                   * Calculate final points at which the game will stopped
                   */
                  const [whitePoints, blackPoints] = checkFinalPoints(newValues, boardSize)
                  const currentPoints = playerColor === colors.WHITE ? whitePoints :
blackPoints;
                  const opponentPoints = playerColor === colors.WHITE ? blackPoints :
whitePoints;
```

```
                      const points = evaluatePoints(currentPoints, opponentPoints);

                      return {outcome: points, y: y, x: x}
                    }

                    if (depthIteration <= 1) {

                      /**
                       * Check points in current game tree state, i.e. this time when game is not yet in
end state, but depth of calculations is reached
                       */
                      const [whitePoints, blackPoints] = checkCurrentPoints(newValues, boardSize)
                      const currentPoints = playerColor === colors.WHITE ? whitePoints :
blackPoints;
                      const opponentPoints = playerColor === colors.WHITE ? blackPoints :
whitePoints;
                      const points = evaluatePoints(currentPoints, opponentPoints);

                      return {outcome: points, y: y, x: x}
                    }

                    // Use this evaluated new board and calculate another set of move max
outcomes for opponent, then take min from that set of opponent's outcomes
                    if (isEvenIteration(depthIteration)) {
                      const newDepthIteration = depthIteration - 1;
                      const opponentColor = playerColor === colors.WHITE ? colors.BLACK :
colors.WHITE
                      const calculatedNextMove = calculateWorstCountermove(newValues,
opponentColor, boardSize, newDepthIteration, alpha, beta)
                      return {outcome: calculatedNextMove.outcome, y: y, x: x}
                    } else {
                      const newDepthIteration = depthIteration - 1;
                      const currentColor = playerColor === colors.WHITE ? colors.BLACK :
colors.WHITE
                      const calculatedNextMove = calculateBestMove(newValues, currentColor,
boardSize, newDepthIteration, alpha, beta);
                      return {outcome: calculatedNextMove.outcome, y: y, x: x}
                    }
            }

            /**
             * Evaluate points from the current player perspective - if higher its better, if lower then
its worse
             */
            const evaluatePoints = (currentPoints, opponentPoints) => {
                    return opponentPoints - currentPoints;
            }

            const isEvenIteration = (n) => {
                    return n % 2 == 0;
            }

            if(move_count%2==1)
            {
             var kk = 0;

             // General case
             for(i = 2; i < 7; i++)
             {
                        for(j = 2; j < 7; j++)
                        {
                                   if(pan[i][j] == 1)
                                   {
                                              console.log(pan);
                                              if(pan[i-1][j]==0 && pan[i+1][j]==2 && pan[i][j-1]==2 &&
pan[i][j+1]==2)
                                              {
                                                         if(pan[i-2][j]!=1 || pan[i-1][j-1]!=1 || pan[i-1][j+1]!=1)
                                                         {
                                                                    play((i-1),j);
                                                                    kk = 1;
                                                         }
                                              }

                                              else if(pan[i+1][j]==0 && pan[i-1][j]==2 && pan[i][j-1]==2 &&
pan[i][j+1]==2)
                                              {
                                                         if(pan[i+1][j-1]!=1 || pan[i+1][j+1]!=1 || pan[i+2][j]!=1)
                                                         {
                                                                    play((i+1),j);
                                                                    kk = 1;
                                                         }

                                              }
                                              else if(pan[i-1][j]==2 && pan[i+1][j]==2 && pan[i][j-1]==0 &&
pan[i][j+1]==2)
                                              {
                                                         if(pan[i-1][j-1]!=1 || pan[i+1][j-1]!=1 || pan[i][j-2]!=1)
```

```
                                        {
                                                play((i),(j-1));
                                                kk = 1;
                                        }

                                }
                                else if(pan[i-1][j]==2 && pan[i+1][j]==2 && pan[i][j-1]==2 &&
pan[i][j+1]==0)
                                {
                                        if(pan[i-1][j+1]!=1 || pan[i+1][j+1]!=1 || pan[i][j+2]!=1)
                                        {
                                                play((i),(j+1));
                                                kk = 1;
                                        }
                                }
                        }

