CROSS-ENTROPY FOR MONTE-CARLO TREE SEARCH

Guillaume M.J-B. Chaslot[1] *Mark H.M. Winands*[1] *István Szita*[1] *H. Jaap van den Herik*[2]

Maastricht, The Netherlands Tilburg, The Netherlands

ABSTRACT

Recently, Monte-Carlo Tree Search (MCTS) has become a popular approach for intelligent play in games. Amongst others, it is successfully used in most state-of-the-art Go programs. To improve the playing strength of these Go programs any further, many parameters dealing with MCTS should be fine-tuned.

In this paper, we propose to apply the Cross-Entropy Method (CEM) for this task. The method is comparable to Estimation-of-Distribution Algorithms (EDAs), a new area of evolutionary computation. We tested CEM by tuning various types of parameters in our Go program MANGO. The experiments were performed in matches against the open-source program GNU GO. They revealed that a program with the CEM-tuned parameters played better than without. Moreover, MANGO plus CEM outperformed the regular MANGO for various time settings and board sizes. From the results we may conclude that parameter tuning by CEM genuinely improved the playing strength of MANGO, for various time settings. This result may be generalized to other game engines using MCTS.

1. INTRODUCTION

Most game engines have a large number of parameters that are crucial for their performance. Tuning these parameters by hand may be a hard and time-consuming task. Although it is possible to make educated guesses for some parameters, for other parameters it is beyond imagination. Here, a learning method can be used to find the "right" values for these parameters (Sutton and Barto, 1998; Beal and Smith, 2000).

Using learning methods for tuning the parameters in order to increase the playing strength of a game program is difficult. The problem is that the fitness function[3] is rarely available analytically. Therefore, learning methods that rely on the availability of an analytic expression for the gradient cannot be used. However, there are several ways to tune parameters despite the lack of an analytic gradient. An important class of such algorithms is represented by temporal-difference (TD) methods that have been used successfully in tuning evaluation-function parameters in Backgammon, Chess, Checkers, and LOA (Tesauro, 1995; Baxter, Tridgell, and Weaver, 1998; Schaeffer, Hlynka, and Jussila, 2001; Winands *et al.*, 2002). Obviously, any general-purpose gradient-free learning method can be used for parameter tuning in games. Just to mention two other examples, Björnsson and Marsland (2003) applied successfully an algorithm similar to Finite-Difference Stochastic Approximations (FDSA) to tune the search-extension parameters of CRAFTY. Kocsis, Szepesvári, and Winands (2006) investigated the use of RSPSA (Resilient Simultaneous Perturbation Stochastic Approximation), a stochastic hill-climbing algorithm, for the game of Poker and for the game of LOA.

In this paper we investigate the use of the *Cross-Entropy Method* (CEM) (Rubinstein, 1999) for parameter tuning in general, i.e., for any game engine. The method is in some sense similar to the Estimation-of-Distribution Algorithms (EDAs) (see Muehlenbein, 1997), which is a new area of evolutionary computation. Similar to EDA,

[1] Games and AI Group, Department of Knowledge Engineering, Faculty of Humanities and Sciences, Maastricht University, Maastricht, The Netherlands. Email: {g.chaslot,m.winands}@micc.unimaas.nl, szityu@gmail.com

[2] Tilburg centre for Creative Computing, Tilburg University, Tilburg, The Netherlands. Email: H.J.vdnHerik@uvt.nl

[3] The fitness function is associated to a learning task and determines how good a solution is; for instance, in games it may be the percentage of won games.

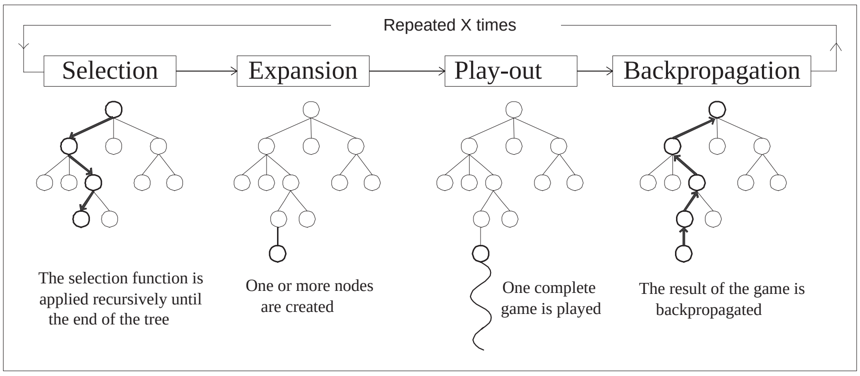
Repeated X times
Selection
Expansion
Play-out
Backpropagation
The selection function is applied recursively until the end of the tree
One or more nodes are created
One complete game is played
The result of the game is backpropagated

Figure 1: Scheme of Monte-Carlo Tree Search.

CEM maintains a *probability distribution* over possible solutions. From this distribution, solution candidates are drawn. This is essentially nothing else but random guessing. However, by using the idea of *Distribution Focusing*, CEM is turned into a highly effective learning method (Rubinstein, 1999).

