Using a Genetic Algorithm to Weight an Evaluation Function for Tetris

Landon Flom and Cliff Robinson

Colorado State University, Department of Computer Science

Fort Collins, CO 80526
Loveland, CO 80538
flom@cs.colostate.edu
robinsoc@cs.colostate.edu

Abstract

Tetris is a popular video-game invented by Alexey Pajitnov. An agent that plays Tetris must be able to place pieces in good positions without knowledge of what pieces will follow. One way such an agent can work is to use an evaluation function to place the current piece. This evaluation function is a weighted sum of features from the board. We used a genetic algorithm, based on Genitor, to discover these weights. We tried several things to improve the efficiency of the search for the weights, including different fitness evaluations and crossover operations.

Introduction

Tetris was invented by mathematician Alexey Pajitnov in the mid-1980s (Breukelaar et al 2004). For this paper we used the standard rules for Tetris as specified by Colin Fahey (Fahey 2003). Tetris is a puzzle game in which the player must arrange falling blocks on a board so that they create 'lines', which are then removed. Play ends if the pile of blocks reaches the top of the board. The board is a 10 by 20 grid of squares, each of which can be either filled or not. The blocks consist of seven different pieces (named I, O, L, J, S, Z, T, after their shapes), each of which is made up of four squares. These pieces can be rotated and moved around the board horizontally as they fall, but cannot be stopped or moved upwards. A line made by filling an entire row of squares. This line is then removed and all of the filled spaces above this line are moved down, preserving their current configuration. It is possible to make several lines at once. Traditionally there is a time element in Tetris, where the block must be placed as it falls, and as the game progresses the blocks falling speed increases. They appear in random order, and the player can either know what the next piece coming up is, or not.

In this paper we discus the development of computer agents that play Tetris based upon strategies outlined by Colin Fahey (Fahey 2003). These agents use an evaluation function, based on features from the current board state, to decide where to place the current piece. This evaluation is a weighted linear sum of the features, and we used a Genitor-based genetic algorithm to determine these weights. We performed experiments to determine the performance of various crossover operations in the genetic algorithm. We tried experiments with different fitness evaluations, in an attempt to make the agents learn faster. Finally, we discus methods for making agents play well in general Tetris games, as opposed to learning a specific game.

Related Work

There has been some work done on the mathematical nature of Tetris. It has been determined that playing an optimal game of Tetris, with prior knowledge of the pieces, is NP-Complete (Breukelaar et al 2004). Heidi Burgiel proved that there exists at least one sequence of pieces that will force the game to end even if the agent is playing optimally (Burgiel 1997). John Brzutowski proved that there exists no winning strategy (playing deterministically) for playing Tetris (Brzutowski 1992). He proves that any game of Tetris that includes the S and Z pieces will eventually end. Tetris has been modeled as a Markov decision process that uses a compact feature based representation (Tsitsiklis and Van Roy 1994).

Genitor was developed at Colorado State University by Darrell Whitley as a "steady state" genetic algorithm (Whitley 1993).

For our experiments, all of our pseudorandom numbers were generated using Sean Luke's fast Java implementation of Makoto Matsumoto and Takuji Nishimura's Mersenne Twister algorithm, MersenneTwisterFast.java (Luke 2004). We chose to use this because it is faster than the standard Java pseudorandom number generator (Luke 2004), and the period of Mersenne Twister is 2^{19937}-1 (Matsumoto and Nishimura 1998), which should be sufficiently large to eliminate bias due to repeated random number sequences.

Our Tetris Simulator

Our Tetris simulator deviates from standard Tetris rules in that the real-time aspect is eliminated. We reasoned that our Tetris agents can evaluate and place a piece fast enough that the real-time aspect would be nullified, and it takes much less time to evaluate agents if they do not have to wait for the next piece. Our agents are also not required to move the current piece to any given column; instead they can specify which column and orientation they would like to land the piece in. Once a piece has landed, it is not allowed to be moved again. This means that pieces cannot "slide" underneath overhanging pieces the way they can in many Tetris implementations. Our Tetris game is not scored using the standard rules; instead we simply give one point per line made.

Our Tetris Agents

Our agents only know which piece is currently being played. To decide how to play it, they use an internal Tetris simulator to speculate on the best position and rotation. The agent plays the piece in every legal position and rotation on the internal simulator. For every play, the next board state is calculated, and an evaluation function is used to determine the utility of the new state. The agent then plays the piece in the actual game in the position and rotation of highest utility. This is the same way that Colin Fahey's agents work.

