How to design good Tetris players

Amine Boumaza

How to design good Tetris players

Amine Boumaza
Université de Lorraine, LORIA
Campus scientifique BP 239 Vandoeuvre-lès-Nancy Cedex, F-54506, France
amine.boumaza@loria.fr

Initial draft: September 10, 2011
Updated: January 8, 2014

1 Introduction

Tetris is a single player game in which the goal is to place random falling pieces onto a game board. Each horizontal line completed is cleared from the board and scores points for the player [11]. It is a very popular game with simple rules which requires, however, lots of practice and elaborate strategies for human players to perform well. Although the rules of the Tetris are very simple, designing artificial players is a great challenge for the artificial intelligence (AI) community.

For instance, Demaine *et al.* [10] proved the NP-Completeness of the offline version of the game[1]. In the offline version a deterministic finite sequence of pieces is fixed, and the player knows the identity and the order of all pieces that will be presented. The proof is based on a reduction between the 3-partition problem and Tetris. Later, Breukelaar *et al.* [6] proposed a simpler proof of the this. Intuitively these results on the off-line version of the game suggest that the on-line version, one in which the sequence is not known in advance, is of at least the same difficulty if not more. Interestingly, Burgiel [7] showed that it is impossible to win at Tetris in the sense that one cannot play indefinitely. There exist sequences of pieces, although rare, that if they occur will end the game with probability one.

When formulated as a decision problem or a learning problem, the number of states of the system becomes intractable. On the standard 10×20 game and 7 pieces, each cell in the board can be in 2 states (filled or empty) the total number of states is thus $2^{10\times 2} \times 7 \approx 10^{60}$ which approaches 10^{60} states. This large state space cannot be explored directly, which pushes the need to use approximation mechanisms and thus design sub-optimal strategies for artificial players.

All these challenges render the problem of designing artificial players very appealing for the AI community, which addressed this problem in many publications using different techniques.

In this paper, we propose to use evolutionary algorithms to design artificial Tetris players. We discuss recent results that provide very successful players scoring many millions of lines on average and compare their performance with the best known artificial players. For that we will first, describe the problem of learning Tetris strategies and review some of the existing literature on the subject (section 3) that ranges from reinforcement learning to pure optimization and evolutionary algorithms. After which we will describe how to apply evolutionary algorithms to learn such players (section 4), and how to reduce the learning time which can be of a crucial concern. Each of our arguments is illustrated on different sets of experiments. Finally, we conclude our discussion and present future ideas we intend to follow to improve our findings.

[1] One of the theorems in the paper states that maximizing the number of rows cleared in a Tetris game is NP-complete

2 The Tetris problem definition

In this section, we discuss the characteristics of the game of Tetris and the simplifications that are usually considered when designing artificial controllers. We also describe the components of the decision function that an agent uses to decide its actions.

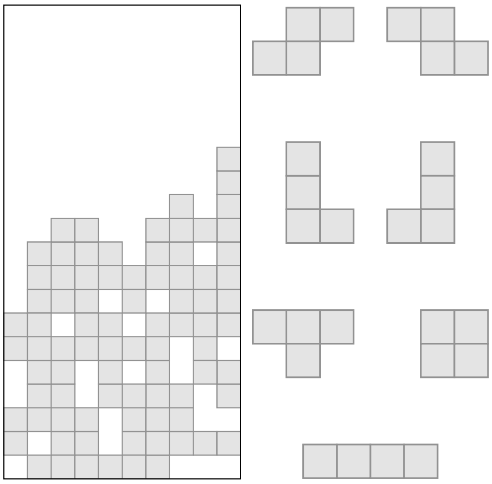

Figure 1: On the left hand side, a sample game board (the wall). The size of the standard board is 10 squares wide and 20 squares tall. On the right hand side, the 7 Tetris pieces. From top to bottom and left to right "S", "Z", "L", "J", "T", "O" and "I".

In the game of Tetris, the player is faced with placing a randomly chosen piece (Fig. 1) on a 10 × 20 game board at each iteration. Pieces can be translated from left to right and rotated. When a piece is placed the player is faced with a new piece. When a row of the board is fully occupied, it is cleared and all pieces above the row collapse by one row. The game ends when the height of the wall reaches the top of the board. The aim of the player is to clear as many line as possible to delay the end of the game.

In the standard game, the pieces fall from the top of the game board with a certain speed that increases with the game level. The player has to decide where to place the falling piece before it reaches the other pieces in the board at which time it stops. The player has to plan its actions within this laps of time, which gets shorter as the player advances in the game.

2.1 Game simplifications

In the existing literature addressing learning Tetris controllers, authors usually address a simplified version of the problem where the dynamics of the falling pieces is not taken into account.

In the simplified version, the artificial player is given the current piece to place and the current state of the board and it has to decide where to place the piece. A decision in this case is a rotation (the orientation of the piece) and a translation (the column) to apply on the current piece. The player tests all possible rotations and all possible translations and decides, based on a decision function, on the best decision to take. Incidentally, this simplification does not affect the performance of artificial players on the original game as their decisions are usually made well within the shortest laps of time a piece takes to fall.

Furthermore, except for few authors, most of the existing works address the so called "one piece strategies", where only the current piece is known and not the subsequent one, which is the case in the standard game. As it will be clearer in few paragraphs the extension to the original game's "two pieces strategy" is straightforward.

2.2 The decision function

In general, a Tetris game-playing agent decides which action (rotation and translation) to choose at a given state of the game based on the reward it will gain after performing that action. The agent chooses, among the several possible actions, the one that will return the best outcome. This reward may be instant i.e. at the next state of the game or delayed until the end of the game.

It is however difficult and sometimes not possible to judge the value of an action, and one way around this is to judge the value of the state to which the action leads. Generally this is achieved using an evaluation function that assigns numerical values to game states. The player then chooses among all possible actions the one that leads to the best valued state.

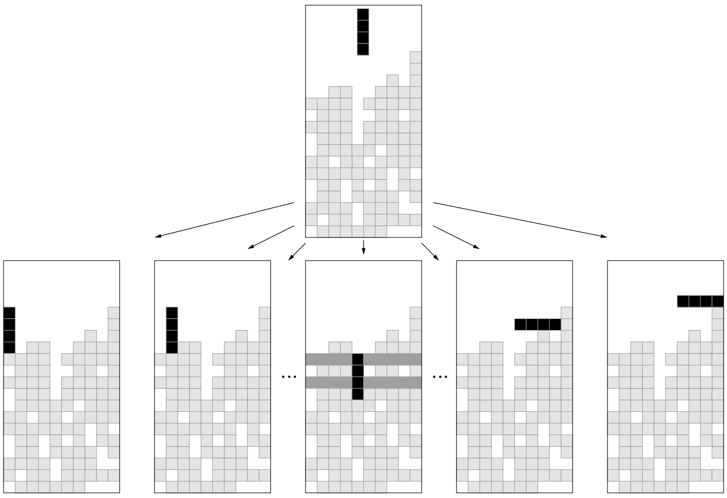

Figure 2: The decision tree of an artificial Tetris player exploring all possible actions for the current piece and evaluating all the outcomes.

Given the current piece to place, the agent will choose the decision that will give the "best board", as illustrated on Figure 2. When a piece is to be placed, the agent simulates and evaluates all the possible boards that would result from all the possible moves of the piece, and then chooses the action that leads to the best valued game board (greedy strategy). In a "two pieces strategy", the player simulates two moves (one for the current and one for the subsequent piece) and selects the best one for the current piece. In this case the decision tree illustrated on Figure 2 is extended by an additional level.

Note that since the agent follows a greedy strategy, the evaluation function is the only component that enters into the decision making process. Hence, designing a Tetris player amounts to designing an evaluation function. Such a function should synthesize the game state giving high values to "good boards" and low values to "bad" ones. In this context "good boards" are ones that increase the chance of clearing lines in future moves.

We note $s \in S$ a state of the game, s represents the configuration of the board at a given time. The evaluation function $V : S \to \mathbb{R}$ maps a state s to a number that estimates the value of the state s. Let us also consider P the set of the seven pieces and A the set of all possible actions. An action is a rotation of which there are 4 possibilities and a translation of which there are 10 possibilities[2] i.e. $|A| \leq 40$.