As test domain we have chosen the Go-playing program MANGO (as designed by the first author). It uses Monte-Carlo Tree Search (Coulom, 2007b; Kocsis and Szepesvári, 2006; Coquelin and Munos, 2007), a best-first search algorithm that has advanced the field of Computer Go considerably (Gelly and Wang, 2007; van der Werf, 2007). MCTS is a relatively new method and when compared to the traditional $\alpha\beta$ (Knuth and Moore, 1975), it is less understood. Parameter tuning for MCTS is a challenging task, making it an appropriate test domain for CEM.

The paper is organized as follows. In Section 2 we describe MCTS and its implementation in MANGO. In Section 3 we explain CEM. We empirically evaluate CEM in combination with MCTS in Section 4. Section 5 provides our conclusions and describes future research.

2. MONTE-CARLO TREE SEARCH

Monte-Carlo Tree Search (MCTS) (Kocsis and Szepesvári, 2006; Coulom, 2007b) is a best-first search method that does not require a positional evaluation function. It is based on randomized explorations of the search space. Using the results of previous explorations, the algorithm gradually grows a game tree in memory, and successively becomes better at accurately estimating the values of the most promising moves. In this section, we describe our implementation of MCTS and highlight the parameters that have to be tuned. The basic structure of MCTS is given in Subsection 2.1. Two important MCTS enhancements, which have parameters that can be optimized by automatic learning methods, are described in Subsection 2.2.

2.1 Structure of MCTS

MCTS consists of four strategic phases: (1) *selection*, (2) *expansion*, (3) *play-out*, and (4) *backpropagation* (shown in Figure 1). They are repeated as long as there is time left. Below we discuss how each of these phases is implemented in MANGO.

2.1.1 Selection

Selection chooses a child to be examined based on previous gained information. It controls the balance between exploitation and exploration. Exploitation is the task to select the move that leads to the best results so far. Exploration deals with moves a priori less promising that still have to be examined, owing to the uncertainty of the evaluation.

In MANGO, we use the selection strategy UCT (Upper Confidence Bound applied to Trees) (Kocsis and Szepesvári, 2006). This strategy is applied in many other MCTS programs. UCT works as follows. Let I be the set of nodes reachable from the current node p. UCT selects a child k of node p that satisfies formula 1:

$$k \in \arg\max_{i \in I} \left(v_i + C \times \sqrt{\frac{\ln n_p}{n_i}} \right) \tag{1}$$

where v_i is the value of the node i, n_i is the visit count of node i, and n_p is the visit count of node p. C is the exploration coefficient, which will be tuned using the Cross-Entropy Method (CEM).

2.1.2 Expansion

Expansion is the strategic phase in which it is decided whether nodes will be added to the tree. We apply a simple rule: one node is added per play-out (Coulom, 2007b). The new node L (a leaf node) corresponds to the first position encountered during the play-out that was not already visited.

Here we remark that as soon as a node has been visited a given number of times T, all the children of this node are added. The coefficient T is also fine-tuned by using CEM (see Section 3).

2.1.3 Play-out

A play-out consists of playing moves in self-play until the end of the game. A play-out might consist of playing plain random moves or – better – pseudo-random moves chosen according to a *play-out strategy*. The play-out strategy of MANGO uses (1) a capture-escape value, (2) a pattern value, and (3) a proximity factor. We discuss them below. Let $\mathcal{M}$ be the set of all possible moves for a given position. Each move $j \in \mathcal{M}$ is given an urgency $U_j \geq 1$. The play-out strategy selects one move from $\mathcal{M}$. The probability of each move to be selected is $p_j = \frac{U_j}{\sum_{k \in \mathcal{M}} U_k}$. The urgency U is the sum of two values: the capture-escape value and the pattern value, which is multiplied by a factor P_{md} indicating the proximity. So, $U_j = (V_{ce} + V_p) \times P_{md}$. We explain the three concepts below.

1. *Capture-escape value.* The value V_{ce} is given to moves capturing stones or escaping captures. It equals a coefficient P_c × the number of captured stones plus coefficient P_e × the number of stones escaping to be captured. Using a capture value was first proposed by Bouzy (2005), and later improved successively by Coulom (2007b) and Cazenave (2007).
2. *Pattern value.* For each possible 3×3 pattern, the value of the central move has been learned by a dedicated algorithm described in Bouzy and Chaslot (2006). The pattern values range from 1 to 2433. The pattern values are raised to the power of a certain exponent P_p, which will be tuned with CEM (see Section 3). When P_p is set to a small value, all patterns will be selected with a nearly-uniform probability. When P_p is set to a large value, only the best patterns will be played in the play-out phase.
3. *Proximity.* Moves within a Manhattan distance of 1 from the previous move have their urgency multiplied by a proximity factor P_{md}. This idea is adapted from the strategy developed by Gelly and Wang (Gelly *et al.*, 2006).

2.1.4 Backpropagation

Backpropagation is the procedure that propagates the *result* of a certain play-out k back from the leaf node L, through the previously traversed nodes, all the way up to the root. The result is scored positively $(R_k = +1)$ if the game is won, and negatively $(R_k = -1)$ if the game is lost. Draws lead to a result $R_k = 0$. A *backpropagation strategy* is applied to the *value* v_L of a node. Here, it is computed by taking the average of the results of all games played through this node (Kocsis and Szepesvári, 2006), i.e., $v_L = (\sum_k R_k)/n_L$.

