The Cross-Entropy Method Applied to SameGame

M.J.W. Tak

June 21, 2010

Abstract

This paper improves a Single-Player Monte-Carlo Tree Search (SP-MCTS) SameGame program by tuning its parameters with the Cross-Entropy Method (CEM). SP-MCTS can be used in two ways: (1) constructing one tree for the whole game and (2) constructing a tree for each move. Both approaches in combination with several simulation strategies were tuned by CEM. Constructing a tree for each move in combination with parameters tuned by CEM resulted in the best performance. With this setup, it was possible to obtain a score of 78,012 on the standardized test set of 20 positions. This is 4,014 points more than the manually tuned SameGame program that constructs one tree for the whole game. A disadvantage of CEM is its long training time. This paper proposes two procedures, called *SimpleCut* and *ConfidenceCut*, to improve the convergence rate. The best procedure, *ConfidenceCut*, reduces the training time of CEM by approximately 40%. **Keywords:** Cross-Entropy Method, Monte-Carlo Tree Search, SameGame

1 Introduction

The performance of a search algorithm typically depends on some set of parameters. Therefore, it is important to find those parameters that give the best performance. There are often several dependencies involved, which makes it difficult to find the optimal parameter setting manually.

Monte-Carlo Tree Search (MCTS) is an example of such a search algorithm [4, 6, 9]. Nowadays it is often used to determine the best moves in games, such as Go [6]. A main difference between Monte-Carlo Tree Search and other search techniques such as $\alpha\beta$ is that game dependent knowledge is not necessary. Therefore, it is well suited for games for which it is difficult to determine an evaluation function. Two examples of such games are Go and SameGame.

In the last few years several papers have been published about applying Monte-Carlo techniques to SameGame. Within these papers, the developed algorithms are evaluated on a predefined test set of 20 positions [12]. In 2008 Schadd *et al.* [16] were able to score 73,998 on this test set by using a method called Single-Player Monte-Carlo Tree Search (SP-MCTS). In 2009 Takes and Kosters [17] introduced a new simulation strategy with which a score of 76,764 is obtained. In the same year Cazenave applied nested Monte-Carlo Search which led to an even higher score of 77,934 [2]. Till 2010 the top score on this set was 84,414, held by the program "spurious_ai". Not much is known about how this score is obtained. In 2010 this record was claimed to be broken with 84,718 points by using a method called Heuristically Guided Swarm Tree Search [8].

The topic of this paper is tuning the parameters of a SameGame program that uses SP-MCTS as described in [16]. Currently, the parameters are tuned by hand. Therefore, by applying machine-learning methods to find the optimal parameter setting it is expected that the score of 73,998 can be increased.

This paper answers the main research question: *How to use the Cross-Entropy Method in order to improve the SP-MCTS SameGame program?*

The Cross-Entropy Method (CEM) [15] is chosen, because it has been successfully applied in parameter tuning for a MCTS Go program [3]. It is an evolutionary optimization method, related to Estimation-of-Distribution Algorithms (EDAs) [14]. Like in EDAs the range of possible solutions is represented with a probability distribution.

Besides the main research question, the following five questions are posed:

1. Currently the SameGame program constructs one large Monte-Carlo Search Tree for the whole game. Can the program be improved by constructing a smaller tree for each move?
2. Can the existing SameGame program be improved by CEM?
3. Currently the SameGame program applies a simulation strategy called TabuColorRandom. Can the program be improved by using other simulation strategies and by tuning their parameters?
4. What is the influence of the time limit during tuning

on the parameters found by CEM?

5. CEM may require quite some training time. Is it possible to reduce the training time?

This paper is organized as follows. First, Section 2 introduces CEM. Within Section 3 the rules of SameGame are explained. Then Section 4 describes SP-MCTS and its parameters to tune. Subsequently, Section 5 describes the main setup of the experiments and Section 6 handles the experiments in more detail. Finally, Section 7 contains the conclusions of this paper.

2 Cross-Entropy Method

The Cross-Entropy Method (CEM) [15] is an evolutionary optimization method, related to Estimation-of-Distribution Algorithms (EDAs) [14]. Like EDAs it is model based, because it represents its range of possible solutions with a parameterized probability distribution. CEM is often compared with genetic algorithms, because they are both evolutionary methods. However, the methods are not the same. A main difference between these methods is that genetic algorithms are not model based, but population based. Instead of using a model to represent their range of possible solutions, genetic algorithms maintain a concrete set of solution candidates [5].

CEM converges to a solution by iteratively changing the parameters of the probability distribution that represents the range of possible solutions. An iteration consists of three main steps. First, a set S of samples is drawn from the probability distribution. In the second step, each sample is evaluated and gets assigned a fitness value. A fixed amount of samples within S having the highest fitness are called the elite samples. In the third step, the elite samples are used to update the parameters of the probability distribution. After these steps, a new iteration begins which uses the updated distribution.

