1 Overview

While Tetris was invented in the 1980’s, it has still maintained its popularity where it holds its position as the most popular video game of all time. While simple and easy to understand and play, there is a lot of complexity within the interpretation itself. For this assignment, we were tasked with developing a larger code base by individually solving and testing smaller sub-problems to construct a game of Tetris. This assignment also emphasizes the skills of building off of others’ code, building with longevity in design solutions, and problem solving when constructing a brain. Additionally, this assignment introduced new goals, such as building stronger test harnesses and detecting more non-intuitive edge-cases given the open nature of the project.

2 High Level Solution Design

For our solution, we had three overarching subproblems to consider to solve the larger problem: the TetrisPiece, TetrisBoard, and the Brain. In both TetrisPiece and TetrisBoard, we implemented the methods that were outlined in the Piece and Board interfaces. We discuss our high level solution designs in the following sections.

2.1 TetrisPiece

Our solution design for TetrisPiece was mostly trivial and just generally involved implementing getters, setters, or simple operations based on the setup from the constructor. The most noteworthy aspect of our TetrisPiece file was our custom-made circularly linked list. In our generateRotations() method, we generated 3 other TetrisPiece classes that pointed to the next clockwise rotation, and when we reached the final TetrisPiece, pointed back to the original spawn TetrisPiece object. This way, our rotations all run in constant time, as we just simply refer to the next TetrisPiece in the linked list for a clockwise rotation or move 3 elements down to get a counterclockwise rotation. We calculated a formula for each pixel to rotate to the clockwise location, where (x,y) maps to (y, width-x-1), which we looped through every point and adjusted to create the rotated piece. We decided to implement our linked list in this way so we wouldn’t have to
loop through the linked list to make it doubly linked, as moving 3 elements down is still constant time and is negligible.

2.2 TetrisBoard

Our board was arguably the biggest aspect of the program, and the majority of the functionality was in the move() method. In it, we included a switch statement to check all the possible actions that could be performed and, after a series of checks, will perform the specified actions. There isn’t a lot to note within the move() method itself, as it consists mostly of for loops and checking to make sure that the operation is valid and in-bounds. The only other high level design detail to detail is how, for the testMove() method, we found it necessary to make deep copies of our arrays rather than shallow copies to make sure we don’t modify the original board and only change the properties of the new test board. We discuss more details about TetrisBoard in Section 3: Design Considerations.

2.3 TetrisBrain

In the time we’ve been working on this project, we have enumerated over many different possibilities, none of which produced anything that we were happy with. We first started with the LameBrain implementation by adding rotations to each enumeration before dropping it. We also tried changing the DROP commands to DOWN to allow for some trickier moves that involve moving the current piece under others to fill up holes and gaps. Then, for our score board, we tried many different permutations and combinations of multiple variables to adjust the behavior, which include the max height, the number of holes/gaps with pieces above it, troughs (one-wide gaps between pieces), the current piece height if dropped, and many more. However, none of the combinations fruited anything much better than the LameBrain. We even considered cases where we moved pieces closer to walls and make things more. After over 8 hours of combined effort on working on the brain, we decided to consult outside resources to figure out if we could find any other metrics that could yield any results while employing the Gilligan’s Island Rule. The References section at the end of this report includes what resources we utilized for reference. However, nothing that we looked at yielded any proper results to what we hadn’t considered before.

Eventually, Ansh figured out that our enumeration method could be vastly improved, and he implemented a superior enumeration method to what Rahul had previously wrote. We discuss more specific details in Section 3.4: The Brain.
3 Design Considerations

3.1 Advantages

The advantages of the way we designed the TetrisPiece and TetrisBoard classes is that pretty much all the methods that were overridden from the appropriate interfaces were of constant time, and most of the harder computation took place outside of these methods via helper methods. For example, in the TetrisPiece class, the methods responsible for computing the 3 skirts we had computed the appropriate arrays and then stored them in a TetrisPiece class variable. This variable was then returned for the overridden skirt algorithms in constant time. A similar approach was used to generate all possible rotations for each piece, which we chose to store in a circularly LinkedList for easy access. We also chose to create an alternative constructor for TetrisPiece to specify rotation arrays and indices, so we could generate a piece in any rotation value and therefore generate any of the 28 possible pieces.

