Tetris AI Generation Using Nelder-Mead and Genetic Algorithms

David Rollinson and Glenn Wagner

February 15, 2010

Abstract

In this paper, we discuss the training of a tetris playing AI using a combination of Nelder-Mead optimization and genetic algorithms. The core AI consisted of a variable depth look-ahead player optimizing a set of 16 features, built on the core Java code provided by the TA. A set of random scores were optimized using MATLAB's implementation of Nelder-Mead, fed through a genetic algorithm, then passed through Nelder-Mead again. When run at depth one look-ahead (testing the placement of two pieces), the resulting program averaged 160,000 lines cleared, and cleared over 430,000 lines in the best of 16 trials.

1 AI Description

1.1 AI Design

Tetris offers significant problems for adaptive control. Specifically, the state space, while discrete, is huge, and both the dynamics and reward function are highly nonlinear and discontinuous. To avoid working with the 2^{210} states that are possible when treating the entire playing area, we focused on a vector of 16 features, described in the next subsection. The mapping from the playing area to the feature vector is decidedly not one-to-one, so attempting to map feature vectors directly to moves would not be feasible. Instead, we implemented receding horizon control. A depth zero search consisted of checking computing a score based on the feature vector after each possible move was tested. The score was a reward function computed by taking the dot product of the feature vector with a weight vector. Moves that resulted in losing the game were penalized by an additional loss cost. For deeper searches, instead of scoring the current board, all possible next pieces, and all possible moves for each next piece, were tested, and the expected score was then computed. Therefore, a depth zero search considered only the current piece. A depth one search considered the current piece, and all possible successor pieces. A depth two search considered placing the current piece and all possible next two successor pieces.

The AI was based on the java framework provided by the TA. A multi-threaded for performance evaluation was tested, but found to be slower than the single threaded variant.

1.2 Description of Features

In order to incorporate domain knowledge about overall tetris strategy, a set of features were created with which to score the game at each time step. The score of feature was summed into the final score with a multiplicative coefficient that would be varied through our optimization process. 16 individual features were used in the final version of the scoring function, although only 12 of these features are truly unique. This is because we chose to use both the score and squared score of features that seemed intuitive to penalize exponentially (like the number of holes). Having separate weights for linear and quadratic scores allowed the optimization to rely on whichever score produced better game scores. A brief description of the features follows. Features that included a linear and quadratic score are indicated as (squared).

Rows cleared - The number rows removed during a turn.

Holes (squared) - The total number open squares in the playing field that have at least one closed square somewhere above. For example a hole that is 2 x 2 squares in size would be counted as four holes, not one.

Maximum hole height (squared) - The highest row that contains at least one hole.

Maximum column height (squared) - The height of the tallest single column.

Columns with holes - The number of columns that contain at least one hole.

Rows with holes - The number of rows that contain at least one hole.

Total column height - The sum of the heights of all the columns. This serves essentially the same role as an average column height.

Lowest playable row - The lowest row that could theoretically be cleared this turn. This accounts for low column heights that are negated by having holes in other columns that are higher up in the playing field.

Roughness - The sum of the absolute value of the difference between each column.

Maximum pit depth (squared) - Depth of the deepest single-width pit.

Slope - The tendency of the tops of the columns to be taller on one side of the field than the other. This was calculated by subtracting adjacent column heights from left to right, and then taking the absolute value of

that score. While this fails to distinguish a positive vs. negative slope, it was important for features to have scores that were either always positive or negative, so that the weights for each score would have a consistent effect.

Concavity / Convexity - The tendency of the tops of the columns to be different in the center than on the sides the field. This was calculated in a similar fashion as slope, by iterating symmetrically from the center of the playing field. For the same reasons as slope, this value only represents a deviation from level with respect to concavity or convexity.

2 Training Techniques

The purpose of training was to find the weight vector and loss cost that resulted in a scoring function that optimized player performance.

From experimentation, we found that random seeds produced better results in Nelder-Mead than in the genetic algorithm. Therefore, we fed the results from the first pass of Nelder-Mead into the genetic algorithm, then fed the top ten set of weights from the genetic algorithm back into Nelder-Mead for further optimization. The theoretical justification for this is that Nelder-Mead is a local optimization approach, and as such is likely to get stuck at local optima. Genetic algorithms are very good at combining the best pieces of different samples, which would hopefully mix the most successful pieces of the Nelder-Mead sample. However, genetic algorithms are not as good at fine tuning of parameters, justifying the final pass with Nelder-Mead.

2.1 Nelder-Mead

To perform the Nelder-Mead optimization, the MATLAB function `fminsearch` was used. Initially the value on which the algorithm iterated was an average of 4 games. However, during the second round of optimization (following the genetic algorithm optimization) this method produced scores with extremely high variance, making reliable optimization difficult. For this reason the function was changed to use an average of 16 games. It should also be noted that during the initial round of Nelder-Mead optimization, the cost of losing a game was not optimized, and was set at a static value of -20000. During the second round, this loss cost became another parameter to be varied in optimization, because it had been parametrized in the genetic algorithm.

2.2 Genetic Algorithm

The genetic algorithms was rather simple. It took a list of candidate weights, typically 700 samples, and ran one game with each of them to find the number of rows cleared, called the fitness. The next generation was populated by choosing two parents randomly, with probability proportional to their fitness. Each element of the new weight vector was chosen randomly from one of the

two parents. There was a small (1%) chance that the weight would be chosen completely at random, a mutation. This process was repeated until a new set of sample weights was selected.

For the weights that were ultimately used, the genetic algorithm was seeded with the results of a Nelder-Mead optimization. For each weight vector from Nelder-Mead, three copies were made for use by the genetic algorithm. The first was left unaltered. The second was subject to a random multiplicative perturbation of each weight, of up to 20%, as well as a random chance of swapping sign. The third was processed with a 10% mutation probability for each weight value. All three copies were kept for the first generation, but the second generation, generated from randomly selected parents, had only the normal number of samples, resulting in a three-to-one reduction in number of samples. This was done because the majority of the perturbed weights would be pretty lousy, so rejecting the worst would result in useful reductions in computation time. For the final weights, the genetic algorithm was seeded from Nelder-Mead with perturbations, and run for 60 generations. The algorithm then used the resulting population to reseed itself, with perturbations, and was run for another 140 generations. This resulted in a total of approximately 140000 games used to optimize weights.

3 Results

Parameters	Depth	Average score	Max score
Fully Optimized	0	22500	70900
	1	163000	434000
	2	90000*	90000*
Nelder and Genetic	0	51000	N/A
*still running at time of writing			