The updated distribution is the parameterized distribution having the smallest distance to the uniform distribution over the elite samples [3]. Applying this procedure, the probability of generating more or less the same elite samples the next iteration increases. To measure the distance between two probability density functions *u(y)* and *v(y)* the cross-entropy distance is used:

$$D = \int u(y) \ln \frac{u(y)}{v(y)}\, dy \tag{1}$$

Below it is outlined how to find the updated distribution in the case of a Gaussian distribution [3, 10]. Symbols written in bold refer to vectors. Let El be the set of elite samples of the current iteration. Each elite sample $\boldsymbol{e} \epsilon El$ is a vector containing values for each parameter to be optimized. For each of these parameters under consideration, a separate, independent, Gaussian distribution is maintained. CEM keeps track of these distributions with a vector $\boldsymbol{\mu}$ containing the means of these distributions and a vector $\boldsymbol{\sigma^2}$ containing the variances of these distributions. It turns out that $\boldsymbol{\mu_{elite}}$ and $\boldsymbol{\sigma^2_{elite}}$ as defined by the Equations 2 and 3 describe the Gaussian distribution having the smallest distance to the uniform distribution over the elite samples.

$$\boldsymbol{\mu_{elite}} = \frac{\sum_{\boldsymbol{e} \epsilon El} \mathbf{e}}{|El|} \tag{2}$$

$$\boldsymbol{\sigma^2_{elite}} = \frac{\sum_{\boldsymbol{e} \epsilon El} (\mathbf{e} - \boldsymbol{\mu_{elite}})^T \cdot (\mathbf{e} - \boldsymbol{\mu_{elite}})}{|El|} \tag{3}$$

Therefore, in the case of Gaussian distributions, the updated distributions have means equal to $\boldsymbol{\mu_{elite}}$ and variances equal to $\boldsymbol{\sigma^2_{elite}}$. This can easily be calculated with the equations above. Often a smoothing factor α is used to make smaller steps into the direction of the updated distributions. With a small enough α there is a high probability of finding the optimal solution [5].

Before CEM can start, the vectors $\boldsymbol{\mu}$ and $\boldsymbol{\sigma^2}$ need to be initialized. In this paper the same initialization as described in [3] is used. For each parameter, the mean of the corresponding distribution is equal to the average of the lower bound and upper bound of that parameter. The standard deviation is equal to a half of the difference between the lower bound and upper bound. The pseudo code of CEM is shown in Algorithm 1.

3 SameGame

SameGame originates from a game called Chain Shot that is invented around 1985 by Kuniaki Moribe [13]. In 1992 the game was released again under several different names, each referring to slightly different rules. Examples are: Bubble Breaker, Clickomania and SameGame. This section outlines the rules of SameGame [16].

SameGame is played by one player on a rectangular board of 15×15 squares, initially filled with pieces of five colors at random. A group refers to at least two pieces of the same color that are orthogonally adjacent. A move consists of selecting a group to remove from the board. Removing groups gives the player points according to $(n-2)^2$ where n refers to the size of the group. When groups are removed from the board the board position is updated according to the following rules:

- Pieces having an empty cell below them fall down till they hit another piece or the bottom of the rectangular board.
- If a column is empty, all columns to the right of it shift one column to the left.

If the board is empty an additional 1,000 points are added to score of the player. If the board is not empty and there are no moves possible, the game ends and points are deducted. The amount of points to deduct

Algorithm 1 CEM in the case of Gaussian distributions

```
Require: Population size $N$
Require: Number of elite samples $E$
Require: Number of iterations $T$
Require: Smoothing factor $\alpha$
Require: Initial vector $\boldsymbol{\mu}$ containing the means of the
    (Gaussian) parameter distributions
Require: Initial vector $\boldsymbol{\sigma}^2$ containing the variances of
    the (Gaussian) parameter distributions
 1: for $i \leftarrow 0$ to $T$ do
 2:     $S \leftarrow \emptyset$
 3:     for $v \leftarrow 0$ to $N$ do
 4:         draw a random sample $\boldsymbol{s_v}$ of parameter values
            from the parameter distributions with means $\boldsymbol{\mu}$
            and variances $\boldsymbol{\sigma}^2$
 5:         Add the vector $\boldsymbol{s_v}$ to the set $S$ containing the
            drawn samples
 6:         $f(v) \leftarrow$ fitness($\boldsymbol{s_v}$)
 7:     end for
 8:     $El \leftarrow$ {the samples $\boldsymbol{s_v} \epsilon S \,\| f(v)$ belongs to the
 9:     highest $E$ fitness values of $\boldsymbol{f}$}
        $\boldsymbol{\mu_{elite}} \leftarrow \frac{\sum_{e \epsilon El} \mathbf{e}}{|El|}$
        $\boldsymbol{\sigma^2_{elite}} \leftarrow \frac{\sum_{e \epsilon El} (\mathbf{e} - \boldsymbol{\mu_{elite}})^T \cdot (\mathbf{e} - \boldsymbol{\mu_{elite}})}{|El|}$
            $\boldsymbol{\mu} \leftarrow \alpha \cdot \boldsymbol{\mu_{elite}} + (1-\alpha) \cdot \boldsymbol{\mu}$
            $\boldsymbol{\sigma} \leftarrow \alpha \cdot \boldsymbol{\sigma_{elite}} + (1-\alpha) \cdot \boldsymbol{\sigma}$
10: end for
```

is calculated by applying $(n-2)^2$ to the pieces still left on the board where it is assumed that pieces of the same color are connected.

4 SP-MCTS and its Parameters

This section explains Monte-Carlo Tree Search (MCTS) [4, 6, 9] and the Single-Player Monte-Carlo Tree Search (SP-MCTS) as used in [16]. Parameters to tune by CEM are written in **bold**.