As for the TetrisBoard class, we tried to take advantage of abstraction wherever we could, and most of the methods were easy to implement in constant time (all they did was return a certain class variable that was modified in assorted helper methods). Additionally, our placePiece() function came in handy, as it was called for specific cases in the DOWN section of the move method as well as every time in the DROP section. The placePiece() method handles a lot of functionality associated with placing pieces on the board, including modifying the static 2D array values, clearing rows, and updating column heights and row widths. The integration of all of these functions within one method allowed for simplicity as we created the move method, which was, of course, the central component of the entire TetrisBoard class. It contains a monstrous switch statement that executes the appropriate action and returns the result of 7 possible actions depending on the board configuration, and our use of a switch statement was handy in skipping unnecessary code for other actions. The last point we would like to discuss for the TetrisBoard class was the testMove() method, which required the creation of deep copies of all instance arrays, including the board itself. The new board then simply called the move method and the method outputs a new boared with the move done, and this method was called repeatedly in the brain implementation when trying to enumerate possible options.

Finally, for the NotLameBrain class, the use of the three-method system derived from the LameBrain system was easy to understand as well as to follow. The enumerateOptions() method is still simply responsible for adding possible moves to the static ArrayList, and we improved upon LameBrain’s implementation by allowing for rotations as well as combinations of rotations and translations. Additionally, we improved upon the scoreBoard() function by modifying the scoring algorithm using 4 primary metrics: the total height of all columns, the magnitudes of the differences in the column heights, the number of rows cleared by a move, and the number of holes left in the block configuration. We used each of these to add or subtract from the score, and multiplied them by constants to change their relative weights. These were the constants that were tested in
the genetic algorithm, and the primary purpose of testing was attempting to figure out which scenarios these constants would help catch to assist the Brain in making a good next move.

### 3.2 Disadvantages

Although many of the implementations we wrote had a lot of advantages and efficiency, there were certainly some disadvantages to the way we decided to do things. For example, in the TetrisPiece class, we used separate helper methods to get the left, right, and bottom skirts because we created each of them at separate times, and a better idea might have been to have a single nested loop going through the entire object and returning all skirts at once. Similarly, in the TetrisBoard class, the dropHeight() method is inefficiently written. When we first wrote it, we forgot to account for the possibility that a piece was dropped below another piece, and we therefore used the columnHeights to determine the drop height. After realizing this mistake, we added an if else statement and then coded the other scenario, but it may have been more efficient to assume said scenario and calculate the difference between the piece and the next highest block based on the skirt values. However, as far as functionality is concerned, the method works for all possible cases, and is therefore a valid implementation.

Lastly, the NotLameBrain class has the major disadvantage that it does not work nearly as long as it could if it were optimized. It seemingly tests all possible drops and moves towards the best one, but from a black box testing standpoint, it simply does not perform as well as it could. Perhaps the reason for this will become clear upon a more thorough inspection of the code.

### 3.3 Edge Cases

One of the external scenarios that we considered, as stated before, was the result of dropping a piece and calculating the height it needed to fall. Our initial implementation involved calculating the difference between each piece and the column height and dropping the piece that amount. However, before long, we learned that this method simply would not work for pieces that happened to lie below other pieces in the same column, such as if an overhang scenario occurred. As such, we had to modify the dropHeight method to consider this special circumstance when calculating how far to drop an object.

Additionally, we realized that the piece was not set to null after it was placed, which would not matter for the game itself, since a new piece is initialized immediately. However, during the SanityTests checking, we realized this could be a possible issue for testing since the placing of a piece should mean the board no longer has a playable piece, so we had to go through and manually set the current piece to null after it was successfully placed using the DROP or DOWN methods.

Another edge case we ran into towards the end of the project arose as a result of our
adjusted implementation for rotations to pass the SanityChecks. After modifying our implementation slightly, we realized that an ArrayIndexOutOfBoundsException would occur for certain pieces in certain orientations only when we tried to drop them while they were below another piece. The solution was to fix an indexing issue with our rotation array as well as modify some of the values in the dropHeight method until it worked for all cases, verified via black box testing.

3.4 The Brain

The primary brain being submitted with this assignment is the NotLameBrain, which is a modified version of the LameBrain class that expands upon the enumeration and scoring methods. The enumeration method is expanded to not only include left and right moves (translations), but also all possible rotations as well as combinations of rotations and translations. The associated firstMove stored with each combination goes by implementing the rotation first, then the translation, then the drop function. Additionally, the more important part of the NotLameBrain class was the scoring method, which, as stated before, is based on four plus an optional two for a total of six possible metrics. The totalHeights metric is the sum of all heights for each column, and the Brain tries to minimize this value. The totalHoles method is the amount of null spaces with filled in blocks above them, and the Brain tries to minimize this value as well. The rowsCleared metric calculates the number of rows cleared by a certain move, and the Brain tries to maximize this value. Finally, the colDifferences metric measures the sum of the absolute value of the the differences of adjacent columns to calculate the variance, and the Brain tries to minimize this value as well. Additionally, we added floorScore and wallScore methods to add a higher preference to moves against the floors and walls, which we found were filled in less often than middle pieces even in extreme scenarios. Combined, these six metrics change the returned score variable for each board and allow it to determine which is the best option, and this is how to the Brain functions.

