On the Playing of Tetris

Eric Whitman, Breelyn Kane, and Donald J. Burnette

***Abstract*—In this paper we propose a method for playing tetris. We were able to demonstrate that a sophisticated heuristic combined with a naive learner can give significant performance over just using a learner with a simplistic heuristic. Combining our heuristic with a one move lookahead allowed us to achieve over 4 million lines in a single game.**

I. Introduction

On our quest to create a great tetris player, we weighed various options. Our initial approach was to use reinforcement learning. The idea was to create a learner that is rewarded, over multiple games, based solely on the final score. After many testing iterations, we were unable to achieve more than a couple lines as our max score. Our second approach was to create a heuristic based controller to evaluate the cost of every possible move, and ensure the best possible piece placement at each turn. Our cost evaluation is described in detail below. The cost function is built on different features useful for evaluating the game of tetris. We saw instant results, achieving an average of 2,000 lines over multiple games. We expanded on our cost function to also include: one and two piece lookahead, and an optimization of the parameters. Currently, the tetris player generates an average of 1.2 million lines per game.

II. The Tetris Framework

All of our testing was done on an in-house simulator created in java. The design of our tetris framework includes a maintained state, and a player class. The state represents an abstraction of the tetris board. The board is indexed by row and column. Some things represented by the state are: all possible moves of each piece, the current filled cells of the board, and the top most row of each column. The player class receives state information to determine the next best move. The player class is responsible for evaluating the cost of a move and taking into account any lookahead. It decides where the next piece should be placed based on the calculation of a heuristic.

A. Heuristic

For a given state, x, our tetris controller selects the action, u, which minimizes the sum of the instantaneous cost, $R(u, x)$, and the cost of the resultant state, $V(F(x, u))$:

$$u = \arg\min R(u, x) + V(F(x, u)). \tag{1}$$

Since there are a small number of potential actions (9-34 depending on the piece), it is feasible to evaluate all of them and select the best one. Our tetris simulator provides the dynamics, $x_{i+1} = F(x_i, u)$, and we use a heuristic to estimate the instantaneous cost and the value function, $V(x)$. The heuristic contains 27 constant parameters (detailed in the Appendix). Learning was used to improve our initial guesses. We divide these parameters into two categories, 7 meta-parameters, which control the overall weighting of the heuristic, and 20 minor parameters, which control the costs of specific situations. (Refer to figure 7)

1) Instantaneous Costs: Losing the game and clearing lines are the only events that can happen in a turn. Since no move that ends the game can be better than any move that does not end it, we add an infinite cost for losing the game. Rows can be cleared singly or in groups of 2, 3, or 4. Since there is no special incentive for clearing multiple rows simultaneously, we assumed that the benefit of clearing rows was linearly related to the number of rows cleared, r,

$$R(u, x) = \begin{cases} ar & \text{; game continues} \\ \infty & \text{; game ends} \end{cases} \tag{2}$$

where a is a meta-parameter, which describes the importance of clearing rows. Since clearing rows is good, we used a large, negative value of a, signifying a significant reward for clearing rows. However, after learning, the value was a small, positive value, indicating that row clears are not only accounted for in the $V(x)$ heuristic, but are actually slightly over-accounted for.

2) State Evaluation Heuristic: We posited that the cost of a board configuration is determined by the difficulty with which the board can be cleared, which we modeled as the sum of the difficulty of clearing each individual row

$$V(x) = \sum f(c_i) \tag{3}$$

where c_i is the cost of clearing the ith row and $f(c)$ is a dampening function. The purpose of the dampening function is to reduce the importance of rows that already have a high cost. The result of this is that the algorithm more strongly avoids increasing the cost (increasing the difficulty of clearing) of rows that currently have a low cost (are easy to clear) as opposed to rows that currently have a high cost (are difficult to clear). In order to accomplish this, $\frac{dV}{dc_i}$ must be a decreasing function of c_i, which in turn requires that the second derivative of $f(c)$ be negative for all c. We use f of the form

$$f(c) = c^{\frac{1}{k}} \tag{4}$$

where k is a meta-parameter greater than 1.

In order to clear a row, all of its gaps (defined as contiguous unoccupied portions of a row) must be filled. This is made more difficult if some of the gaps are holes (have pieces above them), requiring other rows to be cleared first. To handle this, we introduce the concept of dependent rows. A hole's dependent rows consist of any that have occupied spaces above it and any rows dependent on those rows that contain spaces. A row's dependent rows are the union of the dependencies of all of that row's holes. The cost of an individual row is determined according to

$$c_i = \sum g + \sum^{d} (c_d + A) + Bh; d \in D \tag{5}$$

where g is the cost of a gap, A and B are meta-parameters, c_d is the cost of a dependent row, h is the number of hole spots, and D is the set of the ith row's dependencies. The last term provides a constant penalty for each hole spot (a hole of width 2 counts as 2). This reflects the fact that for each hole spot there is a column where any further placement will increase the difficulty of clearing the row.