4.1 MCTS

MCTS is a best-first search technique in which board positions are not evaluated with an evaluation function but by playing random games (Monte-Carlo simulations). Each node in the tree corresponds to a board position. Furthermore, each node stores its visit count and its value. MCTS consists of four phases: *selection*, *simulation*, *expansion* and *backpropagation* [4]. These phases are repeated till time is up. The move to play is the child of the root with the highest value.

The *selection* strategy determines how to traverse the tree from the root to a leaf node. It should balance the exploitation of successful moves with the exploration of new moves. A strategy used in many programs is Upper Confidence bounds applied to Trees (UCT) [9].

When the selection strategy has reached a leaf node, the *simulation* phase starts. Within this phase, a random game is played. The starting point is the board position stored at the leaf node. Often a simulation strategy plays knowledge-based pseudo-random moves, which are better than real random moves.

The *expansion* phase adds one or more nodes to the tree. In [6] it is proposed to add one node to the tree per simulation. The node to add is the first board position encountered during the simulation that is currently not stored in the tree. Moreover, when the visit count of a node passes a certain threshold all the remaining children are added.

The *backpropagation* phase propagates the results of the game (won or loss) backwards from the leaf node to all nodes that are on the path from the root to this leaf node. The value of a node is the average of these results.

4.2 SP-MCTS

This subsection explains SP-MCTS [16] and how it is implemented in the SameGame program.

Selection strategy

As selection strategy a modified version of the UCT formula is used:

$$\bar{X} + C \cdot \sqrt{\frac{\ln t(N)}{t(N_i)}} + \sqrt{\frac{\sum x^2 - t(N_i) \cdot \bar{X}^2 + D}{t(N_i)}} \tag{4}$$

At a node N, the child N_i is selected that maximizes this formula. The first two terms correspond to the original UCT formula. Within these terms, the function *t(n)* returns the visit count for a given node n, $\bar{X}$ is the average game value and C is called the **UCTConstant**. The third term is added to represent a possible deviation of the child node. $\sum x^2$ is the sum of squared results obtained in the child node so far. This is corrected by the expected results given by $t(N_i) \cdot \bar{X}^2$. The **DeviationConstant** D is added to make sure that nodes, which have a high potential, are preferred.

An important difference between a two-player game and a one-player game like SameGame is that in the latter there is a much larger range of game values. For two-player games possible values are 0 for a draw, 1 for a win and -1 for a loss. In SameGame, arbitrary scores can be obtained. To correct for this, the constants C and D should be set such that they are feasible for this larger range of values.

Simulation strategies

The original SameGame program contains four simulation strategies:

- The *TabuColorRandom* strategy determines at the beginning of a simulation the most prominent color. During the rest of the simulation this color can only be played with a small probability determined by the parameter **ChanceChosenColor** or when no other moves are possible. The result is that large groups of this color are formed.
- The *TabuRandom* strategy works almost the same, but the color that is not allowed to be played is chosen randomly.
- The *ConnectionMaximization* strategy does a search of 1-ply deep. Each connection between two pieces within the same group is counted. The move to play maximizes the total number of connections.
- The *Random* strategy plays plain randomly.

TabuColorRandom has been successfully applied [2, 8, 16]. Still, it could be that a mix of strategies is better than playing only one strategy. To apply a mix of strategies a couple of other parameters are introduced.

The parameter **ChanceSameStrategy** determines the probability that within one single simulation the strategy chosen at the beginning is used till the end of this single simulation. To determine which strategy to play, each strategy has a corresponding **strategy weight**. A roulette-wheel selection mechanism uses these weights to determine which strategy to play next.

Another simulation strategy called Monte-Carlo with Roulette-Wheel Selection (MC-RWS) is proposed in [17]. A modified version of this strategy is added to the SameGame program in order to tune its parameters with CEM. The only modification made is leaving out the parameter θ. In the original version, θ is used to prevent problems with large values. However, when using doubles instead of integers data types these problems do not occur. Within this modified version, the probability of playing a certain color is defined as:

$$P(\chi_i) = \frac{1}{\chi - 1} \cdot \left(1 - \frac{g(\chi_i)^\alpha}{\sum_{k=1}^{\chi} g(\chi_k)^\alpha}\right) \\ \text{where } \alpha = 1 + (\beta/N) \cdot \sum_{k=1}^{\chi} g(\chi_k) \tag{5}$$

N refers to the initial amount of pieces, χ refers to the number of different colors and the function $g(\chi_k)$ returns the amount of pieces left of color χ_k. $\boldsymbol{\beta}$ is a constant which defines how much the focus is on creating larger groups. The idea behind this simulation strategy is that for several colors large groups are created. This is in contrast with the TabuColorRandom strategy which focuses only on creating large groups of one color.

Expansion

The expansion phase works as described in Section 4.1. The parameter **VisitsBeforeExpand** determines the minimum number of visits that are needed before all children of a node are added.

Backpropagation

In SP-MCTS a node not only keeps track of the average score, but also of the top score obtained when that node is visited. This is needed because the value of a node is calculated as the plain average of the results combined with the top score. The parameter **TopScoreWeight** determines the weight the top score has on this value.

