BSc Project

Learning to Play Tetris using the Covariance Matrix Adaptation Evolution Strategy

February 11, 2016

Contents

Abstract

In this thesis, we conduct an unbiased comparison between the Cross Entropy Method and the Covariance Matrix Adaption Evolution Strategy (CMA-ES) in training an agent to play the classical game, Tetris. Many researchers have previously explored methods for constructing artificial Tetris agents, but mostly focusing on how to maximize the score of the agents. A common approach is to construct a configurable controller and apply some optimization algorithm from the field of machine learning to set the parameters for the best possible controller. Using the commonly used MDPTetris (MDPTetris, 2009) platform, the Cross Entropy method is benchmarked against the CMA-ES algorithm under the same conditions to reach an empirical understanding of how the two algorithms compare when applied to Tetris.

No conclusion yet.

Abstract

In this thesis, we conduct an unbiased comparison between the Cross Entropy Method and the Covariance Matrix Adaption Evolution Strategy (CMA-ES) in training an agent to play the classical game, Tetris. Many researchers have previously explored methods for constructing artificial Tetris agents, but mostly focusing on how to maximize the score of the agents. A common approach is to construct a configurable controller and apply some optimization algorithm from the field of machine learning to set the parameters for the best possible controller. Using the commonly used MDPTetris (MDPTetris, 2009) platform, the Cross Entropy method is benchmarked against the CMA-ES algorithm under the same conditions to reach an empirical understanding of how the two algorithms compare when applied to Tetris.

No conclusion yet.

1 Introduction

This thesis will cover an experimental approach to compare two optimization algorithms both considered state-of-the-art. The algorithms will be applied to learning the game Tetris. Most often, when machine learning techniques are applied to problems such as games, conventional learning methods fall short due to both very large number of possible actions in the games and to highly unpredictable mappings between actions and their long-term consequences. The games are considered and episodic task where some agent is placed in the game environment and attempt to play the game as well as possible. The agent will make decisions on how to react to the game while playing, and optimizing the agent is hence find as good as possible policies for the agent. Yet, as mentioned, deciding what makes a policy good can be very difficult in problems like games. Hence, the development of the policies is based on reinforcement learning where the only feedback on the policy is the score gained at the end of a game.

1.1 Reinforcement learning

In the field of Machine Learning, the sub field of reinforcement learning deals with training agents to behave in an environment through trial and error. The environment typically consists of a set of states between which an agent can transition through a set of actions. The reinforcement learning methods relates somewhat to supervised and unsupervised learning models. In unsupervised learning, the agents are never given a feedback on their actions and

must attempt to derive some a pattern without decisive responses. In supervised leaning, the agent is trained from labelled data. In this way the agent will always, for any action, receive feedback of whether the committed action was desirable or not. Reinforcement learning appears somewhat similar to supervised learning. In reinforcement learning, the agent will commit actions in an environment and receive rewards based on those. The main difference between supervised learning and reinforcement learning is that in reinforcement learning, the agents cannot necessarily tie a correctness to a specific action as some action, albeit yielding reward, may cause relative loss of reward in the future. An example of how this applies to artificial agents is the game Tic Tac Toe. If supervised learning is applied, the playing agent will for each move in the game know if the move was considered good or bad. In reinforcement learning however, the only feedback the agent will have is rewards that a given to the agent when reaching certain states of the game. Often in the case of games, the time frame naturally falls into episodes in which the goal of the agent is to accumulate as much reward as possible before the episode ends. When the agent interacts with the environment, it transitions between different states through actions. To determine what action to take, the agent will use a function that the yields a reward given to the agent if the action is taken. The goal is then to choose a policy, that when followed by the agent, gives the highest possible cumulative reward over time. It's important to notice that the reward for an action may be negative, and will in such case actively discourage the agent for taking given actions. The environment, in which the agent appears, often or at least for simplicity, has an exit state in which the agent is terminated, and the cumulative reward is revealed. The action transitions may not be deterministic and if they are not, it's required that the probabilities of transitions remain during the entire episode (**?**). If the state space is large enough, it becomes infeasible to attempt to compute the commutative reward when taking actions, as this would require a thorough search of a large space. Thus, in situations with large state spaces, such as Tetris, one must choose an approach for finding the best possible policies by other means than exhaustive search. The methods used in reinforcement learning does specify how the agent should change its policy according to results from earlier episodes, and the specific methods used in the context of Tetris is explained in further detail in later sections.

1.2 Learning Tetris

On the topic of reinforcement learning, a widely used benchmark for learning algorithms are designing agents for playing the classical game of Tetris. Tetris is an appealing benchmarking problem due to it's complexity. The standard games plays on a board made from a grid that is 10 cells wide and 20 cells tall. As the game progress, differently shaped pieces fall from the top of the board. When a row on the board is fully occupied by pieces, the line is removed, all lines above it moved one line down and a score point is given to the player. If a cell above the 20 rows first is occupied, the game ends. The task of the player is to move and rotate the falling pieces in a way that yields the highest score before the game ends.

Tetris is indeed a hard task to computationally optimize, as the game has a very high number of board configurations estimated to be 10^{59} (Scherrer, 2009b). In relation to reinforcement learning, an agent that plays Tetris is placed in an environment with a set of states far too large to exhaustively explore, and a set of transitions that are stochastic. Because of this

complexity, a common approach in the literature is to use *one-piece controllers*[1], such as described in Scherrer (2009a). These controllers agents that are aware of only the current board state and the currently falling piece. Hence, the policy of the agent is only to greedily choose the action that transitions to the most rewarding state from a single piece, which is typically less than 80 states[2]. Using these controllers, the search space is reduced to only looking at the current board, and the possible places to drop the piece. The game used for the benchmark is a simplified version of Tetris, in which controllers need only to decide in what column to drop the current piece, and what orientation the piece should have when dropped. Thus, the simplified version of Tetris differs from the original game mainly in two significant aspects. First, the controller is disallowed to move the piece horizontally while the piece is falling. Second, the controller has 'infinite' time to make its decision on where to drop the current piece. This way, the game only progress in discrete time steps when the controller takes an action, whereas the classical game runs continuously regardless of the players actions. Thus, the controller cannot take advantage of moving the piece during the fall, but is not restricted by the time limitations. This is however a common way of benchmarking Tetris playing agents (Scherrer, 2009b)

When the controllers decide which action to take, it will simulate each of the possible actions and choose the one that leads to the most favourable board state. To evaluate the board state, the controller uses a set of features that defines various qualities of the board, and associate a weight to each feature. This means that the efficiency of the controller is determined by the features the controller is aware of and how heavily they are weighted. This allows the controller with n features to be expressed as an n dimensional real-valued vector, with one dimension per feature, and the value in that dimension the weight. An often referred to controller is the Dellacherie's controller, as described in Scherrer (2009b). This controller takes six features of the board into account, seen in table 6 on page 17. In relation to reinforcement learning, the policy of the agent is the feature set combined with the weights of each feature. When using the *one-piece controllers*, the policy remains the same during the episode, and the learning is based on updating the policy from concluded games, rather than picking up on signals that occur in the game as it plays.

1.3 Goals of the thesis

Both the Cross-Entropy and the CMA-ES methods has been used in learning Tetris with *one piece controllers*, but as mentioned, to our knowledge, only little effort has been put into comparing the two methods. This thesis will explore how the two methods compare against each other under similar conditions. Therefore, we will use a set of features among those commonly used, and compare how the two optimizing algorithms differ. The goal however is not to find a controller that outperforms existing controllers, but only to investigate how the Cross-Entropy and CMA-ES differs when learning Tetris with similar controllers.

In this paper, we will explore how the two state-of-the-art optimization algorithms, Cross-Entropy and CMA-ES, differ when applied to the task of playing Tetris.

The SHARK library (Igel et al., 2008) contains a working implementation of the CMA-ES

[1] Agents and controllers both refer to artificial players.
[2] Assuming 4 possible rotations and 20 columns in which the piece may be dropped.

algorithm. However, the Cross-Entropy method is not present in the SHARK library and thus, a part of the thesis is to implement it ourselves according to the other researchers work. To document the soundness of the implemented CE method, we will replicate the experiment in (Thiery and Scherrer, 2009) and verify that we obtain the similar results.
Then, we will benchmark CMA-ES and CE against each other to determine if one yields better optimization results than the other.

1.4 Scope and limitations

The experiments will be carried out on the simplified version of Tetris using the MDPTetris software found at (MDPTetris, 2009), which is the same simulator used by Scherrer et al. in (Scherrer, 2009a) among other authors in various papers. This software already have the well known feature sets implemented, so we will not ourselves extend any of the features. The source code of the Tetris simulator is used as-is, and is therefore not altered prior to running the experiments. For comparing the optimizers, the SHARK (Igel et al., 2008) library will be used. This library already contains an implementation of the CMA-ES optimizer, but lacks the Cross-Entropy. Therefore, a part of this thesis will be to implement and document the Cross-Entropy method in Shark.

2 Background

2.1 Markov Decision Processes

A Markov Decision Process (MDP) is a model for an automatic decision making system. In the MDPTetris platform the agent that plays the simulated games makes decisions modeled after an MDP (MDPTetris, 2009). An MDP consists of a set of states $\boldsymbol{\mathcal{S}}$, a set of actions $\boldsymbol{\mathcal{A}}$ and a *state transition probability kernel* $\boldsymbol{\mathcal{P}}$. The states in $\boldsymbol{\mathcal{S}}$ denotes all the states of the game including the agent itself. In Tetris, the state is simply the state of board in regards to which cells are occupied by pieces and which are not. The state also contains the current piece to be dropped by the controller. The actions in $\boldsymbol{\mathcal{A}}$ denotes all the actions that an agent may take. The $\boldsymbol{\mathcal{P}}$ is a probability function that determines how likely it is for an action to transition to another given state. As an example, given 3 states s_0, s_1, s_2, an action a_0 and the agent currently being in state s_0. Then $\boldsymbol{\mathcal{P}}$ might determine that taking action a_0 transitions to s s_1 with probability 0.2 and to state s_2 with probability 0.8.
The behaviour of the agent is defined by its policy π which is a function that maps states of the game into actions, formally defined as $\pi : \boldsymbol{\mathcal{S}} \to \boldsymbol{\mathcal{A}}$. The goal is to find the best possible policy which in the case of Tetris, is the policy that leads to the highest average score in the game. The value of the policies are described as a *value function* $V^\pi(s)$ which maps a state of the game into an expected value, namely the score of the agent when starting in the given state and follows the policy π, formally defined as $V^\pi : \boldsymbol{\mathcal{S}} \to \mathbb{R}$. The optimal value function, that plays the game as well as possible is denoted V^*. Optimizing the agent is a task that consists of determining a policy π for which V^π gets as close to V^* as possible. Finding the best policy for Tetris is a very computationally heavy task due to the enormous state space. Also, in the MDPTetris platform, the agent will not be aware of the entire game state exactly to its size, but is rather use policies that are concerned with simplified features of the game (described in detail in later sections). The methods in reinforcement learning provides various ways to search for good policies. One approach is Dynamic Programming (DP). The idea behind DP

is to, for each state, estimate the reward gained by the agent in for each possible action. Yet, the DP approach is in the case of Tetris infeasible as it is a far too overwhelming task to comprehensively compute the full traversal of the games state space. To give an intuition of this, consider an MDP illustrated in figure figure 1. The MDP models a hypothetical game with a set of sates $\boldsymbol{\mathcal{S}} = \{s_0, s_1, s_2, s_3, s_4\}$ where the agent is given a reward of r_n for spending a time step in state s_n. The agent has a set of actions $\boldsymbol{\mathcal{A}} = \{a_0, a_1, a_2, a_3, a_4, a_5, a_6\}$ which causes the agent to transition between states in the game. In the figure, each circle/node illustrates a state, and the arrows shows the possible transitions between states. Next to the arrow the action that causes the transition is listed along with the probability that if the action is committed, the particular transition takes place. For example, if the agent is in state s_0 and takes action a_0, it will with probability 0.8 end up in state a_0 again and receive a reward of r_0. In this case, it is viable to compute the optimal policy by traversing all states and compute the expected reward based on chosen actions. Thus, for a policy π DP allows one to obtain exactly the expected reward given to the agent when following π.

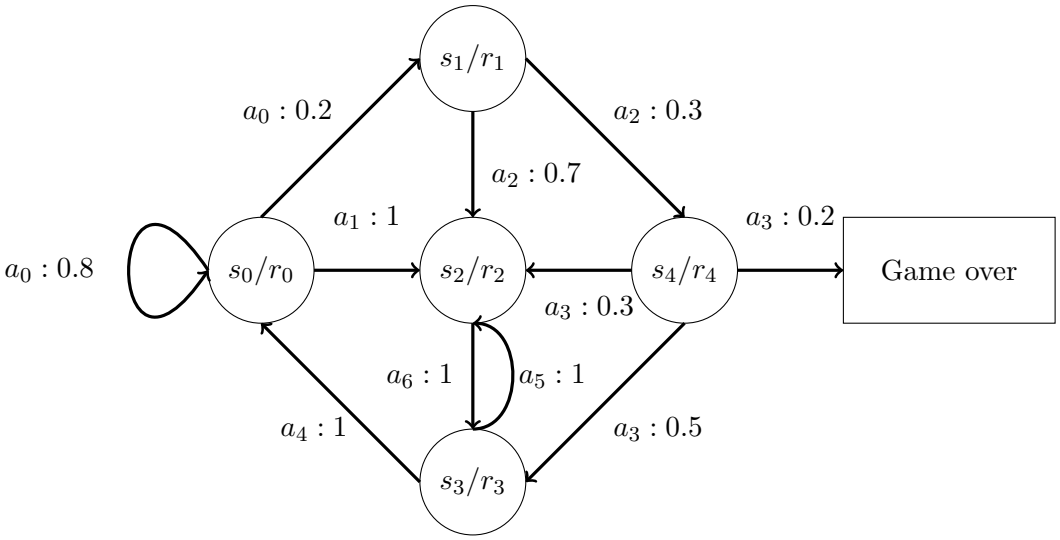


Figure 1: Example of a game formulated as an MDP

Tetris can be formulated very well as an MDP as well. When looking at Tetris as an MDP, the state consists of the current board configuration and the currently falling piece. Figure 2 shows a visual presentation of how the game state is formulated into an MDP state. The black blocks in the bottom of the board notes that these blocks are occupied and the text 'I-block' indicates that the currently falling piece is an I-block. The actions of the decision maker are hence partially deterministic. The agent/controller can fully decide where the piece falls. However, part of the state is the next pieces to be dropped, which is randomly chosen from some distribution. The reward given to the agent is based on how many lines it clears. If the agent appears in a state where it clears lines, the reward is exactly how many lines are cleared. The reward is 0 in all other cases except for a state that ends the game. The game over state typically leads to a massively negative reward to encourage the agent to play for as long as possible. To get an intuition on why the exhaustive search through the Tetris state space is not an option, consider how Tetris is formulated as an MDP. The state of the game is the configuration of the board combined with the current pieces to be dropped. Figure 2 shows how the current board of the game can be considered as an MDP state. In this

case, I-piece can be oriented either vertical or horizontal. In horizontal orientation, it can be dropped in 7 different columns, and in vertical orientation it can be dropped in all 10 different columns. When the piece is dropped, the next state of the game is the one where the I-piece occupies some space on the board, and the new falling piece is one of the possible 7 pieces (see figure 16) from the game which is chosen randomly out of the agents control. The state shown in figure 2 thus have $17 \cdot 7 = 119$ transitions to other distinct states. As all states have approximately the same number of transitions and that the number of distinct states is enormous, the intuition clearly is that this cannot be exhaustive be computed with the Dynamic Programming approach.

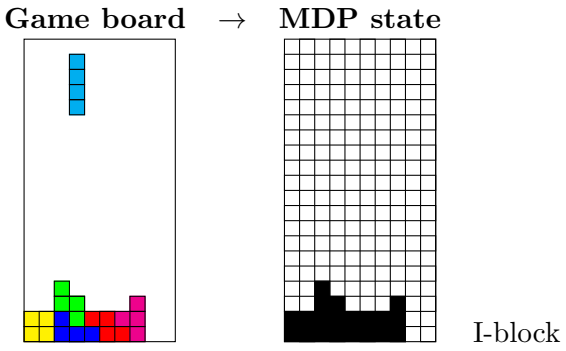


Figure 2: Illustration of how Tetris can be formulated as an MDP.

However, another and more viable solution is in this case the *Monte Carlo* methods (**?**). With the *Monte Carlo* method, the goal is to learn the value of V^π, in our case, we typically wish to estimate the value function from the starting state of the game. Using the method, the episode is simulated and the agent will interact with the environment with the given policy and finally exit with an estimate of V^π, in our case, the final score. Thus, using this method, we need not search through the entire state set, and neither do we need to know all details of the game states the agent went through while playing. Hence, the *Monte Carlo* approach fits the case of Tetris very well as we cannot afford to comprehensively model the entire state space, but only need assess the performance of a policy from the starting state of the game. How the feedback from the simulation is used to obtain better policies is discussed in detail in the section 2.3 on page 11.

2.2 Optimization problem

In section 2.1, it's described that the problem of find a policy that allows an agent to play Tetris well cannot be approached with pure analytical methods. When applying the moreto assess the value function of a policy, the only knowledge one gains is the actual value from the estimation of the value function. Algorithms such as the Cross Entropy Method and CMA are both designed to solve exactly these types of problems. The rest of this section is devoted to present the properties of the general problem to be solved by the two algorithms, and how the problem of optimizing Tetris fits within this model.

2.2.1 General problem

The general problem that the Cross Entropy Method and CMA tries to solve is often referred to as a *black box* scenario (Hansen, 2015). The algorithms works to attempt to optimize a function O that takes a real-values vector and maps it to a real valued scalar.

$$O : \mathbb{R}^n \to \mathbb{R}$$

This function is often either called the *objective function* or the *fitness function*. The name *Objective function* refers to the fact that it's the objective of the algorithm to optimize the function, and *fitness function* is derived from the idea that the function determines how *fit* the input vector is. The term black *black box* originates from the fact that the one does know only few, if any, analytical properties of the function. The gradient is either unknown or not useful, preventing a proper gradient decent approach. In summary, all details of the functions are unknown to the optimizer, which may only query the value of single vectors.

The optimization algorithms may, due to the lacking information about the problem only act on a trial-and-error basis. As the goal of this thesis is to compare how the Cross entropy Method and CMA compares, it's important to determine the metric for cost in terms of how many resources the algorithms consume versus performance. Typically, and also in our case, the cost is measured in terms of function evaluations. The algorithms performance is considered in terms of search cost and solution value. Hence, the algorithm competes both on the quality of their solutions and how few times they need to query the *objective function*.

2.2.2 Optimizing Tetris

To understand how the optimization algorithms will handle the optimization of the policies, it's important to know how the policy is defined. In the MDPTetris platform, the policy is a *one piece controller*. This means that the controller is aware of the current board and the current piece, and it will place the current piece only based on this information. As mentioned in section 2.1, the controller is only aware of a reduced feature set $\Phi = \{\phi_0, \ldots, \phi_n\}$. These features map a game state s to a real value: $\phi : \boldsymbol{\mathcal{S}} \to \mathbb{R}$. The value of the feature represents how significant the feature appears in the current state. As an example, consider the a simplified feature set $\{\phi_0, \phi_1, \phi_2\}$ were the features are defined as in the following table.

Feature	Description
ϕ_0	The depth of the deepest well on the board, where a well is defined as a straight vertical line of unoccupied cells with occupied cell on both sides.
ϕ_1	Number of holes in the board. Precisely, empty cells with all adjacent cells occupied.
ϕ_2	Height of the highest occupied cell.

Figure 3 shows a Tetris board in a very reduced size to illustrate how these feature functions perceive the board. The well is shown as a dashed line, and the holes in the board are marked with $\times$.

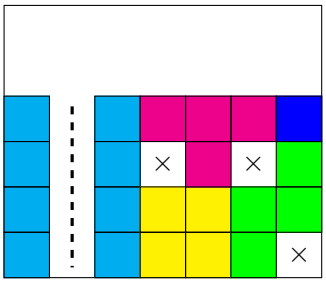


Figure 3: Reduced example board

When the controller is about to make a choice, it will consider all possible locations to drop the piece. Each of the locations corresponds to an action in the MDP. To decide which action to take, a value function V is used to assert the value of the state. Note that this value function nit **not** the same as the value function described in section 2.1. This value function maps a feature set and a set of associated weights to a real value. The value from this function is used to *rank* each possible action. this function is formally defined as

$$V(s) = \sum_{i=1}^{n} w_i \phi_i(s)$$

With this function, the i'th weight determines how much the i'th feature contribute to the decision making. In the case with figure 3, the features yield the following values

$$\begin{aligned}\phi_0(s) &= 4\\ \phi_1(s) &= 3\\ \phi_2(s) &= 4\end{aligned}$$

If all weight were set to -1, then the controllers perception of the state would be

$$\begin{aligned}V(s) &= w_0\phi_0(s) + w_1\phi_1(s) + w_2\phi_2(s)\\ &= -4 - 3 - 4\\ &= -13\end{aligned}$$

When the controller simulates all possible actions, it will use the very same function on the state that each action transition to and rank them in non-descending order. Thus, if the controller have the actions $\{a_0, \ldots, a_n\}$ and action a_i transitions to state s_i, the controller will order all the actions such that

$$V(s_0) \geq \cdots \geq V(s_n)$$

And then choose action a_0. The policy of the agent is then defined by the combination of feature set and the set of weights.

When applying an optimization algorithm to Tetris, the feature set is fixed, and the task of the optimizer is to tune the weights to make the policy as efficient as possible. This is done through the previously mentioned *Monte Carlo* method, where the agent is simulated with a given policy from the starting state, and the final score is used to determine the performance of the policy. The next sections walks through how the optimizers each approach this task.

