Chapter 3

Single-Player Monte-Carlo Tree Search

This chapter is an updated and abridged version of the following publications:

1. Schadd, M.P.D., Winands, M.H.M., Herik, Chaslot, G.M.J-B., H.J. van den, and Uiterwijk, J.W.H.M. (2008a). Single-Player Monte-Carlo Tree Search. *Proceedings of the 20st BeNeLux Conference on Artificial Intelligence (BNAIC'08)* (eds. A. Nijholt, M. Pantic, M. Poel, and H. Hondorp), pp. 361–362, University of Twente, Enschede, The Netherlands.
2. Schadd, M.P.D., Winands, M.H.M., Herik, H.J. van den, and Aldewereld, H. (2008b). Addressing NP-Complete Puzzles with Monte-Carlo Methods. *Proceedings of the AISB 2008 Symposium on Logic and the Simulation of Interaction and Reasoning*, Vol. 9, pp. 55–61, The Society for the Study of Artificial Intelligence and Simulation of Behaviour, Brighton, United Kingdom.
3. Schadd, M.P.D., Winands, M.H.M., Herik, H.J. van den, Chaslot, G.M.J-B. and Uiterwijk, J.W.H.M. (2008c). Single-Player Monte-Carlo Tree Search. *Computers and Games (CG 2008)* (eds. H.J. van den Herik, X. Xu, Z. Ma, and M.H.M. Winands), Vol. 5131 of *Lecture Notes in Computer Science (LNCS)*, pp. 1–12, Springer-Verlag, Berlin, Germany.

The traditional approaches to deterministic one-player games with perfect information (Kendall, Parkes, and Spoerer, 2008) are applying A* (Hart *et al.*, 1968) or IDA* (Korf, 1985). These methods have been quite successful for solving this type of games. The disadvantage of the methods is that they require an admissible heuristic evaluation function. The construction of such a function can be difficult. Since Monte-Carlo Tree Search (MCTS) does not require an admissible heuristic, it may be an interesting alternative. Because of its success in two-player games (cf. Lee, Müller, and Teytaud, 2010) and multi-player games (Sturtevant, 2008a), this chapter investigates the application of MCTS in deterministic one-player games with perfect information.

So far, MCTS has not been widely applied in one-player games. One example is the Sailing Domain (Kocsis and Szepesvári, 2006), which is a non-deterministic game with perfect information. MCTS has also been used for optimization and planning problems which can be represented as deterministic one-player games. Chaslot *et al.* (2006a) applied MCTS in production management problems. Mesmay *et al.* (2009) proposed the MCTS variant TAG for optimizing libraries for different platforms. Schadd *et al.* (2008c) showed that MCTS was able to achieve high scores in the puzzle[1] SameGame.

This chapter answers the first research question by proposing an MCTS method for a one-player game, called Single-Player Monte-Carlo Tree Search (SP-MCTS). MCTS for two-player games, as described in Section 2.7, forms the starting point for this search method. We adapted MCTS by two modifications resulting in SP-MCTS. The modifications are (1) in the selection strategy and (2) in the backpropagation strategy. SP-MCTS is tested in the game of SameGame, because there exists no reliable admissible heuristic evaluation function for this game.

The article is organized as follows. In Section 3.1 we present the rules, complexity and related work of SameGame. In Section 3.2 we discuss why the classic approaches A* and IDA* are not suitable for SameGame. Then, we introduce the SP-MCTS approach in Section 3.3. Section 3.4 describes the Cross-Entropy Method which is used for tuning the SP-MCTS parameters. Experiments and results are given in Section 3.5. Section 3.6 gives the chapter conclusions and indicates future research.

3.1 SameGame

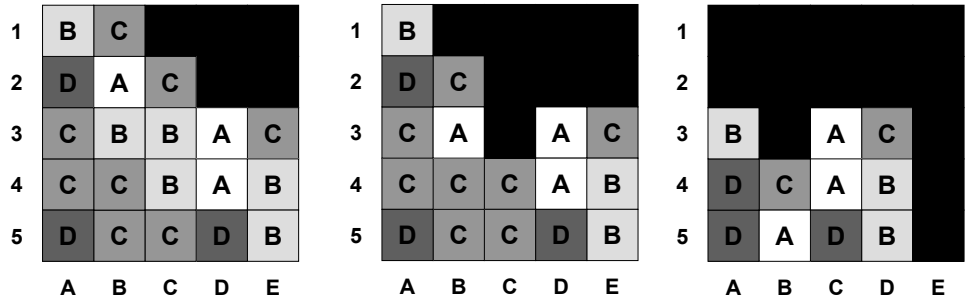
SameGame is a puzzle invented by Kuniaki Moribe under the name *Chain Shot!* in 1985. It was distributed for Fujitsu FM-8/7 series in a monthly personal computer magazine called *Gekkan ASCII* (Moribe, 1985). The puzzle was afterwards re-created by Eiji Fukumoto under the name of *SameGame* in 1992.

