Immediate Versus Delayed Reward for the Game of Go

Chia-Man Hung
Master MVA
chia-man.hung@polytechnique.edu

Dexiong Chen
Master MVA
dexiong.chen@polytechnique.edu

Abstract

The goal of this project is to study the main principles behind the problem of reinforcement learning for the game of Go, and focus on the benefit and drawbacks of using an informative reward given at each time step versus a reward only given at the very end of the game. The experimental study focuses on the Monte-Carlo Tree Search algorithm and some variants of it as well as on min-max trees.

1 Introduction

Monte-Carlo methods have been extensively studied in the last 3 decades and have been successfully applied to the game of Go since 1993 when Brügmann [4] introduced the method. Since the discovery of the upper confidence tree (UCT) in 2006 by Kocsis and Szepesvári [6], Monte-Carlo tree search (MCTS) became the state-of-the-art algorithm and attracted a number of researchers' attention. In the last decade, a number of variants and related work have been proposed and studied. Among them, Browne et al. [3] have made a breakthrough in the theoretical aspect, which guarantees the theoretical performance of MCTS.

Before the success of alphaGo in 2016, Go was one of the few classic games for which professional human players were so far ahead of computer players. One of the main issues is the computational complexity, as the number of possible actions increases exponentially with the time steps. As a consequence, the basic MCTS process requires simulating all feasible actions until the end of the game in order to back-propagate the final result to the current state node. In this situation, the reward of one action is called *delayed reward* as one only gets reward at the end. A possible approach to reduce the set of feasible actions during simulation is to define an *immediate reward* after each action, intuitively according to the current state and the state after an action. This reward should be capable to keep the most potential actions while ignoring the less probable actions.

In this project, we will review the main principles of MCTS and apply it to the game of Go. We will especially focus on the benefit and drawbacks of using an immediate reward at each time step compared to a delayed reward given at the very end of a game. The comparisons will be carried out by considering the number of wins as well as the computational complexity.

The report is organized as follows. In Section 2, we review and study MCTS algorithms, including the UCT algorithm. Then in Section 3, we introduce the notion of immediate reward and explain how it can be used to prune the search tree. In Section 4, the implementation details are presented, including the environment that we chose, code structure as well as some practical remarks. In Section 5, we illustrate and analyze the experimental results.

2 Monte-Carlo Tree Search

The key idea of MCTS is to construct a search tree step by step by repeatedly choosing the most promising child of the already researched tree, expanding it, and then evaluating the new leaf node by simulating the game until the end. The result of the simulation, a win or loss value is then back-propagated to all the parent nodes up to the root, and a new promising node is chosen.

2.1 General Algorithm

In more details, four steps are applied per search iteration [3]:

- *Selection*: Starting at the root node, a child selection policy is recursively applied to descend through the tree until a node with unvisited children is reached. Note that at the first iterations of MCTS, we select the root node directly since it has unvisited children.
- *Expansion*: One child is added to expand the tree.
- *Simulation*: From the chosen node, a default policy is applied to simulate the game.
- *Back-propagation*: The result of the game is back-propagated to all its parent nodes to update the statistics.