Determining the move to play

For a one-player game there is no uncertainty about the opponent moves. Therefore, one large tree for the whole game can be constructed [16]. When time is up, the game is played from beginning till the end by using the moves leading to the top score in the tree.

A different way to determine the way of play is to construct a (smaller) tree for each move. After a tree is constructed for the first move, the sequence of moves leading to the top score found with this tree is stored and only the initial move of this sequence is played. To determine the next move, a new tree is constructed. The root corresponds to the new board position reached by the last move and its children are copied from the previous tree. SP-MCTS starts after this initial tree is constructed. If it finds a new top score, the sequence of moves leading to the new top score is stored and the first move of this sequence is played. However, it is possible that SP-MCTS does not find a better score than the current top score found in a previous constructed tree. If that happens, the first unplayed move of the stored sequence leading to this top score is played.

It is expected that the second search approach leads to higher scores, because for each move approximately the same amount of simulation time is used. This is in contrast with the first approach where the moves at a later stage of the game get less simulation time compared with the moves at the beginning of the game. This may result in suboptimal play at the end of the game.

5 Setup of the Experiments

This section presents the settings of CEM and the parameters to tune for each of the simulation strategies. Furthermore, it explains how the parameters found by CEM are evaluated.

5.1 Settings of CEM

In each experiment, the following settings are used. The sample size is equal to 100, the number of elite samples is equal to 10 and α is set to 0.6. These values are based

Parameters	Range	Tabu-Color	Mix	MC-RWS
UCTConstant	[0.1;10]	x	x	x
DeviationConstant	[0.1;20,000]	x	x	x
VisitsBeforeExpand	[2;20]	x	x	x
TopScoreWeight	[0;1]	x	x	x
ChanceSameStrategy	[0;1]		x	
ChanceChosenColor	[0;1]	x		
β as in MC-RWS	[1;20]			x
Strategy weights	[0;1,000]		x	

Table 1: Parameters to tune for each simulation strategy.

on recommendations made in [7] and [11]. Unless stated otherwise, each sample is evaluated by playing 30 games. The fitness of a sample is calculated as the sum of the scores of these games. 30 is chosen, because that allows most of the experiments to finish within one week. Each new iteration a new set of positions is created randomly which prevents over fitting on a certain training set.

For a puzzle such as SameGame, playing a game just means a search for the best moves. Therefore, a stopping criteria for this search is needed. This is either a time limit, or a limit on the number of nodes in the tree.

5.2 Simulation Strategies

In total three different simulation strategy setups are tuned by CEM. The first setting is **TabuColor** which only plays the TabuColorRandom strategy. The second setting is called **Mix**, because it plays four different strategies: Random, TabuColorRandom, TabuRandom and ConnectionMaximization. The third setting is called **MC-RWS** and plays only the modified version of Monte-Carlo with Roulette-Wheel Selection. Table 1 shows the parameters to tune and their ranges for each of the different settings.

5.3 Evaluating parameters

To determine the performance of the parameters found by CEM an independent test set of 250 randomly created positions is used. Average scores on this test set are obtained for five different time limits: 5, 10, 20, 30 and 60 seconds per position. As explained in Section 4.2 it is possible to construct a tree for each move. In that case, a time limit per move should be set. In order to use the above mentioned time limits per position, the time limits per move are obtained by dividing the time limits per position by 65, because 65 is a rough average of the number of moves per game. In [16] it is determined experimentally that the average number of moves is 64.4.

6 Experiments

For both search approaches explained in Section 4.2 parameters are tuned by CEM. Table 2 shows the performance of the parameters found by CEM when one tree is constructed per game. Table 3 shows the performance of the parameters when for each move a tree is constructed.

Sections 6.1 till 6.4 discuss the results shown in these tables. Subsequently, in Section 6.5 it is investigated whether the training time of CEM can be reduced by cutting of samples. Finally, Section 6.6 shows whether the score on the standardized test set can be increased when parameters are tuned by CEM [12].

6.1 A Tree per Move Compared with a Tree per Game

This section gives an answer to the research question: *Can the SameGame program be improved by constructing a tree for each move?* To answer this question, the two different search approaches described in Section 4.2 are compared for the strategies TabuColor and MC-RWS.

For both strategies the scores in Table 3 are higher than in Table 2. For example, at a time setting of 5 seconds the tuned TabuColor has a score of 2,405 in Table 2, while Table 3 shows a score of 2,652. This is an increase of 10.2% which is statistically significant at a level of 0.01. This means that the SameGame program performs better when for each move a tree is constructed than when one tree is constructed for the whole game.

Therefore, the expectations mentioned in Section 4.2 are confirmed. It seems indeed better to construct a tree for each move instead of one large search tree. An explanation for this result is that in the former case for each move approximately the same simulation time is used, while in the latter case moves done near the end of the game get less simulation time compared with moves done at the beginning of the game. This could lead to suboptimal play at the end of the game.

6.2 Tuned TabuColor Compared with the Manually Tuned Parameters

In this section the manually tuned TabuColor is compared with the TabuColor tuned by CEM in order to answer the research question: *Can the existing SameGame program be improved by CEM?*

In Table 2 the scores of TabuColor tuned by CEM are higher than the scores of the manually tuned TabuColor. For the first four time settings, the differences in score are significant at a level of 0.05. For a time setting of 60 seconds per position the difference is no longer significant. Due to this long time setting the average score of the manually tuned TabuColor is already relatively high which makes it more difficult to improve upon that.