In this section, we first explain the rules in Subsection 3.1.1. Subsequently, we give an analysis of the complexity of SameGame in Subsection 3.1.2. Finally, we present related work in Subsection 3.1.3.

3.1.1 Rules

SameGame is played on a vertically oriented 15×15 board initially filled with blocks of 5 colors at random. A move consists of removing a group of (at least two) orthogonally adjacent blocks of the same color. The blocks on top of the removed group fall down. As soon as an empty column occurs, the columns to the right of the empty column are shifted to the left. Therefore, it is impossible to create separate subgames. For each removed group points are rewarded. The number of points is dependent on the number of blocks removed and can be computed by the formula $(n-2)^2$, where n is the size of the removed group.

[1] From now on, we call one-player deterministic games with perfect information for the sake of brevity *puzzles* (Kendall *et al.*, 2008).



(a) Playing "B" in the center column.

(b) Playing "C" in the center column.

(c) Resulting position.

Figure 3.1: Example SameGame moves.

We show two example moves in Figure 3.1. When the "B" group in the third column with a connection to the second column of position 3.1(a) is played, the "B" group is removed from the game. In the second column the "CA" blocks fall down and in the third column the "C" block falls down, resulting in position 3.1(b). Due to this move, it is now possible to remove a large group of "C" blocks ($n = 6$). Owing to an empty column the two columns at the right side of the board are shifted to the left, resulting in position 3.1(c).[2] The first move is worth 1 point; the second move is worth 16 points.

The game is over if no more blocks can be removed. This happens when either the player (1) has removed all blocks or (2) is left with a position where no adjacent blocks have the same color. In the first case, 1,000 bonus points are rewarded. In the second case, points are deducted. The formula for deducting is similar to the formula for awarding points but now iteratively applied for each color left on the board. Here it is assumed that all blocks of the same color are connected.

There are variations that differ in board size and the number of colors, but the 15×15 variant with 5 colors is the accepted standard. If a variant differs in the scoring function, it is named differently (e.g., Clickomania or Jawbreaker, cf. Biedl *et al.*, 2002; Julien, 2008).

3.1.2 Complexity of SameGame

The complexity of a game indicates a measure of difficulty for solving the game. Two important measures for the complexity of a game are the game-tree complexity and the state-space complexity (Allis, 1994). The game-tree complexity is an estimation of the number of leaf nodes that the complete search tree would contain to solve the initial position. The state-space complexity indicates the total number of possible states.

For SameGame these complexities are as follows. The game-tree complexity

[2] Shifting the columns at the left side to the right would not have made a difference in number of points. For consistency, we always shift columns to the left.

can be approximated by simulation. By randomly playing 10^6 puzzles, the average length of the game was estimated to be 64.4 moves and the average branching factor to be 20.7, resulting in a game-tree complexity of 10^{85}. The state-space complexity is computed rather straightforwardly. It is possible to calculate the number of combinations for one column by $C = \sum_{n=0}^{r} c^n$ where r is the height of the column and c is the number of colors. To compute the state-space complexity we take C^k where k is the number of columns. For SameGame there exist 10^{159} states. This is an over-estimation because a small percentage of the positions are symmetrical.

Furthermore, the difficulty of a game can be described by deciding to which complexity class it belongs (Johnson, 1990). The similar game Clickomania was proven to be NP-complete by Biedl *et al.* (2002). However, the complexity of SameGame could be different. The more points are rewarded for removing large groups, the more the characteristics of the game may differ from Clickomania. In Clickomania the only goal is to remove as many blocks as possible, whereas in SameGame points are rewarded for removing large groups as well.

Theorem. SameGame is at least as difficult as Clickomania.

Proof. A solution S of a SameGame problem is defined as a path from the initial position to a terminal position. Either S (1) has removed all blocks from the game or (2) has finished with blocks remaining on the board. In both cases a search has to be performed to investigate whether a solution exists that improves the score and clears the board.

Clickomania is a variant of SameGame where no points are rewarded and the only objective is to clear the board. Finding only one solution to this problem is easier than finding the highest-scoring solution (as in SameGame). Therefore, SameGame is at least as difficult as Clickomania. □

3.1.3 Related Work

For the game of SameGame some research has been performed. The contributions are benchmarked on a standardized test set of 20 positions.[3] The first SameGame program has been written by Billings (2007). This program applies a non-documented method called *Depth-Budgeted Search* (DBS). When the search reaches a depth where its budget has been spent, a greedy simulation is performed. On the test set his program achieved a total score of 72,816 points with 2 to 3 hours computing time per position. Schadd *et al.* (2008c) set a new high score of 73,998 points by using Single-Player Monte-Carlo Tree Search (SP-MCTS). This chapter will describe SP-MCTS in detail. Takes and Kosters (2009) proposed *Monte Carlo with Roulette-Wheel Selection* (MC-RWS). It is a simulation strategy that tries to maximize the size of one group of a certain color and at the same time tries to create larger groups of another color. On the test set their program achieved a total score of 76,764 points with a time limit of 2 hours. In the same year Cazenave (2009) applied Nested Monte-Carlo Search which led to an even higher score of 77,934. Until the year 2010, the top score on this set was 84,414 points, held by the program

[3]The positions can be found at: www.js-games.de/eng/games/samegame.