2.3 Optimizers

In this thesis, the Cross-Entropy method and the Covariance Matrix Evolution Strategy are compared. Both of the methods fall into the category of *stochastic optimization* methods. These methods are useful for optimization problems that have no gradient. The optimization functions aim to optimize the parameter set $\mathbf{x}$ for the objective function O.

$$\hat{\mathbf{x}} = arg\ \max_{\mathbf{x}}\ O(\mathbf{x})\ O : \mathbb{R}^n \to \mathbb{R}$$

In these methods, the optimizing algorithm uses a family of parametric distributions, and maintain a mean m along with other parameters to search the best possible solution for the objective function. In the case studied in this thesis both the CMA-ES and Cross-entropy methods use a Gaussian distribution to sample solutions to the objective function. Hence, both of the functions aim to find a mean m and an $n \times n$ matrix M[3], such that when a vector $\mathbf{x}$ is sampled by $\mathbf{x} \sim \mathcal{N}(m, M)$, then $O(\mathbf{x})$ is likely to yield preferable results.

The algorithms work iteratively, such that the mean and variance of the distribution is altered for each iteration t. The algorithms start by initializing the parameters either at random or some fixed point. A common configuration is setting the mean to all zeros and the standard deviation to the identity matrix. Thus, for the first iteration $t = 0$, a configuration could be:

$$m_0 = \begin{bmatrix} 0 \\ \vdots \\ 0 \end{bmatrix}, \quad M_0 = \begin{bmatrix} 1 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & 1 \end{bmatrix}$$

Where the subscript notes that the values occur in iteration 0.

In each iteration, the algorithms sample λ vectors and evaluate their fitness against the objective function. When each of the solutions are evaluated, they are ordered according to their fitness:

$$\{\mathbf{x}_1, \dots, \mathbf{x}_N\} \text{ Such that } f(\mathbf{x}_1) \geq f(\mathbf{x}_2), \dots, f(\mathbf{x}_{N-1}) \geq f(\mathbf{x}_N)$$

The mean and standard deviation for the next iteration, that is m_{t+1} and M_{t+1} is then updated usually by considering the best of the ordered solutions. How exactly these parameters are updated is individual for each method and can be seen in the following sections.

2.4 CE (Cross Entropy)

CE is described through many papers in slightly different ways. This section describes the algorithm in a similar fashion to (Thiery and Scherrer, 2009).

This method uses a Gaussian distribution to sample sample vectors for the objective function O. The algorithm aims to adjust the parameters of the distribution such that samples x drawn from the distribution gives the best possible results whan aplied to the objective function $O(x)$.

[3] In (Hansen, 2015), σ is used for step-size in CMA-ES, so M is instead introduced as an arbitrary $n \times n$ matrix in its place.

The Cross-Entropy method starts with an initial mean m which is an n-dimensional vector typically set to 0:

$$m_0 = \begin{pmatrix} 0 \\ \vdots \\ 0 \end{pmatrix}$$

The second argument to the gaussian distribution is a diaginal matrix which contains the variance for each component.

$$M_0^2 = \begin{bmatrix} \sigma_1^2 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & \sigma_n^2 \end{bmatrix}$$

The algorithm will in each generation sample a set of search points and rank these using the objective function. In each generation, N vectors are sampled by $x_i \sim \mathcal{N}\left(m, M^2\right),\ i \in \{1, \dots, N\}$. The vectors are all evaluated against the fitness function and ordered such that $O\left(x_1\right) \geq, \dots, \geq O\left(x_N\right)$, and the μ best are chosen for updating the distribution parameters. The mean is updated as the centroid of the chosen vectors, and the variance in each dimension is set to the variance of the chosen vectors in each dimension.

The pseudo code and details of the algorithm can be seen in figure 4 on page 12.

Noisy cross-entropy method

input
O : The function that estimates the performance of a vector x
(m_0, M_0^2): The mean and variance of the initial distribution
N : The number of vectors sampled per generation/iteration
μ: The number of offspring selected for the new mean
Z_t: The noise added to each generation/iteration

loop
Generate N vectors $x_1, x_2, \dots, x_N$ from $\mathcal{N}(m_t, M_t^2)$
Evaluate each vector using O
Select the μ vectors with the highest evaluation
Update m_{t+1} of the μ best vectors
Update M_{t+1}^2 of the μ best vectors + Z_t
end loop

Figure 4: The pseudo code for the Cross-Entropy algorithm

2.4.1 Input

The objective function
The objective function O is a function used to rank the value of a sampled vector. As described

in the 'Optimizers' section, CE is a general stochastic iterative algorithm that tries to solve an optimization problem of the form (Thiery and Scherrer, 2009):

$$\hat{\mathbf{x}} = arg\ \underset{\mathbf{x}}{max}\ O(\mathbf{x})$$

Where x corresponds to a given vector, and O is our actual objective function.

The mean and variance of the gaussian distribution
Here m_t is the mean and M_t^2 is the variance of the gaussian distribution $(m_t,\ M_t^2)$. More specifically this gaussian distribution is defined as

$$\mathcal{N}(m_t, M_t^2)$$

Where t denotes the current iteration.

The number of vectors
N is the number of vectors sampled in each generation.

The number of selected vectors / parents
μ is the number of vectors which are selected to compute the new mean m_{t+1}, and variance M_{t+1}^2, for next generation/iteration. These offspring vectors gets selected directly by taking the μ best ranked vectors.

The noise term
The noise factor, Z_t, is the amount of noise which is applied to the variance M^2 in iteration/generation t. In general, noise is used to prevent the algorithm from getting narrowing down to a local optimum, but rather explore new solutions. The noise term can be any function of t. Among the most described noise types are: no noise, constant noise and linear decreasing noise (Szita and Lörincz, 2006). When using no noise, Z_t is simply set to zero. When using constant noise, the same value is added to the variance M^2 in each iteration/generation. When using linear decreasing noise, Z_t is often set as $Z_t = max(5-t/10, 0)$.

2.4.2 Loop

Sampling the population
The first step of the loop is to generate the new generation consisting of N vectors. These vectors are drawn from the distribution $x_i \sim \mathcal{N}(m_t, M_t^2)$.

Evaluating the population
After sampling the population, the algorithm needs to order the vectors to find the μ best vectors, each vector x_i , $i \in \{1, \ldots, N\}$ is evaluated using O. The value from the objective function then yields the estimated performance of each individual.

Selecting the parents
As each x_i has an assigned evaluation value, and the μ best vectors gets selected by taking the x_i vectors with the highest ranking.

Updating the distribution parameters
When updating the distribution parameters for the next iteration (m_{t+1},M^2_{t+1}), the mean is updated by computing the centroid of the μ highest ranking vectors. This is formally defined as:

$$m_{t+1} := \frac{\sum_{i=1}^{\mu} x_i}{\mu} \quad \text{where} \quad O(x_1) \geq, \ldots, \geq O(x_\mu) \geq, \ldots, \geq O(x_N)$$

The diagonal matrix of variances M^2_{t+1} is updated to match the variance of the μ best vectors, such that the variance in dimension k matches the variance of the μ in dimension k. Hence, for $k \in \{1, \ldots, n\}$, the (k,k)'th entry in the variance matrix is updated This is formally defined as:

$$M^2_{t+1_{k,k}} := \frac{\sum_i^{\mu}(x_i - m_{t+1})^T(x_i - m_{t+1})}{\mu} + Z_{t+1} \ , \quad k \in \{1, \ldots, n\}$$

2.5 CMA-ES

This description is based on the tutorial written by Nikolaus Hansen (Hansen, 2015). This section will not focus on the theoretical derivation of CMA, but rather how it deviates from Cross Entropy. The CMA operates on a general level much like the Cross Entropy method, but includes some features that increases the adaptability of the algorithm.

The CMA, like the Cross Entropy, uses a Gaussian distribution to search for good solutions to O. Yet, for the variance parameter, the CMA provides a full covariance matrix. As the Cross Entropy method only provides a diagonal matrix of scalers, it's restricted to only scaling the ellipsoid of equal density along the coordinate axes. The CMA however, with a full covariance matrix, allows the ellipsoid to rotate arbitrarily in the search space.
Another difference between the two algorithms is that Cross Entropy only considers the next population when updating the distribution parameters, while the CMA keeps track of some information from earlier generations. This allows the CMA somewhat keep track of the evolution of the sampled vectors.
The CMA also differs from Cross Entropy in how it evaluates the influence of the parent vectors. As Cross Entropy weights all vectors equally when moving the mean. The CMA, at least from the implementation in SHARK, has the option of taking a weighted combination of the offspring in order to bias towards the better vectors.

CMA-ES

input
O: The function that estimates the performance of a vector x
(m, C): The mean and variance of the initial distribution, where C is the covariance matrix usually set to $C = I$
N: The number of vectors sampled per generation/iteration
μ: The number of offspring selected for the new mean

initialization
Set initial internal parameters

loop
Sample new generation
Evaluate each vector using O and recombine
Step-size control
Covariance matrix adaption
end loop

Figure 5: The pseudo code for the Cross-Entropy algorithm

ALL CMA SPECIFIC BELOW THIS IS NOT DONE!

2.5.1 Input

The objective function
This serves the same purpose as in Cross Entropy (see page 12).

The mean and variance of the gaussian distribution
Here m_t is the mean and C_t^2 is the variance of the gaussian distribution (m_t, C_t^2). More specifically this gaussian distribution is defined as

$$\mathcal{N}(m_t, C_t^2)$$

Where t denotes the current iteration.

The number of vectors
N is the number of vectors sampled in each generation.

The number of offspring
μ is the number of vectors which are used to compute the new mean very much like in Cross Entropy. However the CMA algorithm is not bound to weight each vector equally. It has the option to assign a weight to each vector and hence biasing towards the better solutions.

2.5.2 Initialization

Set parameters

$$N, \mu, w_{i...\mu}, c_\sigma, d_\sigma, c_c, c_1, c_\mu$$

To their default values according to table 1 in (Hansen, 2015).
Set evolution path $p_\sigma = 0$, $p_c = 0$, covariance matrix $C = I$ and $t = 0$

2.5.3 Loop

Sample new generation
Sample new population of search points, for $k = 1, \ldots, N$. The aim is to sample vectors in the following form

$$x_i \sim \mathcal{N}(m, \sigma^2 C)$$

To generate the sample for the generation, the covariance matrix is first decomposed as follows:

$$C = BD^2B^T$$

Where B is a matrix of all eigenvectors of the covariance matrix, and D is a diagonal matrix that contains all eigenvalues.

Intermediately, the vectors are sampled from a Gaussian with zero mean and the identity matrix as variance.

$$z_k \sim \mathcal{N}(0, I)$$

The matrix D will then scale the vectors such that the samples are distributed along the principal axes. The matrix B, containing the eigenvectors will rotate samples according to estimated covariance.

$$y_k = BDz_k \sim \mathcal{N}(o, C)$$

The mean is then added, and the vectors are scaled by the step-size.

$$x_k = m + \sigma y_k \sim \mathcal{N}(m, \sigma^2 C)$$

Evaluate each vector using O and recombine
As each vector is evaluated, they are ranked such that $O(x_i) \geq \cdots \geq O(x_\mu)$.

$$\langle y \rangle_w = \sum_{i=1}^{\mu} w_i y_{i:N}, \text{where} \sum_{i=1}^{\mu} w_i = 1, w_i > 0$$

The new mean is computed as:

$$m_{t+1} = m_t + \sigma \langle y \rangle_w = \sum_{i=1}^{\mu} w_i x_{i:N}$$

Step-size control
The evolution path for the step-size is updated according to the path of the last iteration and the covariance matrix.

$$p_\sigma \leftarrow (1 - c_\sigma)\, p_\sigma + \sqrt{c_\sigma\,(2 - c_\sigma)\,\mu_{\text{eff}}} C^{-\frac{1}{2}} \langle y \rangle_w$$
$$\sigma \leftarrow \sigma \times exp\left(\frac{c_\sigma}{d_\sigma}\left(\frac{||p_\sigma||}{E||\mathcal{N}(o, I)||} - 1\right)\right)$$

Covariance matrix adaption
The evolution path for the covariance matrix is updated much like for the step-size, and the covariance matrix is updated to gain shape of the estimated covariance of the new selected vectors.

$$p_c \leftarrow (1 - c_c)\, p_c + h_\sigma \sqrt{c_\sigma\,(2 - c_c)\,\mu_{\text{eff}}} \langle y \rangle_w$$
$$C \leftarrow (1 - c_c - c_\mu)\, C + c_1 \left(p_c p_c^T + \delta\,(h_\sigma)\, C\right) + c_\mu \sum_{i=1}^{\mu} w_i y_{i:\mu} y_{i:\mu}^T$$

Finally the loop is iterated until a certain criteria is met, or if the generation counter exceeds an upper limit.

2.6 Previous work

Over the time, numerous researchers has tried different feature sets and applied various optimizers to find the best possible Tetris controllers. The features used are typically ones that attempt to mimic the board conditions that would normally catch the attention of a human player, such as how high the overall pile of pieces is and how many holes the board has. In (Scherrer, 2009a) table 1, a table presents some feature sets used throughout various publications on the subject. In later works, many authors have had success with applying evolutionary stochastic search methods for tuning the weights of the feature sets towards efficient controllers. For the purpose of this thesis, we are in particular addressing the Cross-Entropy method described in detail in (Pieter-Tjerk De Boer and Rubenstein, 2014) and the Covariance Matrix Adaption Evolution Strategy (CMA-ES), described in (Hansen, 2015). The particular Cross-Entropy method applied is the one described in (Szita and Lörincz, 2006) as the "Noisy Cross Entropy Method".

Feature	**Description**
Landing height	The height of the piece when it lands
Eroded piece cells	Number of rows cleared in the last move times the number of bricks cleared from the last move
Row transitions	Number of horizontal cell transitions
Column transitions	Number of vertical cell transitions
Holes	Number of empty cells covered by a full cell
Board wells	Cumulative sum of cells to the depth of the board wells.

Figure 6: features of the Dellacherie controller

Currently, many researchers have proposed numerous feature sets and multiple optimization methods have been explored. A controller often referred to is the Dellacherie controller (Fahey, 2003). This controller was hand-tuned by trial and error, and did originally, on a regular non-simplified Tetris game, achieve an average of 660 000 lines. The same feature set (see figure 6) is often incoorporated in later works when optimizing controllers. An earlier feature set is the set proposed by (Bertsekas and Tsitsiklis, 1996) referred to as Bertsekas and Tsitsiklis features. In 2006, Szita and Lőrincz (Szita and Lörincz, 2006) applied the Cross-Entropy method using the Bertsekas and Tsitsiklis features. They report that using no noise, their controller converged at 300 000 lines on average. The best result reported in (Szita and Lörincz, 2006) is when decreasing noise is applied, in which the controller's score exceeded 800 000 lines. However, in a later paper, using Dellacherie, Bertsekas and two selfdefined features achieved 35.000.000 lines ±20% (Scherrer, 2009b).

Creating a Tetris-controller is a NP-complete problem, where we want to find a strategy which maximizes the average score. Most researchers utilize three general approaches to create policies.

- Handwritten controllers
- Reinforcement learning approaches
- Optimization algorithms

But as seen in (Scherrer, 2009b), a combination of the methods can yield good results, where Thiery and Scherrer employed Dellacherie's handwritten policy and used CE to optimize it.

2.7 Normalization of samples

As mentioned by some authors (Boumaza, 2009), the vector that describes the agent can very well be normalized such that the vector is a point that lies on the n-dimensional hypersphere.

The reason for this lies in the nature of the evaluation function. When the controller chooses an action, it will evaluate all the possible actions possible with the current piece. It will use the value function V of each state s_i and choose the state with the highest value from the value function. Thus, if the states are ordered such that:

$$V(s_1) > \cdots > V(s_N)$$

The agent then chooses the action that transitions from the current state to state s_1.
Since the value function assess the state by the following:

$$V(s) = \sum_{i=1}^{n} w_i \phi_i(s)$$

Then scaling the input of the agent, the weight vector w, by a number $a \in \mathbb{R},\ a > 0$ the assessment is changed by:

$$\begin{aligned} \sum_{i=1}^{n} a w_i \phi_i(s) &= a \sum_{i=1}^{n} w_i \phi_i(s) \\ &= aV(s) \end{aligned}$$

And the ordering remains:

$$aV(s_1) > \cdots > aV(s_N)$$

Thus the order of the value functions of each state does not change, and the same s_1 is still chosen for any $a \in \mathbb{R},\ a > 0$.

To verify this, the Tetris objective function was executed with the same vector and the same seed for the random generator with a scale $a \in \{0.1, 0.2, 0.3, \ldots, 9.8, 9.9, 10.0\}$, and the agent scored exactly the same for each scale.

This can be used in experiments for various reasons. As reported in (Boumaza, 2009), normalizing the samples will prevent CMA-ES from diverging in step size, and it can prevent loosing precision if the magnitude of weight vector becomes larger than feasible for the used floating point number and avoids size limitations.

2.8 Assesment of controller performance

The performance of a one-piece controller has a very high variation, and is in other research verified to be exponentially distributed. As a result of the high variance of the controllers, the performance of single controllers are often presented along with a confidence interval for the estimated mean score of the controller. The estimated mean of the controllers score is calculated by

$$\hat{m} = \sum_{i=1}^{N} x_i$$

Thus, the maximum likelihood estimation of the rate parameter[4] of the distribution is given by

$$\hat{\lambda} = \frac{1}{\hat{m}}$$

A confidence interval is found by the following

$$\frac{2N}{\hat{\lambda}\chi^2_{1-\frac{\alpha}{2},2N}} < \frac{1}{\lambda} < \frac{2N}{\hat{\lambda}\chi^2_{\frac{\alpha}{2},2N}}$$

However, for these experiments, an approximation for a 95% confidence on lower and upper bound of the rate parameter λ is used

$$\lambda_{low} = \hat{\lambda}\left(1 - \frac{1.96}{\sqrt{N}}\right)$$
$$\lambda_{upp} = \hat{\lambda}\left(1 + \frac{1.96}{\sqrt{N}}\right)$$

[4]Note that λ is in this context not, the population size but instead the rate parameter for the exponential distribution.

By this, the 95% confidence interval for the mean m is

$$\frac{1}{\lambda_{low}} < m < \frac{1}{\lambda_{upp}}$$

When a controllers score is presented as "$s \pm p$" this means has an empirical mean score of s and a real mean that is with 95% likelihood within $s \pm p$.
ADD REFERENCE TO EXPONENTIAL DISTRIBTUTIONS

3 Our Contribution

3.1 Implementation of Cross Entropy

- WRITE STUFF HERE
- MAKE SURE STATE THAT OUR IMPLEMENTATION HAS BEEN INTEGRATED IN SHARK

3.2 Emperical comparison between CMA and CE

WRITE STUFF HERE

3.3 Hypothesis

Prior to any execution of experiments it's assumed that, purely based on a theoretical standpoint, that the CMA algorithm will be able to find better solutions than Cross Entropy. The assumption is based on the fact that CMA is, to a higher degree, able to adjust to the search space. Yet, as Cross Entropy is a simpler algorithm that needs to adjust fewer parameters. Hence, another expectation is that the Cross Entropy will converge faster than CMA, but at lower scores.

MORE INFORMATION ON EXPECTED RESULTS OF TUNING CE AND CMA
Hypothesis - CMA vs. CE Tilføj ekstra section forklare at nu starter den eksperimentelle del hvor vi vil benytte algoritmerne til at teste mod hinanden?

4 Experiments

SOME KIND OF INTRODUCTION

4.1 MDP-Tetris

When running the experiments, the source code of the MDP-Tetris (MDPTetris, 2009) was used to emulate the Tetris games. The source code is accompanied with files that describe the various existing features. These files contains the identifiers of each feature to use, as well as two numbers respectively describing the agents reward function and how to evaluate a game over state. The number for the reward function has remained unchanged at 0 during all experiments. The "game-over" evaluation was for the Bertsekas feature set initially set to 0. Setting the "game-over" evaluation to 0 means that the agent will not distinguish between regular moves and moves that results in losing the game. When running the experiments

with this setting, a large portion of the agents never exceeded a zero mean score. However, setting the value to -1, meaning that a "game-over" move yields $-\infty$ reward, none of the experiments got stuck on only zero scores. An example of the layout of the feature file can be seen in figure 7.

```
0     <- Describes the reward function
-1    <- Actions leading to game over is avoided at all cost
22    <- The policy contains 22 features
8 0   <- The feature with id 8 initially has weight 0
...   <- The remaining 21 features
```

Figure 7: Example of a file that describes a feature set.

4.1.1 Featureset

As described in THEORY SECTION REF, there are different kind of featuresets which impacts the performance of the agents. In our upcoming experiments, we are going to use both the Dellacherie and Bertsekas/Tsitsiklis featureset. For our initial comparison experiments we are going to use the Bertsekas/Tsitsiklis featureset, since other researchers report a lower score with the Bertsekas/Tsitsiklis featureset compared with the Dellacherie featureset (Thiery and Scherrer, 2009). Compared to the (Thiery and Scherrer, 2009) experiments, we don't want to maximize the final score, but rather maximize the score with the lowest possible number of games evaluated.
It's important to note that we are not going to conduct comparison experiments with different featuresets in the same experiment, since the algorithms needs to be on equal terms.

4.1.2 Game complexity

- WRITE STUFF ABOUTGAME COMPLEXITY AND THE DIFFERENCE IT CAN MAKE

- THE STUFF ABOUT HARD TETRIS IN THE COMPARISON SECTION SHOULD BE HERE? THIS PROBABLY REQUIRES SOME RESTRUCTURING OF THE REPORT TO MAKE SENSE. FOR EXAMPLE WE CONCLUDE IN THE FEATURESET COMPARISON SECTION THAT WE ARE GOING TO USE HARD TETRIS FOR CONFIGURATION OF CMA, WHICH DOESN'T MAKE SENSE, SINCE CMA "ALREADY" HAS BEEN CONFIGURED AT THIS POINT.

4.2 Setup

When executing the experiments, various parameters each have an impact on the final result of the learning curve. Thus, the parameters are adjusted, first to match the experiments run by other researchers, and later to conduct as fair as possible comparisons between Cross-Entropy and CMA-ES.