Further, we note $a(p) \in A$ an action applied to the piece $p \in P$. We also note $\gamma : S \times A \to S$ the transition function that maps a pair $(s, a(p))$, a state s and an action $a(p)$, to a new state of the game after taking action a for the piece p in state s.

The player's decision function $\pi(s,p) : S \times P \to A$, will select the action with the highest valued outcome:

$$\pi(s,p) = \hat{a}(p) = \arg \max_{a(p) \in A} V(\gamma(s, a(p))). \tag{1}$$

[2] The actual number of possible actions may depend on the shape of the current pieces and the size of the game.

2.3 Feature functions

A state evaluation function V may be considered as mimicking the evaluation a human player makes when deciding where to drop the piece. It should rank higher game states that increase the chance to clear lines in subsequent moves. For example, such game boards should be low in height and should contain few holes (empty squares). Since the set of all possible game states is too large to be considered, a state is generally projected in a smaller set of features. These feature functions represent the most salient properties of a given state. These properties may be the height of the wall, the number of holes, etc. In the literature, many authors have introduced different features to synthesize different aspects of a game state. A comprehensive review of these features can be found in [34].

The evaluation function V is a combination of these feature functions using generally a weighted linear sum. If we let f_i and w_i, with $i = 1 \ldots n$, denote the n feature functions and their respective weights. Each f_i maps a state s to a real value. The evaluation function V is then defined as:

$$V(s) = \sum_{i=1}^{n} w_i f_i(s) \tag{2}$$

Other combinations of evaluation functions are also possible. For example, Langenhoven *et al.* [22] use a feed forward neural network to which the value of each feature function is an input node. Böhm *et al.* [5] proposed exponential combinations where they considered the feature functions powered to given constants.

In the remainder of this paper we consider eight features which are detailed on fig. 3. Six ($f_{1\ldots6}$) of these were introduced by Dellacherie [11], and two (f_7 and f_8) were introduced by [33].

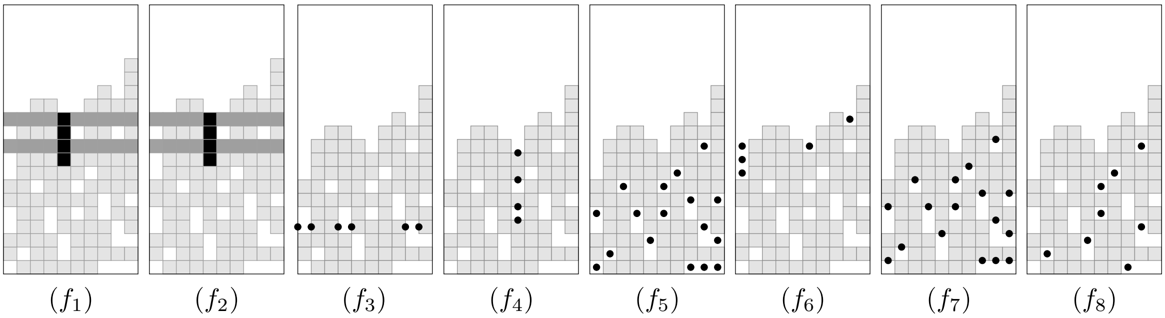

Figure 3: Features functions. (f_1) Landing height: the height at which the current piece fell (8 in the figure). (f_2) Eroded pieces: the contribution of the last piece to the cleared lines time the number of cleared lines (2×2 in the figure). (f_3) Row transitions: number of filled cells adjacent to empty cells summed over all rows (58 in the figure). (f_4) Column transition: same as (f_3) summed over all columns (45 in the figure) note that borders count as filled cells. (f_5) Number of holes: the number of empty cells with at least one filled cell above. (f_6) Cumulative wells: the sum of the accumulated depths of the wells ($(1+2+3)+1+1$ on the figure). (f_7) Hole depth: the number of filled cells above holes summed over all columns. (f_8) Row hole: the number of rows that contains at least one hole (8 in the figure).

3 Related work on designing Tetris players

This section presents some of the works that addresses the design of artificial Tetris players. As described above, to design evaluation functions based controllers, one needs to select the set of feature functions, the way to combine them into the evaluation function and the parameters of this combination.

In most of the literature on the subject, the first two steps are generally performed by the expert designer and the only part left where learning algorithms are applied is the setting of the evaluation function's parameters. In other words, learning an artificial player for Tetris amounts to learning the correct set of parameters of the evaluation function. This task has been performed using different methods, that is what we will review briefly in the following.

3.1 Hand-tuning players

When the designer has enough insight about the desired behavior of the artificial Tetris player, it is possible to tune the parameters of the evaluation function by hand and experimentation. Dellacherie [11] proposed a set of six feature functions that were combined linearly ($f_1 \dots f_6$ in figure 3). He fixed the combination weights by hand based on the importance he gave to the individual features. Interestingly enough this controller was, until not long ago, one of the best performing ones, clearing 63×10^4 lines on average.

3.2 Learning to play

Others have used machine learning methods to learn correct values of the weights of the feature functions. Most of the works to this end use reinforcement learning algorithms. Although little success has been achieved when compared to Dellacherie's hand tuned player; it however, provides insights on the difficulty of the task when the problem is formulated as a decision problem. In the following, we cite some of the works designing Tetris controllers, using reinforcement learning. A more comprehensive review may be found in Donald Carr's thesis [8].

Among the first authors that addressed the problem of designing Tetris controllers, Tsitsiklis and Van Roy [36] applied a feature based value iteration algorithm and reported a score of 30 cleared lines on average. The algorithm operates in the space of features rather than in the state space where the problem is intractable. Incidentally, the authors used only two features (the maximum height of the wall and the number of holes). At the same time, Bertsekas and Tsitsiklis discuss in their book [3] the use of policy iteration algorithms, more precisely the λ-policy iteration algorithm. This algorithm iteratively refines a policy (playing strategy) using simulations. The authors chose a set of $2w + 2$ features where w is the width of the game (22 features on the standard game). They extended Tsitsiklis and Van Roy's features with ones that represent the individual heights of each column and the differences of heights between two adjacent columns, and reported average scores around $3200 \pm 20\%$ lines. Later, on this same feature set, Kakade [18] applied a natural policy gradient method and reported scores of 6800 cleared lines on average. Farias and Van Roy [12] proposed to use linear programming for random constraints sampling and reported scores of 4700 lines on average.

Finally, Lagoudakis *et al.* [21] proposed a reduced set of features with only ten: the maximum height, the total number of holes, the sum of absolute height differences between adjacent columns, the mean height, and the change of these quantities in the next step, plus the change in score and a constant term. Using these features, the authors applied the least square policy iteration algorithm and reported scores between 1000 and 3000 lines on average. On this same feature set, Thiery and Scherrer [35] used the least square-λ-policy iteration algorithm, a generalization of the aforementioned λ-policy in which the algorithm uses a linear approximation of the state space and reported scores of 800 lines.

3.3 Optimizing players

Some authors considered the problem differently and proposed to set the parameters of the evaluation function using optimization techniques. For instance, Szita and Lõrincz [31] applied the noisy cross entropy method algorithm (NCE) [9] to optimize the weights of the features introduced by [3] and reported a score of 35×10^4 $\pm 36\%$. The algorithm treats the problem as a search in the space of all possible weight vectors ($\mathbb{R}^{22}$) for the one that maximizes the average score. Initially random, the search point is moved according to a normal distribution at each step. Incidentally, this method closely resembles evolution strategies [27, 2, 4], with some differences in the step-size update rule.

More recently Thiery and Scherrer [33] used the same NCE algorithm to optimize the weights proposed by Dellacherie and other weights they proposed. They reported the impressive score of $35 \times 10^6 \pm 20\%$

lines on average. Furthermore, the authors give a review of the existing works on learning Tetris players and argue on the difficulty to compare different players using different interpretation of the game since small implementation 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 in the literature so that algorithms can be compared on the same basis. The work presented here uses the same simulator.

3.4 Evolving players

In the same spirit of the work above, some authors propose to optimize the parameters of the evaluation function using evolutionary computation.

