On the evolution of artificial Tetris players

Amine Boumaza

On the evolution of artificial Tetris players*

Amine Boumaza

June 19, 2009

Abstract

We discuss the use of evolutionary algorithms to learn strategies to play the game of Tetris. We describe the problem and discuss the nature of the search space. We presents experiments to illustrate the learning process of our artificial player, and provide a new procedure to speed up the learning time. The results we present compare with the best known artificial player, and show how our evolutionary algorithm is able to rediscover player strategies previously published. Finally we provide some ideas to improve the performance of artificial Tetris players.

1 Introduction

Tetris is a single player game where the goal is to place randomly falling pieces onto a game board. Each horizontal line completed is cleared from the board and the scores points for the player. A detailed technical description may be found in [7].

Tetris is a very popular game with simple rules. It however requires lots of practice and elaborate strategies for human players to play the game well. Designing artificial players represents a great challenge for the artificial intelligence community. This difficulty resides mainly in the large number of states of the game[1] which is close to 10^{60}. This large state space cannot be explored directly, which pushes the need to use approximation mechanisms to reduce its size and thus design strategies for artificial players. Even with such approximations the problem of finding strategies to maximize the score is NP-Complete [4]. Furthermore, is has been shown by [6] that the game cannot last forever and that it finishes with probability 1.

All these challenges renders the problem of designing artificial player very appealing for the AI community, where designing artificial players has been addressed in different articles using different techniques. Section 2 reviews some of these techniques that range from reinforcement learning to pure optimization and evolutionary algorithms.

In the existing literature, authors usually address a simplified version of the problem where time is not taken into account. In the original game the player has to decide where to place the current piece while this last is falling from the top of the game board. In the simplified version, pieces do not fall. The artificial player is given the current piece and the current board and it has to decide on where to place the piece. A decision in this case is a rotation and a translation (the column) to apply to the current piece, and usually the player tests all possible rotation and translation and decides based on a decision function on the bests action to take. Furthermore, except for few authors, most of the existing work addresses "one piece strategies", where only the current piece is known and not the next one, which is the case in the standard game.

In this paper, we discuss the use of evolutionary algorithms to learn Tetris strategies. We will present an algorithm to learn strategies and compare its performance against the best known artificial players. For that we will first state the problem of learning Tetris strategies and review the existing literature of the field (section 2). After which we will describe the evolutionary algorithm we used (section 3), describe the experimental protocols we followed (section 4) and discuss the results we obtained (section 5). Finally, we conclude our discussion and present future ideas we intend to follow to improve our findings (section 6).

*Amine Boumaza is with the Univ. Lille Nord de France, F-59000 Lille, France, ULCO, LIL, F-62100 Calais, France (email: amine.boumaza@lil.univ-littoral.fr)

[1] On the standard 10×20 board with 7 pieces, each cell can be in 2 states and the number of states is $2^{10\times 2} \times 7 \approx 10^{60}$

2 Previous work

Before reviewing the literature on designing Tetris players, let us begin by defining what is an artificial player for this game. In general, a game-playing agent decides which action to choose next at any given state of the game based on the reward it will gain after performing that action. The agent chooses, among several possible actions, the one that will return the best outcome. It is however difficult and sometime not possible to judge the value of an action, and one way around this is to judge the value of the state the action will lead to. Generally this is achieved by using an evaluation function which assigns numerical values to game states. The player thus chooses the action which leads to the best valued state. This can also be seen as a greedy strategy, in which the agent takes the highest valued decision at each step.

In the literature, artificial Tetris players use evaluation functions which evaluate the game board by assigning numerical values. Given a piece, the agent will choose the decision (translation and rotation) of the piece that will give the "best board" when dropped. Figure 1 illustrates this. When a pieces is to be placed, the agents simulates and evaluates all the possible boards that result from all the possible moves of the piece, and chooses the move that leads to the best valued game board.

Note that since the agent follow a greedy strategy, the evaluation function is the only component that enter into the decision making process. Hens, designing a Tetris player amounts to designing an evaluation function. Such a function should synthesise the game state giving high values to "good boards" and low values for "bad" ones. The evaluation function for the game of Tetris may be considered as mimicking the

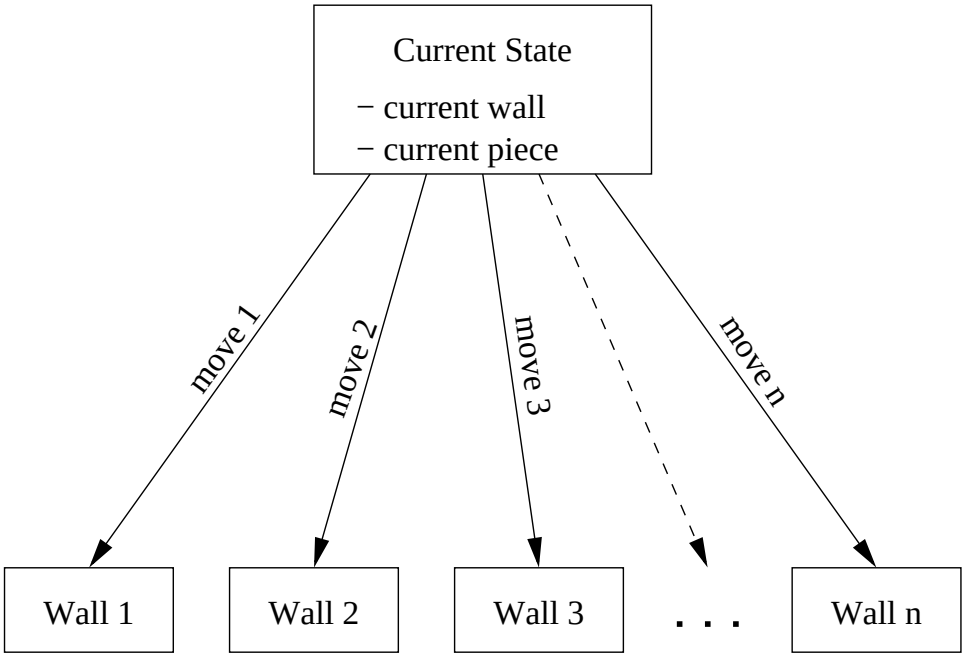

Figure 1: The decision process for an artificial Tetris player.

evaluation a human player makes when deciding where to drop the piece and which orientation to give to it. It should rank higher game states that increase the chance to clear lines in subsequent moves. For example such boards should be low in height and should contain as less wholes as possible. These properties (the height, the number of wholes) are called **features** and an evaluation function is a **weighted linear combination of features**.

