

# A Dive to MCTS

liuhanzuo

January 7, 2025

## Abstract

This survey explores the Monte Carlo Tree Search (MCTS) algorithm, a versatile decision-making framework with applications across various domains. It begins by reviewing the foundational concepts of Markov Decision Processes (MDP) and Monte Carlo methods, which form the basis of MCTS. The core MCTS algorithm is examined, highlighting its four main steps: selection, expansion, simulation, and backpropagation. Several MCTS variants are discussed, including reward-based, visited-based, and hybrid approaches, each balancing exploration and exploitation in unique ways. The survey also covers advanced algorithms such as Upper Confidence Bounds for Trees (UCT) and its variations, including UCB1-tuned and Bayesian UCT, emphasizing both theoretical underpinnings and practical implementations. Additionally, the integration of learning techniques like Temporal Difference (TD) learning, its Monte Carlo variant (TDMC), and Single Player MCTS (SP-MCTS) within MCTS is explored. Experimental results are presented in two parts: the first, demonstrating the effectiveness of advanced MCTS algorithms in the context of Tic-Tac-Toe; the second, introducing and comparing a new state-of-the-art experimental UCT-tuned algorithm for the Gobang game. This survey offers a comprehensive overview of MCTS and its applications, providing valuable insights for both researchers and practitioners in the field.

## 1 Background

### 1.1 MDP

We start the background part by introducing Markov Decision Process(MDP). The basic elements of MDP are as follows:

- State Space  $\mathcal{S}$ : The set of all possible states.
- Action Space  $\mathcal{A}$ : The set of all possible actions.
- Transition Probability  $T(s', s, a)$ : The probability of transitioning to state  $s'$  given that the current state is  $s$  and the action taken is  $a$ .
- Reward Function  $R(s, a, s')$ : The reward received after transitioning from state  $s$  to state  $s'$  by taking action  $a$ .

The goal of MDP is to find a policy  $\pi$  that maximizes the expected cumulative reward. The policy  $\pi$  is a mapping from states to actions. Which indicates our strategy or policy for a given state. Here, we need to note that we usually choose the  $\pi$  to maximize the reward.

## 1.2 Monte Carlo Methods

From the MDP we can see that: when we are making a decision, what we really care about is the "expected reward" or "potential reward" for taking an action  $a$  in state  $s$ . The Monte Carlo Methods is a class of algorithms that estimate the expected reward by sampling the environment. The basic idea is to simulate the environment and collect the reward. The expected reward is then estimated by the average of the collected rewards. The Monte Carlo Methods are widely used in reinforcement learning, game theory, and other fields.

To be more specific, we define the Q-value function as:

$$Q(s, a) = \frac{1}{N(s, a)} \sum_{i=1}^{N(s)} \mathbb{I}_{s,a} R_i$$

Where the definition is:

- $N(s, a)$  is the number of times action  $a$  has been taken in state  $s$ .
- $N(s)$  is the number of times state  $s$  has been visited. i.e.  $N(s) = \sum_a N(s, a)$
- $\mathbb{I}_{s,a}$  is the indicator function that is 1 if the action  $a$  is taken in state  $s$  and 0 otherwise.
- $R_i$  is the reward received after taking action  $a$  in state  $s$ .
- $Q(s, a)$  is the estimated Q-value of taking action  $a$  in state  $s$  (expected reward).

At this point, we have delved into the core aspect of the system: exploration. This involves devising a strategy to effectively explore the state space and accurately estimate the Q-values, thereby determining the optimal policy. Monte Carlo Methods provide a robust approach to address this challenge.

## 1.3 Advanced Works

AlphaGo[8], developed by DeepMind, marks a significant advancement in artificial intelligence, particularly in game playing. It integrates Monte Carlo Tree Search (MCTS) with deep neural networks to achieve superhuman performance in Go. The neural networks evaluate board positions and suggest moves, while MCTS explores promising move sequences, effectively balancing exploration and exploitation. AlphaGo's prowess was demonstrated in its 2016 victory against top Go player Lee Sedol. Subsequently, DeepMind introduced AlphaGo Zero, which learns to play Go from scratch without human data, relying solely on self-play to train its neural networks. This approach enables the discovery of novel strategies and tactics. AlphaGo Zero's success underscores the potential of combining reinforcement learning with advanced search algorithms, heralding future AI advancements. AlphaGo Zero, an evolution of AlphaGo, further refines the integration of MCTS with deep learning. Unlike its predecessor, which relied on human game data for training, AlphaGo Zero learns solely through self-play. It uses a deep neural network to evaluate board positions and predict move probabilities, which are then used to guide the MCTS simulations. The neural network is trained using the outcomes of these simulations, creating a feedback loop that continually improves both the network and the MCTS policy. This approach allows AlphaGo Zero to discover novel strategies and achieve superhuman performance without any human input, demonstrating the power of combining MCTS with deep reinforcement learning.

## 2 Monte Carlo Tree Search

### 2.1 Introduction

This section delves into the Monte Carlo Tree Search (MCTS) family of algorithms. MCTS is based on two core principles: approximating the true value of an action through random simulations and efficiently using these values to steer the policy towards a best-first strategy. The algorithm incrementally constructs a partial game tree, leveraging the outcomes of prior explorations to refine move value estimates. These estimates, especially for the most promising moves, become increasingly accurate as the tree expands.

In this section, we will detail several representative algorithms, present their results, and elucidate the underlying intuitions.

### 2.2 General MCTS

The general MCTS algorithm is as follows:

---

**Algorithm 1** General MCTS

---

```
1: function MCTS( $s_0$ )
2:   create a node  $v_0$  with state  $s_0$ 
3:   while within computational budget do
4:      $v_l \leftarrow \text{TreePolicy}(v_0)$ ,  $\Delta \leftarrow \text{DefaultPolicy}(s(v_l))$ 
5:      $\text{BACKUP}(v_l, \Delta)$ 
6:   end while
7:   return  $\text{BESTCHILD}(v_0)$ 
8: end function
```

---

Basically, we have four steps:

1. **Selection:** Start from the root point, we use a policy – child selecting policy to select the next node to expand. (run the `TreePolicy` function)
2. **Expansion:** Expand the selected node and get the next node.
3. **Simulation:** A simulation about what would happen to run on the new point. (run the `DefaultPolicy` function). Typically, we would use a simple(default) strategy to find out the result.
4. **BackPropagation:** Update the information of the nodes on the path from the root to the leaf node.

This four steps can be also explained as a psuedo code at 1.

After the training steps (what we mentioned above), we get a strategy that label the reward of each action in the current state. For each state, we should run a `BestChild` function to get the best action to take.

The choice of `BestChild` function is mainly listed below:

1. **reward-based:** choose the child with highest-reward.
2. **visited-based:** choose the child with highest-visited times.

3. **hybrid-based**: mix the above two policy with a factor, balance the reward and visited times.

Afterwards, we will enumerate some specific algorithms based on the general MCTS algorithm – mainly focus on how to choose **TreePolicy** and **DefaultPolicy**.

## 2.3 Reward-based MCTS

Reward-based MCTS is the basic version of MCTS. It uses the reward to guide the exploration. The **TreePolicy** function selects the next node to expand based on the estimated Q-value of the nodes. The **DefaultPolicy** function uses a random policy to simulate the environment. The **Backup** function updates the Q-values of the nodes on the path from the root to the leaf node.