SPURIOUS AI.[4] This program applies a method called Simple Breadth Search (SBS), which uses beam search, multiple processors and a large amount of memory (cf. Takes and Kosters, 2009). Further details about this program are not known. Later in 2010 this record was claimed to be broken with 84,718 points by using a method called *Heuristically Guided Swarm Tree Search* (HGSTS) (Edelkamp *et al.*, 2010), which is a parallelized version of MCTS.

3.2 A* and IDA*

The classic approach to puzzles involves methods such as A* (Hart *et al.*, 1968) and IDA* (Korf, 1985). A* is a best-first search where all nodes have to be stored in a list. The list is sorted by an admissible evaluation function. At each iteration the first element is removed from the list and its children are added to the sorted list. This process is continued until the goal state arrives at the start of the list.

IDA* is an iterative deepening variant of A* search. It uses a depth-first approach in such a way that there is no need to store the complete tree in memory. The search continues depth-first until the cost of arriving at a leaf node and the value of the evaluation function exceeds a certain threshold. When the search returns without a result, the threshold is increased.

Both methods are strongly dependent on the quality of the evaluation function. Even if the function is an admissible under-estimator, it still has to give an accurate estimation. Classic puzzles where this approach works well are the Eight Puzzle with its larger relatives (Korf, 1985; Sadikov and Bratko, 2007) and Sokoban (Junghanns, 1999). Here a good under-estimator is the well-known Manhattan Distance. The main task in this field of research is to improve the evaluation function, e.g., with pattern databases (Culberson and Schaeffer, 1998; Felner *et al.*, 2005).

These classic methods fail for SameGame because it is not straightforward to make an admissible function that still gives an accurate estimation. An attempt to make such an evaluation function is by just awarding points to the current groups on the board. This resembles the score of a game where all groups are removed in a top-down manner. However, if an optimal solution to a SameGame problem has to be found, we may argue that an "over-estimator" of the position is required, because in SameGame the score has to be maximized, whereas in common applications costs have to be minimized (e.g., shortest path to a goal). An admissible "over-estimator" can be created by assuming that all blocks of the same color are connected and would be able to be removed at once. This function can be improved by checking whether there is a color with only one block remaining on the board. If this is the case, the 1,000 bonus points for clearing the board may be deducted because the board cannot be cleared completely. However, such an evaluation function is far from the real score for a position and does not give good results with A* and IDA*. Our tests have shown that using A* and IDA* with the proposed "over-estimator" results in a kind of breadth-first search. The problem is that after expanding a node, the heuristic value of a child can be significantly lower than the value of its parent, unless a move removes all blocks with one color from the board. We expect that

[4]The exact date when the scores were uploaded to http://www.js-games.de/ is unknown.

other Depth-First Branch-and-Bound methods (Vempaty, Kumar, and Korf, 1991) suffer from the same problem. Since no good evaluation function has been found yet, SameGame presents a new challenge for puzzle research.

3.3 Single-Player Monte-Carlo Tree Search

Based on MCTS, we propose an adapted version for puzzles: Single-Player Monte-Carlo Tree Search (SP-MCTS). We discuss the four steps (selection, play-out, expansion and backpropagation) and point out differences between SP-MCTS and MCTS in Subsections 3.3.1-3.3.4. SameGame serves as example domain to explain SP-MCTS. The final move selection is described in Subsection 3.3.5. Subsection 3.3.6 describes how randomized restarts may improve the score.

3.3.1 Selection Step

Selection is the strategic task to select one of the children of a given node. It controls the balance between *exploitation* and *exploration*. Exploitation is the task to focus on the moves that led to the best results so far. Exploration deals with the less promising moves that still may have to be explored, due to the uncertainty of their evaluation so far. In MCTS at each node starting from the root, a child has to be selected until a position is reached that is not part of the tree yet. Several strategies have been designed for this task (Chaslot *et al.*, 2006b; Kocsis and Szepesvári, 2006; Coulom, 2007a).

Kocsis and Szepesvári (2006) proposed the selection strategy UCT (Upper Confidence bounds applied to Trees). For SP-MCTS, we use a modified UCT version. At the selection of node p with children i, the strategy chooses the move, which maximizes the following formula.

$$v_i + C \times \sqrt{\frac{\ln n_p}{n_i}} + \sqrt{\frac{\sum r^2 - n_i \times v_i^2 + D}{n_i}} \tag{3.1}$$

The first two terms constitute the original UCT formula. It uses n_i as the number of times that node i was visited where i denotes a child and p the parent to give an upper confidence bound for the average game value v_i. For puzzles, we added a third term, which represents a possible deviation of the child node (Chaslot *et al.*, 2006a; Coulom, 2007a). It contains the sum of the squared results so far $\left(\sum r^2\right)$ achieved in the child node corrected by the expected results $n_i \times v_i^2$. A high constant D is added to ensure that nodes, which have been rarely explored, are considered uncertain. Below we describe two differences between puzzles and two-player games, which may affect the selection strategy.