Let f_i denotes a feature, f_i is a function that maps a state (a board configuration) to a real value. The value of a state s is then defined as:

$$V(s) = \sum_{i=1}^{N} w_i f_i(s) \tag{1}$$

where N is the number of features and w_i are the weights of each one of them.

Let P be the set of all pieces and D the set of all decisions. We note $d_i(p) \in D$ the i^{th} decision applied

Table 1: List of features used for our player.

Id	Name	Description
f_1	Landing height	Height where the last piece is added
f_2	Eroded pieces	(# lines cleared in the last move) × (# bricks eliminated from the last pieces added)
f_3	Row transitions	# horizontal cell transitions (filled/empty)
f_4	Column transition	# vertical cell transitions
f_5	Holes	# empty cells covered by a filled cell
f_6	Cumulative wells	$\sum_{w \in \text{wells}}(1 + 2 + \ldots + \text{depth}(w))$
f_7	Hole depth	# filled cells on top of each hole
f_8	Rows holes	# rows with at least one hole

on piece $p \in P$. A decision is a rotation (4 possibilities) and a translation (10 possibilities[2]) for the current piece. Furthermore, we note γ the function that maps a pair $(s, d_i(p))$, a state s and a decision $d_i(p)$, to the new state of the game after taking decision d_i for the piece p. Given all possible decisions $d_i(p) \in D$ for a current piece p in state s, the player will choose the highest valued one:

$$\hat{d}(s, p) = \arg \max_{d_i(p) \in D} V(\gamma(s, d_i(p))) \tag{2}$$

The function $\hat{d}(s, p)$ is the player s decision function. As stated above the only component that enter into the decision process is the value V, in other words the behavior of the player is conditioned by the value of the feature functions $f_{1\ldots N}$.

Generally, the design of artificial players amounts to fixing the weights of the evaluation function which is generally performed using learning strategies to learn the weights of the feature functions. The set of features is chosen by an expert.

In the literature many authors have introduced different features to synthesize the state of the game, [21] gives a good review of these features. In this work we use eight features six of them are the features introduced by Dellacherie [7] for his artificial player, and two were introduced by [22]. These features are summarized on table 1, features f_1 through f_6 are from [7], f_7 and f_8 are from [22].

In the literature choosing the weights of the feature functions has been done using different techniques. Dellacherie [7] fixed the weights by hand and until not long ago the artificial player he proposed was the best performing one (660 000 cleared lines on average). Furthermore, different authors have used reinforcement learning to learn the weights. Among these we can cite [23, 2, 12, 8, 14, 17], however, these works have had very little success compared with the hand coded player proposed by Dellacherie.

Other authors considered the problem differently and proposed to learn the feature weights using optimization techniques. Szita and Lõrincz [20] applied the noisy cross entropy method (NCE) [5] to optimize the weights of the features introduced by [2] and reported a score of $35 \times 10^4 \pm 36\%$. More recently Thiery and Scherrer [22] used the noisy cross entropy method to learn the weights proposed by Dellacherie and reported scores of $35 \times 10^6 \pm 20\%$. The authors give a review of the existing work on learning Tetris players and argue on the difficulty to compare different players using different interpretation of the game since small differences may lead to large discrepancies in the final score. They also provide a simulator of the game that implements most of the features that exist so that they can be compared on the same basis[3]. The work presented here uses the same simulator.

To our knowledge Siegel and Chaffee [19] were the first to apply evolutionary computation to design artificial players for the game of Tetris. They proposed to use genetic programming [13]. Though their work focuses on improving the speed of evolved programs, they used the game of Tetris as an illustrative application and reported a mean score of 74 over 20 runs[4]. Llima [15] used a genetic algorithm to optimize the weights of the artificial player of the GNU Xtris simulator which uses six features. On Xtris, this player scores 50 000 on average. Böhm et al. [16] used an evolutionary algorithm to design Tetris players for which the authors proposed new features along with Dellacherie s features and three different

[2]The number of translations depends on the game width.

[3]The source code of the simulator can be found at `http://mdptetris.gforge.inria.fr`

[4]In fact their artificial player don t use an evaluation function, the evolved programs use strategies of their own discovered thought evolution.

evaluation functions: a linear combination as in equation 1, a exponential combination and exponential with displacement. The last two functions rate differently differences in heights when the board is low than when it is high. Whoever since they considered the two piece game, there are no results reported for the one piece game considered here.

3 Evolving Tetris players

In this work, we propose to use evolutionary algorithms to learn the weights of an evaluation function for a Tetris player using the features of table 1. We will begin by introducing briefly evolution strategies (ES), and the variant we use here the covariance matrix adaptation evolution strategy (CMA-ES) [10, 11] and will then detail how we proceed to evolve artificial Tetris players.

Evolution strategies [18, 1, 3] are evolutionary algorithms that operate in continuous optimisation problems that can be formulated as follows:

$$\hat{\mathbf{x}} = \arg\max_{\mathbf{x}} f(\mathbf{x}) \;\; with \;\; f : \mathbb{R}^n \rightarrow \mathbb{R}$$

where f is the objective function to optimize. They can be seen as continuously updating a search point $m \in \mathbb{R}^n$ that is the average of a normal distribution. Algorithm 1 describes a canonical ES. Given an

Algorithm 1 Procedure ES

Input $(\mu, \lambda, m, \sigma\ f)$
1: **Begin**
2: $m^0 := m$, $\sigma^0 := \sigma$, $C^0 := \mathbf{I}$, $t := 1$
3: **repeat**
4: **for** $i := 1$ **to** λ **do**
5: $x_i \sim m^t + \sigma^t \times \mathcal{N}_i\,(0, C^t)$
6: `evaluate`(x_i) using $f(x_i)$
7: **end for**
8: $m^{t+1} := \sum_{i=1}^{\mu} x_{i:\lambda}$ where $f(x_{1:\lambda}) \leq \ldots \leq f(x_{\mu:\lambda})$
9: `update`(σ^t)
10: `update`(C^t)
11: $t := t + 1$
12: **until** stopping_criterion
13: **return** m^{t-1}
14: **End**

initial mean value $m \in \mathbb{R}^n$, an initial step size $\sigma^0 \in \mathbb{R}_+$ and an initial covariance matrix $C^0 = \mathbf{I}$. At each step the algorithm samples λ search points (the offspring) $x_{1\ldots\lambda}$ using a normal distribution (lines 4-7). The offspring are then evaluated by computing $f(x_i)$ for $i = 1 \ldots \lambda$ and ranked from best to worst. The mean of the distribution is then updated (line 8) by taking the average of the $\mu < \lambda$ best ranked offspring (the parents).

Lines 9 and 10 of the algorithm are the crucial steps in evolution strategies. This is where the step size and the covariance matrix of the search distribution are updated. This adaptation helps the algorithm to adjust its progress in the search space. Larger step sizes increase exploration of the space while smaller ones increase exploitation of good solutions.

