Bandit Algorithms in Game Tree Search: Application to Computer Renju* 

Ye Fan  
Department of Computer Science  
University of British Columbia  
Vancouver, BC V6T 1Z4  
yefan@cs.ubc.ca

Abstract  
Multi-armed bandit problem is to maximize a cumulated reward by playing arms sequentially without prior knowledge. Algorithms for this problem such as UCT have been successfully extended to computer GO programs and proved significantly effective by defeating professional players. The goal of the project is to implement a Renju AI based on Monte Carlo planning that is able to defeat the oldest known alpha-beta pruning AI. In this project we will give an implementation of the UCT algorithm and attempt to improve the minimax tree search by modeling the dependencies between arms. Our work is greatly inspired by the theoretical advances in GPTS (Gaussian Process for Tree Search) where the mean reward function is drawn from a Gaussian Process. Applying GPTS to game tree search is by no means trivial and it would be interesting to see how it works in practice.

1 Introduction  
Recently, game tree search has been posed as a multi-armed bandit problem. Each node in the tree is considered as a bandit with unknown reward distribution and the goal is to minimize the regret at the end. In practice bandit algorithms are found extremely effective for huge trees due to their efficient trade-off between exploration and exploitation. One example is the first UCT (Upper Confidence Boundary applied to Trees) based computer GO program "MoGo" [4] that won a 19x19 game against a high-level professional. Later more excellent GO programs with improvements showed up, (i.e.GNU GO, Fuego). They are all based on UCT and exhibit high strength in various tournaments.

The idea behind UCT is Monte-Carlo planning. A basic Monte-Carlo program simply collects statistics from all the children of the root and chooses the best arm, but it doesn’t guarantee the convergence to the best one. UCT is a natural extension in the sense that it does a deeper search and promising arms are played more often. Given infinite time, UCT will eventually converge to the best one. But for the game tree search problem, guaranteeing convergence is not good enough, the algorithm has to give the next move in a required time. To optimize the next move, we want the search paths to be as precise as possible. Exploring the dependencies across arms is a promising direction because the information gain could be maximized by updating all related nodes once a reward is evaluated at a leaf node.

*For the rule of Renju game, please visit http://en.wikipedia.org/wiki/Renju
Related research for modeling dependent arms include GPTS[3], clustered arms[8], FLAT-UCB[1], etc. among which GPTS[3] is mostly related to game tree search. GP is to formalize our assumption on the smoothness of tree, assuming the reward function is drawn from GP with noise. Theoretical work by [3][2] has shown that GPTS has a lower order regret than UCT and performs better in a toy problem despite its high computational cost. It would be interesting to apply GPTS to a real game tree search problem to see how it actually performs in practice.

In this project, we first give an implementation of the UCT algorithm for the game Renju, then we tried two approaches to apply GPTS to the game tree search problem. Our expectation is to defeat the αβ AI implemented years ago by our new algorithms. For a review of the αβ pruning, see [5]. Finally, we evaluate the actual effectiveness of these algorithms by letting them play against each other.

The reason for choosing Renju is two-fold. One is that it’s much easier than GO, an αβ pruning based algorithm is simple to implement and performs well. The other reason is that we have been very familiar with this game and don’t want spend time learning a more sophisticated game.

2 UCT Algorithm

The UCT algorithm is a variant of the UCB(Upper Confidence Boundary) algorithm for multi-armed bandit problems in which arms are considered to be independent. It’s an iterative deepening style search. Denote $\mu_x(t)$ as the empirical mean for the arm $x$ and use $\sigma_x(t)$ to describe the uncertainty. The algorithm is to determine the confidence interval $[\mu_x(t) - \sigma_x(t), \mu_x(t) + \sigma_x(t)]$ for the reward value we would get for each arm $x$. At each iteration, the algorithm chooses the arm with the highest upper-boundary $\mu_x(t) + \sigma_x(t)$. The bigger $\sigma_x(t)$ gets, the more likely it is that the reward will be in the confidence interval. The bound of UCB can be proved to be in $O(\log t)$ if $\sigma_x(t)$ is set to $\sigma_x(t) = \sqrt{\frac{2\log \frac{1}{\delta}}{p(x)}}[3]$, where $p(x)$ is the number that arm $x$ has been played so far.

UCT is the extension of UCB1 to tree search proposed by[6]. In UCT the arm selection problem is considered as a separate multi-armed bandit for every explored internal node. Theoretical results show that this algorithm is consistent and has a significant performance advantage over its closest competitors[7]. The formula for UCT value reads:

$$\text{UCT}(x) = \text{win}(x) + \sqrt{\frac{2\log \frac{1}{\delta(p(x'))}}{p(x)}}$$

The UCT algorithm will converge eventually given enough time and memory. In the above formula, the term $\sqrt{\frac{2\log \frac{1}{\delta}}{p(x')}}$ grows quickly at the beginning, then it goes to flat. Since the value is unbounded, it will finally goes up to infinity though very slowly. If given infinite time, it will find any winning move no matter how badly the initial win rates are.

