Assignment #4b - Tetris

Overview and Goals

The purpose of this part of the assignment was to implement a few classes: TetrisBoard - the main logic of the Tetri program, JBetterBrainTetris - a class that extends JBrainTetris but runs with our new "Better Brain", and JAdversaryTetris - a program that extends JTetris and uses our "Better Brain" to increase difficulty. Also, we built new Brain Code, which we called "GeneticBrain". We incorporated this Brain Code into a genetic algorithm, which will be discussed in a later section. Our goals of this assignment were to build a competent Tetris AI, build a challenging Tetris game, and successfully port our failed Genetic Algorithm from the Critter assignment to Tetris.

Description

TetrisBoard

Our TetrisBoard class encompassed the main logic of Tetris. We keep the board, heights, and widths (and backups!) stored as instance variables. Our getMaxHeight() method loops through the array of heights and finds the highest column. Our sanityCheck() method checks the consistency of our widths, heights, and max height by cross-checking them with the actual board data. Our dropHeight() algorithm looped through the size of the piece and calculated the difference of the current height and the skirt of the piece above it. It returns the biggest number, or the y coordinate at which the piece would stop if dropped.

The place() method ensures that the board is in a committed state and backs up the board. Then, we check the bounds of where the piece is to be placed, and if it is in bounds and not conflicting with another piece, adds the piece to the board. Then, we loop through the height and width arrays and update them with their new state. If the variable DEBUG is set to true, sanityCheck() is called to ensure that nothing went awry. If a row is filled, PLACE_ROW_FILLED is returned, and otherwise PLACE_OK is returned.

The clearRows() method backs up the board if it is in a committed state. Then, it finds the first filled row and sets it to toRow. Then, fromRow is set to the next row above toRow. Then, the method checks every row on the board that is less than the height, skips filled rows, and copies rows down past the filled rows. We update the widths after every pass, and at the end of the algorithm update the heights to reflect the state after rows are cleared. We perform a sanity check if DEBUG is true, and return true if a row(s) was cleared, and otherwise false.

Our undo() method copies the backups to the current arrays, and resets the backups. Also, this sets the board into a committed state. We wrote a private backup() method that backups the current board, widths, and heights to backup containers. Also, we wrote a toString() method that was useful for debugging without the GUI.

JBetterBrainTetris

Our JBetterBrainTetris class was almost identical to JBrainTetris. We simply set the brain to our GeneticBrain in the constructor, and called JBetterBrainTetris() in the main method. These were the only differences.

JAdversaryTetris

Our JAdversaryTetris closely mimicked the JTetris class. We added a JSlider as an instance variable, and set our constructor to take a Brain as a parameter. We overrode the createControlPanel() method, and added our JSlider to the bottom of the control panel. We overrode pickNextPiece() to add some new functionality. We pick a random integer from 1 - 99. If the integer is above the slider value, then our brain picks the worst possible piece. Otherwise it simply picks a random piece.

Our Brain Code

Our GeneticBrain extends LameBrain. It reads a Brain from a Serializable object in it's default constructor. Our randomBrain() method returns a new GeneticBrain with coefficients all set to zero. (We talk about coefficients in our Karma Problems section). Our rateBoard() method calculates five important board attributes: holes - open grid squares that have closed blocks on top, wells - vertical groves in the grid blocks of height 2, max height - the maximum height of the board, average height - the average column height of the board, and bumpiness - the sum of the absolute values of the differences between heights of adjacent columns. A board's rating is a linear combination of these attributes, using evolved coefficients. We used some ideas from Flom and Robinson[1] in choosing these board attributes.

Interesting Results / Problems

Testing

Much of the testing was done for us in this assignment. We ran JBrainTetris sending it "test" as a parameter. Our board initially looked different than the picture given in the assignment. However, after three hours of debugging, we found the error to be in clearRows(), where we neglected to add 1 to getMaxHeight(). Doh!

In creating a Genetic Algorithm, much of the pragmatic testing was automated for us, as thousands and thousands of trials are computed in evolving Tetris Brains. We had one error (a definite corner case), in which clearRows() was attempting to clear the top row of the grid, and created an index out of bounds error. We fixed this code immediately. However, no other problems were found, and we have simulated a TON of games during evolutions (surely over a million possible moves checked). As far as other corner cases, we couldn't think of anything else.