To our knowledge Siegel and Chaffee [29] were the first to propose such a design method that uses genetic programming [20]. 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[3]. Llima [23] used genetic algorithms to optimize the weights of the artificial player proposed on the GNU Xtris simulator. The evaluation function uses six features: the maximum height of the wall, the number of holes, the depth at which the current piece is dropped, the length of the frontier between empty and filled squares[4], cumulative wells (f_6 in figure 3) and the sum of the depths of all places where a long piece (I) would be needed. The evolved controller scores 5×10^4 on average.

Furthermore, Böhm *et al.* [5] used an evolutionary algorithm to design Tetris players for which the authors proposed new features along with Dellacherie's features and three different evaluation functions: a linear combination as in equation (2), an 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, reflecting the fact that the height of the wall is less critical when the wall is at the bottom of board. The authors considered the two piece game for which they reported scores in the millions lines. More recently, Langenhoven *et al.* [22] proposed to use particle swarm optimization (PSO) [19] to optimize the weights of a feed forward neural network used as an evaluation function. The values of the feature functions are inputs to the network of which the output is the value of the state. The PSO algorithm is used to optimize the weights of the neural network. The authors reported scores around 15×10^4 lines on average.

4 Evolution strategies to learn Tetris

In this work, we propose to use evolutionary algorithms, more specifically an evolution strategy [27, 2, 4] to learn the weights of an evaluation function for a Tetris player using the features of figure 3. We will begin by introducing briefly evolution strategies (ES), and the variant we use here: the covariance matrix adaptation evolution strategy (CMA-ES) [14, 15] and will then detail how we proceed to evolve artificial Tetris players.

4.1 The CMA evolution strategy

The covariance matrix adaptations evolution strategy is an evolutionary algorithm that operates in continuous search spaces, where the objective function f can be formulated as follows:

$$\hat{\mathbf{x}} = \arg\max_{\mathbf{x}} f(\mathbf{x}) \text{ with } f : \mathbb{R}^n \to \mathbb{R}.$$

It can be thought of as continuously updating a search point $m \in \mathbb{R}^n$ that is the centroid of a normally distributed population. The progress of the search point controls how the search distribution is updated over time to help the convergence of the algorithm.

Algorithm 1 describes the general procedure of CMA-ES. At each iteration, the algorithm samples λ points (the offspring) from the current distribution (line 5) : $\mathbf{x}_i^{t+1} \sim \mathbf{m}^t + \sigma^t \mathbf{z}_i^t$ and $\mathbf{z}_i^t \sim \mathcal{N}(0, \mathbf{C}^t)$ $(i = 1 \cdots \lambda)$,

[3] To be more precise their artificial player does not use an evaluation function, the evolved programs use strategies of their own discovered through evolution.

[4] For one empty square surrounded by filled squares, the value of the frontier is 4.

where $\mathbf{C}^t$ is the covariance matrix and $\sigma > 0$ is a global step-size. The $\mu \leq \lambda$ best of the sampled points (the parents) are recombined by a weighted average (line 11) where $w_1 \geq \cdots \geq w_\mu > 0$ and $\sum_{i=0}^{\mu} w_i = 1$ to produce the new centroid $\mathbf{m}^{t+1}$.

Algorithm 1 Procedure $(\mu/\mu, \lambda)$-CMA-ES

```
Require: $(\lambda, \mathbf{m}, \sigma, f, n)$
 1: Begin
 2: $\mu := \lfloor \lambda/2 \rfloor$, $\mathbf{m}^0 := \mathbf{m}$, $\sigma^0 := \sigma$, $\mathbf{C}^0 := \mathbf{I}$, $t := 1$, $\mathbf{p}_c^0 = \mathbf{p}_\sigma^0 = 0$
 3: repeat
 4:   for $i := 1$ to $\lambda$ do
 5:     $\mathbf{z}_i^t \sim \mathcal{N}_i(0, \mathbf{C}^t)$ {Sample and evaluate offspring}
 6:     $\mathbf{x}_i := \mathbf{m}^t + \sigma^t \times \mathbf{z}_i^t$
 7:     $f(\mathbf{x}_i) := \mathtt{eval}(\mathbf{x}_i)$
 8:   end for
 9:   $\mathbf{m}^{t+1} := \sum_{i=1}^{\mu} w_i \mathbf{x}_{i:\lambda}$ {Recombine $\mu$ best offspring, $f(\mathbf{x}_{1:\lambda}) \leq \ldots \leq f(\mathbf{x}_{\mu:\lambda})$}
10:   $\mathbf{p}_\sigma^{t+1} := (1 - c_\sigma)\,\mathbf{p}_\sigma^t + \sqrt{c_\sigma\,(2 - c_\sigma)\,\mu_{eff}}\,(\mathbf{C}^t)^{\frac{1}{2}}\,\frac{\mathbf{m}^{t+1} - \mathbf{m}^t}{\sigma^t}$ {Update $\sigma$}
11:   $\sigma^{t+1} := \sigma^t \exp\left(\frac{c_\sigma}{d_\sigma}\left(\frac{\|\mathbf{p}_\sigma^{t+1}\|}{\mathbb{E}[\|\mathcal{N}(0,\mathbf{I})\|]} - 1\right)\right)$
12:   $\mathbf{p}_c^{t+1} := (1 - c_c)\mathbf{p}_c^t + \sqrt{c_c\,(2 - c_c)\,\mu_{co}}\frac{\mathbf{m}^{t+1} - \mathbf{m}^t}{\sigma^t}$ {Update $\mathbf{C}$}
13:   $\mathbf{C}^{t+1} := (1 - c_{co})\,\mathbf{C}^t + \frac{c_{co}}{\mu_{co}}\mathbf{p}_c^{t+1}\left(\mathbf{p}_c^{t+1}\right)^T + c_{co}\left(1 - \frac{1}{\mu_{co}}\right)\sum_{i=1}^{\mu} w_i\left(\mathbf{z}_{i:\lambda}^t\right)\left(\mathbf{z}_{i:\lambda}^t\right)^T$
14:   $t := t + 1$
15: until stopping_criterion
16: return $\mathbf{m}^t$
17: End
```

Adaptation of the search distribution in CMA-ES takes place in two steps, first updating the mutation step and then updating the covariance matrix. The step-size adaptation is performed using a cumulative step-size adaptation [26], where the evolution path accumulates an exponentially fading path[5] of the mean m^t (line 13) where the backward time horizon is determined by c_σ^{-1}. The adaptation of the covariance matrix C^t takes into account the change of the mean (rank-1 update), and the successful variations in the last generation (rank-μ update) (respectively lines 16 and 17). An in depth discussion on setting the parameters c_σ, d_σ, c_c, μ_{eff}, μ_{co}, c_{co} and their recommended values can be found in [13].

4.2 Some considerations

An important aspect of using evolutionary computation in problem solving is to gather as much information about the problem 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 give some ideas about the shape of the search space we are addressing.

4.2.1 Weights normalization

Since the evaluation function (equation 2) is a linear combination, let us consider two actions a_1 and a_2 to place a piece p in state s such that $V(s_1 = \gamma(s, a_1(p))) > V(s_2 = \gamma(s, a_2(p)))$.

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 $\|w\| \hat{w} \cdot f(s_1) > \|w\| \hat{w} \cdot f(s_2)$ or $\hat{w} \cdot f(s_1) > \hat{w} \cdot f(s_2)$. This shows that multiplying a weight vector by a constant will not change the ranking of the decisions. In other words, all vectors $c \times w$, with c constant are equivalent with respect to the decisions of the player.

This is 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

[5] The comparison of the travelled path with a path under random selection is used to adjust the step-size.

performance of an evolutionary strategy that bases the adaptation of the step size and the adaptation of the covariance matrix on the progress of the search points, using path accumulation.

Experiments show that in this case the step sizes do not converge and keep increasing, which suggests that the algorithm is following an unattainable moving target. This behavior is illustrated on figure 4 where both the coordinate-wise step sizes and the solution (m^t) diverge.

One way, that we propose, to avoid such a behavior, is to add 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 hyper-cube. 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 algorithm.

