AC/RL HW 1: Tetris

Alberto Rodriguez, Matthew Tesch, Laura Lindzey

1 Introduction

1.1 Problem Statement

The goal of this project is to design a controller to play a simple game of tetris. This controller needs to decide where to place each piece, aiming to maximize its score—in this case, the number of lines completed.

Although the statement is succinct, there are a number of challenges that make this an intruiging problem. For example, each game drops a random series of pieces. Small variations in this series can lead to a large difference in score, even while using the same strategy. The reader has likely experienced this during a game of tetris; over time it is common to develop a deep empty slot that needs to be filled in. There is only one piece (the 1x4 'rod') that can fit in this slot, and the game can spiral out of control if that piece does not drop quickly.

Other factors that contribute to the complexity of this challenge include the large number of possible states of the game board (exponential with respect to board size), and the fact that in every game even the best strategy will eventually lose with probability 1 (see Burgiel [2]). In fact, designing an optimal strategy has been shown by Demaine et. al. [3] to be an NP-complete problem.

1.2 Existing Work

Not surprisingly, Tetris is a popular problem domain for AI research. While some of the most successful algorithms published rely on human intuition to design and tweak evaluation functions[4], a wealth of machine learning techniques have been applied to playing Tetris.

Siegel and Chaffee [7] implemented a evolutionary algorithm to generate successful heuristic/weight combinations. It resulted in average scores of 15-30 lines. Bohn et.al. [1] also

tried an evolutionary approach, with significantly better results. After months of computation, they achieved scores into the millions of lines. Other approaches previously tried include randomized constraint sampling, which Farias used to average 4000 lines [5], and more general reinforcement learning, which Gros et. al.[6] used to improve hand-tuned weights by a factor of 4, ultimately averaging 8000 lines/game.

While the literature is weak on gradient and simplex machine learning techniques being successfully applied to Tetris, word of mouth among our classmates suggests that it is a viable approach, with some impressive results.

2 Our Approach

When first considering the problem, we realized that there was no possibility of learning to play from the ground up – there were simply too many possibilities: the board has 2^{200}, or 1.6×10^{60} states, and a strategy is needed for each of 7 possible dropped pieces. Even by only considering the top few rows, it was not feasible to learn a controller policy over all possible states. Thus, we needed to simplify the information that the controller used to make a decision.

For any piece that drops there are approximately 40 possible combinations of position and rotation. When choosing which move (position/rotation pair) to select, the controller evaluated the state of the board that would result from each move and chooses to execute the one corresponding to the lowest cost board. We decided upon a set of heuristics that would describe the state of the board, such that $cost = w_1H_1 + w_2H_2 + ... + w_nH_n$, where H_i is the i^{th} heuristic and w_i its associated weight. Effectively, this reduced the number of possible state variables from 200 (one for each cell) to a single variable for each heuristic. However, the controller may no longer be able to play optimally, as the diversity of its actions is bounded by the choice of heuristics. In a loose sense, we are searching for orthogonal heuristics that span the state space of the board. Practically, we chose heuristics that encoded human intuition about how to play Tetris.

To change (and hopefully improve) the performance of the controller, it is only necessary to modify the weights which balance the various heuristics. Therefore, after developing a set of heuristics, we can find the best weights using standard learning techniques, grid searches, or even hand-tuning parameters. Unfortunately, due to the random variation in piece order, each set of weights must be tested on many games to yield meaningful results.

2.1 Implementation

After establishing the high-level approach, we still lacked an actual implementation. Prior to actually coding it ourselves, we searched for an easy to use, well documented, existing implementation compatible with our approach, but could not find one that matched these constraints. Therefore, we wrote two versions; the first in MATLAB for ease of development and testing, and then in C for quickly running longer and more computationally intensive trials. The same basic structure was used in both implementations.

As discussed above, each set of weights was run over several games (typically around 50), and the scores averaged to obtain a single score given those weights. This reduced random variation caused by different piece sequences.

2.1.1 Single Game

A single game can be divided into a series of *steps*. For each *step*, the controller is given the current state of the board, as well as the next piece, and is expected to return a valid column and rotation of that piece, which defines where to "drop" that piece. The piece is then dropped at the requested position, and added to the state of the board. For each row completed as a result of this action, it is removed, each row above it drops down by one line, and the score is incremented by one.