The middle term is the cost of clearing the row's dependencies. The constant portion, A, is necessary to represent the cost of needing to clear another row first even if that row has little cost. This term prevents the placement of pieces above existing holes when that will add additional dependencies (figure 1). It also causes the cost of the lower rows to increase exponentially in a configuration with many holes as the upper rows can be counted a large number of times. This is important because large costs are necessary for the dampening function, equation 4, to correctly reduce the importance of the row. This means that there is very little increase in V when placing pieces above a hole in a row that already has several dependencies.

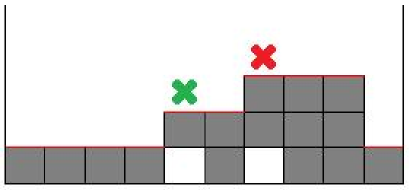


Fig. 1: Filling the spot labeled with a red X will add a row to the bottom row's dependencies, but filling the spot labeled with a green X will not because the bottom row is already dependent on that row.

The first term in equation 5 is the cost of filling all the gaps in the row. Since not all gaps are equally difficult to fill, not all gaps have the same cost. The cost of the gap depends on the width and the shape of the surrounding area. For any gap, the availability of three spots on each side (figure 2, left) determines which pieces can fit into the gap. For a spot to be available it and every spot above it must be unoccupied. Based on the availability of the three spots, each side of the gap can be independently classified into one of four categories (figure 2, right). Treating symmetric situations equally, there are 10 possible combinations of left and right categories. The 20 minor parameters (Refer to Appendix) describe the cost of each of these categories for gaps of width one and two. Since any piece can fit easily into gaps of width three or greater, all large gaps have a cost of 1.0.

Initial (pre-learning) values for the minor parameters were determined by expected number of turns until a piece that can fill the gap without creating a hole appears. Values for gaps of width one ranged from $\frac{7}{6}$ for a gap that is blank on both sides (every piece except the square fits) to 7 for a gap that is tall on both sides (only the "I" piece fits). For gaps of width 2, we only partially counted pieces that would create a large discontinuity (height difference of 2 or greater) by filling the hole. The piece would count as only 0.5 or 0.75 (rather than 1.0) in the denominator if it created a large discontinuity on both or one side respectively. The one exception to this rule is that we used a value of 14 rather than 7 for the gap of width one with high either side. This was done to induce more conservative play that would avoid such dangerous gaps more assiduously.

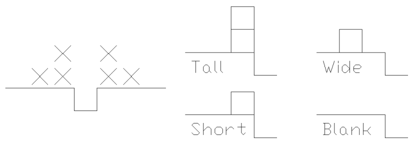


Fig. 2: Left: the x's mark the spots that determine which pieces can fit into the hole. Right: The four ways of classifying one side of a gap.

Additionally, we added a meta-parameter, $\epsilon \in [0, 1]$, designed to control the ratios between the costs of good and bad gaps. The value of ϵ was subtracted from the cost of each gap as described above, which has a much larger effect on gaps that already have a low cost. This value is redundant (does not increase the set of controllers that can be described) if all of the minor parameters are learned, but it did create a single value that scales the minor parameters together in a relevant manner, allowing initial learning in lower dimensions.

Holes were also scored as gaps and included in the first term of equation 5, but when doing so, the hole's dependencies were removed (and the above rows dropped down). For example, in figure (figure 1), the left hole will be classified with just the one row above it removed, meaning blank on the left and wide on the right, whereas the right hole will be classified with both rows above it removed, meaning blank on both sides. A gap that is partially covered (as in figure 3) will be scored seperately as a gap and a hole.

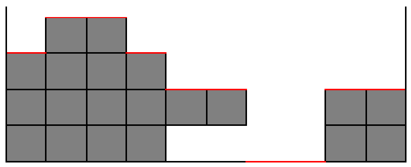


Fig. 3: This is a hole of width 2 and a gap of width 2.