Evolution strategies use different mechanisms to perform this adaptation depending on the nature of the search distribution and more precisely the nature of the matrix C. The search distribution can be view as a N dimensional ellipsoid that reflects the normal distribution. When $C = \mathbf{I}$ the search distribution is isotropic (all axis of the ellipsoid have equal length). And when $C = \mathbf{D}^2$ where $\mathbf{D}$ denotes a diagonal matrix, the search distribution is anisotropic (certain axis are longer than others). Finally, and this is the more general case, when C is a full covariance matrix, the search distribution can take any shape and orientation in the sens that its axes are not linked to the coordinate system as in the previous cases. The direction of the longer axis indicates the direction of the search.

The CMA-ES algorithm used here adapts a full covariance matrix. This algorithm has proved its performance on many test cases [9]. We chose not insert the update procedure since thy are out of the scope of the presentation, however the interested reader could consult [11].

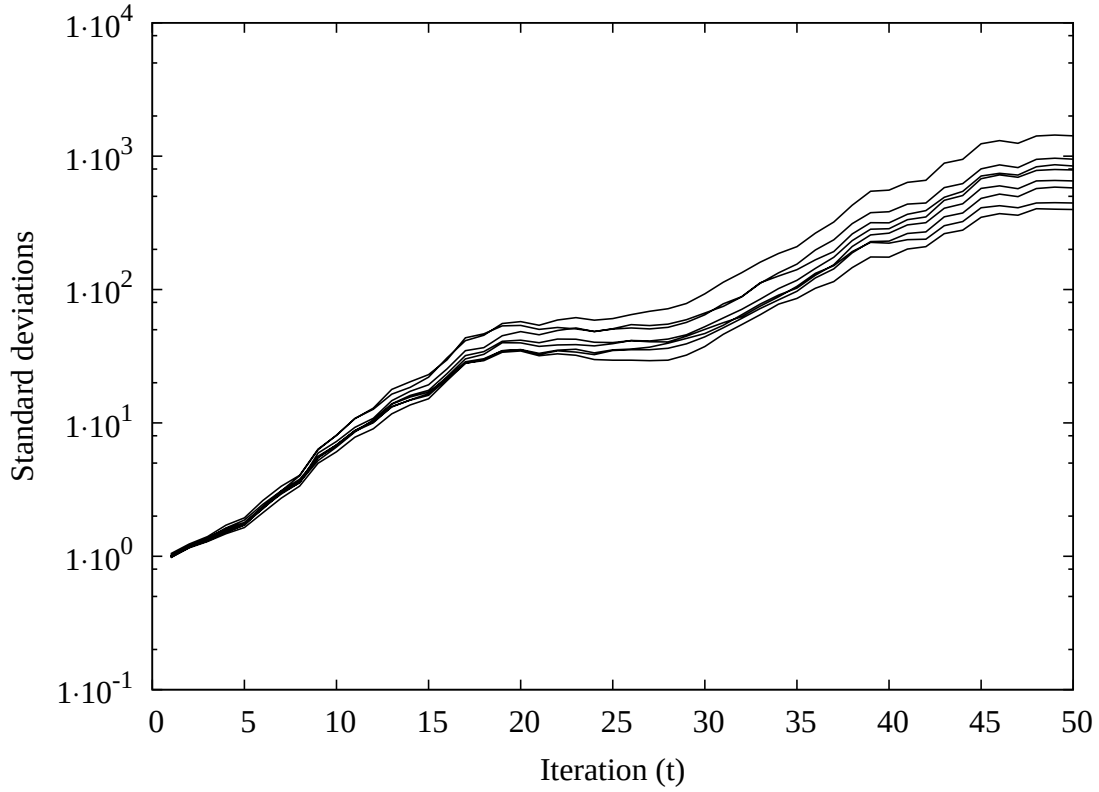

Figure 2: Divergence of the step sizes .

An important aspect of using evolutionary strategies in problem solving is to gather as much information about the objective function as possible to have some insight on the shape of the search space, exploit it and tune the algorithm to perform better. In the following we will try to give some idea about the shape of the search space we are addressing.

Since the evaluation function (equation 1) is a linear combination, let us consider two decisions d_1 and d_2 to place a piece p in state s such that $V(s_1 = \gamma(s, d_1(p)) > V(s_2 = \gamma(s, d_2(p))$. Then $\sum_{i=1}^{N} w_i f_i(s_1) > \sum_{i=1}^{N} w_i f_i(s_2)$. If we view w_i and $f_i(\cdot)$ as the coordinate of vectors w and $f(\cdot)$ we have $w \cdot f(s_1) > w \cdot f(s_2)$ where "$\cdot$" denotes the dot product. Let $\hat{w} = \frac{w}{\|w\|}$, then $\|w\|\hat{w} \cdot f(s_1) > \|w\|\hat{w} \cdot f(s_1)$ or $\hat{w} \cdot f(s_1) > \hat{w} \cdot f(s_1)$. This shows that multiplying a weight vector by a constant will not change the ranking of the decisions. In other words, all vectors $c \times w$ are equivalent with respect to the decisions of the player. This is very important to consider since it states that all the points of the search space that fall on the N dimensional line passing through w will have the same fitness value. This phenomenon may reduce the performance of an evolutionary strategy that bases the adaptation of the step size and the adaptation of the covariance on the fitness of the search points, which is the case for CMA-ES. Our experiments show that in this case the step sizes do not converge and keep increasing as if the algorithm is following an unattainable moving target (figures 2and 3). Similarly the weights do not converge either.

In order to fix this phenomenon, we added a slight modification to the algorithm to limit the search to weight vectors of unit length thus bounding the search space to the surface of the unit ball. After the sampling step (line 5 of algorithm 1), the vectors are normalized. Our experiments show that, this procedure significantly improves the convergence of the step sizes.

One other feature of the search space we can mention is linked to the score distribution of a Tetris player. As stated above, this distribution follows an exponential law and since each point of the search space is a weight vector its fitness value is a random variable that follows the same law. Therefore one evaluation of the offspring is not enough to estimate their fitness, to have better approximation many evaluation are needed. A weights vector is evaluated (line 6 of the algorithm) by playing a certain number of games and its fitness value is the average score.