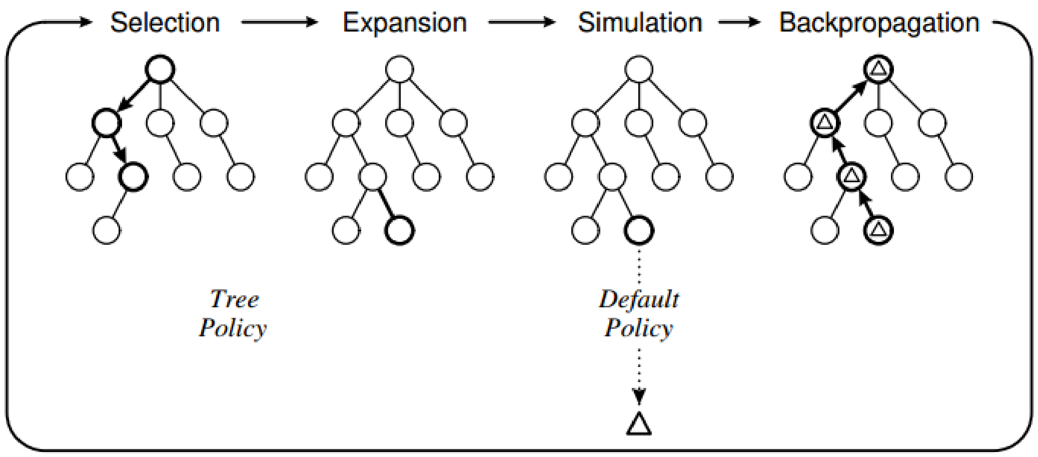

Figure 1: One iteration of the general MCTS approach. Courtesy of [3].

Algorithm 1: General MCTS approach.

```
function MCTSSearch ($s_0$)
create root node $v_0$ with state $s_0$
for i = 1, ..., itermax do
    $v_l \leftarrow$ TreePolicy($v_0$)
    $\Delta \leftarrow$ DefaultPolicy(s($v_l$))
    BackPropagate($v_l$, $\Delta$)
end
```

2.2 UCT Algorithm

During the selection step, a tree policy is required to explore the tree to decide on promising children. For this reason, the widely used Upper Confidence Bound applied for Trees (UCT) [6] was derived from the UCB1 [1] policy. In treating the choice of child node as a multi-armed bandit problem, the value of the child node is the expected reward approximated by the Monte-Carlo simulations. UCT balances the exploitation of rewarding nodes whilst allowing exploration of less visited nodes. Consider a node v, the tree policy determining which promising child node to choose is defined as:

$$v^* = \underset{v_c \in \text{child}(v)}{\arg\max} \frac{W(v_c)}{N(v_c)} + K\sqrt{\frac{lnN(v)}{N(v_c)}} \tag{1}$$

where v_c is a child of v, W is the wins count, N is the visits count, and K is a exploration constant to tune.

A pseudo-code from [3] is illustrated below.

Algorithm 2: The UCT algorithm.

```
function UCTSearch ($s_0$)
create root node $v_0$ with state $s_0$
for i = 1, ..., itermax do
    $v_l \leftarrow$ TreePolicy($v_0$)
    $\Delta \leftarrow$ DefaultPolicy(s($v_l$))
    BackPropagate($v_l$, $\Delta$)
end
return a(BestChild($v_0$, 0))

function TreePolicy($v$)
while $v$ is non-terminal do
    if $v$ not fully expanded then
        return Expand($v$)
    end
    else
        $v \leftarrow$ BestChild($v$, $C_p$)
    end
end
return $v$

function Expand($v$)
choose $a \in$ untried actions from $A(s(v))$
add a new child $v'$ to $v$ with $s(v') = f(s(v), a)$ and $a(v') = a$
return $v'$

function BestChild($v$, $c$)
return $\arg\max_{v' \in \text{child}(v)} \frac{W(v')}{N(v')} + c\sqrt{\frac{lnN(v)}{N(v')}}$

function DefaultPolicy($s$)
while $s$ is non-terminal do
    choose $a \in A(s)$ uniformly at random
    $s \leftarrow f(s, a)$
end
return reward for state $s$

function BackPropagate($v$, $\Delta$)
while $v$ is not null do
    $N(v) \leftarrow N(v) + 1$
    $W(v) \leftarrow W(v) + \Delta$
    $v \leftarrow$ parent of $v$
end
```

It is shown in [6] that the probability of choosing a sub-optimal action at the root of the tree converges to zero at a polynomial rate as the number of iterations grows to infinity. This proves that given enough time and memory, UCT converges to the optimal action. The working of UCT is also demonstrated empirically on a variety of domains.

