HAL
archives-ouvertes.fr

Multiple Overlapping Tiles for Contextual Monte Carlo Tree Search

Arpad Rimmel, Fabien Teytaud

▶ **To cite this version:**

Arpad Rimmel, Fabien Teytaud. Multiple Overlapping Tiles for Contextual Monte Carlo Tree Search. Evostar, Apr 2010, Istanbul, Turkey. 2010. <inria-00456422v1>

HAL Id: inria-00456422

https://hal.inria.fr/inria-00456422v1

Submitted on 15 Feb 2010 (v1), last revised 16 Jan 2011 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Multiple Overlapping Tiles for Contextual Monte Carlo Tree Search

Arpad Rimmel[1] and Fabien Teytaud[1]

TAO (Inria), LRI, UMR 8623(CNRS - Univ. Paris-Sud), bat 490 Univ. Paris-Sud
91405 Orsay, France

Abstract. Monte Carlo Tree Search is a recent algorithm that achieves more and more successes in various domains. We propose an improvement of the Monte Carlo part of the algorithm by modifying the simulations depending on the context. The modification is based on a reward function learned on a tiling of the space of Monte Carlo simulations. The tiling is done by regrouping the Monte Carlo simulations where two moves have been selected by one player. We show that it is very efficient by experimenting on the game of Havannah.

1 Introduction

Monte Carlo Tree Search (MCTS) [5] is a recent algorithm for taking decisions in a discrete, observable, uncertain environment with finite horizon that can be described as a reinforcement learning algorithm. This algorithm is particularly interesting when the number of states is huge. In this case, classical algorithms like Minimax and Alphabeta [7], for two-player games, and Dynamic Programming [2], for one-player games, are too time-consuming or not efficient. As MCTS explores only a small relevant part of the whole problem, this allows it to obtain good performance in such situations. This algorithm achieved particularly good results in two-player games like computer Go or Havannah. But this algorithm was also successfully applied on one-player problems like the automatic generation of libraries for linear transforms [4] or active learning [8].

The use of Monte Carlo simulations to evaluate a situation is an advantage of the MCTS algorithm; it gives an estimation without any knowledge of the domain. However, it can also be a limitation. The underlying assumption is that decisions taken by an expert are uniformly distributed in the whole space of decisions. This is not true in most of the cases. In order to address this problem, one can add expert knowledge in the Monte Carlo simulations as proposed in [3]. However, this solution is application-dependent and limited because all the different situations have to be treated independently.

In this paper, we present a first step to solve this problem in a generic way. We introduce a modification of the Monte Carlo simulations that allows them to be automatically modified depending on the context: Contextual Monte Carlo (CMC) simulations. We show that it improves the performance for the game of Havannah. In order to do that, we learn the reward function on a tiling of the

space of Monte Carlo simulations and use this function to modify the following simulations. The idea is to group simulations where two particular actions have been selected by the same player. Then, we learn the average reward on those sets. And finally, we try to reach simulations from sets associated to a high reward. This modification is generic and can be applied to two-player games as well as one-player games. To the extent of our knowledge, this is the first time a generic and automatic way of adapting the Monte Carlo simulations in the MCTS algorithm has been proposed.

We first present reinforcement learning, the principle of the Monte Carlo Tree Search algorithm and the principle of tiling. Then, we introduced those new simulations: CMC simulations. Finally, we present the experiments and conclude.

2 Value-Based Reinforcement Learning

In a reinforcement learning problem, an agent will choose *actions* in an environment described by *states* with the objective of maximizing a long-term *reward*. The agent will act based on his previous trials (*exploitation*) and try new choices (*exploration*).

Let S be a set of states. Let A be a set of actions. Let $R \subset \mathbb{R}$ be a set of rewards.

At each time $t \in 1, ..., T$, the current state $s_t \in S$ is known. After an action $a_t \in A$ is chosen, the environment returns the new state s_{t+1} and the reward $r_{t+1} \in R$.

