# 1 Overview

Video games have long been a source of education, social interaction, and entertainment. Many games require problem solving to achieve objectives, encouraging users to think critically and express creativity. As technology advances and video games evolve, their impact on society will only continue to grow.

## 1.1 Project Goals

In this project, we were asked to create a fully functional Tetris game. Our implementation is composed of two parts: a **TetrisPiece** class to control the functionality of different piece types and a **TetrisBoard** class to manage the state of the board. In addition, we implemented a Tetris "brain" that analyzes the game state and computes an "optimal" move for each turn, allowing our game to be played without user input. On the implementation side, the goal of this project was to create a Tetris game with standard pieces, rotations, and moves as well as a brain that could generate moves with human-like decision making. On the time complexity side, we wanted to ensure that each accessor method in **TetrisPiece** and **TetrisBoard** operated in constant time.

## 1.2 Personal Goals

Our main personal goal was to learn how to efficiently break down a seemingly large programming project into smaller, independent sections. This encouraged us to think about the trade-offs of using different data structures, which included time/space complexity and code readability. Furthermore, we aimed to create a testing harness using Junit that verified the correct handling of both regular actions and edge cases.

Since this was a pair programming project, we augmented collaborative goals to the technical ones explained above. One of the main ones was to improve communication skills, especially when explaining previously implemented concepts and potential solutions to bugs. Furthermore, we made extra effort to maintain code readability through detailed/concise comments in each method.

# 2 Solution Design

## 2.1 Assumptions

We assumed the given JTetris GUI implemented the methods in the Brain and Piece interfaces as written in the assignment PDF.

We also made some assumptions when testing. This included the following:

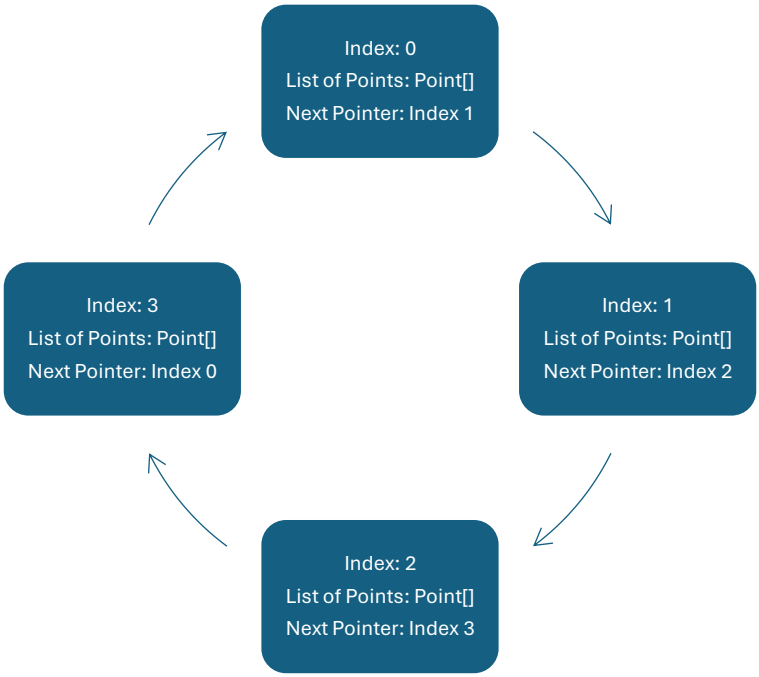
- **TetrisPiece:** If a tested function worked on one piece type, we generalized the behavior because all pieces were implemented the same
- For the brain, we assumed that its moves were drastically faster than the speed at which the pieces fall. Hence, there is no timer for the down command when the brain is running.

## 2.2 Solution Design

We started by breaking the project into smaller parts, tackling each function of Tetris separately. We decided to use helper functions to create modular code, allowing for easier debugging and testing.

### 2.2.1 **TetrisPiece**

**TetrisPiece** initializes the piece with a circularly linked list, storing the rotation index, current points, and a next pointer in each node. Below is a visual representation of the linked list created.