The amount of vectors sampled in each generation N has a high impact on the algorithm

performance. By setting N high, more policies are evaluated per iteration, and leads to a more thorough exploration of the search space. Thus the higher N increases the chances of finding a better mean for the next iteration. However, higher N also results in the need for more evaluations per iteration. The goal for tuning this parameter is then to set N high enough to ensure exploration of good solutions, and yet low enough to avoid unnecessary evaluations.
In the implementation of CMA-ES from SHARK (Igel et al., 2008) , the algorithm itself determines the value of N according to the size of the search space. Cross-Entropy however, does not seem to have a general rule for this parameter, so this value is manually adjusted to fit the problem as well as possible.

As both of the optimizing algorithm uses a subset of the sampled vectors from a generation to update the distribution parameters, the number of offspring μ influences how the next generation is sampled. By setting the value too high, the algorithm risks ceasing to progress any further since the updated mean would be too close to the previous one to significantly make a difference. By setting the value too low, the risk of reaching a local optimum increases since the high-scoring agents might have reached their high performance by chance.
The CMA-ES itself manages setting μ and Cross-Entropy is set according to the problem. Most authors that uses Cross-Entropy for Tetris sets the offspring size to 10% of population size, that is $\mu = \lfloor 0.1 \cdot N \rfloor$.

The number of games, γ, is the number of games which each agent plays in each iteration. An agent's score is defined as the mean of the score of these γ games. We want this value low as possible, because as with the number of agents, N, The number of games, γ, is another major factor in the run-time of the algorithm. As Tetris is stochastic by nature, the score deviates a lot, even when the same agent with the same policy plays multiple games. Hence, when assessing the true performance of a policy it's rarely enough to play just few games. Thus, setting γ high increases the likelihood of correctly choosing the best agents, yet, it also causes longer run times of the experiments.

Specific to the Cross-Entropy method, most authors report that the performance of the algorithm increases dramatically when the sampling distribution is associated with a noise term. The different types of noise are described in section 2.4. The noise term is adjusted in order to prevent the algorithm from reaching a local optimum. The current research shows that noise terms of $Z_t = 4$ and $Z_t = max\,(5 - t/10)$ (Thiery and Scherrer, 2009) produces the best results. The constant noise (such as $Z_t = 4$) ensures that the algorithm never settles in a too small area from which it samples, and forces it to explore solutions that are further away from the mean. The further the algorithm progresses, the less noise is assumed needed, as the mean should approach a global optimum. to address this, the linear decreasing noise is applied as it will lower the noise term as the algorithm progresses.

For the various experiments, these parameters will be tuned for the specific purpose at hand. In the verification of the Cross-Entropy, the parameters are set to match those reported in similar papers (Thiery and Scherrer (2009), Szita and Lörincz (2006)). In the comparison of the two algorithms, the parameters will be set such that the Cross-Entropy operates under as similar conditions as CMA-ES, to ensure an unbiased comparison.

4.3 Verification of CE

Because the SHARK (Igel et al., 2008) library already contains an implementation of CMA-ES, but not an implementation of CE, we extended the library with our own implementation of the algorithm.
In order to verify the correctness of the implementation, we used the same experiments as used by Christophe Thiery and Bruno Scherrer (Thiery and Scherrer, 2009). These experiments were used by Thiery and Scherrer to verify their own CE implementation with various types of noise correction. Therefore, we will perform the same experiments to verify our own contribution to the SHARK (Igel et al., 2008) library, by trying to achieve the same results.

The setup is mirrored from the paper (Thiery and Scherrer, 2009), with 100 agents ($N = 100$) per iteration, and using the $\mu = 10$ best vectors for the update step. After each iteration, an agent with the updated mean plays 30 games and the mean of these scores are recorded for the learning curve.
During evaluation each agent plays one game, that is $\gamma = 1$. A minor derivation from the figures present in (Thiery and Scherrer, 2009), is the unit along the x-axis in the learning curve plots indicates the iteration number. As the experiments in this two algorithms width variable population sizes are compared, the x-axis in all plots indicated the number of tetris games played during the learning. In one generation N agents each play γ games and hence adavance the x-axis by $N\gamma$.

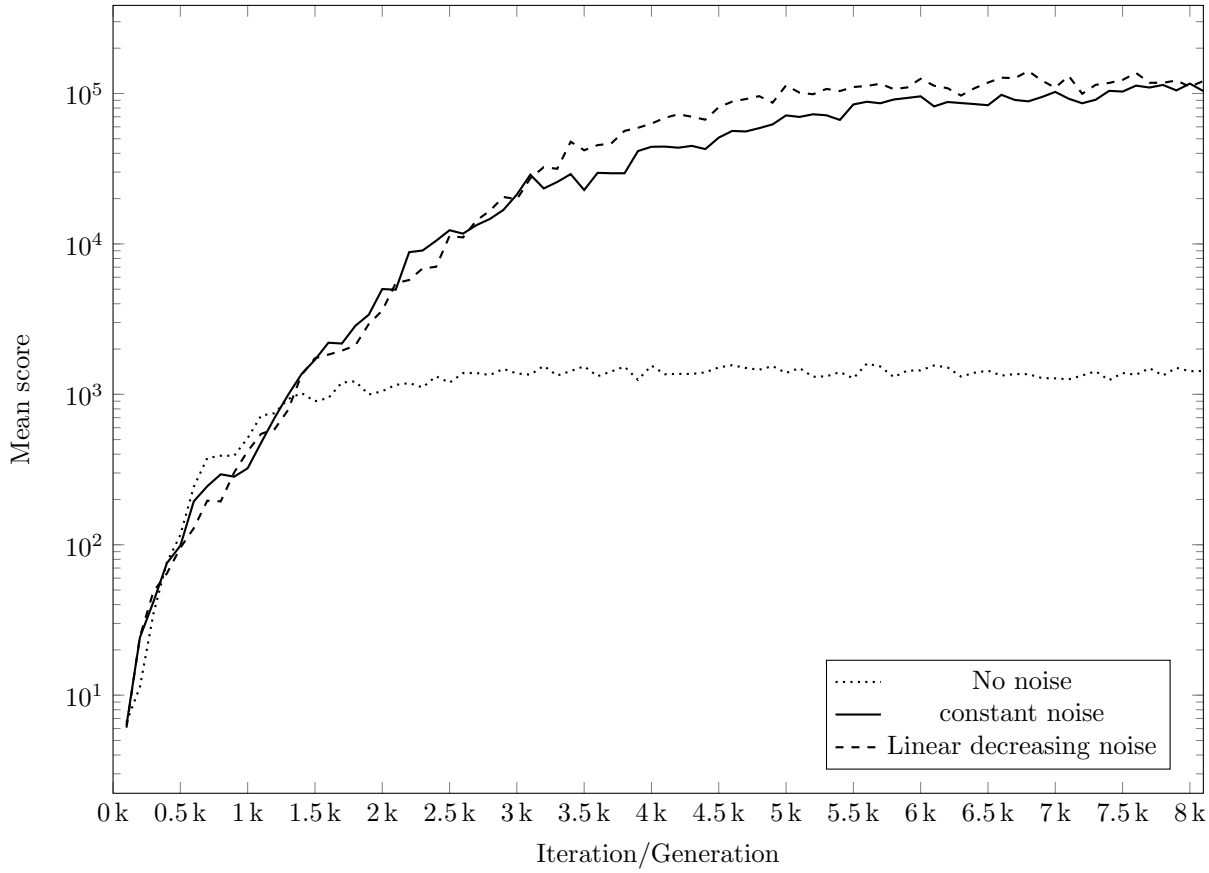


Figure 8: Cross-Entropy mean performance

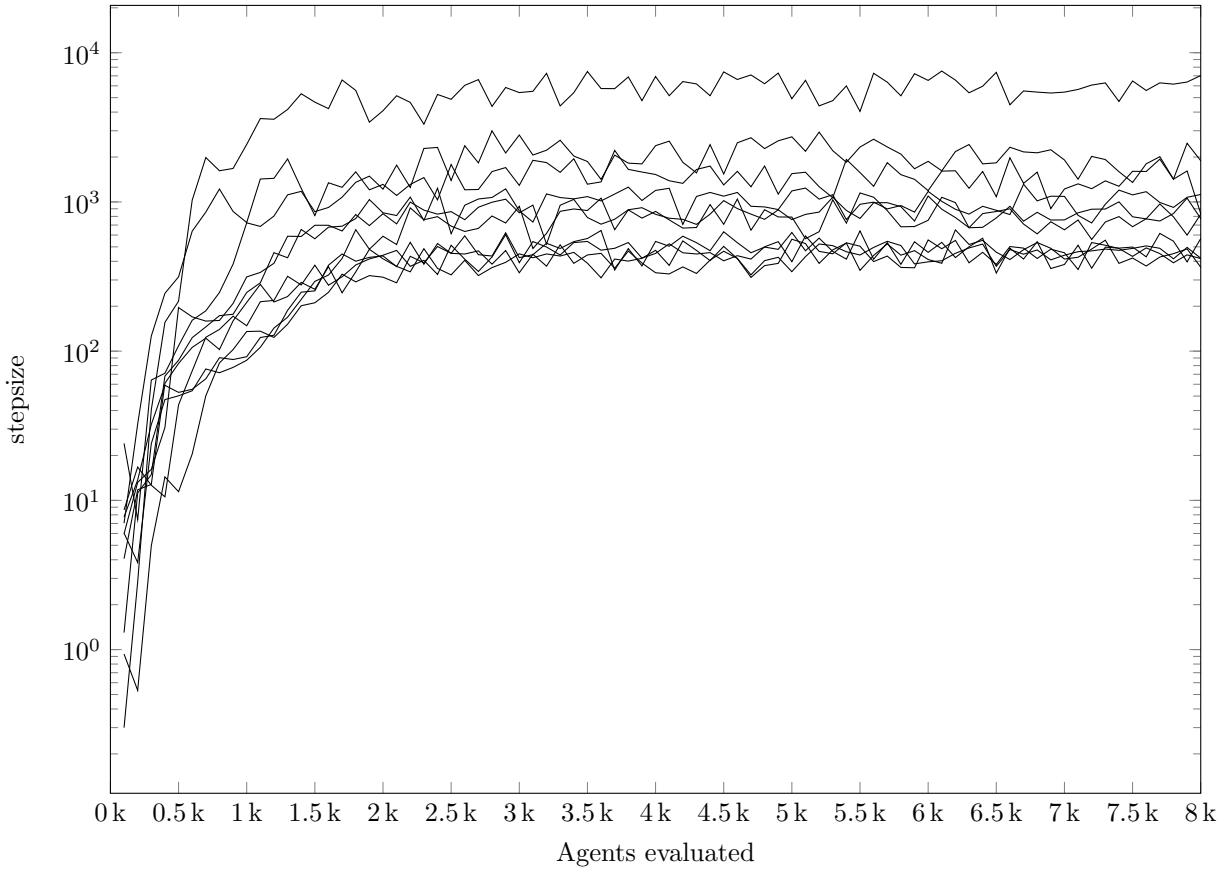


Figure 9: No noise

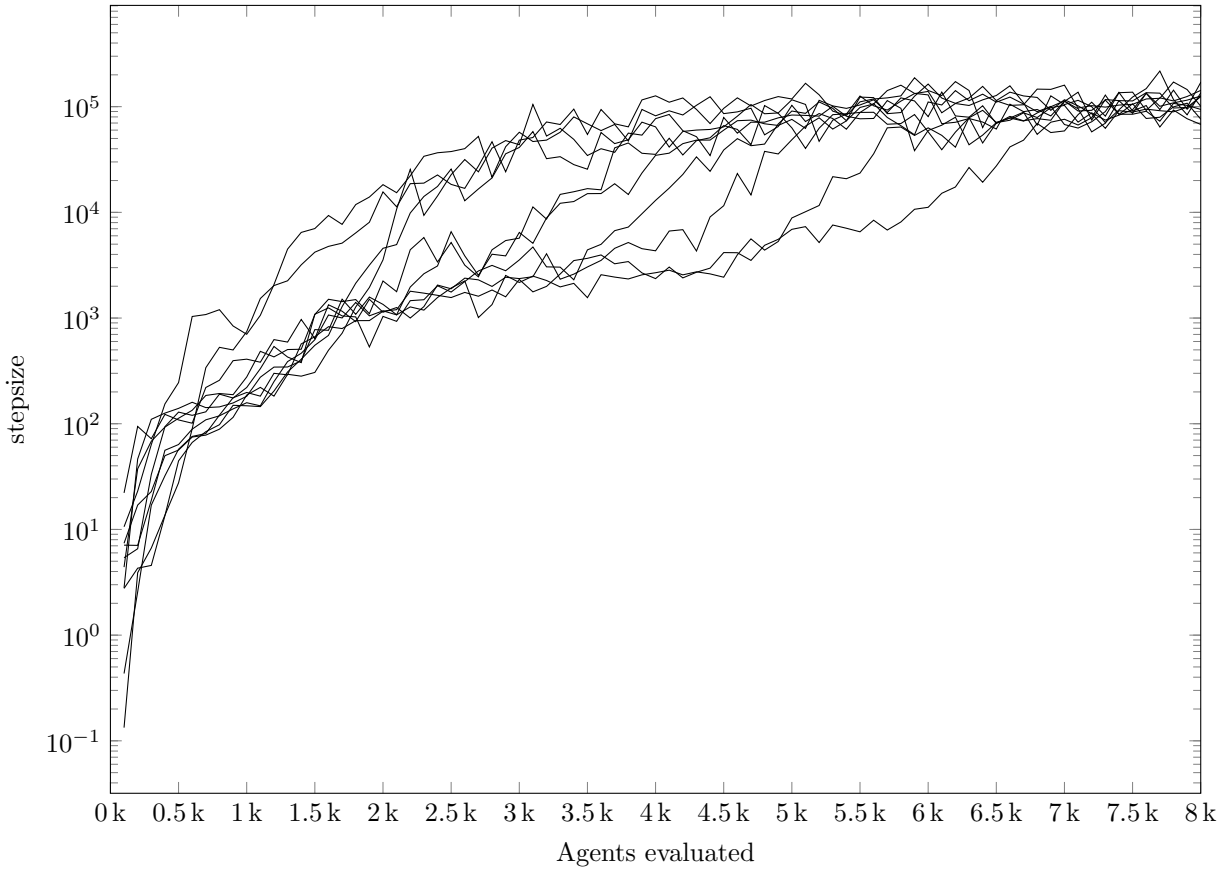


Figure 10: Constant noise

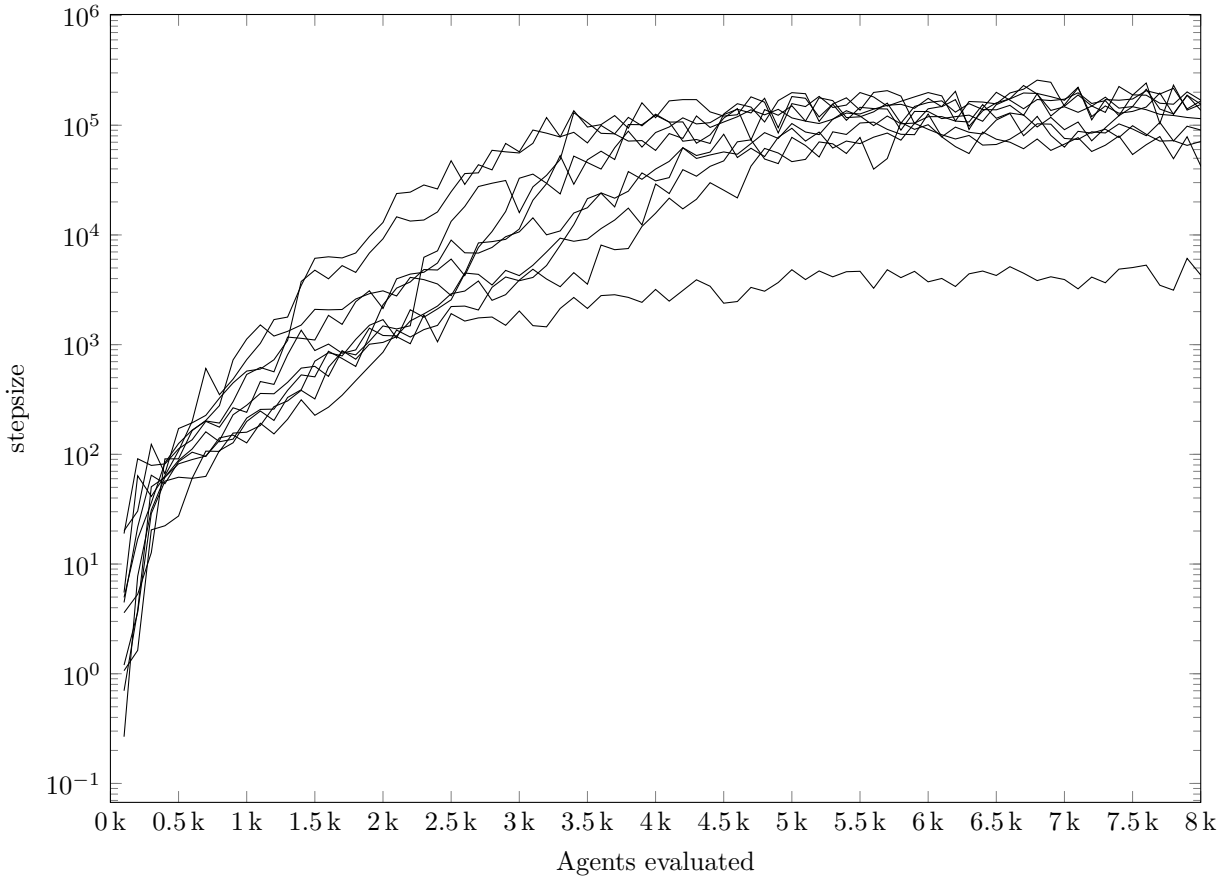


Figure 11: Linear decreasing noise

Figure 9, 10 and 11 shows 10 runs of each noise type. Figure 8 shows the mean graph for each of the noise types. The goal of these experiments were to replicate the experiments reported in (Thiery and Scherrer, 2009). As the results seen from our experiments to a high degree resemble those reported by Thiery et. al, we conclude that our Cross-Entropy implementation works similar to theirs.

When evaluating the score of the agent we also want to compute the confidence interval in verifying the implementation of CE. The mean agent plays 30 games which leads to a confidence interval of ±36% around the estimated mean, which is similar to the confidence intervals in (Scherrer, 2009b).
By looking at the individual graphs for the different noise types (Figure 9, 10 and 11), we get the following average scores.

Without noise (figure 9): The learning curve stabilizes after 1,500 agents evaluated. And as it can be observed the score variates much for the different executions between a score of 300 and 6,000 rows. This results in an average score of 1,400±36% rows.
Constant noise (figure 10): The 10 executions reaches equivalent performances at some point, with a score between 54,000 and 154,000. This results in an average score of 105,000±36% rows.

Linear decreasing noise (figure 11): Most of the executions of this noise type settles around 200,000. However, a single execution settled at a score of only around 5,000. The mean performance of this noise type yielded a score of 120,000±36%

Based on the mean graphs and confidence interval compared to other papers, we can hereby verify that our implementation of CE works as intended. Even though the experiments with linear decreasing noise in this case seems to outperform the constant noise, other runs with linear decreasing noise ended in a mean performance of only 90,000±36%. Yet, the constant noise is both from our own experiments, and described in other research, noted to be the most reliable noise type for reaching high scoring controllers (Scherrer, 2009b). Due to this, the constant noise is used in the benchmarking against CMA-ES.

4.4 Optimal settings for Cross Entropy

In this section we will determine the optimal settings for Cross Entropy through testing of various adjustable parameters.
Compared CMA, Cross Entropy has fewer customizable parameters, more specifically population-parent size and games played per agent.

4.4.1 Population and selection size

Other researchers run the Cross Entropy algorithm with population size of $N = 100$ and an offspring corresponding to 10% of the population size, resulting in $\mu = 10$. As it's not discussed why this exact setting is applied, various settings of the Cross-Entropy was executed to asses the performance of other configurations in our experiments. The experiments includes different population sizes $N \in \{10, 22, 50, 100, 200\}$ and offspring sizes of either 10% and 50% (since the CMA algorithm by default uses 50% selected vectors). A summary of the experiments can be seen in figure 12 on page 28.

N	μ	mean	Q1	Q2	Q3
10	10%	704.6	7.2	48.3	430.3
10	50%	9,272.5	149.6	7626.5	16,919.9
22	10%	35,841.6	20,391.9	42,045.5	48,464.6
22	50%	52,887.4	23,531.9	42,161.0	83,144.1
50	10%	95,623.1	82,738.9	93,388.9	111,351.5
50	50%	69,130.7	52,511.0	64,351.6	91,488.6
100	10%	115,868.7	84,368.5	122,238.5	146,457.0
100	50%	22,910.4	4,037.7	14,353.7	47,215.9
200	10%	85,181.7	45,201.5	96,803.1	117,578.0
200	50%	946.4	585.0	802.5	1,267.7

Figure 12: Cross Entropy configuration test, see Appendix C for the full plots of the experiments.

The experiments with different population and parent sizes does not seem to support a choice for any other configuration than the mostly commonly used $N = 100$ and $\mu = 10$.
However, with a configuration of $N = 50$ and $\mu = 5$ convergence is achieved faster (see figure 13). This means that the score limit is reached faster, which results in longer computation

time, than the $N = 100$ and $\mu = 10$ configuration. In other words, the $N = 100$ and $\mu = 10$ configuration is therefore preferred since it takes shorter computation time and the end-result is similar compared to the $N = 50$ and $\mu = 5$ configuration, even though the latter configuration from our expriments converged faster. The experiments also apppears to suffer from a high noise, yet both the mean and the quantiles favor the extensively tested Cross Entropy configuration of $N = 100$ and $\mu = 10$.