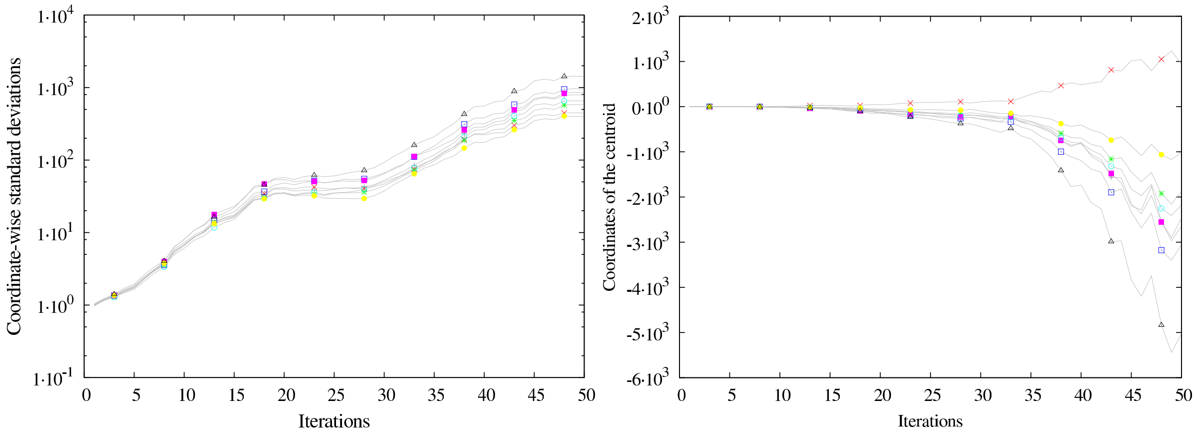

Figure 4: Divergence of the coordinate-wise step sizes in log-scale (left) and the solution variables (right). Typical run without normalization.

4.2.2 Fitness landscape

One other feature of the search space worth mentioning is linked to the score distribution of a Tetris player. As stated above, the score of a Tetris player follows an exponential distribution. As a consequence, each point of the fitness landscape is a random variable. On stochastic objective functions, the fitness values of each individual follow a usually unknown distribution. On such problems, one cannot rely on a single evaluation of the offspring which is usually not a reliable measure, but needs to make several ones and compute statistics to estimate the true fitness.

In algorithms that rely only on the ranking of the population, as it is the case for CMA-ES, ranking based on fitness values is not appropriate if the variability of the values is not taken into account. In such cases the ranking may not be correct and may lead to poorly selected parents.

There is a thriving literature on methods to tackle these class of fitness landscapes [17]. For example [13] proposes UH-CMA-ES which, among other things, increases the population variance when a change in the ranking of the population is detected i.e. when noise is detected. Others [30, 28] proposed methods to reduce the number of evaluations applicable when the distribution of the noise in the fitness function is of a particular type.

In section 5, we will describe more elaborate methods to reduce the number of evaluations, for the time being we will present a simple version of the learning strategy that estimates the fitness by averaging over fixed number of evaluations.

A candidate solution is evaluated (line 6 of the algorithm1) by playing a certain number of games and its fitness value is the average score. Furthermore, since selection is based on ranking, if the offspring plays different games their ranking may be flawed. In order to provide a coherent ranking of the offspring, we evaluate them on the same series of games. At each iteration, a series of independent games (random seeds) are drawn randomly and each individual of the λ offspring plays the same games. This way we can assess of the offspring's performance on the same setting.

4.3 Illustrative experiments

The experiments we conducted are divided in two parts: a learning phase and an evaluation phase. In the learning phase, the weights of the player's strategy are learned using CMA-ES. The player's strategy is then evaluated on a series of games which assess its performance. The games were simulated on the MDPTeris platform[6]. All the expriments were repeated 50 times and the curves represent median values.

Before going into further details, we begin by discussing the score distribution of the game of Tetris. Due to the stochastic nature of the game, the score histogram of an artificial player using a fixed strategy follows an exponential distribution. This observation was confirmed experimentally by [31] where the authors argue that this distribution is an exponential distribution with 0.95 probability. Furthermore [33] 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 ±20% with probability 0.95. This is important for the remaining, since comparing different players solely on their average score does not make sense 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.

In our experiments, a learning episode goes as follows: starting with a zero vector ($m^0 = 0$) and given an initial step size (typically $\sigma^0 = 0.5$), we run the algorithm for a fixed number of generations, in the presented experiments 50 iterations. At the end of one run we record the mean vector produced and we repeat this procedure for a certain number of times typically 20, thus ending with 20 mean vectors. Finally, the average of these mean vectors is the player's strategy we evaluate.

This experimental protocol is different from what is generally used in the literature. Often authors choose the best vector of one run as the player's strategy, which in our sense is not reliable due the nature of the score distribution. In our experiment, the fitness of each individual is the average of 100 games. As it is shown on the left of Figure 5 the fitness of the mean and the best individual fall within the 20% confidence interval. Furthermore, using our experimental protocol enables to reproduce experiments.

4.4 Comments

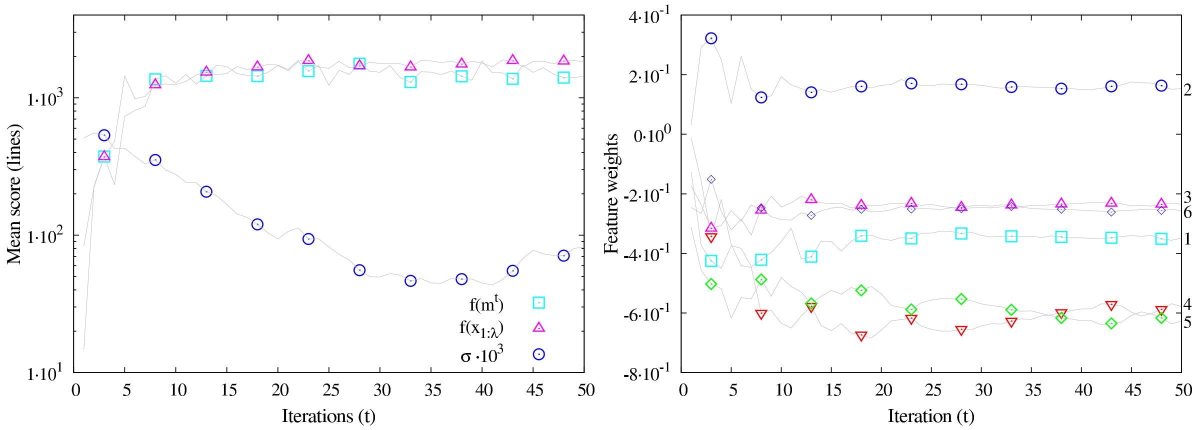

Figure 5: (Left) Fitness of the centroid (m), the best search point ($x_{1:\lambda}$) and the step size (σ) scaled up by 10^3 to fit on the figure. (Right) Weights of the features function (components of mean vector m)

[6] MDPTetris is available at http://mdptetris.gforge.inria.fr

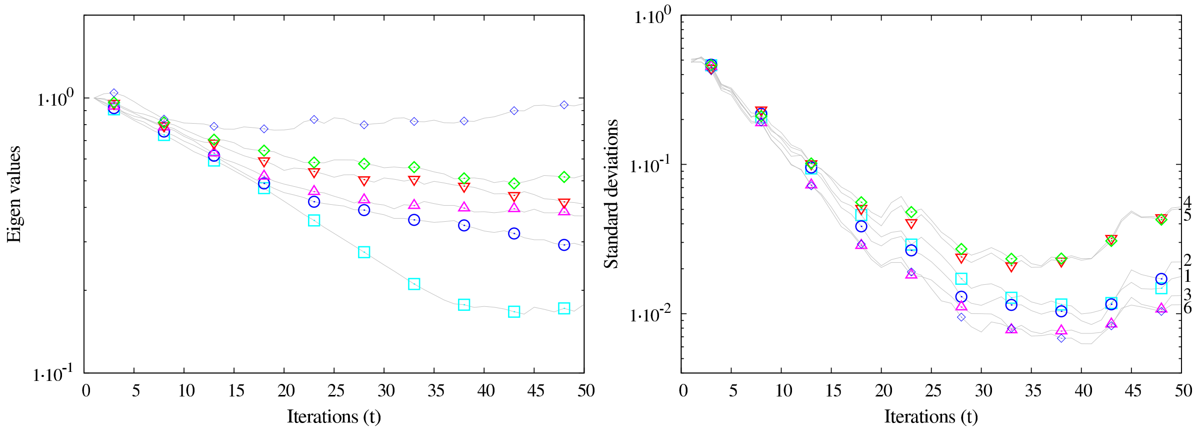

Figure 6: Square root of eigenvalues of C (left) Coordinate-wise standard deviations $\sigma \times \sqrt{C_{ii}}$ (right)

Figures 5 and 6 illustrate the behavior of the algorithm during a typical learning session. On the left of Figure 5 is given the fitness of the centroid (mean) of the population with the fitness of the best search point. The left hand side of Figure 5 shows also the evolution of the step size (σ). A decreasing value shows that the algorithm reduces its steps as it approaches the optimum, which suggests that the adaptation procedure works. This can be further supported by Figure 6, on which the eigenvalues of the covariance matrix also converge as do the coordinate-wise standard deviations.