#### 2.2.1.1 **getType, getRotationIndex, getBody**

In the following table, we list out the class variable names, what they store, and the functions that use them:

| Name | Information stored | Functions that use the variable |
|---|---|---|

| | | |
|---|---|---|
| *pieceType* | This variable stores the piece's type (T, Stick, etc.) | **getType(), getWidth(), getHeight()** |
| *orientation* | This variable stores the node in the circularly linked list that the current piece is on. | **getRotationIndex(), clockwisePiece(), counterclockwisePiece(), getBody(), getSkirt(), equals()** |

Below, we have bullet points of each function and their implementation:

- **getType**: returns the *pieceType* class variable
- **getRotationIndex**: returns the current index stored in the *orientation* node
- **getBody**: returns the Points list stored in the *orientation* node
- **getWidth/getWidth**: returns the width/height of the bounding box, which is determined with a series of if statements that classify *pieceType*
- **clockwisePiece**: returns a new piece with its *orientation* variable set to the node pointed to by the next pointer of the current piece's *orientation* variable
- **counterclockwisePiece**: this function does the same thing as **clockwisePiece**, except it moves 3 nodes forward in the linked list rather than 1.
- **getSkirt**: the program creates a list of integers, called *skirt*, with the same length as the bounding box's width. Each value is initialized with the maximum integer value. Then, we iterate through the points of the current piece. At each Point *i* with values of $x_i$ and $y_i$, we set the value at index $x_i$ in *skirt* as the minimum of $y_i$ and the value that is currently there.
- **equals**: first, as an error check, we make sure that the object inputted is a **TetrisPiece** type. If it isn't, then the function returns false. If it is, the function returns the Boolean of if both the rotation index and *pieceType* are equal.

### 2.2.1 **TetrisBoard**

TetrisBoard controls the creation of the board and its state after each move. The code creates a board in the constructor by creating a 2d array of *PieceTypes* with each square initialized as null to represent an empty square.

In the following table, let's define the class variables of **TetrisBoard.**

| Name | Function |
|---|---|

| | |
|---|---|
| *refX/refY* | This represents the board coordinates of the bottom left corner of the bounding box. This is used to calculate the piece's position on the board rather than relative to its bounding box. |
| *piece* | Saves the current piece that is being dropped. |
| *board* | This holds the current board state, with the already placed pieces on the board. |
| *lastResult* | This saves the result of the last action. |
| *lastAction* | This holds the previous action that was called. |
| *rowsCleared* | This saves the number of rows that were cleared on the previous turn. |

Below are each of the methods and how they were created.

#### 2.2.1.1 **Helper Functions**

We created several helper functions to improve code readability and reusability.

- **getX, getY**: These functions take coordinates relative to the bounding box and return their positions on the *board* by adding *refX*/*refY*.
- **getAbsolutePoints**: This function converts each point in the given piece from bounding box coordinates to board coordinates using **getX** and **getY**.
- **moveValid**: This function returns true if the move, which includes down, left and right, is valid. Otherwise, it returns false. This is used to test the validity of a move function.
- **rotateValid**: This function uses the piece type and current rotation index to initialize the *wallKick* array. It then iterates through each kicker point and shifts the piece accordingly. On an iteration, if every point in the piece is within bounds and does not overlap with another piece, then our code shifts it by that amount and returns true. If our code iterates through all the kicker points and there is no change in piece position to make the rotation valid, then it returns false.
- **getAbsoluteSkirt**: This function is similar to the **skirt** function in **TetrisPiece**, but it returns the coordinates relative to the board rather than the ones relative to the lower left corner of the bounding box.
- **setPiece**: This function calls **getAbsolutePoints** to obtain the board coordinates of the given piece. Then, the program sets each coordinate of the piece on the board equal to its type.

