Tetris: A Heuristic Study

Using height-based weighing functions and breadth-first search heuristics for playing Tetris

Max Bergmark

May 2015

Bachelor's Thesis at CSC, KTH

Supervisor: Örjan Ekeberg

maxbergm@kth.se

1 Abstract

This paper studies the performance of height-based weighing functions and compares the results to using the commonly used non height-based weighing functions for holes.

For every test performed, the heuristic methods studied in this paper performed better than the commonly used heuristic function. This study also analyses the effect of adding levels of prediction to the heuristic algorithm, which increases the average number of cleared lines by a factor of 85 in total. Utilising these methods can provide increased performance for a Tetris AI.

The polynomic weighing functions discussed in this paper provide a performance increase without increasing the needed computation, increasing the number of cleared lines by a factor of 3.

The breadth-first search provide a bigger performance increase, but every new level of prediction requires 162 times more computation. Every level increases the number of cleared lines by a factor of 9 from what has been observed in this study.

Contents

2 Introduction

Tetris is a computer game created in 1984 by Alexey Pajitnov, in which the goal is to place random falling pieces onto a game board. Each line completed is cleared from the board, and increases the player's score. The game has proven to be challenging, and still is to this day.

During the game, new random blocks (also named tetrominoes) will spawn from the top of the board, and the player can both rotate the pieces, and translate them left and right. The player must place the pieces aptly in order to fill horizontal rows, which is the goal of the game. The game ends when the tetrominoes are stacked to the very top of the game board. In the standard version of the game, the tetrominoes fall faster as the game progresses, eventually ensuring failure.

The rules of the game are simple, but designing artificial intelligence for playing the game is challenging. The goal for any artificial intelligence is to maximize the number of cleared rows. Demaine *et al.* [1] proved that maximizing the number of cleared rows in a tetris game is NP-Complete.

Burgiel [2] proved that it is impossible to win a game of Tetris, in the sense that one cannot play indefinitely. There exists a sequence of tetrominoes which will make the player lose after a finite amount of time, and since the sequence of tetrominoes in the game is random, the player will lose with probability 1 in an infinite game.

To create an artificial intelligence capable of placing pieces in linear time, some sort of simplification is of need. An AI capable of finding the global maximum number of cleared rows will be extremely slow, and hard to implement. Thus, a heuristic approach is selected. In this paper, future prediction heuristics and height-based weighing functions are analysed, and compared to the performance of other artificial players.

3 Method

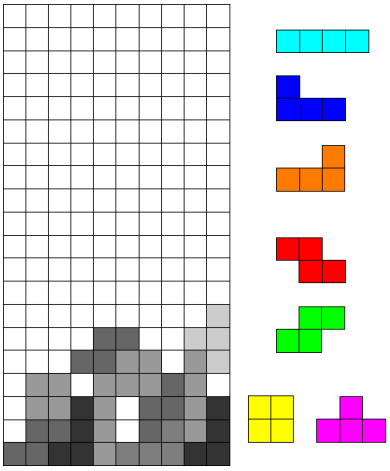

Figure 1: To the left is a sample game board. To the right, the seven tetrominoes are shown. The tetrominoes are named (in order): "I", "J", "L", "Z", "S", "O", "T".

A standard Tetris board is 10 cells wide and 20 cells high. The game is based on seven blocks, seen in Figure 1.

When an artificial intelligence plays the game, certain elements are ignored, while other elements are focused on. The speed of the game is generally ignored. The AI is presented with a game board and a new tetromino to place, and tries to make a decision about where to place it. The tetromino is then transported directly to its intended place, with correct rotation. The AI pauses the game as it makes its decision, and manipulates the tetromino directly to place it.

All testing of different heuristic methods was done on a 10×16 grid. This size was chosen because Boumaza [4] published results using the same grid size, which allowed comparability. The smaller grid size makes the game harder, and thus the games will be shorter.

3.1 Game board suitability

A tetromino can be placed on the game board in many ways, with different rotations and translations. Some of these placements are favorable, and others are disadvantageous. In order to decide which placements are good and which are bad, the AI must try every possible placement. How this is implemented is shown in Figure 2.