The pseudocode for a general UCT algorithm is given by Algorithm 1.

Rewards are given by Monte Carlo simulations where the game is randomly finished from the current position and return 1 for a win, 0 otherwise.

3 GPTS Algorithm

The bandit problem would be trivial if the search space is smaller than the maximum number of iterations we could perform. In global optimization, the search space is usually continuous and the main idea of GPTS is to model the dependencies between arms through smoothness assumption on $f$. The dependent arms assumption helps to make the exploration phases shorter because once an arm is played, knowledge is gained not only on itself but also on similar ones. Their mean reward function could also be updated instead of just the one being played.

In the tree search problem, each possible path is considered as an arm and the whole tree is a bandit problem. The GP assumption across paths makes sense since the more nodes in common, the more likely the two paths would have similar results. There correlations are modeled by assuming that $f$ is a function drawn from a Gaussian Process(GP).
Algorithm 1 UCT algorithm

while Stop criterion is not met do
    Set root to Node \( D \)
    while \( D \) has children do
        Choose from \( D \)'s children node \( C_j \) which maximizes the upper-bound \( B_j \)
    end while
    if node \( D \) has been played enough times then
        Generate children for \( D \)
    else
        Get the reward value \( y \) by randomly finishing the current game;
    end if
    while \( D \) has parents \( P_j \) do
        update visit number \( n_j \)
        update upper-bound \( B_j \)
    end while
end while

3.1 GP assumption

GP is a functional distribution that characterizes a belief on the smoothness of functions. Roughly speaking, the idea is that similar inputs are likely to yield similar outputs and the similarity is characterized by a mean function and a covariance function. By default, we take the mean of the GP prior to be 0 and incorporate our knowledge on how correlated arms are in the GP covariance matrix characterized by a covariance function \( \kappa(x_i, x_j) \). To choose an appropriate kernel for the tree paths, there are several choices.

- Linear Kernel: The linear kernel simply counts the number of nodes in common between two paths. If two paths have \( d \) nodes in common, we have \( \kappa_d(x_i, x_j) = d \). If the total depth is \( D \), the normalized version is \( \kappa_d(x_i, x_j) = d/D \)
- Gaussian Kernel: The Gaussian kernel is an exponential on the number of nodes that differ. It reads: \( \kappa_d(x_i, x_j) = \exp(-D-d^2/l^2) \), where \( l^2 \) is an adjustable character length.

For the game Renju, we could also represent paths by game boards. Strictly speaking, it’s modeling the dependencies between nodes, not tree paths. To measure how similar two boards are, domain knowledge could be helpful.

For a space of arms \( X \), we use \( f(t) + \epsilon \) to represent the reward with noise. \( \epsilon \sim N(0, \sigma^2_{noise}) \). The covariance matrix \( C_{ij} \) and the vector \( k(x)_i \) are given as

\[
C_{ij} = \kappa(x_i, x_j) + \sigma^2_{noise} \delta_{ij} \\
k(x)_i = \kappa(x, x_i)
\]

Using \( y_i \) to denote the corresponding reward after playing arm \( x_i \), \( \mu(x) \) and \( \sigma^2 \) can be written as [9]

\[
\mu(x) = k(x)^T C^{-1} y \\
\sigma^2(x) = \kappa(x, x) - k(x)^T C^{-1} k(x)
\]

The algorithm selects the next arm in a similar fashion as that of UCT. It aims at optimally balancing exploration and exploitation by maximizing an upper confidence function

\[
x_{t+1} = \text{argmax}_x f_t(x) = \mu_t(x) + \sqrt{\beta_t \sigma_t(x)}
\]

The general procedure described in [3] is shown in Algorithm 2

3.2 Complexity Analysis

Let \( N \) be the size of the arm set and \( T \) be the size of the training set. So the covariance matrix and its inverse are both of dimension \( T \times T \). Computing \( f \) involves matrix multiplication with a cost of
Algorithm 2 UCT algorithm

Create root node $n_0$ and dummy node $d_0$
Add $d_0$ to the set of arms $S$

while Stop criterion is not met do
  Choose $x \in S$ with the highest upper-bound
  if $x$ is dummy node then
    Create sibling $x'$ of $x$
    while depth of $x$ is smaller than $D$ do
      Create a child $x'$ and a dummy child $d$ of $x$
      Add $d$ to $S$
    end while
  end if
  Add leaf $x$ to the training set $T$
  Update kernel matrix and its inverse
  for all node $n \in S$ do
    Update the vector of kernel products and upper-bounds
  end for
end while