First, the essential difference between puzzles and two-player games is the *range of values*. In two-player games, the outcome of a game is usually denoted by *loss*, *draw*, or *win*, i.e., $\{-1, 0, 1\}$. The average score of a node always stays within $[-1, 1]$. In a puzzle, an arbitrary score can be achieved that is not by definition within a preset interval. For example, in SameGame there are positions, which result in a

value above 5,000 points. As a first solution to this issue we may set the constants C and D in such a way that they are feasible for a certain interval (e.g., [0, 6000] in SameGame). A second solution would be to scale the values back into the above mentioned interval $[-1, 1]$, given a maximum score (e.g., 6,000 for a SameGame position). When the exact maximum score is not known a theoretical upper bound can be used. For instance, in SameGame a theoretical upper bound is to assume that all blocks have the same color. A direct consequence of such an upper bound is that due to the high upper bound, the game scores are located near to zero. It means that the constants C and D have to be set with completely different values compared to two-player games. We have opted for the first solution in our program.

A second difference is that puzzles do not have any *uncertainty on the opponent's play*. It means that the line of play has to be optimized without the hindrance of an opponent (Chaslot, 2010). Due to this, not only the average score but the top score of a move can be used as well. Based on manual tuning, we add the top score using a weight W with a value of 0.02 to the average score.

Here we remark that we follow Coulom (2007a) in choosing a move according to the selection strategy only if n_p reaches a certain threshold T (we set T to 10). As long as the threshold is not exceeded, the simulation strategy is used. The latter is explained in the next subsection.

3.3.2 Play-Out Step

The play-out step begins when we enter a position that is not part of the tree yet. Moves are randomly selected until the game ends. This succeeding step is called the play-out. In order to improve the quality of the play-outs, the moves are chosen quasi-randomly based on heuristic knowledge (Bouzy, 2005; Gelly *et al.*, 2006; Chen and Zhang, 2008). For SameGame, several simulation strategies exist.

We have proposed two simulation strategies, called TabuRandom and TabuColorRandom (Schadd *et al.*, 2008c). Both strategies aim at creating large groups of one color. In SameGame, creating large groups of blocks is advantageous. TabuRandom chooses a random color at the start of a play-out. The idea is not to allow to play this color during the play-out unless there are no other moves possible. With this strategy large groups of the chosen color are formed automatically. The new aspect in the TabuColorRandom strategy with respect to the previous strategy is that the chosen color is the color most frequently occurring at the start of the play-out. This may increase the probability of having large groups during the play-out. We also use the ϵ-greedy policy to occasionally deviate from this strategy (Sutton and Barto, 1998). Before the simulation strategy is applied, with probability ϵ a random move is played. Based on manual tuning, we chose $\epsilon = 0.003$.

An alternative simulation strategy for SameGame is Monte-Carlo with Roulette-Wheel Selection (MC-RWS) (Takes and Kosters, 2009). This strategy not only tries to maximize one group of a certain color, but also tries to create bigger groups of other colors. Tak (2010) showed that MC-RWS does not improve the score in SP-MCTS because it is computationally more expensive than TabuColorRandom.

3.3.3 Expansion Step

The expansion strategy decides on which nodes are stored in memory. Coulom (2007a) proposed to expand one child per play-out. With his strategy, the expanded node corresponds to the first encountered position that was not present in the tree. This is also the strategy we used for SP-MCTS.

3.3.4 Backpropagation Step

During the backpropagation step, the result of the play-out at the leaf node is propagated backwards to the root. Several backpropagation strategies have been proposed in the literature (Chaslot *et al.*, 2006b; Coulom, 2007a). The best results that we have obtained for SP-MCTS was by using the plain average of the play-outs. Therefore, we update (1) the average score of a node. Additional to this, we also update (2) the sum of the squared results because of the third term in the selection strategy (see Formula 3.1), and (3) the top score achieved so far.

3.3.5 Final Move Selection

The four steps are iterated until the time runs out.[5] When this occurs, a final move selection is used to determine which move should be played. In two-player games (with an analogous run-out-of-time procedure) the best move according to this strategy is played by the player to move. The opponent has then time to calculate his response. But in puzzles this can be done differently. In puzzles it is not required to wait for an unknown reply of an opponent. It is therefore possible to perform one large search from the initial position and then play all moves at once. With this approach all moves at the start are under consideration until the time for SP-MCTS runs out. It has to be investigated whether this approach outperforms an approach that allocates search time for every move. These experiments are presented in Subsection 3.5.3.

