LinUCB Applied to Monte Carlo Tree Search

Yusaku Mandai and Tomoyuki Kaneko

The Graduate School of Arts and Sciences, the University of Tokyo, Tokyo, Japan
`mandai@graco.c.u-tokyo.ac.jp`

Abstract. UCT is a standard method of Monte Carlo tree search (MCTS) algorithms, which have been applied to various domains and have achieved remarkable success. This study proposes a family of LinUCT algorithms that incorporate LinUCB into MCTS algorithms. LinUCB is a recently developed method that generalizes past episodes by ridge regression with feature vectors and rewards. LinUCB outperforms UCB1 in contextual multi-armed bandit problems. We introduce a straightforward application of LinUCB, LinUCT$_{\text{PLAIN}}$ by substituting UCB1 with LinUCB in UCT. We show that it does not work well owing to the minimax structure of game trees. To better handle such tree structures, we present LinUCT$_{\text{RAVE}}$ and LinUCT$_{\text{FP}}$ by further incorporating two existing techniques, rapid action value estimation (RAVE) and feature propagation, which recursively propagates the feature vector of a node to that of its parent. Experiments were conducted with a synthetic model, which is an extension of the standard incremental random tree model in which each node has a feature vector that represents the characteristics of the corresponding position. The experiments results indicate that LinUCT$_{\text{RAVE}}$, LinUCT$_{\text{FP}}$, and their combination LinUCT$_{\text{RAVE-FP}}$ outperform UCT, especially when the branching factor is relatively large.

Keywords: MCTS, Multi-armed Bandit Problem, Contextual Bandit, LinUCB

1 Introduction

UCT [13] is a de facto standard algorithm of Monte Carlo tree search (MCTS) [3]. UCT has achieved remarkable successes in various domains including the game of Go [8].

For game tree search, UCT constructs and evaluates a game tree through a random sampling sequence. At each time step, a *playout* involving Monte Carlo simulation is performed to improve the empirical estimation of the winning ratio at a leaf node and that of the ancestors of the leaf. For each playout, a leaf in the game tree is selected in a best-first manner by descending the most promising move with respect to the upper confidence bound of the winning ratio, UCB1 [1]. After it reaches the leaf, the score of the playout is determined by the terminal position, which is reached by alternatively playing random moves. Therefore, UCT works effectively without heuristic functions or domain knowledge. The fact is a remarkable advantage over traditional game tree search methods based on alpha beta search [12], because such methods require adequate evaluation functions to estimate a winning probability of a position. Generally, the construction of such evaluation functions requires tremendous effort [11].

Although UCT does not explicitly require heuristics, many studies have incorporated domain-dependent knowledge into UCT to improve the convergence speed or playing strength, such as progressive widening [6], prior knowledge [10], and PUCB [16]. Such approaches utilize a set of features, i.e., a *feature vector*, which is observable in each state or in each move.

This study proposes a family of LinUCT as new MCTS algorithms. LinUCT is based on LinUCB [15, 5], which has been studied in contextual multi-armed bandit problem [4]. While UCB1 only considers past rewards for each arm, LinUCB generalizes past episodes by ridge regression with feature vectors and rewards to predict the rewards of a future state. Thus, LinUCB is an alternative means to incorporate domain knowledge in an online manner. We first introduce LinUCT$_{\text{PLAIN}}$ by substituting UCB1 with LinUCB in UCT. However, this straightforward application of LinUCB is not promising, because it does not consider information in the subtree expanded under a node. To overcome this problem, we present LinUCT$_{\text{RAVE}}$ and LinUCT$_{\text{FP}}$, by incorporating two existing techniques, rapid action value estimation (RAVE) [9] and feature propagation that propagates feature vectors in a subtree to its root. We conducted experiments with a synthetic model that is a variant of incremental random trees that have served as good test sets for search algorithms [18, 14, 13, 7, 20]. We extend the trees such that each node has a feature vector while preserving the main property of the incremental random trees. The experiments demonstrate that LinUCT$_{\text{RAVE}}$, LinUCT$_{\text{FP}}$, and their combination LinUCT$_{\text{RAVE-FP}}$ outperform UCT, especially when the branching factor is relatively large.

2 MCTS and Algorithms in Multi-Armed Bandit Problems

In the game of Go [2], minimax tree search does not work effectively because of the difficulties in constructing heuristic evaluation functions. After the emergence of a new paradigm, Monte Carlo tree search (MCTS), the playing strength of computer players has been improved significantly in Go [8]. MCTS relies on a number of random simulations according to the following steps [3].