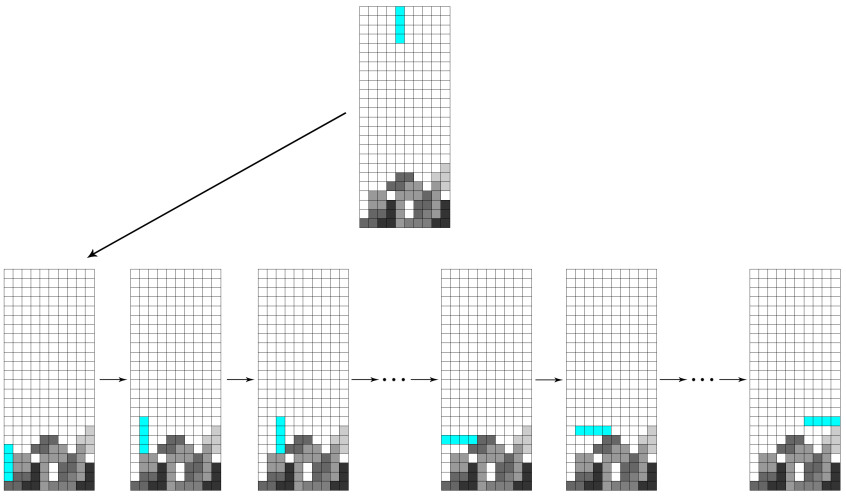

Figure 2: The AI iterates through every possible rotation and translation, in order to find the most suitable one.

As the AI iterates through every possible placement, it must be able to decide whether the current placement is favorable or not. How this scoring is done is discussed in 3.2 and 3.3. After the AI has iterated through every placement, it chooses the one with the best score, according to the scoring system implemented in the AI. The goal is to clear as many lines as possible, and thus the scoring system must be able to find suitable placements for the current tetromino, and make sure that subsequent tetrominoes can be placed favorably.

3.2 The System

The most crucial part of the AI is the scoring system. If the system is chosen poorly, the game will be lost soon. In order to survive as long as possible, the AI must be able to predict future events.

3.2.1 The Basics

When the AI is presented with a game board, and wishes to evaluate the board's suitability, it calculates the number of holes on said board. A hole is defined as follows:

1. An empty cell to the left of a column in which the topmost cell is at the same height or above the empty cell in question.
2. An empty cell under the topmost filled cell in the same column.
3. An empty cell to the right of a column in which the topmost cell is at the same height or above the empty cell in question.

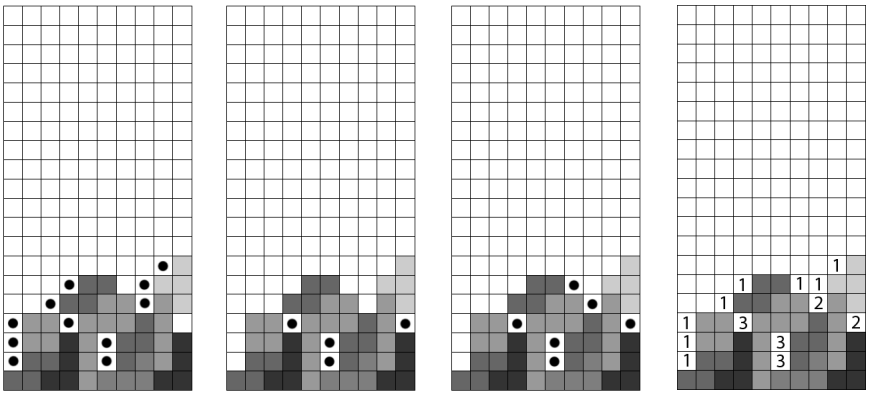

Figure 3: A visualization of the different definitions for holes, according to the rules presented above. From left to right: Rule 1, Rule 2, Rule 3. Rightmost is the hole sum for the game board.

These three rules are not mutually exclusive, meaning that a single cell can be counted as 1, 2 or 3 holes depending on the surrounding grid. Rule 1 and rule 3 are similar to each other, thus they will be treated as if they were a single rule. In Figure 3 the scoring of the holes can be seen. To calculate the total score for the given board, sum the scores for the individual cells. This gives a score of $1 + 1 + 1 + 1 + 1 + 2 + 1 + 3 + 2 + 1 + 3 + 1 + 3 = 21$. As the AI iterates through all possible placements, it searches for the placement which results in the lowest score.