2.3 Variants

In the general MCTS approach, there is freedom in choosing the tree policy and the default policy. Enhancements proposed to the tree policy are generally divided into five categories: bandit-based enhancements, selection enhancements, all-moves-as-first enhancements (update statistics for all actions selected as if they were the first action applied), game-theoretic enhancements (back up the theoretic value of a state to improve reward estimates for non-terminal nodes), and move pruning. Most of the time, the default policy is the basic random policy, as it is simple, does not require domain knowledge and cover different area of the search space.

In the context of Go, Rapid Action Value Estimation (RAVE), an all-moves-as-first enhancement, is commonly used.

3 Immediate Reward

Without enough time budget, we can use domain-specific knowledge to speed up the convergence. More specially, in the selection step, we prune nodes that do not seem promising and focus on nodes that will probably result in optimal actions. In this section, we first define the immediate reward used throughout the project. Then, we describe how it is used to speed up the convergence.

3.1 Problem Setting

The ultimate goal in the game of Go is to control a territory by putting stones on the board. Based on this understanding, we design a simple reward function as follows.

Influence function. Let $p, g \in G$ be the positions of two stones, where G is the set of all positions on the board. We define the influence function of a white stone (respectively black) at position p over q by

$$I_4^W(p,q) = (4 - d_4(p,q))_+, I_4^B(p,q) = -(4 - d_4(p,q))_+, \tag{2}$$

where d_4 denotes the distance of between two positions on the board.
The total influence of the stones on position q at step t is given by

$$\mathcal{I}_t(q) = \sum_{p \in W_t} I_4^W(p,q) + \sum_{p \in B_t} I_4^B(p,q), \tag{3}$$

where W_t is the set of white stones present on the board at time t, and B_t is the set of black stones present on the board at time t.

Boundary. In practice, the board of Go is finite, which constrains the game and creates side effect. We consider two ways to handle the boundary as follows.

- **Empty.** We take into account the lack of freedom induced by the boundary and modify the influence function on the boundary and corners. For instance, $I_4^W(p,q) = (3 - d_4(p,q))_+$ if p is on the boundary, and $I_4^W(p,q) = (2 - d_4(p,q))_+$ if p is one of the 4 corners.
- **Adversarial.** We may simply consider that the boundary of the board is surrounded by adversarial stones. We add a virtual boundary covered with black stones when it is white's turn, and with white stones when it is black's turn. In that case, we redefine W_t, B_t accordingly to include these virtual stones. To avoid confusion, we write I_t^W when the boundary is black (white's turn) and I_t^B when the boundary is white (black's turn).

Reward function. The reward for white increases by 1 when a neutral territory becomes white, and by 2 when a territory controlled by the black turns white. We define the following functions for the τ^{th} play of player white (respectively black):

$$\begin{aligned} r_\tau^W(p) &= \sum_{q \in G} (\mathcal{I}_{2\tau}^W(q) - \mathcal{I}_{2\tau-1}^W(q))_+ \mathbb{1}\{\mathcal{I}_{2\tau-1}^W(q) < 0 \leq \mathcal{I}_{2\tau}^W(q)\} \\ r_\tau^B(p) &= \sum_{q \in G} (-\mathcal{I}_{2\tau+1}^B(q) + \mathcal{I}_{2\tau}^B(q))_+ \mathbb{1}\{\mathcal{I}_{2\tau}^B(q) > 0 \geq \mathcal{I}_{2\tau+1}^B(q)\} \end{aligned} \tag{4}$$

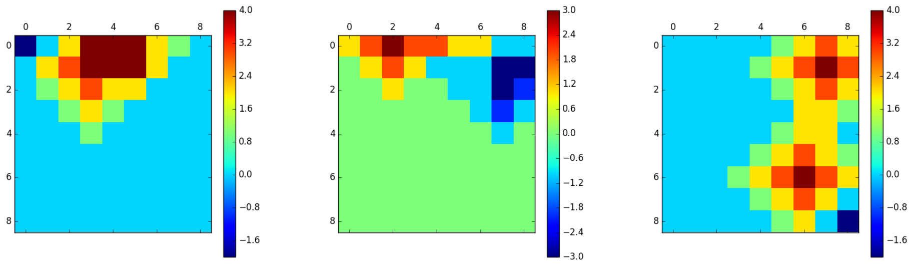
Figure 2: Illustration of the white's value of the influence function with empty boundary. Left: white (1, 3), (0, 5), black (0, 0). Middle: white (0, 2), (0, 6), black (1, 7). Right: white (6, 6), (1, 7), black (8, 8).

