1 Introduction

We implement Tetris, a classic computer game in which a player must arrange variously-shaped falling pieces into rows on a 2D grid. We additionally implement a computer-operated “brain” to play Tetris. The assignment encouraged us to consider the design of the way each component of the program interoperated with the other components. Additionally, we were meant to apply Java-specific concepts relating to interoperability, dependency, inheritance of Objects, implementation of Interfaces, overloading constructors, and algorithmic complexity. Some of our goals in the implementation of the game were to consider the effect of our individual subroutines within the context of the rest of the program, as well as to pay close attention to algorithmic efficiency — ensuring that certain operations would take constant time using caching. Our goals with the brain were to learn about basic reinforcement learning and practice different ways of exploring a parameter input space. One final goal was to practice testing a large and complicated input space such as Tetris, and how to write good tests that build up from smaller components and have as much coverage as possible.

2 Solution Design

2.1 Private TetrisPiece Constructor

In order to achieve constant time rotation and translation of tetrominos, it is necessary to precompute all rotated states and skirts during the construction of the TetrisPiece object. These results are then placed into an array of length four. Within the context of the game, this means that these computations are only done when a new piece appears on the board, and rotating simply accesses data in memory. Since the Piece interface requires us to return another TetrisPiece following a rotation, in order to retain these precomputations we pass (through a private constructor to the new rotated piece) a reference to the body array of possible rotated pieces and the current rotation index. This private constructor therefore has a different signature from the default one, accepting three arguments instead of two: PieceType type, Point[][] possiblePieces, and int rotationIndex.

2.2 Rotations and Wall-Kicks

The rotation computations themselves were performed using a rotation matrix with 90° and −90° for clockwise (0 1 \begin{array}{c} \end{array} \begin{array}{c} 1 \end{array} \begin{array}{c} 0 \end{array}) and counterclockwise (0 −1 \begin{array}{c} \end{array} \begin{array}{c} 1 \end{array} \begin{array}{c} 0 \end{array}), respectively. Since the matrix multiplication operates under the assumption that the center is located at (0,0), one tricky part to rotating is finding the center point to rotate about. Since the center cannot be a block itself, it must be a point between blocks. For this reason, the center is calculated as (width / 2.0 - 0.5, height / 2.0 - 0.5).

To avoid unnecessary computation, wall-kicks are only considered if a regular rotation fails. In that case, the supplied wall-kick arrays are used in order of priority. Once a wall-kick translation...
is found that results in a legal state, the move is accepted. If no wall-kick results in a legal state, then no move is made.

2.3 Illegal State Detection

We found that we often wanted to check if a particular move would result in an illegal state within the current context of the board. That functionality is abstracted out to a private method called `isColliding()`. This method accepts a `Point[]` point body and position on the board as parameters, and checks whether they are legally able to place on the board, including bounds checking and collisions with previous blocks, returning true if so, and false otherwise.

2.4 Row Clearing

The only time that rows can be cleared is after a piece is placed, so we check the entire grid for cleared rows every time a piece is placed. Additionally, we keep track of `columnHeights` and `rowWidths` and update them after each piece placement, to allow us to perform faster row clearing. During a row clearing, we start with the bottom row of the grid and move up. If according to our tally, a row is full, then a counter is incremented by one. This counter denotes how much every subsequent row must move. For instance, if a particular non-full row $R_j$ has one full row somewhere under it $R_i$, then by the time the algorithm reaches the $R_j$, the counter will equal one, and $R_j$ (as well the preceding rows) will have shifted down by one. Similarly, if $R_k$ is above $R_j$ and another full row, then it will shift down by two.

One possible way to make this particular operation faster would be to store the grid as a linked list, where each node contains a row array, although that would be detrimental to lookup time with respect to columns.

2.5 Drop Height Calculation

To emulate gravity properly, it is necessary to calculate the exact drop height at which a tetronimo is supposed to stop moving, and the next piece is generated. To calculate the stopping point, we find the maximum of \((columnHeight[x] - skirt[x])\) for each column “shadowed” by the piece vertically.

2.6 Brain

For the brain, we followed a similar approach to LameBrain, where we try all possible moves and choose the best one according to some fitness metric. Unlike LameBrain, however, our brain also checks every possible rotation of the current piece.