Each step, it chooses the **BestChild** with the highest Q-value to expand. The intuition behind this is that the children with higher Q-values are more likely to lead to a higher reward.

Since it is straight to understand this algorithm, we will not show the psuedo code here.

There is also some variant of this kind of MCTS:

- visited-based MCTS: simply change the choice of **BestChild** function to choose the child with highest visited times. The intuition for this algorithm is that during the exploration, the high-visited node is likely to be the right choice (and the reward is also high).
- randomized MCTS: with probability of  $p$  we randomly explore a point in the tree, with probability of  $1 - p$  we choose the best child.
- hybrid MCTS: mix the above two policy with a factor, balance the reward and visited times. Use a factor  $\alpha$  to control the balance.

## 2.4 SOTA:UCT

UCT is a leading algorithm in Monte Carlo Tree Search for games and Markov Decision Processes, designed to minimize cumulative regret.

**Definition 1** Scheme UCB(c) pulls arm  $i$  that maixmizes upper confidence bound  $b_i$  on the reward

$$b_i = \overline{X}_i + \sqrt{\frac{c \log n}{n_i}}$$

where  $\overline{X}_i$  is the average sample reward obtained from arm  $i$ ,  $n_i$  is the number of times arm  $i$  was pulled, and  $n$  is the total number of pulls so far.

**Definition 2** The simple regret  $\mathbb{E}r$  of a sampling policy for a Multi-armed Bandit Problem is the expected difference between the best true expected reward  $\mu_*$  and the true expected reward  $\mu_j$  of the arm with greatest sample mean  $j = \arg \max_i \overline{X}_i$

$$\mathbb{E}r = \sum_{j=1}^K \Delta_j \Pr[j = \arg \max_i \overline{X}_i], \text{ where } \Delta_j = \mu_* - \mu_j$$

**Definition 3**  $\epsilon$ -greedy sampling pulls current greatest sample mean arm w.p.  $\epsilon$ , others w.p.  $\frac{1-\epsilon}{K-1}$

**Definition 4** Scheme UCB pulls arm  $i$  that maximize  $b_i$ , where

$$b_i = \overline{X}_i + \sqrt{\frac{c\sqrt{n}}{n_i}}$$

**Theorem 1**  $\forall \eta \in (0, 1), \gamma > 1$ , there is a  $N$  such that for any numbers of sampling  $n > N$ , the simple regret of  $\epsilon$ -greedy is bounded by

$$\mathbb{E}r_{\epsilon\text{-greedy}} \leq 2\gamma \sum_{i=1}^K \Delta_i \exp \left( \frac{-2\Delta_j^2 n \epsilon}{\left(1 + \sqrt{\frac{(K-1)\epsilon}{1-\epsilon}}\right)^2} \right)$$

*Proof.* From Chernoff-Hoeffding bound, we have

$$P_i \leq \Pr[\overline{X}_i > \mu_i + \delta_i] + \Pr[\overline{X}_* < \mu_* - (\Delta_i - \delta_i)] \leq \exp(-2\delta_i^2 n_i) + \exp(-2(\Delta_i - \delta_i)^2 n_i)$$

The first inequaility is obtained from the fact that if  $\overline{X}_* \geq \mu_* - (\Delta_i - \delta_i)$  and  $\overline{X}_i \leq \mu_i + \delta_i$ , we always have  $\overline{X}_* \geq \overline{X}_i - \Delta_i$ , will not cause a simple regret.

Observe that when  $n \rightarrow \infty$ ,  $\overline{X}_i \rightarrow \mu_i$ , therefore  $n_* \rightarrow n\epsilon$ ,  $n_i \rightarrow \frac{n(1-\epsilon)}{K-1}$ .

Conclude for  $\eta \in (0, 1)$  and  $\gamma > 1$ , use the approximation above, we have

$$P_i \leq \gamma \left( \exp\left(\frac{-2\delta_i^2 n(1-\epsilon)}{K-1}\right) + \exp(-2(\Delta_i - \delta_i)^2 n\epsilon) \right)$$

we choose the adequate  $\delta_i$  such that

$$\exp\left(-\frac{2\delta_i^2 n(1-\epsilon)}{K-1}\right) = \exp(-2(\Delta_i - \delta_i)^2 n\epsilon), \frac{\delta_i}{\Delta_i - \delta_i} = \sqrt{\frac{(K-1)\epsilon}{1-\epsilon}}$$

Then we meet our conclusion. □

**Theorem 2** For every  $0 < \eta < 1$  and  $\gamma > 1$  there exists  $N$  such that for any number of samples  $n > N$  the simple regret of the  $\text{UCB}\sqrt{\cdot}(c)$  sampling scheme is bounded from above as

$$\mathbb{E}r_{\text{ucb}\sqrt{\cdot}} \leq 2\gamma \sum_{i=1}^K \Delta_i \exp\left(-\frac{c\sqrt{n}}{2}\right)$$

with probability at least  $1 - \eta$ .

*Proof.* We start by bounding the probability  $P_i$  that a non-optimal arm  $i$  is chosen. Split the interval  $[\mu_i, \mu_*]$  at  $\mu_i + \frac{\Delta_i}{2}$ . Apply the Chernoff-Hoeffding bound to get:

$$P_i \leq \Pr\left(X_i > \mu_i + \frac{\Delta_i}{2}\right) + \Pr\left(X_* < \mu_* - \frac{\Delta_i}{2}\right) \leq \exp\left(-\frac{\Delta_i^2 n_i}{2}\right) + \exp\left(-\frac{\Delta_i^2 n_*}{2}\right)$$

Observe that, in probability,  $n_i \rightarrow \frac{c\sqrt{n}}{\Delta_i^2}$ ,  $n_i \leq n_*$  as  $n \rightarrow \infty$ . Conclude that for every  $0 < \eta < 1$ ,  $\gamma > 1$  there exists  $N$  such that for every  $n > N$  and all non-optimal arms  $i$ :

$$P_i \leq 2\gamma \exp\left(-\frac{c\sqrt{n}}{2}\right)$$

Substitute this into the expression for simple regret:

$$\mathbb{E}r_{\text{ucb}\sqrt{\cdot}} \leq 2\gamma \sum_{i=1}^K \Delta_i \exp\left(-\frac{c\sqrt{n}}{2}\right)$$

□

The Upper Confidence bounds for Trees (UCT) algorithm [12] is a popular variant of MCTS. It uses the UCB1 formula to balance exploration and exploitation. The UCB1 formula is given by:

$$UCB1 = \frac{Q(s, a)}{N(s, a)} + C \sqrt{\frac{\ln N(s)}{N(s, a)}}$$

where  $C$  is a constant that controls the balance between exploration and exploitation.  $N(s, a)$  is the number of times action  $a$  has been taken in state  $s$ .  $N(s)$  is the number of times state  $s$  has been visited.  $Q(s, a)$  is the estimated Q-value of taking action  $a$  in state  $s$ . The UCT algorithm uses the UCB1 formula to select the next node to expand.

Here let's consider the intuition for the UCT algorithm.

- For a point that is never visited before, the  $N(s, a)$  term is zero, thus its UCB1 value is infinite large, will be more likely to explore.
- For a point that is visited many times, the  $Q(s, a)$  term will be more important, which means that the point with higher reward will be more likely to be selected.