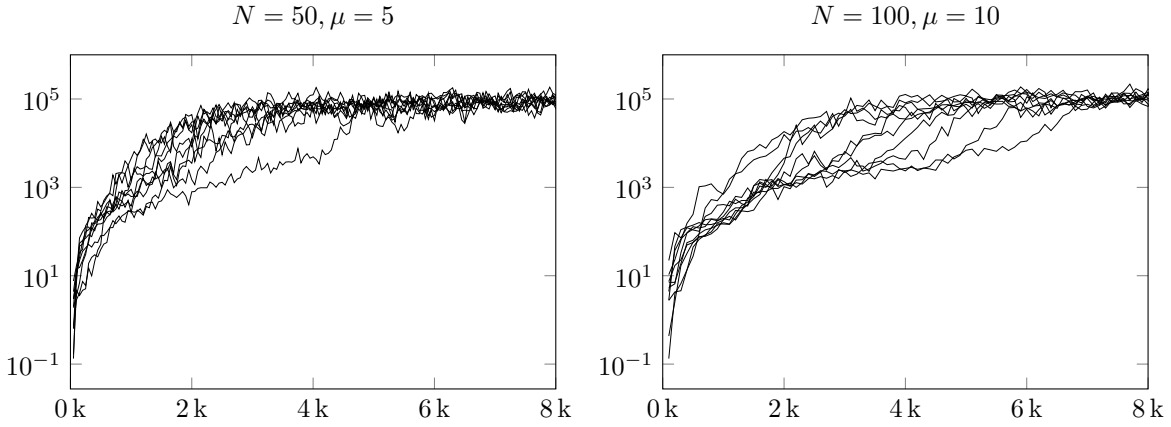


Figure 13: The two best configurations of Cross Entropy

4.4.2 Games per agent

Another thing we can adjust is the number of games played each agent plays per generation. By playing multiple games with each agent and using their mean score to compute the next generation mean. This secures a higher chance of a better offspring generation, but at the cost of more games evaluated.
Table 12 shows the different values of games per agent we are going to be testing.

Population Size	Parent size	Games per Agent
10	10%	1/3/5/7/10
10	50%	1/3/5/7/10
22	10%	1/3/5/7/10
22	50%	1/3/5/7/10
50	10%	1/3/5/7/10
50	50%	1/3/5/7/10
100	10%	1/3/5/7/10
100	50%	1/3/5/7/10
200	10%	1/3/5/7/10
200	50%	1/3/5/7/10

Table 1: Games per agent CE experiment setup

The experiments includes different games per agent of $\{1, 3, 5, 7, 10\}$. By increasing the number of games played per agent, we are increasing the evaluation cost by a multiplication factor in exchange for a more accurate evaluation of each agent. This is to avoid bad agents having a lucky game with an better than average game, or the opposite, an good agent having a

lesser than average game. However, we wish to find the best performing configuration with the lowest cost and maximum benefit.
- CONDUCT EXPERIMENTS, SHOW RESULTS, MAKE CONCLUSION
- MAKE SURE TO MENTION THAT WE USE HARD TETRIS FOR THIS EXPERIMENT

4.5 Optimal settings for CMA

When finding the optimal settings for CMA we have a larger set of parameters we can adjust compared to Cross Entropy. Again, as the same with Cross Entropy, we can adjust the population/parent size and games per agent. Furthermore we can adjust the initial step-size, lower bound and recombination type.
The following sections will go into detail how these parameters affect the performance of CMA algorithm. In the last section we will perform experiments to determine the optimal settings.

4.5.1 Recombination type

Furthermore, CMA also has a unique formula for calculating the updated mean, called the 'Recombination type' 2.5. Where the recombination type determines how much influence each of the offspring vectors has on the next generation. Built into the CMA algorithm is three methods of recombination.

- EQUAL, Each of the offspring vectors has equal influence in the generated mean. Each has $w_i = 1$.
- LINEAR, The best of the offspring vectors has more influence.
- SUPERLINEAR, The vectors are weighted with a logarithmic equation. $w_i = \frac{w'_i}{\sum_{j=1}^{\mu} w'^{+}_j}$

As default, CMA uses Super Linear recombination. However, Tetris is a problem with multiple local optimums in its solution space. This means, though a vector may be the best in its generation, it could be a nearby local optimum. Therefore, Super Linear recombination may not be the optimal recombination type for the Tetris problem.

- WEIGHTS FOR RECOMBINATION COULD HAVE A SYMBOL? (CHANGE POLICY WEIGHT SYMBOL?)
- FIND LINEAR COMBINATION TYPE FORMULA OF CHANGE FORMAT OF ITEMIZE LIST

4.5.2 Population and selection size

By adjusting the population size to that similar of Cross Entropy, we are able to get a fair comparison between the two algorithms, given each generation will contain the same number of agents. By setting the population and parent size to the same values, we in effect test if the covariance matrix and the step-size control has a impact on the algorithm performance compared to Cross Entropy which does not have the features.
Table 2 displays the values that we are going to test for the population and selection size.
The experiments includes different population $N \in \{12, 22, 50, 100\}$ and offspring sizes of either 10%, 25% and 50%. We use 10% because of the Cross Entropy recommended selection

Population size, N	Parent size, μ
12	1
12	3
12	6
22	2
22	5
22	11
50	5
50	12
50	25
100	10
100	25
100	50

Table 2: CMA configuration for population and parent size

size, while we use 25% because of CMA's standard selection size for the EQUAL recombination type. Furthermore we use 50% because of CMA's standard selection size for the LINEAR and SUPERLINEAR recombination type.

4.5.3 Games per agent

As with Cross Entropy we can also adjust the number of games each agent plays per generation. However, because of the recombination type for CMA, one game pr. agent may be insufficient to assess the performance of an agent. The Linear and Super Liner recombination types will value the better agents higher. Therefore, it may occur that some better-on-average agent encounter an unlucky game, achieving a lower score than it's actual potential allows. Thus, evaluating each agent multiple times and using the average score for recombination may allow for a more accurate assessment.
Table 3 displays the values that we are going to test for games per agent.

Games per agent
1
3
5
7
10

Table 3: CMA configuration for games per agent

The experiments includes different games per agent of $\{1, 3, 5, 7, 10\}$. We have chosen the same values as with Cross Entropy to get comparable results (see section 4.4.2).

4.5.4 Initial step-size

Initially, the covariance matrix of CMA in generation $t = 0$ is the identity matrix. The initial step-size, σ_0, will hence in the first iteration scale the area in which the CMA algorithm searches. As from section 2.7, it's known that the scale of the solutions has no impact on the

scores. Hence, it's assumed that the initial step-size should not have any major impact on the results.

σ_0	mean	Q1	Q2	Q3
0.1	50769.3	21301.1	54588.7	73972.4
0.2	42290.6	32180.2	42290.6	49337.4
0.5	53893.7	14211.1	66773.0	85816.7
0.8	37557.7	1422.8	15450.8	93719.4
1.0	49537.9	31369.8	49537.4	58454.6

Figure 14: Results of CMA-ES with adjusted initial step-size

ADD THE DATA-GRAPHS TO APPENDIX

For the initial experiments using CMA-ES, the only adjusted parameter is the initial step-size σ_0. The configurations of step-sizes were $\sigma_0 \in \{0.1, 0.2, 0.5, 0.8, 1.0\}$. As the table shows, the final mean score does not seem to change with the initial step-size. Furthermore, the adjustment of the step-sizes does not appear to have a drastic impact on the mean scores. However, based on both mean score and quantiles, the best configuration seems to be $\sigma_0 = 0.5$. This is is also referred to as a typical initial setting in (Boumaza, 2009). Therefore, the conclusion remains that the initial step-size is not critical for the experiment.

- MAKE SURE TO FULLY CONCLUDE THAT THE INITIAL STEP-SIZE DOESN'T MATTER AND THEREFORE WE DON'T USE THIS PARAMATER IN THE UPCOMING EXPERIMENTS

4.5.5 Lower bound

As with the Cross Entropy method, to avoid early convergence at a too low local optimum, a certain lower threshold for the variance should be applied when sampling vectors for solutions. In the Cross Entropy method, a constant noise term $z_t = 4$ i added to the variance for each component of the sampled vectors. When the i'th component in cross entropy is sampled as follows

$$\begin{aligned} x_i &\sim \mathcal{N}\left(m, \sigma^2\right) \\ &\sim \sigma\mathcal{N}\left(m, 1\right) \end{aligned}$$

Then $\sigma^2 \geq 4$. To gain the same effect for the CMA, a lower bound is applied to the step-size. Such a bound is implemented in the Shark library as the following, where the value of the lower bound is l.

$$\sigma\lambda_n \geq l$$

Where λ_n is the lowest eigenvalue in the covariance matrix. When the vectors are sampled, the samples are scaled by the matrix D containing the eignvalues of the covariance matrix. Hence, if $\sigma\lambda_n \geq l$, then the smallest scaling that takes place is at least l. As the vectors can be written as

$$x \sim m + \sigma B \underbrace{D\mathcal{N}\left(0, I\right)}_{\mathcal{N}(0, D^2)}$$

Where $D\mathcal{N}(0, I)$ corresponds to $\mathcal{N}(0, D^2)$ which, since D is a diagonal matrix, is just the same type of sampling as in Cross Entropy. The matrix B will only rotate the the sampled vectors, and finally, σ will perform an isotropic scaling. To roughly resemble the constant noise configuration of Cross Entropy, when sampling vectors, the lowest variance may not drop below 4. This is achieved by setting a lower bound $l = 2$. If this applies, the sampling can be written as

$$x \sim m + B\mathcal{N}(0, \sigma^2 D^2)$$

Where, with $l = 2$, the lowest entry in the diagonal matrix $\sigma^2 D^2$ is 4.
- WRITE STUFF ABOUT OUR PRACTICAL EXPERIENCES WITH LOWER BOUND AND ADD THE GRAPHS TO APPENDIX
- MAKE SURE TO FULLY CONCLUDE THAT THE LOWER BOUND DOESN'T MATTER AND THEREFORE WE DON'T USE THIS PARAMATER IN THE UPCOMING EXPERIMENTS

4.5.6 Experiment for finding the optimal settings

In this in section we are going to conduct a wide variety of experiments to determine the best settings for CMA. More specifically we are going to find the best combination for population size, parent size and games per agent.
Table 16 shows the different experiments we are going to carry out.

Population Size	Parent size	Recombination Type	Games per Agent
12	1	EQUAL/LINEAR/SUPERLINEAR	1/3/5/7/10
12	3	EQUAL	1/3/5/7/10
12	6	LINEAR/SUPERLINEAR	1/3/5/7/10
22	2	EQUAL/LINEAR/SUPERLINEAR	1/3/5/7/10
22	5	EQUAL	1/3/5/7/10
22	11	LINEAR/SUPERLINEAR	1/3/5/7/10
50	5	EQUAL/LINEAR/SUPERLINEAR	1/3/5/7/10
50	12	EQUAL	1/3/5/7/10
50	25	LINEAR/SUPERLINEAR	1/3/5/7/10
100	10	EQUAL/LINEAR/SUPERLINEAR	1/3/5/7/10
100	25	EQUAL	1/3/5/7/10
100	50	LINEAR/SUPERLINEAR	1/3/5/7/10

Table 4: Full experiments overview

The choice behind using the above population sizes is to use the same population sizes as when we tuned Cross Entropy (see section 4.4). However, we use a population size of 12 instead of 10, because a population size of 12 is the stock setting for Shark CMA (Igel et al., 2008).
In regard to the parent size, they have been determined depending on the recombination type. We use all three different recombination types for the 10% parent sizes to make these experiments correspond to the Cross Entropy settings (see section 4.4). However, Shark CMA has predetermined parent sizes for each recombination types, 25% for EQUAL and 50% for both LINEAR and SUPERLINEAR (Igel et al., 2008). For this reason we also conduct experiments

for each population size to represent these Shark CMA preferred settings. Furthermore we also test different number of games per agent as for the reason presented in section 4.5.3.

Results
SHOW RESULTS, SPECIFY BEST RESUSLTS/CANDIDATES.

Analysis and discussion
DISCUSS BEST RESULTS/CANDIDATES AND CONCLUDE THE OPTIMAL SETTINGS.

4.6 Comparison between Cross Entropy and CMA

- SOME KIND OF INTRODUCTION WHICH STATES THAT THIS ENTIRE SECTION IS WRITTEN "CHRONOLOGICAL"
- IN EACH EXPERIMENT, MAKE SURE TO STATE WHICH PARAMTERS WE USE/ADJUST AND WHAT WE DON'T ADJUST

4.6.1 Global comparison settings

In all papers used for reference we haven't seen any experiments with different population and parent sizes presented side-by-side. All the experiments we've seen that has applied Cross Entropy to Tetris fix the population size to 100. However, in our upcoming experiments CMA and CE are configured with different population and parnet numbers, which means that we cannot compare the learning curves based on iterations/generations. Instead as described in section 4.3, using the *number of games played* as comparison reference, equal terms are secured for both algorithms in regards to learning speed.
Hence x-axis shows the total number of Tetris games evaluated, $\sum_{i=1}^{t} N\gamma$. Meanwhile the y-axis still represents the mean score of the centroid agent at iteration t.

4.6.2 Initial comparison - Bertsekas

For the initial comparison we use the Bertsekas featureset, since the same featureset was used for verifying the Cross Entropy implementation. Furthermore, others researchers has used the Bertsekas featureset as a benchmarking standpoint (Thiery and Scherrer, 2009) & (Szita and Lörincz, 2006).
The goal of this comparison is to get an initial idea of how the Shark implementation of CMA compares to Cross Entropy.

Results
Using Cross Entropy with the constant noise setting and CMA with an initial step-size of 0.5, we get the following results, seen in figure 15.

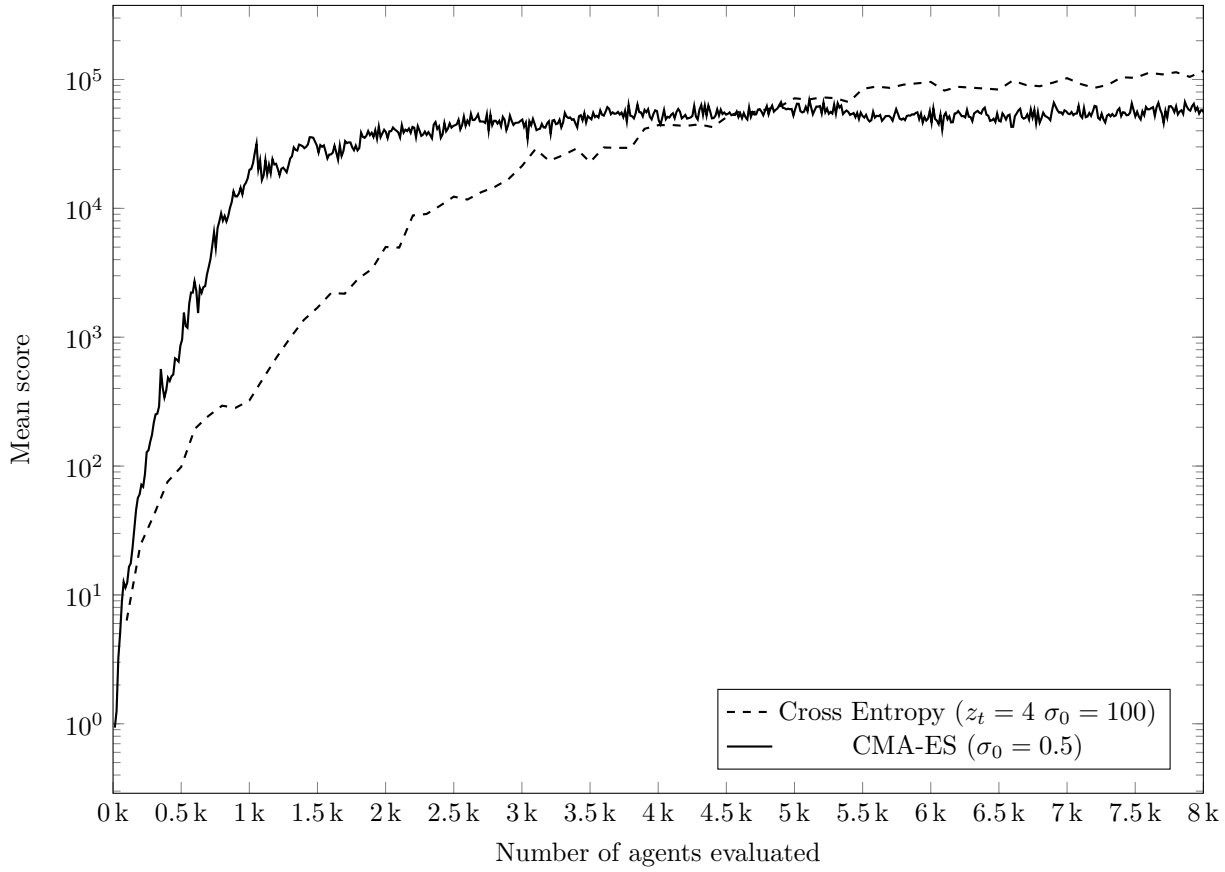


Figure 15: Initial comparison between CMA-ES and Cross Entropy

As figure 15 reveals that CMA converges faster, but reaches a local optimum at around 2,000 games played. Meanwhile CE has a slower convergence but reaches a better mean score compared to CM at around 5,500 agents evaluated. In detail, CMA on average reaches a score of 50,000 rows, and CE reaches a score of 100,000.

ADD THE RAW GRAPHS TO APPENDIX

Analysis and discussion

These results clearly defy our initial hypothesis as we predicted for CMA to outperform CE, due to its more sophisticated nature. One reason for this outcome could possibly be that CMA has a very little population size compared to Cross Entropy, which could be a decisive lack as the objective function is noisy with a high variance. These results indicate a need to adjust the CMA configuration, to prevent the too fast convergance.
Among the possible adjustments are:

Enlargment of population size

As the population size is quite small (only around 13) for this experiment, the CMA algorithm might not be able extract enough information at each generation. As seen in section 4.4, CE looses performance as thepopulation size is set too low. This might be a problem that counts for CMA as well. In figure 15, the CMA algorithm has much the same shape as

Evaluate each agent multiple times
A posible source of poor performance could be that the ranking of agents becomes very difficult due to high noise. To better verify the actual performance of an agent, it could be evaluated multiple times and let the mean of the evaluations determine it's ranking.

Change the recombination type
As described in section 2.5, the CMA algorithm is not bound to update its new mean to just the centroid of the selected vectors. Instead, it can weight better solutions more heavily when moving its mean. When doing so, it risks biasing vectors that appear to be better but in reality, just by faulty ranking, should not be considered a good agent.

None of the experiments seen so far has allowed a consistent mean score of more than 200,000. This brings concern into consideration, that it might be the objective function that poses a natural limit on the score. In this case the objective function models playing Tetris with the Bertsekas featureset. It is unkown to us whether it's possible to construct an agent with a mean score of more than 200,000 lines on average. Yet, as our aim is not to find the best controller, untill both algorithms reaches the same upper limit of scores theres is no need to alter the featureset just to expand the maximum score reachable.

Yet, to minimize the likelihood that the featureset were the cause of the poor performance of CMA, it seemed appropiate to conduct a similar comparison using another featureset. To speed up the process, the game difficulty were adujsted to casue agents to fail faster. The increased difficulty and alternate featureset is described in further detail in section 4.6.3.

4.6.3 Comparison of featuresets

The best configurations of the experiments using the Bertsekas featureset never appears to score much higher than 200,000 lines on average. Due to this, the question of whether the specific featureset causes rapid convergence remains.
To address this, both CMA-ES and Cross Entropy experiments were run 30 times with respectively the Bertsekas and Dellacherie featureset. To prevent long runtimes, the games were simulated using a harder version of Tetris referred to as 'Hard Tetris'. With regular Tetris, all pieces occur with equal likelihood.
To increase the difficulty of the game, in our hard Tetris, the s-block and z-block appear twice as often as the other pieces.

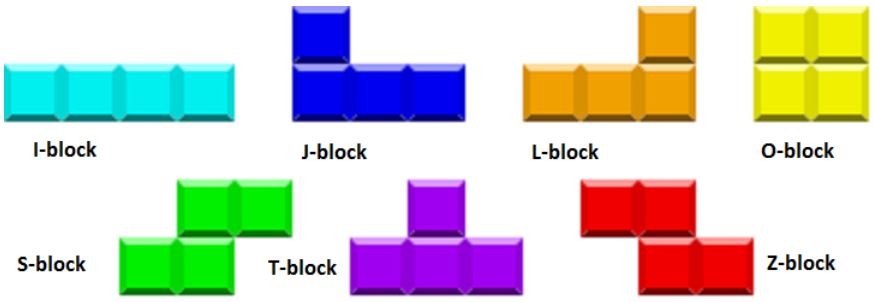


Figure 16: Regular Tetris pieces

Results
In the following figures the mean results of 30 runs of both CMA and Cross Entropy is presented. The settings for Cross Entropy remains at constant noise with a noise term of $z_t = 4$ and an initial sigma of $\sigma_0 = 100.$, and the CMA with $\sigma_0 = 1$.

Figure 17 shows the experiment with the Bertsekas featureset. This shows that when running the algorithms with a harder game, the algorithms behave mostly the same as with regular Tetris. Namely that CMA converges faster than Cross Entropy, but is eventually outperformed.