The Evaluation Function

The function that the agent uses to determine the utility of a board's state is the weighted linear sum of numerical features calculated from the state. Colin Fahey's agents used these features: pile-height, the number of closed holes, and the number of wells (Fahey 2003). The features that we added are the number of lines that were just made, and a number that represents how "bumpy" the pile is.

The pile-height is just the height of the tallest column. This is important because the game ends if the pile-height reaches twenty (Fahey 2003).

Closed holes are unfilled spaces that have filled spaces covering them. They are important because lines can only be made on a row with closed holes when the rows above the holes have been removed. Closed holes increase the difficulty of reducing the pile-height (Fahey 2003).

Wells are deep, narrow spaces that only the I-piece can fit into without making closed holes. These are important features because there is only a one-in-seven chance of getting an I-piece (Fahey 2003).

The number of lines made is important because making lines is the only way to reduce the pile-height. It is also the only way to get to closed holes, and it is also the way we score Tetris.

Bumpiness is a measure of how flat or bumpy the top of the pile is. It is calculated by taking the sum of the differences between the heights of adjacent columns. Bumpiness can be bad because it means there is more than one row on top of the pile that has unfilled spaces. It *can* make it difficult to place pieces without making closed holes. On the other hand, S and Z pieces cannot be played on a perfectly flat surface *without* making closed holes.

The features are weighted because there is epistasis between them. For example, removing lines reduces pile-height and can remove closed holes, but can also open up wells and can increase bumpiness. We expected the agents to reward making lines, and to penalize just about everything else, but we did not know which features would be more important. That is why we used a genetic algorithm to search for the optimal weights.

Using Genitor

The genetic algorithm we used to find the weights was heavily based on the steady-state genetic algorithm Genitor (Whitley 1993). We used a binary representation of the weights for the chromosomes. Every generation two parents were selected completely at random from the population. These were then recombined to make one child. A copy of this child was then mutated by flipping each bit with some probability. Then both the mutated and original children were evaluated and added to the population. After this, the two worst members of the population were removed.

Chromosomes were evaluated by having an agent play Tetris, with the evaluation function's weights determined by the chromosomes. Chromosomes were interpreted as an array of 32-bit IEEE floating point numbers. In our experiments all agents played the same sequence of pieces for every evaluation. This way better agents who got bad piece sequences would not be discarded in favor of worse agents who got good piece sequences.

Fitness Determination

An agent's fitness was determined by the rank of the utility that the agent returned, as in Genitor (Whitley 1993). The utility was calculated in two different ways. First we used the number of lines the agent was able to make before losing or the number of pieces they were allowed ran out. Second we used the ratio of lines per pieces played. We did this because the agents were only allowed to play for a limited number of pieces, and we thought that one utility might be a better representation of how good the agent actually was.

We chose to use the number of lines the agent made because making lines is the goal of Tetris, and is the only way to continue playing.

We used the ratio of lines made per pieces played because an optimal agent should be able to make one line for every 2.5 pieces (Fahey 2003). We also thought it wouldn't be as affected by the limited number of pieces, because it should be less likely that a sub-optimal ratio agent would survive than a sub-optimal lines-made agent. However, this turned out to not be the case; in fact there was no noticeable difference in leaning rate, as shown in Figure 1.

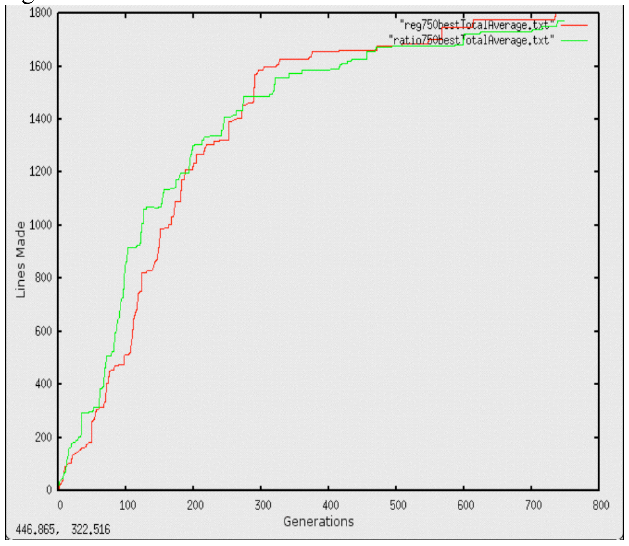

Figure 1

These are the graphs of the number of lines made by the best agent in each generation, averaged over thirty different runs. The red line represents the lines-made agents, and the green line represents the ratio agents. Since we could see no difference, we chose to use the lines-made agents in the rest of our experiments.