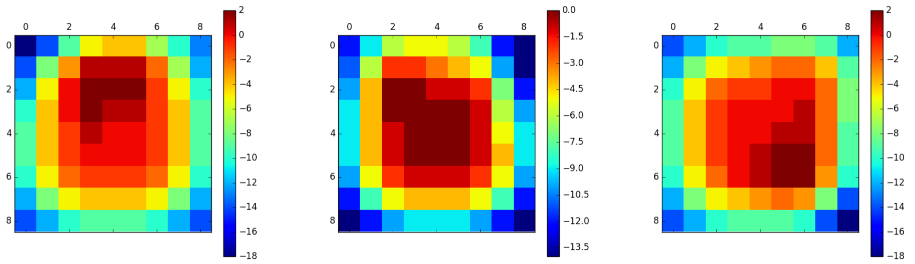
Figure 3: Illustration of the white's value of the influence function with adversarial boundary. Left: white (1, 3), (0, 5), black (0, 0). Middle: white (0, 2), (0, 6), black (1, 7). Right: white (6, 6), (1, 7), black (8, 8).

It then remains to count the number of stones captured by playing a position p. Let c_τ^W denote the number of black stones captured by white when playing for the τ^{th} time. The final reward functions for the τ^{th} play of player white (respectively black) are then defined by

$$\begin{aligned} r_{W,\tau}(p) &= r_\tau^W(p) + c_\tau^W \\ r_{B,\tau}(p) &= r_\tau^B(p) + c_\tau^B \,. \end{aligned} \tag{5}$$

3.2 Pruning

Using this reward, we may prune the search tree by considering the ϵ-optimal actions, i.e. the actions of the the immediate reward is at most ϵ-away from the action with the best immediate reward. $\epsilon = 0$ corresponds to the case where we consider only the actions with the best immediate reward and $\epsilon = 1$ corresponds to keeping all the actions and not pruning at all. The former one leads to a high branching factor, thus speeds up the convergence if the pruning is done correctly (without dropping the optimal action), whereas the latter one keeps necessarily the optimal action since it does not take any risk of dropping actions. This parameter ϵ is non-trivial to tune.

3.3 Min-Max Principle

Go is an adversarial game. In this kind of game, if our goal is to maximize the accumulated reward, choosing the action with the best immediate reward at each step might not always result in winning. This is due to the fact that player A's move affects player B's feasible actions and thus their reward. An action with a poor immediate reward may be considerable if it prevents the opponent from getting a high immediate reward. As a consequence, expanding the set of ϵ-optimal moves to a larger set that includes also the ϵ-optimal set for the opponent may lead to better performance. And one may even extend this exploration up to k further steps.

In a formal way, we can consider the following min-max value which defines the set described above

$$a^* = \max_{a \in A(s)} \min_{b \in A(s(a))} r(a, s) - r(b, s(a)) \tag{6}$$

where s is the initial state, a an action taken by player A, b an action taken by player B, $s(a)$ the state after playing a, and A the set of all feasible actions given a state.

Here, a^* represents the best action of the first player taken two steps into account. We can extend this reasoning to more steps and the computation is done recursively. When considering the best action to take up to k steps, it suffices to replace $r(b, s(a))$ by the best accumulated reward of playing b up to $k - 1$ steps.

In the pruning process, instead of using the immediate reward directly, we may consider the accumulated immediate reward taken two or more steps into account, assuming that both players are smart enough to perform the action that maximizes his accumulated immediate reward.