                }
        }

        // Edge Cases
        for(i = 1; i < 8; i++)
        {
                if(pan[0][i] == 1)
                {
                        if(pan[0][i-1]==0 && pan[1][i]==2 && pan[0][i+1]==2)
                        {
                                play(0,(i-1));
                                kk = 1;
                        }
                        if(pan[0][i-1]==2 && pan[1][i]==0 && pan[0][i+1]==2)
                        {
                                play(1,(i));
                                kk = 1;
                        }
                        if(pan[0][i-1]==2 && pan[1][i]==2 && pan[0][i+1]==0)
                        {
                                play(0,(i+1));
                                kk = 1;
                        }

                }

                if(pan[i][0] == 1)
```

```
		{
			if(pan[i][1]==0 && pan[i-1][0]==2 && pan[i+1][0]==2)
			{
				play(i,1);
				kk = 1;
			}
			if(pan[i][1]==2 && pan[i-1][0]==0 && pan[i+1][0]==2)
			{
				play((i-1),0);
				kk = 1;
			}
			if(pan[i][1]==2 && pan[i-1][0]==2 && pan[i+1][0]==0)
			{
				play((i+1),0);
				kk = 1;
			}


		}
	}

	for(i = 1; i < 8; i++)
	{
		if(pan[8][i] == 1)
		{
			if(pan[8][i-1]==0 && pan[7][i]==2 && pan[8][i+1]==2)
			{
				play(8,(i-1));
				kk = 1;
			}
			if(pan[8][i-1]==2 && pan[7][i]==0 && pan[8][i+1]==2)
			{
				play(7,(i));
				kk = 1;
			}
			if(pan[8][i-1]==2 && pan[7][i]==2 && pan[8][i+1]==0)
			{
				play(8,(i+1));
				kk = 1;
			}

		}

		if(pan[i][8] == 1)
```

```
        {
            if(pan[i][7]==0 && pan[i-1][8]==2 && pan[i+1][8]==2)
            {
                play(i,7);
                kk = 1;
            }
            if(pan[i][7]==2 && pan[i-1][8]==0 && pan[i+1][8]==2)
            {
                play((i-1),8);
                kk = 1;
            }
            if(pan[i][7]==2 && pan[i-1][8]==2 && pan[i+1][8]==0)
            {
                play((i+1),8);
                kk = 1;
            }
        }
      }

// Corner Cases

if(pan[0][0]==1)
{
    if(pan[0][1]==0 && pan[1][0]==2)
    {
        play(0,1);
        kk = 1;
    }
    if(pan[0][1]==2 && pan[1][0]==0)
    {
        play(1,0);
        kk = 1;
    }
}
if(pan[8][8]==1)
{
    if(pan[7][8]==0 && pan[8][7]==2)
    {
        play(7,8);
        kk = 1;
    }
    if(pan[7][8]==2 && pan[8][7]==0)
    {
        play(8,7);
```

```
                        kk = 1;
                }
        }
        if(pan[0][8]==1)
        {
                if(pan[0][7]==0 && pan[1][8]==2)
                {
                        play(0,7);
                        kk = 1;
                }
                if(pan[0][7]==2 && pan[1][8]==0)
                {
                        play(1,8);
                        kk = 1;
                }
        }
        if(pan[8][0]==1)
        {
                if(pan[7][0]==0 && pan[8][1]==2)
                {
                        play(7,0);
                        kk = 1;
                }
                if(pan[7][0]==2 && pan[8][1]==0)
                {
                        play(8,1);
                        kk = 1;
                }
        }
        if(kk == 0)
        {
              for(i = 1; i < 8; i++)
              {
                for(j = 1; j < 7; j++)
                {
                        if(pan[i][j]==1 && pan[i][j+1]==1)
                        {
                                if(pan[i+1][j]==2 && pan[i-1][j]==2 && pan[i][j-1]==2 &&
pan[i][j+2]==2 && pan[i+1][j+1]==2 && pan[i-1][j+1]==0)
                                {
                                        play((i-1),(j+1));
                                        kk = 1;
                                }
```