1. Selection: Starting at the root node, the algorithm descends the tree to a leaf. At each internal node, an algorithm for multi-armed bandit problems is employed to determine and select the move with the highest value with respect to a given criterion. Here, we consider UCB1, UCB1$_{\text{RAVE}}$, and LinUCB, as the selection criteria.
2. Expansion: The children of the leaf are expanded if appropriate, and one child is selected randomly.
3. Simulation: The rest of the game is played randomly. Typically, the game consists of completely random moves; however, some studies have suggested

that a well-designed probability of moves yields improved performance [6, 17].
4. Back propagation: The result (i.e., win/loss) of the simulation is back-propagated to the nodes on the path the algorithm descended in Step 1.

These steps are repeated until the computational budget (e.g. time) is exhausted. Then, the best move, typically the most visited child of the root, is identified.

2.1 UCT

UCT (UCB applied to trees) is one of the most successful MCTS variants. In the selection step at time t, UCT computes the UCB1 value [1] for move a of node s as follows:

$$\mathrm{UCB1}(s,a) = \overline{X}_a + \sqrt{\frac{2\ln N_s}{N_a}} \tag{1}$$

where $\overline{X}_a$ is the empirical mean of the rewards of move a, and N_i is the number of visits to the node or move i. Note that a position after a move is defined without ambiguity in deterministic games. Thus, we use move a or the position after move a interchangeably for simplicity. The best move converges to the same move identified by a minimax algorithm under a certain assumption [13].

2.2 RAVE

RAVE is a remarkable enhancement to MCTS, particularly effective in Go [10]. It is a generalization of the All Moves As First (AMAF) heuristic, where AMAF treats all moves in a simulation as if they were selected as the first move. When RAVE is incorporated, the following interpolation $\mathrm{UCB1}_{\mathrm{RAVE}}$ is used as the selection criterion:

$$\mathrm{UCB1}_{\mathrm{RAVE}}(s,a) = \beta(s)\ \mathrm{RAVE}(s,a) + (1-\beta(s))\ \mathrm{UCB1}(s,a) \tag{2}$$

where $\mathrm{RAVE}(s,a)$ has a similar form as Eq. (1). Note that $\overline{X}_a$ and N_a are counted according to AMAF in Go. Thus, the RAVE value may become rapidly a good estimate of the reward by incorporating various simulation results that are not contained in UCB1 for $\overline{X}_a$. The interpolation weight $\beta(s)$ is $\sqrt{\frac{k}{3N_s+k}}$. Therefore, the value of $\mathrm{UCB1}_{\mathrm{RAVE}}$ converges to that of UCB1 for large N_s, while RAVE covers the unreliable prediction of UCB1 when N_s is small. Parameter k controls the number of episodes when both terms are equal [9].

2.3 LinUCB

LinUCB is an algorithm for the contextual bandit problems [15, 5]. Here, a feature vector is observable for each arm, and it is assumed that the expected reward of arm a is defined by the inner product between the feature vector $\mathbf{x}_a$ and an unknown coefficient vector $\boldsymbol{\theta}^*_a$; $\mathbb{E}[r_a|\mathbf{x}_a] = \mathbf{x}_a^\top \boldsymbol{\theta}^*_a$, where r_a is the reward

Algorithm 1 LinUCB

```
Inputs: $\alpha \in \mathbb{R}_+$
for $t = 1, 2, 3, ...$ do
    for all $a \in \mathcal{A}_t$ do                    ▷ $\mathcal{A}_t$ is a set of available arms at $t$
        if $a$ is new then
            $\mathbf{A}_a \leftarrow \mathbf{I}_{d\times d}$                    ▷ $d$ dimensional identity matrix
            $\mathbf{b}_a \leftarrow \mathbf{0}_{d\times 1}$                    ▷ $d$ dimensional zero vector
        $\hat{\boldsymbol{\theta}}_a \leftarrow \mathbf{A}_a^{-1}\mathbf{b}_a$
        $p_a \leftarrow \mathbf{x}_a^\top \hat{\boldsymbol{\theta}}_a + \alpha\sqrt{\mathbf{x}_a^\top \mathbf{A}_a^{-1}\mathbf{x}_a}$
    $a_t \leftarrow \arg\max_{a\in\mathcal{A}_t} p_a$ with ties broken arbitrarily
    Observe a real-valued payoff $r_t$
    $\mathbf{A}_{a_t} \leftarrow \mathbf{A}_{a_t} + \mathbf{x}_{a_t}\mathbf{x}_{a_t}^\top$
    $\mathbf{b}_{a_t} \leftarrow \mathbf{b}_{a_t} + r_t\mathbf{x}_{a_t}$
```

of arm a. The LinUCB algorithm employs ridge regression to estimate $\boldsymbol{\theta}_a^*$ using the trials performed so far. The criterion in arm selection in LinUCB is expressed as follows:

$$\text{LinUCB}(a) = \mathbf{x}_a^\top \hat{\boldsymbol{\theta}}_a + \alpha\sqrt{\mathbf{x}_a^\top \mathbf{A}_a^{-1}\mathbf{x}_a}, \tag{3}$$

where $\hat{\boldsymbol{\theta}}_a$ is the current estimate of $\boldsymbol{\theta}_a^*$, and $\mathbf{A}_a^{-1}$ is the inverse of the variance-covariance matrix for the regression on arm a. Constant $\alpha > 0$ controls the exploration-exploitation balance. With a probability of at least $1 - \delta$, the difference between the current estimate $\mathbf{x}_a^\top \hat{\boldsymbol{\theta}}_a$ and the expected reward $\mathbb{E}[r_a|\mathbf{x}_a]$ is bounded [15, 19] as follows:

$$|\mathbf{x}_a^\top \hat{\boldsymbol{\theta}}_a - \mathbb{E}[r_a|\mathbf{x}_a]| \leq \alpha\sqrt{\mathbf{x}_a^\top \mathbf{A}_a^{-1}\mathbf{x}_a}, \text{ where } \alpha = 1 + \sqrt{\ln(2/\delta)/2}. \tag{4}$$

Thus, the first term of the right side of Eq. (3) estimates the reward of arm a, and the second term works as the confidence interval of the average reward. Therefore, LinUCB calculates the upper confidence bound of the reward of each arm, similar to UCB algorithms. Algorithm 1 describes the LinUCB algorithm, which updates $\hat{\boldsymbol{\theta}}_a$ via supplementary matrix $\mathbf{A}$ and vector $\mathbf{b}$, at each time step.

Note that we introduce only the basic LinUCB framework for simplicity of the paper. The authors of LinUCB also presented an extended framework in which a feature vector models both a visiting user and an article available at time t [15, 5].

3 LinUCT and Variants

In the original LinUCB, it is assumed that each arm a has its own coefficient vector $\boldsymbol{\theta}_a^*$; therefore matrix $\mathbf{A}_a$ and vector $\mathbf{b}_a$ are maintained individually. However, this configuration prevents LinUCB from generalizing information among positions when we model a move as an arm in deterministic games. On the other

Algorithm 2 Supplementary Procedures in LinUCT

```
1: procedure LinUct-Initialize
2:     $\mathbf{A} \leftarrow \mathbf{I}_{d\times d}$
3:     $\mathbf{b} \leftarrow \mathbf{0}_{1\times d}$
4: procedure Back-Propagation-Added(path, $\Delta$)
5:     for $s \in$ path do
6:         $\mathbf{A} \leftarrow \mathbf{A} + \mathbf{x}_s\mathbf{x}_s^\top$
7:         $\mathbf{b} \leftarrow \mathbf{b} + \Delta_s\mathbf{x}_s$
```

hand, it is reasonable to assume that the expected rewards are under the control of a common $\boldsymbol{\theta}^*$, for all nodes in a searched game tree. Hereafter, we use a common $\boldsymbol{\theta}^*$, matrix $\mathbf{A}$, and vector $\mathbf{b}$ (without subscripts). This decision follows the observation that a common evaluation function is used throughout search by minimax-based methods.

3.1 LinUCT$_{\text{PLAIN}}$: Basic LinUCT

Here we introduce LinUCT$_{\text{PLAIN}}$, which is a straightforward application of LinUCB to MCTS. In LinUCT$_{\text{PLAIN}}$, the selection step described in Section 2 employs LinUCB with the following adjustment in counting the number of simulations:

$$\text{LinUCB'}(s,a) = \mathbf{x}_a^\top\hat{\boldsymbol{\theta}} + \alpha\sqrt{\mathbf{x}_a^\top \cdot \frac{N_{s_0}}{N_a}\mathbf{A}^{-1} \cdot \mathbf{x}_a}, \tag{5}$$

where s_0 is the root node of a given search tree.

Algorithm 2 shows the supplementary procedures used in LinUCT$_{\text{PLAIN}}$. Procedure LinUct-Initialize initializes the global variables $\mathbf{A}$ and $\mathbf{b}$. After each simulation, in addition to the standard back-propagation process of N_s and $\overline{X}_s$ in MCTS, procedure Back-Propagation-Added updates variables $\mathbf{A}$ and $\mathbf{b}$ for each node s in the path from the root using the result of a playout Δ. Variable Δ_s is the relative reward of Δ with respect to the player of s. Consequently, matrix $\mathbf{A}$ is updated multiple times for each playout, while it is updated exactly once in the original LinUCB. This makes the second term for exploration in Eq. (3) too small too rapidly. Therefore, as in Eq. (5), we scale the elements in matrix $\mathbf{A}$ by the total number of playouts divided by the number of visits to the node.

