



**NTNU – Trondheim**  
Norwegian University of  
Science and Technology

# A comparison of feature functions for Tetris strategies

**Jonas Balgaard Amundsen**

Master of Science in Computer Science

Submission date: June 2014

Supervisor: Keith Downing, IDI

Norwegian University of Science and Technology  
Department of Computer and Information Science



## Acknowledgements

Special thanks go to my project supervisor, Prof. Keith L. Downing, for important advice, guidance and encouragement.

Additionally, this research was supported in part with computational resources at NTNU provided by NOTUR, <http://www.notur.no>.



## Abstract

Finding optimal strategies for the game of Tetris is an interesting NP-complete problem that has attracted several AI researchers. Their approaches display subtle variations in the implementation details, with unclear relationships between these details and Tetris performance. This, combined with the absence of confidence intervals in most published results, makes the evaluation and comparison of Tetris strategies and optimization methods very difficult.

To look further into this unclear relationship, we would re-create every environment described in several publications. An evolutionary algorithm was executed within each environment to create multiple AIs and their performance compared against each other. The scores differed *substantially*. This suggests that some aspects of the Tetris environment greatly affects the potentially obtainable performance of an AI. We come to the unfortunate conclusion that nearly no results of existing publications can be used to compare optimization methods against each other in terms of suitability for Tetris due to this reason.

## Sammendrag

Å finne optimale strategier for spillet Tetris er et interessant NP-komplett problem som har tiltrukket seg flere AI-forskere. Deres tilnærminger avviker derimot fra hverandre når det gjelder enkelte implementasjonsdetaljer og det et uklart forhold mellom dette og oppnådd ytelse. Kombinert med manglende konfidensintervaller in de fleste publiserte resultater, så er det vanskelig å evaluere og sammenligne Tetris-strategier og optimaliseringsmetoder.

For å finne ut mer om dette forholdet, så forsøkte vi å gjenskape alle miljøene beskrevet i flere publikasjoner. En evolusjonær algoritme ble kjørt i alle disse miljøene for å lage flere AI-er og resultatene ble sammenlignet mot hverandre. Poengene oppnådd var *svært* forskjellige. Dette antyder at enkelte aspekter ved et Tetris miljø i stor grad påvirker den potensielt oppnåelige ytelsen til en AI. Vi kommer av denne grunn til den uheldige konklusjonen at nesten ingen resultater i eksisterende publikasjoner kan brukes for å sammenligne optimaliseringsmetoder mot hverandre med hensyn på hvor anvendelige de er til Tetris.



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Background</b>	<b>5</b>
2.1	Building a Tetris controller . . . . .	5
2.2	Related work . . . . .	11
2.3	Difficulty of comparing controllers . . . . .	12
2.3.1	Domain variations . . . . .	12
2.3.2	Large deviation in score . . . . .	16
2.3.3	Differing feature functions . . . . .	16
2.4	Computational cost . . . . .	17
2.5	Application optimization . . . . .	20
2.6	Estimating performance of a strategy . . . . .	20
<b>3</b>	<b>Methodology</b>	<b>24</b>
3.1	Inspiration . . . . .	24
3.2	Why evolutionary computing? . . . . .	25
3.3	What is an evolutionary algorithm? . . . . .	25
3.4	Implementation . . . . .	27
3.4.1	Representation . . . . .	27
3.4.2	Fitness function . . . . .	27
3.4.3	Selection . . . . .	27
3.4.4	Elitism . . . . .	28
3.4.5	Crossover . . . . .	28
3.4.6	Mutation . . . . .	28
3.4.7	Parallelization . . . . .	28
3.4.8	Random values . . . . .	29
<b>4</b>	<b>Results</b>	<b>30</b>
4.1	Comparing environments . . . . .	30

4.2	Unbiased feature selection . . . . .	31
<b>5</b>	<b>Discussion</b>	<b>45</b>
5.1	Feature function contribution to search space . . . . .	45
5.2	Large score deviation & halting evolution . . . . .	46
5.3	Competitive general purpose optimization . . . . .	46
5.4	Time constraints . . . . .	47
<b>6</b>	<b>Conclusion</b>	<b>48</b>
<b>A</b>	<b>Formal feature definitions</b>	<b>51</b>
<b>B</b>	<b>Example game</b>	<b>58</b>





# Chapter 1

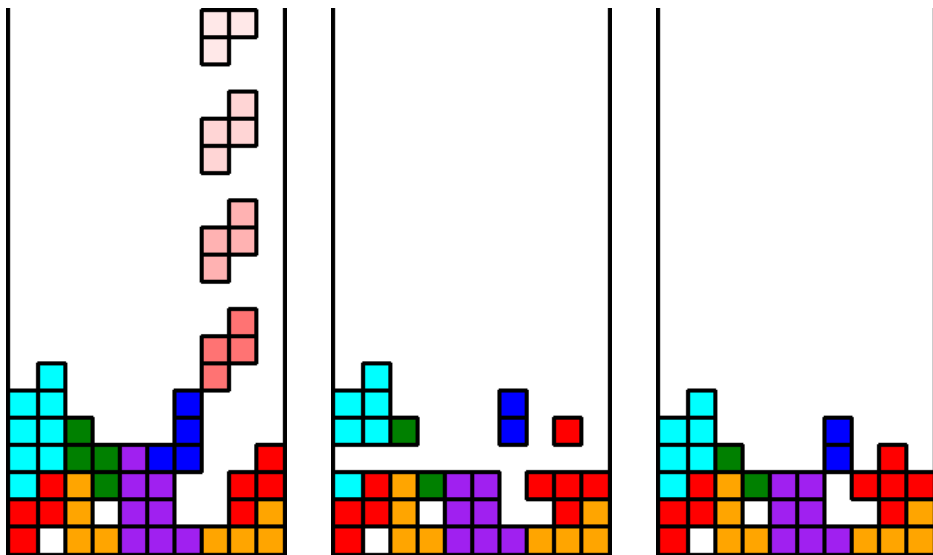
## Introduction

Tetris is a tile-matching puzzle game created by Alexey Pajitnov in 1984. A game of Tetris is played on a 2-dimensional structured grid where each cell can be either empty or full. Additionally, Tetris consists of several distinctive pieces called *tetrominos*, illustrated in figure 1.1. Throughout the game and one by one, pieces will enter from the top of the board and move downwards. A piece stops once it hits a full cell or the bottom of the board. The player is able to rotate falling pieces and move them along the horizontal axis. If a piece is placed such that a row of cells is completely filled, then that row is cleared, I.E. its cells are emptied, and all rows above are moved one step down. This process is depicted in figure 1.2. The game ends once the board become so full that the player is unable to place the next falling piece without overflowing the board. Points are usually awarded each time a row is cleared, but a player may be given additional points for clearing several rows at once, clearing rows consecutively, etc. Thus, by sensibly placing each falling tetromino, the player can avoid ending the game and achieve higher scores. A more thorough specification of Tetris can be found on Fahey's website [1].

The total number of states, if one consider each possible board configuration and



**Figure 1.1:** All tetrominos used in Tetris, as specified by Fahey [1].



**Figure 1.2:** The process of removing a row from the board: 1) a piece is placed such that a row becomes completely filled, 2) that row is emptied, and 3) every row above is moved one step down.

the current falling piece, is vast and a liberal estimate (because it contains some impossible configurations [1]) is  $7.0 \cdot 2^{199} \approx 5.6 \cdot 10^{59}$ . That is many orders of magnitude greater than even the number of grains of sand on all the beaches on earth<sup>1</sup>. Tetris is also an NP-complete problem, even if the sequence of pieces is known in advance [2], and it has been shown that every game finishes with a probability of 1 [3]. This makes it an interesting optimization problem.

The game of Tetris is found in numerous scientific publications where it typically serves one of two purposes: 1) fulfilling the need for a practical and optimizable application used to test a new approach for optimization, or 2) being subject of an ongoing, and possibly everlasting, informal competition for gaining the highest score through non-human play. Existing publications however vary in terms of several important aspects which makes the results difficult to compare.

Firstly, applications vary in terms of the problem domain. For example, a player may be given information in regards to the upcoming pieces. This is called lookahead and is usually one or zero. A lookahead of more than zero enables a player to make more educated actions and possibly yield a higher score. Secondly, subtle implementation details such as when the game is considered to be over differs slightly. Most implementations seem to define game over as when a piece is placed such that it overflows the board, but at least one implementation differs in that it ends when a piece touches the top. Lastly, very few publications utilize the same feature functions to extract information about a Tetris board within their algorithms. In other words, almost no methods have been tested with the same information available to them.

**We hypothesize that variations in all of these aspects, particularly the last one regarding different choice of feature functions, may greatly affect the overall performance of an agent developed to play Tetris.**

In order to test this hypothesis, we will to the best of our ability recreate the exact environment described in each publication (in terms of problem domain, implementation and choice of feature functions) and evolve an agent within each one using the same evolutionary algorithm. If the hypothesis is true, then the overall performance of the agents evolved within these environments should be significantly different.

A truthy hypothesis will have a somewhat unfortunate consequence. If it can be shown that variations in Tetris domain affect the performance of an agent, then the results of agents developed in varying domains is obviously not comparable to each other at all. Additionally, if variations in the choice of feature functions

---

<sup>1</sup>The number of grains of sand on all the beaches on earth has been estimated by researchers at the University of Hawaii. See <http://www.hawaii.edu/suremath/jsand.html>.

also affect the performance of an agent, then the employed optimization methods with varying feature functions also cannot be compared to each other in terms of Tetris play.

Furthermore, we will attempt evolve our very own Tetris controller using feature functions from all previous works. During this experiment, the evolutionary algorithm will be allowed to choose which feature function to utilize without any bias. We suspect that by using an evolutionary algorithm for this purpose, we can gain more insight as to which feature functions are valuable, and with a bit of luck, maybe even beat the current best one-piece controller, which to our knowledge is that of Thiery and Scherrer [4]. Their controller achieves an average performance of 35,000,000 rows cleared per game.

# Chapter 2

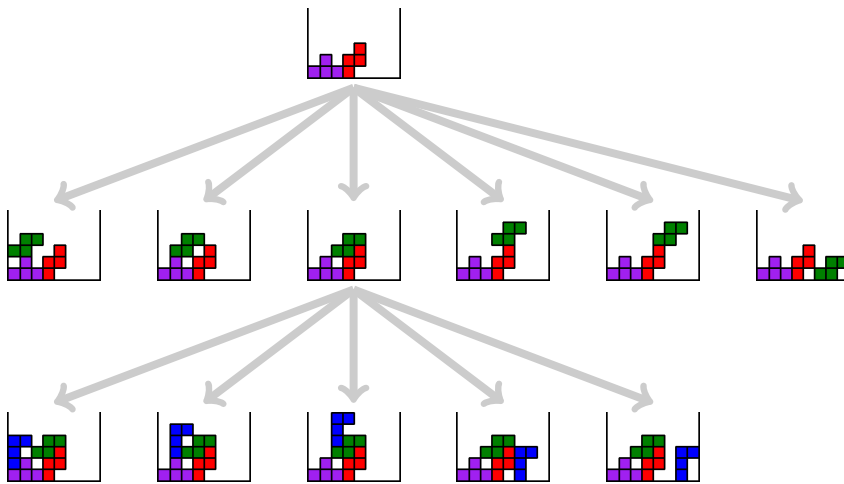
## Background

### 2.1 Building a Tetris controller

The ideal and completely hypothetical strategy for playing Tetris is that where each possible successor state is considered for an infinite amount of levels. By averaging scores obtained through each possible game play and backtracking the results, the controller would be able to perfectly place any falling tetromino. However, due to huge computational requirements, this may never be possible. Instead, a heuristic approach is chosen and employed by all controllers (to the best of our knowledge).

Controllers that utilize the information of the board and the current falling tetromino only are *one-piece controllers*. When the next tetromino is known in advance and that information is also used, the controller is called a *two-piece controller*. A two-piece controller will utilize more information than a one-piece controller in its decision-making and can make more educated guesses of what's a sensible placement. Two-piece controllers generally perform better than one-piece controllers, but take considerably longer time to finish.

To decide which orientation and horizontal placement to chose, the controller will create a tree representing all reachable successor states from the *known tetrominos*, partially illustrated in figure 2.1. A rating function is used to give each leaf node state a score. The state that yields the highest score is selected and the first action leading to that state is played. This process is then repeated for each falling tetromino and thus, the problem of playing Tetris is reduced to creating a good rating function.



**Figure 2.1:** A partial illustration of a tree of states constructed from two known tetrominos in a game of Tetris played on a 8 by 6 grid. The first level illustrates the current state of the board. The second level illustrates all possible successor states (not counting different rotations). The third level illustrates each possible successor state (not counting different rotations) from *one* of the previous states.

A rating function is typically a weighted sum of several *feature functions*. The purpose of a feature function is to numerically value some meaningful feature or characteristic of a Tetris board. By weighting a feature positively or negatively, one can encourage or discourage, respectively, formations of certain features on a Tetris board during a game. If these features affect the duration of the game and helps to avoid ending it, they will contribute to an overall performance improvement. The choice of feature functions and their relative weights is what constitutes a Tetris *strategy*.

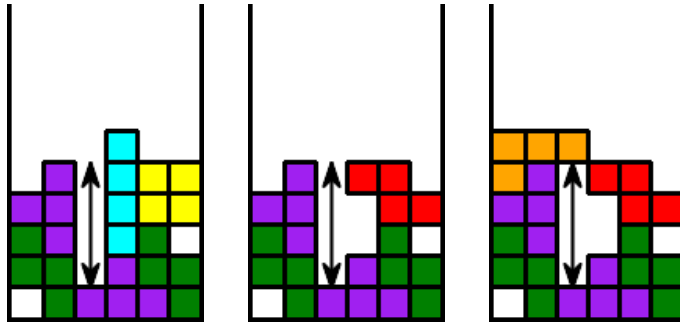
An example of a typical feature function that nearly all related work employ is *holes*. A hole in a Tetris board is an empty cell which is covered by a full cell, I.E. somewhere in the column above an empty cell is a full cell. This is a negative feature because in order to fill it and remove its corresponding row, all rows corresponding to the full cells above the hole must be cleared first. In other words, a hole in the Tetris board causes a build up of tetrominos and that is unfortunate in terms of prolonging the game. This is feature which is usually weighted negatively.