3.3.6 Randomized Restarts

We observed that it is important to generate deep trees in SameGame (see Subsection 3.5.2). However, by exploiting the most-promising lines of play, the SP-MCTS can be caught in local maxima. So, we randomly restart SP-MCTS with a different seed to overcome this problem. Because no information is shared between the searches, they explore different parts of the search space. This method resembles root parallelization (Chaslot *et al.*, 2008b).

Root parallelization is an effective way of using multiple cores simultaneously (Chaslot *et al.*, 2008b). However, we argue that root parallelization may also be used for avoiding local maxima in a single-threaded environment. Because there is no actual parallelization, we call this *randomized restarts*. Subsection 3.5.3 shows that randomized restarts are able to increase the average score significantly.

[5] In general, there is no time limitation for puzzles. However, a time limit is necessary to make testing possible.

3.4 The Cross-Entropy Method

Choosing the correct SP-MCTS parameter values is important for its success. For instance, an important parameter is the C constant which is responsible for the balance between exploration and exploitation. Optimizing these parameters manually may be a hard and time-consuming task. Although it is possible to make educated guesses for some parameters, for other parameters it is not possible. Specially hidden dependencies between the parameters complicate the tuning process. Here, a learning method can be used to find the best values for these parameters (Sutton and Barto, 1998; Beal and Smith, 2000).

The *Cross-Entropy Method* (CEM) (Rubinstein, 2003) has successfully tuned parameters of an MCTS program in the past (Chaslot *et al.*, 2008c). CEM is an evolutionary optimization method, related to *Estimation-of-Distribution Algorithms* (EDAs) (Mühlenbein, 1997). CEM is a population-based learning algorithm, where members of the population are sampled from a parameterized probability distribution (e.g., Gaussian, Binomial, Bernoulli, etc.). This probability distribution represents the range of possible solutions.

CEM converges to a solution by iteratively changing the parameters of the probability distribution (e.g., μ and σ for a Gaussian distribution). An iteration consists of three main steps. First, a set S of vectors $x \in X$ is drawn from the probability distribution, where X is some parameter space. These parameter vectors are called samples. In the second step, each sample is evaluated and gets assigned a fitness value. A fixed number 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.

Generally, CEM aims to find the optimal solution x^* for a learning task described in the following form

$$x^* \leftarrow \underset{x}{\operatorname{argmax}}\, f(x), \tag{3.2}$$

where x^* is a vector containing all parameters of the (approximately) optimal solution. f is a fitness function that determines the performance of a sample x (for SameGame this is the average number of points scored on a set of positions). The main difference of CEM to traditional methods is that CEM does not maintain a single candidate solution, but maintains a distribution of possible solutions.

There exist two methods for generating samples from the probability distribution, (1) random guessing and (2) distribution focusing (Rubinstein, 2003). *Random guessing* straightforwardly creates samples from the distribution and selects the best sample as an estimate for the optimum. If the probability distribution peaked close to the global optimum, random guessing may obtain a good estimate. If the distribution is rather uniform, the random guessing is unreliable. After drawing a moderate number of samples from a distribution, it may be impossible to give an acceptable approximation of x^*, but it may be possible to obtain a better sampling distribution. To modify the distribution to form a peak around the best samples is called *distribution focusing*. Distribution focusing is the central idea of CEM (Rubinstein, 2003).

Table 3.1: Effectiveness of the simulation strategies.

	Random	TabuRandom	TabuColorRandom
Average Score	2,069	2,737	3,038
StdDev	322	445	479

When starting CEM, an initial probability distribution is required. Chaslot *et al.* (2008c) used a Gaussian distribution and proposed that for each parameter, the mean μ of the corresponding distribution is equal to the average of the lower and upper bound of that parameter. The standard deviation σ is set to half the difference between the lower and upper bound (cf. Tak, 2010).

3.5 Experiments and Results

In this section we test SP-MCTS in SameGame. All experiments were performed on an AMD64 2.4 GHz computer. Subsection 3.5.1 shows quality tests of the two simulation strategies TabuRandom and TabuColorRandom. Thereafter, the results of manual parameter tuning are presented in Subsection 3.5.2. Subsequently, Subsection 3.5.3 gives the performance of the randomized restarts on a set of 250 positions. In Subsection 3.5.4, it is investigated whether it is beneficial to exhaust all available time at the first move. Next, in Subsection 3.5.5 the parameter tuning by CEM is shown. Finally, Subsection 3.5.6 compares SP-MCTS to the other approaches.

3.5.1 Simulation Strategy

In order to test the effectiveness of the two simulation strategies, we used a test set of 250 randomly generated positions.[6] We applied SP-MCTS without randomized restarts for each position until 10 million nodes were reached in memory. These runs typically take 5 to 6 minutes per position. The best score found during the search is the final score for the position. The constants C and D were set to 0.5 and 10,000, respectively. The results are shown in Table 3.1.