When using the policy so far described, the failure mode was almost always creating deep wells (gaps of width one that have high boundaries on both sides) that it could not fill. To deal with this, we added two additional factors, penalties for multiple wells and penalties for very deep wells. The multiple well penalty added a flat penalty for each well beyond the first. Since this penalty was not associated with any row, it was added directly to V. This penalty accounts for the fact that wells are bad if there is already another well because multiple "I" pieces are needed to fill them. The tall well penalty added a cost for each continuously empty space below a well. This reflects the fact that rows near the top of a deep well are not

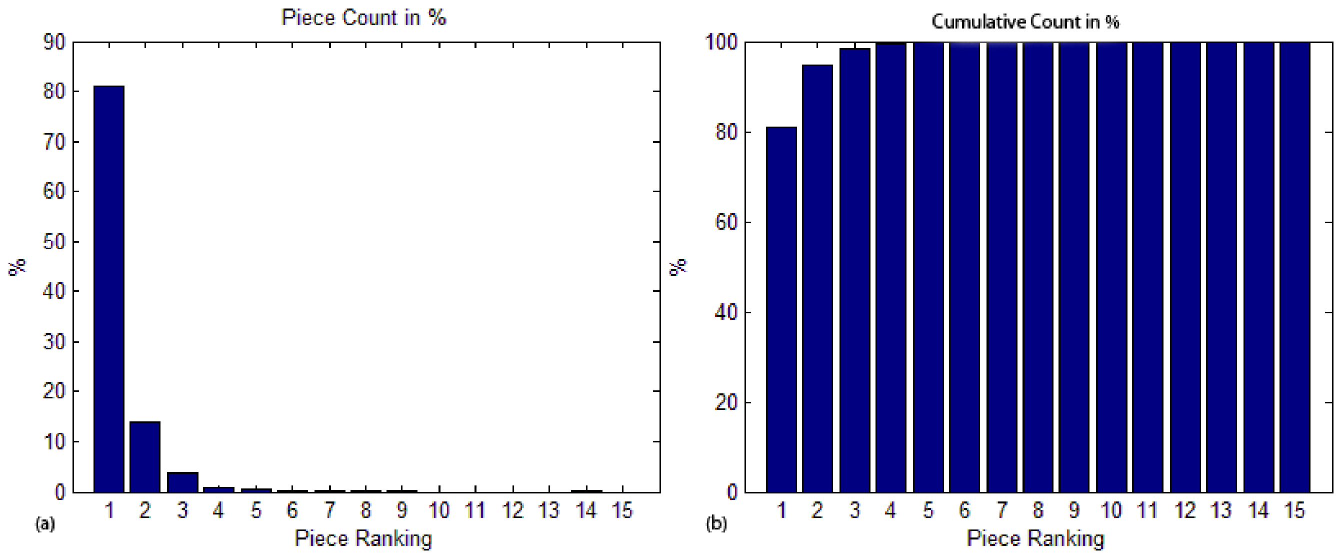


Fig. 4: (a) Graph showing the count (in % of all moves) of the 0 look ahead ranking of the 1 look ahead's best move. When pruning by N moves, any moves that are ranked by the 0 look ahead lower than N would not be found in the search, and thus performance as measured in lines could suffer. (b) The % of all moves less than that ranking. For example, the 5th bar is 99.743, and thus setting N = 5 would result in the same move as the fully expanded 1 look ahead 99.743% of the time. For this reason, we found N = 5 to be a suitable choice when pruning.

easy to clear because multiple "I's" are necessary to fill the well enough to clear that row. In terms of behavior, it helps to correctly identify which rows are easy to clear as well as penalizing deep towers.

B. Lookahead

The heuristic algorithm is designed to score a given state of the board. Normally, in order to make the next move, all legal moves for the given piece are scored by the heuristic and the resulting position with the lowest score is taken. This move, however, doesn't take into account what piece might come next. Independent of how good our current move is, we may change which move to take based on the cost of possible next moves. It'd be beneficial if the cost of a given current move could make a decision based not only on the current board, but rather what the board is likely to end up looking like in the future.

It's not immediately obvious how to implement this technique, as there are stochastic factors such as not knowing which piece will come next, or the piece after that, and so on. The only knowledge the algorithm has is that each piece has a $1/7$ chance of occurring on each turn. For any given legal move with the current piece, the next moves for all 7 pieces have to be considered and scored. There is a question as to whether a player should then take the move that minimizes the expected cost (i.e. the average cost of all 7 possible pieces), or take the more conservative approach of choosing the move of the piece with the highest cost. In our experience, we found the expected cost to be superior, and this technique was used throughout the results shown in this paper.