Lastly, the right part of 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 behavior is under investigation.

The results presented in Table 1 summarize the scores of players obtained using CMA-ES on Dellacherie's features [11] and Thiery's [33] features. As explained in section 4.3, 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 [11] obtained on the simulator we used, and the scores reported by [33].

Table 1: Scores of the players obtained by the CMA-ES.

	CMA-ES		[11]	[33]	
Size	$f_{1\ldots6}$	$f_{1\ldots8}$	$f_{1\ldots6}$	$f_{1\ldots6}$	$f_{1\ldots8}$
10×16	4.7×10^5	8.7×10^5	2.5×10^5	5.3×10^5	9.1×10^5
10×20	16.6×10^6	36.3×10^6	6.6×10^6	17×10^6	35×10^6

As we can see on Table 1, the player learned on CMA-ES performs better than the player using hand fixed weights on both games sizes. On the other hand, it performs equivalently with 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.

When the game size is small enough, an optimal policy may be computed. This has been done by Thiery and Scherrer [32] on the board of size 5×5 using the value iteration algorithm. On this configuration, they reported a mean score of 13.8±0.6% lines measured over 10^5 games. Interestingly we observe a similar average score (13.83974) on the controller learned on this game configuration. This suggest that players learned by CMA-ES are very close to the optimal player on the board of size 5×5. Using Dellacherie's features we obtain slightly lower average scores (13.2959).

5 Tetris is hard to learn !

Although a fascinating research question, designing artificial players for the game of Tetris can be a frustrating experience. This is due to different facts of the game. For instance, evaluating a decent player can take many hours on a modern computer which makes experiments on methods that rely on extensive evaluations such as evolutionary computation very time consuming. Furthermore, when candidate players learn better strategies throughout their evolution, the games last longer. On good players like the ones we present here, learning can last weeks on a state of the art desktop computer. Therefore the sensible thing which many authors often do is to adapt their algorithms to reduce the evaluation time.

One way to do so, is to reduce the number of games the player plays during the evaluation step. This however is not desirable since the estimation of the fitness of the player is less accurate. There are few methods to reduce the length of the evaluation time. In the following, we describe two, which helped us reduce drastically the length of our experiments. The first method changes the rules of the game to make it harder to play which consequently pushes the players to loose faster thus reducing the length of the evaluations. The second method reduces the number of games played such that only the necessary number of games for an accurate evaluation is played. Incidentally, both these methods can be combined.

5.1 Reducing the game length

To reduce the length of the games some authors have reduced the board size. When the board is smaller, the player looses faster. For example Böhm *et al.* [5] evaluated players on games of size 6 × 12 during the learning process. Doing so reduces significantly the learning time however, our experiments showed that players learned on smaller game sizes do not scale up with others learned on larger ones.

We compared two players, one learned on a 10 × 20 game and the other on a 6 × 12 game. When tested on a 10 × 20 game, 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 seem to perform similarly. We think that learning on smaller games may specialize the players for that game size and does not enable them to generalize to larger games.

Rather than learning on smaller games, we propose a 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 (figure 1). These pieces are the hardest to place and are the ones that push the player to produce many holes in the board. Increasing their probabilities reduces the game length significantly and the learning time on a 10 × 20 game drops from months to hours.

Table 2 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.

Table 2: Learning on smaller vs. learning on 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

5.2 Reducing the number of evaluation

As we argued in section 4.2.2, evaluating the offspring population should be done on multiple games since the fitness function follows an exponential distribution. However, in this case the evaluation time of the population is increased and can lead to long experiments. In this section we describe how to reduce significantly the number of evaluations needed to evaluate the fitness of the population while preserving the accuracy of the population ranking.

In order to reduce the number of re-evaluations, we need to disregard all those that are not necessary and that do not improve the information we have on the offspring fitness. In other words reevaluations are not wasted on poorly performing individuals once these are detected.

One way to perform this is to take into account confidence bounds around the estimated fitness value using racing methods [24, 1, 25]. These methods adapt the number of evaluations dynamically until the algorithm reaches a reliable ranking in the population. [16] proposed to use such methods on evolution strategies and reported some success on different machine learning problems. These methods reevaluate only individuals that are statistically indistinguishable from each other *i.e.* individuals whose confidence intervals (CI) around the mean fitness overlap.

Reevaluation in this case is used to reduce the confidence intervals. Once enough individuals with non overlapping intervals are found, the procedure stops. When the distribution of the fitness function is not known, this confidence intervals are computed using empirical methods using for example Hoeffding or Bernstein bounds which will be described below.

5.2.1 Computing confidence bounds

The racing procedure we propose to use is inspired from [16]. At each iteration of Algo 1, the λ offspring undergo multiple evaluations until either : 1) μ outstanding individuals get out of the race, in which case we are sure (with probability $(1-\delta)$) that they are distinct. Or 2) the number of evaluations reaches some upper limit r_{limit} in which case the evaluation budget is exhausted. Furthermore, the upper bound r_{limit} is adjusted by the algorithm. If the actual number of evaluations necessary to distinguish μ individuals is less than the limit r_{limit}, this one is reduced:

$$r_{limit} := \max(1/\alpha\, r_{limit}, 3) \tag{3}$$

with $\alpha > 1$, and 3 is the minimum number of evaluations. On the other hand if r_{limit} evaluations where not enough to distinguish μ individuals, then r_{limit} is increased:

$$r_{limit} := \min(\alpha\, r_{limit}, r_{max}) \tag{4}$$

where r_{max} is a maximum number of evaluations. Initially $r_{limit} = 3$ and it is adapted at each iteration of Algo 1 using the above two rules.

After r re-evaluations, the empirical bounds $c_{i,r}$ around the mean fitness of each individual $\hat{X}_{i,r} = \frac{1}{r}\sum_{j=1}^{r} f^j(x_i)$ where $i = 1 \dots \lambda$, are computed with:

$$c^h_{i,r} = R\sqrt{\frac{\log(2n_b) - \log(\delta)}{2r}} \tag{5}$$

using the Hoeffding bound and

$$c^b_{i,r} = \hat{\sigma}_{i,r}\sqrt{2\frac{\log(3n_b) - \log(\delta)}{r}} + 3R\frac{\log(3n_b) - \log(\delta)}{r} \tag{6}$$

using Bernstein.

Where $n_b \leq \lambda r_{limit}$ is the number of evaluations in the current race, $\hat{\sigma}^2_{i,r} = \frac{1}{r}\sum_{j=1}^{r}\left(f^j(x_i) - \hat{X}_{i,r}\right)^2$ is the standard deviation of the fitness for individual x_i, and $(1-\delta)$ is the level of confidence we fix. $R = |a - b|$ such that the fitness values of the offspring are almost surely between a and b. These two constants are problem dependent.

After each evaluation in the race the lower bounds $lb_{i,r}$ and the upper bounds $ub_{i,r}$ around the mean of each offspring are updated to the tightest values:

$$lb_{i,r} = max\left(lb_{i,r-1}, \hat{X}_{i,r} - c^{h/b}_{i,r}\right)$$

and

$$ub_{i,r} = min\left(lb_{i,r-1}, \hat{X}_{i,r} + c_{i,r}^{h/b}\right)$$

Beside the above empirical bounds, there exists for the game of Tetris estimated bounds on the mean score of the player [34] which we also used in the racing procedure. We have:

$$|X - \hat{X}|/\hat{X} \leq 2/\sqrt{n} \tag{7}$$

with probability 0.95, where $\hat{X}$ is the average score of the player on n games and X is the expected score. In the remainder we will refer to this bound as the Tetris bound.