```
                                    else if(pan[i+1][j]==2 && pan[i-1][j]==2 && pan[i][j-1]==2 &&
pan[i][j+2]==2 && pan[i+1][j+1]==0 && pan[i-1][j+1]==2)
                                    {
                                            play((i+1),(j+1));
                                            kk = 1;
                                    }
                                    else if(pan[i+1][j]==2 && pan[i-1][j]==2 && pan[i][j-1]==2 &&
pan[i][j+2]==0 && pan[i+1][j+1]==2 && pan[i-1][j+1]==2)
                                    {
                                            play((i),(j+2));
                                            kk = 1;
                                    }
                                    else if(pan[i+1][j]==2 && pan[i-1][j]==2 && pan[i][j-1]==0 &&
pan[i][j+2]==2 && pan[i+1][j+1]==2 && pan[i-1][j+1]==2)
                                    {
                                            play((i),(j-1));
                                            kk = 1;
                                    }
                                    else if(pan[i+1][j]==2 && pan[i-1][j]==0 && pan[i][j-1]==2 &&
pan[i][j+2]==2 && pan[i+1][j+1]==2 && pan[i-1][j+1]==2)
                                    {
                                            play((i-1),(j));
                                            kk = 1;
                                    }
                                    else if(pan[i+1][j]==0 && pan[i-1][j]==2 && pan[i][j-1]==2 &&
pan[i][j+2]==2 && pan[i+1][j+1]==2 && pan[i-1][j+1]==2)
                                    {
                                            play((i+1),(j));
                                            kk = 1;
                                    }

                            }
                  }
                }


                for(i = 1; i < 7; i++)
                 {
                  for(j = 1; j < 8; j++)
                  {
                            if(pan[i][j]==1 && pan[i+1][j]==1)
                            {
                                    if(pan[i+2][j]==2 && pan[i-1][j]==2 && pan[i][j+1]==2 &&
pan[i][j-1]==2 && pan[i+1][j+1]==2 && pan[i+1][j-1]==0)
```

```
                                        {
                                                play((i+1),(j-1));
                                                kk = 1;
                                        }
                                        else if(pan[i+2][j]==2 && pan[i-1][j]==2 && pan[i][j+1]==2 &&
pan[i][j-1]==2 && pan[i+1][j+1]==0 && pan[i+1][j-1]==2)
                                        {
                                                play((i+1),(j+1));
                                                kk = 1;
                                        }
                                        else if(pan[i+2][j]==2 && pan[i-1][j]==2 && pan[i][j+1]==2 &&
pan[i][j-1]==0 && pan[i+1][j+1]==2 && pan[i+1][j-1]==2)
                                        {
                                                play((i),(j-1));
                                                kk = 1;
                                        }
                                        else if(pan[i+2][j]==2 && pan[i-1][j]==2 && pan[i][j+1]==0 &&
pan[i][j-1]==2 && pan[i+1][j+1]==2 && pan[i+1][j-1]==2)
                                        {
                                                play((i),(j+1));
                                                kk = 1;
                                        }
                                        else if(pan[i+2][j]==2 && pan[i-1][j]==0 && pan[i][j+1]==2 &&
pan[i][j-1]==2 && pan[i+1][j+1]==2 && pan[i+1][j-1]==2)
                                        {
                                                play((i-1),(j));
                                                kk = 1;
                                        }
                                        else if(pan[i+2][j]==0 && pan[i-1][j]==2 && pan[i][j+1]==2 &&
pan[i][j-1]==2 && pan[i+1][j+1]==2 && pan[i+1][j-1]==2)
                                        {
                                                play((i+2),(j));
                                                kk = 1;
                                        }

                                }
                        }
                }
        }

                if(kk == 0)
        {
                var placement = calculateMinMaxAlphaBetaPrunedMove(pan, 2, 9, 1);
                console.log(pan);
                //console.log(pantwo);

                play(placement.y,placement.x);
            }

           }


        var resign = document.getElementById("resign");
        resign.onclick = function() {

                document.getElementById('path').style.pointerEvents = 'none';

                if(move_count%2==0){
                        alert("Black has resigned, White has won the game");
                }
                else{
                        alert("White has resigned, Black has won the game");
                }