3.5 Assumptions

The primary assumptions made in this project are as stated on the handout. Tetris, over the past few decades, has evolved and grown into many different variations, each of which differ in slight ways that can change the rules and strategies immensely. As such, our implementation picked one specific rule set, namely the one with Super Rotations and a Wall Kicks system. Another main assumption we made when designing the code was that it would mainly be implemented using the GUI provided, not including testing scenarios. As such, a lot of our black box testing and verification was done through the use of this interface, and the same held for the Brain.
4 Assignment Solutions

4.1 Scope of Solution

The Tetris program that we created is mainly designed for the specific version of the game that we have used for testing and implementing. The code has been generalized for boards of various sizes, but many of the aspects of the game itself were hard coded into the starter code and therefore would be difficult to extract and place into a different setting. For example, the system for wall kicks as well as the bounding box and rotation values for each of the pieces is hard coded into our implementation, and any application of our classes outside of this setting would require manual changes of these implementations. However, though the TetrisBoard and TetrisPiece classes are generally specific to the setting we have used, the NotLameBrain class is easier to apply in other areas. It is a very simple artificial intelligence which works by enumerating all possible options, calculating which one is the best one, and then moving the board towards that option. Since it does not consider the inner workings of the program itself, it would be easier to apply to other settings: the only modifications required would be specific enumeration or scoring details. If a firm or other outside source wished to use our implementation of the Brain, the general principle behind it would remain the same, and all they would have to do is figure out various options and scoring algorithms.

4.2 Problems and Limitations

Although we spent a huge amount of time on the Brain, it is very limited in what it can do. Though it manages to run for around 80 pieces somewhat consistently and manages to clear around 20 rows for a good run, it is not as good as many of the AIs we have seen, which can comfortably clear at least 3000 rows consistently. At some point, we simply had to stop trying to improve our Brain further because we had reached an impasse, and the Brain thus became the primary limitation of the Tetris implementation we made. We think the primary problem with the Brain is the way we enumerate the possible options, as there may be certain cases where objects are missed. Alternatively, the problem could also lie with our scoring algorithm, but outside of the metrics we considered, we could not figure out how to improve the scoring algorithm excluding simple constant adjustments, which were handled by Rahul’s genetic algorithm.

5 Additional Practice [Karma]

For extra practice, Rahul implemented the Proper Piece Generation. In the JTetrisProper.java file, Rahul initially copied and pasted the code, which you can view in the early deadline submission. However, he later refactored the code so that it extended JTetris like JBrainTetris. The Proper Piece Generation creates a permutation of all the pieces and goes through that entire permutation before repeating any elements. This way, no piece is guaranteed to appear three times in a row.
Also, with help from Jack Roper, Rahul also added the Tetris A Theme song on loop indefinitely (basically) under the JTetrisProper.java file. It creates an audio stream and plays the theme song on loop when you run the GUI for the user’s playing pleasure.

Additionally, Rahul attempted to write a Genetic Algorithm for the brain. The basic idea for our genetic algorithm is create a large population of brains, run each one five times to remove some randomness factor and hopefully account for regression to the mean, then mate the highest scoring individuals to create the next generation by weighting the genes of the stronger parent more than the weaker parent. While the algorithm was not able to produce any results at the time, we know that the genetic algorithm process is correct since it was modeled after another simple genetic algorithm Rahul constructed earlier in preparation for this Karma. However, at the time of this writing, we were not able to get any meaningful productive results from the genetic algorithm, which we hypothesize to be a result of the scoring and fitness functions rather than the algorithm process itself. We were only able to drop 122 pieces when implementing Proper Piece Generation, which isn’t ideal for a genetic algorithm. However, learning about genetic algorithms was a fascinating exploration about how ideas from biological evolution can shape an optimization technique in computer science.

6 Testing Methodology

6.1 Black Box Testing

Our Black Box Testing was fairly rudimentary. To test general correctness for certain actions, we could visually inspect the Graphical User Interface to see certain actions being performed, such as moving left, right, up, down, drop, and rotations. We could also check for bounding box correctness with piece overlap, as well as certain edge cases we constructed manually, such as dropping a piece under a stack of other pieces or checking for wall kicks against the sides of the board. While this method allowed us to find many bugs and fix them all, to ensure confidence in our program, we developed our testing harness.