Thiery and Scherrer have previously created a summary of feature functions found in existing literature. Their findings with some additions of our own can be seen in table 2.1. Unfortunately, many publications lack any formal definition of their selected features functions and some Tetris expressions are not used consistently in the literature. For instance, *wells* are usually described as a local board configuration where only the longest tetromino can be placed. It becomes clear that this description is somewhat ambiguous when attempting to formally define it. Figure 2.2 shows three different wells, where each requires a more relaxed definition than the previous. Our interpretation of each mentioned feature function, except two, (due to lack of documentation) can be found in the appendix.

Consequently, building a Tetris controller amounts to selecting or inventing feature functions, tuning the feature functions relative weights and implementing a simulator application capable of taking strategies as input. Selecting feature functions is typically done by a person that is well-versed in Tetris and has good domain knowledge. However, it has previously been suggested [4] to make part of the search space the problem of combining "basic features" into high level features that describes a relevant characteristic of the Tetris board. To this day, no one seems to have attempted this. Tuning the feature functions relative weights *can* be done manually as well [1], but is usually done using some kind of optimization method [5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 4, 15, 16].

A simulator application capable of taking strategies as input and playing games of Tetris was implemented early in the project. Such application is only required





**Figure 2.2:** Each Tetris board depicts a well, but each well requires a more relaxed definition than that of the previous board. The first and left-most instance is a typical well. The second instance is a well where the pattern does not persist downwards. The third instance is a well that is covered by a full cell.

to adhere to the specifications of Tetris and its research aspect is limited. Section 2.3, 2.4, 2.5 and 2.6 refers to *early experiments* performed using strategies from related work and our simulator application. An example of a game simulated using our application can be found in the appendix.

**Table 2.1:** A summary of feature functions mentioned in existing literature. The data was originally collected and constructed by Thiery and Scherrer [17] and is shown with some additions of our own. Note that the 33rd feature function was only used by Thiery and Scherrer during the 2008 Reinforcement Learning Competition and not in the controller referred to in this report.

FEATURE	DESCRIPTION	Lippmann <i>et al.</i> , 1993	Tsitsiklis and van Roy, 1996	Bertsekas and Tsitsiklis, 1996	Kakade, 2001	Lagoudakis <i>et al.</i> , 2002	Dellacherie (Fahey, 2003)	Fahey, 2003	Ramon and Driessens, 2004	Lilima, 2005	Böhm <i>et al.</i> , 2005	Farias and van Roy, 2006	Szita and Lőrincz, 2006	Thiery and Scherrer, 2009	Langenhoven, 2010	Scherrer, 2013
1. Jags	Variability in the contour formed by the tops of all pieces (not precisely documented)	x														
2. Max height	Maximum height of a column	x	x	x	x	x			x	x	x	x	x		x	x
3. Holes	Number of empty cells covered by a full cell	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
4. Column height	Height of each column			x	x				x			x	x			x
5. Column difference	Height difference between each pair of adjacent columns			x	x				x			x	x			x
6. Landing height	The position along the vertical axis of which the last piece landed						x			x	x			x	x	
7. Cell transitions	The number of empty cells/borders touching the edges of full cells									x						
8. Deep wells	Sum of well depths, except for wells with depth 1									x						
9. Embedded holes	Sort of weighted sum of holes (not precisely documented)									x						
10. Height differences	Sum of the height differences between adjacent columns					x										
11. Mean height	Mean height of columns					x			x							
12. $\Delta$ max height	Variation of the maximum column height					x										
13. $\Delta$ holes	Variation of the hole number					x										
14. $\Delta$ height differences	Variation of the sum of height differences					x										
15. $\Delta$ mean height	Variation of the mean column height					x										
16. Removed lines	Number of lines cleared during the last move	x				x		x			x				x	
17. Height weighted cells	Full cells weighted by their height							x			x				x	
18. Wells	Sum of the depth of the wells							x	x		x				x	

Continued on next page

Table 2.1 – continued from previous page

FEATURE	DESCRIPTION	Lippmann <i>et al.</i> , 1993	Tsitsiklis and van Roy, 1996	Bertsekas and Tsitsiklis, 1996	Kakade, 2001	Lagoudakis <i>et al.</i> , 2002	Dellacherie (Fahey, 2003)	Fahey, 2003	Ramon and Driessens, 2004	Lima, 2005	Böhm <i>et al.</i> , 2005	Farias and van Roy, 2006	Szita and Lőrincz, 2006	Thiery and Scherrer, 2009	Langenhoven, 2010	Scherrer, 2013
19. Full cells	Number of occupied cells on the board							x			x					x
20. Eroded piece cells	(Number of lines cleared during the last move) × (Number of cells cleared from the last tetromino)						x							x	x	
21. Row transitions	Number of horizontal cell transitions						x				x			x	x	
22. Column transitions	Number of vertical cell transitions						x				x			x	x	
23. Cumulative wells	$\sum_{w \in wells} (1 + 2 + \dots + depth(w))$						x							x		
24. Min height	Minimum height of a column								x							
25. Max - mean height									x							
26. Mean - min height									x							
27. Mean hole depth	Mean depth of all holes								x							
28. Max height difference	Maximum difference of height between two columns										x					x
29. Adjacent column holes	Number of holes, where adjacent holes in the same column count only once										x					x
30. Maximum well depth	Maximum depth of a well										x					x
31. Hole depth	Number of full cells in the column above each hole														x	
32. Rows with holes	Number of rows having at least one hole														x	
33. Pattern diversity	Number of different transition patterns between adjacent columns														x	

## 2.2 Related work

Every related work mentioned in this report employ a controller such as that described in the previous chapter. *Score* is consistently defined as the number of lines cleared during a game. The works vary primarily in which features to utilize and how to optimize their relative weights. However, some works stand out as something different, such as that of Lippmann *et al.* [5]. Furthermore, a summary of all related work, their choice of feature functions, optimization method and obtained score can be found in table 2.2.

Lippmann *et al.* created a framework for pattern classification with neural networks, machine learning and statistics. They successfully constructed a neural network capable of comparing two moves, which together with a preference network was trained by observing another human player. The controller was then able to mimic the strategies exhibited by the player it had observed. If the human player consistently played good moves, the network would gradually become better and better. They reported a score of 18 after 50 pieces had fallen, which is understandably lower than other controllers, as this would never become any better than the human it observed.

Dellacherie and Fahey's Tetris controllers (one-piece and two-piece controllers, respectively) are notable for performing extremely well despite being optimized manually, I.E. they used no optimization method to tune the relative weights of the feature functions. Dellacherie also seems to be the only one who have considered the original implementation of Tetris without the usual simplifications made. This is a significantly harder domain, as it does not permit all moves otherwise considered in the typically simplified domain [17].

Several controllers have been realized using methods of reinforcement learning. Methods include Value Iteration [6],  $\lambda$ -Policy Iteration [7, 16], Natural Policy Gradient [8], Least-Squares Policy Iteration [9], RRL-KBR [10] and Linear Programming [13]. None of the controllers optimized using these methods perform particularly well. This may be due to the nature of RL methods, in which the weights are tuned such that the fitness function approximates the optimal *expected* score from any given state.

Lastly, some controllers utilize more general purpose optimization methods. Llima [11] created a controller of which the weights was tuned by a genetic algorithm. He reports using a population of 50 individuals (corresponding to a set of coefficients) over 18 generations. The evolution took 500 CPU-hours, distributed over 20 workstations. The controller obtained a score of about 50,000 on average. Böhm [12] also utilized a genetic algorithm, but unfortunately does not report any averaged results due to running time reasons. Following that, Szita and

Lörincz [14] created a controller using Noisy Cross-Entropy. This is where it gets interesting, as they report obtaining a score of 350,000. Thiery and Scherrer [4] improved this approach by using the feature functions of Dellacherie along with two of their own. They report obtaining a striking score of 35,000,000, putting them on the top. Last in the category of general purpose optimization methods is Langenhoven [15], which applied Particle Swarm Optimization to train a neural network. Initially, he tried using a network with hidden nodes, but experiments showed that zero hidden nodes yielded highest score. The network without hidden nodes simply amounted to a weighted sum of each feature function.

## 2.3 Difficulty of comparing controllers

The previous section summarized the reported scores, optimization methods and choice of feature functions of fifteen different Tetris controllers. However, it turns out that comparing the scores is somewhat difficult for couple of reasons described in the next sections.

### 2.3.1 Domain variations

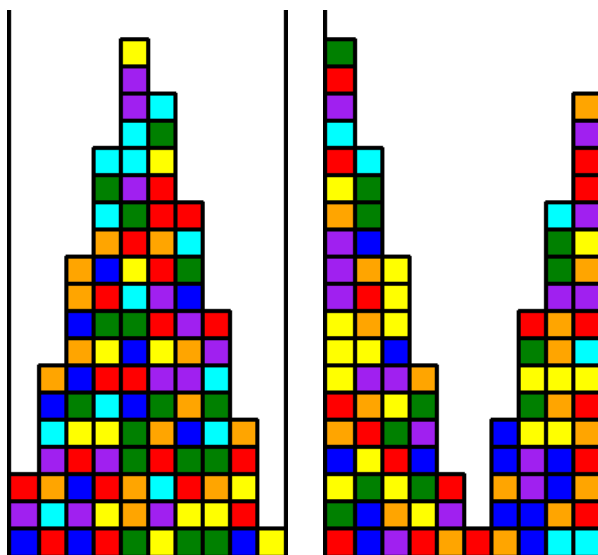
In the original game of Tetris, as specified by Fahey [1], tetrominos appear at the center of the top edge of the board. The game ends if the tetromino does not have sufficient space to enter the board. This property encourages a player to keep that area clear secondary to clearing lines. Figure 2.3 illustrates one particularly bad board and a conceivably better board, respectively, in terms of this particular aspect.

In the interest of focusing on the problem at hand, namely creating an agent that can choose a sensible orientation and position for the next tetromino, most researchers considers a simplified version of Tetris. This version of Tetris deviates from the original game in that the tetrominos are dropped directly in the column decided upon. This allows some actions that otherwise would not be possible and hence, the game is somewhat easier.

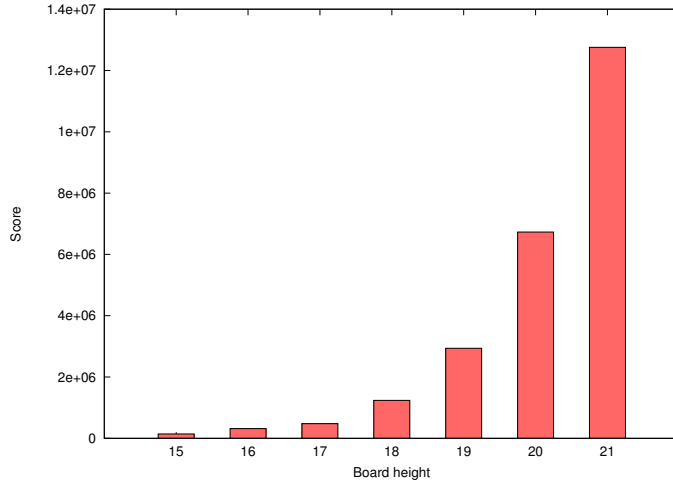
Dellacherie’s controller distinguishes itself from all other related work by adhering to the original specification. The controller performs significantly better than all controllers before its time. Thiery and Scherrer [17] reverse-engineered the algorithm and ran it in the simplified setting previously described. They report a score of 5,200,000. We also did this, but witnessed an average score of 6,700,000. To this time the reason for the differing performance reports remains unknown.

AUTHOR(S)	YEAR	OPT. METHOD	FEATURES	CONTROLLER TYPE	SCORE
Lippmann <i>et al.</i> [5]	1993	N/A	1-3, 16	one-piece controller	N/A
Tsitsiklis and van Roy [6]	1996	Value Iteration	2-3	one-piece controller	31
Bertsekas and Tsitsiklis [7]	1996	$\lambda$ -Policy Iteration	2-5	one-piece controller	3,200
Kakade [8]	2001	Natural Policy Gradient	2-5	one-piece controller	6,800
Lagoudakis <i>et al.</i> [9]	2002	Least-Squares Policy Iteration	2-3, 10-16	one-piece controller	1,000-3,000
Dellacherie [1]	2003	<i>manual</i>	3, 6, 20-23	one-piece controller	650,000
Fahey [1]	2003	<i>manual</i>	3, 16-19	two-piece controller	7,200,000
Ramon and Driessens [10]	2004	RRL-KBR	2-5, 11, 18, 24-27	one-piece controller	50
Lima [11]	2005	Genetic Algorithm	2-3, 6-9	one-piece controller	50,000
Böhm <i>et al.</i> [12]	2005	Genetic Algorithm	2-3, 6, 16-19, 21-22, 28-30	two-piece controller	N/A
Farias and van Roy [13]	2006	Linear Programming	2-5	one-piece controller	4,700
Szita and Lőrincz [14]	2006	Noisy Cross-Entropy	2-5	one-piece controller	350,000
Thiery and Scherrer [4]	2009	Noisy Cross-Entropy	3, 6, 20-23, 31-32	one-piece controller	35,000,000
Langenhoven [15]	2010	PSO	2-3, 6, 16-22, 28-30	one-piece controller	275,000
Scherrer [16]	2013	$\lambda$ -Policy Iteration	2-5	one-piece controller	4,000

**Table 2.2:** A summary of related work and their results.




**Figure 2.3:** In the setting of the original game of Tetris, the left board would likely lead to game over, while the right board provides better chances of prolonging the game.