3.2 LinUCT$_{\text{RAVE}}$: LinUCB Combined with UCB1 in RAVE Form

A concern with LinUCT$_{\text{PLAIN}}$ is that it evaluates a node only by its static feature vector. Consequently, the descendant node information is completely ignored, which is apparently problematic, because in traditional minimax search methods, the minimax value of as deep search as possible is preferable to a mere evaluation function's value for the root node.

Algorithm 3 Back-propagation process of LinUCT$_{\mathrm{FP}}$

```
1: procedure BACK-PROPAGATION-ADDED(path, $\varDelta$)
2:     for $s \in$ path do
3:         $\mathbf{A} \leftarrow \mathbf{A} + \mathbf{x}_s \mathbf{x}_s^\top$
4:         $\mathbf{b} \leftarrow \mathbf{b} + \varDelta_s \mathbf{x}_s$
5:         $p \leftarrow$ parent of $s$
6:         if $p$ is not null then
7:             $\mathbf{x}_p \leftarrow (1 - \gamma)\, \mathbf{x}_p + \gamma\, \mathbf{x}_s$
```

LinUCT$_{\mathrm{RAVE}}$, a combination of LinUCB and UCB1, resolves this problem by simply utilizing LinUCB as the RAVE heuristic function in Eq. (2).

$$\mathrm{LinUCB}_{\mathrm{RAVE}}(s,a) = \beta(s)\ \mathrm{LinUCB'}(s,a) + (1 - \beta(s))\ \mathrm{UCB1}(s,a) \tag{6}$$

The value converges to UCB1 value as the original RAVE presented in Eq. (2) does. In addition, LinUCT$_{\mathrm{RAVE}}$ makes the idea of RAVE more domain-independent. While the original RAVE assumes that the value of a move is independent of move order in most cases in a target game. This assumption holds in Go; however, apparently it does not hold in chess. On the other hand, LinUCT$_{\mathrm{RAVE}}$ can be applied to any game where a positional feature vector is available.

3.3 LinUCT$_{\mathrm{FP}}$: LinUCB with Propagation of Features

LinUCT$_{\mathrm{FP}}$ (feature propagation) is a completely different solution that considers subtrees. In LinUCT$_{\mathrm{FP}}$, by recursively propagating the feature vector of a node to that of its parent, the LinUCB value calculated using Eq. (3) reflects the expected rewards of playouts through the node. Algorithm 3 describes the modified back-propagation process used in LinUCT$_{\mathrm{FP}}$, where $\gamma \in (0, 1)$ controls the learning rate of a feature vector. We also present LinUCT$_{\text{RAVE-FP}}$, which incorporates this propagation scheme into LinUCT$_{\mathrm{RAVE}}$.

4 Incremental Random Game Tree with Feature Vectors

Here we introduce an extension to existing random game tree models. Incremental random trees (or P-game) have served as domain-independent test sets for evaluation of various search algorithms [18, 14, 13, 7, 20]. In this context, a random value is assigned to each edge, and the game theoretical value of a leaf is defined as the summation of the edge values in the path from the root. Moreover, for an internal node, the same summation can serve as a heuristic score returned by an evaluation function for that node. The advantages of this model are that (1) the search space can be controlled easily via the width and height, and (2) a correlation between the heuristic score of a node and that of its descendants is produced, which is expected in real games. Here, we extend the trees such that each node has a feature vector while preserving the main property of incremental random trees.

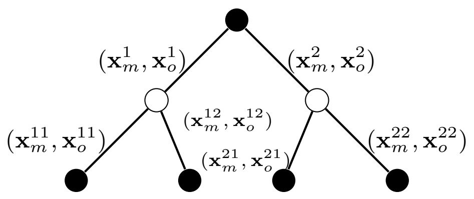

$(\mathbf{x}_m^1, \mathbf{x}_o^1)$
$(\mathbf{x}_m^2, \mathbf{x}_o^2)$
$(\mathbf{x}_m^{11}, \mathbf{x}_o^{11})$
$(\mathbf{x}_m^{12}, \mathbf{x}_o^{12})$
$(\mathbf{x}_m^{21}, \mathbf{x}_o^{21})$
$(\mathbf{x}_m^{22}, \mathbf{x}_o^{22})$

Fig. 1: Example of an incremental random tree with feature vectors.