6.2 Sanity Checks

Although we did not write the SanityChecks.java file, we felt it important to include in this report because it helped us realize many aspects of our code we had failed to consider. The first and foremost was the setting of the current piece in any board to null following a successful placement rather than simply letting it be overridden by whatever piece was called next. Another SanityCheck that we failed was the one that asserted a piece was equal to itself rotated four times, as the test essentially forced for the use of a LinkedList rather than our initial array implementation. However, following the adjustment, the test passed, and all other assertions within the SanityCheck file passed the first time we tried them.
6.3 Testing Harness - PieceTests

The PieceTests.java file was written to test all 7 pieces and their functionality as far as rotation, skirt obtaining, and other aspects were. We designed 7 unit tests, 1 for each piece type, each with an assortment of assertions that must all hold for the test to pass successfully. Some examples of assertions we made included testing the length of the point array to always be 4, asserting a piece was not equivalent to a rotated version of that piece (except for square), testing the height and width of the bounding box, and the most difficult part, which was testing for actual skirt values for specific rotations. While we chose not to enumerate all possible skirt values for all possible rotations, we chose 4 or 5 specific and different instances that tested combinations of methods and values together, which allowed us to ensure to a reasonable degree that the methods were working properly. Once all 7 tests passed, meaning all of the assertions within each test held, we were finished with the test harness for the TetrisPiece class.

6.4 Testing Harness - BoardTests

For our BoardTests.java file, instead of testing every single case like we did in PieceTests, we mostly focused on edge cases and global behaviors. We first tested for basic stuff, such as getHeight, getWidth, and that a square piece returns a square piece type. We also tested for individual method correctness, such as getRowsCleared() to indentify correctness. More interestingly, we tested individual cases for each of the 7 piece types, which sometimes included certain rotations. For example, we tested collision with all the pieces and certain rotations dropping, first onto the floor and then onto each other, and we also tested when they try to move left or right into other pieces. We then tested the bounding boxes by rotating them and then dropping. For our edge cases, we considered a number of possibilities, including moving pieces off the grid and dropping pieces when they are below the columnHeight for that column. We were actually able to catch certain bugs we assumed we had fixed but didn’t by doing this.

6.5 Testing Harness - NotLameBrain

Finally, the last testing harness that we created was for the NotLameBrain class, which overall was very difficult to test because you essentially had to predict what your artificial intelligence would find as the best move. However, there were a few simple scenarios that I was able to trace through the Brain’s process with the time that I had, so I only tested a few scenarios to ensure the Brain was picking the proper move. These scenarios involve the placement of a square, stick, and left dog piece, which was easy to predict because the operations were running on an empty board. Given that all the tests passed, we are reasonably confident that our Brain is picking the best moves based on the algorithm we provided it. Additionally, we added another test to ensure that the brain, when initialized, was not null, as is standard programming convention.
7 Interesting Results Gallery

For your enjoyment, we have prepared a series of images to display some interesting results we encountered during our time developing the Tetris program:

This image resulted from a failure in our initial DROP method: the green piece failed to drop to the bottom when the function was called directly diagonal to the red piece.

This image resulted in a failure with the down method that resulted in the purple piece on the right somehow splitting into various components.
This image contains our first instance of a game that successfully completed without any visual errors as determined by black box testing. Notice the tower in the middle: a lot of our brains gave up after a while and just built a tower in the middle.

This image shows what happened after we completed the SanityChecks but ran into the error of attempting to drop a piece below another piece: an ArrayIndexOutOfBoundsException was thrown.

8 Final Comments

If you look at our Programming Log in the following section, you will see that we have put over 30 hours of time into this project. While the results for the brain were not ideal, especially given the astronomical time both of us invested into it, this project was a lot of fun to implement due to the scope of the problem and how open it was. If we were to make any improvements, it would definitely include making a better and more competent Brain. Even after this assignment is submitted, we will definitely continue to develop our genetic algorithm to make a Brain that could potentially score a lot more points.
Here we describe any resources we used or anybody that we collaborated with in any fashion:

1. We discussed with many, many people to figure out how to develop a competent brain.

2. We looked at this website for help with our brain, specifically to find additional metrics to include, not for the individual values nor the genetic algorithm: https://codemyroad.wordpress.com/2013/04/14/tetris-ai-the-near-perfect-player/

3. We googled what year Tetris was made in (1984)
4. Jack Roper helped us implement an audio system to play the Tetris A Theme song as karma