The goal is to find a policy function $\pi : S \to A$ that maximizes the cumulative reward R_t for each t:

$$R_t = \sum_{k=t+1}^{T} r_k$$

In value-based reinforcement learning, an intermediate *value function* is learned to compute the policy. The value function $V^\pi(s)$ is the expected cumulative reward starting state s and following the policy π thereafter.

$$V^\pi(s) = \mathbb{E}_\pi[R_t | s_t = s]$$

This function is not known and will be empirically evaluated: $\hat{V}^\pi$.

While there is some time left, the algorithm will iterate two different steps:

- *utilization* of the empirical estimation of the value function. $\hat{V}^\pi$ is used in order to choose the actions (often based on a compromise between exploration and exploitation).
- *update* of the empirical estimation of the value function. $\hat{V}^\pi$ is modified based on the rewards obtained.

This is known as the *policy iteration* process [10].

The utilization and update of $\hat{V}^{\pi}$ can be done by different algorithms. For example, [11] propose the $TD(\lambda)$ algorithm in order to do the update. A classical utilization of $\hat{V}^{\pi}$ is the ϵ-greedy algorithm.

3 Monte Carlo Tree Search

The principle of MCTS is to construct a highly unbalanced tree representing the future by using a bandit formula and to combine it with Monte Carlo simulations to evaluate the leaves.

3.1 Bandits

A classical k-armed bandit problem is defined as follows:

- A finite set $J = \{1, \dots, k\}$ of arms is given.
- Each arm $j \in J$ is associated to an unknown random variable X_j with an unknown expectation μ_j.
- At each time step $t \in \{1, 2, \dots\}$:
  - the algorithm chooses $j_t \in J$ depending on $(j_1, \dots, j_{t-1})$ and $(r_1, \dots, r_{t-1})$.
  - Each time an arm j is selected, the bandit gives a reward r_t, which is a realization of X_{j_t}.

The goal of the problem is to minimize the so-called *regret*: the loss due to the fact that the algorithm will not always chose the best arm.

Let $T_j(n)$ the number of times an arm has been selected during the first n steps. The *regret* after n steps is defined by

$$\mu^* n - \sum_{j=1}^{k} \mu_j \mathbb{E}[T_j(n)] \text{ where } \mu^* = \max_{1 \leq i \leq k} \mu_i$$

[1] achieve a logarithmic regret (it has been proved that this is the best regret obtainable in [6]) uniformly over time with the following algorithm: first, tries one time each arm; then, at each step, selects the arm j that maximizes

$$\bar{x_j} + \sqrt{\frac{2ln(n)}{n_j}} \tag{1}$$

$\bar{x_j}$ is the average reward for the arm j.

n_j is the number of times the arm j has been selected so far.

n is the overall number of trials so far.

This formula consists in choosing at each step the arm that has the highest upper confidence bound. It is called the UCB formula.

3.2 Monte Carlo Tree Search

The MCTS algorithm constructs in memory a subtree $\hat{T}$ of the global tree T representing the problem in its whole (see algorithm 1 (left) and figure 1 (left)).

The construction of the tree is done by the repetition while there is some time left of 3 successive steps: *descent, evaluation, growth.*

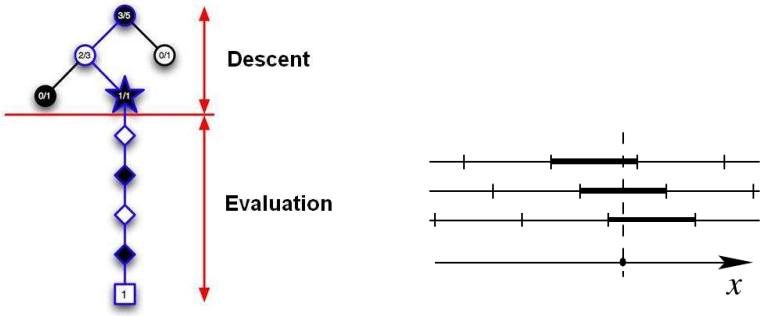

Fig. 1. Left. Illustration of the Monte Carlo Tree Search algorithm from a presentation of Sylvain Gelly. **Right.**Illustration of 3 overlapping tilings from the article [9].