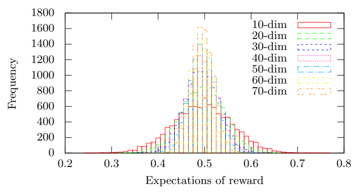


Fig. 2: Reward distributions

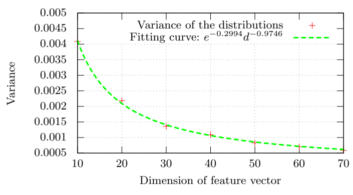


Fig. 3: Variance v.s. dimension

In our new trees, each tree has its own hidden d-dimensional vector $\boldsymbol{\theta}^* \in \mathbb{R}^d$, which cannot be observed by search algorithms. In addition, each node in a tree has two d-dimensional binary feature vectors (the one is for a player to move, and the other for the opponent): $\mathbf{x}_m, \mathbf{x}_o \in \{0,1\}^d$. In MCTS, a leaf returns binary reward $r = \{0,1\}$ for each playout, and the expected reward $\mathbb{E}(r)$ is defined by these vectors as follows:

$$\mathbb{E}(r) = (\mathbf{x}_m - \mathbf{x}_o)^\top \boldsymbol{\theta}^* + 0.5. \tag{7}$$

Similarly, the inner product $(\mathbf{x}_m - \mathbf{x}_o)^\top \hat{\boldsymbol{\theta}}$ gives the estimation of reward in LinUCB algorithms. This setting is equivalent to a Bernoulli multi-armed bandit problem when a tree depth is one.

To model a move in a two-player game, the feature vectors of each child $(\mathbf{x}'_m, \mathbf{x}'_o)$ are computed using those of the parent $(\mathbf{x}_m, \mathbf{x}_o)$. The feature vectors of a child inherit those of the parent with changing their owners, i.e., $\mathbf{x}'_m = \mathbf{x}_o$. This swap of vectors models the zero-sum property. Then, with a fixed probability p, each element (bit) is flipped. These flips correspond to changes in a board made by a move. Figure 1 shows an example of our tree. The left child of the root has feature vectors $(\mathbf{x}_m^1, \mathbf{x}_o^1)$, and its left child has feature vectors $(\mathbf{x}_m^{11}, \mathbf{x}_o^{11})$. These vectors are similar to $(\mathbf{x}_o^1,\, \mathbf{x}_m^1)$. For integer $i \in [0, d]$, the i-th element of $\mathbf{x}_m^{11}$ has the same or flipped value as that of the corresponding element of $\mathbf{x}_o^1$ with the probability p or $1 - p$, respectively.

Here we discuss some of the properties of our trees. We generated 100 trees by varying the dimension d of features. Note that the width and depth of a tree are fixed to 100 and 1, respectively. Figure 2 shows the histogram of the expected rewards, and Figure 3 gives their variances with a fitted curve. As can be seen, the distributions with higher dimension have a smaller variance. The reason for this is that each element of $\boldsymbol{\theta}^*$ is randomly set in $[-0.5/d, 0.5/d]$, to ensure that the probability given in Eq. (7) is within $[0, 1]$. However, the difference in variances may cause a problem because it is difficult for UCB algorithms to select the best arm when the reward of the second best arm is close to that of the best one. Therefore, we adjust the rewards according to the observed variances, and use the following formula rather than Eq.(7):

$$\mathbb{E}(r) = \min(1.0, \max(0.0, \frac{(\mathbf{x}_m - \mathbf{x}_o)^\top \boldsymbol{\theta}}{\sqrt{e^{-3.24994} d^{-0.9746} / \sigma'^2}} + 0.5)), \tag{8}$$

where e is the base of the natural logarithm, and σ'^2 is the desired variance. The constants -3.24994 and -0.9746 come from the curve fitted in Figure 3. We confirmed that this modification yields nearly the same distributions of rewards for various dimensions d and that approximately 96% were in $[0, 1]$ for $\sigma' = 0.5/3$.

5 Experiments

We performed experiments with various configurations of artificial trees to evaluate the performance of the proposed algorithms (LinUCT$_{\text{PLAIN}}$, LinUCT$_{\text{RAVE}}$, LinUCT$_{\text{FP}}$, and LinUCT$_{\text{RAVE-FP}}$) and UCT. These algorithms were investigated in terms of regrets and failure rates following the experiments in the literature [13]. *Regret* R, which is the cumulative loss of the player's selection, is a standard criterion in multi-armed bandit problems. To fit in the range $[0, 1]$, we divide R by the number of playouts n and use the average regret per playout $R/n = \mu^* - 1/n \sum_{t=1}^{n} \mu_{i_t}$. Here, μ^* is the expectation of reward of the optimal (maximum) move, and μ_{i_t} is the expected reward of the pulled arm i_t at time t. For game trees, we defined μ_i for each child i of the root as the theoretical minimax value of node i (i.e., the expected reward of the leaf of the principal variation). The *failure rate* is the rate by which the algorithm fails to choose the optimal move at the root. For each tree instance, each algorithm and each time t, whether the algorithm fails is determined by whether the most visited move in the root so far is the optimal move. By averaging them over tree instances, we obtain the failure rate of each algorithm at time t.