This process repeats until the highest column is above the 20 piece limit, at which time the game ends, and the score, or total number of lines eliminated, is returned.

We focus here on the details of the controller choosing a proper column and rotation during each step. The state s is given by $\{s_{1,1}, s_{1,2}, \ldots, s_{20,10}\}$, where $s_{i,j} \in \{0, 1\}$ represents whether the grid element at the ith row and jth column is filled. The next piece p is randomly chosen from $\{1, 2, \ldots, 7\}$, which map to the seven possible pieces. The output is expected to be a column $c \in \{1, 2, \ldots, 10\}$ and a rotation $r \in \{1, 2, 3, 4\}$. Note that only valid combinations of these values are allowed; most pieces cannot be placed in the 10th column, for example.

A simulator then drops every combination of c and r, and obtains a set of future states $S_f = \{s_{c,r} \; \forall \; c, r\}$, where each $s_{c,r}$ is the state generated by dropping the suggested piece at column c with rotation r.

Denote the set of heuristics by H, where each heuristic $h \in H$ is a map $h(s) : S \to \mathbb{R}$ from states to a real-numbered costs. Note that each heuristic does not dictate where to put a specific piece, but rather gives a cost for any placement of the piece, therefore effectively ranking all possible placements.

Each $s_{c,r} \in S_f$ is then assigned a cost, which is the weighted sum of the cost from each

heuristic, or more explicitly $C(s_{c,r}) = \sum_i w_i h_i(s_{c,r})$, where w is the vector of weights for the current game. The controller then simply chooses c and r to be the values that give the minimum $C(s_{c,r})$, out of all valid choices.

2.1.2 Adjusting Weights over Multiple Games

For a given set of weights, we could then generate the average score over hundreds of games. However, initial guesses of weights only generated a few lines on average. Therefore, we needed to modify the weights to improve the performance.

The methods we used to find better weights are fairly straightforward. Our first approach was using a simplex search to learn better weights. This has the benefit of being a common algorithm for high-dimensional nonlinear optimization, but suffers from several drawbacks:

- Poor performance when using a cost function that doesn't always evaluate to the same value
- Needs tweaking to set initial simplex size
- Requires many iterations, which may be too slow to be useful
- Can converge to very bad weights

Another approach was a grid-based search of the parameter space, which also has intense computation requirements, but can reveal very good results. Finally, the option of hand-tuning parameters allowed us to use our own intuition to balance the weights in a way so as to correct problems we saw occuring during previous trials, which improved performance and computational time at the expense of requiring more human intuition and guidance.

3 Heuristics

Clearly, the success of our approach is dependent on our choice of heuristics. We followed the convention of trying to minimize cost with all weights positive, so larger heuristic values indicate less favorable states. Our final set of heuristics was:

- H_1 Difference between maximum and minimum column heights
- H_2 Variation of column heights

- H_3 Number of holes, where a hole is defined to be an open square with at least one filled square above it. Note that it doesn't need to be fully enclosed, as this version of tetris does not allow horizontal sliding of pieces mid-drop.
- H_4 Inverse of density
- H_5 Negative of rows completed on the previous move
- H_6 Sum of squares of the depth of holes
- H_7 Sum of squares of height differences between neighbouring columns
- H_8 Bias towards the side walls. The heuristic averages the height of all columns, weighted by its distance to the closest wall.
- H_9 Maximum height.
- H_{10} Average height.
- H_{11} Number of jumps in the height of consecutive columns.

One further refinement that we made was adding a lookahead step. That is, rather than only minimizing the board cost after the current piece drops, choose the position and orientation that minimize $cost = cost(currentpiece) + \alpha E[cost(nextpiece)]$. Since the next piece is not known and the distribution of pieces is uniform, the expectation of next piece's cost is an unweighted average of the lowest possible board costs that result from dropping each piece.

This lookahead measure was computationally intensive (slowed down state evaluation by a factor of 280), so we were unable to use it in any of the learning approaches we tried. Similarly, the α parameter could be tuned in theory, but we lacked the time and computational power to do so: we arbitrarily chose to set $\alpha = 1$.