3.2.2 Height based weighing functions

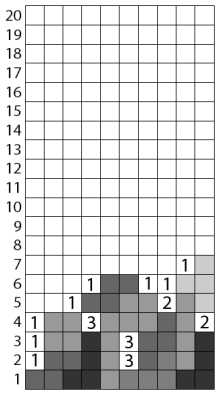

Figure 4: The game board with row enumeration

An improvement of the AI is to add weights to the scoring system. For this study, two functions are chosen, $f(y)$ and $g(y)$, where y describes the row number, seen in Figure 4. The score of a game board is then defined as follows (rules are discussed in 3.2.1):

$$\sum_i g(y_i) + \sum_j f(y_j) + \sum_k g(y_k)$$

$$i \in \{\text{cells obeying rule 1}\}$$

$$j \in \{\text{cells obeying rule 2}\}$$

$$k \in \{\text{cells obeying rule 3}\}$$

To retort to the basic AI, simply choose $f(y) = g(y) = 1$. In order to increase the average number of cleared rows, $f(y)$ is chosen to be a polynomial function of degree n, and $g(y)$ a polynomial of degree $n-1$. The simplest such functions are $f(y) = y$ and $g(y) = 1$.

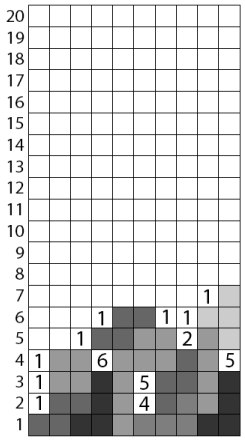

Figure 5: The game board score with $f(y) = y$ and $g(y) = 1$.

In Figure 5, the difference in the scoring can be seen. With $f(y) = y$ and $g(y) = 1$ the game board is given 27 points, instead of 23 (see 3.2.1). The holes in the middle are given higher scores, resulting in the AI avoiding boards with many holes adhering to Rule 1.

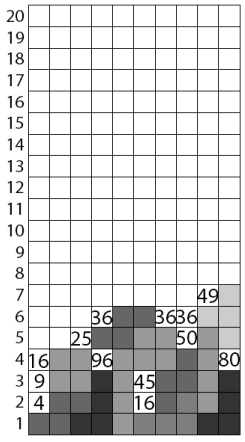

Figure 6: The game board score with $f(y) = y^3$ and $g(y) = y^2$.

The example grid is given 498 points, distributed according to Figure 6. The AI searches for the lowest possible score, and thus wants to avoid holes with high scores. Now it can be seen that the hole in the middle worth 96 points is very unfavorable, and thus the AI will likely choose placements which will remove that hole from the grid.

3.3 Breadth first prediction

In this section, the AI is improved by utilising a breadth first search for future events, which are averaged in order to provide a better scoring for the current game board. Zhou and Hansen [5] discuss breadth-first heuristic searches, inspiring the choice of method in this paper. Rajkumar [6], Lishout, Chaslot and Uiterwijk [7], and Bergmark and Stenberg [8], have utilized Monte-Carlo searches to develop artificial intelligence for chess, backgammon and Go, where randomness is a factor. This paper focuses on complete breadth first searches, meaning that every possible outcome is analysed for a certain amount of future steps.

3.3.1 One step prediction

In order to improve the AI even further, it must be able to predict future events. This is implemented by iterating over every possible placement for the current tetromino. For every possible placement of the current tetromino, the AI places each of the seven tetrominoes in every possible position. For example, if the current piece is a L-piece, the AI tries to calculate the optimal placement for a L-piece followed by an I-piece, then for a L-piece followed by an S-piece, an L-piece followed by a Z-piece and so on, for each of the seven tetrominoes. A visualization of this can be seen in Figure 7.

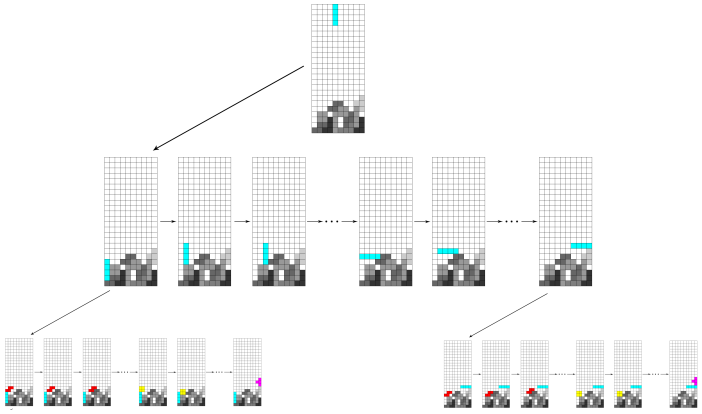

Figure 7: The AI observes every possible outcome for the current tetromino and the one that follows, in order to find the most suitable placement for the current tetromino.

Once every possible pair of tetrominoes has been accounted for, the AI averages the seven scores, and the average is the score for the placement of the current

tetromino. By doing this, the AI predicts how the placement of the current tetromino will affect the future of the game, making the AI less greedy.