Time Log

We split all programming time evenly, working strictly together, alternating driving every 10-30 minutes. We each logged 5 hours of driving for the first part of this assignment. Thomas started the Genetic Algorithm by himself, not thinking it would be successful. Surprisingly, it was. We tweaked it the following day.

Karma Problems

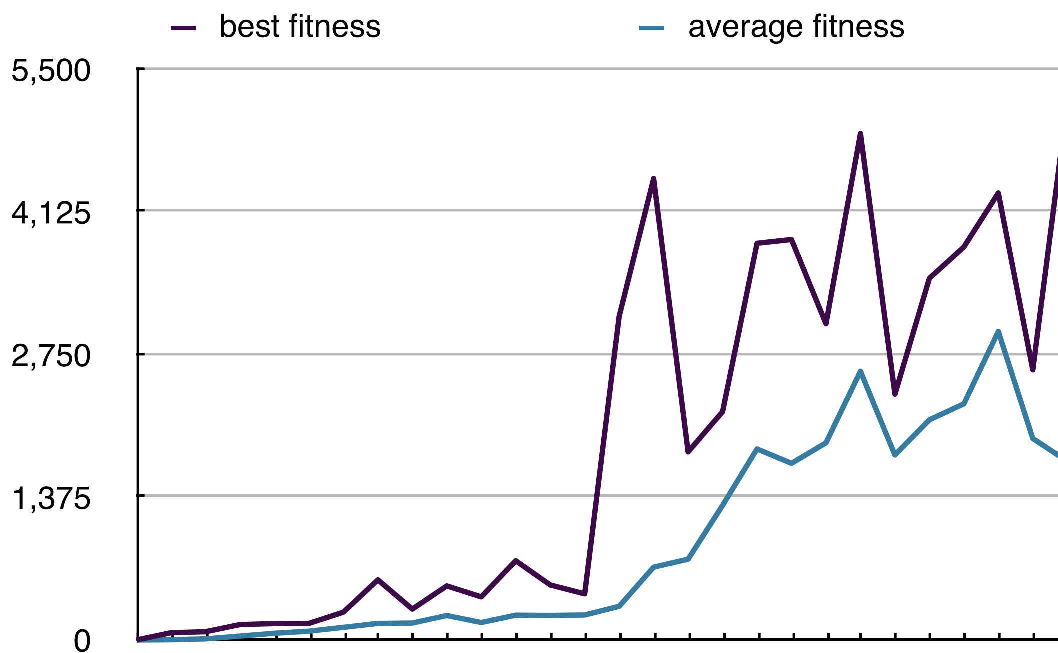
We attempted a genetic algorithm on our Critter assignment, but we had problems with genotype encoding. In this case, the genotype is incredibly simple, perhaps the ideal genotype - an array of doubles. We implement Serializable in our GeneticBrain, in order to save the GeneticBrains as small objects that can be loaded and saved.

Our mate() method in GeneticBrain crosses over two Genetic Brains using uniform crossover, in which 50% of the time doubles are swapped between two parents. Fifteen percent of offspring have one of their doubles mutated by a random Gaussian, in order to keep the species evolving.

[1] *http://www.cs.colostate.edu/~flom/TetrisPaperpdf.pdf*

Our BrainEvolver is our application entry point to kick off the evolutions. It creates a population of a specified size, and evaluates the population until the program is terminated. After every evolution, the most fit Genetic Brain is saved as a Serializable object in best.brain.

An inner class, Population, is the heart of the genetic algorithm. The evaluate() method uses our TetrisSim (a simple simulator without any graphical interface) to take the average number of pieces survived in four trials for each genetic brain. This is the “fitness” of the Genetic Brain. The results of this are used by the getNextPopulation() method to pick (as a weighted selection) two parents from the created brains. The offspring of these are added, and this is repeated until a complete new population is grown.



The graph above shows best fitnesses and average fitnesses for 27 generations. The best fitnesses suddenly peak after about a dozen generations, after which they begin to level out. A large amount of variation is to be expected due to the random nature of the Tetris game. Better solutions could possibly be obtained by combining different board attributes or by improving the genetic algorithm. Some GeneticBrains were lucky enough to score as high as 16,000 on individual trials, but, as of this printing, an average of a little less than 5,184 was the best our algorithm could evolve.