PAPER   *Special Section on Foundations of Computer Science —New Trends in Theory of Computation and Algorithm—*

Efficient Sampling Method for Monte Carlo Tree Search Problem

Kazuki TERAOKA[†], *Nonmember*, **Kohei HATANO**[††a]**,** *and* **Eiji TAKIMOTO**[††]**,** *Members*

SUMMARY   We consider Monte Carlo tree search problem, a variant of Min-Max tree search problem where the score of each leaf is the expectation of some Bernoulli variables and not explicitly given but can be estimated through (random) playouts. The goal of this problem is, given a game tree and an oracle that returns an outcome of a playout, to find a child node of the root which attains an approximate min-max score. This problem arises in two player games such as computer Go. We propose a simple and efficient algorithm for Monte Carlo tree search problem.
key words: *Monte Carlo tree search, random sampling, game, UCT*

1.   Introduction

Computer programs of Go are becoming stronger by using Monte Carlo tree search methods [1]–[3]. Among them, MoGo [4], [5] won a human professional player on the small 9x9 board. After this breakthrough by Mogo, Monte Carlo tree search methods are employed by most of competitive computer Go programs, and some of the state of the art programs reached the rank of 4 or higher dans on the KGS Go server with the full-size board.

   In general, a two player games between the player and the opponent, including Go, is modeled as a game tree, where each node and each edge correspond to a position and a move, respectively. Each leaf in a game tree is assigned a score given by a pre-defined score function. Given a node in the game tree (current position), the problem of choosing the edge (the next move of the player) is often called the best move search problem. A generic method to solve the best move search problem is Min-Max search.

   Min-Max search chooses the edge connected with the min-max node, where the min-max node is the node achieving the min-max score. The performance of the game program using Min-Max search depends on its score function.

   Monte Carlo tree search is a variant of Min-Max search in which a score of each leaf is determined by the winning probability of the player. The winning probability (of the player) at the game position is the probability that the player wins from the game position under the condition that both the player and the opponent repeatedly choose their moves uniformly randomly from their available moves. The winning probability can be estimated from (random) playouts, where playout is a sequence of random moves of the player

and the opponent from the game position to the end of the game. Monte Carlo tree search is different from standard Min-Max search methods in that scores of leaves are not explicitly given but, instead, can be approximately estimated via playouts. We call this problem Monte Carlo tree search problem.

   A naive method to solve Monte Carlo tree search problem is to estimate scores of all leaves by sufficiently many playouts and then performing min-max search based on the estimates of scores. The number of playouts is $\tilde{O}(L/\varepsilon^2)$, where L is the number of leaves in the game tree and ε is the precision parameter for the min-max value (In $\tilde{O}$ notations, we neglect polynomials of logarithmic factors). This method, however, is not efficient enough. In general, winning probabilities of nodes or leaves are diverse, so, estimation of some winning probabilities can be rough and uniformly small precision ε is not necessary.

1.1   Related Researches

UCT algorithm [6], one of Monte Carlo tree search methods, adaptively estimate winning probability of leaves by recursively using UCB algorithm [7] which is originally designed for the multi-armed bandit problem. An advantage of UCT is that it adaptively reduces playouts when winning probabilities are far from uniform. In fact, many of recent Monte Carlo search based GO programs employ UCT. On the other hand, analysis of UCT is done only in the asymptotic sense and there is no theoretical analysis using finite sample or analysis of sample complexity of getting ε-approximate min-max solutions.

   In this paper, we propose an new algorithm for Monte Carlo tree search problem and prove its sample complexity. Our algorithm is designed through improvement of a simple naive method, while UCT is based on online prediction algorithms. As a result, analysis of our algorithms is significantly simpler and more concise. Under the assumption that the diversities among winning probabilities are parametrized as Δ, our algorithm finds an ε-approximate min-max solution using $\tilde{O}(L \min(1/\varepsilon^2, 1/\Delta^2))$ playouts, which can be significantly smaller than the naive bound.

   We note a technical difference between our setting and typical settings of Monte Carlo tree search literature. Typical methods in the literature, given an initial game tree, iteratively grow the tree by replacing potentially important leaves with stumps, i.e., trees with depth 1. Major meth-

ods based on UCT [6] work in this setting (see the details in [1]). On the other hand, we consider a much simpler setting where we are given a relatively large game tree in advance and we do not grow the tree. Instead, we try to prune the tree as much as possible.

1.2 Organization of the Paper

This paper is organized as follows. In Sect. 2, we define basic notations and formalize Monte Carlo tree search problem. In Sect. 3, we propose our algorithm and give the analysis of the algorithm in Sect. 4. In Sect. 5, we show preliminary experimental results. We conclude our result in Sect. 6.