5.3 Illustrative experiments

In the following, we present few experiments in different settings to compare the effect of the racing procedures described above and a version that does not use racing as well. In all experiments the initial search point x_0 was drawn randomly with $\|x_0\| = 1$ and the initial step-size $\sigma_0 = 0.5$. We let the algorithm run for 25 iterations (improvements for larger values were not noticeable), we set $r_{max} = 100$ and $\delta = 0.05$. As in the previous section there are eight feature functions in the evaluation function therefore the dimension of the search space is $n = 8$. CMA-ES parameters c_σ, d_σ, c_c, μ_{eff}, μ_{co} and c_{co} were set to their default values as presented in [13].

In order to reduce the learning time, we adopt the same scheme as presented above and evaluate the offspring on harder games, in which “S” and “Z” appear four times more frequently than in the standard game.

The algorithm maximizes the score of the game i.e. the fitness function is the average of lines scored by a search point on a number of games. When racing is applied, this number of games can go from 3 to r_{max}. When it is not applied, it is r_{max}. The initial bounds a and b used to compute R (eq. 6) were fixed experimentally to $a = 150$ and $b = 2000$.

Finally, as described in the previous section, offspring are normalized and at each iteration a series of r_{max} independent games (random seeds) are drawn randomly such that each individual of the λ offspring plays the same series of games.

5.4 Comments

The effect of racing is clear on the learning time of the algorithm. The number of evaluations is indeed reduced without a loss of performance. Figure 7 on the right shows the fitness function of the mean vector (the centroid of the distribution) for the different racing procedures and without racing. All algorithms reach the same performance at different evaluation costs. Hoeffding bounds perform the best, reaching the performance for the smallest evaluation time, followed by Tetris bounds and Bernstein. Figure 7 on the left shows the median rate of evaluations (r_{limit} being 100%) used in the races. With Tetris bounds this number of evaluations is reduced at the beginning of the learning process and increases towards the end reaching r_{limit}. For Hoeffding bounds the number of evaluations oscillates between r_{limit} and lower values. Finally, Bernstein bounds were not efficient compared to Hoeffding bounds, the number of evaluations is not reduced and at each race each individual is evaluated r_{limit} times. This is due to the fact that the standard deviation of the fitness is of the same order of its average. This is the case for Tetris, the standard deviation of the score distribution is almost equal to its mean. In other words $\hat{\sigma_{i,t}}$ is large compared to R in equation 6. When this is the case, Hoeffding bounds are tighter than Bernstein's, and this is the case in all our experiments.

Recall that r_{limit} is adapted using equations 3 and 4 and is initially set to 3, which explains why in the right of Figure 7, Bernstein races used less evaluations than in the case without racing.

We noticed also that there is a large variability in the racing procedure. Figure 8 shows the rate of evaluation out of r_{limit} of all 50 runs for Hoeffding and Tetris bounds.

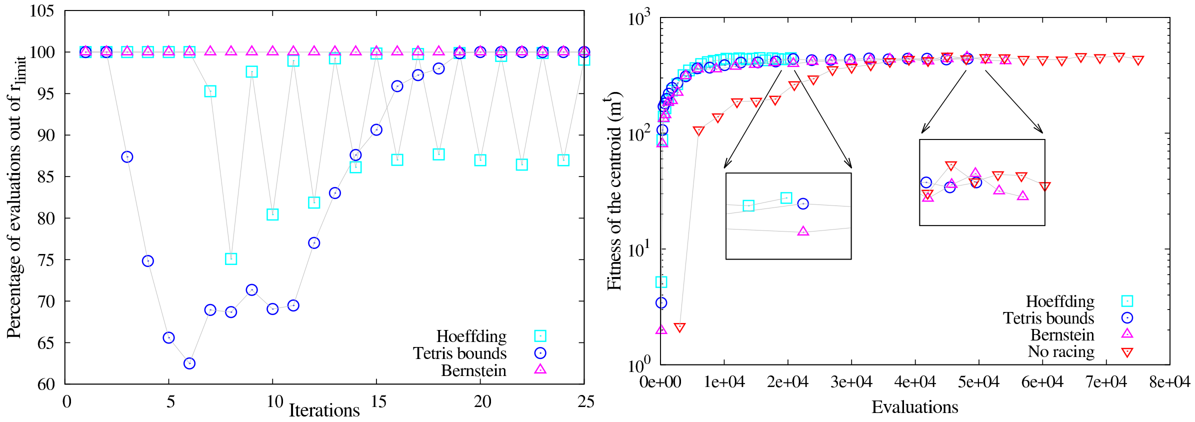

Figure 7: On the left, the number of evaluations with racing using different bounds versus iterations. On the right, log of the fitness of the mean vector m^t versus the number of evaluations. Median values over 50 runs.

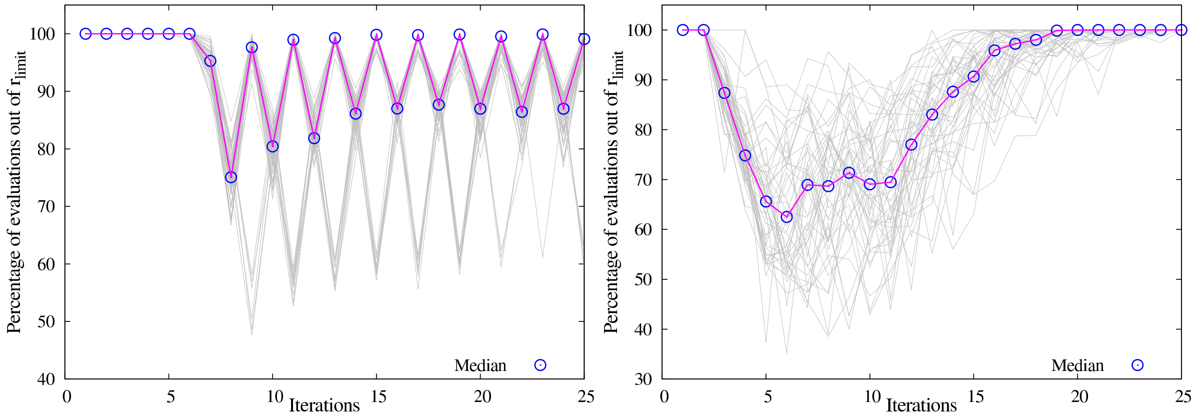

Figure 8: The number of evaluations for 50 runs thick line represent the median value. With Hoeffding races (left). Races with Tetris bounds(right).

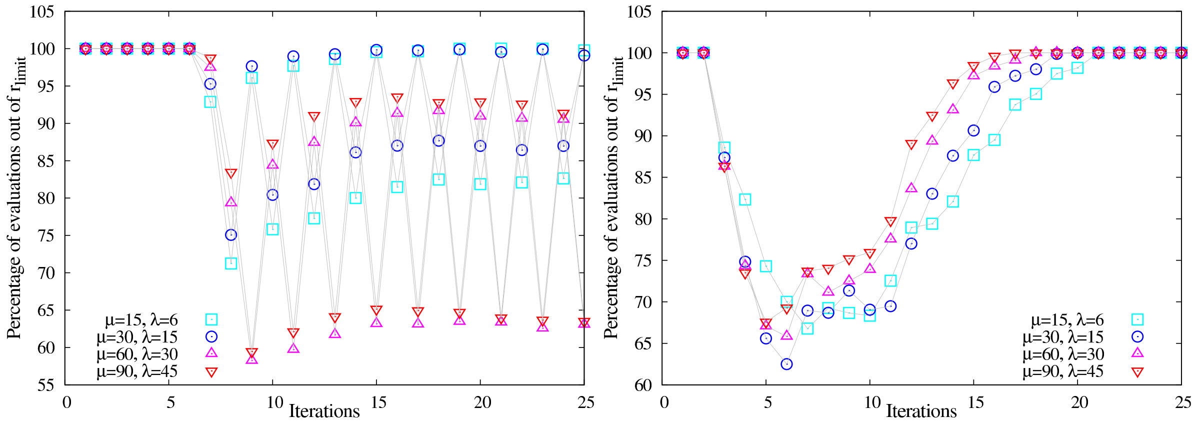

Figure 9: The median number of evaluations over 50 runs for different population sizes. Using Hoeffding races (left). Using Tetris bounds (right)

It is interesting to notice that the racing procedures reduce the number of evaluations early in the process and do not perform as well at the end. At this stage the racing procedure cannot distinguish between the offspring and selection is made only based on the fitness values (averaged over the r_{limit} reevaluation). The CI around the offspring's fitness overlap indicating that they perform equally well. This also suggests (see below) that the ranking of the offspring might not be correct. Incidentally, this corresponds to the time where the convergence of step-sizes start to "flatten" (Figures 10 and 11).