Table 3.1 shows that the TabuRandom strategy has a significantly better average score (i.e., 700 points) than plain random. Using the TabuColorRandom strategy the average score is increased by another 300 points. We observe that a low standard deviation is achieved for the random strategy. In this case, it implies that all positions score almost equally low. The proposed TabuColorRandom strategy has also been successfully applied in Nested Monte-Carlo Search (Cazenave, 2009) and HGSTS (Edelkamp *et al.*, 2010).

3.5.2 Manual Parameter Tuning

This subsection presents the parameter tuning in SP-MCTS. Three different settings were used for the pair of constants (C; D) of Formula 3.1, in order to investigate which balance between exploitation and exploration gives the best results. These

[6] The test set can be found at http://www.personeel.unimaas.nl/maarten-schadd/TestSet.txt

constants were tested with three different time controls on the test set of 250 positions, expressed by a maximum number of nodes. The short time control refers to a run with a maximum of 10^5 nodes in memory. At the medium time control, 10^6 nodes are allowed in memory, and for a long time control 5×10^6 nodes are allowed. We have chosen to use nodes in memory as measurement to keep the results hardware-independent. The parameter pair (0.1; 32) represents *exploitation*, (1; 20,000) performs *exploration*, and (0.5; 10,000) is a balanced setting.

Table 3.2 shows the performance of the SP-MCTS approach for the three time controls. The short time control corresponds to approximately 20 seconds per position. The best results are achieved by exploitation. The score is 2,552. With this setting the search is able to build trees that have on average the deepest leaf node at ply 63, implying that a substantial part of the chosen line of play is inside the SP-MCTS tree. Also, we observe that the other two settings are not generating a deep tree.

For the medium time control, the best results were achieved by using the balanced setting. It scores 2,858 points. Moreover, Table 3.2 shows that the average score of the balanced setting increases most compared to the short time control, viz. 470. The balanced setting is able to build substantially deeper trees than at the short time control (37 vs. 19). An interesting observation can be made by comparing the score of the exploration setting for the medium time control to the exploitation score in the short time control. Even with 10 times the amount of time, exploration is not able to achieve a significantly higher score than exploitation.

The results for the long experiment are that the balanced setting again achieves the highest score with 3,008 points. The deepest node in this setting is on average at ply 59. However, the exploitation setting only scores 200 points fewer than the balanced setting and 100 points fewer than exploration.

Table 3.2: Results of SP-MCTS for different settings.

10^5 nodes (∼20 seconds)	Exploitation (0.1; 32)	Balanced (0.5; 10,000)	Exploration (1; 20,000)
Average Score	2,552	2,388	2,197
Standard Deviation	572	501	450
Average Depth	25	7	3
Average Deepest Node	63	19	8
10^6 nodes (∼200 seconds)	(0.1; 32)	(0.5; 10,000)	(1; 20,000)
Average Score	2,674	2,858	2,579
Standard Deviation	607	560	492
Average Depth	36	14	6
Average Deepest Node	71	37	15
5×10^6 nodes (∼1,000 seconds)	(0.1; 32)	(0.5; 10,000)	(1; 20,000)
Average Score	2,806	3,008	2,901
Standard Deviation	576	524	518
Average Depth	40	18	9
Average Deepest Node	69	59	20

From the results presented we may draw two conclusions. First, it is important to have a deep search tree. Second, exploiting local maxima can be more advantageous than searching for the global maximum when the search only has a small amount of time.

3.5.3 Randomized Restarts

This subsection presents the performance tests of the randomized restarts on the set of 250 positions. We remark that the experiments are time constrained. Each experiment could only use 5×10^5 nodes in total and the restarts distributed these nodes uniformly among the number of searches. It means that a single search can take all 5×10^5 nodes, but that two searches can only use 2.5×10^5 nodes each. We used the exploitation setting (0.1; 32) for this experiment. The results are depicted in Figure 3.2.

Figure 3.2 indicates that already with two searches instead of one, a significant performance increase of 140 points is achieved. Furthermore, the maximum average score of the randomized restarts is at ten threads, which uses 5×10^4 nodes for each search. Here, the average score is 2,970 points. This result is almost as good as the best score found in Table 3.2, but with the difference that the randomized restarts together used one tenth of the number of nodes. After 10 restarts the performance decreases because the generated trees are not deep enough.

3.5.4 Time Control

This subsection investigates whether it is better to exhaust all available time at the initial position or to distribute the time uniformly for every move. Table 3.3 shows the average score on 250 random positions with five different time settings.

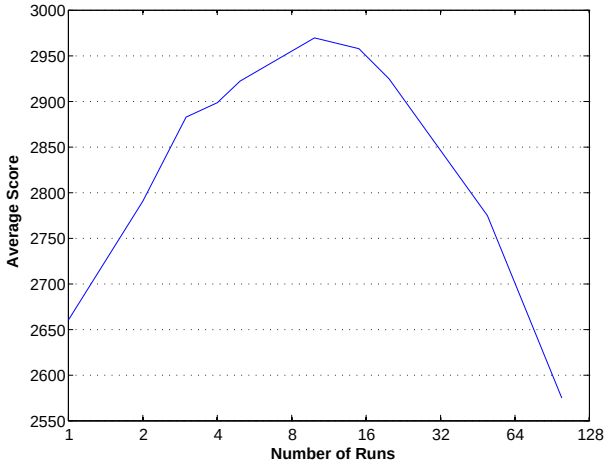


Figure 3.2: The average score for different settings of randomized restarts.