**Figure 2.4:** Average performance of Dellacherie’s controller on different board heights. The scores are measured over 100 games for each board height.

Bertsekas and Tsitsiklis [7] controller also cannot easily be compared to any other as they defined the game to be over when “a square in the top row becomes full and the top of the wall reaches the top of the grid.” We have previously defined game over as when a tetromino cannot be placed without *overflowing* the board. Assuming that our definition is the correct one, this means that they have considered Tetris on a  $10 \times 19$  board. Changing the height of the board greatly impacts the performance of any Tetris controller. Figure 2.4 illustrates average scores for Dellacherie’s controller in environments of different heights.

Another way a domain can vary is in how points are awarded. As previously mentioned, an implementation may award additional points for clearing multiple lines in the same move. As the maximum number of lines that can be cleared during one move is four and can only be accomplished by using , the formation of deep wells might actually be beneficial in such an environment, which it otherwise would not be. Hence, variation on the domain is significant because they may encourage a player to use a different strategy.

This dependency between optimal strategy and environment means that one cannot compare agents that have been optimized to perform in different Tetris domains. Simply putting an agent in a different environment may very well yield a completely different score. For instance, Böhm *et al.* [12] reports one score of over 480,000,000 for their two-piece controller, but our experiments indicate that



the agent only clears lines in the number of hundreds in a one-piece environment.

### 2.3.2 Large deviation in score

We initially ran 500 simulations using Thiery and Scherrer’s Tetris controller and plotted a histogram showing the frequency of scores, illustrated in figure 2.5. The figure shows that the score of a Tetris controller may deviate heavily between each game. Additionally, it shows that the score of a Tetris controller does not converge around a single, expected value. It does however strongly resemble a probability mass function of a geometric distribution. Thiery and Scherrer also argues [17] that it is reasonable to assume that the score follows a geometric distribution.

Unfortunately, most researchers does not provide a confidence interval with their results. Of the mentioned authors here, Szita and Lörincz [14] and Thiery and Scherrer [4] seems to be the only ones. Luckily, however, most researchers reports having performed 100 or more simulations in order to obtain their score, making them quite accurate. Thiery and Scherrer [4] has already explained how to calculate a confidence interval, but for completeness it is repeated below.

The difference of an average score of  $\hat{\mu}$  between  $N$  games and their expected score of  $\mu$  where the standard deviation  $\sigma$  is equal to the expected score  $\mu$  satisfies

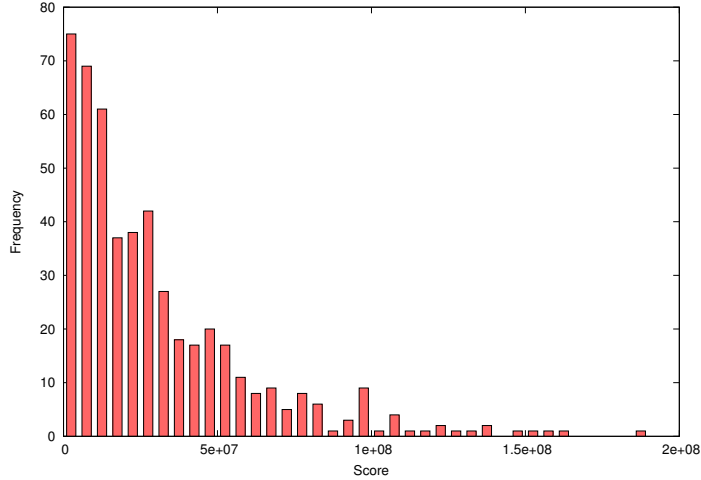
$$|\mu - \hat{\mu}| \leq \frac{k\sigma}{\sqrt{N}} = \frac{k\mu}{\sqrt{N}} \simeq \frac{k\hat{\mu}}{\sqrt{N}} \quad (2.1)$$

$$\frac{|\mu - \hat{\mu}|}{\hat{\mu}} \leq \frac{k}{\sqrt{N}} \quad (2.2)$$

where  $k$  is given by the chosen probability  $p$ . We have chosen  $p = 0.95$  for our confidence intervals and as such  $k = 2$ . This means that with 100 simulated games, the expected value is within  $\pm 2/\sqrt{100} = \pm 20\%$  of the observed average. During the rest of the report the notation  $m \pm c\%$  is used to denote an observed average of  $m$  with a confidence interval of  $\pm c\%$ .

### 2.3.3 Differing feature functions

Table 2.1 summarizes all feature functions mentioned in existing literature. It comes as no surprise that the choice of feature functions varies between almost every publication. As previously mentioned in the introduction, we hypothesize



**Figure 2.5:** A histogram showing frequency of scores of Thiery and Scherrer’s Tetris controller.

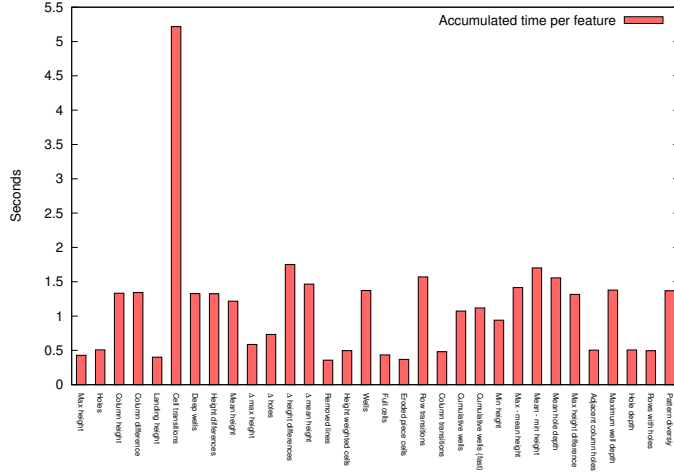
that the choice of feature function may greatly affect the performance of a Tetris controller.

Another way to look at it is that the choice of feature function may somewhat determine the *potential* of an optimization method. This matters when Tetris is used as an optimization application and the intention is to compare the results against other optimization methods. If our hypothesis is correct and it turns out that some feature functions are inherently bad, then the results of some optimization methods may be underrated.

## 2.4 Computational cost

The computational costs of running Tetris simulations are undoubtedly very high. Fahey [1] reports that his first and only game performed using his two-piece controller took 7.6 days. Szita and Lörincz [14] reports a total execution time of a month. Thiery and Scherrer [4] reports using a week to tune the parameters for their experiments, while the experiments itself took about a month. Lets take a closer look at where all this time gets spent during execution of a Tetris controller.

When a Tetris controller is served a new tetromino and is asked to place it on the






















**Figure 2.6:** Distribution of time spent during feature function evaluation of 1,000,000 different Tetris board configurations.

board, it needs to consider each possible placement and evaluate the new board that corresponds to that placement. Table 2.3 shows that the average number of possible placements for a tetromino is 23.14. That's 23.14 boards that needs to be evaluated for each falling tetromino. For the sake of the simplicity of these estimations, it is assumed that all placements are possible. This assumption is quite safe due to the fact that the max column height rarely surpasses 16, making the four topmost rows almost always empty.

Figure 2.6 shows how time is distributed amongst the feature functions when 1,000,000 different boards are evaluated for each type of feature. These experiments was run on an Intel Core 2 Duo 2.40GHz CPU. Since the execution time of a feature function may vary between different board configurations, evaluations was done on the first 1,000,000 board configurations of a simulated game using Dellacherie's strategy. Total cumulated execution time was 36.08 seconds, which gives us an average execution time of 1.1276 microseconds per feature function.

Since each tetromino always fill four cells on the board and each row is ten cells wide, one row must be cleared for every 2.5 tetrominos that are placed in order to sustain a game of Tetris. This means that the average score of 35,000,000 involves placing 87,500,000 tetrominos. Given the average number of placements to consider, average execution time of evaluation functions and assuming that a typical strategy consist of six feature functions, we can estimate the time it takes

Tetrominos	Possible placements on a board
	9
 	(7 + 10 =) 17
 	(8 + 9 =) 17
 	(8 + 9 =) 17
   	(8 + 9 + 8 + 9 =) 34
   	(8 + 9 + 8 + 9 =) 34
   	(8 + 9 + 8 + 9 =) 34

**Table 2.3:** The number of possible ways one can place each tetromino on an empty 10 by 20 Tetris board. The average number is 23.14.

to play such a such game.

$$87.5M \cdot 23.14 \cdot 1.1276\mu s \cdot 6 = 3 \text{ hours } 48 \text{ minutes } 18.65 \text{ seconds} \quad (2.3)$$

Tests show that 93% of execution time during a simulated game is spent during feature function evaluations <sup>1</sup>. With just two pieces known, the number of boards (corresponding to each combination of placements) to consider increases to  $23.14^2 = 535.45$ . This effectively means that the 93% of execution time spent on feature function evaluation is multiplied by 23.14. It becomes clear that increasing lookahead will greatly reduce performance (in terms of execution speed) of any Tetris controller.

Unfortunately, there is no easy way to reduce the computational costs of simulating a game of Tetris. It is not parallelizable in any obvious way and no one reports having attempted that. Reducing the computational costs can therefore only be achieved by optimizing the application itself. However, all optimization algorithms mentioned so far requires simulating multiple games and multiple simulations *are* parallelizable, as they do not depend on each other. For this project, we were given 150,000 CPU hours on NTNUs supercomputer *Vilje*, which allowed us to perform the necessary computations. See section 3.4.7 for a more detailed description regarding parallelization.

<sup>1</sup>Application profiling was done using *perf*: Linux profiling with performance counters.

It should be mentioned for completeness that while Langenhoven [15] points out that genetic algorithms have been used to optimize the speed of programs evolved to play Tetris [18], it is not applicable to any type of Tetris controller used in the mentioned works, nor theirs. The referred publication considers an agent with variable decision time in terms of placing a tetromino, whereas the controllers described here utilize a weighted sum of values to sort board configurations, which is executed in constant time.

## 2.5 Application optimization

Initial experiments using Tetris strategies from related publications indicated that our application for simulating games of Tetris was only able to play around 7,000 moves per second during a single simulated game on an Intel Core 2 Duo 2.40GHz CPU. This was significantly lower than the reported performance of Thiery and Scherrer's controller of between 50,000 and 100,000 moves per second [4]. This was a motivator for optimizing the application.

The application was optimized by changing the way a Tetris board was represented internally in the software. Initially, a board was represented using  $10 \times 20$  integer variables (where the size was configurable). When calculating E.G. the number of holes on the board, each column would be iterated across and considered on its own and the results would be summed together.

After optimization, the board was represented using 20 16-bit integers variables, where 10 bits is used to represent the state of one row, where each cell can either be empty or full. This type of representation allowed us to calculate the number of holes in each column at the same time by utilizing bitwise operations. In other words, the number of holes in each column could be calculated simultaneously instead of considering each column on its own. Figure 2.7 shows how holes are calculated using bitwise operations.

As a result of the optimization, the application was able to play 30,000 moves per second during a single simulated game on the previously mentioned hardware. This amounts to a performance increase of 328%.

## 2.6 Estimating performance of a strategy

In the interest of reducing the computational costs of simulating Tetris games, it has been suggested [1] that the performance of a strategy can in fact be estimated by sampling data from an unfinished game. Figure 2.8 shows how data from the

```

int f_n_holes (struct board * board) {
    int n_holes = 0;

    uint16_t row_holes = 0x0000,
             previous_row = 0x0000;

    for (int y = 0; y < BOARD_HEIGHT; y++) {
        // A cell is a hole if it is empty (~full) and the cell
        // above is full or already determined to be a hole.
        row_holes = ~board->lines[y] & (previous_row | row_holes);

        n_holes += full_cells_on_line[row_holes];

        previous_row = board->lines[y];
    }

    return n_holes;
}

```

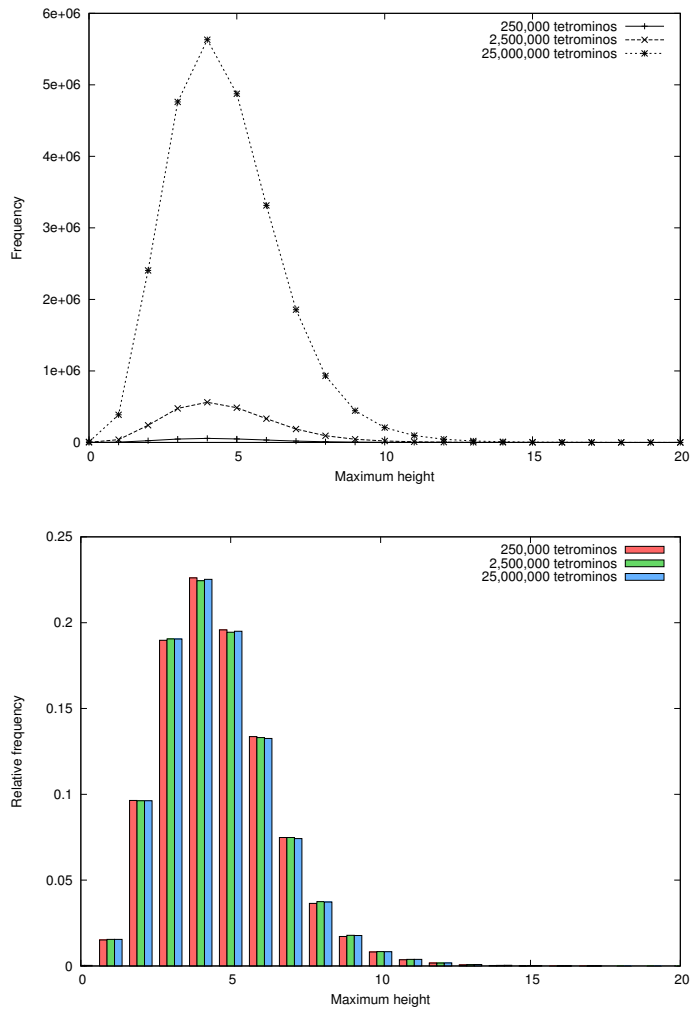
**Figure 2.7:** The method (simplified for clarity) for calculating the number of holes on a Tetris board using bitwise operations.

beginning of the game and spanning a small time frame compares to that of a larger time frame.