The actual brain implementation is based on the concept of Actions, which are a combination of a rotation of the current piece and an x position at which to drop it. If the brain is in the middle of an Action, `nextMove` returns the appropriate move to move the current piece closer to the location and rotation of the next Action. Once the piece reaches that position, the brain drops the piece and computes the next Action. It does this by enumerating all of the possible next Actions and computing the fitness metric for each resulting board state; it then picks the board state with the best fitness metric and its corresponding Action.
The most interesting and nuanced part of the brain is the computation of this fitness metric. The overall fitness metric is always a linear combination of multiple individual metrics, where the weights of the linear combination assign a relative importance to each metric. Over the course of testing our brain, we designed a total of 5 metrics:

1. **Total height**: The sum of all of the heights of the columns (penalized)
   Rationale: The brain should favor moves that don’t increase the height of the board as much

2. **Rows cleared**: The number of rows cleared by the next action (rewarded)
   Rationale: The brain should favor moves that clear rows

3. **Height variance**: the variance (square of standard deviation) of all of the heights of the columns (penalized)
   Rationale: The variance of the column heights gives some notion of “bumpiness”; the brain should favor moves that minimize this bumpiness, since bumpy boards are harder to clear

4. **Holes**: The number of holes in the board, defined as an empty space with a filled space somewhere above it (penalized)
   Rationale: Holes are much harder to clear, so the brain should favor moves that minimize their number

5. **Hole blockage**: The number filled spaces above the highest hole in each column (penalized)
   Rationale: If a hole already exists, it gets harder to clear for every filled space above it that is “blocking” it; the brain should favor moves that minimize this blockage in order to encourage recovering from creating holes

Once we have the metrics, the real challenge with maximizing the performance of the brain is choosing the weights to assign to each metric. By observing the behavior of the brain and manually tuning the weights, we could get performance comparable to an average human (clearing at least 50 lines per game). However, truly maximizing the performance requires a much more principled approach.

The first requirement for a more principled approach is an objective way to evaluate a set of weights. For this purpose, we created a trainer for our brain (which is called BrainyMcBrainface), called BrainyMcBrainfaceTrainer. The trainer implements a way to play many simple Tetris games using the brain without any additional unnecessary features from JTetris, like drawing the pieces to the screen. The trainer implements a method `executeBatch(double[] weights, int size)`. This method plays `size` Tetris games with the brain using `weights`, and computes the average number of lines cleared over all of those games. Since the number of lines cleared can vary immensely from game to game, averaging over many games is necessary to determine the actual mean performance of a strategy.

Using the trainer, we were able to much more effectively search the weight input space and improve the performance of the brain. Our weight optimization strategies are described in detail in section 3.5: Karma below.

### 2.7 Assumptions/Non-Assumptions

Notably, we do not assume anything about the board or the pieces themselves. If a new type of piece is later added to Tetris, then all of the components (with the exception of wall-kicks,
which are only defined for pieces with a certain bounded box size) will work properly. The dropping and row-clearing algorithms also are invariant under any size of board. However, if the bounding box is non-square, then the rotation algorithm will fail.

Behaviourally, we assumed that the reaction time of any human player is an order of magnitude slower than the time it takes to rotate or translate a piece, otherwise the game becomes unplayable. There are also some assumptions about restrictions on the movement of pieces that are imposed by this particular variant Tetris, where the board is always 10x20 with four other rows at the top for new pieces to appear.

For the brain, we assumed that the goal of the brain is to play as long as possible, and the metric we use to assess that is the number of lines cleared until the end of the game.

3 Discussion

3.1 Scope/Quality

By definition, the solution applies to the standard Tetris board and all of the standard Tetris pieces. However, as described earlier, the scope is actually larger than that — with the exception of wall-kicks, the current Tetris gameplay should continue over to arbitrary board size and new arbitrary pieces, provided that the bounding boxes are square. In other words, the rotation and drop algorithms are invariant of the specific pieces in the game.

The resulting Tetris game is correct and follows all of the specs, however, given the restrictive keybindings and medicore animations, it is not enjoyable to play for long periods of time. The quality of the Brain is described below.

3.2 Efficiency

All rotations operate in constant time. This is because all of the rotation calculations occur when a new piece is generated and cached across all rotated versions of that piece. The tradeoff is that generating pieces will take slightly longer, but this is worthwhile because timing is far more important in the middle of a particular piece’s gameplay than it is between pieces.

The board also implements all of its getter methods in constant time. Most of the computation happens when a piece is placed; for example, the column heights and row widths are recalculated and cached. That way, they can all be retrieved in constant time.

3.3 Problems Encountered

We encountered an off-by-one error before we realized that the center of rotation for a piece is not a block; it is actually a point. For instance, if we have a 4x4 bounding box, then dividing both the width and height by two will result in a presumed center point of 2x2. However, this block is not actually in the center of the bounding box (since the bounding box is even) for the purposes of the rotation matrix. To get the real center point, we have to shift the resulting point by 0.5.
There was also an issue with `testMove()` where we initially didn’t create a deep copy of the grid array, so the new board’s state was tied as a reference to the old one. This was discovered when trying to work with a resultant board from `testMove()`.

### 3.4 Brain

With manual tuning of the weights, our basic brain strategy can easily clear 50+ lines per game with the random piece selection algorithm (comparable to a human player). With more advanced tuning, we were able to increase this to a maximum of approximately 150 lines cleared per game. (Of course, this is an average over many games, and the brain could potentially clear many many more lines in a single game if it got lucky).

