Playing Tetris with Genetic Algorithms

Jason Lewis

***Abstract*—The classic video game Tetris can be represented as an optimization problem that can be maximized to produce an efficient player. In this problem, a player picks moves by generating future game states and computing a weighted sum of features for each state. The player picks the move for the state with the largest weighted sum. The values for these weights are optimized through a genetic algorithm. After running the genetic algorithm for 30 generations using a feature set of size 10, one of the best resulting players achieved an average game length of 179,531 moves over 50 trials.**

***Index Terms*—Genetic Algorithm, Machine Learning, Tetris.**

I. Introduction

Tetris is the classic falling blocks video game invented by Russian programmer Alexey Pajitnov in 1984. This immensely popular and deceptively simple game is well suited to being played by a computer player. An individual state of Tetris is easy to represent as: the current game score, the type of Tetris piece (or *tetromino*) will be dropped next, and a 20 by 10 binary grid representing the currently occupied spaces in the Tetris board. From any possible state in Tetris, there is a finite number of possible moves for a human or computer player to consider. Figure 1 below shows the seven tetromino types along with the size of the move set for each tetromino in a constrained version of Tetris that will be described later.

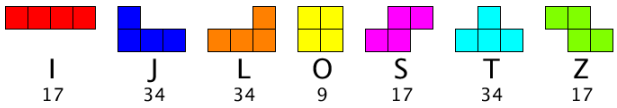

Fig. 1. *The move set sizes of the seven Tetris tetromino types*

If Tetris is represented as an optimization problem, a variety of optimization methods can be applied to generate computer players that play Tetris efficiently. One such method is a *genetic algorithm*, which is an approach to optimization that simulates the process of evolution. This paper describes a framework for applying a genetic algorithm to an optimization problem for Tetris and discusses the results achieved from running the algorithm.

All code used to produce the results in this paper, including the Tetris implementation, the Tetris visualizer, and the genetic algorithm framework, was all written in Java specifically for this project. The Tetris visualizer uses the Java-based graphical environment Processing.

II. Related Works

The concept of applying a genetic algorithm to Tetris has been explored by previous publications. All previous works I have found follow a similar approach of using a feature set to evaluate feature vectors for future Tetris states and using genetic algorithms to find the best weights on these features. The three previous works I found each has some minor differences. Landom Flom and Cliff Robinson (2004) represented their weights as bit vectors where each generation can result in random bits being flipped. Niko Böhm, Gabriella Kókai, and Stefan Mandl (2005) experimented with different rating functions, including a linear function that is just the dot product of the weights and features and an exponential function that takes some exponential of each feature value. David Rollinson and Glenn Wagner (2010) used Nelder-Mead optimization on the best weight vectors from the genetic algorithm to get better results.

In all three of these works, the success of a Tetris player was measured by how many lines are cleared before losing the game, as opposed to how many points are scored by clearing more lines at a time. Since the number of lines cleared is roughly proportional to the number of moves, this metric measures how resilient the player is to losing, but does not measure how well the player can clear multiple lines at a time.

There have also been some previous works related to the computability of good moves in Tetris. Breukelaar, R. et al. (2004) have shown in *Tetris is Hard, Even to Approximate* that in the offline version of Tetris, where the player knows the entire finite tetromino sequence in advance, maximizing the number of rows cleared or pieces placed before losing is NP-hard.

John Brzustowski (1992) analyzes different variations of Tetris to determine if it is possible to "win" at Tetris through some strategy that is guaranteed to continue playing indefinitely. He shows that some variations of Tetris are winnable, such as a game that only produces I and O tetrominos. However, the standard version of Tetris that includes all seven possible tetromino types is not winnable. Specifically, any version of Tetris that produces both the S and Z tetrominos is not winnable and will eventually end.

Keeping in mind that Tetris is not winnable and the offline version of Tetris is incredibly hard to optimize, a Tetris player playing an online version of Tetris (where only the next move's tetromino is known) can only hope to keep the game going as long as possible while consistently making reasonable moves.

III. Genetic Algorithms

Before explaining the Tetris optimization problem in detail, here is a brief summary of genetic algorithms.

Like other optimization methods, a genetic algorithm attempt to find inputs from an input space that maximizes the

output of some function. In a genetic algorithm, this function is referred to as the *fitness function* which is used to evaluate the *fitness* of a *candidate* input. The algorithm maintains a *population* of N candidates, where N typically ranges from hundreds to thousands.

At the beginning of each iteration of the algorithm, the current population represents a *generation* of candidates which is used to generate a new population of N candidates for the next generation. Each generation is numbered according to how many iterations of the algorithm have been run.