2. Preliminaries

In this section, we describe basic definitions. Given a two-player game, a game tree is the directed tree such that each node corresponds to a position of the game and each directed edge from a node corresponds to a feasible move of the player (or the opponent) at the position and the node to which the directed edge is going corresponds to the resulting game position. Figure 1 shows an illustration of a game tree. The root node of a game tree represents the starting position of the game. In general, the size of a game tree could be too large, e.g., when the game is Go or Shogi. So, a given game tree is often a subtree of the whole game tree, where the root node corresponds to a current game position and the depth of the subtree is some fixed constant. We also assume that we are given some score function which assigns a score to each leaf in the game tree.

2.1 Min-Max Search

Min-Max search is a generic method for choosing the best move in the game tree. Given a game tree in which scores of leaves are specified, the score of each node is defined recursively as follows: (i) score of node u is the maximum score among those of child nodes of u, if u corresponds to a position of the player (in other words, depth of u is even), (ii) score of node u is the minimum score among those of child nodes of u, if u corresponds to a position of the opponent (in other words, depth of u is odd). The min-max score of the game tree is the score of the root node. Min-max search

finds an edge such lying between the root node and a node having the min-max score.

2.2 Monte Carlo Tree Search

Monte Carlo tree search method is a variant of Min-Max search method in which the score of each leaf is specified by the winning probability at the leaf. Here, the winning probability is defined as the probability that the player wins the game from the position when both the player and the opponent alternately choose their moves uniformly randomly among available moves. Since exact values of the winning probabilities are not given, Monte Carlo tree search method estimates the winning probability of each leaf by a random sampling procedure called playout. A playout is a sequence of moves which are randomly chosen by both the player and the opponent from the position (corresponding the leaf) to an end of the game.

2.3 Some Inequalities

Proposision 1 (Hoeffding's inequality [8]). *Let* $X_1, \ldots, X_T$ *be* $[0, 1]$-*valued independent random variable such that* $\mathbf{E}(X_t) = \mu$ *for* $t = 1, \ldots, T$. *Then, for any* $c > 0$,

$$\mathbf{P}\left[\frac{1}{T} \sum_{t=1}^{T} X_t > \mu + c\right] \leq \exp\left(-2c^2 T\right) \text{ and}$$

$$\mathbf{P}\left[\frac{1}{T} \sum_{t=1}^{T} X_t < \mu - c\right] \leq \exp\left(-2c^2 T\right).$$

Proposision 2 (Union bound). *For any events* A *and* B, $\mathbf{P}[A \cup B] \leq \mathbf{P}[A] + \mathbf{P}[B]$.

2.4 Computation Model for Monte Carlo Tree Search

In this paper, we model the procedure of playouts as oracle calls. More precisely, we define O_ℓ for each leaf ℓ. When the oracle O_ℓ is called, O_ℓ returns 1 with probability μ_ℓ and returns 0 with probability $1 - \mu_\ell$, respectively, where μ_ℓ is the winning probability of the player at leaf ℓ. Then, a playout at leaf ℓ corresponds to a call of oracle O_ℓ and estimating the winning probability at leaf ℓ using playouts corresponds to estimating the expected output value of oracle O_ℓ. So, from now on, we consider the game tree $\mathcal{T}$ as oracle tree $\mathcal{T}$, in which each leaf ℓ is equipped with oracle O_ℓ. Figure 2 shows an illustration of the oracle tree $\mathcal{T}$.

We denote the set of leaves as leaves($\mathcal{T}$) and root as the root of tree $\mathcal{T}$. Let children(u) and parent(u) be the set of child node of u and parent node of u, respectively. Let $L = |\text{leaves}(\mathcal{T})|$ be the number of leaves of $\mathcal{T}$ and Anc(ℓ) be the set of ancestor nodes including ℓ itself. Finally, we define the score μ_u of node u in the following way.
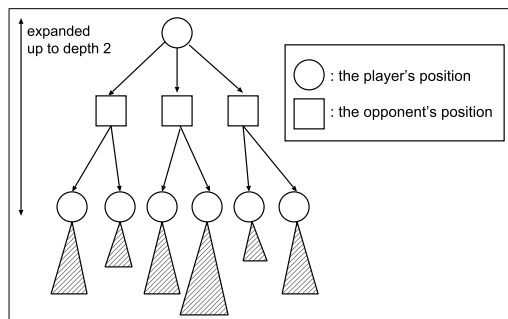

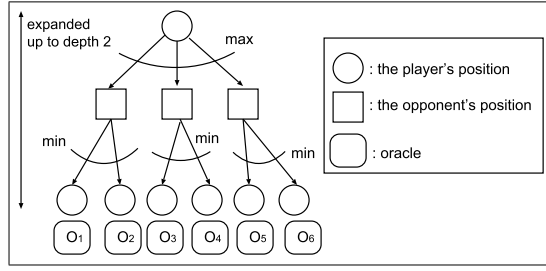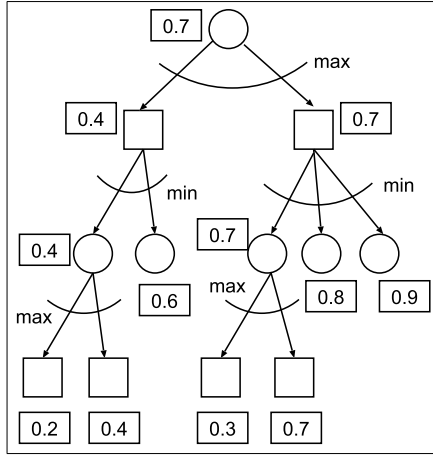**Fig. 1**　Illustration of a game tree with depth 2.