Calculating every possible placement for two pieces is a lot more compute-intensive, slowing the algorithm down by a factor of 162. However, the AI is still able to make its decision quickly, in less than 1 ms.

3.3.2 Two step prediction

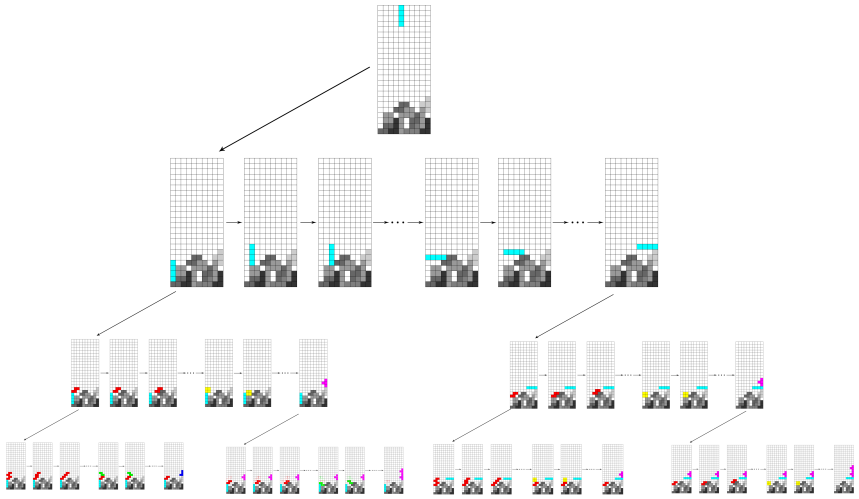

Figure 8: The AI observes every possible outcome for placing the current tetromino and two subsequent tetrominoes, in order to find the most suitable placement for the current tetromino.

In order to improve even further, another layer of prediction is added. For every possible placement of the current tetromino, every possible placement for every possible combination of two tetrominoes is observed, visualized in Figure 8. On average, 607361 game boards are analyzed for every new tetromino that spawns, making the AI 26244 times slower than the very basic AI discussed in 3.2, and 162 times slower than the AI with single step prediction discussed in 3.3.1.

3.3.3 Deeper prediction

Further exploration is easily implemented recursively. However, the algorithm's time complexity is $O(162^n)$, where n is the level of prediction. For three step prediction $162 * 607361 = 98389242$ game boards would have to be observed for every new tetromino that is played. This means that the software has to run on an extremely fast computer on many cores in order to get any results. Four step prediction would require analyzing almost $16 * 10^9$ game boards for each iteration, which simply is not reasonable.

The scoring function implemented in the AI can evaluate $17 * 10^6$ game boards per second. In order to analyze game boards quickly, an optimized function was implemented. How this function works is discussed in Appendix B.For three step prediction, deciding where to place a piece then takes 5.78 seconds, and for four step prediction a decision would take over 15 minutes. Assuming that the game scores obtained using three and four step prediction are at least as high as the scores obtained from two step prediction, a single three step prediction game would take:

$$5.78\,\text{s/tetromino} * 946180\ \text{tetrominoes/game} = 63\ \text{days}$$

A single four step prediction game would take:

$$15.62\,\text{min/tetromino} * 946180\ \text{tetrominoes/game} = 28.1\ \text{years}$$

In order to obtain any kind of results for three or more levels of prediction, a supercomputer is needed.

4 Results

Using weighing functions that are height dependant provides an improvement compared to scoring systems without weighing. The improvement factor varies depending on the recursion depth and the choice of weighing functions $f(y)$ and $g(y)$.

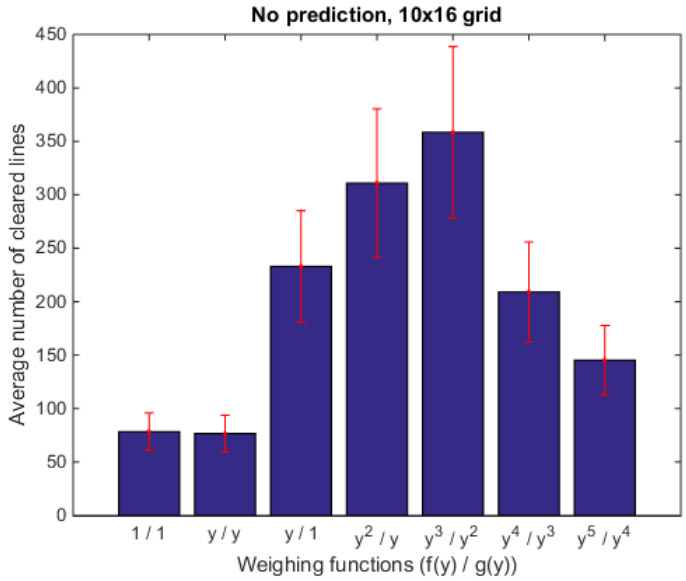

Figure 9: Results for the AI with no prediction.