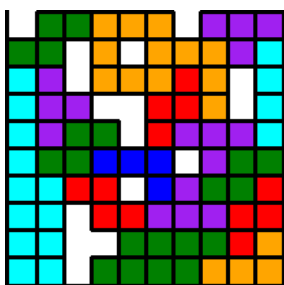
The left chart illustrates a histogram where each bin corresponds to the number of tetromino placements that resulted in a particular maximum column height (after full lines are cleared), hereby denoted as  $h$ . The three plots are of the first 250,000, 2,500,000 and 25,000,000 placed tetrominos during a simulated game using Dellacherie’s strategy. The right chart is a normalized version of the same histogram.

The normalized plots are practically identical and shows a certain tendency of convergence. This strongly indicates that the maximum height of a board can be expressed as a random variable with some expected value and distribution. We denote the random variable as  $X$  and  $P(X = n)$  as the frequency of  $h = n$ . Fahey [1] hypothesized that the tail of the plots has an exponential distribution. Thiery and Scherrer [17] however suggests that the score follows a geometric distribution.


Their theory is that by deciding upon a distribution, one can use regression to estimate the parameters of it. With the distribution known as a function, one can calculate  $P(X = 21)$  (which is when the game of Tetris ends) and thus estimate the expected duration of a game by only playing some  $n$  tetrominos. Unfortunately, neither Fahey nor Thiery and Scherrer has experienced success in their attempts of estimating performance of strategies.



**Figure 2.8:** Frequencies of maximum column height during the first  $n$  moves of a simulated game using Dellacherie's strategy, where  $n$  is 250,000, 2,500,000 and 25,000,000. The bottom chart shows the scaled plots of the top chart.



**Figure 2.9:** A relatively full Tetris board that is certainly doomed to fail during placement of the next tetromino.

However,  $P(X = 21)$  does not include *all* possible ways that a game of Tetris can end. Some board configurations will necessarily result in a board height of at least  $h = 22$  for a given falling tetromino. An example of such board configuration is shown in figure 2.9. Placing  will result in either  $h = 22$  or  $h = 23$ . Hence, both  $P(X = 22)$  and  $P(X = 23)$  must somehow be factored in. We have not investigated this any further.



# Chapter 3

## Methodology

### 3.1 Inspiration

Genetic algorithms are a subset of evolutionary computation, a field of computing that is largely inspired by Charles Darwin's theory of evolution [19] and his explanation of biological diversity in nature and the underlying mechanisms that they are a result of. He explained that in a limited environment that can only host a certain number of individuals and where each individual has an instinctual desire to reproduce, *natural selection* will inevitably occur because of the competition that arise.

Natural selection is the process by which phenotypic traits (the physical and behavioral characteristics that an individual exhibits in a population) change and become more or less common as a result of the effect that they have on the reproductive success of the holding individuals. Good traits, in terms of their effect on the ability to create offspring, will over time dominate a population and bad traits will disappear.

Darwin suggested that phenotypic traits are inherited, but that tiny, random variations occur during reproduction. This gives rise to new combinations of traits and with bad combinations being lost due to natural selection, evolution of the population will progress. With time, it will become more and more adapted to fit in its environment. This is sometimes called macroevolution.

Modern molecular genetics offers another perspective on natural evolution and helps to explain what happens on a lower level. The main observation to make is that every individual can be seen as containing both a genotype and a phenotype,

where the genotype encodes the phenotype. *Genes* are the inheritable units and evolutionary mechanisms such as recombination and mutation that occurs during reproduction works on the genome of an individual. This is sometimes called microevolution.

## 3.2 Why evolutionary computing?

Evolution occurring in nature has proven itself to be a powerful problem solver and has come up with vastly complex solutions such the human brain. Methods of evolutionary computing has previously been employed to a wide variety of problems and often with good results.

One example of a practical problem where an evolutionary algorithm has been successfully applied is the design of NASA's ST5 spacecraft antenna, by Globus et al. [20]. The evolved antenna was superior to the conventional antenna design in regard to power consumption, fabrication time, complexity and performance.

Another example of a similarly successful employment of an evolutionary algorithm is the satellite dish boom holder by Keane and Brown [21]. This ladder connects a satellite to the communication dish. The ladder must dampen vibrations as there is no air resistance in space to do it otherwise. They report that the evolved solution performed 20,000% better than traditional constructions.

Both of these examples are particularly remarkable. The applications of both examples, namely radio antennas and satellite dish booms, is something that traditionally has been crafted by hand and required extensive domain knowledge. This naturally encourages some kind of convention, which is in itself not a good driving force towards quality. They both illustrates the power of the evolutionary approach and its ability to devise seemingly *random, but better* solutions. It does so without any intelligence, but being purely driven by quality and not limited by convention, aesthetic considerations or human thinking and our preference for symmetry.

## 3.3 What is an evolutionary algorithm?

There are many variants of evolutionary algorithms, where some of the most notable ones are genetic algorithms (GA), genetic programming (GP), evolutionary programming (EP) and evolution strategy (ES). However, these are all very similar in that they share the underlying idea of Darwin's theory of evolution: given

```

population = initialize_random_candidate_solutions
until termination_condition_met? do
  parents = select_parents(population)
  children = recombine_pairs_of_parents(parents)

  mutate_children(children)
  evaluate_children(children)

  population = select_surviving_children(children)
end

```

**Figure 3.1:** Pseudo code showing the scheme of a typical evolutionary algorithm.

that a population is restricted by limitation of resources, competition will arise and natural selection will occur.

In a typical evolutionary algorithm, a population of random candidate solutions are initialized. A fitness function is then applied to each candidate. The candidates that are better in terms of fitness value is used to seed the next generation. Seeding typically involves a recombination function that takes multiple candidates as input, as well as a mutation function. This process is often repeated until a targeted solution is achieved or a computational limit is reached. This scheme is outlined with pseudo code in figure 3.1.

There are especially three features that serves as extra important when constructing an evolutionary algorithm:

- Problem and solution representation. The method of representing solutions must be capable of representing a broad spectrum of the actual solution space, as to not lose possibly good solutions.
- Recombination and mutation functions. These must be able create and maintain a certain degree of diversity in the solutions represented by the population.
- Parent and survival selection methods. The selection mechanisms must ensure that the quality of the population increases over time and progresses towards the goal.

## 3.4 Implementation

The genetic algorithm and all subsequent methods for simulating games of Tetris was written in C <sup>1</sup>.

### 3.4.1 Representation

The genotype consist of one sequences of integers. The values encodes the weights for each feature function. Since each feature function can have multiple weights associated with it (such as 4. and 5., see table 2.1), the length of this sequence is  $\sum^{n_f} n_{w,i}$ , where  $n_f$  is the number of features that are chosen and  $n_{w,i}$  is the number of weights of feature  $i$ .

### 3.4.2 Fitness function

As previously mentioned, the genotype of an individual encodes several integer values. These values serves as a candidate solution to the given problem (namely the game of Tetris). The fitness of the individual is determined by the average number of lines cleared during a configurable number of  $n_t$  trial games using its encoded values for feature weights..

### 3.4.3 Selection

Three different types of selection mechanism was implemented: stochastic universal sampling (a development of fitness proportionate selection), tournament selection and a modification of Goldberg’s sigma scaling, where an individual’s modified fitness value is given by

$$f'(i, g) = 1 + \frac{f(i) - f(\bar{g})}{2\sigma(g)} \quad (3.1)$$

where  $f(i)$  is the original fitness of individual  $i$ ,  $f(\bar{g})$  is the average fitness value of the individuals in population  $g$  and  $\sigma(g)$  is the standard deviation of the population’s fitness. There is no survivor selection, only full generational replacement.

---

<sup>1</sup>The source code is available on Github under the MIT license: <https://github.com/badeball/tetris-ea>.

For the tournament selection mechanism, a group of  $t_n$  individuals is randomly selected from the population and the individual with highest fitness value is selected to become a parent. Additionally, with a probability of  $t_p$ , a random individual will be selected from the tournament group instead of the best individual.

#### 3.4.4 Elitism

Elitism was implemented, where a configurable number of  $n_e$  of the best individuals of a generation is automatically transferred to the next generation without competing with the other individuals. The purpose of this feature in an evolutionary algorithm is to not lose good genetic traits due to random events.

The implications are however twofold. By keeping good individuals, evolution will consistently evolve better individuals. On the other side, by always sticking with previously evolved good solution, it will lose some of its ability to search for solutions outside of the locality of its total genetic diversity.

#### 3.4.5 Crossover

Both N-point crossover and uniform crossover was implemented. For each crossover operation, the genotypes are considered to consist of  $n_f$  pieces, meaning that weights that are associated to the same feature are transferred together from the same individual. Crossover between two parents will only occur with a probability  $p_c$ . In case where crossover does not occur, one random parent is selected and copied to the next generation.

#### 3.4.6 Mutation

For mutation, the genotype is iterated across and each allele is randomly modified with a probability  $p_m$ . In case of mutation, the allele is adjusted with a random number in a configurable range  $m_r$ . Hence, the values can *drift* and obtain large relative weights by chance.

#### 3.4.7 Parallelization

In every variant of evolutionary algorithm mentioned above, many candidate solutions are created during execution: firstly by random initialization, then by recombination and mutation. A fitness function usually must be applied

to each individual in order to perform some kind of selection. In some cases, including our own, applying the fitness function to each candidate solution can be done independently. This allows for some parallelization. In our case, almost all execution time is spent during fitness evaluation and as such, parallelization yields a large performance boost.

Parallelization was performed using Message Passing Interface (MPI). With MPI, the same program is spawned multiple times across nodes of processors and each process is given a unique rank ranging from zero and upwards. With our program, every process assumes a role based on their rank: the zero-ranked process is tasked with initializing and maintaining a population of candidate solutions, and every other process simply listens for requests to calculate fitness value of candidate solutions.

Our simulations was executed on NTNU's supercomputer *Vilje*. This computer cluster is composed of 1404 nodes running SUSE Linux Enterprise Server 11, whereas each node consists of 2 Intel Xeon E5-2670 CPUs (2.6 GHz, 8 cores with hyper-threading). A typical execution of the program was performed using 4 nodes and 32 processes was spawned on each node.

### 3.4.8 Random values

Early experiments quickly uncovered issues with our choice of random number generator. Initially, our source of random numbers was just a local implementation of a linear congruential generator. This was seeded using system time added with a multiple of the process rank. However, these linearly correlated seed numbers proved to be inefficient. This was solved by using GNU Scientific Library and seeding it using a hash function of system time and process identifier, as suggested by Katzgraber [22] and illustrated below.

```
int seed () {
    int s, pid;

    s = time(NULL);
    pid = getpid();

    return abs(((s * 181) * ((pid - 83) * 359)) % 104729);
}
```

# Chapter 4

## Results

### 4.1 Comparing environments

In order to test our hypothesis described in the introduction, the evolutionary algorithm was executed in each environment described in all mentioned publications (hereby referred to as just the *environments*). Nearly all the mentioned works utilize completely different sets of feature functions, except for Bertsekas and Tsitsiklis [7], Kakade [8], Farias and van Roy [13], Szita and Lörincz [14] and Scherrer [16], which all have chosen 2-5. However, Bertsekas and Tsitsiklis [7] utilize a definition of game-over which differs from the rest and consequently their result is not trivially comparable to the rest.

Due to time constraints during the project period, we were not able to implement the original specifications of the Tetris game as described by Fahey [1]. A conservative approximation to these specifications is a game of board height of only 16 tiles. Any action possible on such a board with the simplified setting would also be possible on a board of full height with the original specification, as no tetromino is higher than 4 tiles. This approximation was used when considering the environment described by Dellacherie. We were also not able to execute our algorithm in any two-piece environment for the same reason.

Each evaluation was run using the parameters shown in table 4.1, which was found to yield good results. Figure 4.2, 4.3, 4.4, 4.5, 4.6, 4.8, 4.9, 4.10 and 4.11 shows the results of evolution within each environment. The top illustrations depict the progress of evolution, showing the score of the worst and best individual of each generation, as well as the average score and the 25% and 75% quartile. The

	PARAMETER	VALUE
$n_e$	elitism	2
$n_p$	population size	100
$n_g$	generation limit	50
$n_c$	crossover points	2
	selection mechanism	tournament
$t_n$	tournament group size	10
$t_p$	tournament random selection rate	0.1
$p_m$	mutation rate	$1/l$
$p_c$	crossover rate	0.5
$n_t$	number of trials	100
$m_r$	adjusting mutation range	$[-100, 100]$

**Table 4.1:** Parameters found to yield good results and subsequently used in the experiments.  $l$  is the number of integers contained in a genotype, described in section 3.4.1 Representation.

bottom illustrations shows the performance of the resulting evolved controller. 400 evaluations was performed to obtain a confidence interval of  $\pm 10\%$  with probability 0.95. The numbers are summarized in table 4.2.