2.2 MCTS Enhancements

In MANGO we use the two enhancements, *progressive bias* and *progressive widening*, which were introduced for MCTS recently (Chaslot *et al.*, 2008; Coulom, 2007a). Both enhancements significantly increased MANGO's playing strength. This happened after tuning the parameters by trial and error.

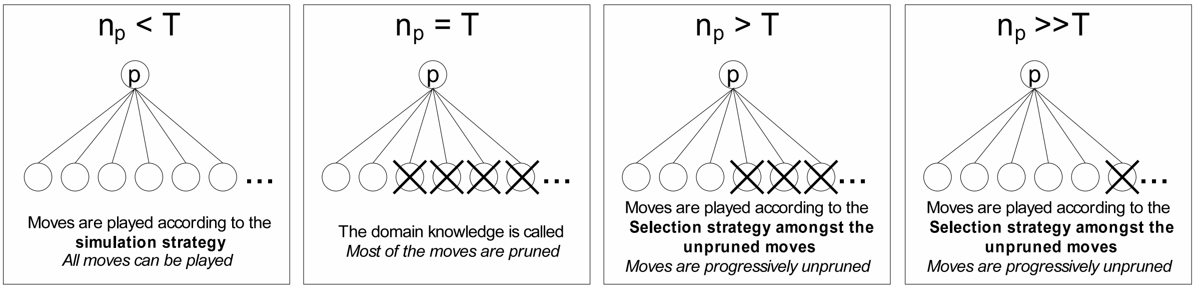
$n_p < T$
p
Moves are played according to the
simulation strategy
All moves can be played
$n_p = T$
p
The domain knowledge is called
Most of the moves are pruned
$n_p > T$
p
Moves are played according to the
Selection strategy amongst the unpruned moves
Moves are progressively unpruned
$n_p >> T$
p
Moves are played according to the
Selection strategy amongst the unpruned moves
Moves are progressively unpruned

Figure 2: Progressive widening in MANGO.

2.2.1 Progressive bias

Progressive bias modifies the UCT formula in such a way that it favours moves that are regarded as "good" by some heuristic knowledge. In MANGO, the heuristic knowledge takes into account capturing, escaping captures, and large patterns. More details can be found in Chaslot *et al.* (2008). We modified the UCT selection mechanism in the following way. Instead of selecting the move which satisfies formula 1, we select the node k that satisfies formula 2.

$$k \in \arg\max_{i \in I} \left(v_i + C \times \sqrt{\frac{\ln n_p}{n_i}} + \frac{PB_f \times H_i^{PB_e}}{n_i + 1} \right) \tag{2}$$

Here H_i represents the heuristic knowledge. The coefficient PB_f and the PB_e will again be tuned by using the CEM. PB_f stands for the *progressive-bias factor* and PB_e for the *progressive-bias exponent* (see Chaslot *et al.*, 2008).

2.2.2 Progressive widening

Progressive widening consists of (1) reducing the branching factor artificially when the selection function is applied, and (2) increasing it progressively as more time becomes available. When the number of games that visits a node p (n_p) equals a threshold T, progressive widening "prunes" most of the children. Initially, only the k_{init} children with the highest heuristic values in the sequence are not pruned. k_{init} was set to 5 in MANGO. Next, the children of a node i are progressively "unpruned". In MANGO, it happens as follows. The k^{th} child node is unpruned when the number of play-outs in i surpasses $A \times B^{k-k_{init}}$ play-outs. The scheme is shown in Figure 2. Obviously, A, B, and k_{init} will be tuned by using CEM.

3. THE CROSS-ENTROPY METHOD

In this section we explain the Cross-Entropy Method. First, we give an informal description (Subsection 3.1). Subsequently, we clarify the method in detail (Subsection 3.2). Finally, we discuss the normalization of parameters in Subsection 3.3.

3.1 Informal Description of the Cross-Entropy Method

The *Cross-Entropy Method* (CEM) (Rubinstein, 1999) is a population-based learning algorithm, where members of the population are sampled from a parameterized probability distribution. In each generation, the parameters of the distribution are updated so that its *cross-entropy distance* from a desired distribution is minimized.

CEM aims to find the (approximate) optimal solution $\mathbf{x}^*$ for a learning task described in the following form

$$\mathbf{x}^* := \arg\max_{\mathbf{x}} f(\mathbf{x}).$$

We remark that $\mathbf{x}^*$ is a vector containing all parameters to be tuned. Moreover, f is a fitness function (which determines how good a solution is; for instance in games, it is the percentage of won games), and $\mathbf{x} \in X$ where X is some (possibly high-dimensional) parameter space. Most traditional learning algorithms maintain a single candidate solution $\mathbf{x}(t)$ in each time step. In contrast, CEM maintains a *distribution* over possible solutions (similar to evolutionary methods). From that distribution, solution candidates are drawn at random. This is essentially *random guessing*, but by the idea of *Distribution Focusing* it is turned into a highly effective learning method. We explain both concepts below.

3.1.1 The Power of Random Guessing

Random guessing is a quite simple 'learning' method: we draw many samples from a distribution g belonging to a family of parameterized distributions $\mathcal{G}$ (e.g., Gaussian, Binomial, Bernoulli, etc.), then select the best sample as an estimation of the optimum. In the extreme case of drawing infinitely many samples, random guessing finds the global optimum.