- **drop**: First, the function calls **getAbsoluteSkirt** to get the skirt relative to the board. Then, this function iterates down the board in each column until it hits a piece or goes out of bounds. The minimum row drop is returned.
- **ClearRows**: This function first iterates through the board, appending the row index of each full row to a list called *clearRows*. If there are no full rows after our check, the code terminates and no action is needed. Otherwise, it starts moving up the board with a starting index at the first element of *clearRows*. At each iteration, the program checks if the row is in the *clearRows* array. If it is, then the row is cleared and a *counter* is incremented. Otherwise, the current row's values are moved down by *counter* amount.

2.2.1.2 **move**

These are the following functions that **move** can detect and execute:

- *Left, right, down*: First, it checks its validity using the *isValid* helper function. If it is, *refX* and *refY* are changed accordingly (+1 to *refX* if its right, -1 to *refX* if its left, and -1 to *refY* if it's down). If the down move is invalid, then that means it has touched another block or has reached the bottom of the board and will return the *Place* enum. Hence, the function will call **setPiece** to place the piece on the board. From here, **clearRows** is called to get rid of any rows that are full after placing the piece and the *rowsCleared* variable is updated too. Finally, it will return the *Success* enum.
- *Clockwise, counterclockwise*: the **rotateValid** helper function checks whether rotating the piece is possible, including adjusting using wall kicks. If it is, then the program rotates the piece using the functions from **TetrisPiece** and returns the *Success* enum. Otherwise, it will return the *Out_Bounds* enum.
- *Drop*: our code calls the drop helper function to calculate how far down to drop the piece. Then, our code decreases the *refY* value by this much. This always returns the *Place* enum.

The *lastAction* and *lastResult* variables are also updated after every command.

2.2.1.3 **testMove**

This tests the move by creating a deep copy of the current board and calling the move function. This deep copy is implemented by creating a new board in the same state as the original board. Then, we copy over *piece*, *refX*, *refY*, and *rowsCleared*.

2.2.1.4 **getCurrentPiece, getCurrentPiecePosition**

**getCurrentPiece** returns *piece*, which we initialize when we start the program.

2.2.1.5 **getCurrentPiecePosition**

Returns the *refX* and *refY* variables, which represent the board coordinates of the bottom left corner of the bounding box of *piece*.

2.2.1.6 **nextPiece**

This method updates the *refX* and *refY* variables to the spawn position of the new piece. Then, it verifies that all the points in the body of the new piece map to in bound, empty squares on the board. If they do, *piece* is set to the newly created piece.

2.2.1.7 **equals**

This method first checks if the object in the parameter is a board type, returning false if it is not. Otherwise, the program iterates through every square on both boards, comparing the piece types in each coordinate. If all the squares are equal, then the method returns True.

2.2.1.8 **getLastResult, getLastAction**

These values are stored when calling **move**. The program simply returns the value that we saved previously.

2.2.1.9 **getRowsCleared**

At each turn, *rowsCleared* is set to 0. The only time when this variable is changed is when the placed piece creates full rows. **getRowsCleared** returns the *rowsCleared* variable.

2.2.1.10 **getWidth, getHeight, getGrid**

The **getWidth** and **getHeight** functions return the number of columns and rows, respectively, of *board*. The **getGrid** function returns the piece type at the given x and y coordinate on *board*.

2.2.1.11 **getMaxHeight**

We get the max height of our board by comparing the heights when iterating through every column and its height using **getColumnHeight**.

2.2.1.12 **dropHeight**

This method starts by using the **getAbsolutePoints** and **getAbsoluteSkirt** helper functions and saving their outputs. Then, we get the number of rows that the piece will fall if it were to drop at that moment using the **drop** helper function. Then, we return the

difference between the skirt value of the inputted column and the value from the **drop** helper function.

#### 2.2.1.13 **getColumnHeight, getRowWidth**

For **getColumnHeight**, our code iterates from the bottom of the column to the top, returning the position of the first empty square. If the whole column is full, the length of the column is returned. For **getRowWidth**, our code iterates through the given row and increments a counter for each square that isn't null. At the end of the row, it returns the counter variable.