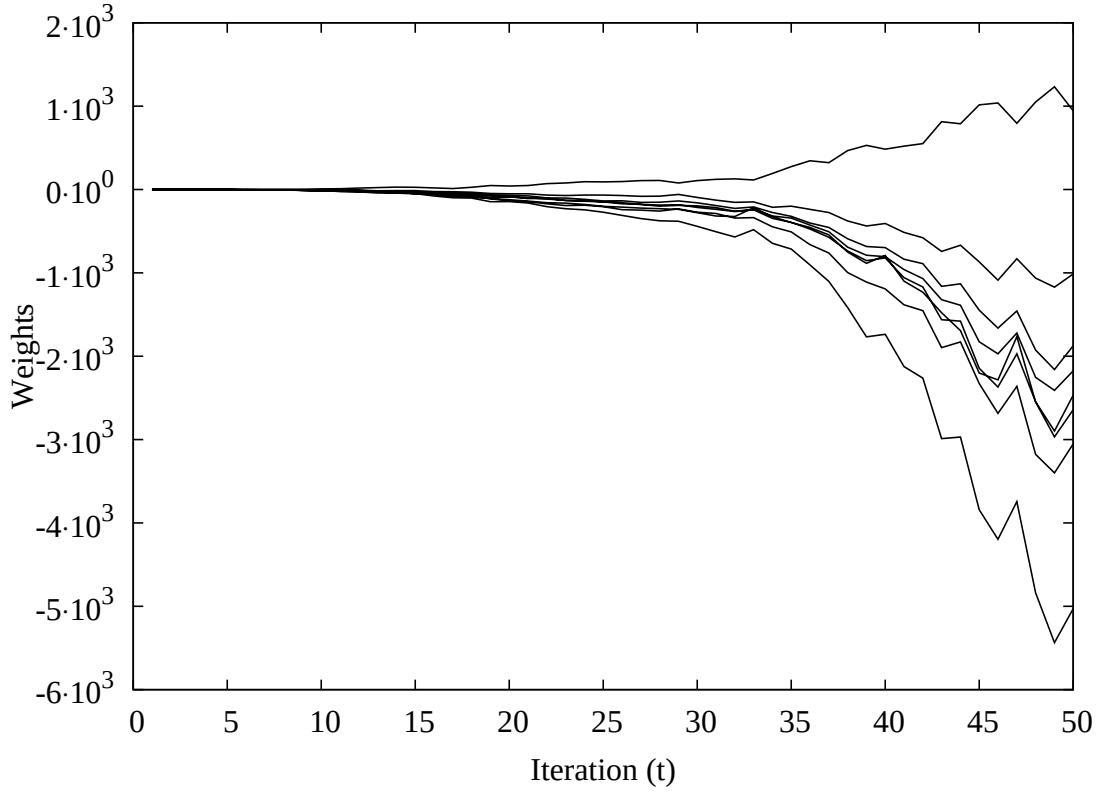

Figure 3: Divergence of the solution.

4 Experiments

The experiments we conducted are divided in two parts: the learning process and the evaluation process. In the learning process, the weights of the player s strategy are learned using CMA-ES. The player strategy is then evaluated on a series of games which asses of its performance.

A learning episode goes as follows: Starting with a zero vector ($m^{t=0} = 0$) and given an initial step size (typically $\sigma^{t=0} = 0.5$) we run the algorithm for a number of generations, in the presented experiments 50 iterations. At the end of the run we record the mean vector produced. We repeat this procedure for a certain number of times typically 20. In our experimental protocol, we take the average of the weight vectors produced at each run as our player strategy. This protocol is different from what is generally used in the literature, where authors chose the best vector of one run as the player s strategy, which is in our sens not reliable due the nature of the score distribution. In order to have a good estimation of the fitness of a weight vector, it is necessary to perform many evaluations which could be very expensive. Furthermore, using our experimental protocol enables to reproduce experiments.

Figures 4 - 7 illustrate the behavior of the algorithm during a typical learning session. On figure 4 is given the fitness of the centroid (mean) of the population with the fitness of the best search point. Notice that these values are very close and fall within the same confidence intervals (see bellow), therefore there is no evidence that choosing the best search point as the solution rather that the mean would give better results. This figure shows also the evolution of the step size (σ). A decreasing value shows that the algorithm reduces its steps as its approaches the optimum which suggest that the adaptation procedure works. This can be further supported by the other figures. The eigenvalues of the covariance matrix also converge (figure 6) as the coordinate-wise standard deviations (figure 7). Lastly figure 5 illustrates the evolution of the centroid of the population, each curve corresponds to a weight of the player s strategy. Most of the weights stabilize except for 4 and 5 (at the bottom of the curve) which seem to follow the opposite path, when one increases the other decreases and vice versa. This behaviour is under investigation.

Due to the stochastic nature of the games of Tetris, the score histogram of an artificial player using a

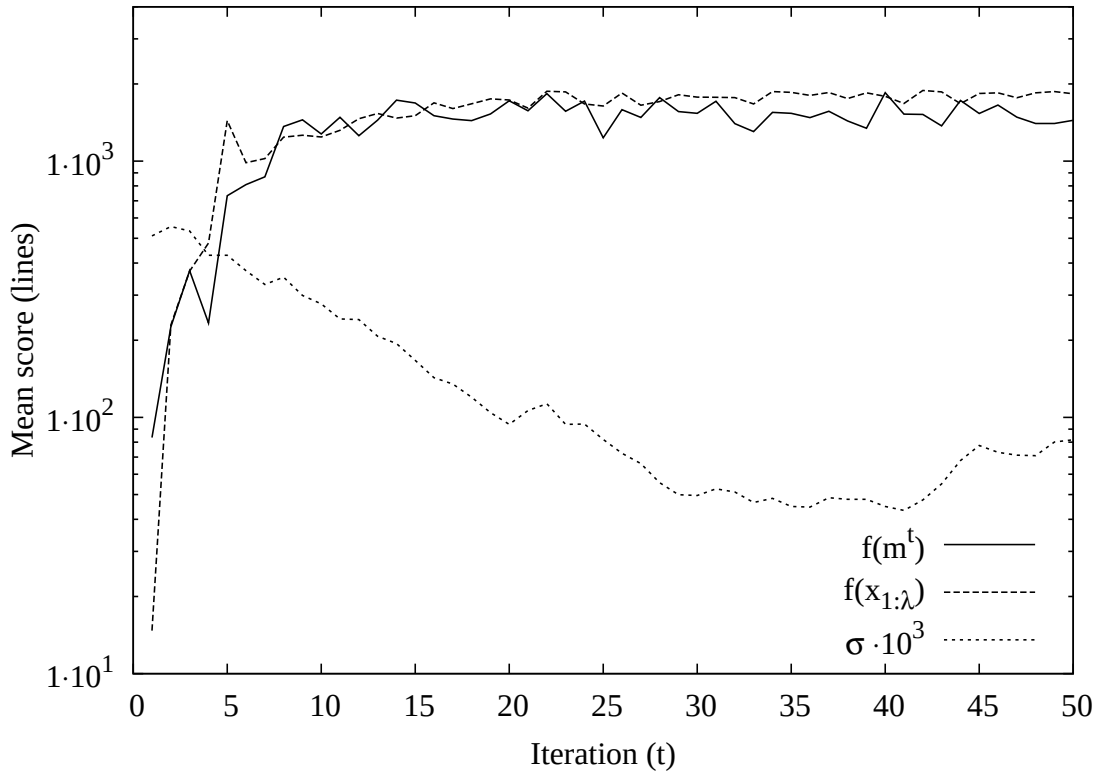

Figure 4: Fitness of the centroid (m), the best search point ($x_{i:\lambda}$) and the step size (σ) scaled up by 10^3 to fit on the figure.