3.4 Back-Propagated Value

In the general approach, we back-propagate the reward at the end of the game according to the official game result, which is 1 in case of win, 0 for a draw, and -1 else. This could be replaced by the accumulated immediate reward. One benefit is that the accumulated immediate reward gives us a rough idea of how the game goes before reaching the end. One drawback is that we do not have the guarantee that it agrees with the official game result.

4 Implementation

4.1 Environment

We use the OpenAI gym environment to simulate the game of Go. Its python API is wrapped over a Go engine written in C named Pachi [2]. It provides us the game simulator, which helps us play actions, observe the game state, and determine the final result.

4.2 Code Structure

Our code is adapted from the basic MCTS in [5]. The core classes are *game_state.py*, *game_node.py*, and *uct.py*. A game state contains the state of the game board, and all relevant information about the strategy used, including whether the pruning process is added, how the game result to be back-propagated is determined, whether the min-max principle is taken into account, how many steps in the min-max principle is considered, etc. Game nodes are used to construct the search tree and contain the statistics needed for the UCB1 formula.

The UCT algorithm returns the next action to perform given a game state as root by running a given number of iterations. At the beginning of the UCT algorithm, the game node corresponding to the root game state is created. At each iteration, a game state is cloned from the root game state. During the four steps *selection*, *expansion*, *simulation*, *back-propagation* in MCTS, the only game state is maintained to the current state. One node may be added in the *expansion*. After the *back-propagation*, the game state will be garbage-collected, but not the the search tree with all the game nodes added gradually.

board.py is a helper class that enables us to compute the immediate reward given the state of the game board.

strategy.py is an abstract class defining the game strategy used. *play_game* allows us to play a game between two players easily by defining their strategy.

4.3 Optimization

When an action is performed, only part of the game board is affected. Due to this observation, some optimization is done to compute the immediate reward, given a state of the game board and an action. In the case where no stone is captured, instead of recomputing the entire value of the influence function, we only need to update it according to the action. This simplifies the computation roughly by a factor of 40.

Scenario 1	
Player A	Random strategy
Player B	UCT strategy: 1000 iterations, without pruning, delayed reward
Wins A/B/draws	2/97/1
Scenario 2	
Player A	UCT strategy: 10 iterations, without pruning, delayed reward
Player B	UCT strategy: 10 iterations, without pruning, immediate reward
Wins A/B/draws	59/40/1
Scenario 3	
Player A	UCT strategy: 100 iterations, without pruning, delayed reward
Player B	UCT strategy: 100 iterations, with pruning, ϵ=0, delayed reward
Wins A/B/draws	0/100/0
Scenario 4	
Player A	UCT strategy: 100 iterations, without pruning, immediate reward
Player B	UCT strategy: 100 iterations, with pruning, ϵ=0, immediate reward
Wins A/B/draws	0/100/0
Scenario 5	
Player A	UCT strategy: 100 iterations, with pruning, ϵ=0, delayed reward
Player B	UCT strategy: 100 iterations, with pruning, ϵ=0 and min-max, delayed reward
Wins A/B/draws	19/80/1
Scenario 6	
Player A	UCT strategy: 10 iterations, with pruning, ϵ=0, delayed reward
Player B	UCT strategy: 10 iterations, with pruning, ϵ=0.5, delayed reward
Wins A/B/draws	75/25/0
Scenario 7	
Player A	UCT strategy: 100 iterations, with pruning, ϵ=0, delayed reward
Player B	UCT strategy: 100 iterations, with pruning, ϵ=0.5 delayed reward
Wins A/B/draws	55/45/0
Scenario 8	
Player A	UCT strategy: 10 iterations, with pruning, ϵ=0, the delayed reward
Player B	UCT strategy: 10 iterations, with pruning, ϵ=0.25, delayed reward
Wins A/B/draws	64/36/0
Scenario 9	
Player A	UCT strategy: 100 iterations, with pruning, ϵ=0, delayed reward
Player B	UCT strategy: 100 iterations, with pruning, ϵ=0.125, delayed reward
Wins A/B/draws	49/51/0
Scenario 10	
Player A	UCT strategy: 10 iterations, with pruning, ϵ=0, delayed reward
Player B	UCT strategy: 10 iterations, with pruning, ϵ=0.125, delayed reward
Wins A/B/draws	63/37/0

Table 1: Play games with different strategies.