# 3 Error Catching

We have error catching in 3 different spots:

- **TetrisBoard** constructor: we make sure that the input width and height are non-negative numbers. As long as this requirement is fulfilled, then the error catching in **nextPiece** will make sure that all piece inputs will be valid.
- **nextPiece** function: in the **nextPiece** function, we make sure that the spawn point will put the piece in a valid position. This means that all the points are within bounds and are not occupying a space that another piece is in.
- **equals** function in **TetrisBoard**: we make sure that the input object is a board type. If it isn't, then we just return and exit.

# 4 Testing

## 4.1 **TetrisPiece**

We tested this class by creating JUnit tests for each method. Our modular code made it much easier to test the functionality of the whole class.

### 4.1.1 **testGetType**

This method creates one piece of each type and asserts to make sure **getType** returns the proper type.

### 4.1.2 **testRotationIndex**

Our method creates a piece and rotates it, checking to make sure the rotation index gets updated with each rotation. We only test one piece because every piece's linked list is implemented using the same code.

### 4.1.3 **testGetWidth, testGetHeight**

These methods create each new piece and assert that the returned dimensions are being returned as expected from the assignment PDF.

### 4.1.5 **testGetBody, testRotations**

These methods compare the returned points with the points that we expect. For **getBody**, our code checks to make sure the body matches that of our expectations for two different pieces. For our rotations, the code rotates 3 different pieces in both directions and verifies that the body is the same as that of our expectations.

### 4.1.6 **testSkirt**

For this method, we manually determined the skirt of different pieces and rotations and compared them to the output of **testSkirt.**

### 4.1.7 **testEquals**

This method creates a variety of pieces in different rotations, checking to make sure that pieces in the same rotation index and piece type return true and any other result returns false. One edge case we tested was the square, as all rotations have the same points. We found that when testing **equals** with two squares with different rotation indices, **equals** return false, which was the intended and expected output.

## 4.2 **TetrisBoard**

We tested most of these functions using only a couple pieces. This is because we have already tested our TetrisPiece class and can therefore assume all piece types behave the similarly.

### 4.2.1 **testGetCurrentPiece**

We test this method by spawning two new pieces using **nextPiece**. By asserting that **getCurrentPiece** returns the correct piece, then the function is working as intended.

### 4.2.2 **testGetCurrentPiecePosition**

We test this method by creating two pieces as separate samples and placing them at different points on the board. Then, we call **getCurrentPiecePosition** and check if the returned Points list is the same as the one we expect from manual calculations.

### 4.2.3 **testNextPiece**

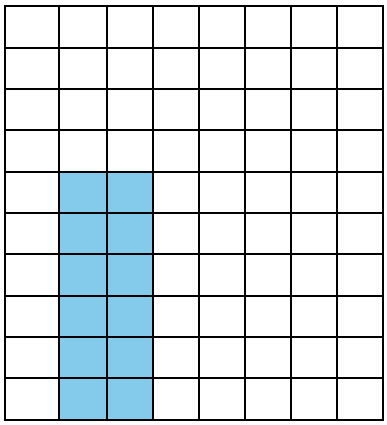
We use **getCurrentPiece** to make sure that the *piece* variable of *board* changes when we call **nextPiece**. In addition, we check that **nextPiece** places the piece in the expected spawn position that we manually set. We can conclude that the method is working because it is resetting *piece* and placing it in the spot that we expect.

### 4.2.4 **testEquals**

We test this method with boards of unequal dimensions and boards of equal dimensions but different pieces, making sure the function returns false. We also manually created two equal boards with the same piece types in the same spots, confirming the function returns true. We can conclude **testEquals** works because it works for both cases.

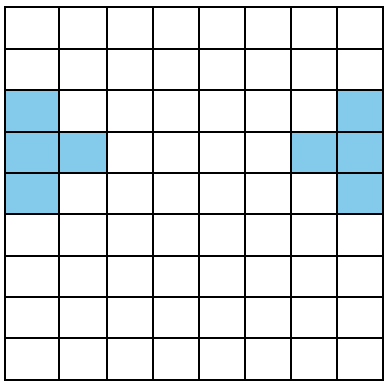
### 4.2.5 **testWallKicks**

Below is a representation of the first test board created for wall kicks.