Fig. 2 Illustration of oracle tree $\mathcal{T}$.

Fig. 3 Illustration of a tree in which each node is attached with score.

$$\mu_u = \begin{cases} \mu_\ell, & \text{node } u \text{ is leaf } \ell, \\ \max_{v \in \text{children}(u)} \mu_v, & \text{depth of node } u \text{ is even}, \\ \min_{v \in \text{children}(u)} \mu_v, & \text{depth of node } u \text{ is odd}. \end{cases}$$

In particular, let $\mu^* = \mu_{\text{root}}$, which is the min-max score. Figure 3 illustrates a tree in which each node is attached with a score. We define Monte Carlo tree search problem as follows.

Definition 1 (Monte Carlo tree search problem). *Monte Carlo tree search problem is, given an oracle tree $\mathcal{T}$, the precision parameter $\varepsilon > 0$ and the confidence parameter δ $(0 < \delta < 1)$ as input, to output a child node $u \in$* children(root) *such that*

$$\mathbf{P}[\mu_u \geq \mu^* - \varepsilon] \geq 1 - \delta.$$

3. Our Algorithm

We introduce the notion of a winner in the following way.

Definition 2 (Winner). *A node u of the oracle tree $\mathcal{T}$ is winner if $\mu_u = \mu_{\text{parent}(u)}$.*

Suppose that a node u is a winner. If the depth of its parent node $p = \text{parent}(u)$ is even (p is the player's position), $\mu_u = \max_{v \in \text{children}(p)} \mu_v$. Otherwise (the depth is odd), we
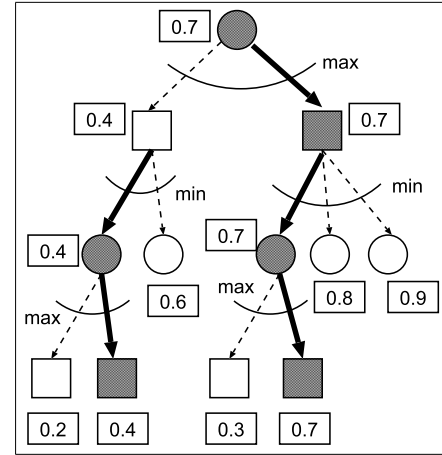

Fig. 4 An example of winners.

have $\mu_u = \min_{v \in \text{children}(p)} \mu_v$. Note that there might be several winners among child nodes of p. In particular, a winner among child nodes of the root attains the min-max score.

The basic idea of our method is to delete subtrees whose root nodes turn out not to be winners, so as to reduce the size of the tree and oracle calls. Observe that, even if we delete a subtree whose root node is not a winner, the min-max score of the tree does not change. In Fig. 4, we give an example of winners. Each dark colored node is a winner. Also, edges between a winner and its parent edge are represented with thick lines.

If we can find winners in an efficient way, we can reduce oracle calls. But, in order to check if each node is a winner, it is necessary to estimate the score of each node accurately enough, which requires sufficiently many oracle calls. To avoid this dilemma, we design a strategy which estimate the score of each node in several stages starting from rough precision to high precision. As a result, we can get rid of nodes which turn out not to be winners in early stages even with rough precision. We will show our algorithm in the next section.

3.1 Our Algorithm

Our algorithm FindTopWinner consists of two procedures EstimateValue and Prune, respectively and it works in rounds.

At each round $m = 1, \ldots, \lceil \log 2/\varepsilon \rceil$, for each node u, the procedure EstimateValue finds an estimate $\hat{\mu}_u$ of the score μ_u so that the error of $\hat{\mu}_u$ is at most ε_m with high probability. More precisely, given a node u, EstimateValues(u, ε_m, δ_m) estimates scores of nodes in the subtree rooted at u in a recursive way and returns an estimate $\hat{\mu}_u$ of the score of u.

