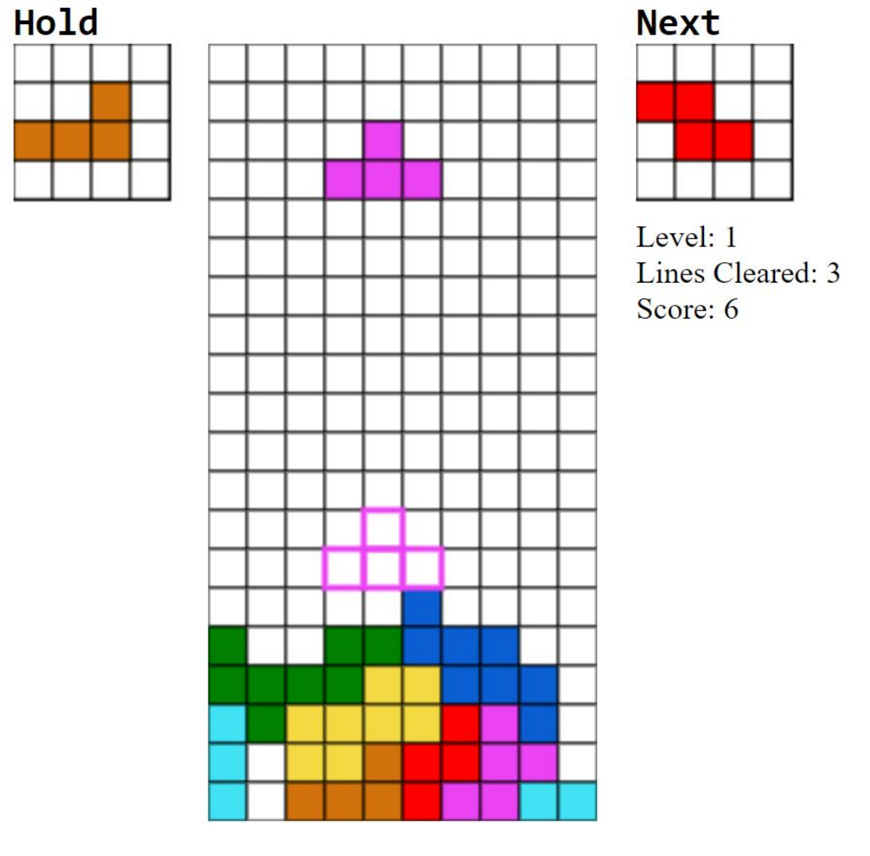
Tetris AI

Vivian Lin

The Game

Tetris is a tile- matching puzzle game. The tiles that you match are called Tetrominoes. There are a total of 7 different shapes. The shapes are randomly chosen and fall down a grid. The goal of the game is the manipulate the tetromino by rotating or translating it on the grid into a position where they form a horizontal line with no gaps. Once this happens the line clears and all the blocks above it shift down.



The Decision Function

The goal of the AI is to clear as the many lines as possible or survive the longest. The purpose of this function is help the AI decide what is the best move.

1. The AI simulates and evaluates all possible boards that would result from all possible moves of the given piece.
2. Given current piece to place, the AI will make a decision based on which action (which rotation and translation) produces the highest score based on evaluation function. The evaluation function is a combination of the feature functions and their respective weights.

```
// AI Functions
//chooses the piece with the best score.
function decision_function(p) {
    let illegalMoves = 0;
    let maxScore = Number.NEGATIVE_INFINITY;
    let move = {
        rotation: 0,
        translation: 0,
    }
//tests all possible moves to find the best move.
    for (var translation = -1; translation < 9; translation ++) {
        for (var rotation = 0; rotation < 4; rotation ++) {
            //simulates the action and calculates the score the new board.
            let score = action(p, rotation,translation);
            if(score === Number.NEGATIVE_INFINITY) {
                illegalMoves ++;
            }
            else if (score > maxScore) {
                maxScore = score;
                move.rotation = rotation;
                move.translation = translation;
            }
        }
    }
    //Mo more possible moves.
    if (illegalMoves == 40) {
        endGame();
    }
    //check if score of current piece less than score of next piece
    makeMove(move);
}

function action(p, rotation, translation) {
    //create a clone of the current piece.
    var pieceClone = new Tetromino(p.tetromino, p.color);
    if (pieceClone.color == YELLOW) {
        rotation = 0;
    }
    let xMove = translation - pieceClone.x;
    //check if piece is playable. Then simulate the piece being placed on the board.
    if (!pieceClone.collision(xMove, 0, pieceClone.tetromino[rotation])) {
        pieceClone.currTetromino = pieceClone.tetromino[rotation];
        pieceClone.x = translation;
        let board_copy = copyMatrix(gameBoard);
        pieceClone.y = pieceClone.calcDropPosition();
        pieceClone.lock(board_copy, true);
        return pieceClone.score;
    }
    return Number.NEGATIVE_INFINITY;
}

//set the values of the current piece to the best move.
function makeMove(move) {
    piece.tetrominoIdx = move.rotation;
    piece.currTetromino = piece.tetromino[move.rotation];
    piece.x = move.translation;
}
```