## 4.2 Unbiased feature selection

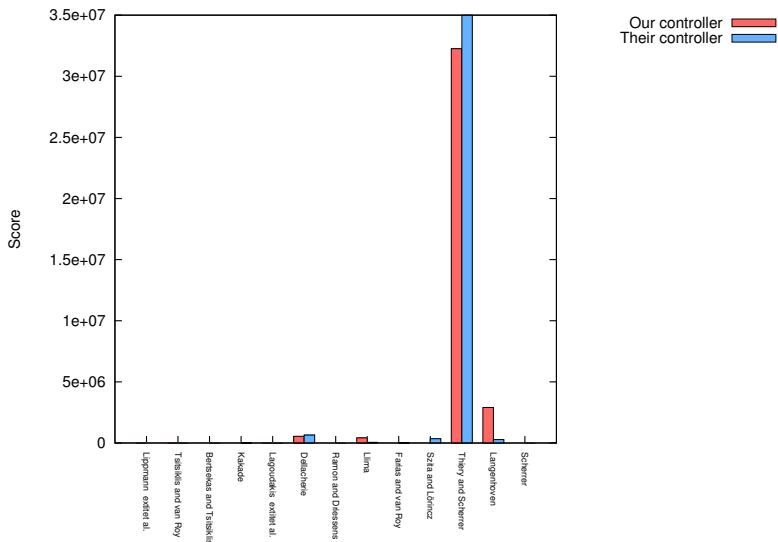
The same genetic algorithm was run in an environment were it could freely choose between every feature function. The "enabled status" of a feature function would be randomly initialized upon execution and would be subject to mutation during evolution. The status bits had the same probability of being mutated as a feature function weight and both possible values (true and false) was equally probable. The mutation rate  $p_m$  was configured to 0.0625 ( $= 2/32$ ) and all other parameters remained the same.

Figure 4.12 shows the result of evolution within this environment. The final controller obtained a score of 14,690,265 and consisted of the weights shown in table 4.3.



COMPETITORS(S)	FEATURES	THEIR SCORE	OUR SCORE	EVOLVED WEIGHTS
Lippmann <i>et al.</i> [5]	1-3, 16	N/A	42 ± 10%	(0*, -44, -3, 75)
Tsitsiklis and van Roy [6]	2-3	31	40 ± 10%	(-45, -33)
Bertsekas and Tsitsiklis [7]	2-5	3,200	2 ± 10%	(7, 37, 16, 42, -44, -25, 31, -91, -24, 4, -3, -43, 50, -13, -37, 10, -36, -30, -22, -27, -43)
Kakade [8]	2-5	6,800	0 ± 10%	(-15, 9, 64, 6, 27, -6, -49, -6, -12, 18, 40, -11, -37, -19, 31, 1, 27, -44, 61, 32, 41)
Lagoudakis <i>et al.</i> [9]	2-3, 10-16	1,000-3,000	1,967 ± 10%	(5, -44, -13, -82, 87, 40, -46, 32, -62)
Dellacherie [1]	3, 6, 20-23	650,000	546,377 ± 10%	(-41, -20, 14, -16, -50, -17)
Fahay [1]	3, 16-19	7,200,000	N/A	N/A/
Ramon and Driessens [10]	2-5, 11, 18, 24-27	50	0 ± 10%	(39, 41, 23, -15, -22, -17, -35, 87, 38, 37, 28, -15, 15, -29, 14, 21, -16, -40, 27, 5, -32, -20, -48, -15, 50, -1, -24)
Lima [11]	2-3, 6-9	50,000	418,944 ± 10%	(7, -90, -15, -17, -36, 0*)
Böhm <i>et al.</i> [12]	2-3, 6, 16-19, 21-22, 28-30	N/A	N/A	N/A
Farias and van Roy [13]	2-5	4,700	0 ± 10%	(-15, 9, 64, 6, 27, -6, -49, -6, -12, 18, 40, -11, -37, -19, 31, 1, 27, -44, 61, 32, 41)
Szita and Lőrincz [14]	2-5	350,000	0 ± 10%	(-15, 9, 64, 6, 27, -6, -49, -6, -12, 18, 40, -11, -37, -19, 31, 1, 27, -44, 61, 32, 41)
Thiery and Scherrer [4]	3, 6, 20-23, 31-32	35,000,000	32,254,604 ± 10%	(5, -33, 9, -20, -76, -31, -2, -65)
Langenhoven [15]	2-3, 6, 16-22, 28-30	275,000	2,905,737 ± 10%	(-60, -155, -46, 22, -2, -83, 14, 33, -72, -151, 45, -97, -32)
Scherrer [16]	2-5	4,000	0 ± 10%	(-15, 9, 64, 6, 27, -6, -49, -6, -12, 18, 40, -11, -37, -19, 31, 1, 27, -44, 61, 32, 41)

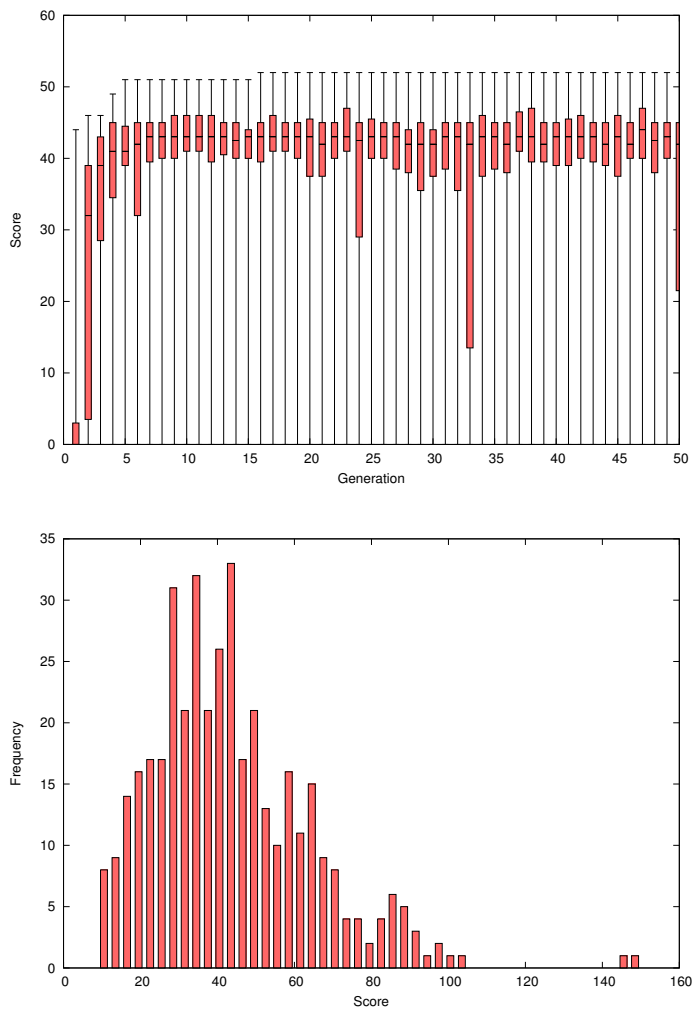
**Table 4.2:** A summary of our results side-by-side with the competitors. The marked (\*) weights are weights of feature functions that we were unable to implement due to lack of documentation.



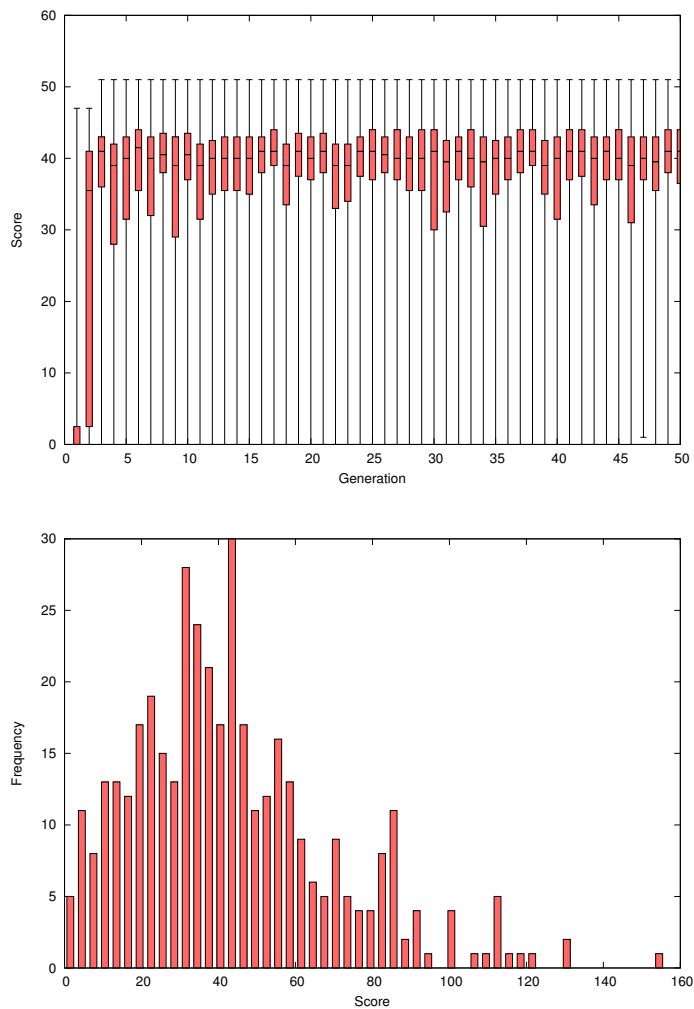
**Figure 4.1:** A comparison of the performances of our controller evolved in each environment and every other reported result.

	FEATURE	VALUE
3.	Holes	-35
6.	Landing height	-51
7.	Cell transitions	-46
8.	Deep wells	-12
10.	Height differences	19
11.	Mean height	6
12.	$\Delta$ max height	50
13.	$\Delta$ holes	25
15.	$\Delta$ mean height	17
18.	Wells	-19
21.	Row transitions	-38
23.	Cumulative wells	-42
24.	Min height	-41
26.	Mean - min height	-60
29.	Adjacent column holes	-155

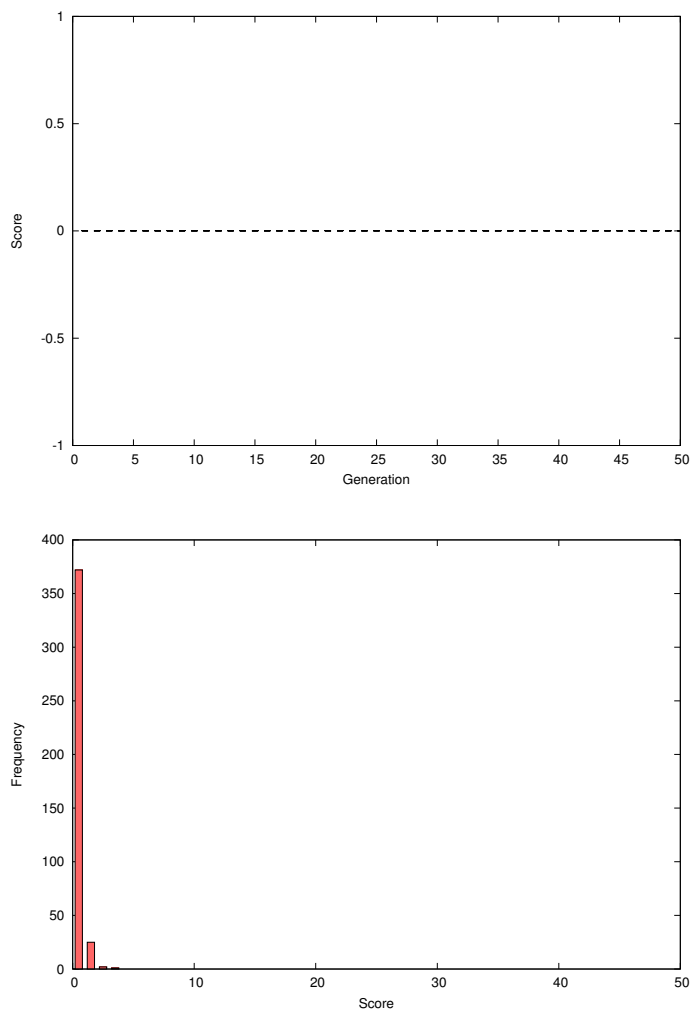
**Table 4.3:** The Tetris strategy evolved in an environment where the evolutionary algorithm could freely choose between every feature function.



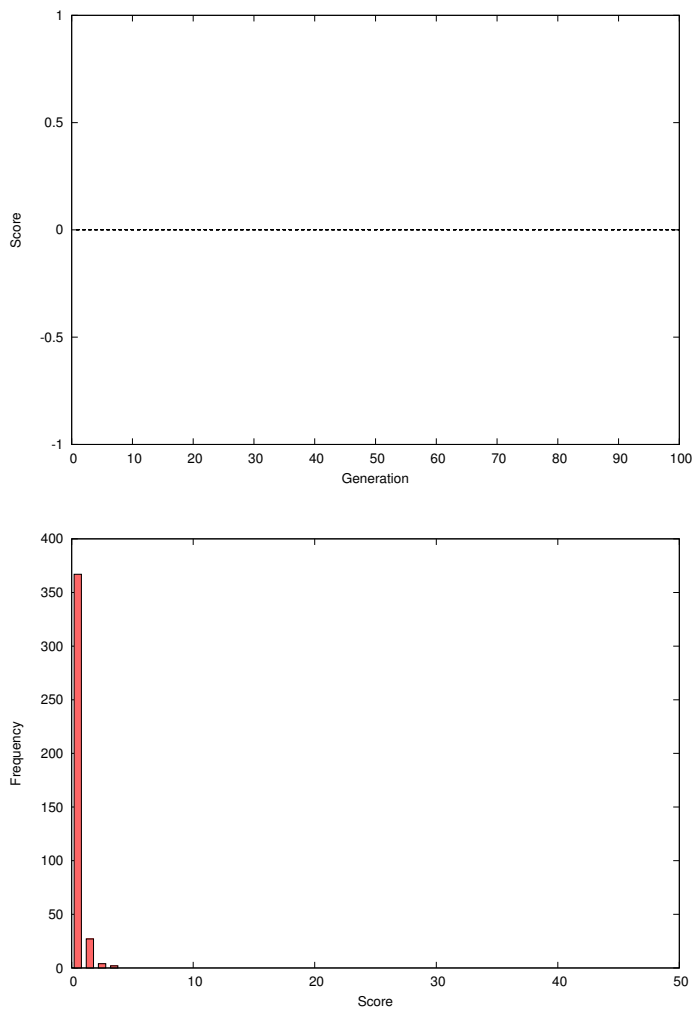
**Figure 4.2:** Progress during evolution using the same environment as Lippmann *et al.* [5] (top chart), and the performance of the resulting Tetris controller, illustrated with a histogram showing the frequency of scores (bottom chart).