- If all points in the graph is visited, the  $N(s, a)$  term will be non-zero. Thus a larger  $C$  means that we focus more on exploration. A smaller  $C$  value, means that we focus more on exploitation.
- As a comparison to reward-based MCTS, the UCT algorithm solve the problem of insufficient exploration of node, which often cause reward-based MCTS to get stuck in a local minimum. Moreover, it consider the  $Q$  value and the visited times of the node, which is more reasonable.

Now we show the psuedo code for the UCT algorithm: ([2] is our reference).

---

**Algorithm 2** UCT

---

```

1: function UCT( $s_0$ )
2:   create a node  $v_0$  with state  $s_0$ 
3:   while within computational budget do
4:      $v_l \leftarrow \text{TREEPOLICY}(v_0)$ ,  $\Delta \leftarrow \text{DEFAULTPOLICY}(s(v_l))$ 
5:     BACKUP( $v_l, \Delta$ )
6:   end while
7:   return BESTCHILD( $v_0$ )
8: end function
9: function TREEPOLICY( $v$ )
10:  while  $v$  is not a terminal node do
11:    if  $v$  is not fully expanded then
12:      return EXPAND( $v$ )
13:    else
14:       $v \leftarrow \text{BESTCHILD}(v)$ 
15:    end if
16:  end while
17:  return  $v$ 
18: end function
19: function EXPAND( $v$ )
20:  choose  $a \in A(s(v))$ 
21:  add a new node  $v'$  as a child of  $v$ 
22:   $s(v') = f(s(v), a)$ 
23:   $a(v') = v$ 
24:  return  $v'$ 
25: end function
26: function BESTCHILD( $v$ )
27:  return  $\arg \max_{v' \in \text{children}(v)} \left( \frac{Q(v')}{N(v')} + C \sqrt{\frac{\ln N(v)}{N(v')}} \right)$ 
28: end function
29: function DEFAULTPOLICY( $s$ )
30:  while  $s$  is not a terminal state do
31:    choose  $a \in A(s)$ 
32:     $s \leftarrow f(s, a)$ 
33:  end while

```

```

34:   return  $R(s)$ 
35: end function
36: function BACKUP( $v, \Delta$ )
37:   while  $v$  is not null do
38:      $N(v) \leftarrow N(v) + 1$ 
39:      $Q(v) \leftarrow Q(v) + \Delta$ 
40:      $v \leftarrow a(v)$ 
41:   end while
42: end function

```

---

### 3 Algorithm Variations

In this part, we will introduce more algorithms based on the UCT algorithm.

#### 3.1 Bandit Algorithm

The definitions we used are as follows:

- $X_{i,n_i} = \frac{1}{n_i} \sum_{t=1}^{n_i} x_t$
- $X_{d',n_{d'}}$ : The reward received at node  $d'$  after  $n_{d'}$  visits.
- $n_d$ : The number of times the node of depth  $d$  in the optimal branch is reached.
- $n$ : The first instant when the optimal leaf is reached.
- $D$ : The depth of the tree.
- $s$ : A constant used in the confidence sequence.
- 

$$B_{i,p,n_i} \stackrel{\text{def}}{=} X_{i,n_i} + \sqrt{\frac{2 \log(p)}{n_i}}$$

The pseudo Code for bandit algorithm is as follows:

---

#### Algorithm 3 Bandit Algorithm

---

```

1: for  $n \geq 1$  do
2:   Run  $n$ -th trajectory from the root to leaf
3:   set current node  $i_0$  to root
4:   for  $d = 1 \cdots D$  do
5:     Select node  $i_d$  as the children  $j$  of node  $i_{d-1}$  that maximizes  $B_{j,n_{i_{d-1}},n_j}$ 
6:   end for
7: end for
8: receive reward  $x_n \sim^{\text{iid}} X_{i_D}$ 
9: for  $d = D \cdots 0$  do
10:  Update the number of visits  $n_{i_d} = n_{i_d} + 1$ 
11:  Update the bound  $B_{i_d,n_{i_{d-1}},n_{i_d}}$ 

```



12: end for

Coquelin and Munos propose flat UCB which effectively treats the leaves of the search tree as a single multiarmed bandit problem.[4] This is distinct from flat Monte Carlo search, in which the actions for a given state are uniformly sampled and no tree is built. Coquelin and Munos demonstrate that flat UCB retains the adaptivity of standard UCT while improving its regret bounds in certain worst cases where UCT is overly optimistic.

**Intuition:** UCT may spend many time exploring an area that there is no best reward. Let's provide an example: the Monte Carlo tree has  $m$  stages, while the  $i$ -th stage has a decision to get a reward of  $\frac{m-(i+1)}{m}$  directly, while in the  $m$ th stage, you could get a reward of 1. However, since the UCT algorithm will tend to visit the first stage with a reward of  $\frac{m-1}{m}$ , it will cost the UCT algorithm many times to explore the last stage whose reward is 1. (It will take UCT  $O(\exp(\exp m))$  time).

**Definition 5**  $n_d$  is the number of times that a node of depth  $d$  in the optimal branch is reached.

**Definition 6**  $n$  is the first instant when the optimal leaf is reached.  $D$  is the depth of the tree.  $s$  is a constant used in the confidence sequence.

**Definition 7**  $X_{d',n_{d'}}$  is the reward received at node  $d'$  after  $n_{d'}$  visits.

**Definition 8**  $B_{i,p,n_i}$  is the upper confidence bound on the reward of arm  $i$  after  $n_i$  visits.

**Theorem 3**

$$n_{d-1} \geq \exp\left(\frac{n_d}{2D^2}\right)$$

To conclude, UCT has asymptotically regret  $O(\log(n))$  in  $n$ , (or  $O(\sqrt{n})$  for the square root sequence)

*Proof.* We now establish a lower bound on the number of times suboptimal rewards are received before obtaining the optimal reward for the first time. Let  $n$  denote the first instance when the optimal leaf is reached, and  $n_d$  the number of times the node at depth  $d$  in the optimal branch is reached. Thus,  $n = n_0$  and  $n_D = 1$ . At depth  $D - 1$ , we have  $n_{D-1} = 2$  since action 2 has been chosen once in node  $D - 1$ .

Consider the square root confidence sequence (2). At depth  $d - 1$ , since the optimal branch is followed by the  $n$ -th trajectory, we have (denoting  $d'$  as the node resulting from action 2 in node  $d - 1$ ):

$$X_{d',n_{d'}} + \frac{s\sqrt{n_{d-1}}}{n_{d'}} \leq X_{d,n_d} + \frac{s\sqrt{n_{d-1}}}{n_d}.$$

Given  $X_{d',n_{d'}} = \frac{D-d}{D}$  and  $X_{d,n_d} \leq \frac{D-(d+1)}{D}$  since the optimal reward has not been received before, we deduce:

$$\frac{1}{D} \leq \frac{s\sqrt{n_{d-1}}}{n_d}.$$

Thus, for the square root confidence sequence,  $n_{d-1} \geq \frac{n_d^2}{D^4}$ . By induction:

$$n \geq \frac{n_1^2}{D^4} \geq \frac{n_2^2}{2D^4(1+2)} \geq \frac{n_3^2}{3D^4(1+2+3)} \geq \cdots \geq \frac{n_{D-1}^2}{(D-1)D^{2D(D-1)}}.$$

Since  $n_{D-1} = 2$ , we obtain  $n \geq \frac{2^{2D-1}}{D^{2D(D-1)}}$ , indicating a double exponential dependency with respect to  $D$ . For instance, for  $D = 20$ ,  $n \geq 10^{156837}$ . Consequently, the regret is  $\Omega(\exp(\exp(D)))$ .