        }
}



// Implementing KO rule
function is_jie(row, col, dead_body) {
        if (dead_body.length === 1) {
                for (var i = 0; i < jie.length; i++) {
                        //If it meets (there is coordinates, and move_count is the first hand)
                        if (        jie[i][0] === dead_body[0][0] &&
                                        jie[i][1] === dead_body[0][1] &&
                                        jie[i][2] === move_count) {
                                return true;
                        }
                }
                //Add the record table
                jie.push([row, col, move_count+1]);
                return false;
        }
        return false;
}
```

```
function can_eat(row, col, color, dead_body) {
        var ret = false;
        var anti_color = 2;
        if (color === 2)
                anti_color = 1;

        if (row+1 <= 9-1 && pan[row+1][col] === anti_color) {
                make_shadow();
                shadow[row][col] = color;
                flood_fill(row+1, col, anti_color);
                if (!anti_fill_block_have_air(anti_color)) {

                        var rret = record_dead_body(dead_body);
                        ret = ret || rret;
                }

        }
        if (row-1 >= 0 && pan[row-1][col] === anti_color) {
                make_shadow();
                shadow[row][col] = color;
                flood_fill(row-1, col, anti_color);
                if (!anti_fill_block_have_air(anti_color)) {
                        var rret = record_dead_body(dead_body);
                        ret = ret || rret;
                }

        }
        if (col+1 <= 9-1 && pan[row][col+1] === anti_color) {
                make_shadow();
                shadow[row][col] = color;
                flood_fill(row, col+1, anti_color);
                if (!anti_fill_block_have_air(anti_color)) {
                        var rret = record_dead_body(dead_body);
                        ret = ret || rret;
                }

        }
        if (col-1 >= 0 && pan[row][col-1] === anti_color) {
                make_shadow();
                shadow[row][col] = color;
                flood_fill(row, col-1, anti_color);
                if (!anti_fill_block_have_air(anti_color)) {
                        var rret = record_dead_body(dead_body);
                        ret = ret || rret;
                }

            }
            return ret;
}

function record_dead_body(db) {
            var ret = false;
            for (var row = 0; row < shadow.length; row++) {
                        for (var col = 0; col < shadow[row].length; col++) {
                                    if (shadow[row][col] === 7) {
                                                db.push([row, col]);
                                                ret = true; // it's true have dead body
                                                //alert("DEAD: "+(row).toString()+","+col.toString());
                                    }
                        }
            }
            return ret;
}


var whited = 0, blacked = 0;
function clean_dead_body(db) {

            for (var i = 0; i < db.length; i++) {
                        if(pan[db[i][0]][db[i][1]] == 2)
                        {
                                    pan[db[i][0]][db[i][1]] = 0;
                           blacked = blacked + 1;
                        }
                        if(pan[db[i][0]][db[i][1]] == 1)
                        {
                                    pan[db[i][0]][db[i][1]] = 0;
                           whited = whited + 1;
                        }
                        //alert("OUT: "+(db[i][0]).toString()+","+(db[i][1]).toString());
            }

            document.getElementById("white-score").innerHTML = whited;
            document.getElementById("black-score").innerHTML = blacked;
            console.log(whited);
            console.log(blacked);

            if(blacked - whited >= 25){
                document.getElementById('path').style.pointerEvents = 'none';
                alert("White has resigned, Black has won the game");
        }
}

/* Is it free around the filling area? */
function fill_block_have_air(row, col, color) {
        for (var i = 0; i < pan.length; i++) {
                for (var j = 0; j < pan[i].length; j++) {
                        if (i !== row || j !== col) {
                                if (shadow[i][j] === 7 && pan[i][j] !== color) {
                                        return true; // This block is free, but it can be downloaded
                                }
                        }
                }
        }
        return false;
}

function anti_fill_block_have_air(color) {
        for (var i = 0; i < pan.length; i++) {
                for (var j = 0; j < pan[i].length; j++) {
                        if (shadow[i][j] === 7 && pan[i][j] !== color) {
                                return true; // live
                        }
                }
        }

        return false; //die
}

/*Creating a copy of the current state of the game board*/
function make_shadow() {
        for (var i = 0; i < pan.length; i++) {
                for (var j = 0; j < pan[i].length; j++) {
                        shadow[i][j] = pan[i][j];
                }
        }
}

function shadow_to_pan() {
        for (var i = 0; i < pan.length; i++) {
                for (var j = 0; j < pan[i].length; j++) {
                        pan[i][j] = shadow[i][j];
                }
        }
}