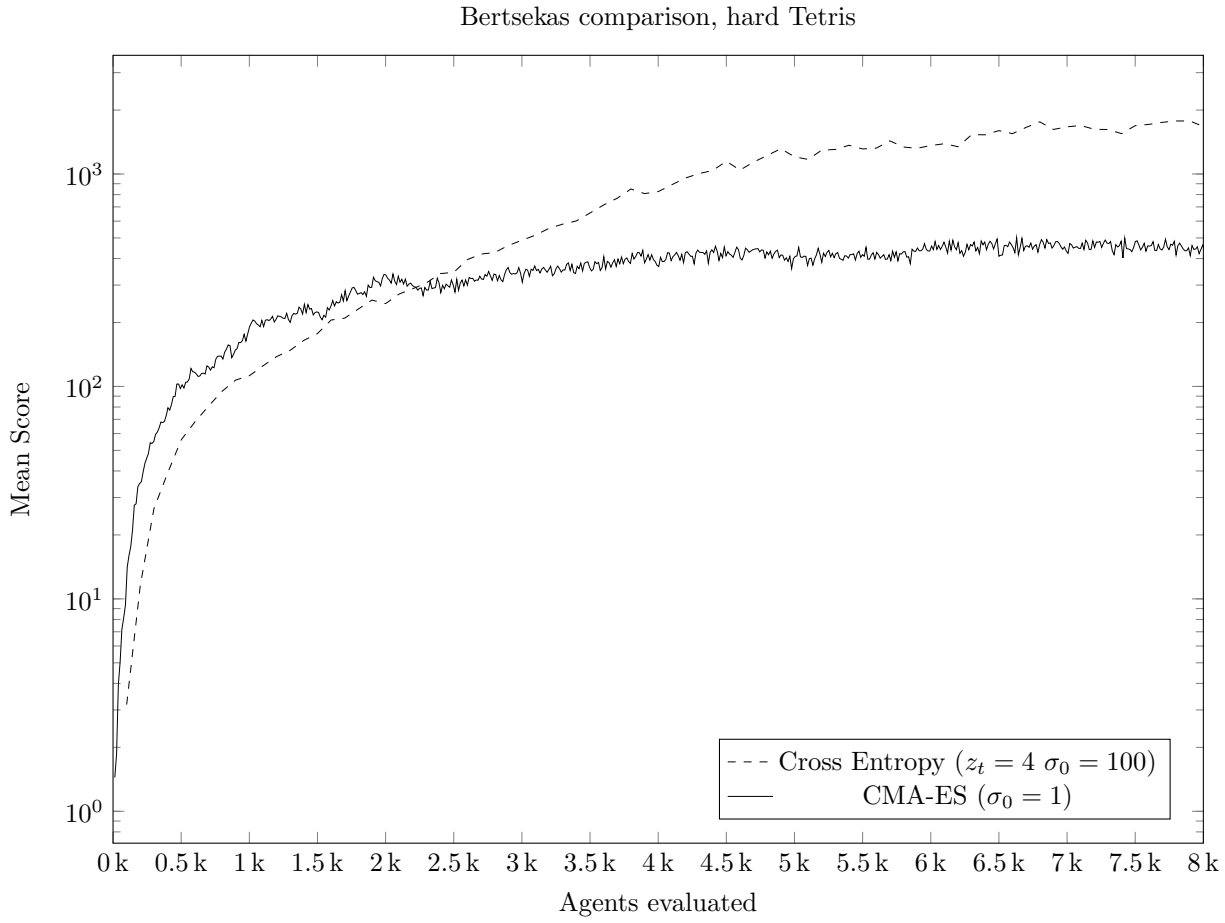


Figure 17: Comparison between CMA-ES and Cross Entropy using hard Tetris and the Bertsekas featureset

When using the Dellacherie featureset a similar behaviour is observed. However, the convergence seems to occur earlier and with a higher score (figure 18).

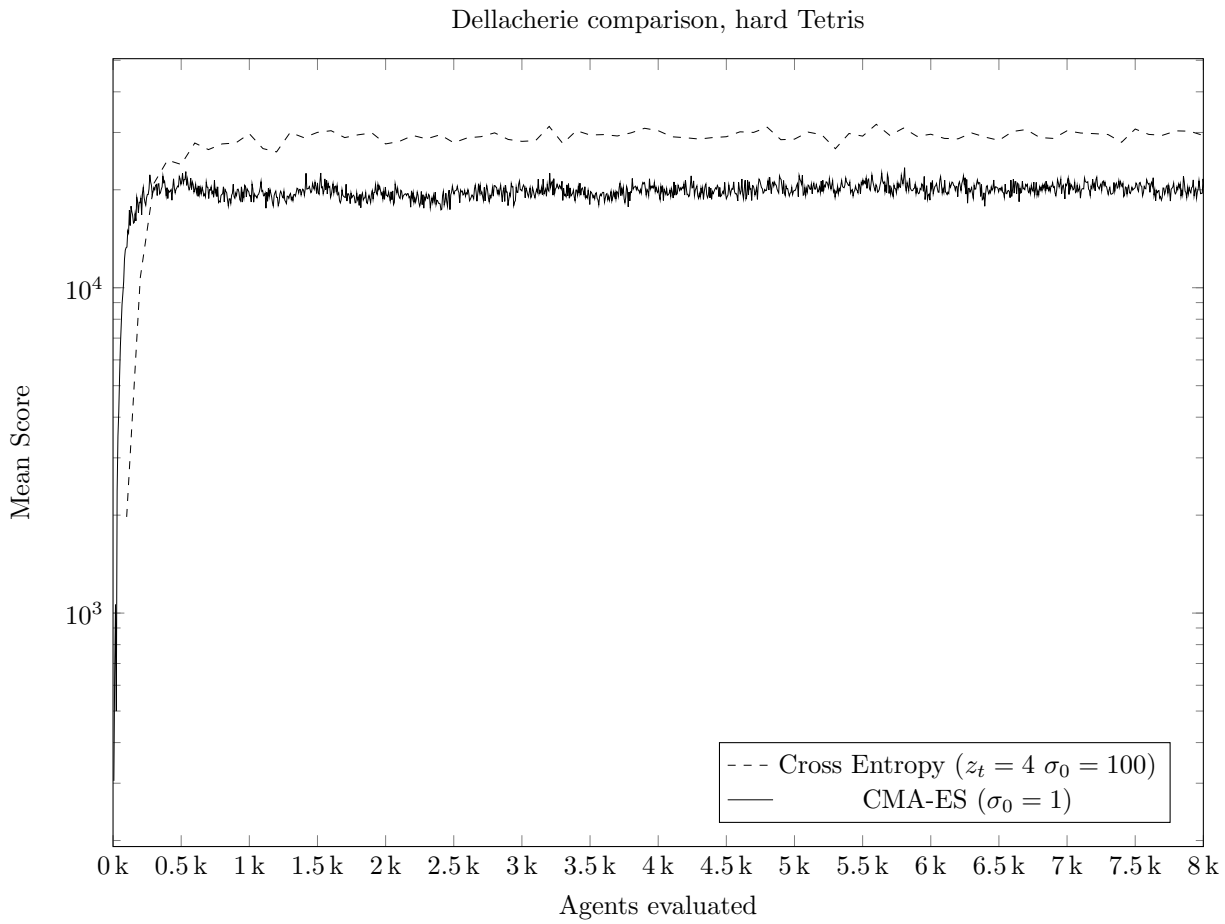


Figure 18: Comparison between CMA-ES and Cross Entropy using hard Tetris and the Dellacherie featureset

ADD FULL DATA GRAPHS AS APPENDIX.

Analysis and discussion

The experiment with different featuresets indicates that the behaviour is not heavily dependant on the featureset, as the development of the graphs closely resemble the initial comparison experiment. Furthermore, it appears that increasing the difficulty of the game simple shifts the score, but does not affect the development of the graphs. Therefore, we conclude that changing the featureset does not invalidate the comparison of the two optimization algorithms. In further experiments, in our configuration of CMA, the harder version of Tetris is applied in experiments used to optimize the CMA configurations.

THIS CONCLUSION DOESN'T MAKE KRONOLOGISK SENSE

4.6.4 Tuned comparison

DOASAS ASD

5 Conclusion

BLAS SLDFL SDFL

References

Bertsekas, D. and Tsitsiklis, J. (1996). Neuro-dynamic programming. *Athena Scientific*.

Boumaza, A. (2009). On the evolution of artificial tetris players. *HAL*, 14:1–14.

Fahey, C. (2003). Tetris ai. `http://www.colinfahey.com/tetris/`. Accessed: 2015-11-21.

Hansen, N. (2015). The CMA evolution strategy: A tutorial.

Igel, C., Heidrich-Meisner, V., and Glasmachers, T. (2008). Shark. *Journal of Machine Learning Research*, 9:993–996.

MDPTetris (2009). mdptetris. *None*.

Pieter-Tjerk De Boer, Dirk P. Kroese, S. M. and Rubenstein, R. Y. (2014). A tutorial on the cross-entropy method. *Springer Science*, 521:20–67.

Scherrer, C. T. B. (2009a). Building controllers for tetris. *International Computer Games Association Journal*, 32:3–11.

Scherrer, C. T. B. (2009b). Improvements on learning tetris with cross entropy. *International Computer Games Association Journal,*, 32:23–33.

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

Thiery, C. and Scherrer, B. (2009). Improvements on learning tetris with cross entropy. *International Computer Games Association Journal*, 32(1):23–33.

Appendices

A Experiments

This appendix contains all of the included experiments from which we have derived our results. Each of the plots, depending on the number of experiments, will depict the mean scores of the experiments or the experiments themselves.
With each experiment, a table detailing the parameter values will be included for others who would want to reproduce these experiments. Depending on the optimizer (weather it is Cross Entropy or CMA), the table will contain some custom rows, unique for the specific algorithm. Similar for all experiments is the 30 games played with the new mean of each generation to generate the learning curve.
The tables are formatted

Table 5: Overview of the two table formats

Optimizer	-
Number of Evaluations	-
Number of Learning Games	-
Population size	-
Parent size	-
Games per Agent	-
Tetris Type	-
Recombination Type	-
Initial Sigma	-

Optimizer	-
Number of Evaluations	-
Number of Learning Games	-
Population size	-
Parent size	-
Games per Agent	-
Tetris Type	-
Sigma	-
Noise Type	-
Noise	-

A.1 Verification of Cross Entropy

Using the same configuration as in "reference paper", we will reproduce the experiments to verify our implementation of Cross Entropy into the Shark Library.

Table 6: Cross Entropy - No noise

Optimizer	Cross Entropy
Number of Evaluations	8000
Population size	100
Parent size	10
Games per Agent	1
Tetris Type	Normal
Sigma	100
Noise Type	No noise
Noise	-

Table 7: Cross Entropy - Constant noise

Optimizer	Cross Entropy
Number of Evaluations	8000
Population size	100
Parent size	10
Games per Agent	1
Tetris Type	Normal
Sigma	100
Noise Type	Constant
Noise	4

Table 8: Cross Entropy - Linear decreasing noise

Optimizer	Cross Entropy
Number of Evaluations	8000
Population size	100
Parent size	10
Games per Agent	1
Tetris Type	Normal
Sigma	100
Noise Type	Linear decreasing
Noise	$max\left(5 - \frac{t}{10}, 0\right)$

A.2 Population and selection size

We want to investigate if there exists a better configuration than the 100/10 which was also previously used by other researchers.
In Cross Entropy for the Tetris problem, 10 % Parent selection is used as standard. However, we will also test 50 % Parent selection is a configuration from CMA which we will also test.
The general testing parameters are as follows

Table 9: General setup for Population/Parent size

Optimizer	Cross Entropy
Number of Evaluations	8000
Population size	See table 10
Parent size	See table 10
Games per Agent	1
Tetris Type	Normal
Sigma	100
Noise Type	Constant
Noise	4

with the following population/Parent size

Table 10: Population/Parent size

Population size	Parent size
10	1
10	5
22	2
22	11
50	5
50	25
100	10
100	50
200	20
200	100

INSERT PLOTS HERE.

quantile table and two graphs over the mean plots of 10 % and 50 % Parent size

A.3 Optimal settings for Cross Entropy - Games per agent

Experiment to determine the optimal number of games played per agent for best resulting score at lowest evaluation cost.

Table 11: General setup for Population/Parent size

Optimizer	Cross Entropy
Number of Evaluations	80000
Population size	See table 12
Parent size	See table 12
Games per Agent	See table 12
Tetris Type	Normal
Sigma	100
Noise Type	Constant
Noise	4

with the following population/Parent size and number of games played per agent

Population Size	Parent size	Games per Agent
10	10%	1/3/5/7/10
10	50%	1/3/5/7/10
22	10%	1/3/5/7/10
22	50%	1/3/5/7/10
50	10%	1/3/5/7/10
50	50%	1/3/5/7/10
100	10%	1/3/5/7/10
100	50%	1/3/5/7/10
200	10%	1/3/5/7/10
200	50%	1/3/5/7/10

Table 12: Games per agent CE experiment setup

INSERT PLOTS HERE.

quantile table and two graphs over the mean plots

A.4 Optimal settings for CMA - Initial Step-size

Experiments to find the best initial step-size for CMA.

Table 13: Overview of the two table formats

Optimizer	CMA
Number of Evaluations	8000
Number of Learning Games	30
Population size	13
Parent size	6
Games per Agent	1
Tetris Type	Normal
Recombination Type	Superlinear
Initial Sigma	See table 14

with the following initial sigma

σ_0	0.1	0.2	0.5	0.8	1.0

Table 14: Initial sigma configurations

INSERT PLOTS HERE.

σ_0	mean	Q1	Q2	Q3
0.1	50769.3	21301.1	54588.7	73972.4
0.2	42290.6	32180.2	42290.6	49337.4
0.5	53893.7	14211.1	66773.0	85816.7
0.8	37557.7	1422.8	15450.8	93719.4
1.0	49537.9	31369.8	49537.4	58454.6

Figure 19: Results of CMA-ES with adjusted initial step-size

A.5 Optimal settings for CMA - Experiment for finding the optimal settings

Experiments finding the best configuration of population and parent size with recombination type. The parent size is dependent on the recombination type, therefore we tested these parameter together.

Table 15: Overview of the two table formats

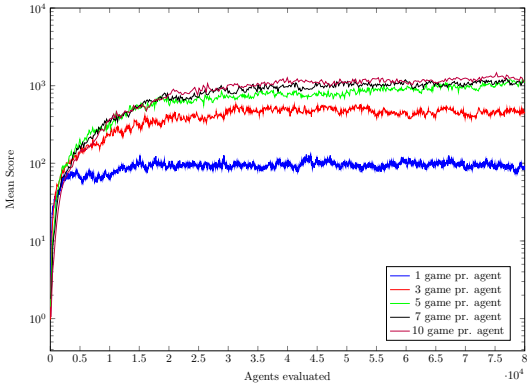
Optimizer	CMA
Number of Evaluations	80000
Number of Learning Games	30
Population size	See table 16
Parent size	See table 16
Games per Agent	See table 16
Tetris Type	Hard
Recombination Type	See table 16
Initial Sigma	1

Population Size	Parent size	Recombination Type	Games per Agent
12	1	EQUAL/LINEAR/SUPERLINEAR	1/3/5/7/10
12	3	EQUAL	1/3/5/7/10
12	6	LINEAR/SUPERLINEAR	1/3/5/7/10
22	2	EQUAL/LINEAR/SUPERLINEAR	1/3/5/7/10
22	5	EQUAL	1/3/5/7/10
22	11	LINEAR/SUPERLINEAR	1/3/5/7/10
50	5	EQUAL/LINEAR/SUPERLINEAR	1/3/5/7/10
50	12	EQUAL	1/3/5/7/10
50	25	LINEAR/SUPERLINEAR	1/3/5/7/10
100	10	EQUAL/LINEAR/SUPERLINEAR	1/3/5/7/10
100	25	EQUAL	1/3/5/7/10
100	50	LINEAR/SUPERLINEAR	1/3/5/7/10

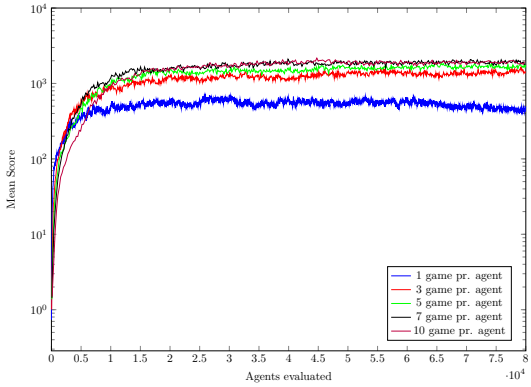
Table 16: Full experiments overview

WHEN ONLY TESTING 1 GAME PR AGENT, WE OBSERVE THAT AFTER 8000 AGENTS IT STARTS TO DECLINE IN SCORE, THUS FINDING WORSE PERFORMING AGENTS - SOME CONCLUSION THAT CMA WITH 1 GAME PR AGENT SHOULD BE RUN ONLY FOR SOME TIME AND NOT INFINITYLY AS SEEN ON EX. POPULATION 12, PARENT SIZE 6, LINEAR, IF PLAYED WITH DYNAMIC GAMES, MEANING 1 GAME UP TO 500 AGENT AND THEN A HIGHER GAMES PER AGENT AFTERWARDS, A FAST CONVERGENCE AND HIGH SCORE IS ACHIEVED.

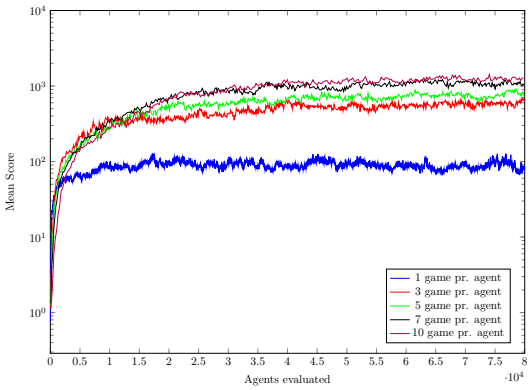
(a) Population size 12, Parent size 1, Meanscore of EQUAL recombination



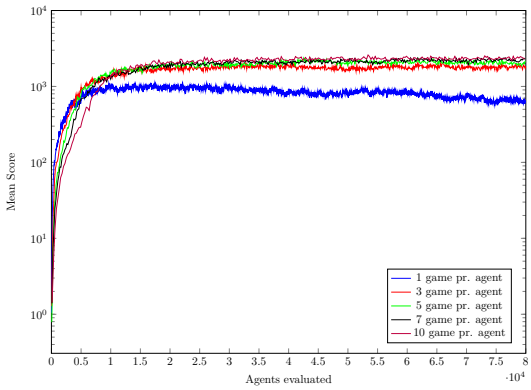
(b) Population size 12, Parent size 3, Meanscore of EQUAL recombination



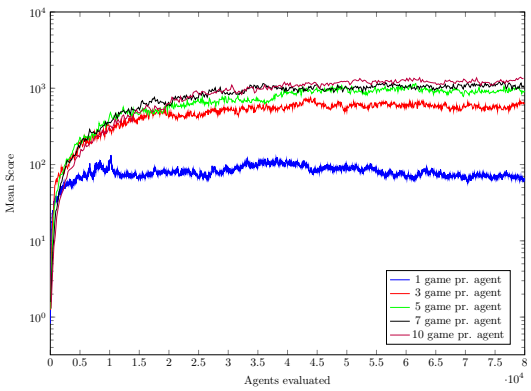
(c) Population size 12, Parent size 1, Meanscore of LINEAR recombination



(d) Population size 12, Parent size 6, Meanscore of LINEAR recombination



(e) Population size 12, Parent size 1, Meanscore of SUPERLINEAR recombination



(f) Population size 12, Parent size 6, Meanscore of SUPERLINEAR recombination

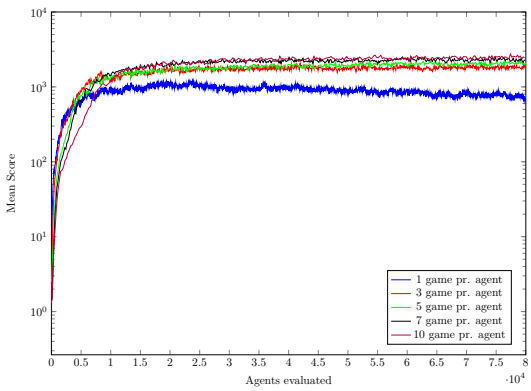
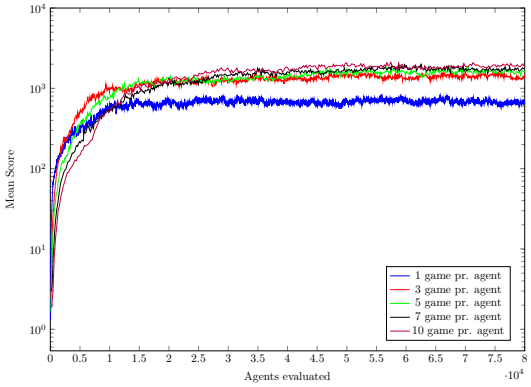
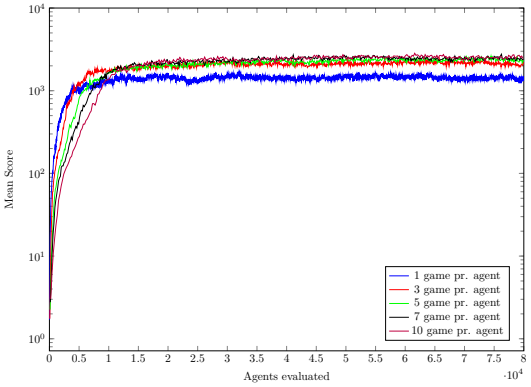


Figure 20: Mean results for population size 12 with variating recombination

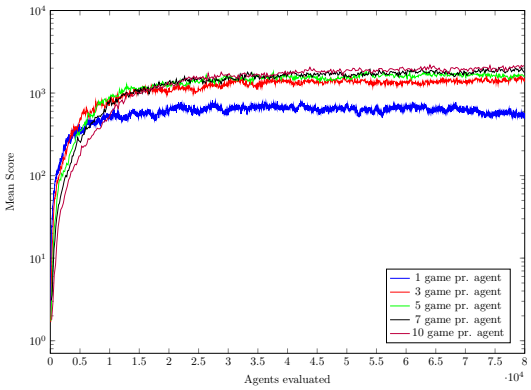
(a) Population size 22, Parent size 2, Meanscore of EQUAL recombination