Then, for each node u, the procedure Prune checks if u is a winner. If Prune judges that u is not a winner, Prune deletes the subtree rooted at u. More specifically, given a node u, Prune(u) judges that a child node v of u is not a winner if

Algorithm 1 FindTopWinner

1: **Input:** $\varepsilon, \delta, \mathcal{T}$
2: **Initialization:** $\varepsilon_0 = 1, \delta_0 = \delta/L$.
3: **for** $m = 1$ **to** $\lceil \log_2 2/\varepsilon \rceil$ **do**
4:      $\varepsilon_m = \varepsilon_{m-1}/2$ and $\delta_m = \delta_{m-1}/2$.
5:      **if** $|\text{children}(\text{root})| = 1$ **then**
6:          **break**
7:      **end if**
8:      $\hat{\mu}_{\text{root}} = \text{EstimateValues}(\text{root}, \varepsilon_m, \delta_m)$
9:      $\text{Prune}(\text{root}, \varepsilon_m)$
10: **end for**
11: **return** $I = \arg\max\limits_{v \in \text{children}(\text{root})} \hat{\mu}_v$

Algorithm 2 EstimateValues($u, \varepsilon_m, \delta_m$)

1: **if** $u \in \text{leaves}(\mathcal{T})$ **then**
2:      call oracle O_u until the total calls of O_u reaches $n_m = \lceil (1/(2\varepsilon_m^2)) \ln(2/\delta_m) \rceil$. Let N_u be the sum of outputs of O_u.
3:      **return** N_u/n_m
4: **end if**
5: **for all** $v \in \text{children}(u)$ **do**
6:      $\hat{\mu}_v = \text{EstimateValues}(v, \varepsilon_m, \delta_m)$
7: **end for**
8: **if** depth of u is even **then**
9:      **return** $\max\limits_{v \in \text{children}(u)} \hat{\mu}_v$
10: **else**
11:      **return** $\min\limits_{v \in \text{children}(u)} \hat{\mu}_v$
12: **end if**

Algorithm 3 Prune(u, ε_m)

1: **for all** $v \in \text{childlen}(u)$ **do**
2:      **if** $|\hat{\mu}_u - \hat{\mu}_v| > 2\varepsilon_m$ **then**
3:          $\text{childlen}(u) = \text{childlen}(u) \setminus \{v\}$
4:      **else if** $v \notin \text{leaves}(T)$ **then**
5:          $\text{Prune}(v, \varepsilon_m)$
6:      **end if**
7: **end for**

$$|\hat{\mu}_u - \hat{\mu}_v| > 2\varepsilon_m. \tag{1}$$

3.2 Analysis

Let $\text{nodes}_m(\mathcal{T})$ and $\text{leaves}_m(\mathcal{T})$ be the set of nodes and leaves in $\mathcal{T}$ which are not deleted yet at the beginning of round m, respectively. Similarly, let $\text{children}_m(u)$ be the set of child nodes of u which are not deleted at the beginning of round m. Then, we define the event such that all estimates of scores of nodes and leaves are obtained approximately correctly while FindTopWiiner is running.

Definition 3 (Event A). *The event* A *is such that, at each round m and for each leaf $\ell \in \text{leaves}_m(\mathcal{T})$, the estimate $\hat{\mu}_\ell$ obtained by* EstimateValues($\ell, \varepsilon_m, \delta_m$) *satisfies $|\mu_\ell - \hat{\mu}_\ell| \leq \varepsilon_m$.*

Lemma 1. *The event* A *occurs with probability at least $1 - \delta$.*

Proof. By Hoeffding's inequality (Proposition 1), at each round m and for each leaf $\ell \in \text{leaves}_m(\mathcal{T})$, the estimate $\hat{\mu}_{\ell,m}$ obtained by EstimateValues at round m satisfies

$$\mathbf{P}[|\mu_\ell - \hat{\mu}_{\ell,m}| > \varepsilon_m] \leq 2\exp\left(-2\varepsilon_m{}^2 n_m\right) \leq \frac{\delta}{2^m L},$$

where the second inequality holds since $n_m = \lceil (1/(2\varepsilon_m^2)) \ln(2/\delta_m) \rceil$. Therefore, event

$\mathbf{P}[\text{the event A does not occur}]$
$= \mathbf{P}[\exists m \geq 1, \exists l \in \text{leaves}_m(\mathcal{T}), |\mu_l - \hat{\mu}_{l,m}| > \varepsilon_m]$
$\leq L \sum\limits_{m=1}^{\infty} \frac{\delta}{2^m L} = \delta,$

where the second inequality holds by the union bound (Proposition 2).

$\square$

Then, for each round m, we define the following events B(m) and C(m).

Definition 4 (Event B(m)). *The event* B(m) *is the event in which for any node $u \in \text{nodes}_m(\mathcal{T})$, there exists a winner in* $\text{children}_m(u)$.

Definition 5 (Event C(m)). *The event* C(m) *is the event in which for any node $u \in \text{nodes}_m(\mathcal{T})$,* EstimateValues($u, \varepsilon_m, \delta_m$) *outputs the estimate $\hat{\mu}_u$ such that $|\mu_u - \hat{\mu}_u| \leq \varepsilon_m$.*

Intuitively, we think that if these events occur at each round of FindTopWinner, the algorithm is successful.

Lemma 2. *If event the* A *and the* B(m) *occurs, the event* C(m) *also occurs.*

Proof. Suppose that both the events A and B(m) happen. Then, we prove that the event C(m) also happens by induction on the height of nodes, where the height of a node u is the height of the subtree rooted at u. Note that height and depth are different notions.