Generation 0 is initialized with N random inputs from the input space. A new generation is created by probabilistically selecting candidates from the current generation. The probability of a candidate being selected is proportional to its fitness evaluated by the fitness function.

After all candidates are evaluated, the candidate selection takes place in three phases, which each contributes a certain percentage of candidates to the next generation:

- *Selection* – Candidates are selected and added to the next generation unchanged
- *Mutation* – Candidates are selected and each candidate's input is slightly altered in some way before being added to the next generation
- *Crossover* – Candidate pairs are selected and then each pair of corresponding inputs are combined in some way to form a new candidate to be added to the next generation

IV. Tetris Implementation

The implementation of Tetris used for this problem uses a board of size 20 by 10 and allows the player to see the type of tetromino that will be used in the next turn as well as the current turn. For simplicity, a player's *move* only consists of the rotation of the tetromino and the column on the board from which to drop the tetromino, disallowing any additional movements that are normally allowed in Tetris. This results in the move set sizes seen in Figure 1.

In this implementation, there is no timed element where the tetromino slowly falls down the board, as it is assumed that a computer player will pick moves very quickly. This assumption does not consider the possibility of instant gravity of tetrominos possible in some versions of Tetris, which can make some moves impossible.

If at least one line is cleared after a move, the game score increases by a certain amount: 2 points are awarded for 1 line, 5 points for 2 lines, 15 points for 3 lines, and 60 points for 4 lines (also known as a *tetris*). This scoring system is proportionally equivalent to the one used in the Nintendo Entertainment System version of Tetris.

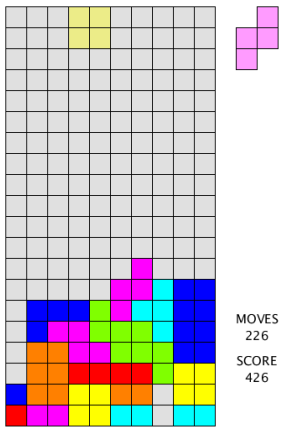

Fig. 2. *Example visualization of a Tetris State.*

V. Tetris Optimization Problem

In the Tetris optimization problem used in this paper, a candidate Tetris player is characterized by a fixed *weight vector* in $\mathbb{R}^M$, where M is the number of features used to characterize a state of Tetris. Given the current state of the Tetris game which contains the next two tetrominos to use, the player generates every possible future state of Tetris after the next two turns. For each future state, a *feature vector* is computed. A particular state is scored as the dot product of the state's feature vector and the player's weight vector. The move that is chosen is the one that produces the largest possible dot product two moves from now.

The abilities of a Tetris player can be judged two factors:

- Making a large number of moves without losing
- Scoring points as efficiently as possible by clearing multiple lines at a time, preferably by making tetrises

Both of these factors are captured by the fitness function, which is the *average number of points per move* that a player earns during a game of Tetris. To prevent games for continuing exceedingly long, the game can be stopped after a specific maximum number of moves. If the player gets a game over before reaching that many moves, the average score is still computed as the number of points earned divided by the maximum number of moves. This penalizes players that lose early by a factor of how early the player loses.

Another metric used to understand the abilities of a player is the player's *score efficiency*, which is the percentage of the maximum possible score the player earns. A player that only clears lines through tetrises will earn 60 points every 10 moves, making 6 the highest possible average score per move. So, a player's efficiency is measured as its average score per move divided by 6 and represented as a percentage.

The tetromino sequence generated for a game is controlled by a random number generator that when given an initial random seed will always generate the same sequence. To prevent overfitting on a particular sequence of tetrominos, the

random seed is different for each generation. All players within a generation are playing with the same tetromino sequence so that the candidate selection process is fair.

VI. Feature Set

The feature set used in the Tetris optimization problem is possibly the most significant factor in generating a player that reliably plays long games with high efficiency. The previous works mentioned earlier provided many ideas for possible features. I combined some of these ideas from previous works along with some of my own ideas into a list of 23 potential features. Not all of these features are beneficial to the feature set, so forward search was used to select only the features that are shown to increase the potential efficiency of a player when added to the feature set.

The problem was slightly altered for the forward search to prevent overfitting on the weight vectors and tetromino sequences. The values for each weight are constrained to the range [-1, 1]. Each potential feature set was evaluated over 10 Tetris games with different random seeds. To speed up the computation, the maximum number of moves per game was limited to 200 and the players only consider Tetris states one move in advance instead of two moves. Starting with the best weights found in the previous iteration of forward search, the addition of a new feature involves finding the best weight for the new feature from a fixed set of values, then finding a local optimum in the neighborhood of the resulting weight vector.