Feature Functions

Feature functions determine the score of given board.

```
calcFeatures(board) {
  //the height where right above it the whole row is empty.
  let height = 0;
  let rowsArr = board.reduce((a, row) => a.concat(row.filter(col => col != "WHITE").length), []);
  for (var i = rowsArr.length-1; i >= 0;  i --) {
    if (rowsArr[i] == 0) {
      height = ROW - i - 1;
      break;
    }
  }
  this.features.height = height;

 //the sum of the absolute differeces between the heights of adjacent columns.
 let colHeights = [0,0,0,0,0,0,0,0,0,0];
 for (let c = 0; c < board[0].length; c ++) {
  for (let r = 0; r < board.length; r++) {
    if (board[r][c] != "WHITE") {
      colHeights[c] = ROW - r;
      break;
      }
    }
  }
  let bumpiness = 0;
  for (let j = 0; j < colHeights.length - 1; j++) {
    bumpiness += Math.abs(colHeights[j] - colHeights[j+1]);
  }
  this.features.bumpiness = bumpiness;

  //the number of empty sqaures with a filled sqaure above it.
  let holes = 0;
  for (let r = ROW-1; r > 0; r --) {
    board[r].forEach((col, c) => {
      if (col == "WHITE" && board[r - 1][c] != "WHITE") {
        holes ++;
      }
    })
  }
  this.features.holes = holes * 5;
```

```
  //Number of lines cleared
  let linesCleared = 0;
  for (var r = 0; r < ROW; r++) {
    let filled = 0;
      for (var c = 0; c < COL; c ++) {
        if (board[r][c] != "WHITE") {
          filled ++;
        }
      }
      if (filled == 10) {
        linesCleared ++;
      }
    }
     switch(linesCleared) {
  case 1:
    linesCleared = 1;
    break;
  case 2:
    linesCleared = 3;
     break;
  case 3:
    linesCleared = 6;
     break;
  case 4:
    linesCleared = 12;
     break;
 }
  this.features.cleared = linesCleared;

  //calculate number of empty sqaures below placed piece.
  var columns = [];
  var rows = [];
  this.currTetromino.forEach((row,i) => row.forEach((col, j) => {
      if (col) {
        if (columns.indexOf(j + this.x) == -1) {
          columns.push(j + this.x);
        rows.push(i + this.y);
        }
      }
    }))
let vacant = 0;
    for (var j = 0; j < columns.length; j++) {
      let c = columns[j];
      for (var r = rows[j]; r < ROW; r++) {
        if (board[r][c] == "WHITE") {
          vacant ++;
        }
      }
    }
  this.features.vacant = vacant;
}
```

Feature Functions (cont.)

1. **Height**- You would want to minimize the height because this means you can place more pieces.
2. **Holes**- You would also want to minimize the number of holes because a hale makes the line harder to clear.
3. **Cleared**- You want to maximize the number of cleared lines.
4. **Bumpiness**- You want to minimize bumpiness because a flatter board makes it easier to clear lines.
5. **Vacant**- You want to minimize the number of vacant spots under where your piece is placed because it makes it harder to clear the tiles below where your piece is placed.

The evaluation function is based on the feature functions and is used to calculate the score the game state. Higher score means this is a good place to drop the piece.

```
  //Combination of the features and their respective weights.
  evaluation_function(features) {
    this.score = features.height*weights.a + features.holes*weights.b + features.cleared*weights.c + features.bumpiness*weights.d + features.vacant*weights.e;
  }
}
```

The Genetic Algorithm

In this project I used the genetic algorithm to teach the computer how to play Tetris.

1. Create a random initial population.
2. Selection- select parents to create children.
3. Reproduction- produce children then replace individuals from the current generations with the newly generated population of children.
4. Return to step 2 until generation number is equal to 25.