fixed strategy follows a exponential distribution. This observation was confirmed experimentally by [20] where the authors argue that the score distribution of a player using a fixed strategy is an exponential distribution with 0.95 probability. Furthermore [22] extends the analysis and derives a way to compute confidence intervals of the empirical average score of the player:

$$\frac{|\mu - \hat{\mu}|}{\hat{\mu}} \leq \frac{2}{\sqrt{N}}$$

where $\hat{\mu}$ is the average score of the player, μ is the expected score and N is the number of games on which we average. Therefore when we compute the average score over 100 games, the confidence intervals is $\pm 20\%$ with probability 0.95. This is important for the remaining, since comparing different players solely on their average score does not make sens if they fall within the same confidence intervals. In the remaining all the scores presented are the average of 100 games and thus have to be considered within 20%, unless otherwise stated.

5 Results and discussion

The results presented on table 2 summarize the scores of players obtained using CMA-ES on Dellacherie s features [7] and Thiery s [22] features. As explained in section 4, these players are the mean of the weights vectors obtained by running 20 times CMA-ES. The scores are compared with the scores of the hand fixed weights from [7] obtained on the simulator we used, and the scores reported by [22]. As we can see on table 2, the player learned on CMA-ES performs better than the player using fixed weight on both games sizes. On the other hand, it performs equivalently as the one learned on NCE, the average score is higher on the 10×20 game however it falls within the same confidence interval. Similarly, NCE scores higher on the 10×16 game, but taking into account the confidence interval the scores are equivalent.

One other aspect we were interested in was the nature of the learned weight vectors. Surprisingly,

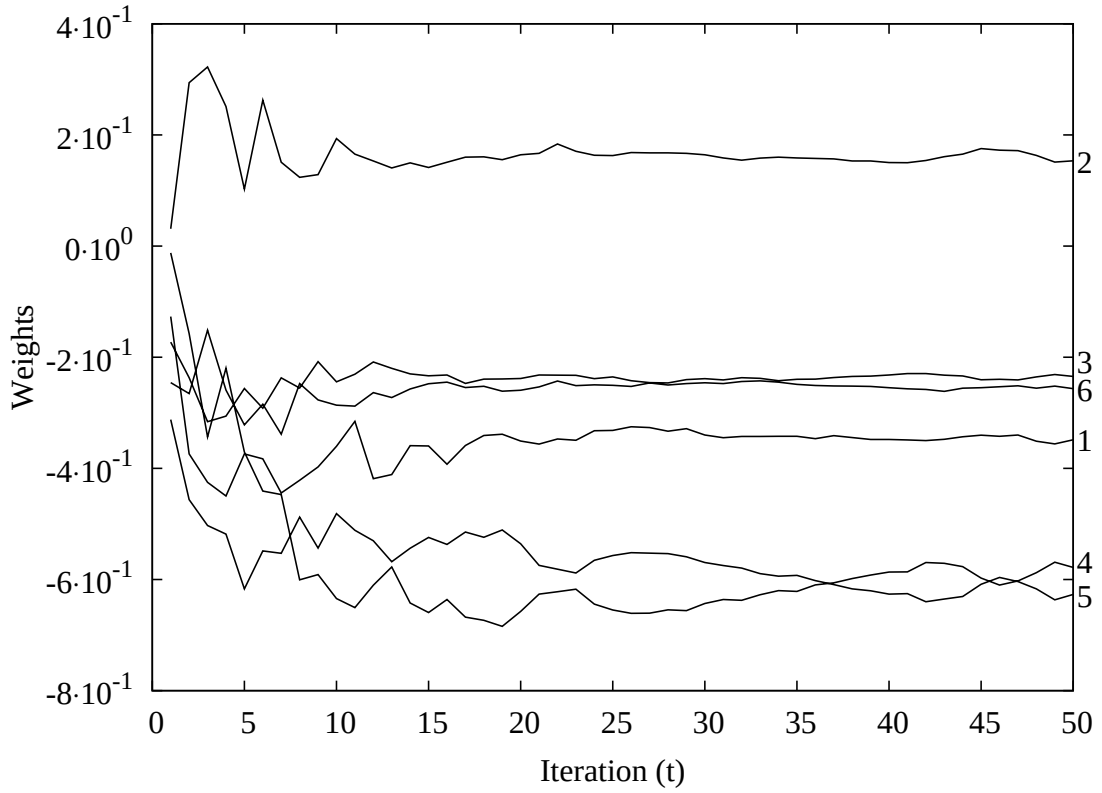

Figure 5: Components of mean vector (m).

Table 2: Scores of the players obtained by the CMA-ES.

	CMA-ES		[7]	[22]	
Game size	$f_{1...6}$	$f_{1...8}$	$f_{1...6}$	$f_{1...6}$	$f_{1...8}$
10×16	4.7 $\times 10^5$	8.7 $\times 10^5$	2.5 $\times 10^5$	5.3 $\times 10^5$	9.1 $\times 10^5$
10×20	16.6 $\times 10^6$	36.3 $\times 10^6$	6.6 $\times 10^6$	17 $\times 10^6$	35 $\times 10^6$

we noticed that even thought the search space (fitness function) is very noisy which suggest a hard optimization problem, the learned vectors throughout different independent experiments seem close. In other words the algorithm converges toward the "same" region. This is more noticeable in the case of features $f_1 \dots f_6$ than for the case of features $f_1 \dots f_8$. Figures 8 and 9 show a series of learned vectors produced by 20 independent runs using box and whiskers plot. In the case of the feature set $f_1 \dots f_8$ the quartiles are larger which suggest that the corresponding weights did not converge. This said, the average vector (the solid line) for which the score is given in table 2 performs well.

Furthermore, even though both features sets share features $f_1 \dots f_6$, the learned weights for these features do not converge similarly in both experiments. This behavior suggests that the features f_7 and f_8 conflicts somehow with f_4 and f_5. This is purely speculative since these features measure different things, however this phenomenon should be further investigated.