WeightedBlocks	0.359										
ConnectedHoles	0.028	1.129									
LinesCleared	0.002	0.359	1.746								
Roughness	0.005	0.359	1.323	2.458							
Tetrises	0.001	0.359	1.129	1.722	3.009						
PitHolePercent	0.012	0.359	1.633	1.863	2.458	3.167					
ClearableLine	0.001	0.527	1.714	1.621	2.458	3.030	3.253				
DeepestWell	0.002	0.369	1.412	1.876	2.211	3.009	3.167	3.536			
Blocks	0.002	0.359	1.507	1.768	2.458	3.009	3.063	3.138	3.558		
ColHoles	0.028	0.666	1.129	1.907	2.475	3.009	3.219	3.27	3.540	3.628	

Concavity	0.001	0.359	1.129	1.746	2.458	3.009	3.207	3.037	3.224	3.558	3.628
ColumnVariance	0.027	0.416	1.163	1.772	1.902	3.009	3.202	3.432	3.536	3.32	3.483	3.628
HeightDiff	0.012	0.481	1.226	1.746	2.286	3.009	3.009	3.087	3.536	3.558	3.483	3.628
HeightSum	0.183	0.939	0.951	1.746	2.38	3.009	3.140	3.487	3.552	3.552	3.628	3.628
Holes	0.042	0.933	1.129	1.746	2.458	3.009	3.167	3.263	3.487	3.558	3.628	3.628
LowestClearableLine	0.001	0.579	1.273	1.746	2.352	3.009	2.994	3.360	3.518	3.391	3.628	3.628
MaxHeight	0.102	0.359	1.129	1.474	1.954	3.009	3.188	2.990	3.432	3.558	3.628	3.628
One	0.001	0.359	1.129	1.746	2.458	3.009	3.167	3.253	3.536	3.558	3.628	3.628
Pit	0.002	0.359	1.293	1.899	2.146	3.058	3.058	3.145	3.187	3.558	3.628	3.628
Score	0.002	0.359	1.153	0.959	2.897	2.897	2.853	3.487	3.536	3.503	2.789	3.628
SideStackQuality	0.003	0.359	1.129	1.898	2.458	3.009	3.243	3.466	3.432	3.558	3.628	3.628
WeightedBlocksLog	0.297	0.297	1.218	2.001	2.015	3.009	3.167	2.849	3.067	3.382	3.628	3.628
Wells	0.023	0.684	1.129	1.791	1.935	2.322	2.815	3.466	2.683	3.506	3.628	3.628

Fig. 3. *Results of forward search on features*

Each column in the graph of Figure 3 shows the results of one iteration of forward search, in which the best performing feature is added to the feature set. The cells within a column are colored closer to red the farther the value is from the top cell in the column.

After the 10 top features were found, no additional features were able to increase the metric, so the final feature set used is those 10 features. The last 11 iterations of forward search are not shown in Figure 3; every value in those iterations are 3.628.

To explain these features, the terms I use are defined as:

- *block* – a filled space on the board
- *hole* – an unfilled space that is located below a filled space
- *pit* – an unfilled space that is not a hole and has a filled space (or the board edge) on its left and right
- *column height* – the row where a column's highest block is located

The M features are as follows:

- *WeightedBlocks* – weighted sum of blocks, where a block's weight is the row it's on
- *ConnectedHoles* – number of vertically connected holes
- *LinesCleared* – number of lines currently cleared during the game
- *Roughness* – sum of absolute difference between adjacent column heights
- *Tetrises* – number of tetrises currently made during the game
- *PitHolePercent* – number of pits / (number of pits + number of holes)
- *ClearableLine* – maximum number of lines potentially clearable by a single I (straight) piece
- *DeepestWell* – the row containing the lowest unfilled block that is not a hole
- *Blocks* – number of blocks on the board
- *ColHoles* – number of columns containing at least one hole

VII. Algorithm Results

After computing the optimal feature set with size M=10, the genetic algorithm was run for 30 generations with a population of N=1000. For candidate selection, the Selection, Mutation, and Crossover phases each contributes 20%, 40%, and 40% of the new candidates respectively.

