Evolutionary AI for Tetris

Elad Shahar
Department of Computer Science
University of Massachusetts Lowell
1 University Avenue, Lowell MA 01854
eshahar@cs.uml.edu

Ross West
Department of Computer Science
University of Massachusetts Lowell
1 University Avenue, Lowell MA 01854
rwest@cs.uml.edu

ABSTRACT
In this project, we examine an evolutionary approach to the video game Tetris using genetic algorithms to evolve a set of optimal weights for features used by feature detectors in a state space search of the Tetris game. We are able to set almost any arbitrary goal, along with any arbitrary restrictions or obstacles. The genetic algorithms then evolve the weights to maximize the fitness values associated with accomplishing the set goal, getting progressively and visibly better over time. However, very quickly the bot is ably to play well enough that the game lasts a significantly long amount of time, where a single evolution takes a greater amount of time than was provided for the entire project!

Author Keywords
Genetic algorithms, Tetris, video game automation, evolution, reproduction, Artificial Intelligence, JGAP.

INTRODUCTION
In the game Tetris, pieces named tetrominoes (made up of four unit squares arranged into predetermined shapes) fall from the top of the screen until they collide with something (the bottom of the board or a tetromino already on the board). By filling in a full horizontal row, the row is cleared and the player is awarded with points. If the pieces reach the top of the screen, the game ends. Tetris is an interesting game to play with because it combines both luck and skill (obviously required because there is a professional gaming league for Tetris players). Additionally, Tetris has been a common topic in artificial intelligence, and has been solved using other AI techniques such as reinforcement learning and state space search. Tetris has also been used as a test case for machine learning algorithms. Using genetic algorithms exploits the fact that we don't need a Tetris grand master to train a Tetris bot, and using an evolutionary technique allows "smart" solutions to develop naturally.

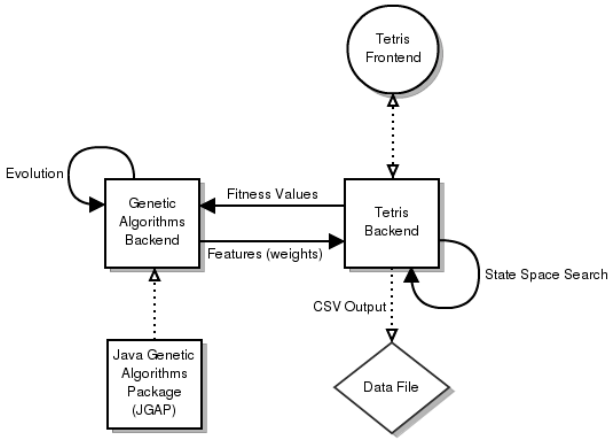

Figure 0. Technology Diagram

PROJECT DESCRIPTION
We built a computer player that chooses the best move by rating each possible next-state using an evaluation function. The evaluation function is a weighted sum of various features. A genetic algorithm finds optimal weights for the function. A genetic algorithm simulates adaptation, mutation, natural selection, reproduction, chromosome crossover, and evolution. In this situation, we have a few terms to define.

First, we define the make up of a population. At the lowest level, we have an allele, which is an integer value representing a weight. One level up we have the gene, which represents a specific weight, each of which is associated with an allele. This can be thought of as a key-value pairing, where the key is the gene and the value is the allele. A collection of genes, or weights, will be known as the chromosome. A collection of chromosomes that evolve amongst themselves is called a population. Finally, a population of chromosomes can also be called a genotype. Now that we know how a population looks, we must describe how it evolves.

A population evolves in one of three ways. First, each chromosome is evaluated by a fitness evaluator function, which assigns a fitness value to how fit a state of the population is (the specifics of this function is described further down). The fittest members of the population are then *duplicated* into the next generation. Some of the

population is then selected to be *mutated* (that is, we randomly alter a gene of one chromosome and forward it to the next generation). This mutation allows us to evade local maxima and head in a radically different direction that may end up being better. Finally, we chose another subset of the population and allow them to *crossover* in pairs. When two chromosomes crossover, we switch gene values between the two and pass them onto their offspring.