If the next piece is unknown, there are 7 possible pieces totaling 162 possible next moves. In order to look ahead by 1 move, the player first has to consider these 162 possible next moves for each of the current piece's legal moves. Taking the block as an example of the current piece, there are only 9 legal moves, the least of any other piece. The player would have to perform $9*162 = 1,458$ heuristic evaluations, leading to a 162x slowdown in piece selection time. In order to look two moves ahead, a total of $9*162*162 = 236,196$ heuristic evaluations must be performed. The branching factor of 162 becomes computationally intensive, and thus a many piece look ahead is not possible. In our implementation, a full 2 piece look ahead required about 20 seconds per move on average, which is prohibitively slow. Even a 1 piece look ahead player performed about 3 moves per second. At this rate, a game lasting a million lines would take over 9 days!

To address this issue, we switched from a depth first look ahead, where all possible moves are expanded and evaluated, to a breadth first and prune approach. That is, for any given piece with a set of legal moves, the moves are all evaluated, and then only the best N moves are selected for expansion. Choosing values such as N = 5, a significant portion of the move tree can be eliminated. Following the previous example, where the current piece is the block, there are 9 possible moves. Each of these moves would be evaluated, and then the top 5 would be expanded and all pieces and moves following those 5 are evaluated. This leads to evaluating $9+5*162 = 819$ positions, as opposed to $1,458$ in the 1 look ahead case. The pruned tree is much more significant in the 2 look ahead case. After pruning to N = 5 at the current move level, each of the moves at next level are also pruned, before proceeding to the final level of moves. This requires a total of $9+5*162+7*5*162 = 6,489$ evaluations, an impressive 36x less than without pruning. As it turns out, when the heuristic is good enough to produce games of around $1,000,000$ lines, no amount of pruning at the 2 piece look ahead level is sufficient.

The question then arises as to what penalty is paid in terms of lines when this pruning technique is employed. For instance, the best overall move might not fall into the best 5 moves at the top level, and thus would never be discovered due to the pruning process. To investigate this, we played a game with no pruning where we ranked each possible move with a full 1 look ahead expansion, and also with no look ahead. Then we look at where the best move of the 1 look ahead falls in the ranking of the no look ahead moves. A count of these rankings is kept and is shown in Figure 4. Figure 4a is a graph showing the count (in % of all moves) of the 0 look ahead ranking of the 1 look ahead's best move. When pruning by N moves, any moves that are ranked by the 0 look ahead lower than N would not be found in the search, and thus performance as measured in lines could suffer. Figure 4b shows the % of all moves less than that ranking. For example, the 5th bar is 99.743, and thus setting N = 5 would result in the same move as the fully expanded 1 look ahead 99.743 % of the time. For this reason, we found N = 5 to be a suitable choice when pruning.

C. *Parameter Optimization*

When we initially defined the heuristic, we made educated guesses as to the values of each of the parameters. While the performance was decent, we expected the performance to greatly enhance if we ran the parameters through an optimization process. Due to the variation between scores of any two games, it's hard to calculate the gradient in parameter space, and this makes standard gradient descent methods difficult. To remedy this, we devised a pseudo-gradient descent technique that more mirrors a gradual diffusion process. The idea is to vary one single parameter at a time, leaving all others fixed. The value of the parameter is incremented by some ϵ, and then a single game is played. The value is then decremented by some ϵ, and another game is played. The value of the parameter is then modified slightly in the direction of the game that produced a higher score. This modification is very small, and on average, over many iterations, the parameter should drift in the direction that results in better performance. This process was run on 0 look ahead games, lasting only a few seconds each, which allowed several hundred thousand games to be played.

At first, only the 7 meta-parameters were optimized, and this improved the average 0 look ahead score from around 200 lines to over 4,000 lines(Figure 5). The 1 move look ahead performance improved much more significantly, from 4,500 lines per game with the initial parameters to around 1.2 million lines after optimization. Pleasantly surprised by the improved performance, we adjusted the optimization to also include the 20 minor parameters, and we saw a further improvement in the 0 look ahead performance to around 8,000 lines per game (Figure 6). Unfortunately, with 1 look ahead games lasting as much as two days long, we were unable to produce a new (and presumably higher) average score for the 1 look ahead game by the time of writing.