The results in Table 3 are similar to the results in Table 2. However, in Table 3 the differences between the manually tuned TabuColor and the TabuColor tuned by CEM are smaller. They are only significant for the time limits 10 seconds and 20 seconds per position.

	TabuColor tuned manually	**TabuColor**	**Mix**	**MC-RWS**		
Stopping criteria per position during tuning:	-	10^5 nodes	10 s	5 s	10 s	20 s
Time limit per position: 5 s						
Avg. score	2,223	2,405	2,374	2,162	2,157	2,165
Standard deviation	466	509	496	457	495	520
Avg. generated nodes	172,280	101,973	75,697	34,572	31,674	38,092
Time limit per position: 10 s						
Avg. score	2,342	2,493	2,491	2,293	2,269	2,256
Standard deviation	484	506	484	479	467	502
Avg. generated nodes	337,801	197,461	150,698	71,312	63,371	78,008
Time limit per position: 20 s						
Avg. score	2,493	2,598	2,613	2,448	2,417	2,432
Standard deviation	541	481	473	526	509	518
Avg. generated nodes	648,117	359,927	297,853	141,586	131,111	160,832
Time limit per position: 30 s						
Avg. score	2,555	2,734	2,706	2,498	2,512	2,533
Standard deviation	521	492	498	533	503	526
Avg. generated nodes	933,778	567,749	442,078	208,657	208,505	249,893
Time limit per position: 60 s						
Avg. score	2,750	2,804	2,790	2,651	2,649	2,694
Standard deviation	560	481	488	494	512	505
Avg. generated nodes	1,643,233	1,130,711	848,898	444,679	435,312	514,648

Table 2: Comparison of the tuned strategies with the manually tuned TabuColor when only one tree is constructed.

	TabuColor tuned manually	**TabuColor**		**MC-RWS**
Stopping criteria per position during tuning:	-	1s per move	10 s per move	1 s per move
Time limit per move: 76 ms (± 5 s per position)				
Avg. score	2,588	2,614	2,652	2,361
Standard deviation	551	566	554	578
Avg. generated nodes	319,644	290,187	404,639	133,622
Time limit per move: 153 ms (± 10 s per position)				
Avg. score	2,644	2,723	2,749	2,453
Standard deviation	520	532	544	591
Avg. generated nodes	580,093	547,169	636,911	206,608
Time limit per move: 307 ms (± 20 s per position)				
Avg. score	2,742	2,797	2,856	2,548
Standard deviation	554	522	543	552
Avg. generated nodes	999,169	1,002,271	1,117,380	403,384
Time limit per move: 461 ms (± 30 s per position)				
Avg. score	2,822	2,858	2,876	2,577
Standard deviation	545	497	510	538
Avg. generated nodes	1,441,959	1,401,971	1,551,133	545,323
Time limit per move: 923 ms (± 60 s per position)				
Avg. score	2,880	2,919	2,913	2,714
Standard deviation	515	472	473	542
Avg. generated nodes	2,739,923	2,568,111	2,747,464	1,128,896

Table 3: Comparison of the tuned strategies with the manually tuned TabuColor when per move a tree is constructed.

Parameter	**Tuned manually**	**CEM**
UCTConstant	0.1	5.96
DeviationConstant	32	67.98
VisitsBeforeExpand	10	13
TopScoreWeight	0.02	0.49
ChanceChosenColor	0.003	0.0007

Table 4: The manually tuned parameters for TabuColor compared with the parameters found by CEM.

It can be concluded that CEM is indeed able to improve the SameGame program. The improvement by CEM is highest when one tree per game is constructed. Less improvement over the manually tuned TabuColor is gained when per move a tree is constructed. There are two reasons why there is less improvement in the second case by CEM than in the first case. The first reason is that when per move a tree is constructed the scores obtained by manual tuning are quite high making it more difficult to improve upon that. The second reason is that per move the time spent in the tree is small. Even on a time setting of 60 seconds per game, the time per move is still less than 1 second. This means that, compared with constructing one tree per game, the importance of finding a good move quickly increases. Therefore, the results of the random simulations become more important, while constructing the tree in an optimal way becomes less important. This makes the parameters, which determine how to construct the tree, also less important.

Table 4 shows the parameters in the case of constructing one tree per game. It is clear that the parameters found by CEM differ from the manually tuned parameters.

6.3 Simulation Strategies Compared

The original SameGame program applies the TabuColor strategy. This section answers the question: *Can the SameGame program be improved by using other simulation strategies and by tuning their parameters?*

Table 2 shows that the differences between TabuColor tuned by CEM and the strategy Mix are small. Therefore, using a mix of strategies does not lead to any improvement over a tuned TabuColor strategy. Also MC-RWS does not lead to any improvement, because it scores even lower than the manually tuned TabuColor.

These results are unexpected, because MC-RWS focuses on creating several large groups for different colors. In contrast, TabuColor creates only large groups for one color. The main reason why TabuColor performs better is that it is computationally much faster than the other strategies. In the same amount of time TabuColor can do more simulations which leads to higher scores. Table 2 confirms this, because it shows that for each time setting TabuColor generates more nodes than the other simulation strategies do for the same time setting.