Crossover Operations

We thought that using different crossover operations would have an effect on the rate at which agents improved. We tried four different crossover operations: uniform, one-point, HUX, and reduced surrogate one-point (Whitley 1993).

We expected one-point to be the worst of the four, and uniform and HUX to be the best. We thought this because one-point is likely to breed into convergence using the Genitor model. This is because it is possible to reproduce one of the parents, and if this parent is good enough to stay in the population the child will also stay, meaning there will be multiple copies of this genome in the population. When this happens the likelihood of selecting that genome increases in subsequent generations and the genome will start to dominate the population sending it into convergence. On the other hand, it is impossible for HUX to reproduce one of the parents, and it is unlikely that uniform will reproduce one of the parents unless they are very similar.

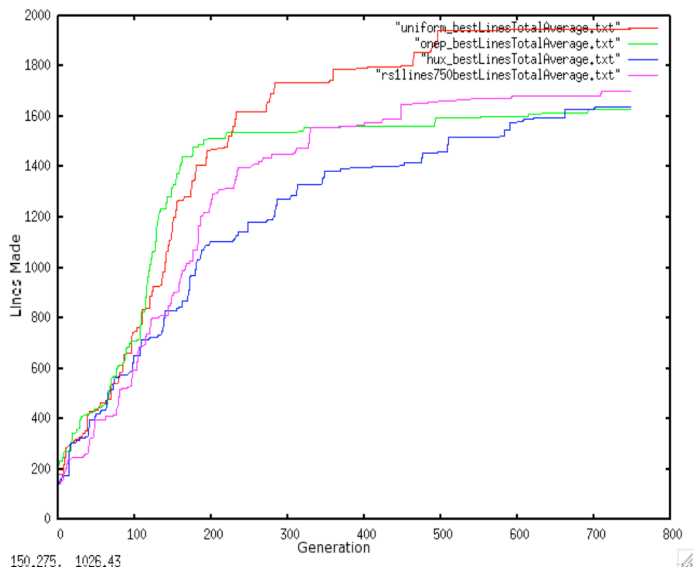

Figure 2

To test these operations, we ran them each for 750 generations thirty times and took the average for each crossover type at each generation. We then graphed them, using both lines made (Figure 2) and pieces-per-line ratio (Figure 3).

It should be noted that the two graphs are from two separate experiments. The first graph (Figure 2) makes it appear that they are all about the same, except for uniform, which appeared to slightly better. In the second graph (Figure 3), it appears that they are all about the same,

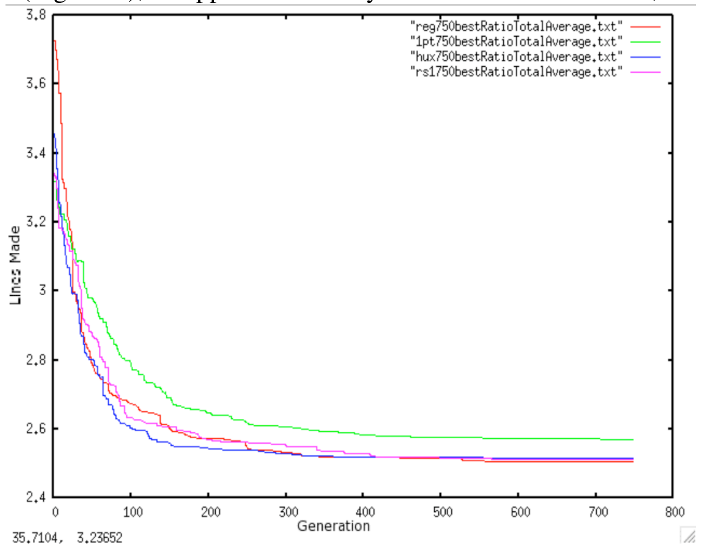

Figure 3

except for one-point, which appears to do worse. However, because we have not done any statistical analysis (confidence intervals, hypothesis testing), we cannot conclude that they are in fact different. Since it didn't seem to be any worse, we decided to use uniform crossover for the rest of our experiments.