Instances of trees were generated randomly as described in Section 4. The parameter p for controlling the similarity between a parent and a child was fixed to 0.1, while the dimension d of feature vectors was selected according the number of leaves. In addition, we removed trees in which the optimal move is not unique. Each MCTS algorithm grows its search tree iteratively until it covers all nodes of the generated tree. All MCTS algorithms expand the children of a node at the second visit to the node unless the node is a leaf of the generated tree. For each playout, a move is selected randomly until it reaches the leaf. Then, the reward is set randomly by the probability associated with the node given in Eq. (8).

5.1 Robustness with Respect to Parameters

LinUCT algorithms depend on the parameters, i.e., exploration parameter α in the LinUCB value in Eq. (3), k in LinUCB$_{\mathrm{RAVE}}$ and its variants, and propagating rate γ for LinUCT$_{\mathrm{FP}}$. To observe the dependency of performance of LinUCT algorithms on these parameters, we evaluated the combinations of various configurations: for α, the values $1.0, 1.59, 1.83$, and 2.22 were tested, where 1.0 is the minimum value and the rest correspond to $\delta = 1.0, 0.5$, and 0.1 in Eq. (4), respectively. For k and γ, the values $100, 1000$, and 10000 and $0.1, 0.01$, and 0.001 were tested, respectively. To observe the dependence on the tree size, various pairs of (depth, branching factor) were tested: $(1, 256), (2, 16), (4, 4)$ and $(8, 2)$. Note that the dimension of feature vectors d was set to 8. Therefore each tree has exactly 256 leaves; thus, LinUCB can distinguish leaves in ideal cases.

The top, middle, and bottom two panels in Figure 4 show the failure rates and average regrets for each algorithm with varying α, k, and γ, respectively. Each point represents the average over 100 trees. As can be seen, the constants $\alpha = 1.0$, $k = 100$, and $\gamma = 0.01$ are slightly better than others, although the differences among the algorithms are more crucial than those between the parameters for the same algorithm. Therefore, we used these parameters for the rest of our experiments. Note that we only show the results for trees $(4, 4)$, because, the results obtained with different tree configurations were similar.

5.2 Comparison with UCT

We compared LinUCT algorithms to UCT, which is the standard algorithm in MCTS. For this experiment, the depth of the tree was fixed to four, while various branching factors (4-16) were used. Figure 5 shows the failure rates and regrets for each algorithm, where each point (x, y) is the branching factor of trees for x, and the average failure rate or regret over 100 trees at time 10,000 for each algorithm for y. As expected, LinUCT$_{\mathrm{PLAIN}}$ resulted in the highest failure rate (i.e., worst performance) for most cases. UCT outperformed the others for very small trees (e.g., branching factor four), LinUCT$_{\mathrm{RAVE\text{-}FP}}$ performed better with a branching factor of 5-9, and LinUCT$_{\mathrm{FP}}$ achieved good results for trees with a branching factor of greater than 12. These results suggest that LinUCT$_{\mathrm{FP}}$ is effective in games that have a relatively large branching factor. Figure 6 shows that the failure rates and average regrets decreased along with an increased number of playouts. The performance of compared algorithms is similar up to a certain point; however, they differ substantially at time 10,000.

6 Conclusion

We presented a family of LinUCT algorithms that incorporate LinUCB into UCT for tree search: LinUCT$_{\mathrm{PLAIN}}$, LinUCT$_{\mathrm{RAVE}}$, LinUCT$_{\mathrm{FP}}$, and LinUCT$_{\mathrm{RAVE\text{-}FP}}$. LinUCT$_{\mathrm{PLAIN}}$ is the simplest algorithm in which the LinUCB value is used rather than the UCB1 value. However, there is room for improvement. Feature vectors