For the logarithmic confidence sequence defined by (1), we similarly show  $n_{d-1} \geq \exp(n_d/(2D^2))$ , thus:

$$n \geq \exp(\exp(\cdots \exp(2) \cdots)) \quad (\text{composition of } D-1 \text{ exponential functions}).$$

Therefore, although the UCT algorithm has asymptotic regret  $O(\log(n))$  in  $n$  (or  $O(\sqrt{n})$  for the square root sequence), the transitory regret is  $\Omega(\exp(\exp(\cdots \exp(2) \cdots)))$  (or  $\Omega(\exp(\exp(D)))$  in the square root sequence).  $\square$

The reason for this bad behavior is that the algorithm is too optimistic (it does not explore enough and may take a very long time to discover good branches that looked initially bad) since the bounds (1) and (2) are not true upper bounds.

## 4 Learning in MCTS

### 4.1 Temporal Difference Learning

The Temporal Difference (TD) learning algorithm [10] is a model-free reinforcement learning algorithm that learns the value function by bootstrapping. Based on the Bellman equation, which states that the value of a state is equal to the immediate reward plus the discounted value of the next state. Also, it uses the difference between the estimated value and the target value to update the value function.

Here we show the psuedo code for the TD learning algorithm:

---

**Algorithm 4** Temporal Difference Learning

---

```

1: function TD( $s_0$ )
2:   initialize the value function  $V(s)$ 
3:    $s \leftarrow s_0$ 
4:   while  $s$  is not a terminal state do
5:     choose  $a$  from  $A(s)$ 
6:      $s' \leftarrow f(s, a)$ 
7:      $r \leftarrow R(s, a, s')$ 
8:      $V(s) \leftarrow V(s) + \alpha(r + \gamma V(s') - V(s))$ 
9:      $s \leftarrow s'$ 
10:  end while
11: end function

```

---

The algorithm 4 illustrates a typical TD learning algorithm, where  $\alpha$  represents the learning rate,  $\gamma$  is the discount factor,  $r$  denotes the immediate reward, and  $s'$  is the subsequent state.

**Intuition:** Unlike MCTS, the TD learning algorithm updates the value function through bootstrapping, meaning it does not require a model of the environment. Instead, it directly uses the rewards obtained from the environment to update the value function.

**Challenges:** Gerald Tesauro highlighted in [9] that the TD learning algorithm suffers from high variance due to the noisy and often inaccurate rewards from the environment. This high variance can lead to instability and divergence, particularly when combined with function approximation and off-policy learning. For instance, when Tesauro trained the TD learning algorithm to play backgammon, the network (a feedforward fully-connected architecture with either no hidden units or a single hidden layer with 10 to 40 hidden units) achieved only a 40% to 60% win rate against top human players.

The TDMC algorithm, proposed by Yasuhiro OSAKI [6], addresses these issues.

**Intuition:** TDMC leverages Monte Carlo simulations to estimate the value function, thereby reducing variance by averaging the results of multiple simulations. This approach provides a more accurate estimate of the expected reward, enhancing policy evaluation and improvement. By combining the strengths of Temporal Difference learning and Monte Carlo methods, TDMC offers a more stable and reliable learning process.

Below is the pseudocode for the TDMC algorithm:

---

**Algorithm 5** TDMC Algorithm

---

```

1: function TDMC( $s_0, \alpha, \gamma, \lambda, T$ )
2:   initialize the feature weights  $w$ 
3:   for each episode do
4:     initialize the state  $s \leftarrow s_0$ 
5:     initialize the eligibility traces  $e \leftarrow 0$ 
6:     for  $t = 1, 2, \dots, T - 1$  do
7:       observe features  $x_t$  of state  $s$ 
8:       compute the value  $V(x_t, w) \leftarrow x_t \cdot w$ 
9:       perform action  $a$  and observe reward  $r_t$  and next state  $s'$ 
10:      simulate the environment to get  $r_i$  for  $i = t + 1, \dots, T$ 
11:      compute the return  $R_t \leftarrow \sum_{i=t}^{T-1} \gamma^{i-t} r_i$ 
12:      compute the n-step return  $(n)R_t \leftarrow r_t + \gamma r_{t+1} + \dots + \gamma^{n-1} r_{t+n-1} + \gamma^n V(x_{t+n}, w)$ 
13:      compute the  $\lambda$ -return  $\lambda R_t \leftarrow (1 - \lambda) \sum_{n=1}^{T-t} \lambda^{n-1} (n)R_t + \lambda^{T-t} R_t$ 
14:      compute the gradient  $\nabla V(x_t, w)$ 
15:      update the eligibility traces  $e \leftarrow \gamma \lambda e + \nabla V(x_t, w)$ 
16:      update the weights  $w \leftarrow w + \alpha (\lambda R_t - V(x_t, w)) e$ 
17:       $s \leftarrow s'$ 
18:     end for
19:   end for
20: end function

```

---

## 4.2 Single-Player MCTS

The article [7] highlights the limited application of modern MCTS in single-player games and proposes modifications to the UCT algorithm for the SameGame. Traditional search algorithms like  $A^*$  and  $IDA^*$  are less effective in this context due to their reliance on evaluation functions.

SP-MCTS modifies the UCT selection strategy as follows:

$$\frac{Q(v')}{N(v')} + C\sqrt{\frac{\log N(v)}{N(v')}} + \sqrt{\frac{\sum_{u \text{ is } v's \text{ descendent}} N(u)^2 - Q(v')^2/N(v') + D}{N(v')}}$$

Here,  $\sum_{u \text{ is } v's \text{ descendent}} N(u)^2 - Q(v')^2/N(v')$  represents the variance at node  $v$ , indicating potential deviations of the child node.

#### Key Differences and Intuitions:

- **Selection Policy:** Unlike two-player games with bounded scores, single-player games often have unbounded rewards. Proper scaling of  $C$  and  $D$  is necessary to fit the reward range.
- **Simulation Strategy:** Utilizes a "TabuRandom" strategy to enhance random simulations by prohibiting the selection of a randomly chosen color unless no other moves are available.
- **Back-Propagation Strategy:** Propagates the average score, the sum of squared results, and the best score achieved so far.
- **Final Play:** Allows for complete process search at the game's start, eliminating the need to wait for another player's response.

## 5 UCT-based algorithm and Tree MCTS

### 5.1 UCB1-tuned

The UCB1-Tuned algorithm is an enhancement suggested by Auer et al. [1] to tune the bounds of UCB1 more finely. It replaces the upper confidence bound  $\sqrt{\frac{2\ln n}{n_j}}$  with:

$$\sqrt{\frac{\ln n}{n_j} \min\left(\frac{1}{4}, V_j(n_j)\right)}, V_j(s) = \left(\frac{1}{2} \sum_{\tau=1}^s X_{j,\tau}^2 - \bar{X}_{j,s}^2 + \sqrt{\frac{2\log t}{s}}\right)$$

where  $V_j(n_j)$  is the empirical variance of the rewards obtained from action  $j$  after  $n_j$  plays. The UCB1-Tuned algorithm aims to reduce the exploration of suboptimal actions by incorporating the variance of the rewards into the confidence bounds.