Playing Well in General

For all of our previous tests we had each member of the population play the same game (piece sequence) as everyone else. We chose to do this because it seemed like a more fair way to compare agents. However, it is possible that the agents we evolved were only able to play that particular game well. Another thing to consider is that we limited the number of pieces an agent could play in the interest of generating data quickly. This means that our algorithm cannot tell the difference between an agent that can play for two-thousand pieces and an agent who can play for two-million pieces. Whether the agents we evolved could play well in general (meaning any given sequence of pieces) is unknown. We thought about this and came up with ideas for agents who might be able to evolve better.

First we thought that we should make populations that were evaluated with a new random game every generation. Fitness would be determined by the average number of lines made per game. This has the advantages of being able to compare the agents fairly against each other, and also exposes them to a bigger variety of situations while still limiting play length, thus putting an upper bound on the time it takes to evaluate a generation. The disadvantage is that it takes much longer to evaluate every generation, and time was a concern.

Another thing we tried was allowing the agents to play for an unlimited number of pieces. The thought behind this was that a sufficiently long game exposed the agents to the same number of situations as playing multiple short

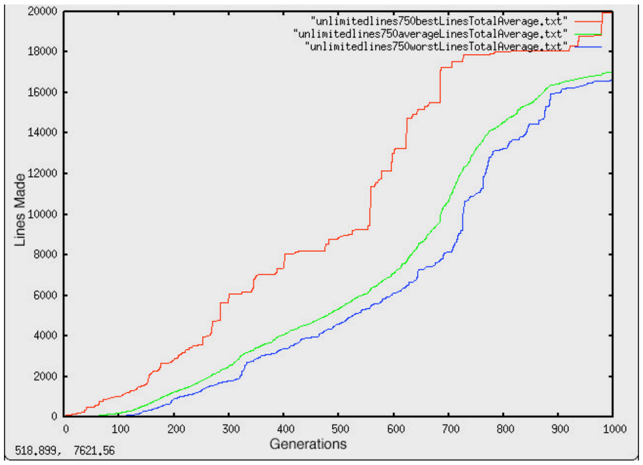

Figure 4

games. Again, we have the advantage of being able to compare the agents fairly, and only having to evaluate two agents each new generation. However, because the agents were allowed to play indefinitely, as the agents started to improve, it took a very long time to evaluate successive agents. Since we have not built a satisfactory testing program to test the agents made by these algorithms, it is unknown which approach is actually better for evolving general-case agents.

Another thought we had was to have a limited number of pieces to play, but have each new agent play a new random game without re-evaluating the rest of the population. This should be just as fast as our current experiments, but should explore more situations. This has the disadvantage possibly having a good agent receive a pathological set of pieces and thus losing to a bad agent who got a good set of pieces. However, over a long enough time we think that these situations should be amortized. Unfortunately, we were not able to test this approach.

Results

We ran some agents for an unlimited amount of pieces to see how well they would learn. What we found is that they appear to continue learning each generation (Figure 4). The red line represents the number of lines the best agent in the population was able to make, the green is the average number of lines made for the population, and the blue represents the worst agent in the population at that generation. This graph represents data taken from 20 separate runs averaged together. It appears that they hadn't yet bred into convergence. We don't know if this meant that they were actually getting better in general, or if it was just getting really good at that particular game, but it looks encouraging.

We examined the genomes produced by some of the best unlimited agents. See Table 1 for some examples of these genomes. We were hoping to see some

Table 1

Lines Made	**Pile Height**	**Bumpiness**	**Closed Holes**	**Wells**
4.75091E33	-1872253.9	-2.4589E33	-1.48574E34	-2.9534E34
-3.442E-38	-2.7495E-10	-5.5145E25	-3.1095E26	-6.5143E19
9.5182E14	8.627E-32	-1.3699E21	-3.4514E22	-1.9210E36

consistent relationship between the feature weights. We found that there was a fairly consistent ranking in value of the weights. Meaning, in general lines made was the most rewarded, followed by pile height, bumpiness, closed-holes, and wells, in that order. However, the relationship is more complex than this. When we entered genomes that followed the same ranking, the results were not as good. In fact, our genomes often made no lines at all, and reducing the precision of the floating-point numbers had a large negative impact on how they were able to perform. This was unexpected because some of these weights were vastly larger than the rest. For example, reducing the precision down to two significant digits on a genome that could make 100,000 lines resulted in an agent that could only make 10, 000 lines.