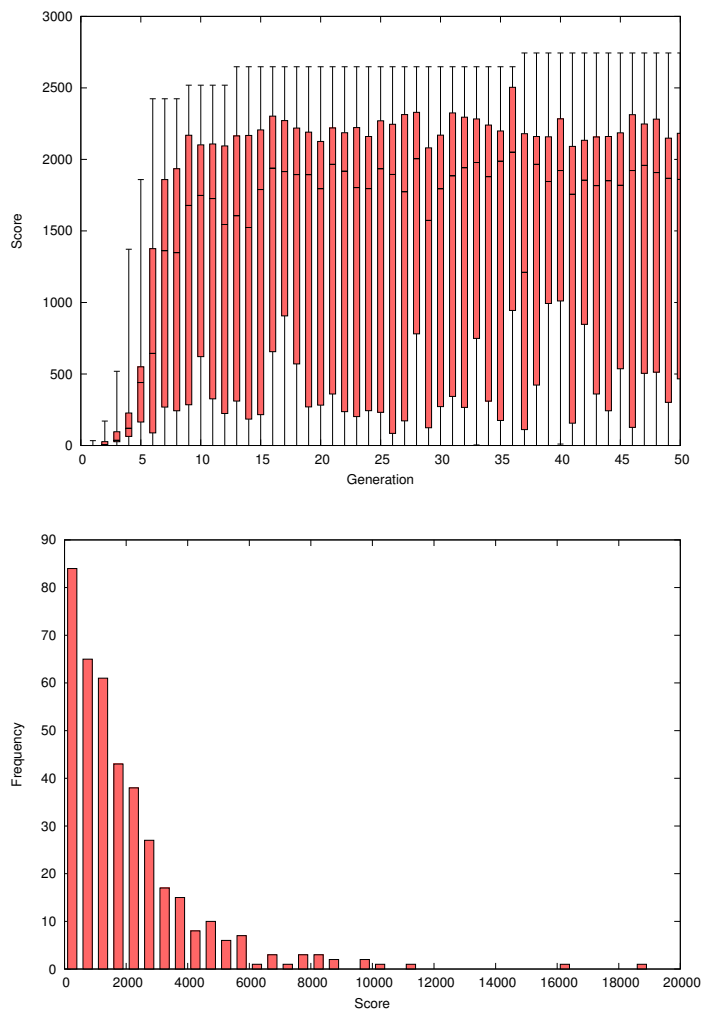
**Figure 4.3:** Progress during evolution using the same environment as Tsitsiklis and van Roy [6] (top chart), and the performance of the resulting Tetris controller, illustrated with a histogram showing the frequency of scores (bottom chart).



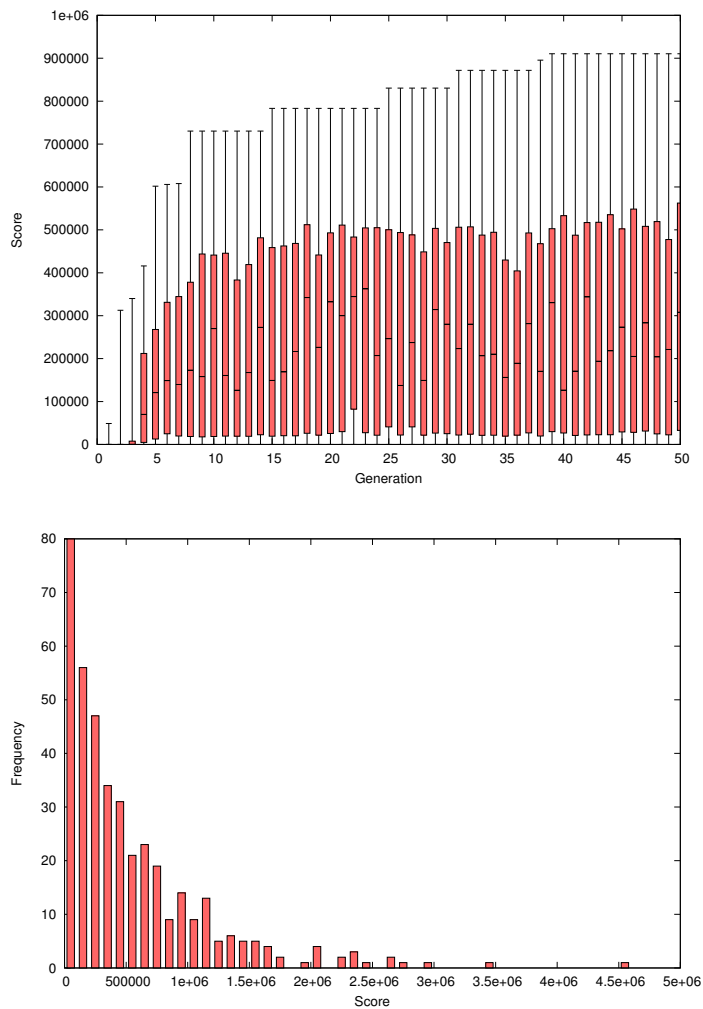
**Figure 4.4:** Progress during evolution using the same environment as Bertsekas and Tsitsiklis [7] (top chart), and the performance of the resulting Tetris controller, illustrated with a histogram showing the frequency of scores (bottom chart).



**Figure 4.5:** Progress during evolution using the same environment as Kakade [8] (top chart), and the performance of the resulting Tetris controller, illustrated with a histogram showing the frequency of scores (bottom chart).

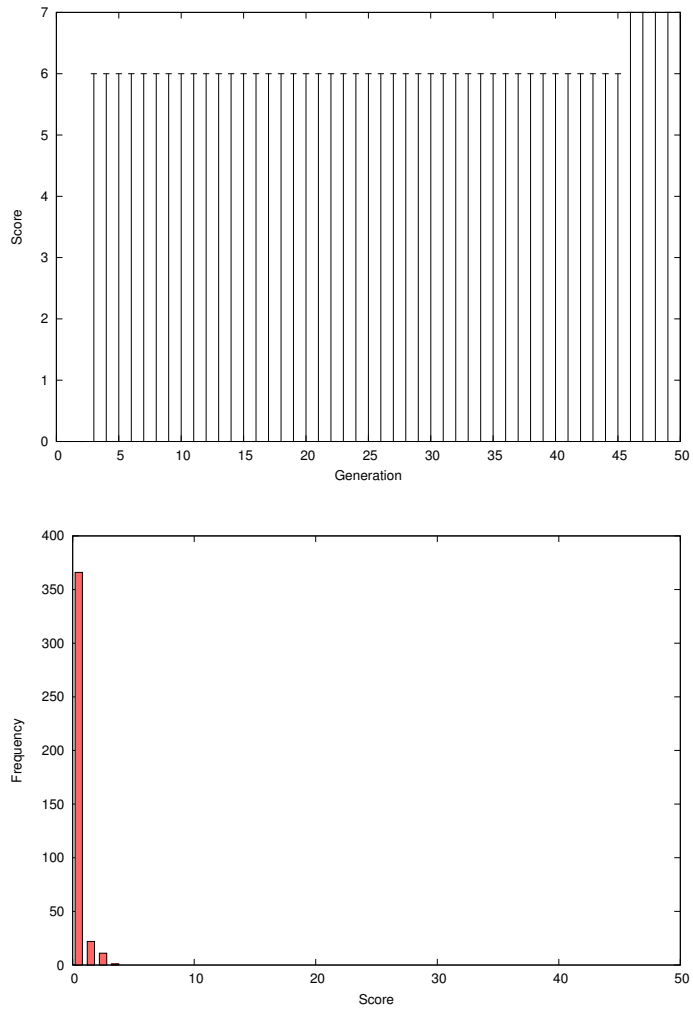


**Figure 4.6:** Progress during evolution using the same environment as Lagoudakis *et al.* [9] (top chart), and the performance of the resulting Tetris controller, illustrated with a histogram showing the frequency of scores (bottom chart).

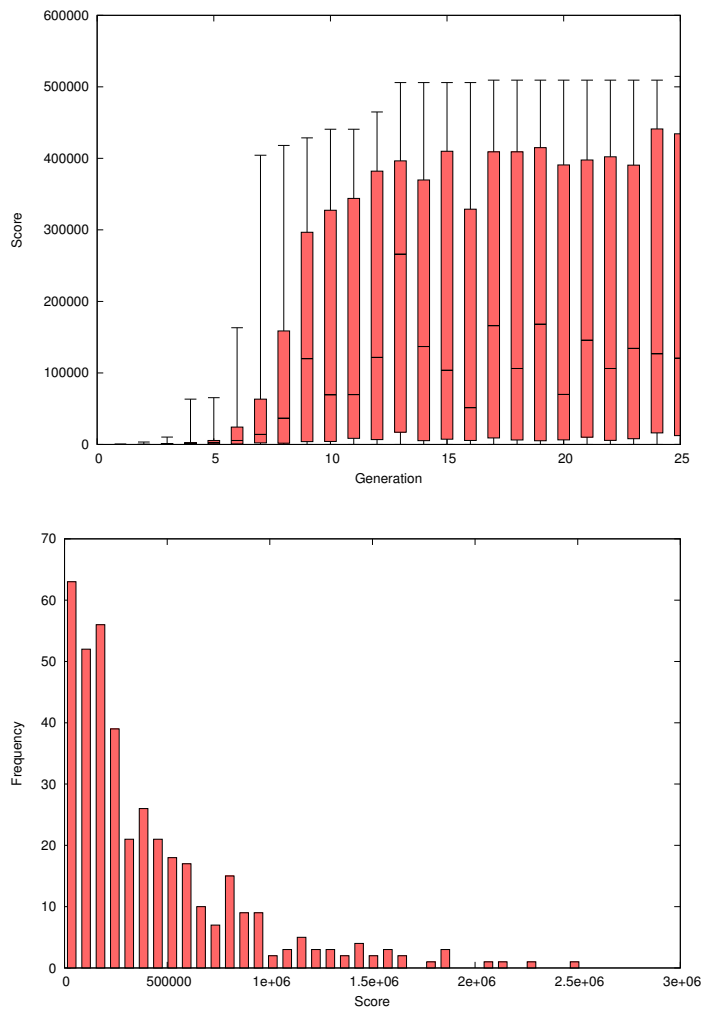


**Figure 4.7:** Progress during evolution using the same environment as Dellacherie [1] (top chart), and the performance of the resulting Tetris controller, illustrated with a histogram showing the frequency of scores (bottom chart).