$f(y)$	1	y	y	y^2	y^3	y^4	y^5
$g(y)$	1	y	1	y	y^2	y^3	y^4
Lines	78.40	76.65	233.00	310.95	357.95	209.10	145.30

Table 1: The average number of cleared lines for functions $f(y)$ and $g(y)$ on a 10×16 grid with no prediction.

For zero step prediction, the maximum number of cleared rows is obtained when $f(y) = y^3$ and $g(y) = y^2$. The improvement factor is $\frac{357.95}{78.40} = 4.57$ compared to $f(y) = g(y) = 1$.

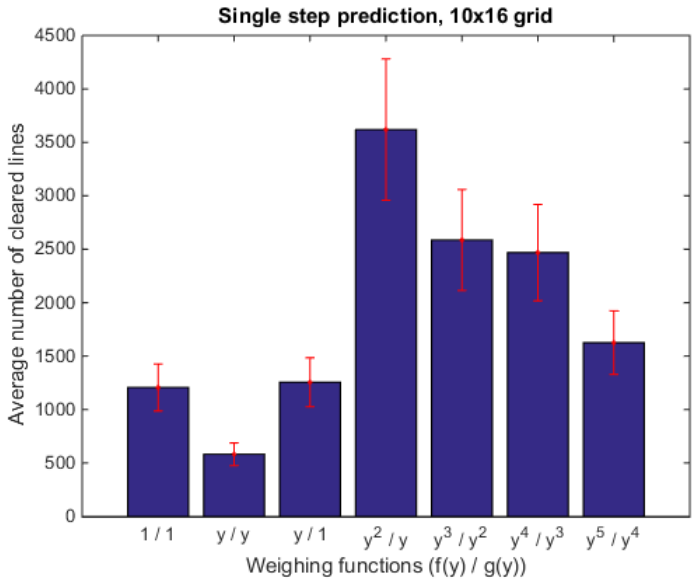

Figure 10: Results for the AI with one step prediction on a 10×16 grid.

$f(y)$	1	y	y	y^2	y^3	y^4	y^5
$g(y)$	1	y	1	y	y^2	y^3	y^4
Lines	1206	582	1256	3620	2587	2468	1626

Table 2: The average number of cleared lines for functions $f(y)$ and $g(y)$ on a 10×16 grid with one step prediction.

For one step prediction, the maximum number of cleared rows is obtained when $f(y) = y^2$ and $g(y) = y^1$. The improvement factor is $\frac{3620}{1206} = 3.00$ compared to $f(y) = g(y) = 1$.

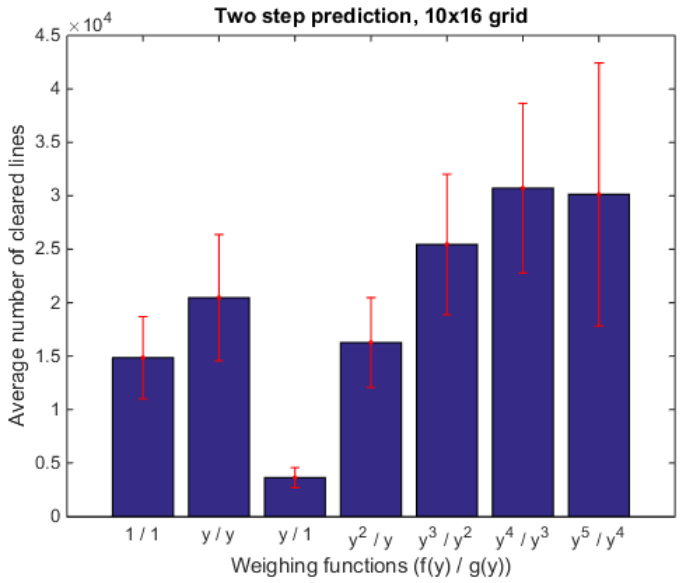

Figure 11: Results for the AI with two step prediction on a 10×16 grid.

$f(y)$	1	y	y	y^2	y^3	y^4	y^5
$g(y)$	1	y	1	y	y^2	y^3	y^4
Lines	14857	20471	3634	16271	25444	30718	30132

Table 3: The average number of cleared lines for functions $f(y)$ and $g(y)$ on a 10×16 grid with two step prediction.