Descent. The descent in $\hat{T}$ is done by considering that taking decision is a k-armed bandit problem. We use the formula 1 to solve this problem. In order to do that, we suppose that the necessary information is stored for each node. Once a new node has been reached, we just repeat the same principle until we reached a situation S outside of $\hat{T}$.

Evaluation. Now that we have reached S and that we are outside of $\hat{T}$, there is no more information available to take a decision. As we are not at a leaf of T, we can not directly evaluate S. Instead, we use a Monte Carlo simulation (taking decisions uniformly until a final state is reached) to have a value for S.

Growth. We add the node S to $\hat{T}$. We update the information of S and of all the situations encountered during the descent with the value obtained with the Monte Carlo evaluation.

3.3 Monte Carlo Tree Search as a Reinforcement Learning Algorithm

The tree representing the problem solved by MCTS can be described as a reinforcement learning problem with the following correspondence: states $\sim$ nodes of the tree, actions $\sim$ branches of the tree, rewards $\sim$ results at the terminal nodes of the tree.

The MCTS algorithm is a value-based reinforcement learning algorithm with a UCB policy. The value $\hat{V}^{UCB}(s)$ is stored in the node corresponding to the state s. It corresponds to the average reward for the situation s so far.

The *utilization* part of the algorithm is defined as follows: the action a chosen in the state s is selected according to

$$\begin{cases} argmax_a(\hat{V}^{UCB}(s_a) + \sqrt{\frac{2ln(n_s)}{n_{s_a}}}) & \text{if } s \in \hat{T} \\ mc(s) & \text{otherwise} \end{cases} \tag{2}$$

s_a is the situation reached from s after choosing the action a.

n_{s_a} is the number of times the action a has been selected so far from the situation s.

n_s is the overall number of trials so far for situation s.

$mc(s)$ returns an action uniformly selected among all the possible actions from the state s.

When s is in $\hat{T}$, the action is chosen according to the UCB formula 1 (descent part). When s is outside of $\hat{T}$, the action is chosen uniformly among the possible actions (evaluation part).

The *update* part of the reinforcement learning algorithm is done after a final state is reached and a new node has been added to $\hat{T}$. The reward is the value r associated to the final state. For all states $s \in \hat{T}$ that were reached from the initial state to the final state

$$\hat{V}^{UCB}(s) \leftarrow \hat{V}^{UCB}(s) + \frac{r}{n_s}$$

4 Tile Coding

When the number of states is very large or even infinite in the case of continuous parameters, it is necessary to use a *function approximation* to learn the value function.

In *tile coding* (see [10]), the space D is divided into *tiles*. Such a partition is called a *tiling*. It is possible to use several overlapping tilings. A weight is associated by the user to each tile. The value of a point is given by the sum of the weight of all the tiles in which the point is included. A representation of 3 overlapping tilings for a problem with one continuous dimension is given on figure 1 (right).

Tile coding will lead to a piecewise constant approximation of the value function:

$$\forall p \in D, \exists z \in \mathbb{R} \text{ such that, } \forall p' \in D \wedge distance(p, p') < z, \hat{V}^{\pi}(p) = \hat{V}^{\pi}(p')$$

5 Contextual Monte Carlo

In this section, we present how we learn the reward function on the space of Monte Carlo simulations by defining a tiling on this space. Then, we explain how we use this function to improve the following Monte Carlo simulations.

5.1 A New Tiling on the Space of Monte Carlo Simulations

We consider a planning problem, the goal is to maximize the reward. We consider that a Monte Carlo Tree Search algorithm is used to solve this problem. Let G be the set of the possible actions. We focus on the space of Monte Carlo simulations E_{MC}. A Monte Carlo simulation is the sequence of moves from outside the tree $\hat{T}$ until a final state. Each Monte Carlo simulation is therefore associated to a reward. We define the tiles $L(a_1, a_2)$ on E_{MC} where $(a_1, a_2) \in G^2$. $L(a_1, a_2)$ is composed of all the simulations containing a_1 and a_2 and where a_1 and a_2 has been selected by one player P.

$$L = \{\{\text{sim } s \text{ such that } a_1 \in s \wedge a_2 \in s \wedge P_s(a_1) \wedge P_s(a_2)\}; (a_1, a_2) \in G^2\} \quad (3)$$

We define $\hat{V}_{CMC}$: the empirical reward function based on L. In order to learn the value $\hat{V}_{CMC}(a_1, a_2)$, each time that a simulation s is rewarded with a value r, we update the values for each tiles containing s.