5 Experiments and Results

There is some freedom in the above definitions of rewards, due to the way the boundary is handled. We compare the two boundaries as follows. We simulate an amount of games with both players using the random strategy. We then compare how many times the game result given by the accumulated immediate reward differs from the official game result. In 1000 games, the number of times with different game result is roughly 200 games, independent of whether we use the empty boundary or the adversarial boundary. This means that these two boundaries have nearly the same performance. In the following, we choose to define our immediate reward by using the empty boundary.

In this section, we illustrate and analyze the advantages and drawbacks of using an immediate reward to prune the search tree. In order to compare different strategies, we list the results of our experiments in Table 1.

From Scenario 1, we see that the UCT strategy significantly outperforms the random strategy. From Scenario 2, the UCT strategy using the delayed reward as the value to be back-propagated in the MCTS is slightly better than using the immediate reward. From Scenario 3 and 4, choosing the optimal action according to the immediate reward is significantly better than not pruning. From Scenario 5, taking the min-max principle into account greatly outperforms only pruning. In Scenario 6 to 10, we try different values of ϵ. $\epsilon = 0$ seems to perform better in most of the cases, especially when the number of iterations is low. In fact, a low number of iterations leads to a low number of game nodes, thus resulting in a low-depth search tree when the search space is large. Finally, in the case where $\epsilon = 0.125$ with 100 iterations, $\epsilon = 0$ and $\epsilon = 0.125$ have nearly the same performance. We expect that as the number of iterations grows even larger, a non-zero ϵ will perform better than a zero one, as choosing the action with the best immediate reward does not necessarily result in the best action as discussed previously.

6 Conclusion

In this report, the general Monte-Carlo Tree Search algorithm and the most popular Upper Confidence Tree algorithm are presented. The immediate reward is defined based on the knowledge of Go and its benefits are explained. The way our implementation of UCT is done in the game of Go is shown. Multiple experiments are run to compare different strategies.

We have compared quantitatively the performance of whether or not using an immediate reward in the UCT algorithm. Due to the time complexity, we have fixed the number of iterations at each step time and noticed that using an immediate reward for pruning performs significantly better than using only the delayed reward. We have also compared a min-max strategy to a one-step optimal strategy and found better performance for the min-max strategy. Besides, we have also taken into account the influence of the choice of ϵ. We have noticed that more iterations will be needed as the value of ϵ gets larger.

As future work, it would be interesting to examine the k-step min-max strategy and study the impact of the choice of k. We could also fix a time budget at each time we select an action to perform, rather than fixing the number of iterations in MCTS. This would make the comparison easier. Further optimization could be done with parallel computing. Other variants of MCTS combined with immediate reward might result in better performance.

Acknowledgments

We would like to express our special thanks of gratitude to our supervisor Odalric-Ambrym Maillard, who guided us through the project and provided insightful advice, as well as our professor Alessandro Lazaric, who taught us the foundations of reinforcement learning.

References

[1] P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2-3):235–256, 2002.

[2] P. Baudi. *MCTS with information sharing*. PhD thesis, Ph. D. dissertation, Charles University in Prague-Faculty of Mathematics and Physics, 2011.

[3] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2012.

[4] B. Brügmann. Monte carlo go. Technical report, Citeseer, 1993.

[5] P. I. Cowling, E. J. Powley, and D. Whitehouse. Information set monte carlo tree search. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(2):120–143, 2012.

[6] L. Kocsis and C. Szepesvári. Bandit based monte-carlo planning. In *European conference on machine learning*, pages 282–293. Springer, 2006.