	Time limit per position during tuning		
Parameter	5 s	10 s	20 s
UCTConstant	5.97	5.81	6.95
DeviationConstant	40.49	50.37	36.58
VisitsBeforeExpand	9	9	8
TopScoreWeight	0.34	0.55	0.33
β as in MC-RWS	16.57	17.14	16.67

Table 5: Parameters found by CEM for MC-RWS.

6.4 The Influence of Time Limits

To reduce the training time of CEM, the time limit of the games played in the evaluation phase of a sample should be limited. Therefore, the following question needs to be answered: *What is the influence of the time limit during tuning on the parameters found by CEM?*

Two experiments are conducted. In the first experiment MC-RWS is tuned for constructing one tree for the whole game. In the second experiment TabuColor is tuned for constructing a tree per move. These experiments are chosen to be completely different with respect to the search approach and simulation strategy in order to investigate whether the influence of the time limit is independent of the search approach and strategy.

In the first experiment, MC-RWS is tuned with three different time settings: 5, 10 and 20 seconds per position. Table 2 shows that there is not much difference in performance between the three different tuning settings. Table 5 indicates that the resulting parameters of the three different settings are close to each other which explains why the performance is more or less the same.

The second experiment is of the same kind, but now for each move a tree is constructed and the TabuColor strategy is used. The results are shown in Table 3. In this experiment the TabuColor strategy is tuned with two different time settings: 1 second and 10 seconds per move. For the case of 1 second, only 10 games are played per sample and in the case of 10 seconds only 1 game is played per sample. Less games are played, because with this setup the training time is already two weeks.

Table 3 shows that the differences between 1 second and 10 seconds are not significant. The reason is that the parameters found by CEM are close to each other. They are shown in Table 6.

Both experiments indicate that the time limit during tuning has not much influence on the parameters found by CEM. Therefore, it is possible to use a shorter time limit in order to reduce training time.

6.5 Reducing the Training Time

Most of the experiments took around one week to complete, due to the long evaluation phase of a sample. Therefore, this section answers the question: *Is it possible to reduce the training time?* It is suggested in [3] that a large reduction in training time should be possible

	Time limit per move during tuning	
Parameter	1 s	10 s
UCTConstant	4.10	4.31
DeviationConstant	80.99	96.67
VisitsBeforeExpand	14	11
TopScoreWeight	0.34	0.28

Table 6: Parameters found by CEM for TabuColor.

by cutting of samples in their evaluation phase when it is already clear that they will not become elite samples. Cutting off a sample means that the remaining positions in the evaluation phase of the sample are not played and that the sample is treated as being non-elite.

Procedure to cut off samples

The method to cut off samples works as follows. It keeps track of the elite samples (El) of the current iteration. During the evaluation phase of a sample, it is checked after each played game whether it is likely that the sample becomes an elite sample. In order to become an elite sample, the fitness should become at least as high as that of the worst elite sample. If during the evaluation phase it is expected that this is no longer possible, the sample is cut off. The expected fitness of a sample is based on the scores of the positions that are already played and on approximated scores for the unplayed positions. Two different approximation procedures are implemented:

SimpleCut Approximate the score of an unplayed position with the maximum score for that position obtained in previous samples of the current iteration.

ConfidenceCut Approximate the score of an unplayed position i according to Equation 6.

$$scorePosition_i = averagePosition_i + S_i \cdot t \quad (6)$$

In this equation, $averagePosition_i$ represents the average score on position i and S_i is an approximation of the standard deviation for that position. Both statistics are based on the scores for position i obtained in previous samples of the current iteration. With the variable t it can be tuned how easily samples are cut off. This approximation procedure has similarities with ProbCut [1], a forward pruning method used in $\alpha\beta$ search.

The expected fitness of the sample is the scores of the played positions added to the approximated scores for the unplayed positions. If this sum is below the fitness of the worst elite sample, the sample is cut off.

The first $|El|$ samples of each iteration are not allowed to be cut off. They are needed to gather information about the maximum scores, average scores and standard deviations. Furthermore, a lower bound on the unplayed position scores is set to prevent problems when the first $|El|$ samples are not performing well.

	SimpleCut	**ConfidenceCut**	
Stopping criteria per position during tuning:	10^5 nodes	10^5 nodes	10^5 nodes
t value	-	1.0	0.5
Time limit per position: 5 s			
Avg. score	2,422	2,396	2,376
Standard deviation	476	485	495
Avg. generated nodes	112,467	107,788	113,180
Time limit per position: 10 s			
Avg. score	2,510	2,527	2,486
Standard deviation	495	466	501
Avg. generated nodes	218,434	203,710	201,338
Time limit per position: 20 s			
Avg. score	2,626	2,650	2,641
Standard deviation	492	483	470
Avg. generated nodes	419,502	383,340	431,282
Time limit per position: 30 s			
Avg. score	2,735	2,709	2,698
Standard deviation	477	484	463
Avg. generated nodes	625,386	540,521	634,444
Time limit per position: 60 s			
Avg. score	2,835	2,813	2,810
Standard deviation	485	467	476
Avg. generated nodes	1,173,835	1,056,520	1,229,736

Table 7: Performance of the parameters found by CEM when samples are cut off and only one tree is constructed.