If we compare the learned vectors with the vectors provided by [7] and [22]. We notice that on some features our vectors have similar weights. Although we cannot compare directly these vectors since the were produced using different experimental protocols[5]. However, on the one hand this observation further support our previous argument stating that the search space may not be extremely multimodal[6]. On the other hand, it was interesting to realize that some of the weights our vectors learned are close to the weights that Dellacherie has fixed by hand. Event thought our player performs better than his when tested

[5]recall that our vectors are averaged over several runs (sec. 4) which is not the case for [22] and also not the case for [7] since they were fixed by hand

[6]This argument may only be valid for these sets of features, whoever at this time we have not tested for other features sets.

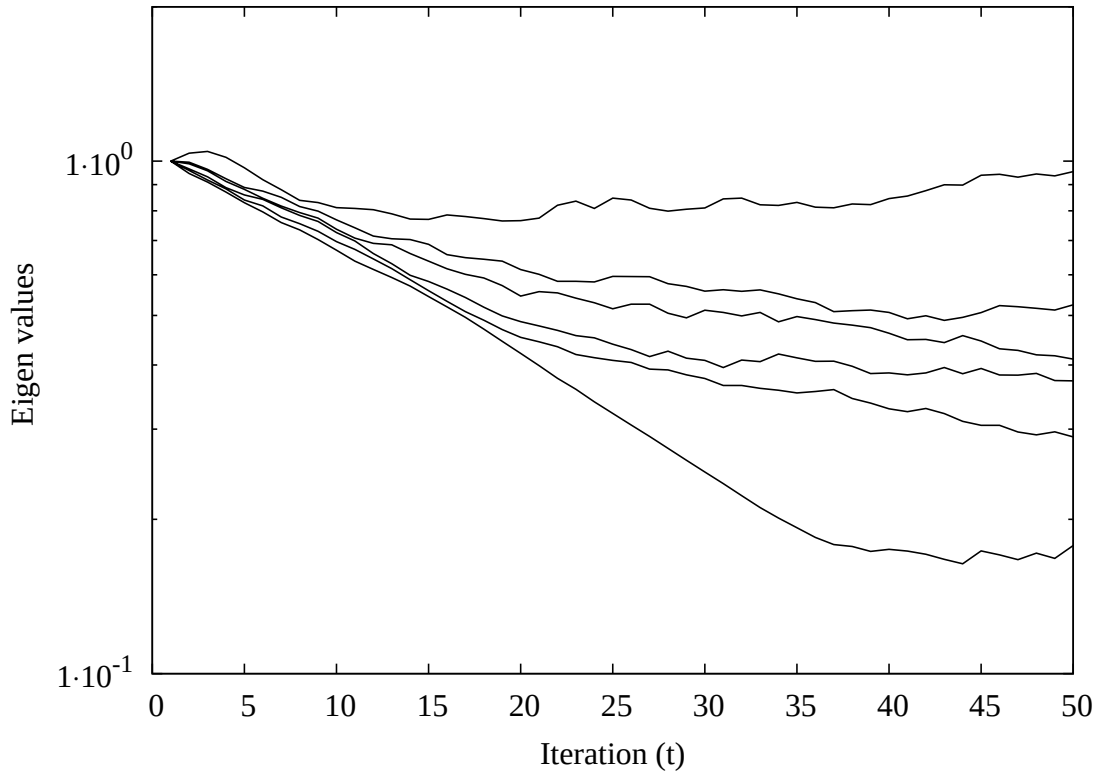

Figure 6: Square root of eigenvalues of C

on a game (table 2), the major differences in the weights resides in features f_4 and f_5 which Dellacherie weights respectively higher and lower than CMA-ES. The same conclusion can be drawn from figure 9 between the weights learned using CMA-ES and NCE, in which they seem very close.

As the players learn better strategies throughout their evolution, their scores get higher. This is usually the desired behaviour. However, for the game of Tetris playing better mean playing longer thus the evaluation time increases during the learning process which in turn slows down learning. On good player like the ones we present here, learning can last for weeks on a state of the art desktop computer. Therefore many authors adapt their algorithms to reduce this time. One way to do so, is to reduce the number of games the player plays during the evaluation step. However, this is not very efficient since the fitness of the player is less accurate.

Another way used by different authors to reduce the players evaluation time is to play smaller games, for example [16] evaluated players on games of size 6×12. Although doing so reduces significantly the learning time, our experiments show that players learned on smaller games sizes do not scale up with others learned on larger sizes. We compared two player, one learned on a 10×20 game and the other on a 10×16 game. When tested on a 10×20 games, the one that learned on this game size performs better (36.3×10^6 versus 17×10^6). We drew the same observation for other smaller size except for 10×16 where the players seems to perform similarly. We think that learning on smaller game may specialise the players for that game size and does not enables then to generalize to larger games.

As opposed to learning on smaller games, we propose different approach to shorten the evaluation time, let the player learn on "harder games" of size 10×20. These games can be obtained by increasing the probability at which "Z" and "S" pieces appear. These pieces are the hardest to place and are the ones that push the player to produce empty holes in the board. Increasing their probabilities reduces the game length significantly and the learning time on a 10×20 game drops from weeks to hours. Table 3 summarizes the scores of players learned on smaller games (size 6×12) and players learned on harder games (probability $\frac{3}{11}$ for "Z" and "S") on games of size 10×16 and 10×20. The results show that a better way to reduce the learning time is to increase the probability of "Z" and "S" pieces.