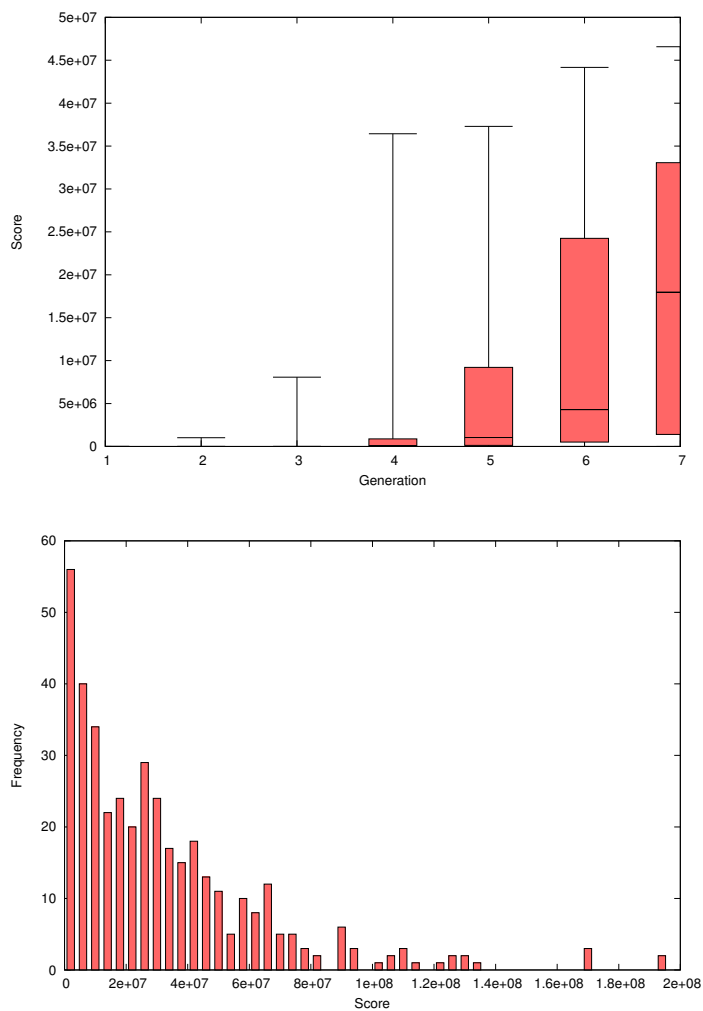




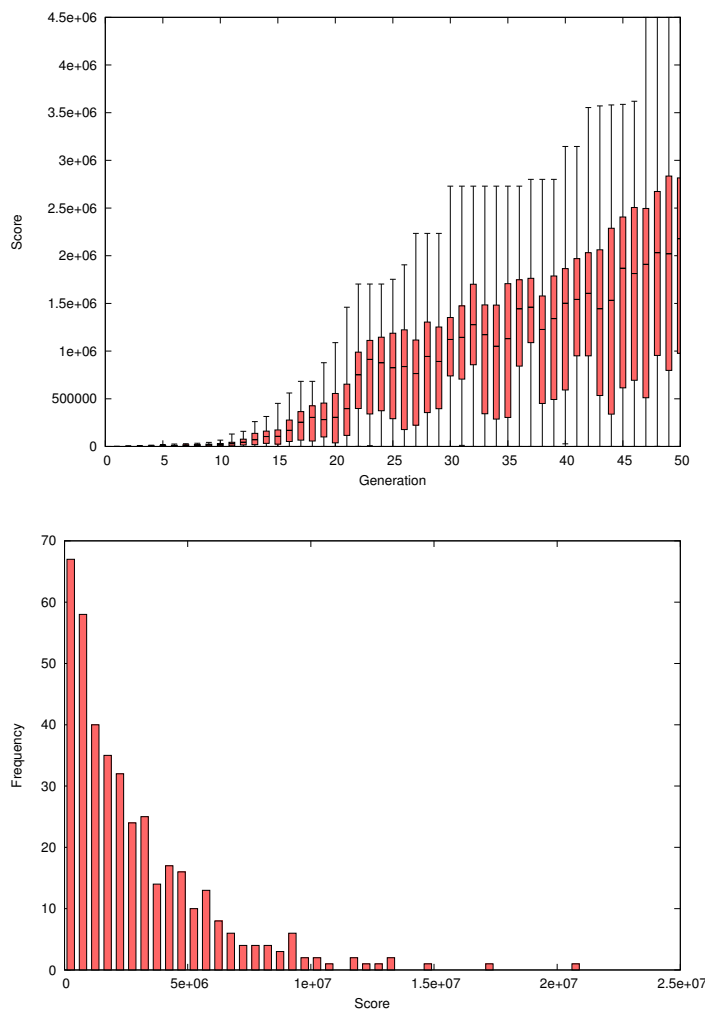
**Figure 4.8:** Progress during evolution using the same environment as Ramon and Driessens [10] (bottom chart), and the performance of the resulting Tetris controller, illustrated with a histogram showing the frequency of scores (bottom chart).



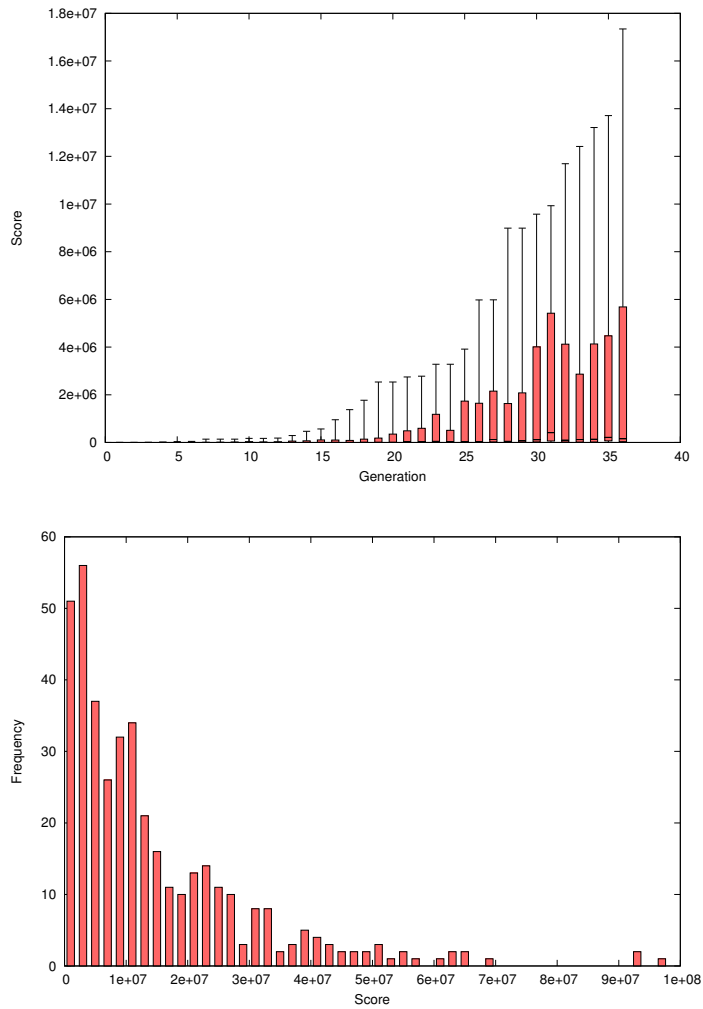
**Figure 4.9:** Progress during evolution using the same environment as Llima [11] (top chart), and the performance of the resulting Tetris controller, illustrated with a histogram showing the frequency of scores (bottom chart).



**Figure 4.10:** Progress during evolution using the same environment as Thiery and Scherrer [4] (top chart), and the performance of the resulting Tetris controller, illustrated with a histogram showing the frequency of scores (bottom chart).



**Figure 4.11:** Progress during evolution using the same environment as Langenhoven [15] (top chart), and the performance of the resulting Tetris controller, illustrated with a histogram showing the frequency of scores (bottom chart).



**Figure 4.12:** Progress during evolution using an unbiased selection of feature functions, and the performance of the resulting Tetris controller, illustrated with a histogram showing the frequency of scores (bottom chart).

# Chapter 5

## Discussion

### 5.1 Feature function contribution to search space

A curious observation to make is that the presented evolutionary algorithm performs very poorly in environments that consists of the feature functions 4 and 5. It does not manage to evolve a controller that is capable of clearing any lines at all. During evolution within these environment, the algorithm would search frantically for candidate solutions, but not find anything that yielded any score. One possible explanation for this particular result is that with these two feature functions as part of the environment, the search space increases dramatically, as they require 10 and 9 weights to be tuned, respectively. Comparatively, every other feature function only depend upon *one* weight and only increase the search space by a single dimension.

An increase of search space will make a problem more difficult for any general optimization method. However, most of the related work that consists of the two previously mentioned feature functions utilize reinforcement learning methods, which are fundamentally different in the way they find a solution. In the works of these methods, weights are tuned such that the fitness function approximates the optimal *expected score* from any given state. This does not in itself explain why these optimization methods are able to provide a solution, while our general purpose method is not. It does however provide some grounds to the conception that these two types of optimization methods might perform differently in some areas.

## 5.2 Large score deviation & halting evolution

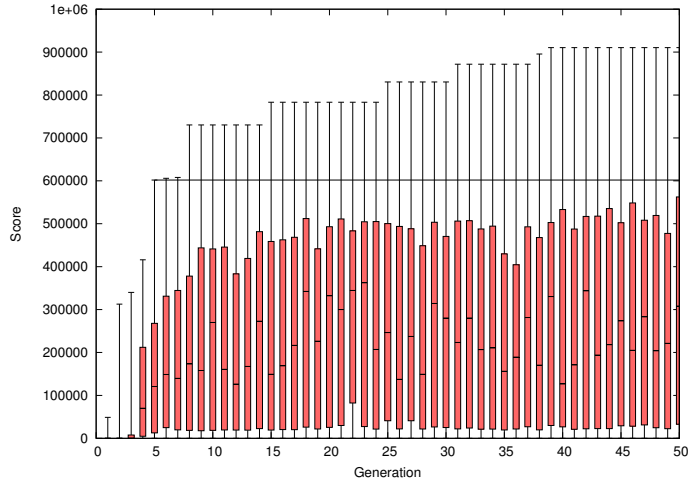
It turned out that the large deviation of Tetris score affected the general performance of our evolutionary algorithm more than we had hoped for. Due to the random nature of an evolutionary algorithm, states of the search spaces, IE. candidate solutions, would be re-visited and re-evaluated. Together with elitism, identical solutions would be evaluated multiple times and the best observed average of a limited number of trials would be kept.

This "duped" the evolution and contributed to seed generations with genetic traits that were really not as good as its fitness value indicated. We observed that the progress of evolution would halt once an individual received an unusual high score and was mistakenly threated as an elite. This phenomenon is illustrated in figure 5.1. The horizontal line marks the generations where the best individual actually never changed, but was simply re-evaluated. It can be seen that the average fitness of the population stops to progress.

This issue was addressed by increasing the number of trials that was performed for each individual during evolution, from 30 to 100, making the scores more statistically significant. This increased the overall performance of the algorithm. For the environment described by Llima [11], our algorithms score increased from 287,791 to 418,944. Another way this could have be solved would be to simply keep a record of every evolved individual of every generation and never calculate an individuals fitness value if it had previously been visited. Possibly even better would be a combination of never re-evaluating candidate solutions and having a high number of trial runs per individual.

## 5.3 Competitive general purpose optimization

Another interesting observation to make is that the genetic algorithm evolves Tetris controllers that are very competitive compared to that of other general purpose optimization methods [4, 11, 15]. This does not come as a complete surprise, as the problem at hand (namely evolving coefficients/relative weights) exhibits attributes that are beneficial for an evolutionary algorithm. First of all, the domain has a fitness landscape that is not jagged, but quite smooth. Secondly, the mutation operators previously described are such that they don't change an individual too much. Lastly, it possible to combine two sets of coefficients to create another candidate solution and get decent results. This very same genetic algorithm has previously been tested in the domain of Rubik's Cube [23], but it achieved very poor results due to its lack of these three attributes.



**Figure 5.1:** The line marks the generations where the best individual remains the same, but still achieves a higher score. This halts the progress of evolution.

## 5.4 Time constraints

It can be seen in figure 4.10 and 4.12 that some executions of the evolutionary algorithm did not continue long enough for the progress of evolution to stagnate and halt. This was due to constraints of the granted CPU hours on the super computer and was obviously very unfortunate. How an evolutionary algorithm can perform in the corresponding environments is therefore still not completely answered.



## Chapter 6

# Conclusion

We hypothesised initially that the choice of feature functions may greatly affect the overall performance of a Tetris controller. In order to test this, we tried to evolve a Tetris controller using the same feature functions and Tetris environment described in several scientific publications. The results clearly shows scores that differs substantially. This strongly suggest that the choice of feature functions is important and will greatly affect the overall performance of a Tetris controller. It also confirms the notion that it is impossible to compare optimization methods tested in the Tetris domain unless one has chosen the exact same feature functions. This is unfortunately *not* the case for most publications.

Thiery and Scherrer [17] specifically asks under what conditions the cross-entropy method could perform better than genetic algorithms or if they do. For the time being, it would seem like it beats the evolutionary approach in the domain of Tetris. However, we strongly believe that our implementation will experience *some* performance improvement when properly addressing the issues of re-visited candidate solutions. Given that our results are so similar, we hesitate to exclude the option that this might put the evolutionary approach on the top. To address these issues is left as further work.

# Bibliography

- [1] C. P. Fahey, “Tetris.” <http://www.colinfahey.com/tetris/tetris.html>. [Online; accessed 15-March-2014].
- [2] E. D. Demaine, S. Hohenberger, and D. Liben-Nowell, “Tetris is hard, even to approximate,” in *Computing and Combinatorics*, pp. 351–363, Springer, 2003.
- [3] H. Burgiel, “How to lose at tetris,” *Mathematical Gazette*, vol. 81, pp. 194–200, 1997.
- [4] C. Thiery and B. Scherrer, “Improvements on learning tetris with cross-entropy,” *International Computer Games Association Journal*, vol. 32, 2009.
- [5] R. P. Lippmann, L. Kukulich, and E. Singer, “Lnknet: Neural network, machine-learning, and statistical software for pattern classification,” *Lincoln Laboratory Journal*, vol. 6, pp. 249–249, 1993.
- [6] J. N. Tsitsiklis and B. V. Roy, “Feature-based methods for large scale dynamic programming,” *Machine Learning*, vol. 22, no. 1-3, pp. 59–94, 1996.
- [7] D. P. Bertsekas and J. N. Tsitsiklis, “Neuro-dynamic programming,” *Athena Scientific, Belmont, MA*, 1996.
- [8] S. Kakade, “A natural policy gradient,” *Advances in Neural Information Processing Systems*, vol. 14, 2001.
- [9] M. G. Lagoudakis, R. Parr, and M. L. Littman, “Least-squares methods in reinforcement learning for control,” in *Methods and Applications of Artificial Intelligence*, pp. 249–260, Springer, 2002.
- [10] J. Ramon and K. Driessens, “On the numeric stability of gaussian processes regression for relational reinforcement learning,” in *ICML-2004 Workshop on Relational Reinforcement Learning*, pp. 10–14, Citeseer, 2004.