The pseudo code for the UCB1-Tuned algorithm is as follows:

---

#### Algorithm 6 UCB1-Tuned

---

```

1: function UCB1-TUNED( $s_0$ )
2:   create a node  $v_0$  with state  $s_0$ 
3:   while within computational budget do
4:      $v_l \leftarrow \text{TREEPOLICY}(v_0)$ ,  $\Delta \leftarrow \text{DEFAULTPOLICY}(s(v_l))$ 
5:     BACKUP( $v_l$ ,  $\Delta$ )
6:   end while
7:   return BESTCHILD( $v_0$ )
8: end function
9: function TREEPOLICY( $v$ )
10:  while  $v$  is not a terminal node do

```

```

11:     if  $v$  is not fully expanded then
12:         return EXPAND( $v$ )
13:     else
14:          $v \leftarrow \text{BESTCHILD}(v)$ 
15:     end if
16: end while
17: return  $v$ 
18: end function
19: function EXPAND( $v$ )
20:     choose  $a \in A(s(v))$ 
21:     add a new node  $v'$  as a child of  $v$ 
22:      $s(v') = f(s(v), a)$ 
23:      $a(v') = v$ 
24:     return  $v'$ 
25: end function
26: function BESTCHILD( $v$ )
27:     return

```

$$\arg \max_{v' \in \text{children}(v)} \left( \frac{Q(v')}{N(v')} + \sqrt{\frac{\ln N(v)}{N(v')} \min \left( \frac{1}{4}, \frac{V(v')}{N(v')} + \sqrt{\frac{2 \ln N(v)}{N(v')}} \right)} \right)$$

```

28: end function
29: function DEFAULTPOLICY( $s$ )
30:     while  $s$  is not a terminal state do
31:         choose  $a \in A(s)$ 
32:          $s \leftarrow f(s, a)$ 
33:     end while
34:     return  $R(s)$ 
35: end function
36: function BACKUP( $v, \Delta$ )
37:     while  $v$  is not null do
38:          $N(v) \leftarrow N(v) + 1$ 
39:          $Q(v) \leftarrow Q(v) + \Delta$ 
40:          $v \leftarrow a(v)$ 
41:     end while
42: end function

```

---

## 5.2 Bayesian UCT

Tesauro[11] propose that the Bayesian framework potentially allows much more accurate estimation of node values and node uncertainties from limited numbers of simulation trials.

**Intuition:** They change their Bayesian MCTS to Bayes-UCT1

$$\text{maximise } B_i = \mu_i + \sqrt{\frac{2 \log N}{n_i}}$$

Where  $\mu_i$  replaces the average reward of the node with the mean of an extremum (minimax) distribution  $P_i$ .

Bayes-UCT2 is

$$\text{maximise } B_i = \mu_i + \sqrt{2 \log N \sigma_i}$$

Where we choose  $\sigma_i$  as the mean square root of the variance of  $P_i$ ,  $\sigma^2 \sim \frac{1}{n_i}$

Note that Tesauro mentioned that the first equation is a strictly improvement towards UCT if the independence assumption and leaf node priors are correct.

**Theorem 4** central limit theorem: whenever the initial distribution is, as long as we sample many times, the distribution of the sample mean will converge to a normal distribution.

Based on this central limit theorem, they use Bayes-UCT2 to estimate the value of the node.

**Methodology:** they use beta function priors  $x^{\alpha-1}(1-x)^{\beta-1}/B(\alpha, \beta)$ , where  $\alpha$  and  $\beta$  is effectively the number of wins and losses. In the 0/1 cases, after  $W$  sample wins and  $L$  samples losses, we add  $W$  to  $\alpha$  and  $L$  to  $\beta$ . The back propagation part is same:

$$P_{\max}(X) = \sum_i P_i(X) \prod_{j \neq i} C_j(X)$$

Where  $C_j(x)$  is the CDF of  $P_j$  i.e.  $C_{\max} = \int P_{\max}(X) dX$ ,  $C_{\max}(X) = \prod_i C_i(X)$ , min part is similar.

**Convergence Proof** (proof sketches, from [11]):

**Lemma 1** Consider a bandit arm leaf node which generates 0/1 payoffs at a true win rate of  $\mu^*$  such that  $0 < \mu^* < 1$ . Assume a prior probability distribution  $P_{\text{prior}}(p)$  which is strictly positive on  $(0, 1)$  and which has bounded derivatives. Then as the number of samples  $n \rightarrow \infty$ , the posterior distribution  $P_{\text{post}}(p)$  converges to  $\delta(p - \mu^*)$  with probability 1.

*Proof.* Write the posterior by Bayes' theorem,

$$P_{\text{post}}(p) = \frac{P_{\text{prior}}(p) \cdot P(\text{data} | p)}{P(\text{data})}$$

where  $P(\text{data} | p)$  is the likelihood of the observed data given  $p$ , and  $P(\text{data})$  is the marginal likelihood. Differentiating the log of the posterior,

$$\frac{d}{dp} \log P_{\text{post}}(p) = \frac{d}{dp} (\log P_{\text{prior}}(p) + \log P(\text{data} | p) - \log P(\text{data}))$$

Since  $P(\text{data})$  is independent of  $p$ , we have

$$\frac{d}{dp} \log P_{\text{post}}(p) = \frac{d}{dp} \log P_{\text{prior}}(p) + \frac{d}{dp} \log P(\text{data} | p)$$

The likelihood  $P(\text{data} \mid p)$  for  $n$  samples with  $W$  wins and  $L$  losses is given by the binomial distribution,

$$P(\text{data} \mid p) = p^W (1 - p)^L$$

Thus,

$$\frac{d}{dp} \log P(\text{data} \mid p) = \frac{W}{p} - \frac{L}{1 - p}$$

Setting the derivative of the log-posterior to zero to find the peak,

$$\frac{d}{dp} \log P_{\text{post}}(p) = \frac{d}{dp} \log P_{\text{prior}}(p) + \frac{W}{p} - \frac{L}{1 - p} = 0$$

At  $p = \mu^*$ , the correction term is  $O(1/\sqrt{n})$ . The second derivative is

$$\frac{d^2}{dp^2} \log P_{\text{post}}(p) = \frac{d^2}{dp^2} \log P_{\text{prior}}(p) - \frac{W}{p^2} - \frac{L}{(1 - p)^2}$$

At  $p = \mu^*$ ,

$$\frac{d^2}{dp^2} \log P_{\text{post}}(p) = -\frac{n}{\mu^*(1 - \mu^*)} + O(\sqrt{n})$$

Thus, the posterior approaches a Gaussian with mean  $\mu^*$  and variance  $\mu^*(1 - \mu^*)/n$  as  $n \rightarrow \infty$ .  $\square$

**Theorem 5** (Off-policy convergence) Consider a fixed finite-sized bandit tree with binary reward leaf nodes and priors as per Lemma 1. Assume no two sibling nodes have exactly identical minimax payoff rates. Then for any sampling policy  $\Pi$  (e.g., uniform random sampling) that samples all leaf nodes an unbounded number of times as the total number of samples  $N \rightarrow \infty$ , all interior node posterior distributions converge with probability 1 to delta functions at the minimax optimal values.

*Proof.* Consider a parent node  $v$  with child nodes  $v_1, v_2, \dots, v_k$ . Assume that each child node  $v_i$  has a posterior distribution  $P_i$  that converges to a delta function at the correct minimax value  $\mu_i^*$ . That is,  $P_i \rightarrow \delta(p - \mu_i^*)$  as the number of samples  $n \rightarrow \infty$ .