For one step prediction, the maximum number of cleared rows is obtained when $f(y) = y^4$ and $g(y) = y^3$. The improvement factor is $\frac{30718}{14857} = 2.07$ compared to $f(y) = g(y) = 1$.

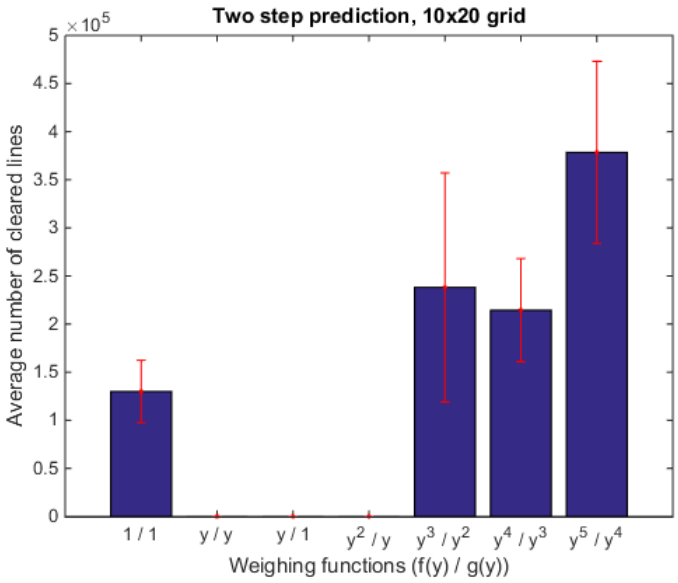

Figure 12: Results for the AI with two step prediction on a 10×20 grid.

$f(y)$	1	y	y	y^2	y^3	y^4	y^5
$g(y)$	1	y	1	y	y^2	y^3	y^4
Lines	129910				238119	214400	378472

Table 4: The average number of cleared lines for functions $f(y)$ and $g(y)$ on a 10×20 grid with two step prediction. Three pairs of weighing functions have not been tested because of time constraints.

For one step prediction, the maximum number of cleared rows is obtained when $f(y) = y^5$ and $g(y) = y^4$. The improvement factor is $\frac{378472}{129910} = 2.91$ compared to $f(y) = g(y) = 1$.

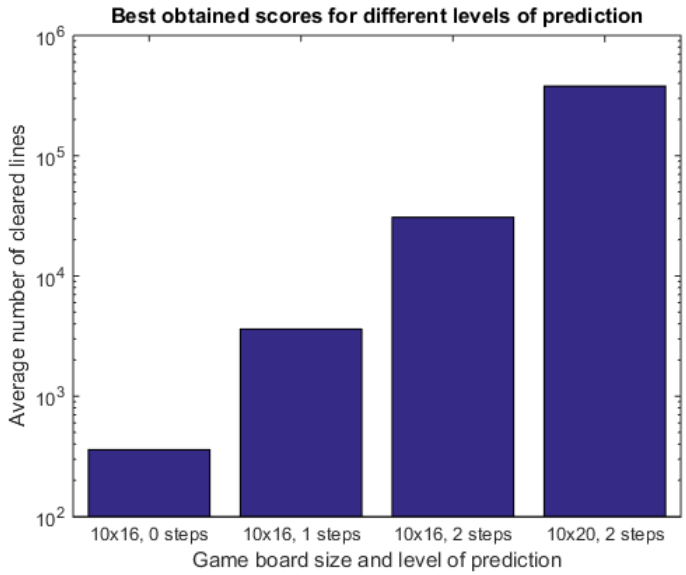

Figure 13: Best scores for different levels of prediction.

For every level of prediction, the number of cleared rows is increased by a factor of 8.5 or 10. In order to find a correlation between the level of prediction and the number of cleared rows, more analysis would be needed. As discussed in 3.3.3, obtaining results for three or more levels of prediction is extremely time-consuming. Thus, no clear link can be found.

5 Discussion

The results show a clear correlation between height-based weighing functions and increased score, on average increasing the number of cleared lines by a factor of 3 on average compared to the commonly used hole definition. Since both the height-based weighing functions and the non height-based weighing function have the same time complexity, using the methods presented here can improve an AI without affecting the computation costs.

The results also show a clear correlation between the level of prediction and the number of cleared rows, increasing the number of cleared rows by a factor of 9 on average. However, this increase comes at a massive cost, where each new level of prediction required 162 times more computing.