Now, observe that for each node $u \in \text{nodes}_m(\mathcal{T})$, EstimateValues($u, \varepsilon_m, \delta_m$) at round m outputs the estimate $\hat{\mu}_u$ such that

$$\hat{\mu}_u = \begin{cases} N_u/n_m, & u \text{ is a leaf,} \\ \max\limits_{v \in \text{children}_m(u)} \hat{\mu}_v, & \text{depth of } u \text{ is even,} \\ \min\limits_{v \in \text{children}_m(u)} \hat{\mu}_v, & \text{depth of } u \text{ is odd.} \end{cases}$$

(i) Assume that the depth of u is 0. Then u is a leaf and $|\mu_u - \hat{\mu}_u| \leq \varepsilon_m$ by definition of the event A. (ii) Then, assume that for any node $v \in \text{nodes}_m(\mathcal{T})$ with its height less than $h - 1$, it holds that $|\mu_v - \hat{\mu}_v| \leq \varepsilon_m$. Let u be any node in $\text{nodes}_m(\mathcal{T})$ of height h. For simplicity, we assume that the depth of u is odd (the proof is similar for the case when the depth is even). Since the event B(m) occurs, there exists a winner $v_1 \in \text{children}_m(u)$ such that following property:

$$v_1 = \arg\min\limits_{v \in \text{children}_m(u)} \mu_v, \ \mu_u = \mu_{v_1}.$$

Then, let

$$v_2 = \arg\min\limits_{v \in \text{children}_m(u)} \hat{\mu}_v$$

That is, $\hat{\mu}_u = \hat{\mu}_{v_2}$. Since the height of v_1 or v_2 is at most $h-1$, by the inductive assumption, it holds that $\hat{\mu}_{v_1} \leq \mu_{v_1} + \varepsilon_m$, and $\mu_{v_2} \leq \hat{\mu}_{v_2} + \varepsilon_m$, respectively. So, we have

$$\hat{\mu}_u \leq \hat{\mu}_{v_1} \leq \mu_{v_1} + \varepsilon_m = \mu_u + \varepsilon_m,$$
$$\mu_u \leq \mu_{v_2} \leq \hat{\mu}_{v_2} + \varepsilon_m = \hat{\mu}_u + \varepsilon_m.$$

This implies that $|\mu_u - \hat{\mu}_u| \leq \varepsilon_m$, i.e., the event C(m) occurs. □

Lemma 3. *If the event* A *and* B(m) *occur, then the event* B(m + 1) *also occurs.*

Proof. Suppose that both the event A and B(m) occur. By Lemma 2, the event C(m) also holds. Let u be any node in $\text{nodes}_{m+1}(\mathcal{T})$. Then, clearly, we have $u \in \text{nodes}_m(\mathcal{T})$. Further, since B(m) occurs, there exists a winner $v \in \text{children}_m(u)$. Note that, since v is a winner, $\mu_u = \mu_v$. On the other hand, since C(m) occurs, it holds that, after EstimateValues is called,

$$|\mu_u - \hat{\mu}_u| \leq \varepsilon_m, \text{ and, } |\mu_v - \hat{\mu}_v| \leq \varepsilon_m.$$

Therefore,

$$\begin{aligned}|\hat{\mu}_u - \hat{\mu}_v| &= |(\hat{\mu}_u - \mu_u) + (\mu_v - \hat{\mu}_v)| \\ &\leq |\hat{\mu}_u - \mu_u| + |\mu_v - \hat{\mu}_v| \\ &\leq 2\varepsilon_m.\end{aligned}$$

This implies that v is not deleted by Prune(v) at round m and thus we have $v \in \text{children}_{m+1}(u)$. So, the event B(m + 1) occurs. □

Lemma 4. *Suppose that the event* A *occurs. Then, it follows that*

1. *for any* $m \geq 1$, *for any* $u \in \text{nodes}_m(\mathcal{T})$, EstimateValues($u, \varepsilon_m, \delta_m$) *at round m outputs $\hat{\mu}_u$ such that $|\mu_u - \hat{\mu}_u| \leq \varepsilon_m$, and*
2. *Any node $v^* = \underset{v \in \text{children(root)}}{\arg\max} \mu_v$ will not be deleted.*

Proof. By Lemma 2 and 3, for any round m, the events B(m) and C(m) occur. This implies the first statement of the lemma. Further, any node attaining the min-max score is a winner, the second statement holds. □

Definition 6. *For each leaf ℓ, we define the parameter Δ_ℓ as follows.*

$$\Delta_\ell = \max_{u \in \text{Anc}(\ell) \setminus \{\text{root}\}} |\mu_u - \mu_{\text{parent}(u)}|. \tag{2}$$