When SP-MCTS is applied for every move, this time is divided by the average game length (64.4). It means that depending on the number of moves, the total search time varies. These time settings are exact in the case that SP-MCTS is applied per game. This experiment was performed in collaboration with Tak (2010).

Table 3.3: Average score on 250 positions using different time control settings (Tak, 2010).

Time in seconds	∼5	∼10	∼20	∼30	∼60
SP-MCTS per game	2,223	2,342	2,493	2,555	2,750
SP-MCTS per move	2,588	2,644	2,742	2,822	2,880

Table 3.3 shows that distributing the time uniformly for every move is the better approach. For every time setting a higher score is achieved when searching per move. The difference in score is largest for 5 seconds, and smallest for 60 seconds. It is an open question whether for longer time settings it may be beneficial to exhaust all time at the initial position.

3.5.5 CEM Parameter Tuning

In the next series of experiments we tune SP-MCTS with CEM. The experiments have been performed in collaboration with Tak (2010). The following settings for CEM were used. The sample size is equal to 100, the number of elite samples is equal to 10. Each sample plays 30 games with 1 minute thinking time for each game. The 30 initial positions are randomly generated at the start of each iteration. The fitness of a sample is the average of the scores of these games. The five parameters tuned by CEM are presented in Table 3.4. C, D, T and W were described in Subsection 3.3.1. The ϵ parameter was described in Subsection 3.3.2. The CEM-tuned parameters differ significantly from the manually tuned ones. For more results on tuning the parameters, we refer to Tak (2010).

Table 3.4: Parameter tuning by CEM (Tak, 2010).

Parameter	Manual	CEM per game	CEM per move
C	0.1	5.96	4.31
D	32	67.98	96.67
T	10	13	11
W	0.02	0.49	0.28
ϵ	0.003	0.00007	0.00007[7]

To determine the performance of the parameters found by CEM an independent test set of 250 randomly created positions was used. Five different time settings were investigated. Table 3.5 shows the results of the CEM experiments. Here, the search time is distributed uniformly for every move.

[7] This parameter was not tuned again because it was obvious that the optimal weight is close to or equal to zero.

Table 3.5: Average scores of CEM tuning (Tak, 2010).

Time in seconds	∼5	∼10	∼20	∼30	∼60
Manual tuned	2,588	2,644	2,742	2,822	2,880
Average Depth	22.7	27.4	30.3	32.8	35.9
Average Deepest Node	31.8	36.8	39.1	41.4	44.3
CEM tuned	2,652	2,749	2,856	2,876	2,913
Average Depth	4.9	5.4	6.2	6.8	9.1
Average Deepest Node	9.0	10.2	12.2	13.5	19.2

Table 3.5 shows that for every time setting CEM is able to improve the score. This demonstrates the difficulty of finding parameters manually in a high-dimensional parameter space. The CEM-tuned parameters are more explorative than the manually tuned parameters. This difference may be due to the fact that the CEM parameters are tuned for the "per move" time control setting. The average depth and average deepest node achieved by the CEM parameters are closest to the results of the balanced setting in Table 3.2.

3.5.6 Comparison on the Standardized Test Set

Using two hours per position, we tested SP-MCTS on the standardized test set. We tested three different versions of SP-MCTS, subsequently called SP-MCTS(1), SP-MCTS(2), and SP-MCTS(3). SP-MCTS(1) builds one large tree at the start and uses the exploitation setting (0.1; 32) and randomized restarts, which applied 1,000 runs using 100,000 nodes for each search thread. SP-MCTS(2) uses the same parameters as SP-MCTS(1), but distributes its time per move. SP-MCTS(3) distributes its time per move and uses the parameters found by CEM. Table 3.6 compares SP-MCTS with other approaches, which were described in Subsection 3.1.3.

SP-MCTS(1) outperformed DBS on 11 of the 20 positions and was able to achieve a total score of 73,998. This was the highest score on the test set at the point of our publication (cf. Schadd *et al.*, 2008c). SP-MCTS(2) scored 76,352 points, 2,354 more than SP-MCTS(1). This shows that it is important to distribute search time for every move. SP-MCTS(3) achieved 78,012 points, the third strongest method at this point of time. All SP-MCTS versions are able to clear the board for all 20 positions.[8] This confirms that a deep search tree is important for SameGame as shown in Subsection 3.5.2.