The efficiency of random guessing depends largely on the distribution g from which the samples are drawn. For example, if g is sharply peaked in the neighbourhood of the optimal solution $\mathbf{x}^*$, then a few samples may be sufficient to obtain a good estimate. In contrast, if the distribution is sharply peaked around a vector $\mathbf{x}$, which is far away from the optimal solution $\mathbf{x}^*$, a large number of samples is needed to obtain a good estimate of the global optimum.

3.1.2 Distribution Focussing

We can improve the efficiency of random guessing by the idea of *Distribution Focussing*. After drawing a moderate amount of samples from distribution g, we may not be able to give an acceptable approximation of $\mathbf{x}^*$, but we may still obtain a *better sampling distribution*. The basic idea of CEM is that it selects the best samples, and modifies g so that it becomes more peaked around the best samples. Distribution Focussing is the central idea of CEM (Rubinstein, 1999).

Let us consider an example, where $\mathbf{x}$ is an m-dimensional vector and g is a Gaussian distribution for each coordinate. Assume that we have drawn 1000 samples and selected the 10 best. If the i^{th} coordinate of the best-scoring samples has an average of μ_i, then we may hope that the i^{th} coordinate of $\mathbf{x}^*$ is also close to μ_i, so we may shift the distribution's centre towards μ_i. In the next subsection, we describe the update rule of CEM in a more formal way.

3.2 Formal Description of the Cross-Entropy Method

In this subsection we will choose g from a family of parameterized distributions (e.g., Gaussian, Binomial, Bernoulli, etc.), denoted by $\mathcal{G}$, and describe an algorithm that iteratively improves the parameters of this distribution g.

Let N be the number of samples to be drawn, and let the samples $\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(N)}$ be drawn independently from distribution g. For each $\gamma \in \mathbb{R}$, the set of high-valued samples,

$$\hat{L}_\gamma := \{\mathbf{x}^{(i)} \mid f(\mathbf{x}^{(i)}) \geq \gamma, 1 \leq i \leq N\},$$

provides an approximation to the level set

$$L_\gamma := \{\mathbf{x} \mid f(\mathbf{x}) \geq \gamma\}.$$

Let U_γ be the uniform distribution over the level set L_γ. For large values of γ, this distribution will peak around $\mathbf{x}^*$, so it would be suitable for random sampling. This approximation procedure raises two potential problems, which are discussed below. The first problem is solved by *elite samples* and the second problem by the *cross-entropy distance*.

3.2.1 Elite Samples

The first problem is that for (too) large γ values $\hat{L}_\gamma$ will only contain a few samples (possibly none), making learning impossible. This problem could be easily solved by choosing lower values for γ. However, setting γ too low causes a slow convergence to a (sub)optimal solution. Therefore, the following alternative is used: CEM chooses a ratio $\rho \in [0,1]$ and adjusts $\hat{L}_\gamma$ to be the set of the best $\rho \cdot N$ samples. This corresponds to setting $\gamma := f(\mathbf{x}^{(\rho \cdot N)})$, provided that the samples are arranged in decreasing order of their values. The best $\rho \cdot N$ samples are called the *elite samples*. In practice, ρ is typically chosen from the range $[0.01, 0.1]$.

3.2.2 Cross-Entropy Distance

The second problem is that the distribution of the level set L_γ is not a member of any kind of parameterized distribution family and therefore it cannot be modelled accordingly. This problem is solved by changing the approximation goal: CEM chooses the distribution *g from the distribution family* $\mathcal{G}$ that approximates the empirical distribution over $\hat{L}_\gamma$ best. The best g is found by minimizing the distance between $\mathcal{G}$ and the uniform distribution over the elite samples. The distance measure is the *cross-entropy distance* (also called the Kullback-Leibler divergence (Kullback, 1959)). The cross-entropy distance of two distributions g and h is defined as

$$D_{CE}(g||h) = \int g(\mathbf{x}) \log \frac{g(\mathbf{x})}{h(\mathbf{x})} \, \mathrm{d}\mathbf{x}.$$

It is known that under mild regularity conditions, CEM converges with probability 1 (Costa, Jones, and Kroese, 2007). Furthermore, for a sufficiently large population, the global optimum is found with a high probability.

For many parameterized distribution families, the parameters of the minimum cross-entropy member can be computed easily from simple statistics of the elite samples. Below we sketch the special case when x is sampled from a Gaussian distribution. This distribution is used in the remainder of this paper.

Let the domain of learning be $D = \mathbb{R}^m$, and each component be drawn from independent Gaussian distributions with parameters μ_j, σ_j^2, $1 \le j \le m$, that is, a distribution $g \in \mathcal{G}$ is parameterized with $2m$ parameters.