For each $L(a_1, a_2)$ such that $s \in L(a_1, a_2)$,

$$\hat{V}^{CMC}(a_1, a_2) \leftarrow \hat{V}^{CMC}(a_1, a_2) + \frac{r}{n_{CMC}(a_1, a_2)}$$

$n_{CMC}(a_1, a_2)$ is the number of times a simulation in $L(a_1, a_2)$ has been played.

$\hat{V}^{CMC}(a_1, a_2)$ corresponds to the estimated reward for any simulation in which two particular actions have been selected. If this value is high, it means that each time the player manages to play a_1 and a_2 in a simulation, there is a high chance that the simulation will give a high reward for that player.

5.2 Improving Monte Carlo Simulations

We focus on tiles where the estimated reward is high (superior to a user-defined threshold B). The policy should try to reach simulations in those tiles. In order to do that, if a tile associated with two actions a_1 and a_2 and with an estimated value inferior to B exists and if one player previously selected one of the actions, we will then select the other. In fact, if several such tiles exist, we will select the action that will lead to the simulation from the tile with the highest average reward.

As the policy for situations in $\hat{T}$ is already efficient, we modify the policy for situations outside $\hat{T}$.

The utilization part previously defined in Eq. 2 is now defined as follows: The action a chosen in the state s is selected according to

$$\begin{cases} argmax_a(\hat{V}^{UCB}(s_a) + \sqrt{\frac{2ln(n_s)}{n_{s_a}}}) & \text{if } s \in \hat{T} \\ \begin{cases} cmc(s) \text{ if } random() < prob \\ mc(s) \text{ otherwise} \end{cases} & \text{otherwise} \end{cases} \quad (4)$$

prob is a parameter between 0 and 1. It corresponds to the probability of choosing $cmc(s)$ instead of $mc(s)$. $random()$ is a random number generated between 0 and 1. $cmc(s)$ is defined as follows:

$$cmc(s) = argmax_{a,a \in E_s}(S(\hat{V}^{CMC}(a,b)))$$

E_s is the set of the possible actions in the state s. b is the previous move played by the same player. S is a threshold function with a threshold at B, formally defined as follows:

$$S(x) = \begin{cases} x & \text{if } x > B \\ 0 & \text{otherwise} \end{cases} \quad (5)$$

In order to keep the diversity in Monte Carlo simulation (the importance of diversity is discussed in [3]), we apply this modification with a certain probability *prob*. This probability is defined by the user.

The resulting algorithm is given in algorithm 1 (right).

Algorithm 1 Left. MCTS(s) **Right.** CMCTS(s) //s a situation

Initialization of $\hat{T}$, $\hat{V}^{UCB}$, n
while there is some time left **do**
  $s' = s$
  Initialization of $game$
  //$DESCENT$//
  while s' in $\hat{T}$ and s' not terminal **do**
    s' = reachable situation chosen according to the UCB formula (1)
    $game = game + s'$
  end while
  $S = s'$
  //$EVALUATION$//
  while s' is not terminal **do**
    $s' = mc(s')$
  end while
  $r = result(s')$
  //$GROWTH$//
  $\hat{T} = \hat{T} + S$
  for each s in $game$ **do**
    $n_s \leftarrow n_s + 1$
    $\hat{V}^{UCB}(s) \leftarrow \hat{V}^{UCB}(s) + \frac{r}{n_s}$
  end for
end while