The two highest scoring programs (1) SPURIOUS AI and (2) HGSTS achieved more points than SP-MCTS. We want to give the following remarks on these impressive scores. (1) SPURIOUS AI is memory intensive and it is unknown what time settings were used for achieving this score. (2) HGSTS utilized the graphics processing unit (GPU), was optimized for every position in the standardized test set, and applied our TabuColorRandom strategy. Moreover, the scores of HGTS were not independently verified to be correct.

[8] The best variations can be found at the following address:
http://www.personeel.unimaas.nl/maarten-schadd/SameGame/Solutions.html

Table 3.6: Comparing the scores on the standardized test set.

Position no.	DBS	**SP-MCTS(1)**	**SP-MCTS(2)**	MC-RWS
1	2,061	2,557	2,969	2,633
2	3,513	3,749	3,777	3,755
3	3,151	3,085	3,425	3,167
4	3,653	3,641	3,651	3,795
5	3,093	3,653	3,867	3,943
6	4,101	3,971	4,115	4,179
7	2,507	2,797	2,957	2,971
8	3,819	3,715	3,805	3,935
9	4,649	4,603	4,735	4,707
10	3,199	3,213	3,255	3,239
11	2,911	3,047	3,013	3,327
12	2,979	3,131	3,239	3,281
13	3,209	3,097	3,159	3,379
14	2,685	2,859	2,923	2,697
15	3,259	3,183	3,295	3,399
16	4,765	4,879	4,913	4,935
17	4,447	4,609	4,687	4,737
18	5,099	4,853	4,883	5,133
19	4,865	4,503	4,685	4,903
20	4,851	4,853	4,999	4,649
Total:	72,816	**73,998**	**76,352**	76,764

Position no.	Nested MC	**SP-MCTS(3)**	SPURIOUS AI	HGSTS
1	3,121	2,919	3,269	2,561
2	3,813	3,797	3,969	4,995
3	3,085	3,243	3,623	2,858
4	3,697	3,687	3,847	4,051
5	4,055	4,067	4,337	4,633
6	4,459	4,269	4,721	5,003
7	2,949	2,949	3,185	2,717
8	3,999	4,043	4,443	4,622
9	4,695	4,769	4,977	6,086
10	3,223	3,245	3,811	3,628
11	3,147	3,259	3,487	2,796
12	3,201	3,245	3,851	3,710
13	3,197	3,211	3,437	3,271
14	2,799	2,937	3,211	2,432
15	3,677	3,343	3,933	3,877
16	4,979	5,117	5,481	6,074
17	4,919	4,959	5,003	5,166
18	5,201	5,151	5,463	6,044
19	4,883	4,803	5,319	5,019
20	4,835	4,999	5,047	5,175
Total:	77,934	**78,012**	84,414	84,718

3.6 Chapter Conclusions and Future Research

In this chapter we proposed a new MCTS variant called Single-Player Monte-Carlo Tree Search (SP-MCTS). We adapted MCTS by two modifications resulting in SP-MCTS. The modifications are (1) in the selection strategy and (2) in the backpropagation strategy. Below we provide five observations and one conclusion.

First, we observed that our TabuColorRandom strategy significantly increased the score of SP-MCTS in SameGame. Compared to the pure random play-outs, an increase of 50% in the average score is achieved. The proposed TabuColorRandom strategy has also been successfully applied in Nested Monte-Carlo Search (Cazenave, 2009) and HGSTS (Edelkamp *et al.*, 2010). Second, we observed that exploiting works better than exploring at short time controls. At longer time controls a balanced setting achieves the highest score, and the exploration setting works better than the exploitation setting. However, exploiting the local maxima still leads to comparable high scores. Third, with respect to the randomized restarts, we observed that for SameGame combining a large number of small searches can be more beneficial than performing one large search. Fourth, it is better to distribute search time equally over the consecutive positions than to invest all search time at the initial position. Fifth, CEM is able to find better parameter values than manually tuned parameter values. The parameters found by CEM resemble a balanced setting. They were tuned for applying SP-MCTS for every move, causing that deep trees are less important.

The main conclusion is that we have shown that MCTS is applicable to a one-player deterministic perfect-information game. Our variant, SP-MCTS, is able to achieve good results in SameGame. Thus, SP-MCTS is a worthy alternative for puzzles where a good admissible estimator cannot be found.

There are two directions of future research for SP-MCTS. The first direction is to test several enhancements in SP-MCTS. We mention two of them. (1) The selection strategy can be enhanced with RAVE (Gelly and Silver, 2007) or progressive widening (Chaslot *et al.*, 2008d; Coulom, 2007a). (2) This chapter demonstrated that combining small searches can achieve better scores than one large search. However, there is no information shared between the searches. This can be achieved by using a transposition table, which is not cleared at the end of a small search. The second direction is to apply SP-MCTS to other domains. For instance, we could test SP-MCTS in puzzles such as Morpion Solitaire and Sudoku (Cazenave, 2009) and Single-Player General Game Playing (Méhat and Cazenave, 2010). Other classes of one-player games, with non-determinism or imperfect information, could be used as test domain for SP-MCTS as well.