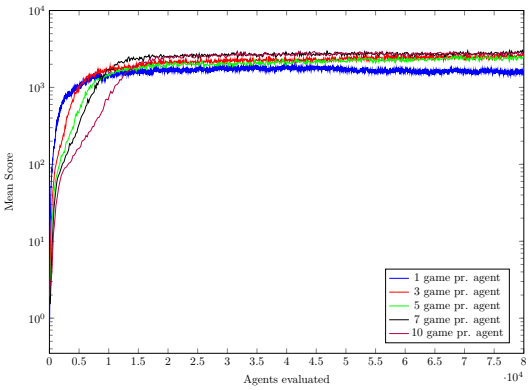
(b) Population size 22, Parent size 5, Meanscore of EQUAL recombination



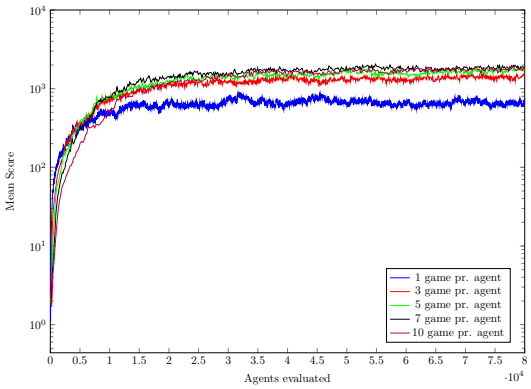
(c) Population size 22, Parent size 2, Meanscore of LINEAR recombination



(d) Population size 22, Parent size 11, Meanscore of LINEAR recombination



(e) Population size 22, Parent size 2, Meanscore of SUPERLINEAR recombination



(f) Population size 22, Parent size 11, Meanscore of SUPERLINEAR recombination

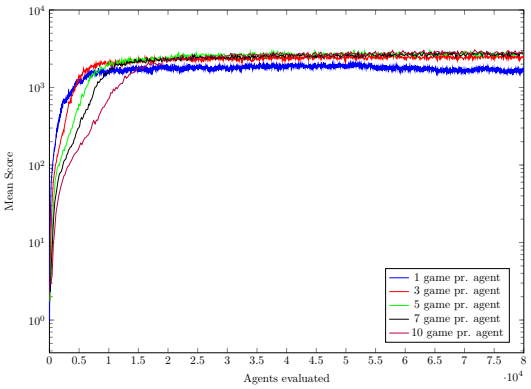
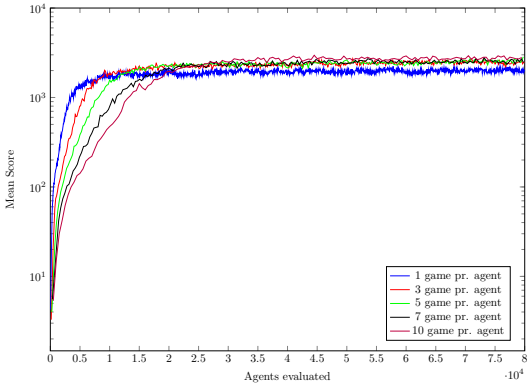
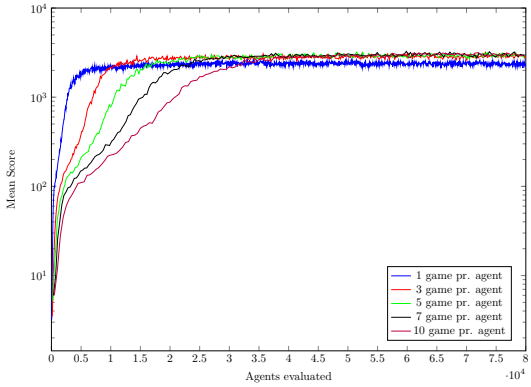


Figure 21: Mean results for population size 22 with variating recombination

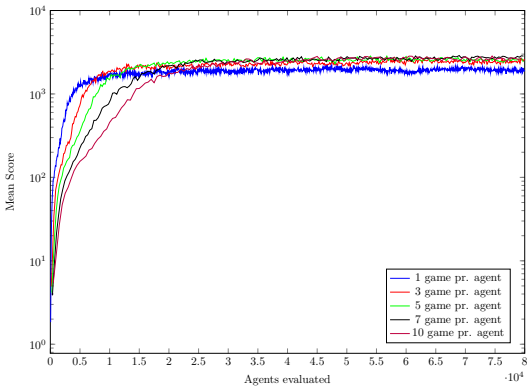
(a) Population size 50, Parent size 5, Meanscore of EQUAL recombination



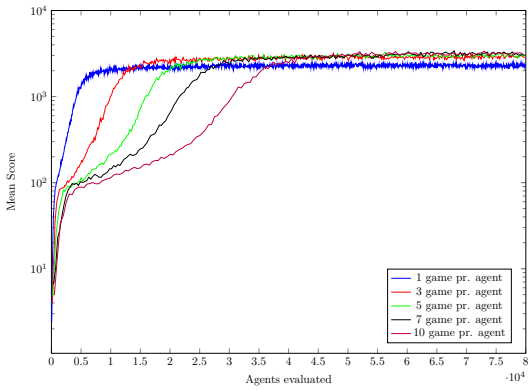
(b) Population size 50, Parent size 12, Meanscore of EQUAL recombination



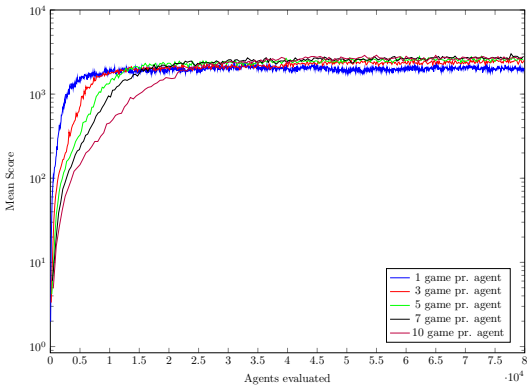
(c) Population size 50, Parent size 5, Meanscore of LINEAR recombination



(d) Population size 50, Parent size 25, Meanscore of LINEAR recombination



(e) Population size 50, Parent size 5, Meanscore of SUPERLINEAR recombination



(f) Population size 50, Parent size 25, Meanscore of SUPERLINEAR recombination

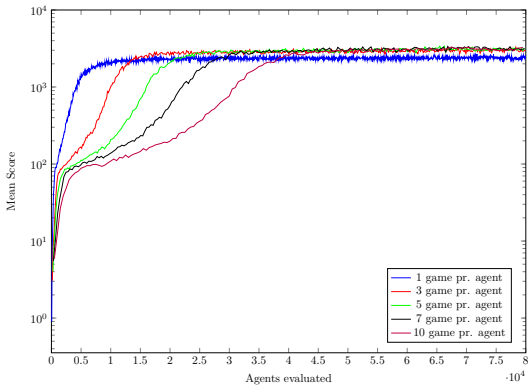
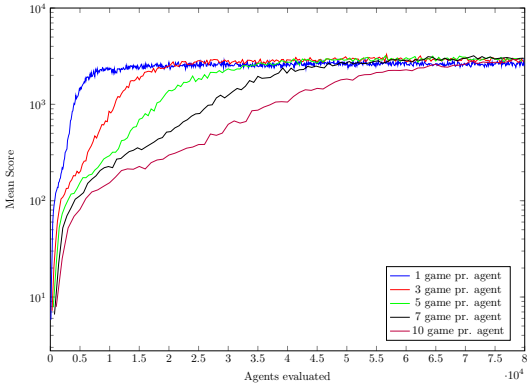
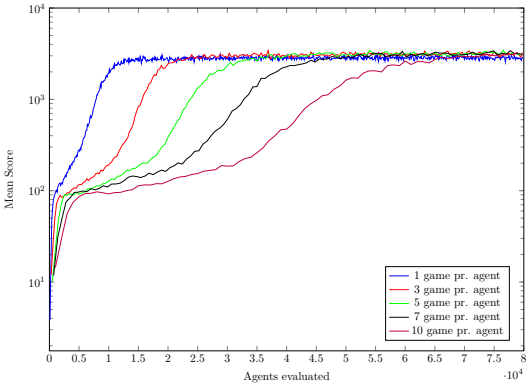


Figure 22: Mean results for population size 50 with variating recombination

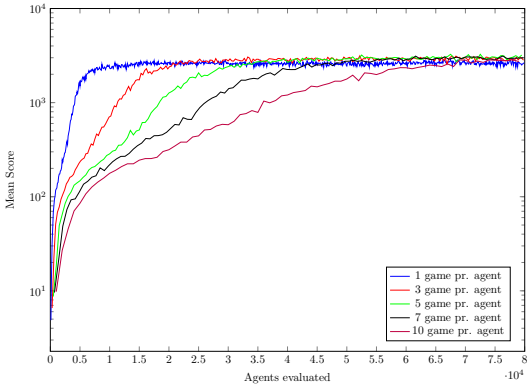
(a) Population size 100, Parent size 10, Meanscore of EQUAL recombination



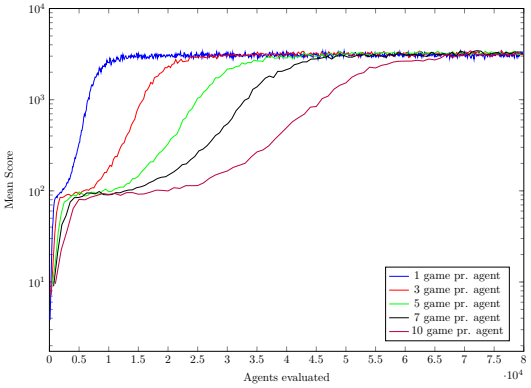
(b) Population size 100, Parent size 25, Meanscore of EQUAL recombination



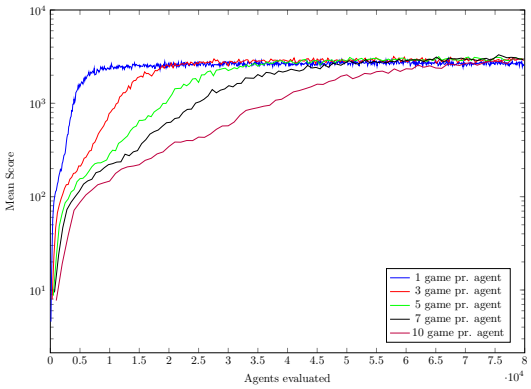
(c) Population size 100, Parent size 10, Meanscore of LINEAR recombination



(d) Population size 100, Parent size 50, Meanscore of LINEAR recombination



(e) Population size 100, Parent size 10, Meanscore of SUPERLINEAR recombination



(f) Population size 100, Parent size 50, Meanscore of SUPERLINEAR recombination

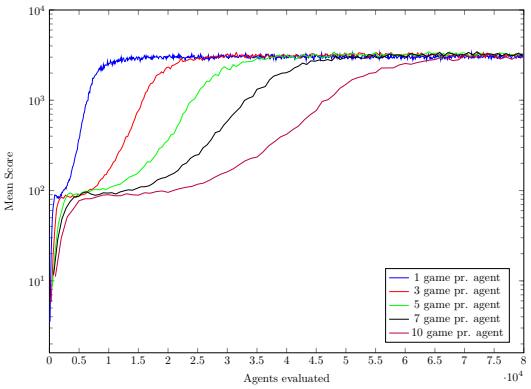


Figure 23: Mean results for population size 100 with variating recombination

Population	Parent	Recombination	Games per Agent	mean	Q1	Q2	Q3
12	1	EQUAL	1	86.998	53.557	63.417	111.313
12	1	EQUAL	3	458.616	256.637	349.033	576.899
12	1	EQUAL	5	1089.245	850.220	1104.535	1299.039
12	1	EQUAL	7	1081.248	918.049	1089.400	1248.619
12	1	EQUAL	10	1134.339	863.523	1061.050	1297.359
12	1	LINEAR	1	76.228	45.587	53.917	74.353
12	1	LINEAR	3	717.883	380.890	556.950	878.597
12	1	LINEAR	5	720.393	503.130	708.484	893.517
12	1	LINEAR	7	1043.547	732.367	969.217	1234.040
12	1	LINEAR	10	1276.041	1135.609	1196.120	1502.029
12	1	SUPERLINEAR	1	64.211	45.687	64.483	78.090
12	1	SUPERLINEAR	3	646.760	429.043	661.817	860.470
12	1	SUPERLINEAR	5	925.410	726.360	948.300	1158.021
12	1	SUPERLINEAR	7	1034.002	792.567	924.350	1269.131
12	1	SUPERLINEAR	10	1401.300	1194.430	1440.500	1569.012

Table 17: Population size 12, Parent size 1 - "Cross Entropy"-inspired configuration

Population	Parent	Recombination	Games per Agent	mean	Q1	Q2	Q3
12	3	EQUAL	1	461.762	266.047	408.034	639.013
12	3	EQUAL	3	1364.176	1199.310	1438.880	1579.560
12	3	EQUAL	5	1754.634	1464.239	1621.035	1945.010
12	3	EQUAL	7	1884.765	1614.661	1800.485	2156.972
12	3	EQUAL	10	1940.783	1566.721	1913.415	2210.251
12	6	LINEAR	1	626.867	447.440	538.150	724.913
12	6	LINEAR	3	1844.453	1568.829	1831.265	2227.479
12	6	LINEAR	5	2148.853	1846.459	2118.335	2588.010
12	6	LINEAR	7	2144.345	1860.340	2160.900	2428.348
12	6	LINEAR	10	2365.089	2072.719	2263.665	2637.732
12	6	SUPERLINEAR	1	669.005	506.817	648.167	836.377
12	6	SUPERLINEAR	3	1901.814	1572.260	1932.515	2151.189
12	6	SUPERLINEAR	5	2098.654	1965.472	2115.250	2453.069
12	6	SUPERLINEAR	7	2402.888	2055.329	2387.700	2635.870
12	6	SUPERLINEAR	10	2472.293	2194.049	2430.780	2709.040

Table 18: Population size 12, Parent size 3/6 - CMA configuration

Population	Parent	Recombination	Games per Agent	mean	Q1	Q2	Q3
22	2	EQUAL	1	684.551	429.890	597.534	877.150
22	2	EQUAL	3	1473.504	1197.290	1490.865	1713.091
22	2	EQUAL	5	1648.777	1425.150	1630.535	1826.372
22	2	EQUAL	7	1762.786	1521.832	1769.180	2033.799
22	2	EQUAL	10	1895.305	1630.891	1849.75	2043.442
22	2	LINEAR	1	513.271	398.023	524.850	589.467
22	2	LINEAR	3	1374.423	1082.859	1255.985	1561.119
22	2	LINEAR	5	1607.568	1513.389	1656.465	1791.069
22	2	LINEAR	7	1770.294	1538.900	1691.165	1881.590
22	2	LINEAR	10	2158.396	1966.351	2085.235	2162.541
22	2	SUPERLINEAR	1	698.811	359.367	622.467	937.423
22	2	SUPERLINEAR	3	1447.704	1281.119	1452.680	1640.468
22	2	SUPERLINEAR	5	1714.875	1362.931	1733.230	2023.590
22	2	SUPERLINEAR	7	1783.623	1603.389	1720.335	1937.749
22	2	SUPERLINEAR	10	1859.642	1636.870	1915.650	2208.830

Table 19: Population size 22, Parent size 2 - "Cross Entropy"-inspired configuration

Population	Parent	Recombination	Games per Agent	mean	Q1	Q2	Q3
22	5	EQUAL	1	1411.458	1207.060	1528.050	1707.590
22	5	EQUAL	3	2209.730	2129.521	2213.285	2471.751
22	5	EQUAL	5	2274.419	1933.249	2187.800	2570.399
22	5	EQUAL	7	2386.088	2112.841	2328.500	2573.261
22	5	EQUAL	10	2462.409	2277.180	2418.100	2600.411
22	11	LINEAR	1	1604.434	1181.603	1549.735	1678.111
22	11	LINEAR	3	2774.378	2460.992	2594.900	3030.310
22	11	LINEAR	5	2530.796	2382.050	2658.915	2990.851
22	11	LINEAR	7	2763.353	2597.450	2705.080	3000.242
22	11	LINEAR	10	2707.983	2146.209	2800.435	3273.401
22	11	SUPERLINEAR	1	1666.214	1447.430	1660.900	1856.908
22	11	SUPERLINEAR	3	2581.119	2306.02	2548.380	2759.428
22	11	SUPERLINEAR	5	2635.429	2409.620	2574.785	2863.849
22	11	SUPERLINEAR	7	2661.638	2411.232	2575.030	2945.200
22	11	SUPERLINEAR	10	2849.220	2500.231	2835.450	3143.121

Table 20: Population size 22, Parent size 5/11 - CMA configuration

Population	Parent	Recombination	Games per Agent	mean	Q1	Q2	Q3
50	5	EQUAL	1	1999.620	1789.150	1902.020	2161.948
50	5	EQUAL	3	2471.173	2178.651	2389.070	2660.220
50	5	EQUAL	5	2849.482	2459.460	2875.835	3276.190
50	5	EQUAL	7	2466.801	2355.349	2517.335	2644.751
50	5	EQUAL	10	2801.923	2479.088	2915.980	3081.018
50	5	LINEAR	1	1931.703	1659.412	1885.150	2130.479
50	5	LINEAR	3	2612.345	2380.759	2529.150	2796.729
50	5	LINEAR	5	2410.633	2067.111	2357.830	2675.572
50	5	LINEAR	7	2842.090	2497.952	2922.965	3136.509
50	5	LINEAR	10	2696.952	2518.290	2678.400	2923.010
50	5	SUPERLINEAR	1	1962.410	1715.568	1781.865	2009.488
50	5	SUPERLINEAR	3	2493.884	2220.200	2398.230	2762.609
50	5	SUPERLINEAR	5	2541.065	2268.881	2593.150	2974.558
50	5	SUPERLINEAR	7	2647.412	2448.010	2745.400	3058.870
50	5	SUPERLINEAR	10	2757.388	2357.062	2886.950	3187.282

Table 21: Population size 50, Parent size 5 - "Cross Entropy"-inspired configuration

Population	Parent	Recombination	Games per Agent	mean	Q1	Q2	Q3
50	12	EQUAL	1	2395.091	2095.030	2442.485	2778.160
50	12	EQUAL	3	2878.667	2550.399	2794.915	3102.639
50	12	EQUAL	5	2760.846	2509.000	2695.450	3006.988
50	12	EQUAL	7	2913.846	2627.310	2809.600	3128.439
50	12	EQUAL	10	2957.794	2636.478	2843.380	3222.420
50	25	LINEAR	1	2355.564	2262.991	2543.665	2752.431
50	25	LINEAR	3	2857.170	2464.251	2830.765	3209.929
50	25	LINEAR	5	3036.881	2807.050	3095.300	3416.750
50	25	LINEAR	7	3043.399	2821.049	3099.235	3239.032
50	25	LINEAR	10	3167.873	2941.590	3130.385	3415.520
50	25	SUPERLINEAR	1	2418.057	2352.851	2553.465	2718.759
50	25	SUPERLINEAR	3	2958.686	2715.579	2941.800	3173.361
50	25	SUPERLINEAR	5	3211.658	2889.689	3305.485	3694.480
50	25	SUPERLINEAR	7	3147.611	2799.510	3123.550	3457.999
50	25	SUPERLINEAR	10	3265.516	2928.771	3162.250	3389.121

Table 22: Population size 50, Parent size 12/25 - CMA configuration

Population	Parent	Recombination	Games per Agent	mean	Q1	Q2	Q3
100	10	EQUAL	1	2528.158	2380.122	2680.785	2782.850
100	10	EQUAL	3	2763.118	2488.878	2696.685	2962.190
100	10	EQUAL	5	2783.246	2399.949	2693.285	2900.391
100	10	EQUAL	7	3016.342	2835.049	2985.150	3292.910
100	10	EQUAL	10	2940.037	2703.859	2920.615	3216.410
100	10	LINEAR	1	2655.099	2409.162	2691.850	2961.091
100	10	LINEAR	3	2823.674	2526.560	2792.500	2974.549
100	10	LINEAR	5	3204.395	2928.050	3248.515	3371.008
100	10	LINEAR	7	2993.126	2639.021	3031.065	3277.642
100	10	LINEAR	10	2726.273	2402.502	2748.535	3009.198
100	10	SUPERLINEAR	1	2571.936	2211.051	2634.850	2811.190
100	10	SUPERLINEAR	3	2853.959	2552.749	2870.600	3005.292
100	10	SUPERLINEAR	5	3009.495	2816.660	2989.900	3189.310
100	10	SUPERLINEAR	7	2935.614	2540.342	2887.085	3232.508
100	10	SUPERLINEAR	10	2941.535	2705.290	2893.580	3153.510

Table 23: Population size 100, Parent size 10 - "Cross Entropy"-inspired configuration