- [11] R. E. Llima, “Xtris readme.” <ftp://ftp.x.org/contrib/games/xtris.README>. [Online; accessed 15-March-2014].
- [12] N. Böhm, G. Kókai, and S. Mandl, “An evolutionary approach to tetris,” in *6th Metaheuristics International Conference (6th Metaheuristics International Conference Wien August 22-26, 2005)*, 2005.
- [13] V. F. Farias and B. V. Roy, “Tetris: A study of randomized constraint sampling,” in *Probabilistic and Randomized Methods for Design Under Uncertainty*, pp. 189–201, Springer, 2006.
- [14] I. Szita and A. Lörincz, “Learning tetris using the noisy cross-entropy method,” *Neural computation*, vol. 18, no. 12, pp. 2936–2941, 2006.
- [15] L. Langenhoven, W. S. van Heerden, and A. P. Engelbrecht, “Swarm tetris: Applying particle swarm optimization to tetris,” in *Evolutionary Computation (CEC), 2010 IEEE Congress on*, pp. 1–8, IEEE, 2010.
- [16] B. Scherrer, “Performance bounds for  $\lambda$  policy iteration and application to the game of tetris,” *The Journal of Machine Learning Research*, vol. 14, no. 1, pp. 1181–1227, 2013.
- [17] C. Thiery, B. Scherrer, *et al.*, “Building controllers for tetris,” *International Computer Games Association Journal*, vol. 32, pp. 3–11, 2009.
- [18] E. V. Siegel, A. D. Chaffee, and L. EarthWeb, “Genetically optimizing the speed of programs evolved to play tetris,” *Advances in genetic programming*, vol. 2, pp. 279–298, 1996.
- [19] C. Darwin, *On the Origin of Species*. 1859.
- [20] A. Globus, G. Hornby, D. Linden, and J. Lohn, “Automated antenna design with evolutionary algorithms,” 2006.
- [21] A. Keane, “The design of a satellite beam with enhanced vibration performance using genetic algorithm techniques,” *The Journal of the Acoustical Society of America*, vol. 99, no. 4, pp. 2599–2603, 1996.
- [22] H. G. Katzgraber, “Random numbers in scientific computing: An introduction,” *arXiv preprint arXiv:1005.4117*, 2010.
- [23] J. B. Amundsen, “Attempting to use an unbiased evolutionary approach for solving the rubik’s cube,” 2013.

# Appendix A

## Formal feature definitions

Below is a formal definition of each feature function mentioned in table 2.1, except for *Jags* and *Embedded holes* due to their lack of documentation. The width and height of the board considered in these definitions are denoted as  $w$  and  $h$ , respectively, with  $W$  and  $H$  denoting the corresponding sets  $W = \{0, 1, \dots, w-1\}$  and  $H = \{0, 1, \dots, h-1\}$ . References to cells are denoted as  $b(x, y)$  and is defined as

$$b(x, y) = \begin{cases} c_{x,y} & \text{if } x \in W \wedge y \in H \\ nan & \text{if } x \notin W \vee y \notin H \end{cases} \quad (\text{A.1})$$

where  $c_{x,y}$  is a reference to the cell values depicted in figure A.1. A full cell has a value of 1, while an empty cell has a value of 0. A set-builder notation is used and has the form  $\{x : \Phi(x)\}$ . Here,  $x$  is a free variable and the set contains all possible values of  $x$  such that  $\Phi(x)$  evaluates to 'true'. Some feature functions use helper functions defined in some other definition. In such case, the *last* definition of matching name applies, unless specified otherwise.

.....

$$f_{\text{max height}} = \max(\{c_h(x) : x \in W\}) \quad (\text{A.2})$$

where

$C_{0,19}$	$C_{1,19}$	$C_{2,19}$	$C_{3,19}$	$C_{4,19}$	$C_{5,19}$	$C_{6,19}$	$C_{7,19}$	$C_{8,19}$	$C_{9,19}$
$C_{0,18}$	$C_{1,18}$	$C_{2,18}$	$C_{3,18}$	$C_{4,18}$	$C_{5,18}$	$C_{6,18}$	$C_{7,18}$	$C_{8,18}$	$C_{9,18}$
$C_{0,17}$	$C_{1,17}$	$C_{2,17}$	$C_{3,17}$	$C_{4,17}$	$C_{5,17}$	$C_{6,17}$	$C_{7,17}$	$C_{8,17}$	$C_{9,17}$
$C_{0,16}$	$C_{1,16}$	$C_{2,16}$	$C_{3,16}$	$C_{4,16}$	$C_{5,16}$	$C_{6,16}$	$C_{7,16}$	$C_{8,16}$	$C_{9,16}$
$C_{0,15}$	$C_{1,15}$	$C_{2,15}$	$C_{3,15}$	$C_{4,15}$	$C_{5,15}$	$C_{6,15}$	$C_{7,15}$	$C_{8,15}$	$C_{9,15}$
$C_{0,14}$	$C_{1,14}$	$C_{2,14}$	$C_{3,14}$	$C_{4,14}$	$C_{5,14}$	$C_{6,14}$	$C_{7,14}$	$C_{8,14}$	$C_{9,14}$
$C_{0,13}$	$C_{1,13}$	$C_{2,13}$	$C_{3,13}$	$C_{4,13}$	$C_{5,13}$	$C_{6,13}$	$C_{7,13}$	$C_{8,13}$	$C_{9,13}$
$C_{0,12}$	$C_{1,12}$	$C_{2,12}$	$C_{3,12}$	$C_{4,12}$	$C_{5,12}$	$C_{6,12}$	$C_{7,12}$	$C_{8,12}$	$C_{9,12}$
$C_{0,11}$	$C_{1,11}$	$C_{2,11}$	$C_{3,11}$	$C_{4,11}$	$C_{5,11}$	$C_{6,11}$	$C_{7,11}$	$C_{8,11}$	$C_{9,11}$
$C_{0,10}$	$C_{1,10}$	$C_{2,10}$	$C_{3,10}$	$C_{4,10}$	$C_{5,10}$	$C_{6,10}$	$C_{7,10}$	$C_{8,10}$	$C_{9,10}$
$C_{0,9}$	$C_{1,9}$	$C_{2,9}$	$C_{3,9}$	$C_{4,9}$	$C_{5,9}$	$C_{6,9}$	$C_{7,9}$	$C_{8,9}$	$C_{9,9}$
$C_{0,8}$	$C_{1,8}$	$C_{2,8}$	$C_{3,8}$	$C_{4,8}$	$C_{5,8}$	$C_{6,8}$	$C_{7,8}$	$C_{8,8}$	$C_{9,8}$
$C_{0,7}$	$C_{1,7}$	$C_{2,7}$	$C_{3,7}$	$C_{4,7}$	$C_{5,7}$	$C_{6,7}$	$C_{7,7}$	$C_{8,7}$	$C_{9,7}$
$C_{0,6}$	$C_{1,6}$	$C_{2,6}$	$C_{3,6}$	$C_{4,6}$	$C_{5,6}$	$C_{6,6}$	$C_{7,6}$	$C_{8,6}$	$C_{9,6}$
$C_{0,5}$	$C_{1,5}$	$C_{2,5}$	$C_{3,5}$	$C_{4,5}$	$C_{5,5}$	$C_{6,5}$	$C_{7,5}$	$C_{8,5}$	$C_{9,5}$
$C_{0,4}$	$C_{1,4}$	$C_{2,4}$	$C_{3,4}$	$C_{4,4}$	$C_{5,4}$	$C_{6,4}$	$C_{7,4}$	$C_{8,4}$	$C_{9,4}$
$C_{0,3}$	$C_{1,3}$	$C_{2,3}$	$C_{3,3}$	$C_{4,3}$	$C_{5,3}$	$C_{6,3}$	$C_{7,3}$	$C_{8,3}$	$C_{9,3}$
$C_{0,2}$	$C_{1,2}$	$C_{2,2}$	$C_{3,2}$	$C_{4,2}$	$C_{5,2}$	$C_{6,2}$	$C_{7,2}$	$C_{8,2}$	$C_{9,2}$
$C_{0,1}$	$C_{1,1}$	$C_{2,1}$	$C_{3,1}$	$C_{4,1}$	$C_{5,1}$	$C_{6,1}$	$C_{7,1}$	$C_{8,1}$	$C_{9,1}$
$C_{0,0}$	$C_{1,0}$	$C_{2,0}$	$C_{3,0}$	$C_{4,0}$	$C_{5,0}$	$C_{6,0}$	$C_{7,0}$	$C_{8,0}$	$C_{9,0}$

**Figure A.1:** Cell indices on a Tetris board used in the formal definitions of feature functions.

$$c_h(x) = \begin{cases} \max(\{y : y \in H \wedge b(x, y) = 1\} \cup \{0\}) & \text{if } x \in W \\ \infty & \text{if } x \notin W \end{cases} \quad (\text{A.3})$$

.....

$$f_{\text{holes}} = |\{(x, y) : b(x, y) = 0 \wedge c_h(x) > y\}| \quad (\text{A.4})$$

.....

$$f_{\text{column height}} = c_h(x) \quad (\text{A.5})$$

.....

$$f_{\text{column difference}} = c_h(x) - c_h(x + 1) \quad (\text{A.6})$$

.....

$$f_{\text{landing height}} = t_y + \frac{t_h - 1}{2} \quad (\text{A.7})$$

where  $t_y$  is the landing height of the *lowest* part of the tetromino, I.E. the y-coordinate of the lowest cell filled, and  $t_h$  is the height of the tetromino.

.....

$$f_{\text{cell transitions}} = |\{(x, y) : b(x, y) = 0 \wedge b(x, y - 1) = 1\}| \quad (\text{A.8})$$

$$+ |\{(x, y) : b(x, y) = 0 \wedge b(x, y + 1) = 1\}| \quad (\text{A.9})$$

$$+ |\{(x, y) : b(x, y) = 0 \wedge b(x - 1, y) = 1\}| \quad (\text{A.10})$$

$$+ |\{(x, y) : b(x, y) = 0 \wedge b(x + 1, y) = 1\}| \quad (\text{A.11})$$

.....

$$f_{\text{deep wells}} = \sum_{(x,d) \in S_{w_d}} d \tag{A.12}$$

where

$$S_{w_d} = \{(x, d) : (x, d) \in S_w \wedge d > 1\} \tag{A.13}$$

$$S_w = \{(x, \max(\min(c_h(x-1), c_h(x+1)) - c_h(x), 0)) : x \in W\} \tag{A.14}$$

.....

$$f_{\text{mean height}} = \frac{1}{w} \sum_{x=0}^{w-1} c_h(x) \tag{A.15}$$

.....

$$f_{\text{height weighted cells}} = \sum_{(x,y) \in S_c} y \tag{A.16}$$

where

$$S_c = \{(x, y) : b(x, y) = 1\} \tag{A.17}$$

.....

$$f_{\text{wells}} = \sum_{(x,d) \in S_w} d \tag{A.18}$$

.....

$$f_{\text{full cells}} = |S_c| \quad (\text{A.19})$$

.....

$$f_{\text{row transitions}} = |\{(x, y) : b(x, y) \oplus b(x + 1, y) = 1\}| \quad (\text{A.20})$$

$$+ |\{y : b(0, y) = 0\}| \quad (\text{A.21})$$

$$+ |\{y : b(w - 1, y) = 0\}| \quad (\text{A.22})$$

.....

$$f_{\text{column transitions}} = |\{(x, y) : b(x, y) \oplus b(x, y + 1) = 1\}| \quad (\text{A.23})$$

$$+ |\{x : b(x, 0) = 0\}| \quad (\text{A.24})$$

$$+ |\{x : b(x, h - 1) = 1\}| \quad (\text{A.25})$$

.....

$$f_{\text{cumulative wells}} = \sum_{(x, y, d) \in S_w} d \quad (\text{A.26})$$

where

$$S_w = \{(x, y, d(x, y)) : b'(x - 1, y) = 1 \wedge b'(x, y) = 0 \wedge b'(x + 1, y) = 1\} \quad (\text{A.27})$$

$$b'(x, y) = \begin{cases} 1 & \text{if } x < 0 \\ 1 & \text{if } x \geq w \\ b(x, y) & \text{if } x \geq 0 \wedge x < w \end{cases} \quad (\text{A.28})$$



$$d(x, y) = \max(\{i : i \in H \wedge \forall_n \in \{0, 1, \dots, i\} (b(x, y - n) = 0)\}) \quad (\text{A.29})$$

.....

$$f_{\text{min height}} = \min(\{c_h(x) : x \in W\}) \quad (\text{A.30})$$

.....

$$f_{\text{max - mean height}} = f_{\text{max height}} - f_{\text{mean height}} \quad (\text{A.31})$$

.....

$$f_{\text{mean - min height}} = f_{\text{mean height}} - f_{\text{min height}} \quad (\text{A.32})$$

.....

$$f_{\text{mean hole depth}} = \frac{f_{\text{hole depth}}}{f_{\text{adjacent column holes}}} \quad (\text{A.33})$$

.....

$$f_{\text{max height difference}} = f_{\text{max height}} - f_{\text{min height}} \quad (\text{A.34})$$

.....

$$f_{\text{adjacent column holes}} = |\{(x, y) : b(x, y) = 1 \wedge b(x, y - 1) = 0\}| \quad (\text{A.35})$$

.....

$$f_{\text{max well depth}} = \max(\{d : (x, d) \in S_w\}) \quad (\text{A.36})$$

where  $S_w$  is as defined in  $f_{\text{deep wells}}$ .

.....

$$f_{\text{hole depth}} = \sum_{(x,y,d) \in S_d} d \quad (\text{A.37})$$

where

$$S_d = \{(x, y, d(x, y)) : b(x, y) = 0 \wedge b(x, y + 1) = 1\} \quad (\text{A.38})$$

$$d(x, y) = \max(\{i : i \in H \wedge \forall_n \in \{0, 1, \dots, i\} (b(x, y + n) = 1)\}) \quad (\text{A.39})$$

.....

$$f_{\text{rows with holes}} = |\{y : y \in H \wedge \exists_{x \in W} (b(x, y) = 0 \wedge c_h(x) > y)\}| \quad (\text{A.40})$$

.....

$$f_{\text{pattern diversity}} = |\{p : p \in \{-2, \dots, 2\} \wedge \exists (f_{\text{column difference}} = p)\}| \quad (\text{A.41})$$

# Appendix B

## Example game

The following illustrations show the progress of one game using the implemented controller described in this paper and the strategy depicted in the table below. The illustrations should be read from left to right and from top to bottom.

	FEATURE	VALUE
3.	Holes	5
6.	Landing height	-33
20.	Eroded piece cells	9
21.	Row transitions	-20
22.	Column transitions	-76
23.	Cumulative wells	-31
31.	Hole depths	-2
32.	Rows with holes	-65