The stack of pieces is created with 3 square pieces. Our code then moves a stick piece into the space to the left of the squares, testing the wall kicks and ensuring that the stick piece kicks to the right position and orientation or does not kick at all. We test the wall kicks with both clockwise and counterclockwise implementations to make sure they work similarly.

We tested the other wall kick methods with the following board configuration.



We rotated each of the pieces both clockwise and counterclockwise, checking that the wall kicks for both rotations worked as expected. This means that all points were within bounds and the piece was rotated. Finally, we rotated pieces onto each other and confirmed wall kicks off different pieces worked correctly.

### 4.2.6 **testEnums**

This tests the **getLastAction** and **getLastResult** functions.

This function calls **move** with every action enum and checks to make sure **getLastAction** returns the previous action requested. We do the same thing with the result enums, ensuring that the right one is stored after every move.

### 4.2.6 **testGetRowsCleared**

For this function, we fill up the bottom row with pieces and clear the row, checking to see if **getRowsCleared** returns the correct number of rows cleared. We do the same thing with a full row that isn't at the bottom to make sure our function works in this scenario as well. For both of these tests, we ensure that all pieces above the cleared row move down. Finally, we create a board with no full rows, checking that **getRowsCleared** returns 0**.**

#### 4.2.7 Testmove

This function tests **move** by creating a piece on a board and calling every move function on it. With each call, we check that the piece's points relative to the board are changing as expected. We can generalize our tests to the other pieces as well since the move method uses the same logic regardless of piece type.

#### 4.2.8 testDropHeight

We tested this function by creating squares at the top of the board and checking whether the intended value was returned. We tested with nothing on the board, with a piece under our given piece, and a piece and floor under the given piece. Hence, this tests all possible situations when we call **testDropHeight**.

#### 4.2.9 testTestMove

We test this function by creating two boards, each with a piece on it. We then shift one of the pieces left and call **TestMove(***left***)** on the other board. By checking for equality using the **equals** function, we were able to assert whether the boards are equal. Finally, we do the same thing again but using **TestMove(***right***)** instead of **TestMove**(left) and assert that the boards will not be equal.

#### 4.2.10 testGetDimensions

This function tests the **getWidth** and **getHeight** methods by creating boards of different sizes and checking to see if the right values are returned. We also tested invalid dimensions of <= 0 to ensure these cases were being handled.

#### 4.2.11 testGetMaxHeight

This was tested by placing pieces on the board and asserting that the calculated maximum height was the same as that of our manual calculations. We additionally tested **MaxHeight** when rows were cleared from our board and when the maximum height was 0.

#### 4.2.12 testGetcolumnheight

This method was tested by placing pieces on a board and checking the maximum heights of different columns, both with and without pieces. In each scenario, we asserted that the returned column height value was the same as that of our manual calculations.

#### 4.2.13 testGetRowWidth

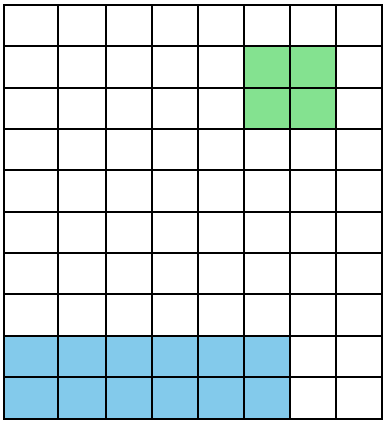
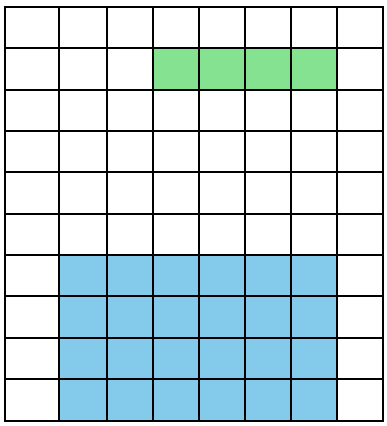
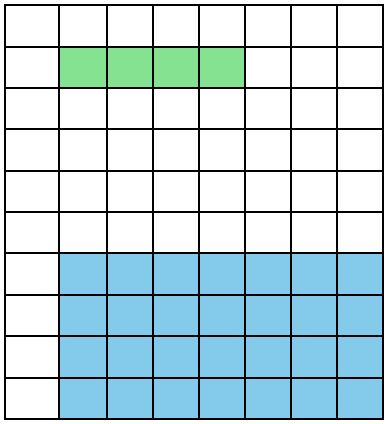
This method was tested by placing pieces to the ground of the board and checking the row width of the ground row. After each additional piece, we checked the new row width and verified the output by comparing it with our manual calculations. Finally, we filled a row and cleared it, validating that the rowWidth was reset to 0.