4 Roadblocks

After we developed our MATLAB simulator and added the initial set of heuristics, we discovered a problem. Our simulator required several minutes to run the hundreds of trials needed to obtain a reliable average score over multiple games. This meant that each iteration of the simplex search was too time-consuming when running many trials per iteration. If we ran fewer trials and allowed the variance of the average score to increase, the simplex search didn't converge.

As a result, we ported the program to C, which increased the speed, but not sufficiently to allow simulating enough games to use standard learning approaches. At this point, we brainstormed possible solutions to overcome this speed issue:

- **Supercomputer cluster!** While this would probably provide enough computation, it isn't exactly realistic or readily available to us.
- **Pre-generate games:** If we randomly generate about 50 piece sequences, each several hundred pieces long, then we reduce the major source of variance. This would allow us to average over fewer trials for each set of weights, while still obtaining reasonable results. In fact, the cost function would then be deterministic.

  Unfortunately, this method has several drawbacks. First, unless infinite-length piece sequences are generated, the games may end by running out of pieces. Also, there is no guarantee that the optimal weights generated for this test set will generalize well to all possible piece sequences. The latter is a common problem in machine learning, and it could be addressed in this case by running more trials. (Which is exactly what we're trying to avoid, due to the poor speed of our simulator)
- **Smaller game board:** Reducing the game board from 20 lines to 10, as suggested in Bohn [1], would result in shorter games and improve the speed of evaluation. However, the exact nature of the relationship between scores in 10 and 20 line games has not been proven.
- **Adjust simplex search parameters:** By setting better initial weights and initial simplex size, the simplex search can perform more effectively. Unfortunately, it is difficult, if not impossible, to evaluate the effectiveness of a particular parameter set without running the simplex search with those parameters, as well as several others to compare. As the simplex search already takes a long time to run, this isn't a feasible solution until we solve the speed problem via another method. Therefore, this becomes an improvement of convergence suggestion, rather than a speed improvement suggestion.
- **Hand-Tune parameters:** We can harness the power of human intuition by watching games and hand-tuning the weights on heuristics *because we have knowledge of what the heuristic is supposed to do.* Although this requires hands-on attention, human intuition can quickly improve the weights to a reasonable level. These can then be used as the final optimized weights or as a starting value for a learning approach.
- **Choose better heuristics:** By improving the quality of the heuristic, or how well it can judge the desirability of a certain state of the board, the controller can drastically improve. Unfortunately, many times better heuristics require more computation, also slowing down the speed of the controller.

A smaller game board allowed us to obtain results from the simplex search in more reasonable amounts of time. Unfortunately, we were unable to adequately configure the parameters in matlab's built-in simplex search. This led to it searching a too-small area of the state space, and even converging on severely suboptimal weights. The simplex search

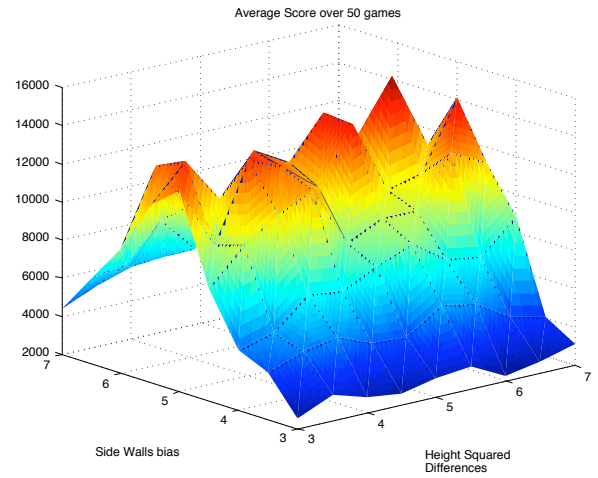

Figure 1: Score function with variation of weights of heuristics H_7 and H_8.

was also sensitive to initial conditions. We hypothesize that the difficulties may result in part from the large variance in the cost function, and the fact that it contains a significant high frequency component, as shown in Figure 1.

The last two of the proposed solutions proved most useful. We embarked on a literature search for inspiration, which resulted in heuristics H_6 and H_7 [6], both of which proved highly informative. However, this alone was not a sufficient improvement. We used hand-tuning of weights to find a starting condition for any automated searches: after less than 30 minutes of work, we obtained results significantly better than those resulting from a simplex search on the same heuristics.