One of the best hand tuned artificial players was designed by Dellacherie [3],

clearing 63×10^4 lines on average on a 10×20 grid. The AI implemented by Dellacherie uses 6 weighing functions to provide an evaluation for the game board. The AI implemented in this paper is able to clear 38×10^4 rows on average using 2 weighing functions and two levels of prediction. These results are definitely comparable, and perhaps implementing height-based weighing functions in Dellacherie's AI can provide an even better score.

Boumaza's [4] AI uses a genetic algorithm, obtaining an average score of 36.3×10^6 cleared rows. Boumaza's algorithm uses 8 different weighing functions, which are dynamically weighed using a genetic approach. One of these 8 functions calculates the number of holes on the game board, and perhaps implementing a height-based weighing function instead can provide an even better result.

6 Conclusion

The results show a clear correlation between level of prediction and the average number of cleared rows, as well as an increase in the average number of cleared rows when using height-based weighing functions as opposed to the commonly used heuristic method.

Implementing a breadth-first search for predicting future events or using a height-based weighing function can increase the performance of a Tetris AI. How this can be implemented efficiently can be seen in Appendix B.

The polynomic weighing functions provide a performance increase without increasing the needed computation, increasing the number of cleared lines by a factor of 3.

The breadth-first search provide a bigger performance increase, but every new level of prediction requires 162 times more computation. Every level increases the number of cleared lines by a factor of 9 from what has been observed in this study.

A Tables

$f(y)$	1	y	y	y^2	y^3	y^4	y^5
$g(y)$	1	y	1	y	y^2	y^3	y^4
Scores	29	51	87	434	307	34	50
	93	24	60	692	201	47	274
	24	39	156	361	349	450	74
	77	95	68	66	144	362	50
	113	126	149	596	222	91	184
	84	69	159	134	415	195	45
	59	77	117	200	398	46	75
	26	69	24	248	94	257	386
	172	119	57	374	391	51	58
	126	76	453	34	468	194	72
	67	41	28	293	170	171	37
	43	32	370	1005	896	578	225
	94	58	641	520	315	122	204
	132	238	231	132	1270	213	194
	110	77	312	36	465	316	53
	51	98	171	292	92	25	149
	74	34	210	302	77	31	146
	33	52	53	52	46	106	47
	126	107	252	248	93	314	340
	35	51	1062	200	746	579	243
Average	78.40	76.65	233.00	310.95	357.95	209.10	145.30
Variance	17.53	17.14	52.10	69.53	80.15	46.76	32.49

Table 5: The number of cleared lines for functions $f(y)$ and $g(y)$ on a 10×16 grid with zero step prediction for different random seeds.

$f(y)$	1	y	y	y^2	y^3	y^4	y^5
$g(y)$	1	y	1	y	y^2	y^3	y^4
Scores	422	462	833	1910	3069	2196	428
	596	847	844	7313	8740	602	206
	1539	231	1934	3509	317	1433	3019
	1451	182	60	4152	6956	2746	409
	88	533	334	3522	3250	6875	1660
	4873	516	962	1424	197	207	596
	1708	1864	1113	511	1474	1896	2235
	1750	477	2568	5599	5108	20	6114
	1299	395	1826	2567	1343	3911	793
	292	98	705	5037	5898	2060	1575
	1776	163	501	5552	1160	3562	1128
	57	352	175	2390	3826	5566	1815
	129	1497	3118	608	1508	1501	377
	1883	1221	1283	9181	3128	2443	1884
	1436	1296	3675	11132	726	1402	4173
	224	331	1840	94	1102	205	973
	235	358	399	285	1916	6464	755
	231	962	1548	2497	8654	460	1925
	323	703	513	4796	4564	5998	246
	1847	84	2355	726	1771	713	1245
	1077	146	119	136	2870	826	825
	1864	209	237	10269	1962	528	1491
	4721	70	1048	1712	1635	3879	1391
	202	365	4156	6901	458	3747	470
	458	247	23	3891	423	257	256
	2685	95	1941	2270	74	1701	2346
	1139	2374	1346	2363	1606	19	3243
	983	265	1335	3020	912	9534	1562
	185	1039	862	4877	1811	1043	3453
	723	73	17	349	2139	1268	2195
Average	1206.53	581.83	1255.70	3619.77	2586.57	2468.73	1626.23
Variance	220.28	106.23	229.26	660.86	472.24	450.73	296.91

Table 6: The number of cleared lines for functions $f(y)$ and $g(y)$ on a 10×16 grid with one step prediction for different random seeds.