Definition 7. *Let m_ℓ be the shortest round m such that $\varepsilon_m < \Delta_\ell/4$ for leaf $\ell \in L(\mathcal{T})$.*

$$m_\ell = \left\lfloor \log_2 \frac{4}{\Delta_\ell} \right\rfloor + 1. \tag{3}$$

Lemma 5. *If the event* A *occurs, for each leaf ℓ, ℓ is deleted at round $m \leq m_\ell$.*

Proof. Suppose that leaf ℓ is not deleted by the end of round $m_\ell - 1$. Then by the definition of Δ_ℓ, there exits a node $u \in \text{Anc}(l)$ such that

$$|\mu_u - \mu_{\text{parent}(u)}| = \Delta_\ell.$$

If the event A occurs, by the first statement of Lemma 4, EstimateValues($u, \varepsilon_m, \delta_m$) outputs the estimate $\hat{\mu}_u$ such that $|\mu_u - \hat{\mu}_u| \leq \varepsilon_{m_\ell}$ and $|\mu_{\text{parent}(u)} - \hat{\mu}_{\text{parent}(u)}| \leq \varepsilon_{m_\ell}$. Therefore,

$$\begin{aligned}&|\hat{\mu}_u - \hat{\mu}_{\text{parent}(u)}| \\ &= |(\hat{\mu}_u - \mu_u) + (\mu_u - \mu_{\text{parent}(u)}) \\ &\quad + (\mu_{\text{parent}(u)} - \hat{\mu}_{\text{parent}(u)})| \\ &\geq -|\hat{\mu}_u - \mu_u| + |\mu_u - \mu_{\text{parent}(u)}| \\ &\quad - |\mu_{\text{parent}(u)} - \hat{\mu}_{\text{parent}(u)}| \\ &\geq -\varepsilon_{m_l} + \Delta_l - \varepsilon_{m_l} \\ &> 4\varepsilon_{m_l} - 2\varepsilon_{m_l} = 2\varepsilon_{m_l},\end{aligned}$$

where the last inequality holds since $\Delta_\ell > 4\varepsilon_{m_\ell}$. The above inequality ensures that node u is deleted by Prune(u) at round m_ℓ, which implies that the leaf ℓ, a descendant of u, is also deleted. □

Theorem 1. *The algorithm* FindTopWinner *satisfies the following two conditions with probability at least $1 - \delta$.*

1. *$\mu_I \geq \mu^* - \varepsilon$,*
2. *The number of oracle calls is at most*

$$\sum_{\ell:\Delta_\ell > 2\varepsilon} \left(\frac{32}{\Delta_\ell^2} \ln \frac{16L}{\Delta_\ell \delta} + 1 \right) + \sum_{\ell:\Delta_\ell \leq 2\varepsilon} \left(\frac{8}{\varepsilon^2} \ln \frac{8L}{\varepsilon \delta} + 1 \right).$$

Proof. Suppose that the event A occurs. First, we prove the first statement of the theorem. FindTopWinner terminates if and only if one of the following two cases hold.

1. (The condition$|\text{children(root)}| = 1$ is satisfied before the last round M starts.) In this case, by the second statement of Lemma 4, the remaining child node attains the min-max score. Therefore, we have $\mu_I = \mu^* \geq \mu^* - \varepsilon$.
2. (The above condition is not satisfied until the end of round M.) In this case, note that $\varepsilon_M \leq \varepsilon/2$. Then, by the first statement of Lemma 4,

$$\begin{aligned}\mu^* - \frac{\varepsilon}{2} &\leq \mu^* - \varepsilon_M \\ &= \mu_{\text{root}} - \varepsilon_M \\ &\leq \hat{\mu}_{\text{root}} \\ &= \hat{\mu}_I \\ &\leq \mu_I + \varepsilon_M \\ &\leq \mu_I + \frac{\varepsilon}{2}.\end{aligned}$$

Therefore, we have $\mu_I \geq \mu^* - \varepsilon$.

Then we prove the second statement of the theorem. By Lemma 5, for each leaf ℓ, the number of calls of O_ℓ is at most n_{m_ℓ}. Note that, for any leaf ℓ such that $m_\ell > M$, the number of oracle calls is upper bounded trivially by n_M. So,

total number of oracle calls is at most

$$\sum_{\ell:\Delta_\ell \geq 2\varepsilon} n_{m_\ell} + \sum_{\ell:\Delta_\ell < 2\varepsilon} n_M, \qquad (4)$$

where we use the fact that $m_\ell > M \iff \Delta_\ell < 2\varepsilon$. Further, the following inequalities hold.

$$n_m < 2^{2m-1} \ln \frac{2^{m+1}L}{\delta} + 1,$$

$$m_l \leq \log_2 \frac{4}{\Delta_\ell} + 1,$$

$$M < \log_2 \frac{2}{\varepsilon} + 1.$$

By using these inequalities, the term (4) is upper bounded as follows.