5 Optimization Algorithms

We formulate the optimization problem as selecting weights for the hand-designed heuristics that will maximize the number of completed rows, when used as described in section 2.1.1. We have tried various approaches to solve this optimization problem.

5.1 Nelder-Mead/Simplex Search

First, we tried to learn the weights using a gradient-free algorithm with arbitrary initial conditions. We averaged scores over 100 games, and used this as input to Matlab's

fminsearch, a built-in implementation of Nelder-Mead. This learning algorithm explores the parameter space by repeatedly evaluating the score at each vertex of a simplex, then reflecting the lowest-scoring vertex through the hyperplane defined by the others. The method improved the results obtained (by a factor of 5), but didn't get the improvememnt that we were expecting. There are several features of the search space that might contribute to this poor performance:

- The score function we are trying to optimize contains numerous local extrema.
- The variances on the average scores are on the same order of magnitude as the scores themselves. This will make it almost impossible to distinguish between noise and true improvement.

Next, we tried using Nelder-Mead again, but this time giving it *reasonable* weights generated by manual search as the initial condition. Again, the simplex search returned unsatisfactory results; the score after the optimization was almost the same as before.

5.2 Human Intuition

Given our failure to produce significant performances using machine learning alone, we opted to experiment with tuning the weights by human intuition. This worked surprisingly well, averaging 2000 lines/game after about half an hour of work. We suspect that this is due to the human ability to reason about *why* a particular combination of weights is working well or poorly, and thus allows weights to be changed logically based on the flaws in the previous game.

5.3 Local Search

In order to improve our human-generated weights, we decided to run a local grid-based search overnight. This would have been infeasible over the whole space, but it worked to find a local maximum resulting in an average of 17000 lines/game.

6 Results

In the final implementation of the tetris player, we decided to use only heuristics H_3, H_5, H_7 and H_8 from section 3. Note that the behaviour based on each of these heuristics independently is shown in the movie submissions. After several trials we concluded that the introduction of more heuristics requires finely tuning all the weights to avoid harming the performance. Therefore, we opted to focus on a small subset of the proposed heuristics. We believe that the heuristics we chose are in some sense independent. This means that

they do not interfere destructively, which would explain why a restricted set makes it easier to find a *reasonable* initial guess for the weights.

The final weights obtained from the optimization process are:

H_3 92.5 Number of holes.

H_5 40.0 Negated number of deleted lines.

H_7 5.0 Sum of squares of height differences between neighboring columns

H_8 5.0 Bias towards the side walls.

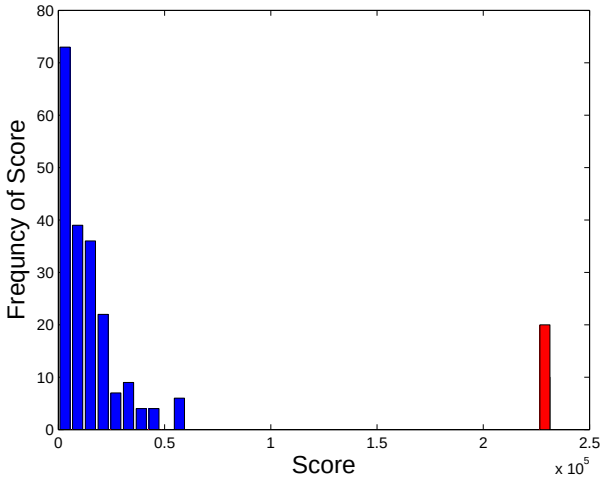

Figure 2: Using optimized weights, a single lookahead trial (indicated by the red bar) performs several standard distributions above the mean of trials without lookahead (distribution given by the blue bars).

These weights were obtained from the following process:

- First, we optimizated using a simplex search from trivial initial weights [1.0 1.0 1.0 1.0]. This resulted in no significant improvement from the initial results, and an average a score of **30** lines.
- In a second stage we hand tuned the weights of the entire set of heuristics. We reached the weights [10.0 5.0 40.0 5.0 30.0 10.0 10.0 10.0] and averaged approximately **200** lines. This low score is caused by the difficulty of hand tuning a large set of weights.
- In order to address the high dimensionality of the search space, we restricted the heuristics to those listed above and, through a hand-tuning process, obtained the

weights [100.0 50.0 5.0 10.0]. This boosted the average number of lines per game to **1800**.