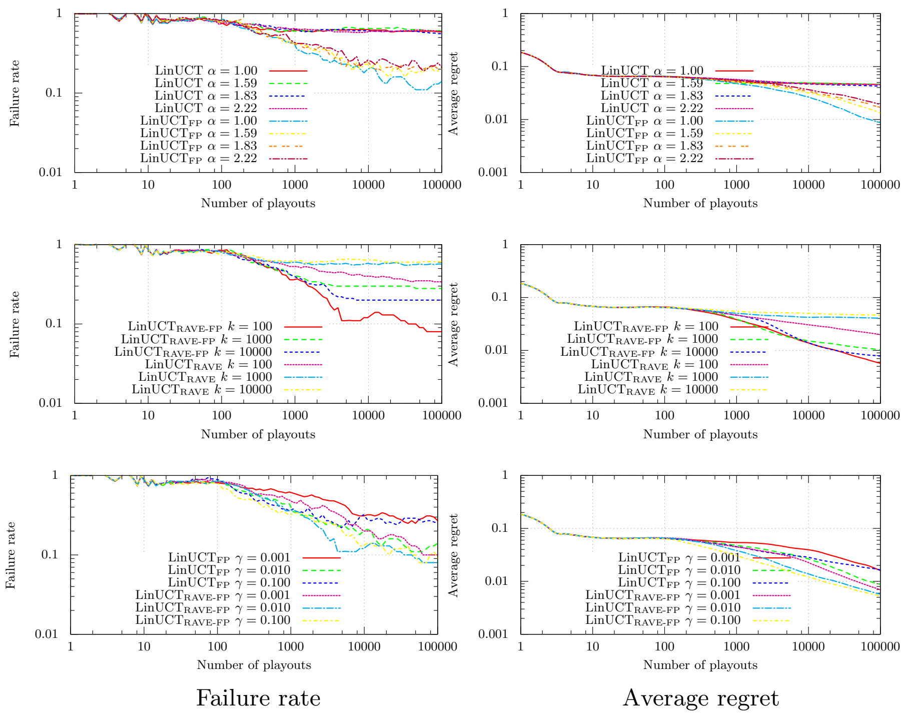
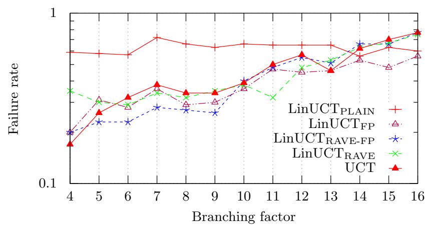


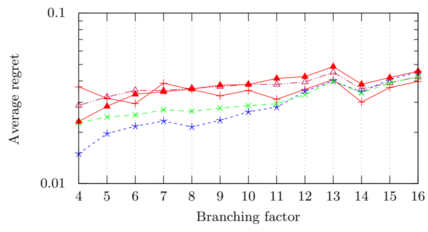
Fig. 4: Effects by constants α, k, and γ: depth=4, width=4, d=8

observed for a node by the algorithm do not contain the information about the structure of the subtree expanded thus far. To address the problem, we incorporated existing techniques: a RAVE framework and feature propagation. $\text{LinUCT}_{\text{RAVE}}$ combines LinUCB and UCB1 in a RAVE framework. $\text{LinUCT}_{\text{FP}}$ is a modified version of $\text{LinUCT}_{\text{PLAIN}}$ in which the feature vectors of descendants are propagated to ancestors. $\text{LinUCT}_{\text{RAVE-FP}}$ is a combination of $\text{LinUCT}_{\text{RAVE}}$ and $\text{LinUCT}_{\text{FP}}$.

Experiments were performed with incremental random trees to assess the proposed algorithms in terms of the failure rates and regrets. In these experiments, each random tree was extended to have its own coefficient vector and feature vectors for each node, where the expected reward at each leaf is defined by the inner product of the feature vector and the coefficient vector. The results obtained with trees of a branching factor of 4-16 showed that $\text{LinUCT}_{\text{RAVE}}$, $\text{LinUCT}_{\text{FP}}$ and $\text{LinUCT}_{\text{RAVE-FP}}$ outperformed UCT, with the exception of small trees, and $\text{LinUCT}_{\text{FP}}$ demonstrated the best performance with a branching factor greater than 11.

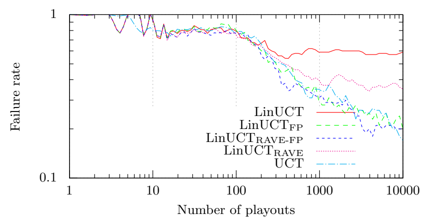


(a) Failure rate

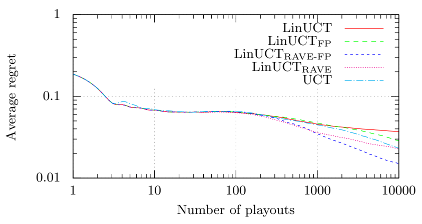


(b) Average regret

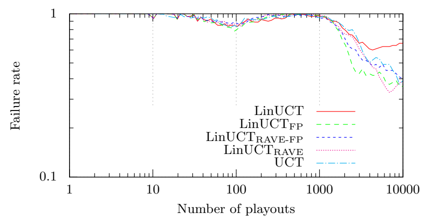
Fig. 5: Comparison of LinUCT algorithms with UCT for various trees: depth=4, d=16