Initialization of $\hat{T}$, $\hat{V}^{UCB}$, n, $\hat{V}^{CMC}$, n^{CMC}
while there is some time left **do**
  $s' = s$
  Initialization of $game$,$gamemc$
  //$DESCENT$//
  while s' in $\hat{T}$ and s' not terminal **do**
    s' = reachable situation chosen according to the UCB formula (1)
    $game = game + s'$
  end while
  $S = s'$
  //$EVALUATION$//
  while s' is not terminal **do**
    if $random() < prob$ **then**
      $s' = cmc(s')$
    else
      $s' = mc(s')$
    end if
    $gamemc \leftarrow gamemc + s'$
  end while
  $r = result(s')$
  //$GROWTH$//
  $\hat{T} = \hat{T} + S$
  for each s in $game$ **do**
    $n_s \leftarrow n_s + 1$
    $\hat{V}^{UCB}(s) \leftarrow \hat{V}^{UCB}(s) + \frac{r}{n_s}$
  end for
  for each $(P(a_1), P(a_2))$ in s', P being one player **do**
    $n^{CMC}(a_1,a_2) \leftarrow n^{CMC}(a_1,a_2) + 1$
    $\hat{V}^{CMC}(a_1,a_2) \leftarrow \hat{V}^{CMC}(a_1,a_2) + \frac{r}{n^{CMC}(a_1,a_2)}$
  end for
end while

6 Experiments

We have tested the effect of contextual Monte Carlo simulations on the game of Havannah. In this section, we first describe the game of Havannah and then give our results.

6.1 Havannah

Havannah is a two-player board game recently introduced in the community of computer game [12]. Invented by Christian Freeling, the game of Havannah is played on an hexagonal board with hexagonal locations, and different board sizes (size of 8 or 10 is usual). The rules are really simple. White player starts, and after that each player plays alternately by putting a stone in an empty location. If there is no any empty location free, and if no player has won yet, then the game is a draw. To win, a player has to realize :

- a ring, which is a loop around one or more cells (empty or not, occupied by black or white stones).
- a bridge, which is a continuous string of stones connecting to one of the six corners to another one.
- a fork, which is a continuous string of stones linking three edges of the board (corner locations are not considered as belonging to the edges).

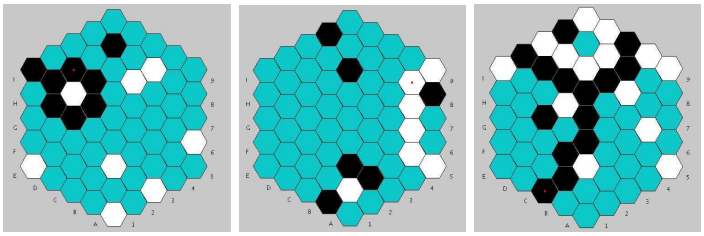

Fig. 2. Three finished games: a ring (a loop, by black), a bridge (linking two corners, by white) and a fork (linking three edges, by black).

Fig. 2 presents these three ways to win a game.

The game of Havannah is known as hard for computers for different reasons. First, there is a large action space, for instance, in size 10, there are 271 possible moves for the first player. Second, there is no pruning rule for reducing the number of possible moves. Another important reason is that, there is no natural evaluation function. A such function is really useful, in the sense that, it gives a very good evaluation of a position, as, for instance in chess. And finally, a last reason is the lack of patterns in the game of Havannah. A pattern is an expert

knowledge which give information on a move or a position. For instance, in chess, it is always good having his king in a safe place, therefore, a pattern could be castling if it is possible.

6.2 Results

In this section we experiment our Havannah program with the CMC improvement against the same player without this improvement. The reward is 1 (if the situation is a won game) or 0 (if the situation is a loss). The experiments are done with 1000 simulations per move for each player.

We study the impact of the two parameters *prob* and B. *prob* defines the percentage of time our modification will be applied (see Algo. 1). B defines which tiles we want to reach (see Eq. 5). The results are given on figure 3 and are commented below.

First, we see that the utilization of CMC is efficient. It leads to 57% of victory against the base version for $prob = 35$ and $B = 0$.

The first figure 3 (left) shows the effect of changing B for a fixed value of $prob = 75\%$. The winning percentage starts at 56% for $B = 0$ and is stable until $B = 60$ where it starts going down to 50% for $B = 100$. When B is too high, CMC is used less often and therefor the results are worse. When B is low, it means that we will select the best tile even if all the possible tiles have a bad average reward. It seems that this is never worst than playing randomly. In the following, we use $B = 0$.