We also implemented the fair piece selection algorithm (described in section 3.5: Karma below), which makes the game significantly easier. With this algorithm, the brain was able to clear over 900 lines per game on average.

While this performance is good, it is certainly not as good as it could be. From a little bit of online research [2], as well as talking to other students, it seems that with the right metrics and weights, it is feasible for a brain to be able to play essentially forever (clearing lines in the millions). We think our metrics are good, and it was the weight optimization that is holding our brain back from that kind of performance. If we had more time, we would try some sort of genetic algorithm or policy gradient method to find the global maximum of more weights.

### 3.5 Karma

#### 3.5.1 Fair Piece Selection

Rather than the naive default algorithm, which simply chooses the next piece randomly, we also implemented the “fair” algorithm described at [tetris.wikia.com/wiki/Random_Generator](tetris.wikia.com/wiki/Random_Generator). The fair algorithm simply puts all 7 tetrominos in a list, shuffles them, and then serves all of them in that random order. It only picks a new 7 once the previous 7 have all been served to the player. The implementation only required the addition of 2 more state variables to a game implementation, `currentPieceBatch`, the list of pieces, and `currentPieceBatchIndex`. This algorithm makes the game significantly easier, since it prevents the player from getting a long string of “bad” tetrominos, and ensures they get at least one of each in each batch of 7.

#### 3.5.2 Brain Weight Tuning

After realizing that manually tuning the weights for the brain wasn’t the most effective, we then primarily tried random search: randomly choose a set of weights, measure the performance over a batch of games, and repeat while recording the best set of weights so far.

With all 5 metrics (and random piece selection), running a random search overnight yielded a maximum of approximately 100 lines cleared per game. We knew this wasn’t nearly as good as it could be, and reasoned that the primary issue was the size of the search space: 5 dimensions is too large to find the global maximum by searching randomly. Since we didn’t have time to implement a more advanced optimization strategy (such as a genetic algorithm), we came up with a much easier way to find the maximum: reduce the search space.
We reasoned that some of the metrics may have some redundancy, and given our limited time, did not add enough information to be worth the increased search space. After some experimentation, we found that by far the two most important metrics are holes and height variance. (Interestingly enough, it is exactly these two metrics that are important: either one of them alone performs terribly, as does any combination of two metrics that doesn’t include both of them). Two weights is small enough that the entire space can be searched fairly quickly with a random search; after only a few minutes, the training found a maximum of approximately 150 lines cleared per game.

In fact, we realized that with two weights, the input is actually just a two-dimensional vector that can be plotted on the x-y plane; furthermore, since a linear combination is scale-invariant, the magnitude of the vector doesn’t matter: only its direction/angle. That means that the weight vector can be parameterized by a single value–its angle from the positive x-axis–while its magnitude can just be 1 (on the unit circle). We then used this property to plot the entire weight space vs. performance, since we thought it would be very interesting to actually visualize the entire input-output space of a problem like Tetris.

![Figure 1: Angle of 2D weight vector vs. average performance of brain; random piece selection. The plot was generated using Python and matplotlib. The source code can be found in plot.py.](image)

Since we know that holes and height variance should both be penalized (have negative weights), we only have to cover angles from $\pi$ to $\frac{3\pi}{2}$ (where both the x- and y-coordinates are negative). Each blue dot is the performance of the brain at that angle averaged over 300 games. Even with an average over so many games, we can see there is still a lot of variance in the performance. For that reason, we then applied a Savitzky-Golay filter to smooth out the data (the red line), and found the global maximum of the performance to be around 150 lines per game at 3.82 radians (the black dashed line). The point at which the weights are equal is $\frac{5\pi}{4} = 3.92$ radians, so this maximum point of 3.82 corresponds to weighting holes slightly more than height variance (we...
treat height variance as the y-axis). This makes intuitive sense.

We found the shape of the graph to be very interesting—it is surprisingly simple and clearly convex (only has one local extremum at the global maximum). We expected the input-output space of Tetris to be much more complicated, since the weights go through so many transformations before impacting performance; however, while there is a lot of variance between games, the simplicity of the overall shape is clear. This implies that the input-output space in higher dimensions (more weights and metrics) may be fairly simple and convex as well, meaning it would perhaps not be as difficult as we thought to find the optimum for a larger number of weights (using a slightly more advanced strategy than random search).