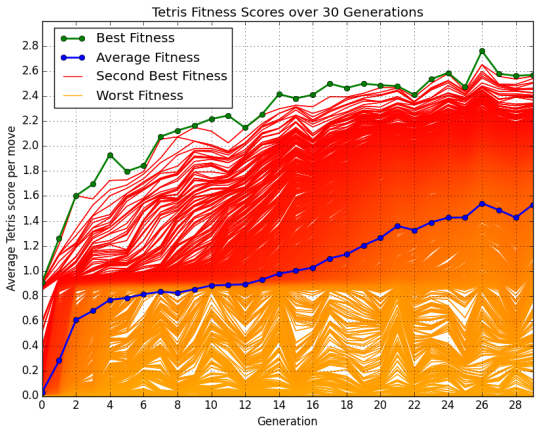

Fig. 4. *Fitness of candidates over 30 generations*

The graph in Figure 4 summarizes the fitness of the population over all 30 generations, focusing on the best fitness and average fitness achieved by each generation. The fitness scores for the rest of the candidates are shown from red (second best fitness) to bright orange (worst fitness) to visualize the overall trend that the population generally improves over each generation.

Based on the results from Figure 4, the algorithm has mostly converged after 30 generations, resulting in players that can usually achieve a minimum average score of 2.4 per move (equivalent to a minimum efficiency of 40%). It is possible that running more generations might result in a slight increase in the performance of the best player, but these results are enough to demonstrate the effectiveness of the algorithm.

VIII. Additional Testing

The best player of generation 28 was selected for additional evaluation. 50 trials were run in which the player plays a full game of Tetris without a maximum move limit until it reaches a game over. The number of moves and the fitness score was recorded for each trial. The results are shown in the scatter plot in Figure 5. A summary of statistics from Figure 5 are shown in Figure 6.

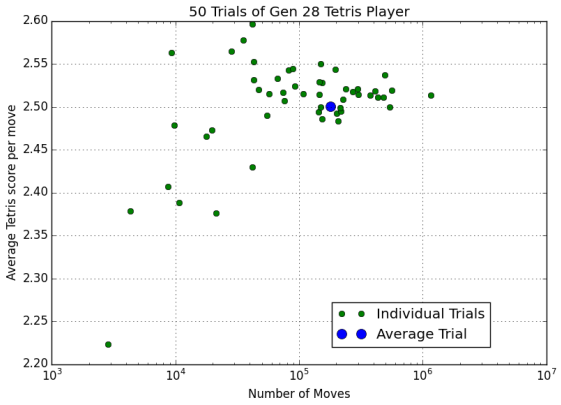

Fig. 5. *Scatter plot of 50 trials of gen 28 Tetris player*

	NUM_MOVES	AVG_SCORE
Average	179,531	2.5006
Median	126,032	2.5143
Max	1,162,156	2.5962
Min	2,832	2.2232
StdDev	208798.3143	0.0607

Fig. 6. *Summary statistics from Figure 5 plot*

While not listed above, it is interesting to note that during these trials, the average move takes under 1.3 milliseconds to compute. Over the 50 trials, the player achieved an average fitness of 2.5 points per move (an efficiency of 41.67%) and an average game length of 179,531 moves (under 3.9 minutes in real time). A human player making one move per second would take over two days to reach the same game length.

The best players from generations 1, 4, 12, and 28 were visualized side-by-side in the Tetris visualizer playing games with the same sequence. In a series of unrecorded trials, it was observed subjectively that a higher generation player almost always gets a better score efficiency than a lower generation player.

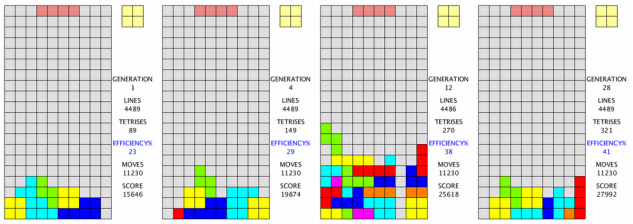

Fig. 7. *Visualization of players from four generations playing Tetris*

The screenshot in Figure 7 shows an example of the Tetris visualization of the four players. A short video of the visualization can be accessed at the following link: https://vimeo.com/148293176.

References

[1] Böhm, N., Kókai, G., & Mandl, S. (2005). An Evolutionary Approach to Tetris. MIC2005: The Sixth Metaheuristics International Conference, Vienna.

[2] Breukelaar, R. et al (2004). Tetris is Hard, Even to Approximate. International Journal of Computational Geometry & Applications.

[3] Brzustowski, J. (1992) Can you win at Tetris? Department of Mathematics, University of British Columbia.

[4] Flom, L., Robinson, C. (2004) Using a Genetic Algorithm to Weight an Evaluation Function for Tetris. Colorado State University.

[5] Rollinson, D., Wagner, G. (2010) Tetris AI Generation Using Nelder-Mead and Genetic Algorithms.