- Using a grid-based searches in a local neighborhood of the hand selected weights, we found the local optimum, of [92.5 40.0 5.0 5.0], as listed above. This increased the average score per game to **17000** lines.
- Finally, we decided to test the lookahead strategy using the same weights that performed well in the greedy solution. This algorithm was much more time consuming, and we were not able to run more than a couple of simulations. However, given that this algorithm scored over **229000** lines (16.5 standard deviations about the average score without lookahead), we can conclude that lookahead provides a significant improvement (see Figure 2).

The attached movies demonstrate the performance of the hand-tuned set of weights, as well as the set that was then further optimized by a local search.

It is also interesting to note that it is not enough to implement lookahead, without a good set of weights and heuristics. To drive this point home, we ran several games using a non-optimal set of weights, specifically [40.0 85.0 1.0 2.5], both using the lookahead technique and without it. Although the average score improved significantly from 314 to 598 when lookahead was added, the score is still much worse than using better weights without lookahead.

The results obtained from the local search corroborated what we suspected about the high frequency noise in the score function. Figure 1 shows an example of the evolution of the score function in a neighborhood of the optimal weights, for two of the weight dimensions searched.

We also provide a comparison of a random game played using human tuned weights as well as the optimal weights according to the local search. As seen in figure 3, the maximum height of the board is slightly higher, on average, when using the human tuned weights rather than the further optimized weights; however, the results are still quantitatively similar; both controllers are still solid after 1000 dropped pieces.

Overall, these results demonstrate that although this is a difficult problem, we were able to make significant progress. Furthermore, due to the performance limitations of our code, we were not able to achieve successful results using standard learning techniques, as the cost function proved to noisy and uninformative. Finally, the importance of human intuition in determining heuristics and tuning weights cannot be overstated: although straitforward techniques such as look-ahead can vastly improve performance, without a solid set of weights and heuristics, the performance will be very poor under any circumstances.

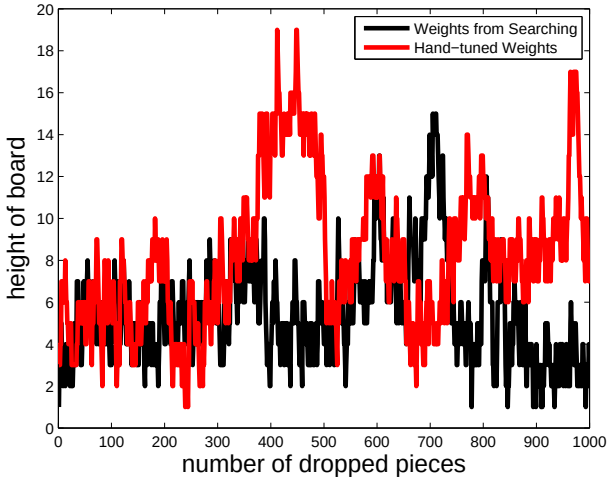

Figure 3: The height of the board after each piece is dropped for human tuned weights and optimum weights from the local search.

References

[1] Bohm, Niko, Gabriella Kokai and Stefan Mandl, "An Evolutionary Approach to Tetris", The Sixth Metaheuristics International Conference, 2005.

[2] Heidi Burgiel, "How to Lose at Tetris", Mathematical Gazette, July 1997.

[3] Demaine, Erik D., Susan Hohenberger, and David Liben-Nowell, "Tetris is Hard, Even to Approximate" Tech Report, 2003.

[4] Colin P. Fahey. "Tetris" 2009. `http://www.colinfahey.com/tetris/tetris_en.html`

[5] Farias, Vivek F., and Benjamin Van Roy , "Tetris: A Study of Randomized Constraint Sampling", `web.mit.edu/~vivekf/www/papers/tetrischapter.pdf`

[6] Gros, Alexander, Jan Friedland and Friedhelm Schwenker, "Learning to Play Tetris Applying Reinforcement Learning Methods", `www.dice.ucl.ac.be/Proceedings/esann/esannpdf/es2008-118.pdf`

[7] Siegel, Eric V., Alexander D. Chaffee, "Genetically Optimizing the Speed of Programs Evolved to Play Tetris", Advances in Genetic Programming: Volume 2.