#### 4.2.14 testGetgrid

This method places a piece on the board and checks to see if our method returns the correct piece type at each point of the grid. We also check that coordinates on the board with no pieces return null.

## 4.3 Brain Testing

We tested our brain by calling **NextMove** on a board with easily verifiable correct solutions. Below are the two board scenarios tested.

In the first scenario, we verify that **NextMove** moves the square right and drops. In the second and third scenario, we verify that **NextMove** moves to the right side of the board, rotates, and drops. Our function will always call either left, right, clockwise, or drop, and we can conclude it works since we have now tested with all moves. Additionally, we can conclude that our brain plays the game well and isn't placing pieces randomly on the board.

### 4.4 Edge Case Testing

- We tested the scenario of when a new board is created with nonpositive integers. In this situation, our code throws an illegal argument exception.
- We tested the function **nextPiece** for when the spawn position places the piece in an invalid place. It exits the function and doesn't add the requested piece if the spawn position is not valid.

# 5 Discussion of Solution

## 5.1 Scope of Solution

The primary functionality of our game is entertainment. Tetris is a game that is easy to learn and mentally stimulating, making it enjoyable for users of all ages. In addition, the game can be used in an educational setting for younger children, teaching them geometry and spatial relationships. Older demographics can also benefit from the increased creativity and social interaction that Tetris brings.

## 5.2 Quality of Solution

Our TetrisPiece class works well to create and modify pieces within the game. It handles many edge cases and runs all accessor methods in constant time. Our

**TetrisBoard** class also works in constant time to update the board, allowing for smooth gameplay and quick rotations to be made by users. Furthermore, our code optimizes the space complexity by only having the board with O(width*height) size.

Regarding our code, the result and movement abstractions work well to simplify our solution and increase readability. Additionally, by decomposing the solution into many different parts, our code is easier to read. Our test code also works well to check edge cases within each method, ensuring that each part of the game works correctly.

## 5.3 Problems Encountered

We encountered a variety of problems when creating Tetris, testing, and implementing a Brain. Below is a list of problems faced and how we worked past them.

- Within TetrisPiece, each accessor method had to run in constant time. To fufill this requirement, we implemented a circularly linked list, allowing each rotation to run quickly.
- The *board* we created had point (0,0) in the top left corner, while the spawn positions given to us had (0,0) in the bottom left corner. In addition, the points of each piece were given relative to the bounding box. We fixed these problems by creating helper methods to convert piece points to coordinates on the board.
- While creating our brain, we struggled to come up with a good way to rank different moves. We were able to solve this by drawing diagrams and running through different scenarios of the game together.

## 5.4 Interesting Results

The most interesting part of our program was our Brain Implementation. We were encouraged to think about different Tetris strategies, and it was surprising to see how complex such a simple game could become. Furthermore, we found that increasing the dimensions of the board made the game easier for the brain to play. For example, when the width was 20 and the height was 40, the brain was able to play indefinitely. Another interesting result we encountered was when testing our program. We were able to find a lot of small bugs that we didn't think about when creating our methods, and it was interesting to see firsthand the importance of testing.