We then found a fitness evaluation function that associates a fitness value with a chromosome. In this case, the solution is quite obvious -- we simply provide the linear vector of weights for the chromosome to a Tetris-playing program that uses the weights to evaluate a state space and advance to the best next state based on an evaluation function based on a linear vector of weights. When the Tetris-playing program finishes a game of Tetris, it returns to us the score it was able to obtain that game, and we use that value as our fitness value. This value can also be seen as the phenotype of the chromosome. An important thing to note is that two chromosomes that are identical should get the same fitness value, and as such we must use the same seed for all the games played during a generation. However, we retain the randomness of the game in between generations, to ensure that the AI generated is good for any random game of Tetris, and not just one specific configuration.

Finally, we must choose our genes -- that is, chose what our linear weights would be. We plan on using the following feature detectors for our evaluation function:

- *Filled Spot Count* - The number of filled spots on the game board
- *Weighted Filled Spot Count* - Similar to the above, but spots in row *i* counts *i* times as much as blocks in row 1 (the bottom)
- *Maximum Altitude* - The height of the tallest column on the game board
- *Hole Count* - The number of unfilled spots that have at least one filled spot above them
- *Lines Cleared* - The number of full horizontal rows on the game board
- *Altitude Delta* - The difference in height between the tallest column and the shortest column
- *Deepest Hole* - The depth of the deepest hole (a width-1 hole with filled spots on both sides)
- *Sum of all Holes* - The total depth of all the holes on the game board
- *Horizontal Roughness* - The number of times a spot alternates between an empty and a filled status, going by rows
- *Vertical Roughness* - The number of times a spot alternates between an empty and a filled status, going by columns
- *Well Count* - The number of holes that are 3 or more blocks deep
- *Weighted Holes* - Same as the hole count, but rows are weighted by the row they're in
- *Highest Hole* - The height of the highest hole on the board
- *Game Status* - Based on the game status, 1 for a losing state, 0 otherwise

Most of these values were adopted from Colin Fahey (Fahey, 2003 [1]). Alternatively, rather than having the fitness value be the score of that game, we can tweak it be any arbitrary statistic such as lines cleared or tetrominoes placed. It is also important to note that an allele might end up being zero, effectively ignoring one of the factors.

Our data set is simply the standard Tetris game board, following the standard Tetris rule set. Scoring will be done as per the standard Tetris rules. The weights that we use when we start our generation 0 population will be randomly generated, and will evolve as the program runs. At each generation, the weights will be based on the previous generation, after applying reproductions, mutations, and crossovers. The only possible downside to following the standard rule set is that as the program gets better, it may take longer for it to complete just one game (previous Tetris AI bots have been able to clear millions of lines in one game!).

The goal of this project is to have an AI that, after a reasonable amount of time spent "learning," is able to play Tetris well. We will have a front-end that shows a game being played, and a back- end where evolution occurs. If observed, the initial AI will appear to move randomly and stupidly, but over time, it should learn how to play better and better. After each evolution, the program outputs the current state of the population to a CSV file.

ANALYSIS OF RESULTS

We found that while the early-stage success of the program depends greatly on the initial configuration, it always rapidly approaches the position of being able to out-perform a human player.

We decided to test the program in three separate and distinct tests.

1. **100 Line Limit** (Force game end after 100 lines cleared) - This was used to ensure that games did not last too long and a lot of evolution could occur. We found that due to the line limit, the bot learned to get larger line clears and combos (2, 3, or 4 line clears at once, rather than single line clears), which reward more points. Within 100 evolutions, the bot was clearing about 75% of the lines with a Tetris (4 lines at once).
2. **1000 Line Limit** (Force game end after 1000 lines cleared) - Similar to the last one, the bot learned to get multiple line clears and combos.