1. Initialize a random population.

Create a population of POPSIZE games each with randomly generated DNA.

```
function setup() {
  initialize_training_varaibles();
  population = new Population();
  population.games[num_of_games-1].startGame();
  genetic_algorithm();
}
```

```
//POPULATION OBJECT
function Population() {
  this.games = [];

  for (var i = 0; i < POPSIZE; i++) {
    this.games[i] = new Game();
  }
```

The genes are randomly generated weights for each feature function.

```
function DNA(genes) {
  // Recieves genes and create a dna object
  if (genes) {
    this.genes = genes;
  }
  // If no genes just create random dna
  else {
    this.genes = {
      a : Math.random() - 0.5,
      b : Math.random() - 0.5,
      c : Math.random() - 0.5,
      d : Math.random() - 0.5,
      e : Math.random() - 0.5
    }
  }
```

2. Selection

1. Calculate the fitness for each individual. The fitness of each individual is the game score. The game score is based on the number of lines the AI cleared before the game ended.

```
this.update = function() {
    if (gameOver == true){
      gameOver = true;
        this.lines = lines;
        this.fitness = game_score;
    }
}
```

2. After the fitness has been calculated for each individual. I used the probabilistic method to select individuals from the population. Elements with higher fitness values with have a higher chance of participating in the reproductive process.

```
this.selection = function() {
  var newGames = [];
  for (var i = 0; i < (this.games.length/2); i++) {
    var parentA = this.pickOne(this.games);
    var parentB = this.pickOne(this.games);
    // Creates child by using crossover function
    var child = parentA.dna.crossover(parentB.dna, parentA, parentB);
    child.mutation();

    newGames[i] = new Game(child);
  }
  //replace the bottom half of the population with newGames.
  this.games.splice(this.games.length/2);
  this.games = this.games.concat(newGames);

 }
}
```

3. Reproduction(Crossover)

```
this.crossover = function(partner, pA, pB) {
  //child will have most of the genes from the parent with the better fitness.
  //if A has a larger fitness it's genes will be close to A.
  let alpha = this.genes;
  let beta = partner.genes;
  if ( pA.fitness < pB.fitness) {
    alpha = partner.genes;
    beta = this.genes;
  }
   var newgenes = {
   a: (alpha.a * alpha_multiplier + beta.a * beta_multiplier),
   b: (alpha.b * alpha_multiplier + beta.b * beta_multiplier),
   c: (alpha.c * alpha_multiplier + beta.c * beta_multiplier),
   d: (alpha.d * alpha_multiplier + beta.d * beta_multiplier),
   e: (alpha.e * alpha_multiplier + beta.e * beta_multiplier)
 }
  return new DNA(newgenes);
}
```

Crossover creates a child out of the genes of the two parents. The parent with the higher fitness is the alpha and the other one is the beta. Since the alpha multiplier is greater multiplier than the beta multiplier the genes of the child will closer to the fitter parent.

3. Reproduction(Mutation)

```
// Adds random mutation to the genes to add variance.
this.mutation = function() {
  if (Math.random() < mutation_rate) {
    this.genes.a = this.genes.a + Math.random() * mutation_multiplier;
  }
  if (Math.random() < mutation_rate) {
    this.genes.b = this.genes.b + Math.random() * mutation_multiplier;
  }
  if (Math.random() < mutation_rate) {
    this.genes.c = this.genes.c + Math.random() * mutation_multiplier;
  }
  if (Math.random() < mutation_rate) {
    this.genes.d = this.genes.d + Math.random() * mutation_multiplier;
  }
  if (Math.random() < mutation_rate) {
    this.genes.e = this.genes.e + Math.random() * mutation_multiplier;
  }
 }
}
```

After a child is created a mutation can occur. This creates more variety in the population. During mutation a random number is added to the genes of the child. This process is repeated 50 times.

4. Repeat

```
function genetic_algorithm() {
  population.games[game_num-1].update();

  if (gameOver == true) {
    moves = 0;
    game_num ++;
    population.games[game_num-1].startGame();
  }

  if (game_num == POPSIZE) {
    population.evaluate();
    population.selection();
    game_num = 1;
    population.games[game_num-1].startGame();
    generation ++;
  }
  if (generation <= 25) {
    requestAnimationFrame(genetic_algorithm);
  }
}
```

Repeat steps 2-4 until generation equals 25.

The Results

```
POPSIZE = 50;
generation = 1;
maxFitness = 0;
maxLines = 0;
game_num = 1;
mutation_rate = 0.05;
mutation_multiplier = 0.4;
alpha_multiplier = 0.7;
beta_multiplier = 0.3;
best_weights = {
  a:0,
  b:0,
  c:0,
  d:0,
  e:0,
}
weights = {
  a:0,
  b:0,
  c:0,
  d:0,
  e:0,
}
```

After running the genetic algorithm for about 8 hours using these variables it produced these weights.

```
weights = {
  a:0.012986105043601821,
  b:-0.33099889329580323,
  c:0.5446471620000896,
  d:-0.25120763453283845,
  e:-0.13253702980064244,
}
```