Results

In the experiments parameters for TabuColor are tuned by CEM where the above cut-off procedure is used. For *ConfidenceCut* two different t values are tested: 1.0 and 0.5. In the experiments one tree is constructed for the whole game, because constructing a tree per move would take too much time. In the latter case, with a tuning limit of 1 second per move and playing 10 games in the evaluation phase the training time is already two weeks.

In Table 7 the performance of the resulting parameters on the test set of 250 positions is shown. There are no statistically significant differences between the scores in this table and the scores in Table 2 of the tuned TabuColor that is not using the cut-off procedure. Therefore, it is possible to cut off samples without harming the performance of the parameters found by CEM.

Figure 1 shows the average score of the elite samples depending on the total number of games played. It is clear from this figure that the average score of the elite samples converges slowest when no cut-off procedure is used. *ConfidenceCut* with a t equal to 0.5 leads to the fastest convergence. The reason is that according to Figure 2 this method cuts off the most positions per iteration. Apparently the elite samples are not cut off, because otherwise the average score of the elite samples would not converge so fast. This again confirms that it

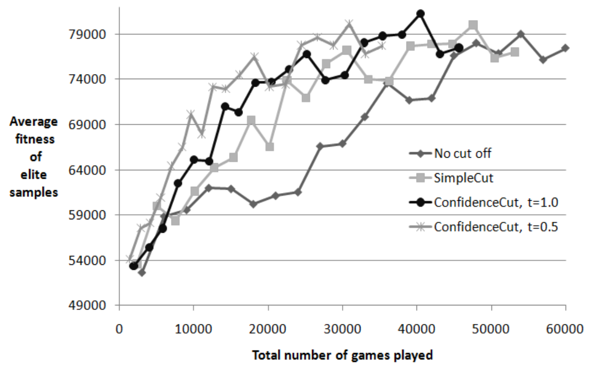

Figure 1: Average score of the elite samples depending on the number of games played.

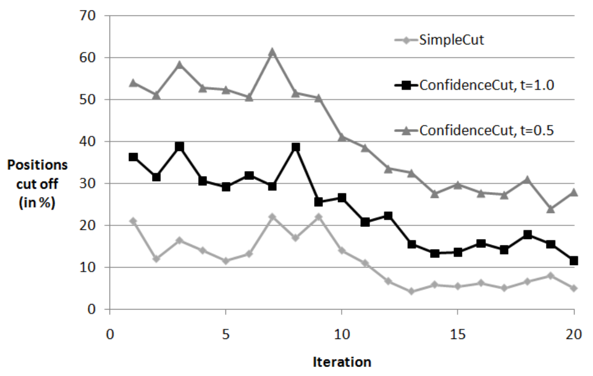

Figure 2: Positions cut off per iteration (in %).

is possible to cut off samples without harming the performance of the resulting parameters.

Figure 2 shows that the overall trend for the cut-off methods is that the number of positions that are cut off per iteration decreases. The reason is that the standard deviations of the parameters tuned by CEM, in the long run, go to 0. Therefore, within later iterations the generated samples have their parameter values closer to each other. This leads to less differences between the position scores of the samples which makes it more difficult to determine whether a sample should be cut off or not.

In total 40% of the positions can be cut off with *ConfidenceCut* and a t equal to 0.5. This reduces the training time of CEM by approximately 40%, because almost all time is spent in the evaluation phase of the samples.

6.6 Standardized Test Set

This section investigates whether the score of 73,998 found by Schadd *et al.* [16] on the predefined test set of 20 positions [12] can be increased by CEM.

Section 6.1 reveals that it is best to construct a tree per move. However, the score of 73,998 was obtained by constructing one tree per game. Therefore, to make a fair comparison between the manually tuned parameters and the tuned parameters by CEM two experiments are

Position	Tuned manually	Tuned by CEM		Tuned manually	Tuned by CEM
1	2,969	2,919	11	3,013	3,259
2	3,777	3,797	12	3,239	3,245
3	3,425	3,243	13	3,159	3,211
4	3,651	3,687	14	2,923	2,937
5	3,867	4,067	15	3,295	3,343
6	4,115	4,269	16	4,913	5,117
7	2,957	2,949	17	4,687	4,959
8	3,805	4,043	18	4,883	5,151
9	4,735	4,769	19	4,685	4,803
10	3,255	3,245	20	4,999	4,999
Total:				**76,352**	**78,012**

Table 8: Performance on the standardized test set [12].

conducted. The first experiment uses parameters tuned manually by Schadd *et al.* [16] which are shown in Table 4. The second experiment uses parameters found by CEM, these are shown in the last column of Table 6.

The setup of the experiments is as follows. For each of the first 30 moves a tree is constructed with a time limit of 10 seconds per move. The remaining moves are played according to the best move sequence found till then. The reason to stop searching after 30 moves is that the current best move sequence barely changes after the 30th move. Moreover, a meta search [16] of 480 is used. This means that each position is played 480 times and the top score found is the final score for that position. On a cluster with 20 cores, this setup leads to a similar time setting as used by Schadd *et al.* [16] of 2 to 3 hours per position. Furthermore, the same simulation strategy, TabuColorRandom, is applied.

The results shown in Table 8 confirm two of the conclusions drawn earlier. First, CEM is indeed able to improve the SameGame program, because the tuned parameters obtain 1,660 points more than the manually tuned parameters. Second, it again shows that constructing a tree per move is better than constructing one large search tree, because the score of 76,352 is higher than the original score of 73,998.

