Project 4 Writeup

Matthew Anderson, Ryan Zhou

Overview

After hearing the horror stories about Tetris from upperclassmen, one of our top goals was to make a decent Tetris bot using a genetic algorithm. We also aimed to improve on the aspects we lost points on in project 3, namely more comprehensive and robust code and testing, more commenting, and better discussion of edge cases.

Solution Description

The first part of the base assignment, *TetrisPiece*, was fairly simple both in concept and in implementation. We decided to do all calculations in the constructor, making the accessor methods trivial.

We made good use of the *Set* structure in *TetrisPiece*. Because of its non-duplicative nature and quick operations, we used it in our *.equals()* method, allowing it to check for inequality even if the points were provided in a different order.

In order to pre-calculate *width*, *height* and *skirt* we used a *HashMap* from *Integer* to *Integer* to store the minimum *y* value corresponding to each *x*. This allowed us to kill three birds with one stone, since we could find the largest *x* difference from the values in the *HashMap*.

We similarly generated the rotations in the constructor. At first, this caused an infinite recursion in the constructor, which led to the implementation of a second constructor specifically for creating rotations. Unfortunately, this meant that most of the code in the original was duplicated. While we tried to find a way to move the repeated code into a method, it wasn't easy and we decided to focus on finishing the rest of the project.

Rotations were a big hassle when doing this project. Coordinate-based system with it's minimum bounding grid does not lend itself well to the "standard" Tetris system, wherein each piece is rotated about a central point in a bounding square. To mimic this, we wrote a method, *generatePieceMatrix()*, that takes an array of points and returns the piece in a bounding square. After that, rotating pieces was much easier and also adhered to the standard set forth in the SRS guidelines.

As dictated by the handout, *TetrisBoard* was abstracted to be a board of booleans representing placed pieces. The bulk of *TetrisBoard* logic lies in the *move()* method and its associated helper functions. Similar to the *executeCritter* method in the last project, we decided to abstract the method into a bunch of switch cases and handle the movements that way. Some of the

In designing *TetrisBoard*, we made a few design decisions that impacted how we implemented the class. We first of all decided to have the piece position contained in the board rather than in the piece itself. This made the *testBoard()* method easier to implement since we didn't need to make a copy of *nextPiece*, however, when I was writing the genetic algorithm I wished that the pieces contained their own position.

We also decided to create *int[]* arrays representing the widths and heights of each row and column respectively. This made some of the default getter methods far easier to implement, and was also useful in other game logic.

The brain was "trained" (more on this in the reflection) using a genetic algorithm. The algorithm takes into account the number of rows cleared, the number of holes in the board, the overall height of the columns in the board, the "bumpiness" of the board, the maximum height, number of full cells, and finally the largest difference between two adjacent columns.

The genetic algorithm uses tournament selection to create children of the fittest parents. The crossover function is simply a weighted average of the two vectors, and each child has a 10% chance of mutating.

The brain is tuned in *JBrainTetrisTuner*, which essentially runs a gui-less tetris game on the population of weight vectors. Over a few generations, the improvements are noticeable, though perhaps not as good as they should be.

Reflection

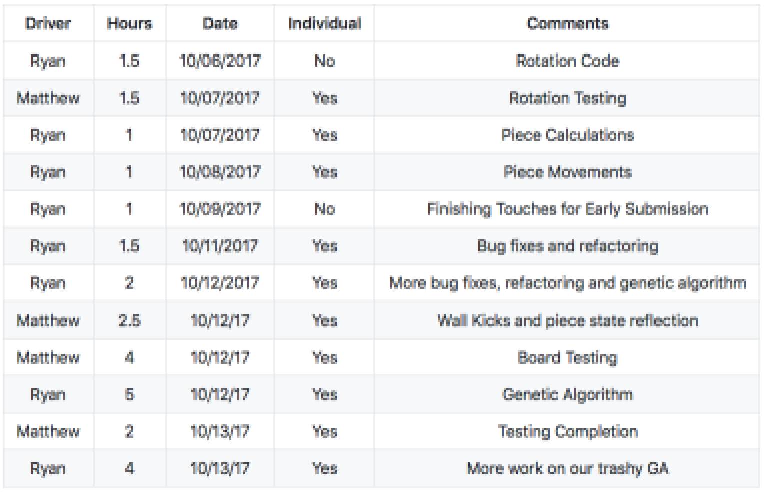
Perhaps our biggest assumption in this project is that the pieces would be given in a certain orientation every time. We decided to follow the guidelines set out by SRS on how piece rotations should happen, i.e. around a central point in a bounding square. In order to get this to work with the sanity tests, we had to do some fudgy math that will not work should the orientation change.