The effect of the population size on the racing procedure can be seen on Figure 9 where is shown the number of evaluations out of r_{limit} for different population sizes. Apparently increasing the population size reduces the number of evaluations within the race using Hoeffding bounds. On the other hand the opposite can be seen when using Tetris bounds.

Furthermore the population size does not effect the performance of the algorithms, the median value of the mean vector fitness reaches similar values for different population sizes. This is the case for both Tetris and Hoeffding bounds (figures 10 and 11).

Figure 12 shows the effect of the confidence value on the evaluation time. It presents four setting with different confidence $(1-\delta)$ levels. Using Hoeffding bounds, if we reduce the confidence value the algorithm stops earlier. This is the expected behavior: reducing the confidence reduces the bounds and races are decided faster. On the other hand, Bernstein bounds are not affected at all. The races are not able to distinguish statistical differences between the offspring even with low confidence values.

In order to assess the results of the racing procedure, we test the learned players on a standard game[7] and compare the scores with previously published scores. We follow the same scheme as in Sec. 4.3 to construct the player. We record the last mean vector m^t of each run of the algorithm and compute the mean of all these vectors (in our setting 50). This is merely a convention and others are possible[8]. The score of a player is the average score on 100 games. Taking into account variability of the score distribution, a confidence bound of ±20% is to be taken into account (see equation 7).

6 Further thoughts and considerations

One other aspect we are interested in is the nature of the learned weight vectors and consequently the strategies. Surprisingly, we notice that even thought the search space (fitness function) is noisy, which suggests a hard optimization problem, the learned vectors throughout different independent experiments seem close to each other. In other words the algorithm converges toward the "same" region of the search space. This is more noticeable in the case of features $f_1 \dots f_6$ than for the case of features $f_1 \dots f_8$.

[7] A game of size 10×20 with all seven pieces equally probable.

[8] One could also choose the best performing vector out of the whole set. However testing them all on a reasonable number of games might be long. For example testing one player on 10 games took 15 hours on a 2 Ghz CPU.

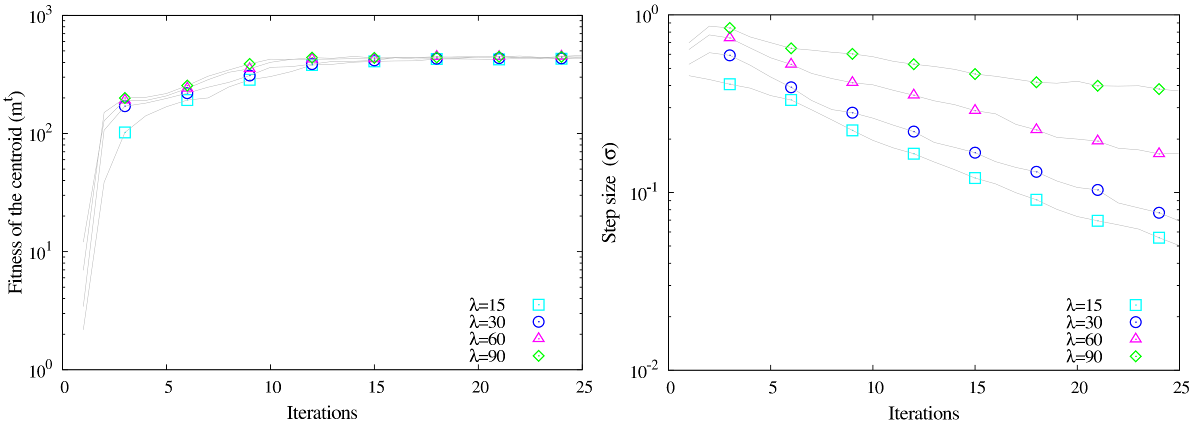

Figure 10: CMA-ES and Tetris bounds with multiple population sizes. Log of the fitness of the mean vector m^t (left) and the step-size σ (right). Median values over 50 runs.

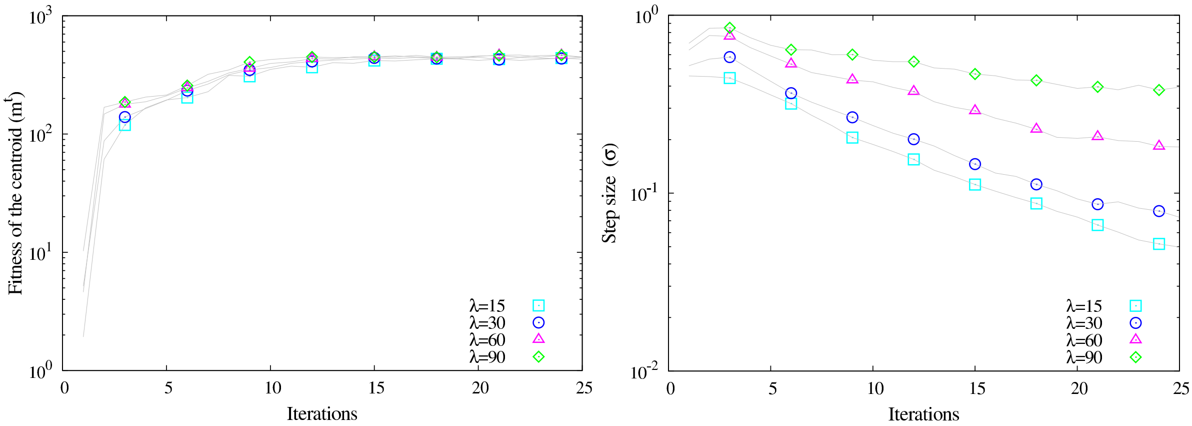

Figure 11: CMA-ES and Hoeffding races with multiple population sizes. Log of the fitness of the mean vector m^t (left) and the step-size σ (right). Median values over 50 runs.

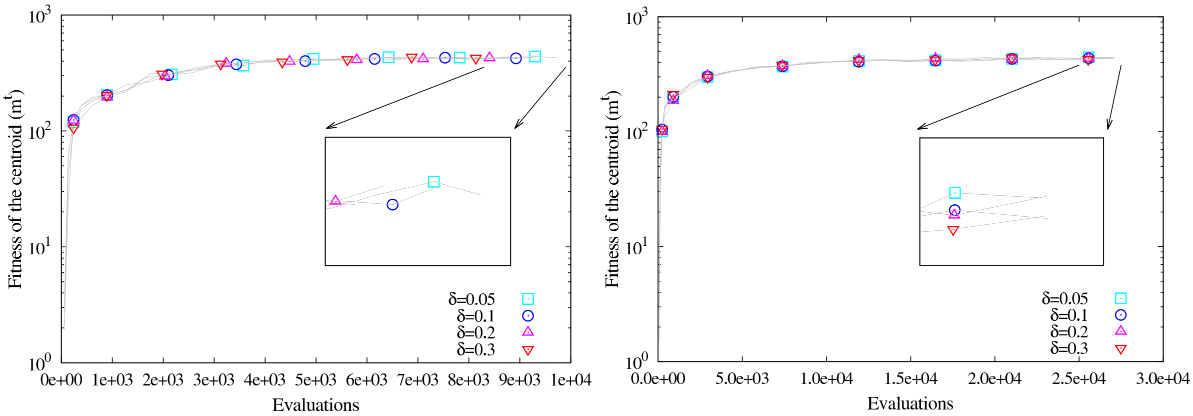

Figure 12: Log of the fitness of the population centroid for different confidence values. Hoeffding races (left) and Bernstein races (right). Median values over 50 runs.