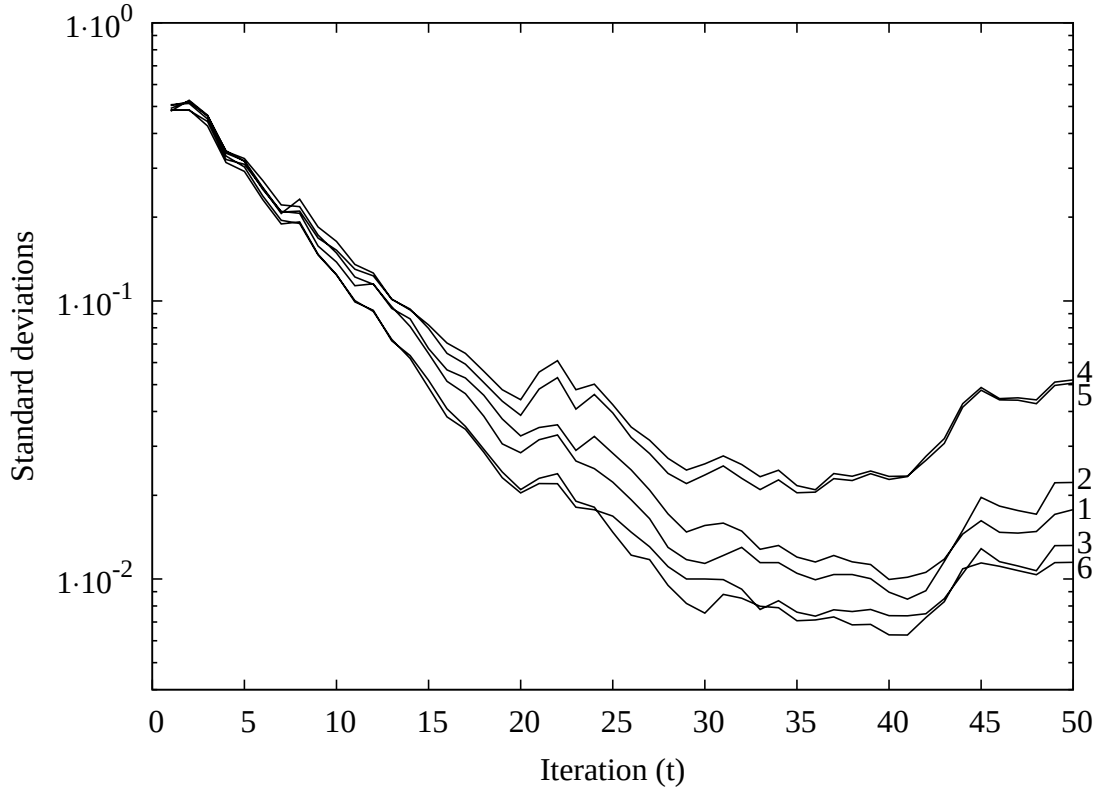

Figure 7: Coordinate-wise standard deviations ($\sigma \times \sqrt{C_{ii}}$).

Table 3: Learning on smaller vs. harder games.

Game size	Smaller games	Harder games
10×16	8.79×10^5	8.7×10^5
10×20	17×10^6	36.3×10^6

6 Conclusion and future work

Our aim was to show that evolution strategies can be applied for learning good Tetris strategies that can compete with the best reported players. However, to take full advantage of such methods, one should take good care at exploiting the properties of the problem. We have shown that by analyzing the shape of the search space, we were able to derive a method to help the algorithm converge better in this particular case. We are presently, extending this work to other classes of problems. We presented a new way to reduce the evaluation time, which can be important to speed-up the learning time, and thus push the experiments even further. We sowed also that our learned strategies are similar, to the best known strategies, in the sens that the algorithm converges to weights that are close to the ones published for the feature set we used, thus rediscovering weights that were fixed by hand by an expert, and weights that were found using a different learning scheme.

This being said, we think that designing good artificial players should not rely only in optimizing the weights of a set of features, but also in choosing good feature functions. In fact our different experiments showed us that for the feature set presented here, we have probably obtained the best player and further improvements will not be significant if we take into account confidence intervals. Improvements can be achieved using better feature functions, ones that synthesise the game better. A more interesting research question would be to learn new feature functions rather than only learn the weights. This is what we are investigating at the moment using genetic programming. These methods have proved their efficiency on different symbolic regression problems, where the goal is to find the right functions to fit a data set.

An other direction of research, would be the study of optimization on noisy fitness functions. Although,

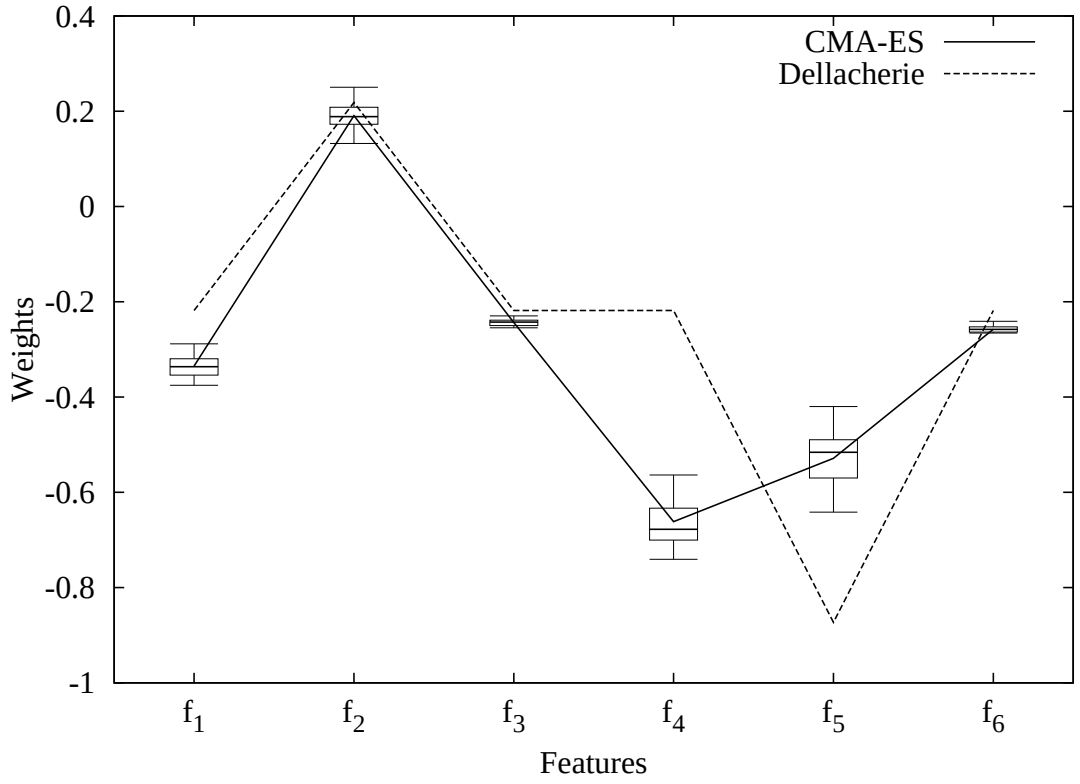

Figure 8: Learned weights of 20 independent runs of CMA-ES on features $f_{1...6}$ and the weights fixed by Dellacherie [7].