On the second figure 3 (right), we modify the value of *prob* while keeping B fixed. When *prob* is too high, the diversity of the Monte Carlo simulations is not preserved and the results are worse. On the other hand, if *prob* is too low, the modification has not enough effect. There is a compromise between this two properties.

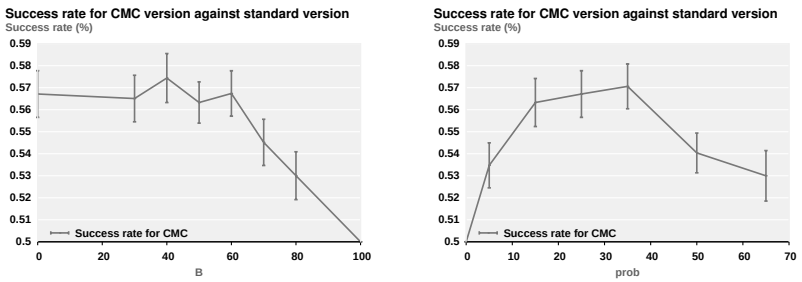

Fig. 3. Left. Winning percentage for the CMC version against the basic version with $prob = 35$. **Right.**Winning percentage for the CMC version against the basic version with $B = 0$.

7 Conclusion

We presented a domain-independent improvement of the Monte Carlo Tree Search algorithm. This was done by modifying the MC simulations by using a reward function learned on a tiling of the space of MC simulations: CMC. To the extent of our knowledge, this is the first time that an automatic modification of the Monte Carlo has been proposed.

It achieves very good results for the game of Havannah with a winning percentage of 57% against the version without CMC.

It is, for the moment, tested only in the case of one example of two-player game. An immediate perspective of this work is to experiment CMC on other problems.

It is possible to apply an infinite amount of tilings to the space of Monte Carlo simulations. We proposed a successful specific one but others can surely be found as we used only a small part of the information contained in this space. In the future, we intend to give a formal description of the learning in the space of Monte Carlo simulations.

References

1. P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47(2/3):235–256, 2002.
2. D. P. Bertsekas. Neuro-dynamic programming. In *Encyclopedia of Optimization*, pages 2555–2560. 2009.
3. G. Chaslot, C. Fiter, J.-B. Hoock, A. Rimmel, and O. Teytaud. Adding expert knowledge and exploration in Monte-Carlo Tree Search. In *Advances in Computer Games*, Pamplona Espagne, 2009. Springer.
4. F. De Mesmay, A. Rimmel, Y. Voronenko, and M. Püschel. Bandit-Based Optimization on Graphs with Application to Library Performance Tuning. In *International Conference on Machine Learning*, Montréal Canada, 2009.
5. L. Kocsis and C. Szepesvari. Bandit-based monte-carlo planning. In *European Conference on Machine Learning 2006*, pages 282–293, 2006.
6. T. Lai and H. Robbins. Asymptotically efficient adaptive allocation rules. *Advances in Applied Mathematics*, 6:4–22, 1985.
7. J. Pearl. *Heuristics. Intelligent search strategies for computer problem solving.* Addison-Wesley, 1984.
8. P. Rolet, M. Sebag, and O. Teytaud. Optimal active learning through billiards and upper confidence trees in continous domains. In *Proceedings of the European Conference on Machine Learning*, 2009.
9. E. A. Sherstov and P. Stone. Function approximation via tile coding: Automating parameter choice. In *of Lecture Notes in Artificial Intelligence*, pages 194–205. Springer Verlag, 2005.
10. R. Sutton and A. Barto. *Reinforcement learning: An introduction.* MIT Press., Cambridge, MA, 1998.
11. R. S. Sutton. Learning to predict by the methods of temporal differences. In *Machine Learning*, pages 9–44. Kluwer Academic Publishers, 1988.
12. F. Teytaud and O. Teytaud. Creating an Upper-Confidence-Tree program for Havannah. In *Advances in Computer Games 12*, Pamplona Espagne, 2009.