We also assume that there will only be 7 different pieces. Since we assign a category to each piece on creation (I, O and other), an addition of a new piece could cause some inconsistencies in our code.

On piazza, it was mentioned that the minimum board width was 3. However, this would not be able to accommodate a horizontally-oriented line. Thus, we assumed that the minimum board size would be a 4x4 square, the minimum size able to accommodate all default Tetris pieces.

We attempted the genetic algorithm for more of a challenge than anything else. We have very little experience with any form of machine learning, so early on we decided that if we were to attempt any of the karma, we would try the genetic algorithm first.

Due to being rushed on time, we did use some external sources for information. In particular, the descriptions here and here helped tremendously in implementing this algorithm. Funnily enough, though, this algorithm only performs slightly better and nowhere near what we thought it would be. One reason for this could be the lack of a look-ahead piece, which causes the AI to make decisions that are not necessarily the best in the long term. Also, due to the lack of a GUI, debugging the tuning process and making sure it is running correctly was very difficult. We spent a whole night fiddling with values, but there was no real improvement. This would definitely be interesting to look more into and hopefully implement correctly.



Driver	Hours	Date	Individual	Comments
Ryan	1.5	10/06/2017	No	Rotation Code
Matthew	1.5	10/07/2017	Yes	Rotation Testing
Ryan	1	10/07/2017	Yes	Piece Calculations
Ryan	1	10/08/2017	Yes	Piece Movements
Ryan	1	10/09/2017	No	Finishing Touches for Early Submission
Ryan	1.5	10/11/2017	Yes	Bug fixes and refactoring
Ryan	2	10/12/2017	Yes	More bug fixes, refactoring and genetic algorithm
Matthew	2.5	10/12/17	Yes	Wall Kicks and piece state reflection
Matthew	4	10/12/17	Yes	Board Testing
Ryan	5	10/12/17	Yes	Genetic Algorithm
Matthew	2	10/13/17	Yes	Testing Completion
Ryan	4	10/13/17	Yes	More work on our trashy GA

Table 1 Driving Hours

While working on the AI, we noticed some interesting quirks in our implementation of Tetris. For one, clearing rows sometimes leaves floating blocks. From some internet searching, it appears this can be standard behavior depending on the version of Tetris. We weren't sure what was wanted in this project so we decided to leave it as is. While messing around, we also set the weight of holes during a Tetris game to 100. This caused the AI to become “afraid” of holes, so it instead built up spires of blocks so as to not cover any open spaces.

We had even less time for pair programming this project. With the discrete midterms and in general a lot of other homework and other commitments, we struggled to find time to work together. However, even though we were working individually, we still worked together, talking about potential implementations, asking questions about confusion points and making a list of what needed to be accomplished.

Testing

Compared to the last project, this one had fewer user input test cases that we had to worry about, but it had a greater amount of possibilities in the code and variance of what needs to be done in the section. After we implemented the *TetrisPiece* class, we decided to test it immediately after to make sure that it functioned as expected so we would have a correct implementation going into the *TetrisBoard* implementation. With 7 types of blocks, and different positions and actions for each one, there were many scenarios for the state of each piece, and the behavior of the program overall.

To clarify testing, we moved all testing code into a separate package, and split it by both class and class functionality. Because of all the readable attributes of the classes, we didn’t need to create a test framework of the board as it all necessary information was already available to us. Using the provided sanity checks to ensure basic behavior was as expected, we wrote 47 tests for the functions provided in both the *Board* and *Piece* interfaces (including negative tests). Some scenarios were simple, having only one or less pieces on the board, while some were more complex to allow us to catch both simple and complex bugs. There were also tests for error checking on possible inputs that threw exceptions in regard to the arguments passed.

Edge cases were a big concern on this project due to the sheer number of possible outcomes from a single piece. As we wrote we tried to consider how certain pieces would behave in comparison to other pieces (specifically the square and “I” cases). The “I” block was especially tricky as it had a rotational bounding box of 4 instead of 3. This led to many edge scenarios with our standard rotation code, and required us to modify the *TetrisPiece* and some *TetrisBoard* functions to properly handle its differences.

Overall, we feel really good about our testing framework and the amount of cases it helped us catch. The number of bugs we might have missed would be well into the dozens if not for us writing test code to verify functions ran as expected after changes to either class.