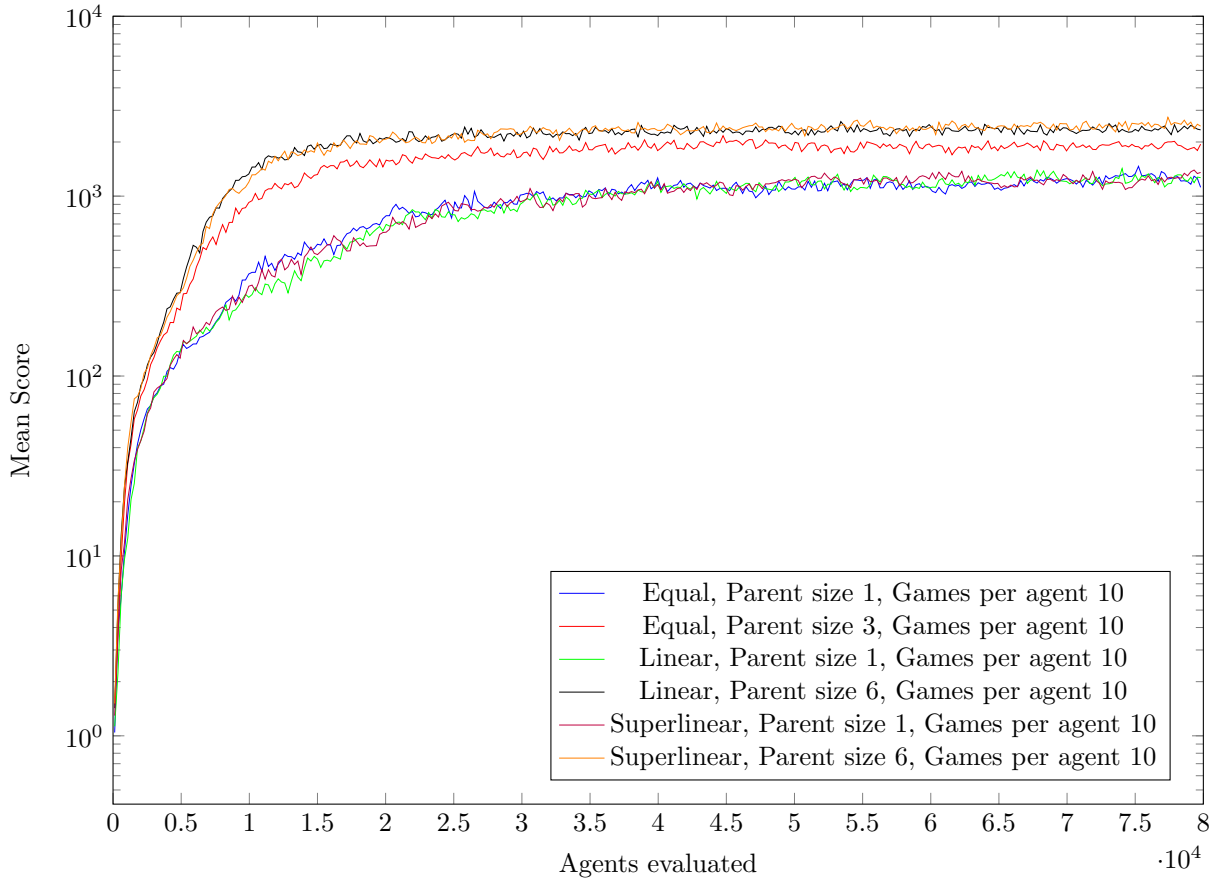
Population	Parent	Recombination	Games per Agent	mean	Q1	Q2	Q3
100	25	EQUAL	1	2885.131	2565.572	2886.735	3231.600
100	25	EQUAL	3	3244.544	2924.488	3270.885	3525.671
100	25	EQUAL	5	3065.143	2773.848	3032.830	3347.358
100	25	EQUAL	7	3125.085	2758.298	3054.950	3486.672
100	25	EQUAL	10	3258.533	2976.271	3144.750	3598.868
100	50	LINEAR	1	3099.651	2814.781	3019.830	3466.380
100	50	LINEAR	3	3125.019	2706.618	3087.815	3368.681
100	50	LINEAR	5	3171.761	2941.030	3138.515	3436.670
100	50	LINEAR	7	3322.076	3160.098	3289.370	3537.850
100	50	LINEAR	10	3320.153	3040.611	3291.335	3633.491
100	50	SUPERLINEAR	1	3162.245	2815.099	3093.050	3482.748
100	50	SUPERLINEAR	3	3188.324	2781.382	3258.400	3413.710
100	50	SUPERLINEAR	5	3304.119	3004.140	3226.235	3679.359
100	50	SUPERLINEAR	7	3072.331	2682.300	2964.850	3403.132
100	50	SUPERLINEAR	10	3106.917	2914.349	3162.250	3324.970

Table 24: Population size 100, Parent size 25/50 - CMA configuration

Population	Parent	Recombination	Games per Agent	mean	Q1	Q2	Q3
12	1	EQUAL	10	1134.339	863.523	1061.050	1297.359
12	1	LINEAR	10	1276.041	1135.609	1196.120	1502.029
12	1	SUPERLINEAR	10	1401.300	1194.430	1440.500	1569.012
12	3	EQUAL	10	1940.783	1566.721	1913.415	2210.251
12	6	LINEAR	10	2365.089	2072.719	2263.665	2637.732
12	6	SUPERLINEAR	10	2472.293	2194.049	2430.780	2709.040

Table 25: Best performing configurations for population size 12

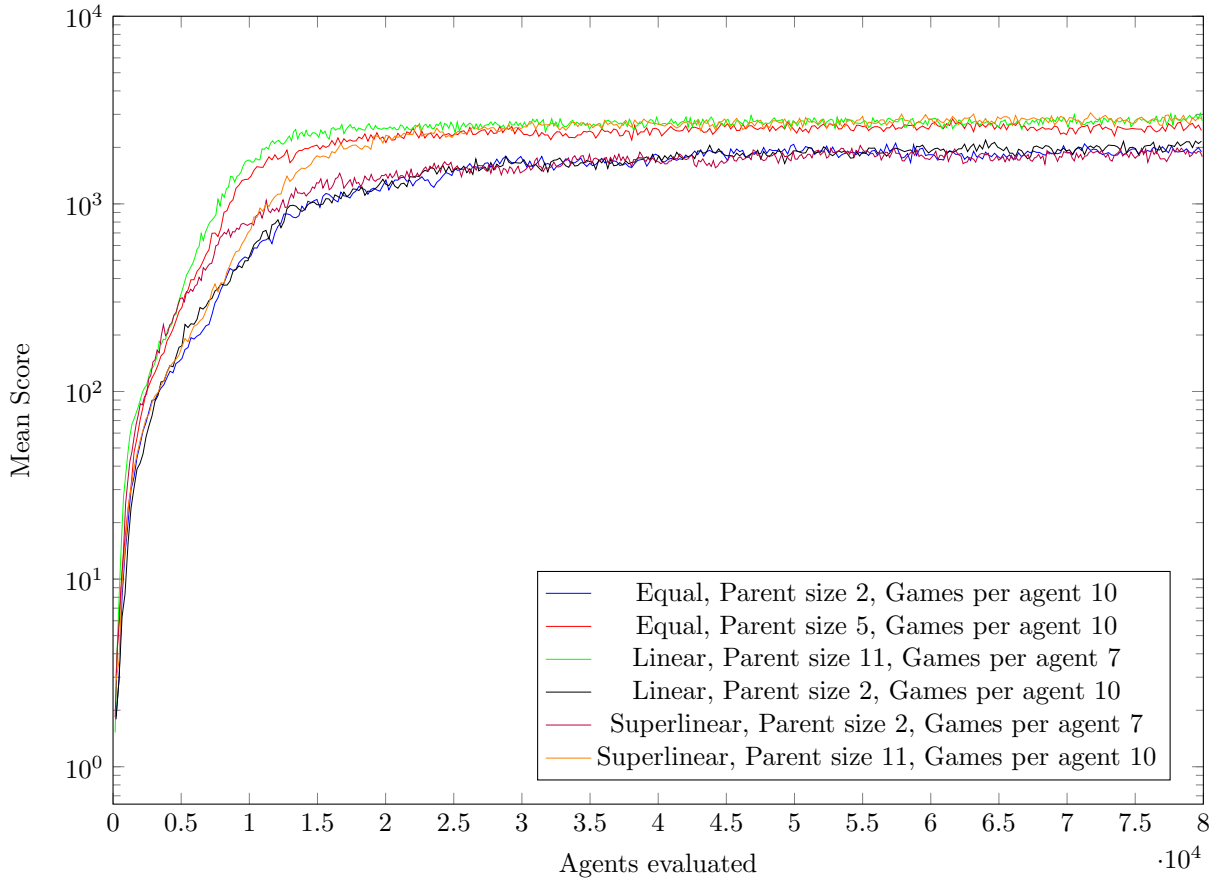
Figure 24: Best performing configurations for population size 12



Population	Parent	Recombination	Games per Agent	mean	Q1	Q2	Q3
22	2	EQUAL	10	1895.305	1630.891	1849.75	2043.442
22	2	LINEAR	10	2158.396	1966.351	2085.235	2162.541
22	2	SUPERLINEAR	7	1783.623	1603.389	1720.335	1937.749
22	5	EQUAL	10	2462.409	2277.180	2418.100	2600.411
22	11	LINEAR	7	2763.353	2597.450	2705.080	3000.242
22	11	SUPERLINEAR	10	2849.220	2500.231	2835.450	3143.121

Table 26: Best performing configurations for population size 22

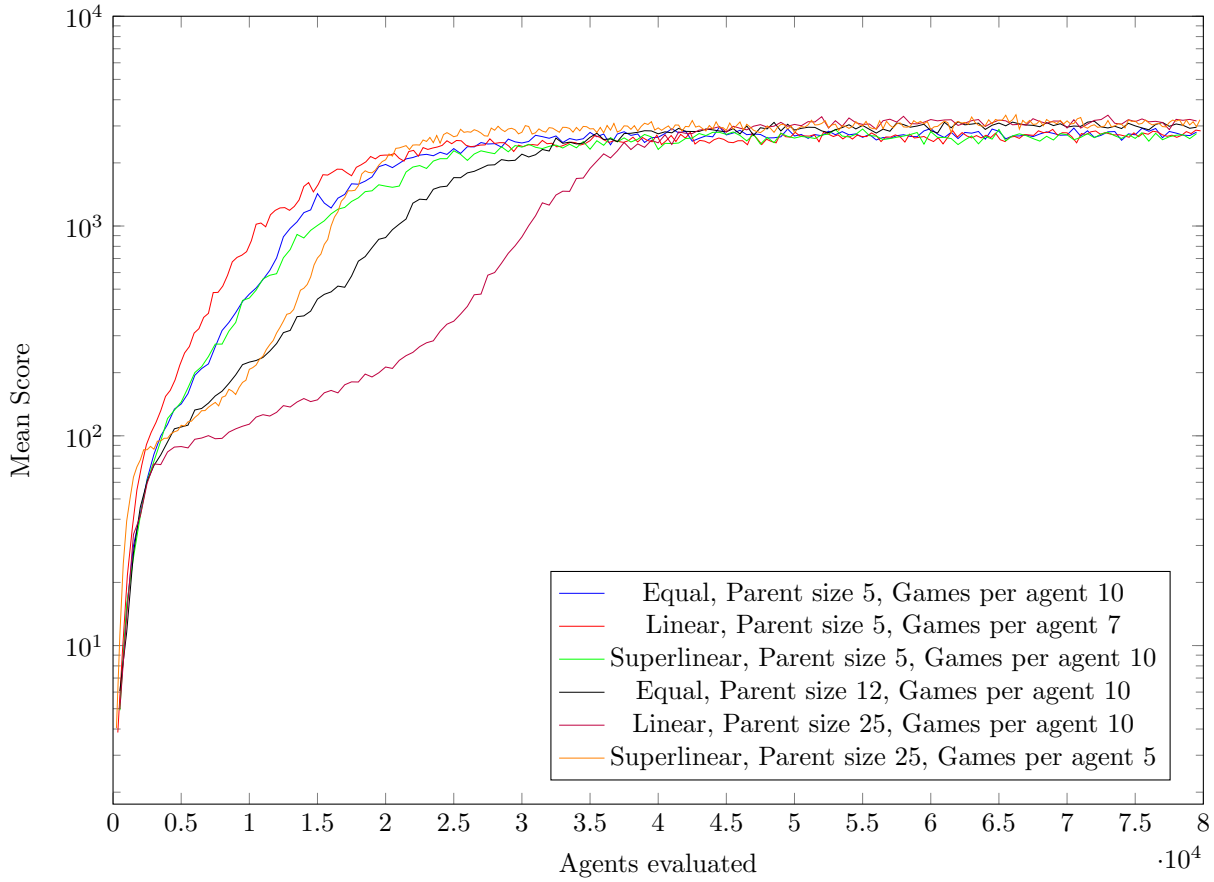
Figure 25: Best performing configurations for population size 22



Population	Parent	Recombination	Games per Agent	mean	Q1	Q2	Q3
50	5	EQUAL	10	2801.923	2479.088	2915.980	3081.018
50	5	LINEAR	7	2842.090	2497.952	2922.965	3136.509
50	5	SUPERLINEAR	10	2757.388	2357.062	2886.950	3187.282
50	12	EQUAL	10	2957.794	2636.478	2843.380	3222.420
50	25	LINEAR	10	3167.873	2941.590	3130.385	3415.520
50	25	SUPERLINEAR	5	3211.658	2889.689	3305.485	3694.480

Table 27: Best performing configurations for population size 50

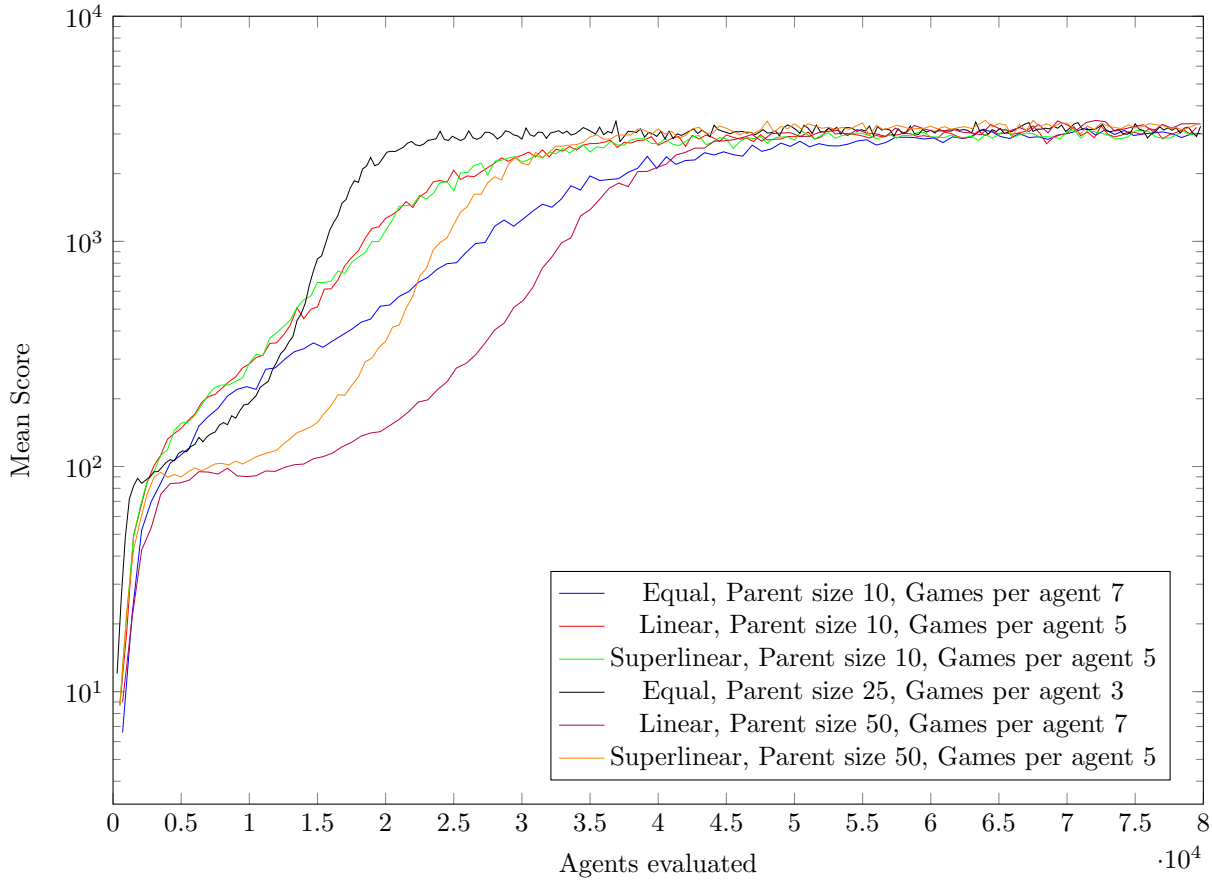
Figure 26: Best performing configurations for population size 50



Population	Parent	Recombination	Games per Agent	mean	Q1	Q2	Q3
100	10	EQUAL	7	3016.342	2835.049	2985.150	3292.910
100	10	LINEAR	5	3204.395	2928.050	3248.515	3371.008
100	10	SUPERLINEAR	5	3009.495	2816.660	2989.900	3189.310
100	25	EQUAL	3	3244.544	2924.488	3270.885	3525.671
100	50	LINEAR	7	3322.076	3160.098	3289.370	3537.850
100	50	SUPERLINEAR	5	3304.119	3004.140	3226.235	3679.359

Table 28: Best performing configurations for population size 100

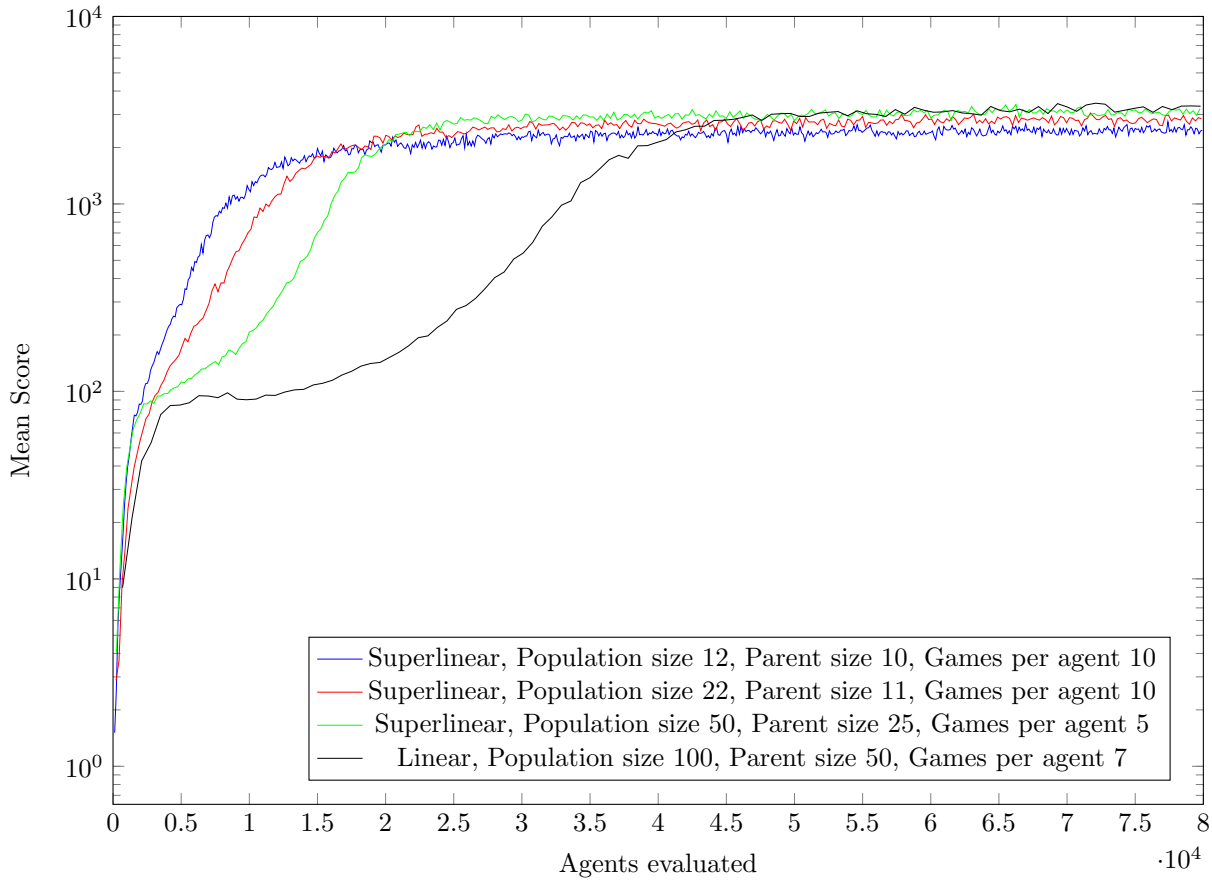
Figure 27: Best performing configurations for population size 100



Population	Parent	Recombination	Games per Agent	mean	Q1	Q2	Q3
12	6	SUPERLINEAR	10	2472.293	2194.049	2430.780	2709.040
22	11	SUPERLINEAR	10	2849.220	2500.231	2835.450	3143.121
50	25	SUPERLINEAR	5	3211.658	2889.689	3305.485	3694.480
100	50	LINEAR	7	3322.076	3160.098	3289.370	3537.850

Table 29: Best performing configurations of all population sizes

Figure 28: Best performing configurations for population size 100



A.6 Comparison - Initial comparison

Comparison between the verified Cross Entropy and Shark (reference to shark) stock setting CMA.

Table 30: Shark stock CMA and Verified Cross Entropy

Optimizer	CMA	Optimizer	Cross Entropy
Number of Evaluations	8000	Number of Evaluations	8000
Number of Learning Games	30	Number of Learning Games	30
Population size	13	Population size	100
Parent size	6	Parent size	10
Games per Agent	1	Games per Agent	1
Tetris Type	Normal	Tetris Type	Normal
Recombination Type	Superlinear	Sigma	100
Initial Sigma	(what is stock)	Noise Type	Constant
		Noise	4

INSERT PLOTS HERE.