All of the above research and data is using the naive random piece selection algorithm. Once we implemented fair piece selection, we did the same thing using that algorithm:

![Figure 2: Angle of 2D weight vector vs. average performance of brain; fair piece selection. The plot was generated using Python and matplotlib. The source code can be found in plot.py.](image)

First of all, the performance is clearly much better across the board, since fair piece selection makes clearing lines much easier in general. The shape of the graph is still simple and convex, generally similar to that of random piece selection. It differs in that the peak is narrower and more bell-shaped; it is also more symmetric, falling off exponentially on both sides. Additionally, the variance near the peak is much higher, even when each point is averaged over 300 games. The maximum is approximately 900 lines per game at 3.77 radians, meaning it again weights holes slightly more, and by a greater amount than the optimal strategy for random piece selection.
4 Testing

All testing is done using JUnit 4 in a class called “Tester”. Unlike critter, a separate “test harness” class is not required, because the board and piece interfaces already include all methods necessary to mutate them and retrieve their state.

4.1 Piece Testing

The first unit test verifies that all the basic functions of the TetrisPiece work correctly, since the board and all other components are dependent on the piece. The test uses the left dog, and verifies that all of its getters, its body, and its skirt return the correct values after rotating different numbers of times.

4.2 Board Testing

The second unit tests verifies all the basic functions of the board, but not wallkicks and line clearing, since these more advanced test assume that the basic functionality of the board works first. It first tests the edge cases of the board (e.g. a width and height of 0, pieces placed out of bounds). It then tests basic behavior on a normal-sized board—all of the movements, placing pieces by moving them down or dropping them, etc—and verifies that all of the getters return the correct values for each state. It also tests equals and testMove at the same time by creating new boards and putting them in various equal and unequal states.

4.3 Wallkick Testing

The next unit test is of wallkicks, which is a more advanced functionality of board. It tests wallkicks against the 2 walls and the floor using the stick tetromino.

4.4 Row Clearing Testing

The next unit test is of row clearing, which is another more advanced functionality of board: full rows must clear, and any pieces above cleared rows must shift down by the correct amount. In order to reduce the complexity of the testing, we tried to come up with a single formation that would test most of the aspects:

![Diagram of a formation for row clearing testing]

This formation exists on a board that is only 5 wide, and is completed by placing a vertical stick piece in the rightmost column, which should clear two rows. This tests several things at once:
1. The ability of the board to clear multiple rows (two)
2. The ability of the board to clear multiple nonconsecutive rows (it should not clear the row with the hole)
3. The ability of the board to shift down pieces after clearing (the resulting formation should have the top two blocks shifted down by two, and any blocks below that shifted down by one)

The test sets up this formation, drops in the vertical stick on the right, and then checks that the number of lines cleared and the resulting formation is correct (as well as that the column heights and row widths have been updated).

4.5 Brain Testing

The final unit test makes sure that the brain is working. The brain is the only component that does actually need an additional test harness to get it to interact with the board and read its output. Thankfully, the BrainyMcBrainfaceTrainer already contains all the code to test the brain in a real Tetris environment, and measure its output (we can assume at this point that the Piece and Board work properly due to the previous tests). Therefore, the brain unit test simply initializes a trainer, and verifies that the brain can clear more than 50 lines with its optimal weights.

4.6 What We Learned

Testing Tetris was easier than testing Critter in one sense, which was that it did not require as comprehensive of a test harness. However, writing the actual test cases was much harder due to the size of the input space. There are simply too many states that the board can be in, and many actions that can be applied to each of those states; it’s intractable to see if the board behaves correctly in every single case. Instead, we had to break it down into what we thought were the most significant or unique cases, meaning that if those cases work, then it is unlikely for any other cases to not work. For example, if moving a piece works in one spot in the middle, one spot against the edge, and one spot against another piece, then it probably works in all other spots. Of course, the testing will never be 100% sound, since there could always be edge cases at spots in the input space that we didn’t think of. The challenge is thinking of as many of the unique and edge cases as possible to make sure that as much of the input space is covered as possible.

We also learned how to write dependent tests that build on each other. We tried to write the tests in an incremental order such that larger components first have a separate test for their dependencies: for example, Board depends on Piece, so Piece is tested in isolation before Board. This means that if there is a failure in the smaller component (i. e. Piece), it will be easier to track down using its isolated test, rather than having to look at all of the dependent component (Board) only to find a failure in the smaller component.

5 Time Log

We started with implementing the TetrisPiece. Implementing the rotations took roughly two hours. Dual “parts” of the TetrisBoard needed two hours each: basic movements and piece
placement/line clearing. Wall-kicks required half an hour of work.

The Brain required the largest amount of time because not only did we have to implement it, but also “invent” what we were implementing. In other words, the developing it wasn’t a straightforward process — it was the result of experimentation and repeated tweaking, resulting in eight hours of work.

Writing tests didn’t take very long (one or two hours), but changing the actual code to pass the tests needed some additional time.

All code was produced through pair programming, with the exception of wall-kicks (Ojas) and the Brain (Kevin). However, the strategies used in this separately written code were discussed beforehand and afterwards, and all code was reviewed by both team members.

References