To conclude, using CEM and constructing a tree per move leads to 78,012 points which is even more than the 77,934 points obtained by Cazenave [2].

7 Conclusions

This paper answered the question: *How to use CEM in order to improve the SP-MCTS SameGame program?*

SP-MCTS can be used in two ways: (1) constructing one tree for the whole game and (2) constructing a tree for each move. Both search approaches in combination with different simulation strategies were tuned by CEM. Experiments showed that constructing a tree per move performs better than constructing a tree per game.

The manually tuned parameters for the TabuColorRandom strategy were compared with the parameters found by CEM. For both search approaches the parameters found by CEM obtained higher scores on the test set of 250 positions than the manually tuned parameters.

The experiments revealed that the strategies Mix and MC-RWS tuned by CEM did not improve the SameGame program. The strategy Mix performed almost equally as the TabuColorRandom strategy. MC-RWS performed even worse. The main reason is that MC-RWS is computationally more intensive.

A disadvantage of CEM is its long training time. Two methods were presented to decrease the training time. The first method is to reduce the time limit of the games played in the evaluation phase of a sample. The experiments indicated that this time limit has not much influence on the parameters found by CEM which makes it possible to reduce it. The second method is to cut off samples in their evaluation phase when it is already clear that they will not become elite samples. Two procedures were proposed to cut off samples, namely *SimpleCut* and *ConfidenceCut*. The latter performed best and reduces the training time of CEM by approximately 40%.

In the last experiment it is shown that by constructing a tree for each move and using parameters tuned by CEM it is possible to obtain a score of 78,012 on the standardized test set. This is 4,014 points more than the manually tuned SameGame program that constructs one tree for the whole game.

Based on all the results in this paper it is clear that CEM can be used to significantly increase the playing strength of the SP-MCTS SameGame program.

As future research, other ways to determine the fitness of a sample can be investigated. One suggestion is to incorporate the maximum score in the fitness. This could be beneficial, because in a meta search the maximum score is more important than the average. Another suggestion is to use a meta search in the evaluation phase of a sample. This increases the training time, but that can be limited with the *ConfidenceCut* procedure.

References

[1] Buro, M. (1995). Probcut: An effective selective extension of the alpha-beta algorithm. *ICCA Journal*, Vol. 18, No. 2, pp. 71–76.

[2] Cazenave, T. (2009). Nested Monte-Carlo search. *IJCAI'09*, pp. 456–461, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[3] Chaslot, G.M.J-B., Winands, M.H.M., Szita, I., and Herik, H.J. van den (2008a). Cross-Entropy for Monte-Carlo Tree Search. *ICGA Journal*, Vol. 31, No. 3, pp. 145–156.

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

[5] 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.

[6] Coulom, R. (2006). Efficient selectivity and backup operators in Monte-Carlo Tree Search. *CG2006* (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.

[7] 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, No. 1, pp. 19–67.

[8] Edelkamp, S., Kissmann, P., Sulewski, D., and Messerschmidt, H. (2010). Finding the needle in the haystack with heuristically guided swarm tree search. *Multikonferenz Wirtschaftsinformatik 2010* (eds. M. Schumann, L.M. Kolbe, M.H. Breitner, and A. Frerichs), pp. 2295–2308, Universitätsverlag Göttingen, Göttingen, Germany.

[9] Kocsis, L. and Szepesvári, C. (2006). Bandit based Monte-Carlo planning. *Proceedings of the EMCL 2006* (eds. J. Fürnkranz, T. Scheffer, and M. Spiliopoulou), Vol. 4212 of *LNCS*, pp. 282–293, Springer-Verlag, Heidelberg, Germany.

[10] Kroese, D.P., Rubinstein, R.Y., and Porotsky, S. (2006). The Cross-Entropy method for continuous multiextremal optimization. *Methodology and Computing in Applied Probability*, Vol. 8, No. 3, pp. 383–407.

[11] Mannor, S., Peleg, D., and Rubinstein, R. (2005). The Cross Entropy method for classification. *ICML '05: Proceedings of the 22nd international conference on Machine learning*, pp. 561–568, ACM, New York, NY.

[12] Misch, S. and Schulze, A. (2007). Js-games.de. `www.js-games.de/eng/games/samegame`.

[13] Moribe, K. (1985). Chain shot! *Gekkan ASCII*. In Japanese.

[14] Mühlenbein, H. (1997). The equation for response to selection and its use for prediction. *Evolutionary Computation*, Vol. 5, No. 3, pp. 303–346.

[15] 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.

[16] Schadd, M.P.D., Winands, M.H.M., Herik, H.J. van den, Chaslot, G.M.J-B., and Uiterwijk, J.W.H.M. (2008). Single-Player Monte-Carlo Tree Search. *CG2008* (eds. H.J. van den Herik, X. Xu, Z. Ma, and M.H.M. Winands), Vol. 5131 of *LNCS*, pp. 1–12, Springer.

[17] Takes, F.W. and Kosters, W.A. (2009). Solving SameGame and its chessboard variant. *BNAIC 2009* (eds. T. Calders, K. Tuyls, and M. Pechenizkiy), pp. 249–256, Eindhoven, The Netherlands.