$$\sum_{\ell:\Delta_\ell>2\varepsilon} n_{m_l} + \sum_{\ell:\Delta_\ell\leq 2\varepsilon} n_M$$

$$< \sum_{\ell:\Delta_\ell>2\varepsilon} \left(2^{2m_\ell-1}\ln\frac{2^{m_\ell+1}L}{\delta}+1\right)$$

$$+ \sum_{\ell:\Delta_\ell\leq 2\varepsilon} \left(2^{2M-1}\ln\frac{2^{M+1}L}{\delta}+1\right)$$

$$< \sum_{\ell:\Delta_\ell>2\varepsilon} \left(\frac{32}{\Delta_\ell^2}\ln\frac{16L}{\Delta_\ell\delta}+1\right)$$

$$+ \sum_{\ell:\Delta_\ell\leq 2\varepsilon} \left(\frac{8}{\varepsilon^2}\ln\frac{8L}{\varepsilon\delta}+1\right).$$

$\square$

Let

$$\Delta = \min\{\mu^* - \mu_u \mid u \in \text{children(root)}, \mu_u < \mu^*\}.$$

Proposition 3. $\Delta = \min\limits_{\ell\in\text{leaves}(\mathcal{T})} \Delta_\ell.$

Proof. For any leaf ℓ, there exists the child node of the root and an ancestor of ℓ. That is,

$$\text{children(root)} \cap \text{Anc}(\ell) = \{u\}.$$

By the definition of Δ_ℓ and Δ, we have $\Delta_\ell \geq \mu^* - \mu_u$ and $\mu^* - \mu_u \geq \Delta$. So, $\Delta_l \geq \Delta$. Since ℓ is any leaf, we have $\min\limits_{\ell\in\text{leaves}(\mathcal{T})} \Delta_\ell \geq \Delta$.

On the other hand, there exists a node $u \in$ children(root) such that $\mu^* - \mu_u = \Delta$. Then, there also exists a leaf ℓ' such that there is a path from u to ℓ' so that each node in the path is a winner. Since the score of each node in the path is μ_u, we have $\mu_u = \mu_{\ell'}$. By the definition of Δ_ℓ, we have $\Delta_{\ell'} = \mu^* - \mu_u = \Delta$. This implies $\Delta \geq \min\limits_{\ell\in\text{leaves}(\mathcal{T})} \Delta_\ell.$ $\square$

In particular, if the event A occurs and there is only one winner among child node of the root of $\mathcal{T}$ (i.e., $|\{u \in$ children(root) $\mid \mu_u = \mu^*\}| = 1$), then the number of oracle calls can be much smaller, which we prove in the next corollary.

Corollary 1. *If there is only one winner among child node of the root of $\mathcal{T}$, the number of oracle calls of FindTopWinner is, with probability at least $1 - \delta$,*

$$\tilde{O}\left(L\min\left(\frac{1}{\varepsilon^2}, \frac{1}{\Delta^2}\right)\right).$$

Proof. We consider the following two cases.

1. ($\Delta > 8\varepsilon$) In this case, FindTopWinner terminate at round M' such that

$$\varepsilon_{M'-1} < \frac{\Delta}{4}.$$

Then calls of the oracle is done at round at most $M' - 1$. By rearranging the inequality above, we get

$$M' = \left\lfloor \log_2 \frac{8}{\Delta} \right\rfloor + 1.$$

The total number of oracle calls is at most

$$\frac{128}{\Delta^2}\ln\frac{16L}{\Delta\delta} + 1 \leq \frac{2}{\varepsilon^2}\ln\frac{2L}{\varepsilon\delta} + 1.$$

2. ($\Delta \leq 8\varepsilon$) Otherwise, FindTopWinner terminates at round at most M. The total number of oracle calls is at most

$$\frac{8}{\varepsilon^2}\ln\frac{8L}{\varepsilon\delta} + 1 \leq \frac{512}{\Delta^2}\ln\frac{64L}{\Delta\delta} + 1.$$

So, in both cases, FindTopWinner call oracles at most

$$\min\left(\frac{8}{\varepsilon^2}\ln\frac{8L}{\varepsilon\delta} + 1, \frac{512}{\Delta^2}\ln\frac{64L}{\Delta\delta} + 1\right)$$

times.

$\square$

4. Experiments

In this section, we show some experimental result using artificial data. We compare FindTopWinner with UCT [6] and the naive sampling method based on Hoeffding's inequality.

In our artificial data, we construct an oracle tree with depth 3 in which each node has 10 outgoing edges. So, the number of leaves of the tree is 1000. The score of each leaf is determined randomly from $[0, 1]$. We set the parameters as $\varepsilon = 0.01$ and $\delta = 0.1$.