$O(T^2)$. The cost for computing all the $f$ values will be $O(T^2 N)$ In addition, we need to take matrix inversion into account. A naive implementation results in a cost of $O(T^3)$. Note that the covariance matrix is increasing in an iterative manner, so that using Schur Complement could reduce the time complexity to $O(T^2)$. In total, the cost is $O(T^2 N) + O(T^2) = O(T^2 N)$. Since each leaf in the training set has at most $D$ corresponding dummy nodes along the path, $D$ is the depth of the leaf, the cost can also be written as $O(N^3)$. In comparison, the basic UCB algorithm is much more efficient and has a cost of only $O(N)$ because of its constant cost in each iteration. To be as competitive as UCT, GPTS must select the most promising arm as accurate as possible.

One optimization for GPTS to reduce the computational cost is by reducing the size of the training set. A simple FIFO (First In First Out) strategy would be maintaining a training set of size $T$. We keep adding new arms to the training set until it’s full. Then we remove the oldest arm in the set once a new arm is coming in. One tempting choice for the size $T$ is $T = \sqrt{N}$ so that the amount of information that GPTS uses will be $O(\sqrt{N}^2 = O(N))$, the same as that of UCT. Then the cost of each iteration is reduced to $O(N^2)$, but with additional cost for updating the training set.

As for the performance, it’s getting worse as expected, but still superior to UCB. If the training set is further reduced to $\sqrt{N}$, it tends to perform a random selection policy, which is shown by [2]

3.3 Apply GPTS to game tree

It seems straightforward to implement GPTS and apply it to a game tree search, but we encountered two major problems during implementations.

3.3.1 Smoothness Assumption

First, the assumption on the smoothness of $f$ is optimistic. The slight difference between a game tree and an ordinary tree is that a game tree is actually a two-player game, the objective of the optimization problem differs at different depths. For example, player A wishes to maximize his reward at tree depth $k$, while at depth $k + 1$, player B wants to maximize his own and his maximum win rate is the minimum for player A. If we still insist on the smoothness assumption on paths of different depths, the search result for player A would be the situation that A always chooses the best move but B always chooses his worst.

To make GPTS work on a game tree, we have to modify the assumption that the reward function is smooth only for the nodes of the same depth, or to be more general, for the nodes that have the same color.
• Approach 1: Nodes of different colors are modeled separately. This is based on the assumption that the reward function for the same color is smooth. The game tree remains the same but there are two training sets now, one for the black nodes and the other for the white nodes. When the algorithm needs to pick the next path to explore, it chooses the path with the highest upper-bound from white nodes and black nodes in an alternating manner.

• Approach 2: Instead of using the win rates as the reward, we explore a different reward representation by using the difference of win rates between the current move and its parent’s move. If the difference is negative, it means the chosen move will lower the win rate, thus it’s a bad move. If it’s positive, then this move will possibly lead to the final winning and should be considered as a good move. By maximizing the upper-bound of this value, we hope the computer will play the good moves more often and the algorithm could converge to the winning state eventually.

3.3.2 Computational Overhead

Game trees are notorious for their huge sizes and Algorithm 2 is being too optimistic. For the game Renju, there are around 400 possible moves each round. Given a search depth \( D \), there will be \( 400^D \) possible paths and the average depth \( \bar{D} \) is about 30, picking a path that leads to the real leaf and evaluating the reward at that leaf is infeasible. Most of the game tree search algorithms are using iterative tree growing methods. Still, a straightforward implementation of GPTS needs to prevent the training set from being too large. Recall that the time complexity of GPTS is \( O(t^2N) \), which means 100 iterations only will make the resulting search procedure 10000 times slower! If we don’t want the algorithm to run forever, a reasonable value for \( D \) must be very small compared to the total depth of the tree.

4 Experiments

The experiment is designed to let two AIs play against each other. Since both UCT and GPTS are by Monte Carlo planning, the result is non-deterministic. The AIs will play for ten games to estimate the winning rate. For the sake of fairness, the number of first hands for each AI is the same. The rule of Renju is simplified in this implementation. It’s known that the first hand has a larger chance to win the game. If the first hand keeps playing the optimal move, there exists a sequence of moves that leads to the winning. The game board is \( 15 \times 15 \). The entries in the table 1 represent the winning times for the UCT algorithm.