For a MAX node, the value of the parent node  $v$  is determined by the maximum value among its children:

$$\mu_v = \max(\mu_1, \mu_2, \dots, \mu_k)$$

Since the child nodes' distributions converge to delta functions, the parent's distribution  $P_v$  will converge to the distribution of the child with the highest minimax value:

$$P_v \rightarrow \delta(p - \max(\mu_1^*, \mu_2^*, \dots, \mu_k^*))$$

Similarly, for a MIN node, the value of the parent node  $v$  is determined by the minimum value among its children:

$$\mu_v = \min(\mu_1, \mu_2, \dots, \mu_k)$$

The parent's distribution  $P_v$  will converge to the distribution of the child with the lowest minimax value:

$$P_v \rightarrow \delta(p - \min(\mu_1^*, \mu_2^*, \dots, \mu_k^*))$$

By induction, starting from the leaf nodes and moving up the tree, we can show that all interior nodes' distributions converge to delta functions at the correct minimax values. This completes the proof.  $\square$

**Theorem 6** (On-policy convergence) For the sampling policy specified in Bayes-UCT2: maximize  $B_i = \mu_i + \sqrt{2 \ln N} \sigma_i$ , all interior nodes converge with probability 1 to correct minimax optimal point values.

*Proof.* First of all, all points have non-vanishing variance.

Secondly, we show that if the children have a point all have non-vanishing variance, the parent will have a non-vanishing variance.

$$\mu_v = \max(\mu_1, \mu_2, \dots, \mu_k)$$

There is a  $j \in \{1, 2, \dots, k\}$  with  $\frac{1}{k}$  of the sampling,  $\mu_v = \mu_j$ . Note that  $\mu_j$  have a non-vanishing variance, thus the total variance is larger than this part, indicating a non-vanishing variance for  $\mu_v$ . Further more, all finitely sampled interior nodes have finite variance.

Now let's consider the situation that one node remains unsampled, while its parents and siblings receive infinite samples. Note that in this situation, the term  $B_i = \mu_i + \sqrt{2 \ln N} \sigma_i$  for this point will grow without bounds, while all other siblings have a bounded value (converge to the true value). Thus, all points will be sampled.

Also, note that in an infinite process, if a point is sampled for only finite times, with its siblings sampled for infinite times, the point will be sampled for infinite times. (Otherwise its  $B_i$  is unbounded).

Apply **Off-Policy Convergence Proof** here, we know that all points converge to their true mean.  $\square$

**Gaussian Approximation** Gaussian approximation may introduce errors due to two main factors. Firstly, leaf node distributions with limited trials and certain priors (e.g., uniform priors) may not be well-represented by Gaussians. Secondly, the Gaussian family is not closed under the max operation, potentially introducing further errors. Clarke [3] derived closed-form expressions for the mean  $\mu$  and variance  $\sigma^2$  of the maximum distribution of two Gaussian random variables. Let  $\mu_1, \mu_2$  be the means and  $\sigma_1, \sigma_2$  the standard deviations of two input Gaussians with correlation coefficient  $\rho$ . The moment generating function expansion provides the following expressions:

$$\mu = \mu_1 \Phi(\alpha) + \mu_2 \Phi(-\alpha) + \phi(\alpha) \sigma_m$$

$$\sigma^2 = (\mu_1^2 + \sigma_1^2) \Phi(\alpha) + (\mu_2^2 + \sigma_2^2) \Phi(-\alpha) + (\mu_1 + \mu_2) \sigma_m \phi(\alpha) - \mu^2$$

where  $\sigma_m = \sqrt{\sigma_1^2 - 2\rho\sigma_1\sigma_2 + \sigma_2^2}$ ,  $\alpha = \frac{\mu_1 - \mu_2}{\sigma_m}$ ,  $\phi(\cdot)$  denotes a standard Gaussian PDF with zero mean and unit variance, and  $\Phi(\cdot)$  denotes the CDF of  $\phi(\cdot)$ .

It could also be written as

$$\mu = \mu_2 + \sigma_m F_1(\alpha); \sigma^2 = \sigma_2^2 + (\sigma_1^2 - \sigma_2^2) \Phi(\alpha) + \sigma_m^2 F_2(\alpha)$$

Where  $F_1(\alpha) = \alpha \Phi(\alpha) + \phi(\alpha)$ ;  $F_2(\alpha) = \alpha^2 \Phi(\alpha)(1 - \Phi(\alpha)) + (1 - 2\Phi(\alpha))\alpha\phi(\alpha) - \phi(\alpha)^2$



### 5.3 First Play Urgency

FPU, proposed by [5], is an enhancement to MCTS that addresses unexplored nodes. It modifies the UCT algorithm to handle these nodes more effectively. Additionally, it incorporates strategies for efficient training and data storage, such as only retaining nodes that have been visited more than twice to avoid the initial infinite UCB1 value. The pseudocode for the FPU algorithm is as follows:

---

**Algorithm 7** FPU

---

```
1: function P(1)ayoneSequence(rootnode):
2:   node[0] := rootnode
3:   i := 0
4:   while node[i] is not first visited do
5:     createNode(node[i]);
6:     node[i].value := getValueByMC(node[i]);
7:     updateValue(node, -node[i].value);
8:     node[i+1] := descendByUCB1(node[i])
9:     i = i + 1
10:  end while
11: end function
```

---

The final algorithm FPU use is UCB1-tuned, which is also included in our final implementation.

## 6 Experiment

In this section, we evaluate the performance of various algorithms through a series of games. We train models over a specified number of epochs and then have different algorithms compete against each other.

### 6.1 Introduction

The key terms are defined as follows:

- **Selection:** The algorithm chosen to train the model.
- **Game:** The game in which the algorithm will compete.
- **Max-iter:** The maximum computational budget for the algorithm, indicating how many nodes can be explored in each training epoch.
- **Epochs:** The number of iterations the algorithm will train the model.

The code is available at <https://github.com/liuhanzuo/algorithm-design-survey>.

We introduce the experiment using the game **Tic-Tac-Toe**. Tic-Tac-Toe is a simple game where two players alternately place their marks on a 3x3 grid. The objective is to be the first to align three marks horizontally, vertically, or diagonally. The game results in a draw if the grid is filled without any player achieving this alignment. The repository is organized into several components:

- **node.py**: Defines the basic element, node, which stores the current state, parameters (such as the current board, value, win count, etc.).
- **tictactoe.py**: Implements the game class, providing functions for user moves and board updates.
- **agents.py**: Implements various search algorithms, including reward-based, visited-based, beta (random exploration with rate  $\beta$ ), UCT, and UCB1-tuned.
- **gaming.py**: Utilizes agents to play the game, training them over several epochs (1000 in practice) and then having them compete against each other (1000 games in practice) to test win, loss, and draw rates.
- **gui.py**: Provides a graphical user interface for user interaction with the game.
- **mcts.py**: Contains testing and plotting programs.
- **playground.py**: Serves as a playground for user-AI interaction games.

## 6.2 Results

Our result here is listed as **win rate,lose rate,draw rate**.

Test result is from training 1000 epochs and play 1000 epochs, random starter.