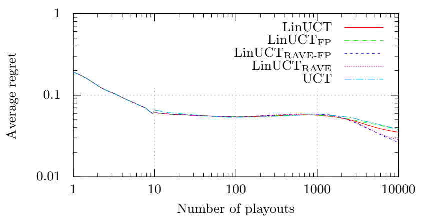
(a) Failure rate (width=4)



(b) Average regret (width=4)



(c) Failure rate (width=10)



(d) Average regret (width=10)

Fig. 6: Performances of each algorithm (depth=4, d=16)

There are two directions for future work. The most important direction for future work would be to examine the practical performance of the proposed family of LinUCT algorithm with major games such as Go. The other interesting direction is convergence analysis of the proposed LinUCT algorithms.

Acknowledgement

A part of this work was supported by JSPS KAKENHI Grant Number 25330432.

References

1. Auer, P., Cesa-Bianchi, N., Fischer, P.: Finite-time analysis of the multiarmed bandit problem. Machine learning 47(2-3), 235–256 (2002)
2. Bouzy, B., Cazenave, T.: Computer Go: An AI-oriented survey. Artificial Intelligence 132(1), 39–103 (2001)
3. Browne, C., Powley, E., Whitehouse, D., Lucas, S., Cowling, P., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., Colton, S.: A survey of monte carlo tree search methods. Computational Intelligence and AI in Games, IEEE Transactions on 4(1), 1–43 (March 2012)
4. Bubeck, S., Cesa-Bianchi, N.: Regret analysis of stochastic and nonstochastic multi-armed bandit problems. Foundations and Trends in Machine Learning 5(1), 1–122 (2012)
5. Chu, W., Li, L., Reyzin, L., Schapire, R.E.: Contextual bandits with linear payoff functions. In: Gordon, G.J., Dunson, D.B., Dudík, M. (eds.) Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics, AISTATS 2011. JMLR Proceedings, vol. 15, pp. 208–214. JMLR.org (2011)
6. Coulom, R.: Computing elo ratings of move patterns in the game of go. ICGA Journal 30(4), 198–208 (2007)
7. Furtak, T., Buro, M.: Minimum proof graphs and fastest-cut-first search heuristics. In: Boutilier, C. (ed.) Proceedings of the 21st IJCAI. pp. 492–498 (2009)
8. Gelly, S., Kocsis, L., Schoenauer, M., Sebag, M., Silver, D., Szepesvári, C., Teytaud, O.: The grand challenge of computer go: Monte carlo tree search and extensions. Commun. ACM 55(3), 106–113 (Mar 2012)
9. Gelly, S., Silver, D.: Combining online and offline knowledge in uct. In: Proceedings of the 24th ICML. pp. 273–280. ACM (2007)
10. Gelly, S., Silver, D.: Monte-carlo tree search and rapid action value estimation in computer go. Artificial Intelligence 175(11), 1856–1875 (2011)
11. Hoki, K., Kaneko, T.: Large-scale optimization for evaluation functions with minimax search. Journal of Artificial Intelligence Research pp. 527–568 (2014)
12. Knuth, D.E., Moore, R.W.: An analysis of alpha-beta pruning 6(4), 293–326 (1975)
13. Kocsis, L., Szepesvári, C.: Bandit based monte-carlo planning. In: Machine Learning: ECML 2006, pp. 282–293. Springer (2006)
14. Korf, R.E., Chickering, D.M.: Best-first minimax search. Artificial Intelligence 84, 299–337 (1996)
15. Li, L., Chu, W., Langford, J., Schapire, R.E.: A contextual-bandit approach to personalized news article recommendation. In: Proceedings of the 19th international conference on World wide web. pp. 661–670. ACM (2010)
16. Rosin, C.: Multi-armed bandits with episode context. Annals of Mathematics and Artificial Intelligence pp. 1–28 (2011)
17. Silver, D., Tesauro, G.: Monte-carlo simulation balancing. In: Proceedings of the 26th Annual ICML. pp. 945–952. ACM (2009)
18. Smith, S.J., Nau, D.S.: An analysis of forward pruning. In: AAAI. pp. 1386–1391 (1994)
19. Walsh, T.J., Szita, I., Diuk, C., Littman, M.L.: Exploring compact reinforcement-learning representations with linear regression. In: Proceedings of the Twenty-Fifth Conference on Uncertainty in Artificial Intelligence. pp. 591–598. AUAI Press (2009)
20. Yoshizoe, K., Kishimoto, A., Kaneko, T., Yoshimoto, H., Ishikawa, Y.: Scalable distributed monte-carlo tree search. In: the 4th SoCS. pp. 180–187 (2011)