/* Flood filling, only the shadow board */
function flood_fill(row, col, color) {
        if (row < 0 || row > 9-1 || col < 0 || col > 9-1)
                return;

        var anti_color = 2;
        if (color === 2)
                anti_color = 1;

        if (shadow[row][col] !== anti_color && shadow[row][col] !== 7) {
                shadow[row][col] = 7; // Indicates that it has been filled
                flood_fill(row+1, col, color);
                flood_fill(row-1, col, color);
                flood_fill(row, col+1, color);
                flood_fill(row, col-1, color);
        }
}

/* Checking if there are any liberties around the stone for different scenarios*/
function have_air(row, col) {
        if (row > 0 && row < 9-1 && col > 0 && row < 9-1) {
                if (    pan[row+1][col] !== 0 &&
                                pan[row-1][col] !== 0 &&
                                pan[row][col+1] !== 0 &&
                                pan[row][col-1] !== 0 ) {
                        //alert("have no air");
                        return false;
                } else {
                        //alert("have air");
                        return true;
                }
        } else if (row === 0 && col > 0 && col < 9-1) { // side
                if (    pan[row+1][col] !== 0 &&
                                pan[row][col+1] !== 0 &&
                                pan[row][col-1] !== 0 ) {
                        //alert("have no air");
                        return false;
                } else {
                        //alert("have air");
                        return true;
                }
        } else if (row === 9-1 && col > 0 && col < 9-1) {
                if (    pan[row-1][col] !== 0 &&
                                pan[row][col+1] !== 0 &&
                                pan[row][col-1] !== 0 ) {
                        return false;
                } else {
                        return true;
                }
        } else if (col === 0 && row > 0 && row < 9-1) {
                if (    pan[row][col+1] !== 0 &&
                                pan[row+1][col] !== 0 &&
                                pan[row-1][col] !== 0 ) {
                        return false;
                } else {
                        return true;
                }
        } else if (col === 9-1 && row > 0 && row < 9-1) {
                if (    pan[row][col-1] !== 0 &&
                                pan[row+1][col] !== 0 &&
                                pan[row-1][col] !== 0 ) {
                        return false;
                } else {
                        return true;
                }
        } else if (row === 0 && col === 0) {
                if (    pan[row][col+1] !== 0 &&
                                pan[row+1][col] !== 0) {
                        return false;
                } else {
                        return true;
                }
        } else if (row === 0 && col === 9-1) {
                if (    pan[row][col-1] !== 0 &&
                                pan[row+1][col] !== 0) {
                        return false;
                } else {
                        return true;
                }
        } else if (row === 9-1 && col === 0) {
                if (    pan[row][col+1] !== 0 &&
                                pan[row-1][col] !== 0) {
                        return false;
                } else {
```

```
                        return true;
                }
        } else if (row === 9-1 && col === 9-1) {
                if (        pan[row][col-1] !== 0 &&
                                pan[row-1][col] !== 0) {
                        return false;
                } else {
                        return true;
                }
        }
}