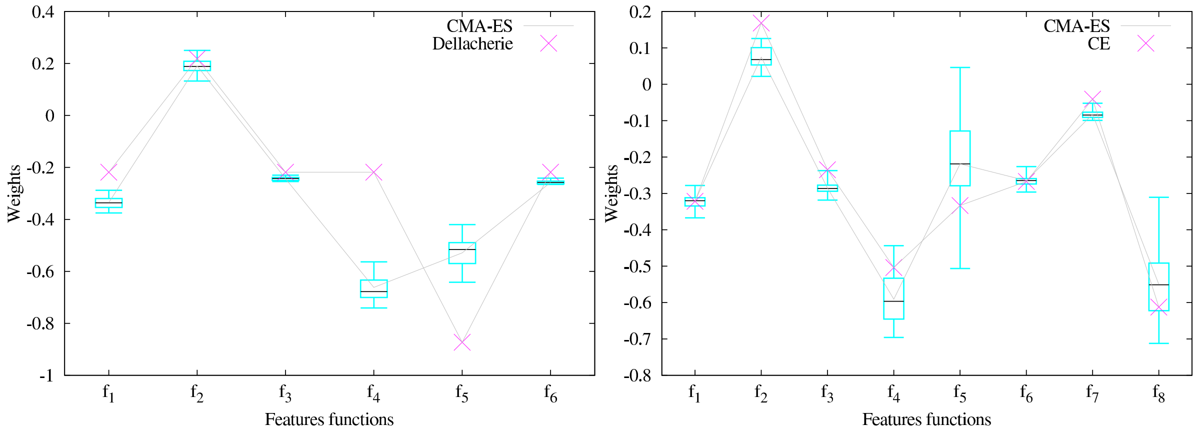

Figure 13: The learned weights of 20 independent runs of CMA-ES. On the left, features functions $f_{1\ldots6}$ and the hand fixed weights by Dellacherie [11]. On the right, features functions $f_{1\ldots8}$ and the weights given by [33] learned using NCE

Figure 13 shows a series of learned vectors produced by 20 independent runs using box and whiskers plot. In the case of the feature set $f_1 \ldots f_8$ the quartiles are larger, which suggests that the corresponding weights did not converge. This said, the average vector (the solid line) for which the score is given in Table 1 performs well.

Furthermore, even though both feature sets share features $f_1 \ldots f_6$, the learned weights for these features do not converge to the same values in both experiments. This behavior suggests that 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 Dellacherie [11] and Thiery [33], we notice that on some features our vectors have similar weights. Although we cannot compare directly these vectors since they were produced using different experimental protocols[9]. However, on the one hand this observation further supports our previous argument stating that the search space may not be extremely multimodal[10]. On the other hand, it was interesting to realize that some of the weights our vectors learned are close to the weights that an expert has fixed by hand. Though our player performs better than his when tested (table 1), the major differences in the weights resides in features f_4 and f_5 that Dellacherie weights respectively higher and lower than CMA-ES. The same conclusion can be drawn from Figure 13 between the weights learned using CMA-ES and NCE, on which they seem very close.

Finally, if we consider values of the weights obtained we notice that the algorithm produces a “sensible” playing strategy. It minimizes the landing height (f_1), the number of transitions from filled to empty vertically and horizontally (f_3, f_4), the number of holes (f_5) and the number of rows with holes (f_8). It also minimizes the depth of the wells (f_6) and the depth of the holes (f_7). The only feature that is maximized is the number of eroded pieces (f_2) which is directly related to the number of lines scored. Furthermore, features f_4 and f_8 were considered by the algorithm as more important than the others without considering f_2.

7 Conclusions

The game of Tetris is probably among the few games that are popular worldwide and designing artificial players is a very challenging question. Our aim in this paper is to show that evolutionary computation can be applied to design good strategies for the game of Tetris that can compete with the best reported players. The player strategy uses a feature-based game evaluation function to evaluate the best moves for

[9] Recall that our vectors are averaged over several runs (section 4.3) which is not the case for [33] and also not the case for [11] since they were fixed by hand

[10] This argument may only be valid for these sets of features.

a given state and which is optimized using the CMA evolution strategy.

We have shown that by analyzing the shape of the search space, we were able to adapt the algorithm to converge better in this particular case. We also discuss how to reduce the length of the games which serves greatly when running lengthy learning algorithms. The racing procedures presented allow to learn very good player at a fraction of the learning cost.

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 could be achieved using better feature functions, ones that synthesize the game better. An interesting research question would be to devise algorithms to learn new feature functions rather than only learn the weights.

References

[1] J-Y Audibert, R. Munos, and C. Szepesvári. Tuning bandit algorithms in stochastic environments. In M. Hutter et. al., editor, *Algorithmic Learning Theory*, volume 4754 of *LNCS*, pages 150–165. Springer, 2007.

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

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

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

[5] N. Böhm, G. Kókai, and S. Mandl. An Evolutionary Approach to Tetris. In University of Vienna Faculty of Business; Economics and Statistics, editors, *Proc. of the 6th Metaheuristics International Conference*, page CDROM, 2005.

[6] Ron Breukelaar, Hendrik Jan Hoogeboom, and Walter A Kosters. Tetris is Hard, Made Easy. Technical report, Leiden Institute of Advanced Computer Science, Universiteit Leiden, 2003.

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

[8] Donald Carr. *Adapting Reinforcement Learning to Tetris.* PhD thesis, Rhodes University, 2005.

[9] 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.

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

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

[12] V. Farias and B. van Roy. *Probabilistic and Randomized Methods for Design Under Uncertainty*, chapter Tetris: A study of randomized constraint sampling. Springer-Verlag, 2006.

[13] N. Hansen, S.P.N. Niederberger, L. Guzzella, and P. Koumoutsakos. A method for handling uncertainty in evolutionary optimization with an application to feedback control of combustion. *IEEE Trans. Evol. Comp.*, 13(1):180–197, 2009.

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

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

[16] V. Heidrich-Meisner and C. Igel. Hoeffding and bernstein races for selecting policies in evolutionary direct policy search. In *Proc. of the 26th ICML*, pages 401–408, New York, 2009. ACM.

[17] Yaochu Jin and J. Branke. Evolutionary optimization in uncertain environments-a survey. *Evolutionary Computation, IEEE Transactions on*, 9(3):303 – 317, june 2005.

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

[19] J. Kennedy, R. C. Eberhart, and Y. Shi. *Swarm Intelligence.* Morgan Kaufmann, 2001.

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

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

[22] 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*, pages 1 –8, july 2010.

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

[24] O. Maron and A. W. Moore. Hoeffding races: Accelerating model selection search for classification and function approximation. In *In Proc. Advances in neural information processing systems*, pages 59–66. Morgan Kaufmann, 1994.

[25] Oden Maron and Andrew W. Moore. The racing algorithm: Model selection for lazy learners. *Artificial Intelligence Review*, 11:193–225, 1997.

[26] A. Ostermeier, A. Gawelczyk, and N. Hansen. A derandomized approach to self-adaptation of evolution strategies. *Evolutionary Computation*, 2(4):369–38, 1994.

[27] 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.

[28] C. Schmidt, J. Branke, and S. Chick. Integrating techniques from statistical ranking into evolutionary algorithms. In Franz et. al. Rothlauf, editor, *Applications of Evolutionary Computing*, volume 3907 of *LNCS*, pages 752–763. Springer, 2006.

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

[30] P. Stagge. Averaging efficiently in the presence of noise. In A. Eiben et. al., editor, *Proc. of PPSN 5*, volume 1498 of *LNCS*, pages 188–197. Springer, 1998.

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

[32] C. Thiery and B. Scherrer. Personal communication.

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

[34] C. Thiery and B. Scherrer. Building Controllers for Tetris. *International Computer Games Association Journal*, 32:3–11, 2010.

[35] C. Thiery and B. Scherrer. Least-Squares λ Policy Iteration: Bias-Variance Trade-off in Control Problems. In *Proc. ICML*, Haifa, 2010.

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