From our experiments with the populations that played new limited random games every generation, we saw that they were able to evolve agents that reached the same piece-per-line ratio on random games as agents who played the same limited game repeatedly (see Figure 5). The red line represents the best member from the population that evaluated each generation with a new random game, averaged over five separate runs of 250 generations each.

Conclusions

We discovered that genetic algorithms seem to work very well for searching for good weights for the evaluation function used to allow an agent to play Tetris. This is what we expected because this approach also worked well for Colin Fahey (Fahey 2003).

We concluded that there is a non-obvious relationship between the weights of the features. We speculate that this is because there are strategic tradeoffs between making different plays. For example, it doesn't matter if you increase the pile height by one if you are likely to make a line on your next turn. Since our agent is able to play effectively without knowing what future pieces will come up, we think that the relationships are somehow expressed in the weights that are evolved. Meaning, the agent gains knowledge about how to play current pieces that will help create lines in the future.

At this time, we cannot conclude with confidence that there is a difference in performance between the different crossover operators, but it appears that one-point crossover might not work as well as the other crossovers we tried.

Future Work

One of the main things that we would like to do is more extensive testing to see how well our various agents do in general games. This testing could consist of running the best populations from different approaches for multiple trials on random unlimited games. This should give us a better idea as to which approach, if any, is superior.

Another thing we considered, and would like to test, is that the relations between the weights may not be linear. We think this might be the case because of the sensitivity of the weights, and the fact that the weights are evolved to be very far apart. One way to do this would be to make the evaluation a polynomial function rather than a linear summation. The genome would consist of coefficients and exponents for each feature. It may also be possible to try other types of mathematical functions, but the genomes may be more difficult to represent.

Another thing we would like to try is to use different genetic algorithm strategies and see if they work any better than Genitor in this domain. For example, adding an island migration model to Genitor, or using either the canonical genetic algorithm or CHC (Whitley 1993). We also could try parallelizing these approaches in order for them to run faster.

Finally, we would like to have a graphical user interface so that we could actually watch our agents play Tetris after they have learned. This was actually in our original plans

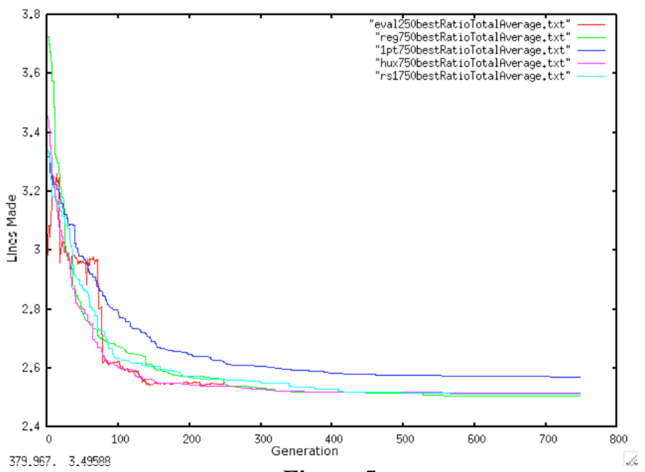

Figure 5

for the project, but was never implemented due to time constraints. This could make it easier to understand what strategies the agent had discovered. This is difficult to infer from examining the genomes.

References

Breukelaar, R. et al 2004. Tetris is Hard, Even to Approximate. *International Journal of Computational Geometry & Applications.*

Brzustowski, J. 1992. Can you win at Tetris? MS thesis, Department of Mathematics, University of British Columbia.

Burgiel, H. 1997. How to lose at Tetris. *Mathematical Gazette*, July.

Fahey, C. 2003. Tetris AI, http://www.colinfahey.com/2003jan_tetris/2003jan_tetris.htm

Luke, S. 2004, Evolutionary Computation, http://www.cs.umd.edu/users/seanl/gp/

Matsumoto, M. and Nishimura, T. 1998, Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator, *ACM Transactions on Modeling and Computer Simulations: Special Issue on Uniform Random Number Generation.*

Whitley, D. 1993, A Genetic Algorithm Tutorial, Technical Report, CS-93-103, Department of Computer Science, Colorado State University.