/* Checking if stones near the current stone are the same color or not */
function have_my_people(row, col) {
        if (row > 0 && row < 9
                -1 && col > 0 && row < 9-1) {
                if (move_count % 2 === 0) { //Before placing a stone,it is white
                        if (        pan[row+1][col] === 1 ||
                                        pan[row-1][col] === 1 ||
                                        pan[row][col+1] === 1 ||
                                        pan[row][col-1] === 1 ) {
                                //alert("have my people");
                                return true;
                        }
                } else {
                        if (        pan[row+1][col] === 2 ||
                                        pan[row-1][col] === 2 ||
                                        pan[row][col+1] === 2 ||
                                        pan[row][col-1] === 2 ) {
                                //alert("have my people");
                                return true;
                        }
                }
        } else if (row === 0 && col > 0 && col < 9-1) { // side
                if (move_count % 2 === 0) {
                        if (        pan[row+1][col] === 1 ||
                                        pan[row][col+1] === 1 ||
                                        pan[row][col-1] === 1 ) {
                                //alert("have my people");
                                return true;
                        }
                } else {
                        if (        pan[row+1][col] === 2 ||
                                        pan[row][col+1] === 2 ||
                        pan[row][col-1] === 2 ) {
                    //alert("have my people");
                    return true;
                }
            }
        } else if (row === 9-1 && col > 0 && col < 9-1) { //side
            if (move_count % 2 === 0) {
                if (        pan[row-1][col] === 1 ||
                            pan[row][col+1] === 1 ||
                            pan[row][col-1] === 1 ) {
                    //alert("have my people");
                    return true;
                }
            } else {
                if (        pan[row-1][col] === 2 ||
                            pan[row][col+1] === 2 ||
                            pan[row][col-1] === 2 ) {
                    //alert("have my people");
                    return true;
                }
            }
        } else if (col === 9-1 && row > 0 && row < 9-1) {
            if (move_count % 2 === 0) {
                if (        pan[row+1][col] === 1 ||
                            pan[row-1][col] === 1 ||
                            pan[row][col-1] === 1 ) {
                    //alert("have my people");
                    return true;
                }
            } else {
                if (        pan[row+1][col] === 2 ||
                            pan[row-1][col] === 2 ||
                            pan[row][col-1] === 2 ) {
                    //alert("have my people");
                    return true;
                }
            }
        } else if (col === 0 && row > 0 && row < 9-1) {
            if (move_count % 2 === 0) {
                if (        pan[row+1][col] === 1 ||
                            pan[row-1][col] === 1 ||
                            pan[row][col+1] === 1 ) {
                    //alert("have my people");
                    return true;
                }
            } else {
                if (        pan[row+1][col] === 2 ||
                            pan[row-1][col] === 2 ||
                            pan[row][col+1] === 2 ) {
                        //alert("have my people");
                        return true;
                }
            }
        } else if (row === 0 && col === 0) { // Corner
            if (move_count % 2 === 0) {
                if (        pan[row+1][col] === 1 ||
                            pan[row][col+1] === 1 ) {
                        //alert("have my people");
                        return true;
                }
            } else {
                if (        pan[row+1][col] === 2 ||
                            pan[row][col+1] === 2 ) {
                        //alert("have my people");
                        return true;
                }
            }
        } else if (row === 0 && col === 9-1) {
            if (move_count % 2 === 0) {
                if (        pan[row+1][col] === 1 ||
                            pan[row][col-1] === 1 ) {
                        //alert("have my people");
                        return true;
                }
            } else {
                if (        pan[row+1][col] === 2 ||
                            pan[row][col-1] === 2 ) {
                        //alert("have my people");
                        return true;
                }
            }
        } else if (row === 9-1 && col === 0) {
            if (move_count % 2 === 0) {
                if (        pan[row-1][col] === 1 ||
                            pan[row][col+1] === 1 ) {
                        //alert("have my people");
                        return true;
                }
                } else {
                        if (        pan[row-1][col] === 2 ||
                                            pan[row][col+1] === 2 ) {
                                //alert("have my people");
                                return true;
                        }
                }
        } else if (row === 9-1 && col === 9-1) {
                if (move_count % 2 === 0) {
                        if (        pan[row-1][col] === 1 ||
                                            pan[row][col-1] === 1 ) {
                                //alert("have my people");
                                return true;
                        }
                } else {
                        if (        pan[row-1][col] === 2 ||
                                            pan[row][col-1] === 2 ) {
                                //alert("have my people");
                                return true;
                        }
                }
        }

        return false;
}

function stone_down(row, col) {
        if (move_count % 2 === 0) {
                pan[row][col] = 1;
        } else {
                pan[row][col] = 2;
        }
        move_count ++;
        move_record.push([row, col, move_count]);
}
```

5) script.js:

```
function startGame(opponent) {

  if (opponent === 'human') {

    window.location.href = 'human_game.html';
  } else if (opponent === 'bot') {

    window.location.href = 'human_vs_bot.html';

  }
}


// Show rules modal dialog
function showRules() {
  var rulesModal = document.getElementById("rules-modal");
  rulesModal.style.display = "block";
}

// Hide rules modal dialog
function hideRules() {
  var rulesModal = document.getElementById("rules-modal");
  rulesModal.style.display = "none";
}

// Close rules modal dialog when user clicks outside of it
window.onclick = function(event) {
  var rulesModal = document.getElementById("rules-modal");
  if (event.target == rulesModal) {
    rulesModal.style.display = "none";
  }
}

/* Modal for end game */

// Get the modal
var modal = document.getElementById("myModal");

// Get the <span> element that closes the modal
var span = document.getElementsByClassName("close")[0];

// Show the modal when the game is over
modal.style.display = "block";

// When the user clicks on <span> (x), close the modal
span.onclick = function() {
  modal.style.display = "none";
}
```

CSS:

1) **game_menu.css:**

```
/* general styles */
body {
	/* background-color: #F5F5DC ; */
	font-family: 'Helvetica Neue', sans-serif;
	font-size: 16px;
	font-weight: 300;
	background-image: url('../goboard.png') ;
	background-repeat: no-repeat;
	background-size: cover;
}

h1 {
	text-align: center;
	margin-top: 50px;
	font-size: 40px;
	color: #f3c383;
}

.container {
	display: flex;
	flex-direction: column;
	align-items: center;
	margin-top: 100px;
	background-color: rgba(0,0,0,0);
}

/* button styles */
.btn {
	padding: 15px 40px;
	color: #fff;
	font-size: 20px;
	border: none;
	border-radius: 5px;
	cursor: pointer;
	margin-bottom: 20px;
	transition: all 0.3s ease-in-out;

	text-transform: uppercase;
	letter-spacing: 1px;
	font-weight: bold;
```

```css
	box-shadow: 0px 5px 10px rgba(0, 0, 0, 0.2);
	background-image: linear-gradient(to bottom, #D2B48C, #8B4513);
}

.btn:hover {
	background-color: #4d4dff;
}

/* beautification styles */
.container {
	box-shadow: 0px 5px 20px rgba(0, 0, 0, 0.1);
	border-radius: 10px;
	padding: 40px;
}

/* .container:hover {
	transform: scale(1.05);
	box-shadow: 0px 10px 25px rgba(0, 0, 0, 0.2);
} */


.btn:first-child {
	/* background-color: #3d3dff; */
	margin-right: 8px;
	width: 373.5px;
}

.btn:nth-child(2) {
	/* background-color: #3d3dff; */
	margin-right: 8px;
	width: 373.5px;
}

.btn:nth-child(3) {
	/* background-color: #3d3dff; */
	margin-right: 8px;
}

.btn:hover {
	transform: translateY(-2px);
	box-shadow: 0px 8px 15px rgba(0, 0, 0, 0.3);
}
```

```
/* Styling the Modal dialog box for rules and instructions*/

/* Modal styles */
.modal {
        display: none; /* Hidden by default */
        position: fixed; /* Stay in place */
        z-index: 1; /* Sit on top */
        padding-top: 100px; /* Location of the box */
        left: 0;
        top: 0;
        width: 100%; /* Full width */
        height: 100%; /* Full height */
        overflow: auto; /* Enable scroll if needed */
        background-color: rgba(0, 0, 0, 0.4); /* Black w/ opacity */
}

/* Modal Content */
.modal {
        display: none; /* Hidden by default */
        position: fixed; /* Stay in place */
        z-index: 1; /* Sit on top */
        left: 0;
        top: 0;
        width: 100%; /* Full width */
        height: 100%; /* Full height */
        overflow: auto; /* Enable scroll if needed */
        background-color: rgba(0, 0, 0, 0.4); /* Black with opacity */
  }

  /* Modal Content */
  .modal-content {
        background-color: #ffffff;
        margin: auto;
        padding: 20px;
        border: 1px solid #d4d4d4;
        border-radius: 10px;
        box-shadow: 0px 0px 20px rgba(0, 0, 0, 0.3);
        max-width: 600px;
        width: 90%;
        max-height: 65vh;
        overflow-y: auto;
  }