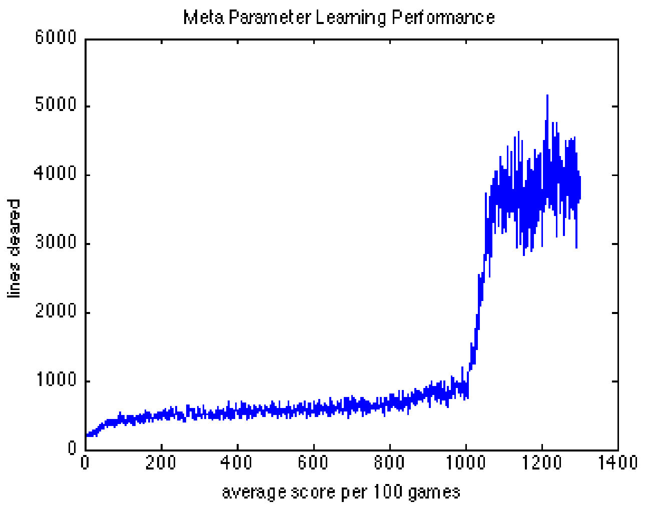


Fig. 5: Example of progress of optimization for the seven meta-parameters.

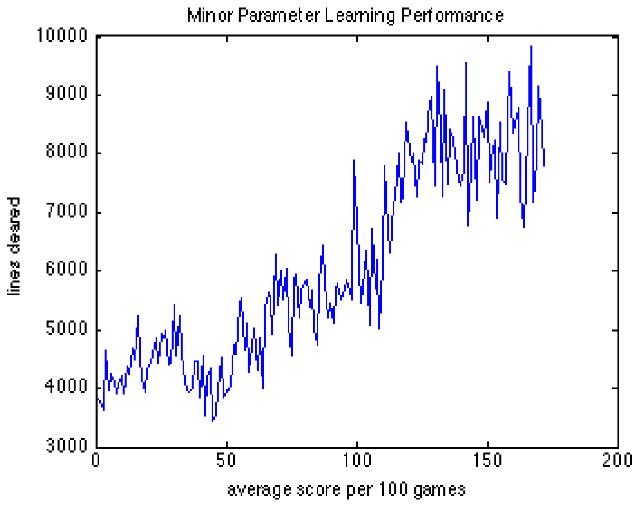


Fig. 6: Example of progress of optimization over all parameters.

III. Results and Conclusions

After optimization, and pruning our lookahead branching, we achieved an average of 1.2 million lines cleared and a max score of 4.5 million lines cleared using our heuristic approach.

We found that a heuristic based controller performed better than using our initial reinforcement learning approach. Our game also showed significant improvement through the use of optimizing previously defaulted parameters. Having a heuristic that can account for all possible tetris sub states and scenarios is vital in getting initial good results. Our ability to model the full tetris state gives an advantage in starting with an already decent tetris player. Using lookahead was the key to pushing over the one million threshold of lines cleared. By properly optimizing the parameters used in the evaluation of this heuristic, our player continued to produce even better scores. We also believe that given more optimization time our scores could of been even higher than the max score of 4.5 million lines cleared.

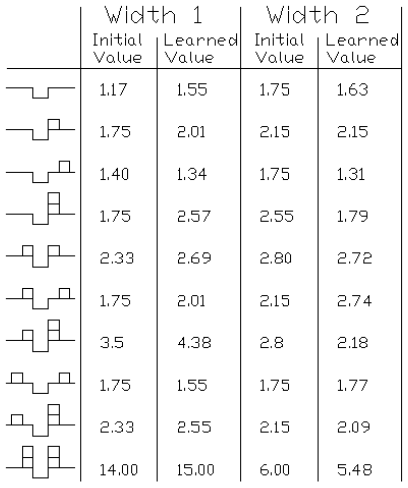
Appendix
Constant Parameters

The following table shows the seven meta-parameters we used for weighting the cost of our heuristic function.

Parameters	Initial	Learned
$\epsilon->$epsilon	0.5	0.344
$B->$hole cost	5.0	17.72
$A->$dep row cost	5.0	1.16
extra tower cost	5.0	0.44
narrow gap cost	5.0	7.9
$a->$clear row reward	-7.0	0.3
$K->$row exponential	2.0	4.384

TABLE I: Meta-Parameters

These are the 20 minor parameters that represent the values for one and two width gap sub states respectively. In evaluating the cost function, the values were categorized based on the following table. Note: these states were also evaluated for all their symmetric cases.



	Width 1		Width 2	
	Initial Value	Learned Value	Initial Value	Learned Value
	1.17	1.55	1.75	1.63
	1.75	2.01	2.15	2.15
	1.40	1.34	1.75	1.31
	1.75	2.57	2.55	1.79
	2.33	2.69	2.80	2.72
	1.75	2.01	2.15	2.74
	3.5	4.38	2.8	2.18
	1.75	1.55	1.75	1.77
	2.33	2.55	2.15	2.09
	14.00	15.00	6.00	5.48

Fig. 7: Possible gap sub states for 1 and 2 width gaps.