After drawing N samples $\mathbf{x}^{(1)}, \ldots \mathbf{x}^{(N)}$ and having a threshold value γ (see 3.2.1), let E denote the set of elite samples, i.e.,

$$E := \{\mathbf{x}^{(i)} \mid f(\mathbf{x}^{(i)}) \ge \gamma\}. \tag{3}$$

With this notation, the distribution g' with minimum CE-distance from the uniform distribution over the elite set has parameters

$$\begin{aligned} \boldsymbol{\mu}' &:= (\mu_1', \ldots, \mu_m'), \quad \text{where} \\ \mu_j' &:= \frac{\sum_{\mathbf{x}^{(i)} \in E} x_j^{(i)}}{\sum_{\mathbf{x}^{(i)} \in E} 1} = \frac{\sum_{\mathbf{x}^{(i)} \in E} x_j^{(i)}}{\rho \cdot N} \end{aligned} \tag{4}$$

and

$$\begin{aligned} \boldsymbol{\sigma^{2}}' &:= ({\sigma^2}'_1, \ldots, {\sigma^2}'_m), \quad \text{where} \\ {\sigma^2}'_j &:= \frac{\sum_{\mathbf{x}^{(i)} \in E} (x_j^{(i)} - \mu_j')^T (x_j^{(i)} - \mu_j')}{\sum_{\mathbf{x}^{(i)} \in E} 1} \\ &= \frac{\sum_{\mathbf{x}^{(i)} \in E} (x_j^{(i)} - \mu_j')^T (x_j^{(i)} - \mu_j')}{\rho \cdot N}. \end{aligned}$$

In other words, the parameters of g' are simply the componentwise empirical means and variances of the elite set. For the derivation of this rule, we refer to de Boer *et al.* (2005). Changing the distribution parameters from $(\boldsymbol{\mu}, \boldsymbol{\sigma}^2)$ to $(\boldsymbol{\mu}', \boldsymbol{\sigma^2}')$ may be a too large step, so moving only a smaller step towards the new values (using step-size parameter α) is preferable. The resulting algorithm is summarized in Table 1.

```
input: $\boldsymbol{\mu}_0 = (\mu_{0,1}, \ldots, \mu_{0,m})$ and $\boldsymbol{\sigma}_0^2 = (\sigma_{0,1}^2, \ldots, \sigma_{0,m}^2)$
                                              % initial distribution parameters
input: $N$                                    % population size
input: $\rho$                                 % selection ratio
input: $T$                                    % number of iterations
for $t$ from 0 to $T-1$,                      % CE iteration main loop
    for $i$ from 1 to $N$,
        draw $\mathbf{x}^{(i)}$ from $\mathit{Gauss}^m(\boldsymbol{\mu}_t, \boldsymbol{\sigma}_t^2)$ % draw $N$ samples
        compute $f_i := f(\mathbf{x}^{(i)})$  % evaluate them
    sort $f_i$-values in descending order
    $\gamma_{t+1} := f_{\rho \cdot N}$        % level set threshold
    $E_{t+1} := \{\mathbf{x}^{(i)} \mid f(x^{(i)}) \geq \gamma_{t+1}\}$   % get elite samples
    $\mu'_j := \left(\sum_{\mathbf{x}^{(i)} \in E} x_j^{(i)}\right)/(\rho \cdot N)$   % get parameters of nearest distribution
    ${\sigma^2}'_j := \left(\sum_{\mathbf{x}^{(i)} \in E} (x_j^{(i)} - \mu'_j)^T (x_j^{(i)} - \mu'_j)\right)/(\rho \cdot N)$
    $\mu_{t+1,j} := \alpha \cdot \mu'_j + (1-\alpha) \cdot \mu_{t,j}$   % update with step size $\alpha$
    $\sigma_{t+1,j}^2 := \alpha \cdot {\sigma^2}'_j + (1-\alpha) \cdot \sigma_{t,j}$
end loop
```

Table 1: Pseudo-code of the Cross-Entropy Method for Gaussian distributions.

3.3 Normalizing Parameters

The value of each parameter x_i has to be selected from a range $[a_i; b_i]$. Due to the fact that the domain of a Gaussian distribution is unbounded, we sometimes have to throw away samples, which have one or more out-of-bounds values. Theoretically, this does not cause any complications: we may assume that samples having out-of-bound values are not discarded, they are only given a large negative score. With this assumption, we are able to apply the above algorithm without changes.

Furthermore, we apply two transformations to the parameters. First, the parameters are transformed to a logarithmic scale. We illustrate the reason by mentioning the progressive bias coefficient as an example. The progressive bias coefficient in MANGO has the following range $[0.1; 100]$. Without using a logarithmic scale, half of the values would be chosen in $[0.1; 50]$ and the other half in $[50; 100]$. Small values (say between 0.1 and 1), which could belong to the optimal solution, would be hardly drawn. Using a logarithmic scale, half of the values are picked in $[0.1; 3.16]$ and the other half in $[3.16; 100]$. Second, parameters that are only allowed to have integer values, are rounded off to the closest integer. Both transformations are part of the fitness function f.

4. EXPERIMENTS

In this section we are going to apply CEM to tune the MCTS parameters of MANGO. This is done by playing against GNU GO 3.7.10 (2007), level 0, on a 9×9 Go board. In each generation CEM draws 100 samples selecting the best 10 (*the elite*) samples.[4] A sample consists of playing a certain number of games for a CEM-generated parameter setting. So, the fitness function straightforwardly measures the winning rate for a batch of games. To obtain results rather quickly, MANGO only performs $3,000$ play-outs per move.