## 5.5 Limitations

One limitation is that we only tested a couple of pieces for some of the **TetrisPiece** functions. This was under the assumption that since all pieces were implemented using the same code, functions working on one piece could be generalized to others. Finally, our

brain was not optimized using the genetic algorithm. If we did apply this method, then results could drastically improve. Furthermore, our model considers every possible position that piece could fall in to, which is O(n*n*k) time, where n is the width of the board and k is the height. This is because there are a total of n possible positions of the current piece, and each time the score of the potential position is calculated, the entire board array is copied. Hence, as the board gets bigger, the time it takes for the brain to run will increase.

## 5.6 Reflection

The creation of Tetris was a challenging project that taught us the design process required to create a program from scratch. We were given more freedom with this project than before, forcing us to think carefully about trade-offs with solutions. Additionally, by decomposing the big project into smaller pieces and implementing smaller methods, we learned a lot about modular programming. As a result, testing was easier. Furthermore, it was interesting to see how methods interacted to produce a full game. Overall, this project taught us a lot about how to tackle a large problem when coding and how to collaborate effectively on large projects.

# 6 Creating the Brain

When creating our brain, we considered options like reinforcement learning and convolutional neural networks. Due to the over-complexity of these models and their difficult implementation in Java, we decided to go another route.

Before each turn, our brain determines all the next possible board states given the current piece. This can be done using a nested loop: the outside iterates through the rotations of the piece and the inside iterates through the possible positions with the rotation. Then, drop will be called at the end of each iteration. After getting each possible board state, we needed to create a method to score them.

We used 2 different quantitative measures of the board along with individual weights to determine the score of each potential next move. The first parameter is *getUmbrella(newBoard) – totalUmbrellas*. This value calculates the change in number of umbrellas when completing a move. *totalUmbrellas* represents the number of umbrellas in the current board, while the *getUmbrella* function finds all null pieces with a non-null piece above it in our new board. Then, the umbrella value at that point is the geometric sum, with ratio 2, from 1 to $2^n$, where n is the number of null pieces under the initial null piece. The higher the change in umbrella value, the more we want to punish the model. Consider the case where placing the new piece would cause a blockage to an empty space that could

potentially get rid of a lot of rows. Hence a piece placed there would ruin the game and cause a lower total score.

We also punished the model for large differences in adjacent column heights. We added this weight when we kept on seeing a common problem: the brain would continuously build on the middle and ignore the sides. A major part of Tetris strategy is keeping an even surface. Hence, we deployed the *averageDiffHeights* function which returns the average of the quartic difference between adjacent columns. We quartic the difference to punish the larger differences more.

Finally, we manually adjusted the weights of each scoring metric for optimal performance. We did this by taking the average Tetris score of 10 trials for each set of weights and finding a local maxima. The average Tetris score of our final model ended up being 358.5 pieces placed, with the maximum being over 2,000. Given more time, we would implement a genetic algorithm to find optimal weights for each parameter.

Some other functions we tried were floodfill to find the number of holes and *getMaxHeight()* to make sure our brain was distributing blocks evenly horizontally. However, in the end, we found the three functions above to be the most effective.

Calculating the space and time complexity, we get O(n*n*k) and O(n*k) respectively, where n and k are the width and height of the grid.

# 7 Log of Time Spent

| **Date:** | **Activity:** |
|---|---|
| 10/1 | X drove for 2 hours |
| 10/2 | Y drove for 2 hours |
| 10/5 | X drove for 3 hours |
| 10/5 | Y drove for 3 hours |
| 10/6 | Y drove for 2 hours |
| 10/7 | X worked for 2 hours(report) |
| 10/7 | Y worked for 2 hours(report) |
| 10/8 | Y worked for 3 hours(report) |
| 10/9 | Brainstormed and implemented testing, each drove for 2 hours |
| 10/10 | Y drove for 3 hours(brain) |
| 10/10 | X drove for 1 hour(brain) |
| 10/11 | Y worked for 2 hours(report) |
| 10/11 | X worked for 2 hours(report) |