A.7 Comparison - Comparison of featuresets

Experiments to test Bertsekas and Dellacherie featuresets in regards to achieved score. This will tell if a different featureset affects the algorithms performance. If the behaviour is the same, then the Tetris complexity problem

Table 31: Shark stock CMA and Verified Cross Entropy

Optimizer	CMA
Number of Evaluations	8000
Number of Learning Games	30
Population size	13
Parent size	6
Games per Agent	1
Tetris Type	Hard
Recombination Type	Superlinear
Initial Sigma	(what is stock)

Optimizer	Cross Entropy
Number of Evaluations	8000
Number of Learning Games	30
Population size	100
Parent size	10
Games per Agent	1
Tetris Type	Hard
Sigma	100
Noise Type	Constant
Noise	4

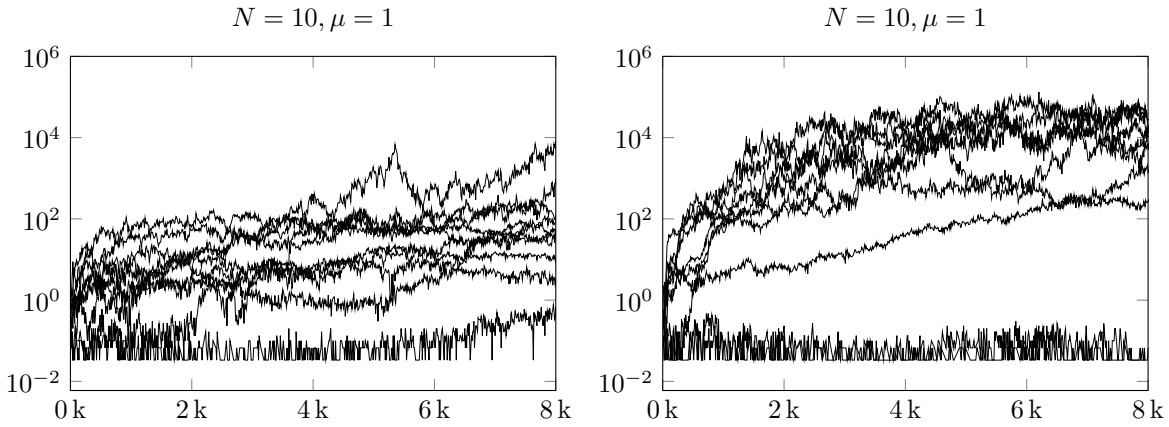
Used on the bertsekas and Dellacherie featuresets
INSERT PLOTS HERE.

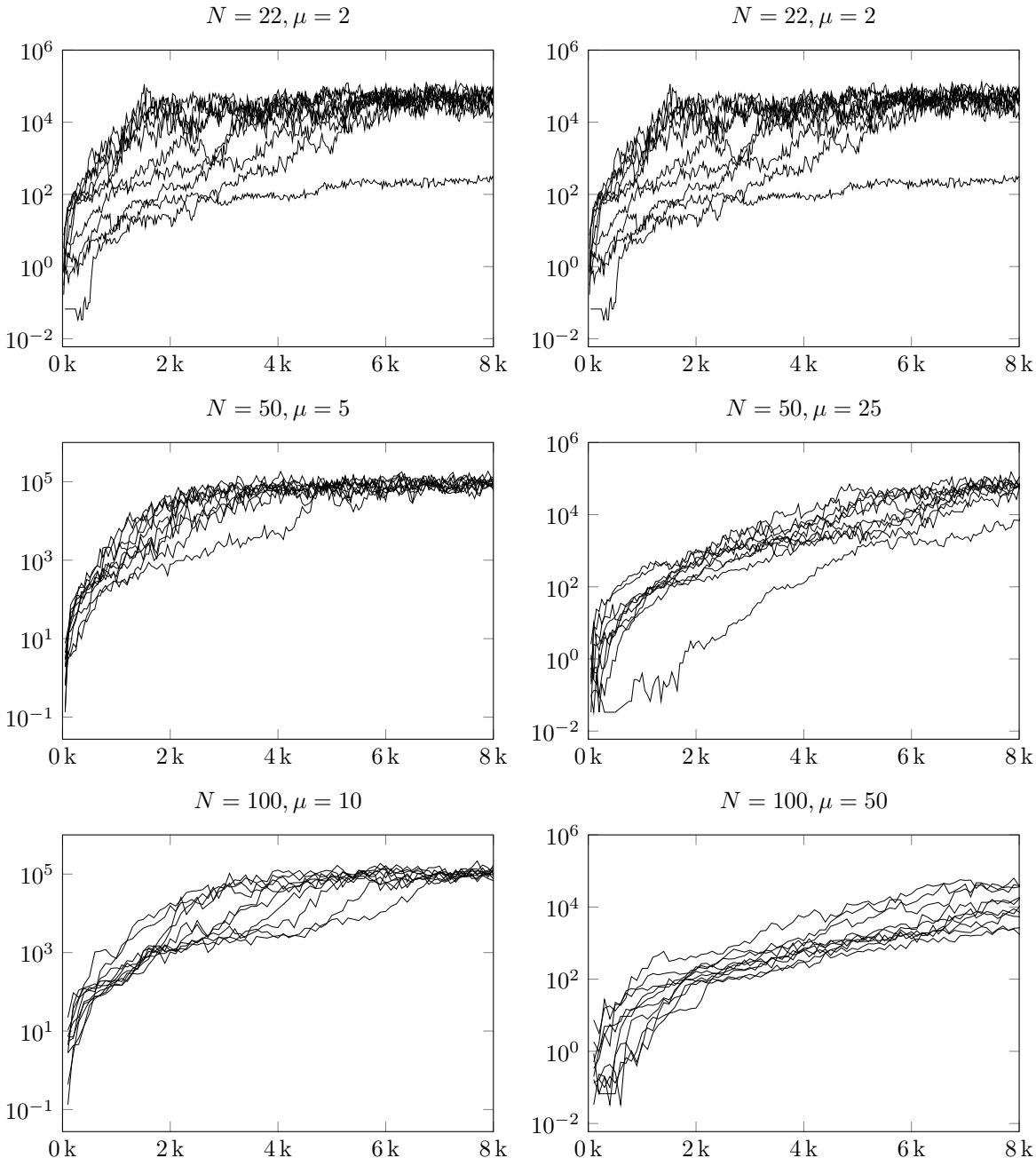
B CMA initial step-size

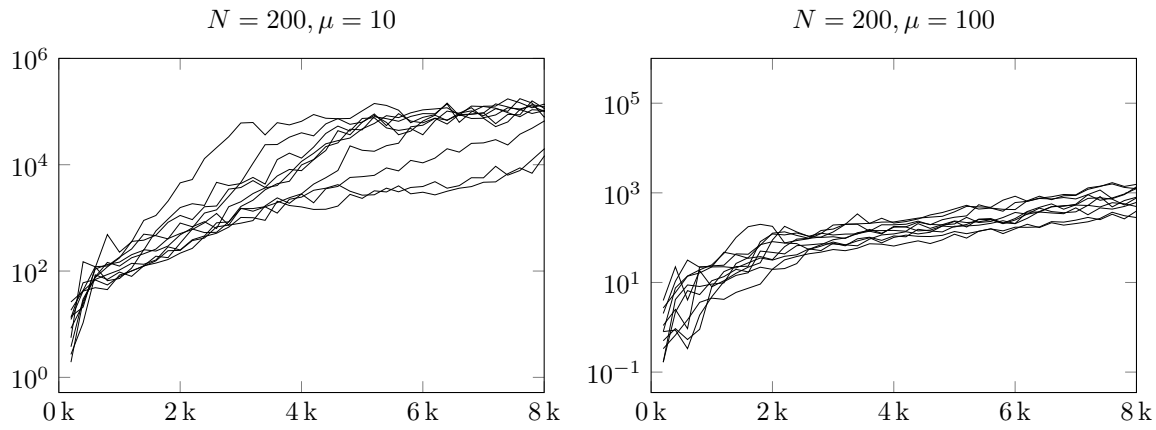
Shows the graps of the configuration test, with initial sigma settings.
- INSERT IN ABOVE SECTION

C Cross Entropy configuration settings

The plots from the configuration of cross entropy. All plots shows the numbers of games played along the x-axis and the mean score of the centroid agent along the y-axis.




$N = 22, \mu = 2$
$N = 22, \mu = 2$
$N = 50, \mu = 5$
$N = 50, \mu = 25$
$N = 100, \mu = 10$
$N = 100, \mu = 50$
10^{6}
10^{4}
10^{2}
10^{0}
10^{-2}
10^{5}
10^{3}
10^{1}
10^{-1}
0 k
2 k
4 k
6 k
8 k



- INSERT IN ABOVE SECTION

D Cross Entropy Implementation - Shark library

D.1 CrossEntropyMethod.h

```
//===========================================================================
/*!
 *
 * \brief       Implements the Cross Entropy Algorithm.
 *
 * Christophe Thiery, Bruno Scherrer. Improvements on Learning Tetris with
    Cross Entropy.
 * International Computer Games Association Journal, ICGA, 2009, 32. <inria
    -00418930>
 *
 *
 * \author      Jens Holm, Mathias Petraeus and Mark Wulff
 * \date        January 2016
 *
 * \par Copyright 1995-2015 Shark Development Team
 *
 * <BR><HR>
 * This file is part of Shark.
 * <http://image.diku.dk/shark/>
 *
 * Shark is free software: you can redistribute it and/or modify
 * it under the terms of the GNU Lesser General Public License as published
 * by the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * Shark is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public License
 * along with Shark.  If not, see <http://www.gnu.org/licenses/>.
 *
 */
//===========================================================================


#ifndef SHARK_ALGORITHMS_DIRECT_SEARCH_CROSSENTROPY_H
#define SHARK_ALGORITHMS_DIRECT_SEARCH_CROSSENTROPY_H

#include <shark/Core/DLLSupport.h>
#include <shark/Algorithms/AbstractSingleObjectiveOptimizer.h>
#include <shark/Statistics/Distributions/MultiVariateNormalDistribution.h>
#include <shark/Algorithms/DirectSearch/Individual.h>

#include <boost/shared_ptr.hpp>

namespace shark {

        /**
         * \brief Implements the Cross Entropy Method.
         *
         *  This class implements the noisy cross entropy method
```

```
 *  as descibed in the following article.
 *
 *  Christophe Thiery, Bruno Scherrer. Improvements on Learning Tetris
     with Cross Entropy.
 *  International Computer Games Association Journal, ICGA, 2009, 32.
    <inria-00418930>
 *
 *  The algorithm aims to minimize an objective function through
    stochastic search.
 *  It works iteratively until a certain stopping criteria is met. At
    each
 *  iteration, it samples a number of vectors from a Gaussian
    distribution
 *  and evaluates each of these against the supplied objective
    function.
 *  Based on the return value from the objective function, a subset of
     the
 *  the best ranked vectors are chosen to update the search parameters
 *  of the next generation.
 *
 *  The mean of the Gaussian distribution is set to the centroid of
    the best ranked
 *  vectors, and the variance is set to the variance of the best
    ranked
 *  vectors in each individual dimension.
 *
 */
class CrossEntropyMethod : public AbstractSingleObjectiveOptimizer<
    RealVector >
{
public:

        /**
         * \brief Interface class for noise type.
         */
        class INoiseType {
        public:
                virtual double noiseValue (int t) const { return 0.0;
                    };
                virtual std::string name() const { return std::string(
                    "Default noise of 0"); }
        };

        /**
         * \brief Smart pointer for noise type.
         */
        typedef boost::shared_ptr<INoiseType> StrongNoisePtr;

        /**
         * \brief Constant noise term z_t = noise.
         */
        class ConstantNoise : public INoiseType {
        public:
                ConstantNoise ( double noise ) { m_noise = noise; };
                virtual double noiseValue (int t) const { return std::
                    max(m_noise, 0.0); }
                virtual std::string name() const {
```

```
                std::stringstream ss;
                ss << "z(t) = " << m_noise;
                return std::string(ss.str());
        }
private:
        double m_noise;
};

/**
 * \brief Linear noise term z_t = a + t / b.
 */
class LinearNoise : public INoiseType {
public:
        LinearNoise ( double a, double b ) { m_a = a; m_b = b;
             };
        virtual double noiseValue (int t) const { return std::
            max(m_a + (t * m_b), 0.0); }
        virtual std::string name() const {
                std::stringstream ss;
                std::string sign = (m_b < 0.0 ? " - " : " + ")
                    ;
                ss << "z(t) = " << m_a << sign << "t * " <<
                    std::abs(m_b);
                return std::string(ss.str());
        }
private:
        double m_a, m_b;
};


/**
* \brief Default c'tor.
*/
SHARK_EXPORT_SYMBOL CrossEntropyMethod();

/// \brief From INameable: return the class name.
std::string name() const
{ return "Cross Entropy Method"; }

/**
* \brief Sets default value for Population size.
*/
SHARK_EXPORT_SYMBOL static unsigned suggestPopulationSize(  )
    ;

/**
* \brief Calculates selection size for the supplied population
     size.
*/
SHARK_EXPORT_SYMBOL static unsigned suggestSelectionSize(
    unsigned int populationSize ) ;

void read( InArchive & archive );
void write( OutArchive & archive ) const;

using AbstractSingleObjectiveOptimizer<RealVector >::init;
/**
```

```
        * \brief Initializes the algorithm for the supplied objective
            function and the initial mean p.
        */
        SHARK_EXPORT_SYMBOL void init( ObjectiveFunctionType& function
            , SearchPointType const& p);

        /**
         * \brief Inits the Cross Entropy, only with the objective
             function
         */
        SHARK_EXPORT_SYMBOL void init( ObjectiveFunctionType& function
             );

        /**
        * \brief Initializes the algorithm for the supplied objective
            function.
        */
        SHARK_EXPORT_SYMBOL void init(
                ObjectiveFunctionType& function,
                SearchPointType const& initialSearchPoint,
                unsigned int populationSize,
                unsigned int selectionSize,
                RealVector initialSigma
        );

        /**
        * \brief Executes one iteration of the algorithm.
        */
        SHARK_EXPORT_SYMBOL void step(ObjectiveFunctionType const&
            function);

        /** \brief Access the current variance. */
        RealVector const& variance() const {
                return m_variance;
        }

        /** \brief Set the variance to a vector */
        void setVariance(RealVector variance) {
                assert(variance.size() == m_variance.size());
                m_variance = variance;
        }

        /** \brief Set all variance values */
        void setVariance(double variance){
                for(int i = 0; i < m_variance.size(); i++)
                        m_variance(i) = variance;
        }

        /** \brief Access the current population mean. */
        RealVector const& mean() const {
                return m_mean;
        }

        /**
         * \brief Returns the size of the parent population.
         */
        unsigned int selectionSize() const {
```

```
                return m_selectionSize;
        }

        /**
         * \brief Returns a mutable reference to the size of the
             parent population.
         */
        unsigned int& selectionSize(){
                return m_selectionSize;
        }

        /**
         * \brief Returns a immutable reference to the size of the
             population.
         */
        unsigned int populationSize()const{
                return m_populationSize;
        }

        /**
         * \brief Returns a mutable reference to the size of the
             population.
         */
        unsigned int & populationSize(){
                return m_populationSize;
        }

        /**
         * \brief Set the noise type from a raw pointer.
         */
        void setNoiseType( INoiseType* noiseType ) {
                m_noise.reset();
                m_noise = boost::shared_ptr<INoiseType> (noiseType);
        }

        /**
         * \brief Get an immutable reference to Noise Type.
         */
        const INoiseType &getNoiseType( ) const {
                return *m_noise.get();
        }


protected:
        /**
        * \brief Updates the strategy parameters based on the supplied
             parent population.
        */
        SHARK_EXPORT_SYMBOL void updateStrategyParameters( const std::
            vector<Individual<RealVector, double, RealVector> > &
            parents ) ;

        std::size_t m_numberOfVariables; ///< Stores the
            dimensionality of the search space.
        unsigned int m_selectionSize; ///< Number of vectors chosen
            when updating distribution parameters.
        unsigned int m_populationSize; ///< Number of vectors sampled
```

```
                    in a generation.

                RealVector m_variance; ///< Variance for sample parameters.

                StrongNoisePtr m_noise; ///< Noise type to apply in the update
                     of distribution parameters.

                RealVector m_mean; ///< The mean of the population.

                unsigned m_counter; ///< Counter for generations.

                Normal< Rng::rng_type > m_distribution; ///< Normal
                    distribution.

        };
}

#endif
```

D.2 CrossEntropyMethod.cpp

```
/*!
 *
 * \brief       Implements the Cross Entropy Algorithm.
 *
 * Christophe Thiery, Bruno Scherrer. Improvements on Learning Tetris with
    Cross Entropy.
 * International Computer Games Association Journal, ICGA, 2009, 32. <inria
    -00418930>
 *
 *
 * \author      Jens Holm, Mathias Petraeus and Mark Wulff
 * \date        January 2016
 *
 * \par Copyright 1995-2015 Shark Development Team
 *
 * <BR><HR>
 * This file is part of Shark.
 * <http://image.diku.dk/shark/>
 *
 * Shark is free software: you can redistribute it and/or modify
 * it under the terms of the GNU Lesser General Public License as published
 * by the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * Shark is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public License
 * along with Shark.  If not, see <http://www.gnu.org/licenses/>.
 *
 */
 #define SHARK_COMPILE_DLL

#include <shark/Core/Exception.h>
#include <shark/Algorithms/DirectSearch/Operators/Evaluation/
    PenalizingEvaluator.h>
#include <shark/Algorithms/DirectSearch/FitnessExtractor.h>
#include <shark/Algorithms/DirectSearch/Operators/Selection/ElitistSelection.h
    >
#include <shark/Algorithms/DirectSearch/CrossEntropyMethod.h>

using namespace shark;

/**
* \brief Suggest a population size of 100.
*/
unsigned CrossEntropyMethod::suggestPopulationSize(  ) {
        /*
         * Most papers suggests a population size of 100, thus
         * simply choose 100.
         */
        return 100;
}
```

```
/**
* \brief Calculates Selection Size for the supplied Population Size.
*/
unsigned int CrossEntropyMethod::suggestSelectionSize( unsigned int
    populationSize ) {
        /*
         * Most papers says 10% of the population size is used for
         * the new generation, thus, just take 10% of the population size.
         */
        return (unsigned int) (populationSize / 10.0);
}

CrossEntropyMethod::CrossEntropyMethod()
: m_variance( 0 )
, m_counter( 0 )
, m_distribution( Normal< Rng::rng_type >( Rng::globalRng, 0, 1.0 ) )
, m_noise (boost::shared_ptr<INoiseType> (new ConstantNoise(0.0)))
{
        m_features |= REQUIRES_VALUE;
}

void CrossEntropyMethod::read( InArchive & archive ) {
        archive >> m_numberOfVariables;
        archive >> m_selectionSize;
        archive >> m_populationSize;

        archive >> m_variance;

        archive >> m_mean;

        archive >> m_counter;
}

void CrossEntropyMethod::write( OutArchive & archive ) const {
        archive << m_numberOfVariables;
        archive << m_selectionSize;
        archive << m_populationSize;

        archive << m_variance;

        archive << m_mean;

        archive << m_counter;
}


void CrossEntropyMethod::init( ObjectiveFunctionType & function,
    SearchPointType const& p) {

        unsigned int populationSize = CrossEntropyMethod::
            suggestPopulationSize( );
        unsigned int selectionSize = CrossEntropyMethod::suggestSelectionSize(
             populationSize );
        RealVector initialVariance(p.size());

        // Most papers set the variance to 100 by default.
```

```
        for(int i = 0; i < p.size(); i++)
        {
                initialVariance(i) = 100;
        }
        init( function,
                p,
                populationSize,
                selectionSize,
                initialVariance
        );
}

/**
* \brief Initializes the algorithm for the supplied objective function.
*/
void CrossEntropyMethod::init(
        ObjectiveFunctionType& function,
        SearchPointType const& initialSearchPoint,
        unsigned int populationSize,
        unsigned int selectionSize,
        RealVector initialVariance
) {
        checkFeatures(function);
        function.init();

        m_numberOfVariables = function.numberOfVariables();
        m_populationSize = populationSize;
        m_selectionSize = static_cast<unsigned int>(::floor(selectionSize));
        m_variance = initialVariance;

        m_mean.resize( m_numberOfVariables );
        m_mean.clear();

        m_mean = initialSearchPoint;
        m_best.point = initialSearchPoint;
        m_best.value = function(initialSearchPoint);
        m_counter = 0;

}

/**
* \brief Updates the strategy parameters based on the supplied parent
    population.
*/
void CrossEntropyMethod::updateStrategyParameters( const std::vector<
    Individual<RealVector, double> > & parents ) {

        /* Calculate the centroid of the parents */
        RealVector m(m_numberOfVariables);
        for (int i = 0; i < m_numberOfVariables; i++)
        {
                m(i) = 0;
                for (int j = 0; j < parents.size(); j++)
                {
                        m(i) += parents[j].searchPoint()(i);
                }
                m(i) /= double(parents.size());
```

```
        }


        // mean update
        m_mean = m;

        // Variance update
        size_t nParents = parents.size();
        double normalizationFactor = 1.0 / double(nParents);

        for (int j = 0; j < m_numberOfVariables; j++) {
                double innerSum = 0.0;
                for (int i = 0; i < parents.size(); i++) {
                        double diff = parents[i].searchPoint()(j) - m(j);
                        innerSum += diff * diff;
                }
                innerSum *= normalizationFactor;

                // Apply noise
                m_variance(j) = innerSum + m_noise->noiseValue(m_counter);
        }

}

/**
* \brief Executes one iteration of the algorithm.
*/
void CrossEntropyMethod::step(ObjectiveFunctionType const& function){

        std::vector< Individual<RealVector, double> > offspring(
            m_populationSize );

        PenalizingEvaluator penalizingEvaluator;
        for( unsigned int i = 0; i < offspring.size(); i++ ) {
                RealVector sample(m_numberOfVariables);
                for (int j = 0; j < m_numberOfVariables; j++)
                {
                        sample(j) = m_distribution(m_mean(j), m_variance(j));
                            // N (0, 100)
                }
                offspring[i].searchPoint() = sample;
        }

        penalizingEvaluator( function, offspring.begin(), offspring.end() );

        // Selection
        std::vector< Individual<RealVector, double> > parents( m_selectionSize
             );
        ElitistSelection<FitnessExtractor> selection;
        selection(offspring.begin(),offspring.end(),parents.begin(), parents.
            end());
        // Strategy parameter update
        m_counter++; // increase generation counter
        updateStrategyParameters( parents );

        m_best.point= parents[ 0 ].searchPoint();
        m_best.value= parents[ 0 ].unpenalizedFitness();
```

```
}

void CrossEntropyMethod::init(ObjectiveFunctionType& function ){
        if(!(function.features() & ObjectiveFunctionType::
            CAN_PROPOSE_STARTING_POINT))
                throw SHARKEXCEPTION( "[AbstractSingleObjectiveOptimizer::init
                    ] Objective function does not propose a starting point");
        CrossEntropyMethod::init(function,function.proposeStartingPoint());
}
```