The section is organized as follows. An overview of the parameters together with their range is given in Subsection 4.1. Subsection 4.2 and 4.3 test the performance of a fixed and variable batch size, respectively. The best parameter setting against GNU GO is discussed in Subsection 4.4. The setting of MANGO (plus CEM) is compared against the old MANGO in four self-play experiments in Subsection 4.5.

[4] The values have been chosen based on previous results (de Boer *et al.*, 2005).

4.1 Overview of MCTS Parameters

In total 11 parameters are tuned by CEM, 1 parameter for the selection, 1 parameter for the expansion, 4 parameters for the play-out, 2 parameters for progressive bias, and 3 parameters for progressive widening. The parameters under consideration together with their range are given in Table 2. The table shows that for most parameters the value range is quite wide. This is done to assure that the optimal parameter value can be found. Regarding the initial distribution for each parameter, the mean is computed as the average of the lower and upper bound. The standard deviation is computed as half of the difference between the lower and upper bound. We remark that they are computed in a logarithmic way, since the logarithmic values of the parameters are used by CEM.

Parameter type	Parameter name	Range
Selection	UCT exploration coefficient C	$[0.2; 2]$
Expansion	Threshold T	$[2; 20]$
Play-out	Capture value P_c	$[100; 20,000]$
	Escape value P_e	$[50; 2,000]$
	Pattern exponent P_p	$[0.25; 4]$
	Proximity factor P_{md}	$[10; 500]$
Progressive Bias	PB factor PB_f	$[0.1; 100]$
	PB exponent PB_e	$[0.25; 4]$
Progressive Widening	PW initial # nodes k_{init}	$[2; 10]$
	PW factor A	$[20; 100]$
	PW base B	$[1.01; 1.5]$

Table 2: Parameters with their range.

4.2 Fixed Batch Size

In the following series of experiments we tested the learning performance of three batch sizes: 10, 50, and 500. The learning curves for the different batch sizes are depicted in Figure 3. The x-axis represents the *total* number of games played by the samples. The y-axis represents the average winning percentage against GNU GO of all the (100) samples in a generation.

A batch size of 10 games leads to an increase from a winning score of 30% to a winning score of more than 50% against GNU GO, after playing a total of 10,000 games. However, the performance increase stops after $20,000$ games, due to uncertainty caused by the small batch size. Subsequently, a batch size of 50 takes more than three times longer to achieve a score of 50%, but converges to a score of a little more than 60%. Finally, a batch size of 500 games is even slower, but the performance increase is steadier and the final score is therefore even better: 63.9%. With these settings, the results were obtained by using a cluster of 10 quad-core computers running for 3 days.

4.3 Variable Batch Size

In the previous subsection we saw that learning with a small batch size quickly leads to a performance increase, but convergence is to a suboptimal score only. In contrast, learning with a large batch size is slower but it converges to a higher score. In order to benefit from both approaches (quick increase, higher convergence), we propose to increase progressively the batch size after each generation. The scheme that we use is the following. At the first generation, the algorithm uses a batch size of 10. Next, at generation n, the algorithm uses a batch size of $10 \times 1.15^{n-1}$. The value of 1.15 has been chosen to ensure that, after 20 generations, the total number of games performed is the same as the number of games performed when using a batch size of 50. In the next series of experiments we compared the variable batch-size scheme with the three fixed batch sizes of the previous subsection. The results are depicted in Figure 4. The figure shows that a variable batch size performs a little bit worse than a fixed batch size of 50 or 500. These results suggest that a (sufficiently) large fixed batch size may be better than a variable batch size.

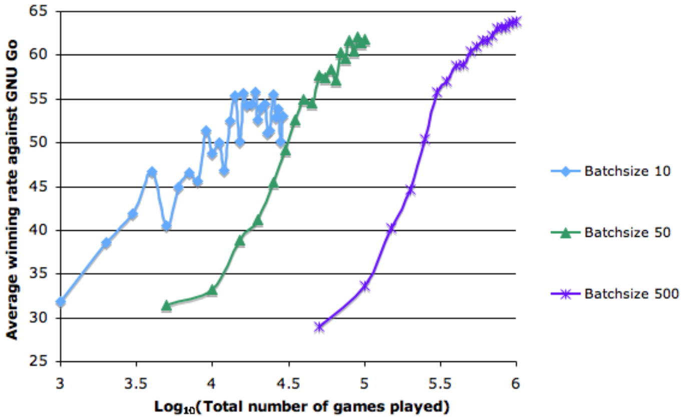

Figure 3: Learning curves for different batch sizes.

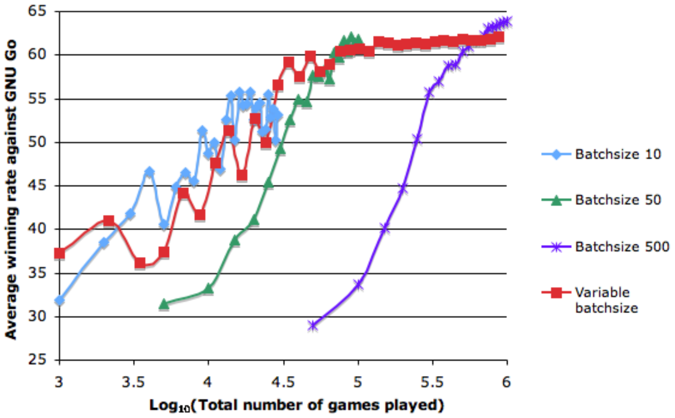

Figure 4: Using a variable number of games.