$f(y)$	1	y	y	y^2	y^3	y^4	y^5
$g(y)$	1	y	1	y	y^2	y^3	y^4
Scores	23525	3356	1199	9197	614	92456	87411
	27709	29005	6155	4203	44402	6798	25462
	3787	3716	157	6941	2575	23673	2128
	13477	3962	7619	8364	7183	7734	3445
	36426	6705	5730	2451	17721	76184	9710
	47453	5101	1157	3019	38527	43246	52635
	486	2567	7544	18113	58665	47675	
	14582	46783	22	5631	21307	7091	
	12361	117759	3874	9261	20872	4206	
	5711	9520	4592	38454	38979	27245	
	1767	1876	1774	60867	13197	40676	
	6707	15299	1410	4011	92099	1413	
	16566		6025	9623	5503	6023	
	9397		1276	55297	9473	14561	
	2901		5977	8640	32637	61796	
Average	14857.00	20470.92	3634.07	16271.47	25444.27	30718.47	30131.83
Variance	3836.06	5909.44	938.31	4201.27	6569.68	7931.47	12301.27

Table 7: The number of cleared lines for functions $f(y)$ and $g(y)$ on a 10×16 grid with two step prediction for different random seeds.

$f(y)$	1	y	y	y^2	y^3	y^4	y^5
$g(y)$	1	y	1	y	y^2	y^3	y^4
Scores	72628				245008	92456	327886
	318496				204255	284216	38952
	189279				471121	382979	507232
	129124				32091	145190	298181
	76368					91799	154081
	47453					252994	848026
	2605					479349	602971
	50952					133480	256764
	292540					465712	156110
	24587					179451	138712
	82130					142139	656865
	91684					50997	509083
	203183					291087	410856
	254213					202804	466260
	178612					371366	151372
	64710					120182	532205
Average	129910.25				238118.75	214400.06	378472.25
Variance	32477.5625				119059.38	53600.02	94618.06

Table 8: The number of cleared lines for functions $f(y)$ and $g(y)$ on a 10×20 grid with two step prediction for different random seeds.

B Optimized hole calculation

In order to achieve the performance needed to run the AI with two step prediction, a very efficient method for analysing game boards is needed. The software developed for this paper uses the following method:

```
public int calcHolesConverted(int[] grid) {

    int underMask   = 0;
    int lneighborMask = 0;
    int rneighborMask = 0;
    int foundHoles  = 0;
    int minY = 0;

     while ( minY < ysize && grid[minY] == EMPTY_ROW ) {
         minY++;
     }

     for (int y = minY; y < ysize; y++) {
         int line  = grid[y];
         int filled = ~line & EMPTY_ROW;

         underMask    |= filled;
         lneighborMask |= (filled << 1);
         rneighborMask |= (filled >> 1);

         foundHoles += scoreFun(3, y)*setOnes[underMask & line];
         foundHoles += scoreFun(2, y)*setOnes[lneighborMask & line];
         foundHoles += scoreFun(2, y)*setOnes[rneighborMask & line];
     }
     return foundHoles;
 }

public long scoreFun(int p, int y) {
    long result = 1;
    for (int i = 0; i < p; i++) {
       result *= (ysize-y);
    }
    return result;
}
```

The grid to be analysed is represented as an array of integers, where every integer represents a row in the grid. The bits in the integer represents the cells in a row. An empty cell gives a 1-bit, and a filled cell gives a zero bit. Using bitwise operations is extremely efficient, and provides the performance that is needed. `setOnes` is a pre-calculated array where `setOnes[x]` returns the number of set bits in the integer `x`.

References

[1] E. D. Demaine, S. Hohenberger, and D. Liben-Nowell. Tetris is hard, even to approximate. In *Proc. 9th COCOON*, pages 351–363, 2003.

[2] H. Burgiel. How to lose at Tetris. *Mathematical Gazette*, 81:194–200, 1997.

[3] C. P. Fahey. Tetris AI, Computer plays Tetris, 2003. On the web http://colinfahey.com/tetris/tetris_en.html.

[4] Amine Boumaza. How to design good Tetris players. 2013. ¡hal-00926213¿

[5] Rong Zhou, Eric A. Hansen. Breadth-first heuristic search. Artificial Intelligence 170 (2006) 385–408

[6] Prahalad Rajkumar. A Survey of Monte-Carlo Techniques in Games.

[7] François Van Lishout, Guillaume Chaslot, Jos W.H.M. Uiterwijk. Monte-Carlo Tree Search in Backgammon.

[8] Fabian Bergmark, Johan Stenberg. Heuristics in MCTS-based Computer Go.