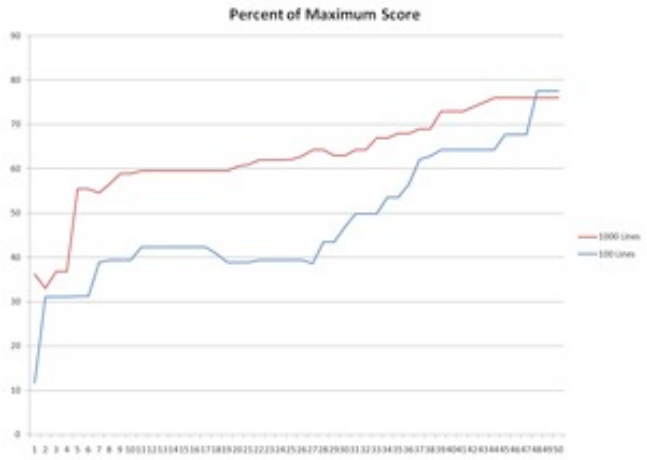

Figure 0. Results of 100-line and 100-line runs

3. **Unlimited** (No forced game end, other than hitting the top) - This one was arguably the most interesting. The bot tends to focus on keeping as much of the game area clean as it can, to minimize the chance of it losing the game. The problem with this is that the games started lasting an incredibly long time. While each move took at most 5ms, a single game would take as long as 3.5 hours on generation 2! Considering a population size of 50, this means an average population will take 175 hours to complete -- that's over a week's worth of time! Additionally, this would only get worse as the program evolved more and got better. While we would like to keep running it in unlimited and see how it does, doing so is outside the time scope of this project.

Each of these tests produced CSV output, as described in the description. For each stage of evolution, we calculated the score of the best individual, and the average score of all individuals. We then plotted these points on a graph to see how they changed over time. For the first two tests, the program rapidly approached the maximum possible score, and then fluctuated slightly on a near-flat line as mutations appeared. Unfortunately our data for the last test is significantly short, since we couldn't get past 3 evolutions in any reasonable amount of time.

DISCUSSION

Working on this program has opened up my eyes to the world of genetic algorithms. As ridiculous as it sounds, we want to apply genetic algorithms to solving all kinds of problems now -- even ones that genetic algorithms shouldn't be used with.

It is truly fascinating and interesting to create a situation where you define some feature extractors to grade a state, with no idea of what features is good, and have the program figure it out for you. Not only that, but watching the program run is mesmerizing. You get to watch the program getting better right in front of you! You can see the individuals that play badly, and those that play very well. Even those with no knowledge of artificial intelligence enjoyed watching the program run.

Using frameworks and libraries made by someone else also led to other unique learning experiences that come with using someone else's work (especially when you do so in ways it wasn't originally intended to be used).

All in all, genetic algorithms extended what we learned in class with material we didn't learn, and did so in a fun and engaging manner.

CONCLUSION

We believe that genetic algorithms are a viable approach to creating an artificially intelligent program that can play Tetris well. This statement can be extended to encompass any task where a state space search can be conducted and each state can be summarized as some value, where a larger value is better -- in Tetris, each state can be summarized as the score at that state, or the number of lines cleared to get to that state.

In the future, we would like to play with the unlimited game mode more, and see just how well our bot can actually play without restrictions.

Additionally, we'd like to extend our work to other games, and see how well genetic algorithms can be applied to them.

ACKNOWLEDGMENTS

The work described in this paper was conducted as part of a Fall 2010 Artificial Intelligence course, taught in the Computer Science department of the University of Massachusetts Lowell by Prof. Fred Martin. Elad Shahar did the work for implementing genetic algorithms (using the Java Genetic Algorithms Package (JGAP) [3]), while Ross West did the work for Tetris (based on Jetris [2]).

REFERENCES

1. Fahey, Colin P. "Tetris." *Colin Fahey*. 2003. Web. 19 Nov. 2010. <http://www.colinfahey.com/tetris/tetris_en.html>.
2. Georgiev, Nikolay G. "Jetris - A Java Based Tetris Clone." *SourceForge.net*. 28 Sept. 2006. Web. 18 Dec. 2010. <http://sourceforge.net/projects/jetris/>.
3. Meffert, Klaus, and Neil Rotstan. "Java Genetic Algorithms Package." JGAP: Java Genetic Algorithms Package. 2002. Web. 18 Dec. 2010. <http://jgap.sourceforge.net/>.