4.4 Comparison of Default and CEM Parameters

As we have seen in Subsection 4.2, after 20 generations the best parameter score was obtained by using a fixed batch size of 500. The achieved score of 63.9% is an underestimation because it represents the *average* score of *all* the samples at the 20^{th} generation. When updating the parameters by using the parameter means of the elite samples, the score may become better. This was tested in the following series of experiments. We compared the manual (default) parameter setting to the CEM parameter setting by playing twice (Default and CEM) 10,000 games against GNU GO. The parameter settings are reported in Table 3 together with their score against GNU GO.

We see that MANGO using the CEM parameters plays better against GNU GO than the default one (65.0% against 61.8%). We would like to remark that the parameters for the default version were already intensively tuned (but independently of each other). In Table 3, it can be seen that the values obtained by CEM are quite different from the default one. However, most parameter modifications do not affect the level of the program much. We observed that the fitness landscape is quite flat around the optimum. For instance, modifying the capture value from $5,000$ to $7,509$ has almost no influence on the playing strength of the program.

4.5 Self-Play Experiment

As we saw in the previous subsection, the parameters were fine-tuned by playing with a short time setting ($3,000$ play-outs per move) against GNU GO. To check whether the parameters were not overfitted for a specific

Parameter	Default	CEM
UCT exploration coefficient C	0.7	0.43
Expansion threshold T	10	3
Capture value P_c	$5,000$	$7,509$
Escape value P_e	500	911
Pattern exponent P_p	1	0.7
Proximity factor P_{md}	150	85
PB factor PB_f	8	5.3
PB exponent PB_e	1	1.1
PW initial # nodes k_{init}	5	3
PW factor A	40	58
PW base B	1.2	1.12
Winning rate against GNU GO	61.8%	65.0%
Number of games	$10,000$	$10,000$
Standard deviation	0.5%	0.5%

Table 3: Comparison of Default and CEM parameters.

Board size	Play-outs per move	Winning rate of CEM	Number of games	Standard deviation
9×9	$200,000$	57.4%	$1,000$	1.6%
9×9	$3,000$	55.4%	$10,000$	0.5%
13×13	$3,000$	61.3%	$10,000$	0.5%
19×19	$3,000$	66.3%	$10,000$	0.5%

Table 4: Self-play experiments: CEM vs. Default.

opponent or a specific time setting, four self-play experiments between MANGO with the CEM parameters and MANGO without the CEM parameters were executed. In the first experiment a short time setting of $3,000$ play-outs per move was used. MANGO with the CEM parameters won 55.4% of the $10,000$ games played against the default version. In the second experiment a longer time setting of $200,000$ play-outs per move was used. The CEM version won 57.4% of the $1,000$ games played. The third and fourth experiment were performed on two different board sizes, a 13×13 and a 19×19 Go board, respectively. The short time setting of $3,000$ play-outs per move was used. The CEM version won 61.3% of the games for 13×13 and 66.3% of the games for 19×19. The results suggest that the fine-tuning of parameters by CEM genuinely increased the playing strength of MANGO (see Table 4).

5. CONCLUSIONS AND FUTURE RESEARCH

This paper investigated the merit of the Cross-Entropy Method (CEM) regarding the automatic tuning of game parameters with a fixed and variable batch size. We tested CEM by tuning 11 parameters of our MCTS program MANGO. Experiments revealed that using a batch size of 500 games gave the best results, although the convergence was slow. To be more precise, these results were obtained by using a cluster of 10 quad-core computers running for 3 days. Interestingly, a small (and fast) batch size of 10 still gave reasonable results when compared to the best one. A variable batch size performed a little bit worse than a fixed batch size of 50 or 500. Subsequently, we showed that MANGO with the CEM parameters performed better against GNU GO than the MANGO version without. Moreover, in four self-play experiments with different time settings and board sizes, the CEM version of MANGO defeated each time the default version convincingly. Based on these results, we may conclude that parameter tuning by CEM genuinely improved the playing strength of MANGO, for various time settings and board sizes. The nature of our research allows the following generalization: a hand-tuned, MCTS-using game engine may improve its playing strength when re-tuning the parameters with CEM.

As future research, we believe that there is an interesting direction to improve the convergence speed. At this moment CEM evaluates the samples by playing all the games according to the batch size. The evaluation of a sample is never stopped prematurely, even if it is obvious that it will not belong to the set of elite samples. A kind of Multi-Armed Bandit algorithm (Robbins and Monro, 1951) could be applied to discontinue playing games when it is not likely that the sample will become an elite sample. As 90% of the samples in our experiments were *not* elite samples, it may therefore speed-up the convergence drastically.

Acknowledgements

This work is financed by the Dutch Organization for Scientific Research (NWO) in the framework of the project Go for Go, grant number 612.066.409, and the ROLEC project, grant number 612.066.406.

6. REFERENCES

Anonymous (2007). GNU GO. www.gnugo.org.

Baxter, J., Tridgell, A., and Weaver, L. (1998). Experiments in Parameter Learning Using Temporal Differences. *ICCA Journal*, Vol. 21, No. 2, pp. 84–99.