  /* Close Button */
```

```
.close {
        color: #aaaaaa;
        float: right;
        font-size: 28px;
        font-weight: bold;
}

.close:hover,
.close:focus {
        color: #000;
        text-decoration: none;
        cursor: pointer;
}

/* Header Styles */
h2 {
        font-size: 24px;
        font-weight: bold;
        margin-top: 0;
}

/* List Styles */
ul {
        margin-top: 10px;
        margin-bottom: 0;
        padding-left: 20px;
}

ul li {
        margin-bottom: 10px;
        font-size: 18px;
        line-height: 1.5;
        color: #333333;
}

/* Button Styles */
.button {
        background-color: #0077cc;
        color: #ffffff;
        border: none;
        border-radius: 5px;
        padding: 10px 20px;
        font-size: 18px;
        font-weight: bold;
        cursor: pointer;
        transition: background-color 0.3s ease;
  }

  .button:hover {
        background-color: #005ea8;
  }
```

2) style.css:

```
body {
        margin: 0 auto;
        width:600px;
        background-color: #d0fefe;
        overflow: hidden;
}

h2 {
        /*
        text-align: center;
        margin: 0 auto;
        */
}

button {
        background-color: #FFFFFF;
        border: 2px solid #000000;
        color: #000000;
        padding: 10px;
        text-align: center;
        text-decoration: none;
        display: inline-block;
        font-size: 16px;
        margin: 25px;
        cursor: pointer;
        border-radius: 4px;
}

button:hover {
        background-color: #000000;
        color: #FFFFFF;
}
```

```css
.buttons {
	display: flex;
	flex-direction: row;
	justify-content: center;
	align-items: center;
	margin-top: 60px;
	margin-right: -600px;
}




#weiqi {
	border: 0px solid blue;
	position: absolute;
	top:10px;
	left: 0px;
	/* height: 600px; */
}
#path {
	position: absolute;
	top:10px;
	/*
	left:0;
	*/
	z-index: 200;
	left: 0px;
}

.parent{

	display: grid;
	grid-template-columns: 1fr 1fr;
	width: 700px;
	column-gap: 0px;
	margin-left: 250px;
	margin-top: 20%;
}

.child{

	height: 39px;
	margin-top: 10px;
	background: white;
```

```
        display: flex;
        align-items: center;
        border-radius: 10px;
}

.child-1{

        height: 39px;
        margin-top: 10px;
        background: black;
        color: white;
        display: flex;
        align-items: center;
        border-radius: 10px;
        /* justify-content: center; */
}

/* The Modal (background) */
.modal {
        display: none; /* Hidden by default */
        position: fixed; /* Stay in place */
        z-index: 1; /* Sit on top */
        padding-top: 100px; /* Location of the box */
        left: 0;
        top: 0;
        width: 100%; /* Full width */
        height: 100%; /* Full height */
        overflow: auto; /* Enable scroll if needed */
        background-color: rgba(0, 0, 0, 0.4); /* Black w/ opacity */
}

/* Modal Content */
.modal-content {
        background-color: #fefefe;
        margin: auto;
        padding: 20px;
        border: 1px solid #888;
        border-radius: 10px;
        box-shadow: 0px 0px 20px rgba(0, 0, 0, 0.3);
        max-width: 600px;
        width: 90%;
        max-height: 80vh;
        overflow-y: auto;
}
```

```
/* Close Button */
.close {
            color: #aaaaaa;
            float: right;
            font-size: 28px;
            font-weight: bold;
}

.close:hover,
.close:focus {
            color: #000;
            text-decoration: none;
            cursor: pointer;
}

/* Coordinate Labels */
.coord-labels {
    position: absolute;
    top: 0;
    left: 0;
    font-size: 12px;
    font-weight: bold;
}
.x-coords {
    width: 600px;
    height: 20px;
    text-align: center;
}
.y-coords {
    width: 20px;
    height: 600px;
    text-align: right;
    padding-right: 5px;
}
```