For the oracle tree above, we run FindTopWinner and UCT, respectively. We plot the error w.r.t. the min-max score $\mu^* - \mu_I$ of FindTopWinner at the end of each round. Note that FindTopWinner, given ε and δ and the oracle tree, automatically determines the number of oracle calls during its run. We also plot that of UCT. UCT is an online algorithm which calls the oracle once at each round. We run UCT for sufficiently many rounds so that its total number of oracle calls is the same as that of FindTopWinner. We repeat this experiment for 10 times. All the results are shown
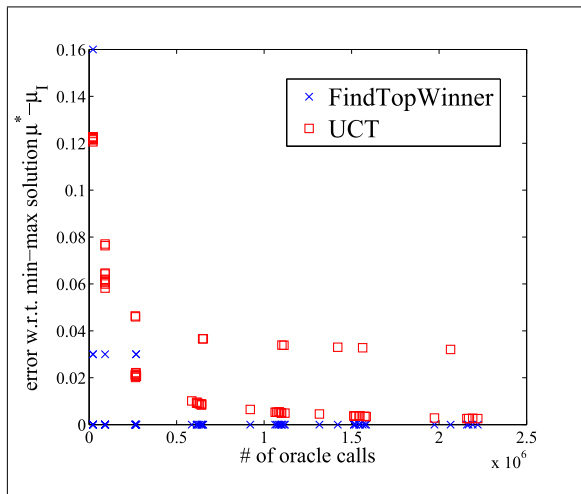
Fig. 5　Error w.r.t. the min-max score for the artificial oracle tree.

Table 1　Total oracle calls to obtain ε-approximate solution for the min-max score of the artificial oracle tree.

Naive method	FindTopWinner
198,070,000	1,970,085

in Fig. 5. As can be seen in Fig. 5, as the number of total oracle calls increases, FindTopWinner performs better than UCT.

　　Then, we compare FindTopWinner with the naive method for the same artificial oracle tree with the same parameters, i.e., $\varepsilon = 0.01$ and $\delta = 0.1$. Here, we refer the naive method to as the method that calls the oracle of each leaf sufficiently many times as suggested by Hoeffding's inequality. More precisely, for each leaf ℓ in the tree, the naive method calls the oracle O_ℓ for $2\ln(2L/\delta)/\varepsilon^2$ times. Then, the naive method performs the min-max search on the tree with the estimated scores. We run both algorithms for 10 times. The average number of total oracle calls of each algorithm to obtain ε-approximate solutions of the min-max score is shown in Table 1, respectively. Roughly speaking, total oracle calls of FindTopWinner is about 100 times smaller than those of the naive method. This results shows the effectiveness of deleting leaves which are judged as non-winners.

5.　Conclusion

In this paper, we consider the Monte Carlo tree search problem and propose a new algorithm.

　　One of our future work is to apply α-β pruning [9] to our algorithm. α-β pruning is a search method for full information games, which performs better than Min-Max search method by removing redundant subtrees in a game tree. α-β pruning, however, needs exact scores of each node, which can be only approximately estimated in Monte Carlo tree search problem. It might be possible to incorporate α-β pruning with our algorithm using approximated scores, which could reduce number of oracle calls further.

References

[1] C. Browne, E.J. Powley, D. Whitehouse, S.M. Lucas, P.I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A survey of Monte Carlo tree search methods," IEEE Trans. Computational Intelligence and AI in Games, vol.4, no.1, pp.1–43, 2012.

[2] R. Coulom, "Efficient selectivity and backup operators in Monte-Carlo tree search," Proc. 5th international conference on Computers and games, CG'06, pp.72–83, 2007.

[3] H. Yoshimoto, K. Yoshizoe, T. Kaneko, A. Kishimoto, and K. Taura, "Monte Carlo go has a way to go," Proc. National Conference on Artificial Intelligence, pp.1070–1075, 2006.

[4] S. Gelly, Y. Wang, R. Munos, and O. Teytaud, "Modification of UCT with patterns in Monte-Carlo Go," Research Report RR-6062, INRIA, 2006.

[5] S. Gelly and D. Silver, "Combining online and offline knowledge in UCT," Proc. International Conference on Machine Learning 2007, pp.273–280, 2007.

[6] L. Kocsis and C. Szepesvári, "Bandit based Monte-Carlo planning," Proc. European Conference on Machine Learning 2006, pp.282–293, 2006.

[7] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," Machine Learning, vol.47, pp.235–256, 2002.

[8] W. Hoeffding, "Probability inequalities for sums of bounded random variables," J. American Statistical Association, pp.13–30, 1963.

[9] D.E. Knuth and R.W. Moore, "An analysis of alpha-beta pruning," Artificial Intelligence, vol.6, no.4, pp.293–326, 1975.

Kazuki Teraoka　　received B.E and M.E. degrees from Kyushu University in 2010 and 2012, respectively. He now works for Fujitsu Limited.

Kohei Hatano　　received Ph.D. from Tokyo Institute of Technology in 2005. Currently, he is an assistant professor at Department of Informatics in Kyushu University. His research interests include boosting, online learning and their applications.

Eiji Takimoto　　received Dr. Eng. degree from Tohoku University in 1991. Currently, he is a professor at Department of Informatics in Kyushu University. His research interests include computational complexity, computational learning theory, and online learning.