Algorithm	reward	visited	UCT	UCB1-tuned
reward	32%, 32%, 36%	66%,14%,20%	21%,36%,43%	20%, 35%, 45%
visited	X	45%,45%,10%	14%, 68%, 18%	14%, 66%, 20%
UCT	X	X	24%, 24%, 52%	22%,23%,55%
UCB1-tuned	X	X	X	22%,22%,56%

Table 1: itermax 40 Performance Comparison

Algorithm	reward	visited	UCT	UCB1-tuned
reward	32%, 32%, 36%	62%, 19%, 19%	15%, 37%, 48%	13%, 36%, 51%
visited	X	45%, 44%, 11%	11%, 71%, 18%	11%, 69%, 20%
UCT	X	X	20%, 20%, 60%	17%, 19%, 64%
UCB1-tuned	X	X	X	17%, 17%, 66%

Table 2: itermax 80 Performance Comparison

Algorithm	reward	visited	UCT	UCB1-tuned
reward	31%, 31%, 38%	62%, 19%, 19%	13%, 37%, 50%	11%, 37%, 52%
visited	X	45%, 45%, 10%	10%, 73%, 17%	9%, 70%, 21%
UCT	X	X	20%, 20%, 60%	16%, 20%, 64%
UCB1-tuned	X	X	X	15%, 15%, 70%

Table 3: itermax 120 Performance Comparison

Next, we show different strategies' performance comparison under different epochs: ( $X$  vs  $Y$  with itermax  $n$  means  $X$  start first, play against  $Y$  with search depth  $n$ , all results is trained 1000 epochs and test 1000 epochs)

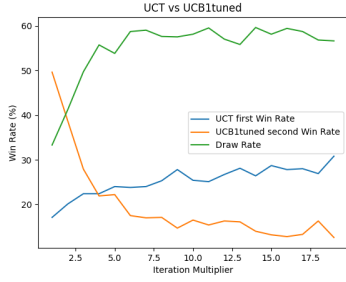


Figure 1: UCT vs UCB1-tuned with itermax 40

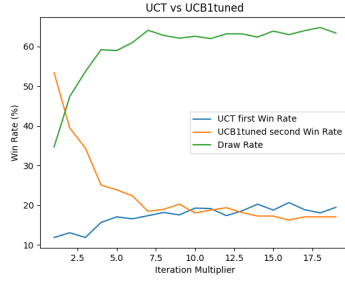


Figure 2: UCT values UCB1-tuned with itermax 80

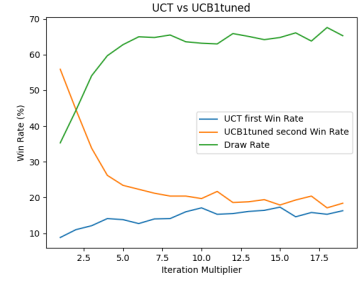


Figure 3: UCT vs UCB1-tuned with itermax 120

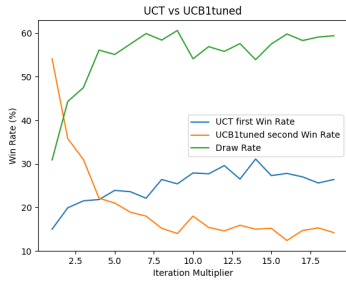


Figure 4: UCB1-tuned vs UCT with itermax 40

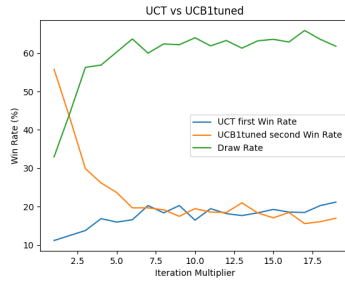


Figure 5: UCB1-tuned vs UCT with itermax 80

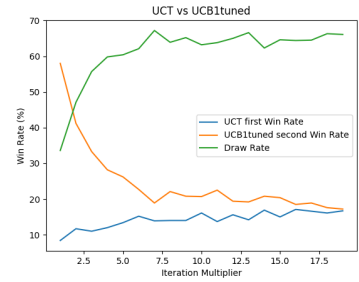


Figure 6: UCB1-tuned vs UCT with itermax 120

For more result, you can look at the repository, the `mcts/figs` directory contains all test result and we can not list all of them due to the limitation of space.

### 6.3 Conclusion

The results from both the table and graph indicate that UCT and UCB1-tuned algorithms achieve state-of-the-art performance, surpassing other search algorithms. As highlighted by the FPU, UCB1-tuned effectively handles unexplored nodes and optimizes memory usage. Additionally, given the simplicity of Tic-Tac-Toe and the high likelihood of a draw when agents possess strong predictive capabilities, the draw rate increases with deeper searches.

## 7 Gobang

### 7.1 Introduction

In this task, we integrate the UCT-tuned algorithm (a refined version of UCT) with the standard UCT algorithm to evaluate their performance in the game of Gobang. Given the larger board size of Gobang compared to Tic-Tac-Toe, this setup provides a more rigorous assessment of the search algorithms' capabilities.

### 7.2 Experimental New SOTA Algorithm: UCT-tuned

To better balance exploration weights across different nodes, we integrate the UCT and SP-MCTS algorithms. The primary adjustment in SP-MCTS involves the variance tail. For our two-player game, we eliminate the hyperparameter  $D$  in the summation part and instead multiply the variance tail by a hyperparameter  $D$ :

$$\arg \max_{v' \in \text{children}(v)} \left( \frac{Q(v')}{N(v')} + \sqrt{\frac{\log N(v)}{N(v')}} \cdot \min\left\{C, \frac{V(v')}{N(v')} + \sqrt{\frac{2 \log N(v)}{N(v')}}\right\} + D \cdot f(v) \right)$$
$$f(v) = \text{var}(N(u) \text{ where } u \text{ is a descendent of } v)$$

**Intuition:** Initially,  $\min\{C, \frac{V(v')}{N(v')} + \sqrt{\frac{2 \log N(v)}{N(v')}}\}$  balances low-explored states and higher win probabilities. As exploration progresses, the variance tail indicates node confidence; an optimal solution should exhibit higher variance, reflecting certainty in subsequent steps. In our experiments, we set  $C = 0.25$  and  $D = 2500$ , based on parameters from UCB1-tuned and SP-MCTS literature.

### 7.3 Main Code

- `Board.py`: Defines the board and provides functions for user moves and board updates.
- `Node.py`: Implements the basic MCTS node for tree updates.
- `MCTS.py`: Contains MCTS algorithms for node expansion and backpropagation.
- `Players.py`: Implements player agents for GUI interactions and AI vs AI evaluations.

### 7.4 Results

We run the experiment under a 12\*12 board with 5 chess pieces Gobang, and compare our algorithm UCT-tuned with current SOTA: UCT, the result is as follows:

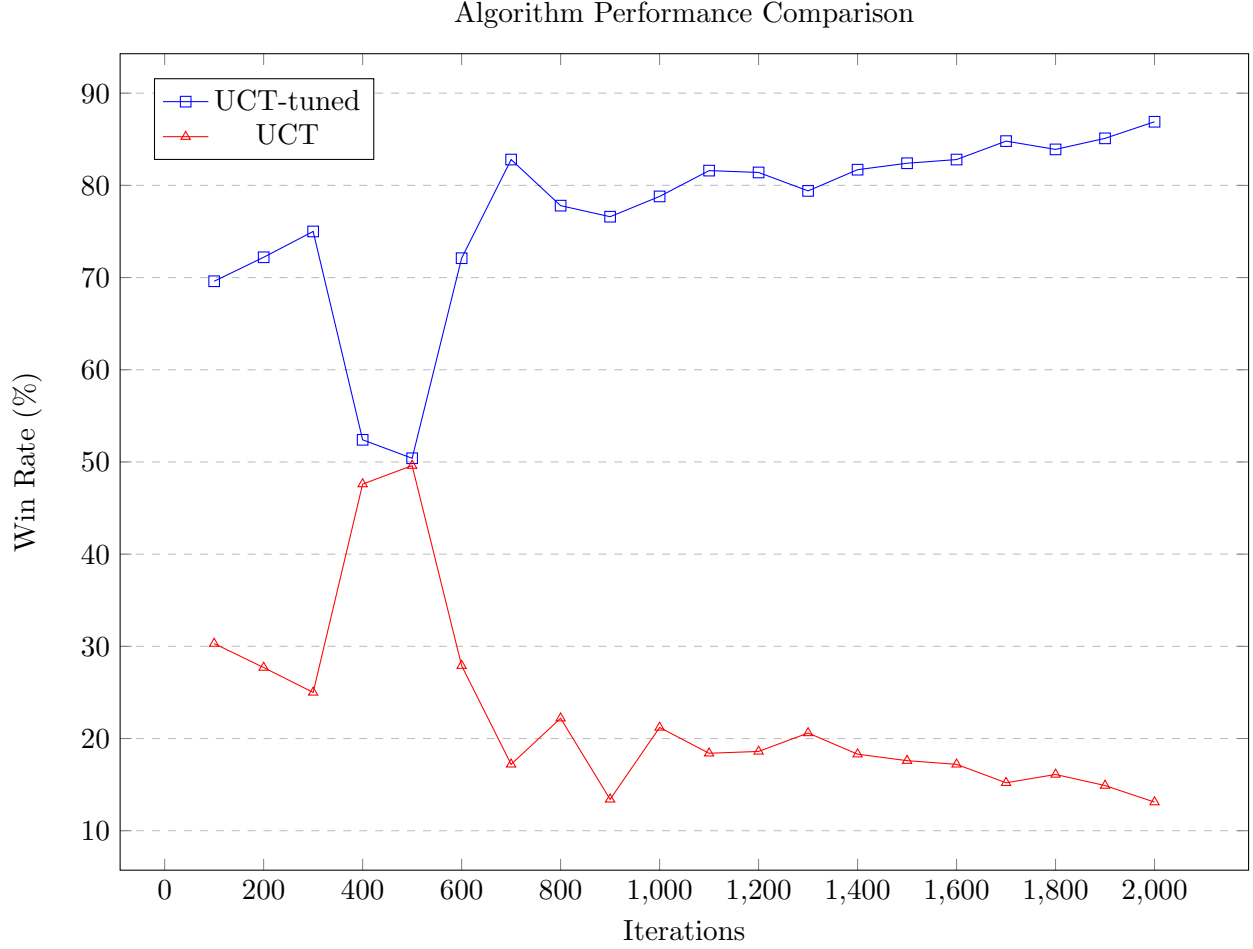


Figure 7: Win Rate Comparison of UCT and UCT-tuned Algorithms

The figure illustrates that the UCT-tuned algorithm exhibits a performance dip between 400-500 iterations compared to UCT. This fluctuation supports our hypothesis: the min term aids in early exploration, while the variance term enhances performance with sufficient iterations, ultimately leading to a higher win rate for UCT-tuned.

Additionally, the results demonstrate that UCT-tuned consistently outperforms UCT in terms of win rate as the number of iterations increases. This indicates that the modifications introduced in UCT-tuned, such as the variance tail and adjusted exploration-exploitation balance, provide significant advantages in more complex game scenarios like Gobang. The higher win rates achieved by UCT-tuned in later iterations suggest that it is more effective at identifying and exploiting optimal strategies over time.

Overall, the experimental results validate the effectiveness of the UCT-tuned algorithm, highlighting its potential as a superior alternative to traditional UCT in various applications. The integration of advanced techniques and careful parameter tuning in UCT-tuned contribute to its enhanced performance, making it a valuable tool for decision-making in complex environments.

## 8 Conclusion

In this survey, we have explored the intricacies of Monte Carlo Tree Search (MCTS), examining its foundational concepts, algorithmic implementations, and practical applications. We began by discussing the background of MCTS, including the essential elements of Markov Decision Processes (MDP) and Monte Carlo methods, providing a solid foundation for understanding MCTS’s operation and relevance in decision-making processes.

We introduced the general MCTS algorithm, detailing its four main steps: selection, expansion, simulation, and backpropagation. This was followed by an examination of specific MCTS variants, such as reward-based, visited-based, and hybrid MCTS, highlighting their unique approaches to balancing exploration and exploitation.

The survey also covered state-of-the-art algorithms like UCT (Upper Confidence bounds for Trees) and its variations, including UCB1-tuned and Bayesian UCT, analyzing their theoretical foundations, practical implementations, and performance improvements over basic MCTS.

Furthermore, we explored the integration of learning techniques within MCTS, such as Temporal Difference (TD) learning and its Monte Carlo variant (TDMC), which enhance the MCTS framework by incorporating value function approximations and reducing variance in value estimation.

Our experimental results, particularly in the context of the Tic-Tac-Toe game, demonstrated the effectiveness of advanced MCTS algorithms like UCT and UCB1-tuned, which consistently outperformed simpler variants, showcasing their robustness and adaptability in various scenarios.

In conclusion, MCTS remains a powerful and versatile tool in artificial intelligence and decision-making. Its ability to balance exploration and exploitation, coupled with advancements in algorithmic strategies, ensures its continued relevance and applicability in solving complex problems. Future research will likely yield even more sophisticated and efficient algorithms, further expanding the horizons of MCTS.

## References

- [1] P Auer. Finite-time analysis of the multiarmed bandit problem, 2002.
- [2] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.
- [3] Charles E Clark. The greatest of a finite set of random variables. *Operations Research*, 9(2):145–162, 1961.
- [4] Pierre-Arnaud Coquelin and Rémi Munos. Bandit algorithms for tree search. *arXiv preprint cs/0703062*, 2007.
- [5] Sylvain Gelly and Yizao Wang. Exploration exploitation in go: Uct for monte-carlo go. In *NIPS: Neural Information Processing Systems Conference On-line trading of Exploration and Exploitation Workshop*, 2006.
- [6] Yasuhiro Osaki, Kazutomo Shibahara, Yasuhiro Tajima, and Yoshiyuki Kotani. An othello evaluation function based on temporal difference learning using probability of winning. In *2008 IEEE Symposium On Computational Intelligence and Games*, pages 205–211, 2008.

- [7] Maarten PD Schadd, Mark HM Winands, H Jaap Van Den Herik, Guillaume MJ B Chaslot, and Jos WHM Uiterwijk. Single-player monte-carlo tree search. In *Computers and Games: 6th International Conference, CG 2008, Beijing, China, September 29-October 1, 2008. Proceedings 6*, pages 1–12. Springer, 2008.
- [8] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *nature*, 550(7676):354–359, 2017.
- [9] Gerald Tesauro. Practical issues in temporal difference learning. *Advances in neural information processing systems*, 4, 1991.
- [10] Gerald Tesauro et al. Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3):58–68, 1995.
- [11] Gerald Tesauro, VT Rajan, and Richard Segal. Bayesian inference in monte-carlo tree search. *arXiv preprint arXiv:1203.3519*, 2012.
- [12] David Tolpin and Solomon Shimony. Mcts based on simple regret. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 26, pages 570–576, 2012.