4.1 Experiments with UCT

<table>
<thead>
<tr>
<th>Simulations</th>
<th>Depth</th>
<th>UCT first</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>( \alpha\beta ) first</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 500, 000</td>
<td>1</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>1, 300, 000</td>
<td></td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1, 000, 000</td>
<td></td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

UCT is doing very well against \( \alpha\beta \) pruning based AI. If the search depth of \( \alpha\beta \) AI is less than five, the UCT algorithm wins almost all the games no matter first hand or not, which is more stable than expected. Based on the experiment results, we estimate that the current UCT based Renju AI is as competitive as the \( \alpha\beta \) AI with search depth five. UCT took much more time than \( \alpha\beta \) based AI at each step. It’s partially because UCT doesn’t require any prior knowledge of the game. If the rule is changed to another game, say GO, it could run without any other modifications. On the other hand, the evaluation procedure in the \( \alpha\beta \) AI utilizes much of the domain knowledge. Since Renju is a relatively easy game compared to GO, writing such an efficient evaluation algorithm with high accuracy is feasible.

But during the experiments, we also found some potential problems. Most of the time, UCT would choose the best move, but sometimes suboptimal arms are chosen especially near the end of the
Figure 1: Suboptimal Move, red circles are optimal ones leading to quick winning while the blue circle is a more defensive move.

For example, in Figure 4.1, white stones represent UCT. Red circles are the optimal moves that could win the game quickly. But the UCT algorithm chose the position represented by the blue circle which is actually a defensive move. Although the while color win the game eventually, it took many more moves. Also, we found that the fewer simulations the more likely the UCT would choose a suboptimal move.

4.2 Experiments with GPTS

Table 2: GPTS vs. $\alpha\beta$ (10 Games)

<table>
<thead>
<tr>
<th>Paths</th>
<th>GPTS first</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>$\alpha\beta$ first</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Approach 1)</td>
<td>500</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>400</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>300</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(Approach 2)</td>
<td>500</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>400</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>300</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Similar experiments are done for the GPTS algorithm. The results are shown in table 2. Unfortunately, it did poorly and wasn’t able to find a good move in a reasonable time. The approaches proposed in section 3.3.1 are implemented and tested but were not as effective as expected. By further scrutiny into the algorithms, the reasons for the poor performance are summarized as follows:

- **Approach 1:** The algorithm works correctly but still fails to converge to the best move. Roughly speaking, it’s because the GP procedure tends to choose paths where the opponent takes bad moves, so that the random simulations couldn’t give accurate rewards and the algorithm may not even converge eventually. Actually, choosing a path with the highest upper-confidence value globally doesn’t make much sense for a minimax tree. One simple example is that player A always takes the optimal move while its opponent always selects its worst move. The resulting path will have a very high mean-reward and GPTS would very readily choose such a path. Therefore approach 1 only solves the problem superficially and fails to search the game tree properly.

- **Approach 2:** By observation, we believe that approach 2 performs slightly better than approach 1. Locally, choosing a move that maximize the win rates change is the optimal move, but it suffers from the same problem as approach 1. If the node’s parent makes a bad move and the node makes a perfect move, the win rates change is large enough to make the current move super excellent. However, the path itself may not deserve further exploitation. So this approach solves the problem only locally and doesn’t perform well as a whole.
4.3 Experiments on Computational Cost

As can be seen from figure 4.3 and 4.3 The empirical running time matches our expectation. The relationship between time and number of simulations is linear for UCT. During the same time UCT finishes 1,500,000 simulations, GPTS could examine only hundreds of paths, which reveals that its time complexity is cubic.

5 Conclusion

In this project, we implemented the UCT algorithm based on [6] and proved successful by playing against the old-fashioned $\alpha\beta$ pruning AI algorithm. To our estimation, the UCT AI that runs 1,500,000 simulations per move is as competitive as an $\alpha\beta$ AI with a search depth of five. Due to time limit, we didn’t optimize the parameters in the UCT algorithm. Its true strength should be more powerful than that reported in this report. UCT is an amazing algorithm. It’s so simple and only requires knowing the rule of the game, but it performs like a well-trained player.

However, much of the work is devoted to improving the game tree search by GPTS, unfortunately all the attempts didn’t end up with nice results. The reasons have been given at section 4.2. Our conclusion is that a straightforward implementation of GPTS doesn’t work properly for a game tree. Always searching for a path with the highest upper-confidence value globally may not even converge to the best arm. It needs more effort to make it work correctly. Our experiments also showed that the GPTS algorithm itself could cause computational overhead. Although our implementation could be further optimized much more aggressively, the $O(T^2N)$ time complexity doesn’t allow too many training samples. On the other hand, randomly finishing a game is extremely fast for a computer. It’s still unknown whether GPTS could perform better than the UCT algorithm for searching a real game tree.
References