Beal, D. F. and Smith, M. C. (2000). Temporal Difference Learning for heuristic search and game playing. *Information Sciences*, Vol. 122, No. 1, pp. 3–21.

Björnsson, Y. and Marsland, T. A. (2003). Learning Extension Parameters in Game-Tree Search. *Information Sciences*, Vol. 154, No. 3, pp. 95–118.

Boer, P.-T. de, Kroese, D. P., Mannor, S., and Rubinstein, R. Y. (2005). A Tutorial on the cross-entropy method. *Annals of Operations Research*, Vol. 134, pp. 19–67.

Bouzy, B. (2005). Associating Domain-Dependent Knowledge and Monte-Carlo Approaches within a Go Program. *Information Sciences*, Vol. 175, No. 4, pp. 247–257.

Bouzy, B. and Chaslot, G. M. J.-B. (2006). Monte-Carlo Go Reinforcement Learning Experiments. *IEEE 2006 Symposium on Computational Intelligence in Games, Reno, USA*, pp. 187–194.

Cazenave, T. (2007). Playing the Right Atari. *ICGA Journal*, Vol. 30, No. 1, pp. 35–42.

Chaslot, G. M. J.-B., Winands, M. H. M., Uiterwijk, J. W. H. M., Herik, H. J. van den, and Bouzy, B. (2008). Progressive Strategies for Monte-Carlo Tree Search. *New Mathematics and Natural Computation*, Vol. 4, No. 3, pp. 343–357.

Coquelin, P.-A. and Munos, R. (2007). Bandit Algorithms for Tree Search. *23rd Conference on Uncertainty in Artificial Intelligence (UAI 2007)*, Vancouver, Canada.

Costa, A., Jones, O. D., and Kroese, D. P. (2007). Convergence Properties of the Cross-Entropy Method for Discrete Optimization. *Operations Research Letters*, Vol. 35, No. 5, pp. 573–580.

Coulom, R. (2007a). Computing “Elo Ratings” of Move Patterns in the Game of Go. *ICGA Journal*, Vol. 30, No. 4, pp. 199–208.

Coulom, R. (2007b). Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. *Proceedings of the 5th International Conference on Computers and Games* (eds. H. J. van den Herik, P. Ciancarini, and H. H. L. M. Donkers), Vol. 4630 of *LNCS*, pp. 72–83, Springer-Verlag, Heidelberg, Germany.

Gelly, S. and Wang, Y. (2007). Mogo Wins 19 ×19 Go Tournament. *ICGA Journal*, Vol. 30, No. 2, pp. 111–113.

Gelly, S., Wang, Y., Munos, R., and Teytaud, O. (2006). Modifications of UCT with Patterns in Monte-Carlo Go. Technical Report 6062, INRIA.

Knuth, D. E. and Moore, R. W. (1975). An Analysis of Alpha-Beta Pruning. *Artificial Intelligence*, Vol. 6, No. 4, pp. 293–326.

Kocsis, L. and Szepesvári, C. (2006). Bandit Based Monte-Carlo Planning. *Machine Learning: ECML 2006* (eds. J. Fürnkranz, T. Scheffer, and M. Spiliopoulou), Vol. 4212 of *LNAI*, pp. 282–293, Springer-Verlag, Heidelberg, Germany.

Kocsis, L., Szepesvári, C., and Winands, M. H. M. (2006). RSPSA: Enhanced Parameter Optimisation in Games. *Advances in Computer Games Conference (ACG 2005)* (eds. H. J. van den Herik, S.-C. Hsu, T.-S. Hsu, and H. H. M. L. Donkers), Vol. 4250 of *LNCS*, pp. 39–56, Springer-Verlag, Berlin Heidelberg.

Kullback, S. (1959). *Information Theory and Statistics*. John Wiley and Sons, NY, USA.

Muehlenbein, H. (1997). The Equation for Response to Selection and its Use for Prediction. *Evolutionary Computation*, Vol. 5, No. 3, pp. 303–346.

Robbins, H. and Monro, S. (1951). A Stochastic Approximation Method. *Annals of Mathematical Statistics*, Vol. 22, No. 3, pp. 400–407.

Rubinstein, R. Y. (1999). The Cross-Entropy method for combinatorial and continuous optimization. *Methodology and Computing in Applied Probability*, Vol. 1, No. 2, pp. 127–190.

Schaeffer, J., Hlynka, M., and Jussila, V. (2001). Temporal Difference Learning Applied to a High-Performance Game-Playing Program. *International Joint Conference on Artifical Intelligence (IJCAI)*, pp. 529–534.

Sutton, R. and Barto, A. (1998). *Reinforcement Learning: An Introduction*. MIT Press.

Tesauro, G. (1995). Temporal Difference Learning and TD-Gammon. *Communications of the ACM*, Vol. 38, No. 3, pp. 58–68.

Werf, E. van der (2007). Steenvreter Wins 9 ×9 Go Tournament. *ICGA Journal*, Vol. 30, No. 2, pp. 109–110.

Winands, M. H. M., Kocsis, L., Uiterwijk, J. W. H. M., and Herik, H. J. van den (2002). Temporal Difference Learning and the Neural MoveMap Heuristic in the Game of Lines of Action. *GAME-ON 2002*, pp. 99–103, SCS Europe Bvba, Ghent, Belgium.