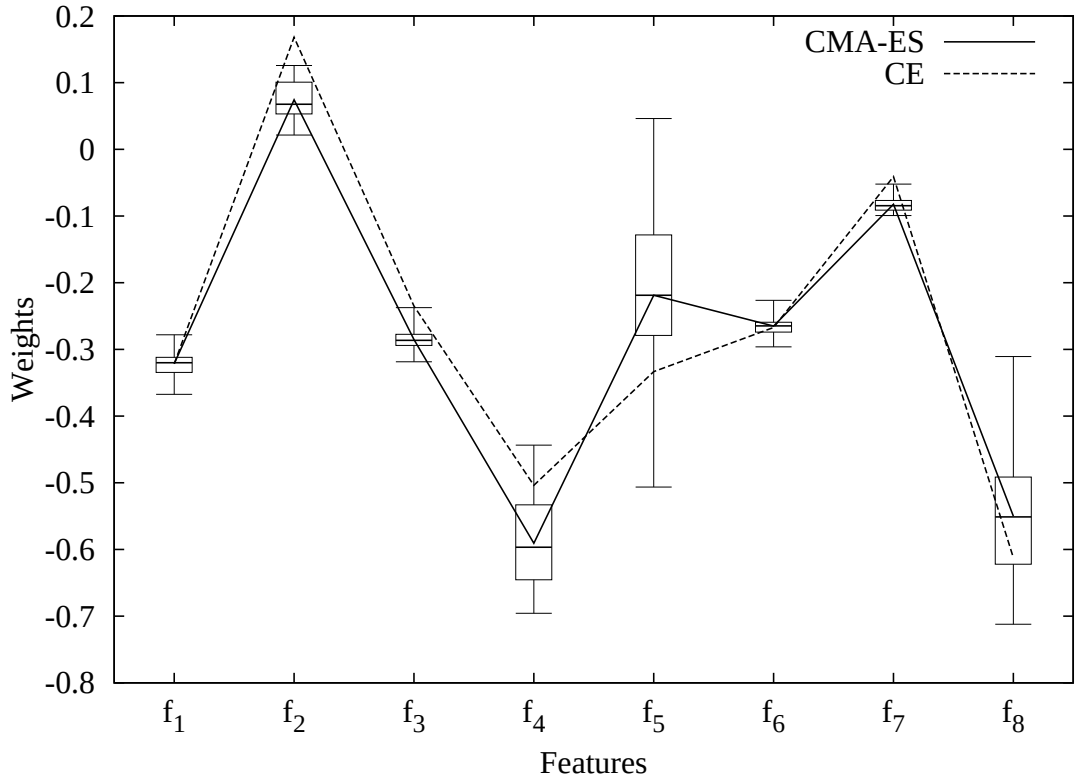

Figure 9: Learned weights of 20 independent runs of CMA-ES on features $f_{1...8}$ and the weights given by [22] learned by NCE.

there exists a substantial body of work on the subject most of it does not exploit the distribution of the fitness values, and gets around the problem by increasing the number of evaluations. For our problem, it would be better to exploit the fact that the scores follow an exponential distribution, if we could estimate it, we would be able to reduce the evaluation time since we would not have to take the average of several games.

Finally, we present in appendix the values of the weights that our Tetris player learned using CMA-ES along with the parameters we used in our experiments.

Appendix

CMA-ES parameters: $\lambda = 30$, $\mu = 15$, $\sigma^0 = 0.5$, $m^0 = 0$, number of evaluation of each offspring 100. All other parameters were kept at their default values.

Learned vectors : features $f_{1...6}$ (-0.3351050, 0.1905147, -0.2435687, -0.6617281, -0.5285897, -0.2576421). Features $f_{1...8}$ (-0.3213555, 0.07440212, -0.285174, -0.5907552, -0.2188104, -0.2650657, -0.08226664, -0.5499683).

References

[1] T. Bäck. *Evolutionary Algorithms in Theory and Practice*. Oxford University Press, 1996.

[2] D.P. Bertsekas and J. Tsitsiklis. *Neuro-Dynamic Programming*. Athena Scientific, 1996.

[3] H.-G. Beyer. and H.-P. Schwefel. Evolution strategies: A comprehensive introduction. *Natural Computing*, 1(1):3–52, 2002.

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

[5] P. de Boer, D. Kroese, S. Mannor, and R. Rubinstein. A tutorial on the cross-entropy method. *Annals of Operations Research*, 1(134):19–67, 2004.

[6] E. D. Demaine, S. Hohenberger, and D. Liben-Nowell. Tetris is hard, even to approximate. In *Proc. 9th International Computing and Combinatorics Conference (COCOON 2003)*, pages 351–363, 2003.

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

[8] V. Farias and B. van Roy. *Tetris: A study of randomized constraint sampling*. Springer-Verlag, 2006.

[9] N. Hansen. Benchmarking a BI-population CMA-ES on the BBOB-2009 function testbed. In *Workshop Proceedings of the GECCO Genetic and Evolutionary Computation Conference*. ACM, July 2009.

[10] N. Hansen and A. Ostermeier. Adapting arbitrary normal mutation distributions in evolution strategies: The covariance matrix adaptation. In *Proceedings of the 1996 IEEE International Conference on Evolutionary Computation*, pages 312–317, 1996.

[11] N. Hansen and A. Ostermeier. Completely derandomized self-adaptation in evolution strategies. *Evolutionary Computation*, 9(2):159–195, 2001.

[12] S. Kakade. A natural policy gradient. In *Advances in Neural Information Processing Systems (NIPS 14)*, pages 1531–1538, 2001.

[13] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.

[14] M. G. Lagoudakis, R. Parr, and M. L. Littman. Least-squares methods in reinforcement learning for control. In *SETN 02: Proceedings of the Second Hellenic Conference on AI*, pages 249–260, London, UK, 2002. Springer-Verlag.

[15] R. E. Llima. Xtris readme, 2005. http://www.iagora.com/~espel/xtris/README.

[16] Niko Böhm, Gabriella Kókai, and Stefan Mandl. An Evolutionary Approach to Tetris. In University of Vienna Faculty of Business; Economics and Statistics, editors, *6th Metaheuristics International Conference*, page CDROM, 2005.

[17] 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*, pages 10–14, 2004.

[18] I. Rechenberg. Evolution strategy. In J.M. Zurada, R.J. MarksII, and C.J. Robinson, editors, *Computational Intelligence imitating life*, pages 147–159. IEEE Press, Piscataway, NJ, 1994.

[19] Eric V. Siegel and Alexander D. Chaffee. Genetically optimizing the speed of programs evolved to play tetris. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, pages 279–298. MIT Press, Cambridge, MA, USA, 1996.

[20] István Szita and András Lörincz. Learning tetris using the noisy cross-entropy method. *Neural Comput.*, 18(12):2936–2941, 2006.

[21] C. Thiery and B. Scherrer. Building controllers for tetris, a review. *to aprear in The International Computer Game Association Journal*, -:–, 2009.

[22] C. Thiery and B. Scherrer. Construction d un joueur artificiel pour tetris. *Revue d Intelligence Artificielle*, 23:387–407, 2009.

[23] J. N. Tsitsiklis and B. van Roy. Feature-based methods for large scale dynamic programming. *Machine Learning*, 22:59–94, 1996.