Upper Confidence Bound Algorithm for Monte-Carlo Tree Search in Othello

May 8, 2022

Abstract

The goal of this project is to make a strong computer program to play a board game, Othello, without using deep neural networks. Applying the upper confidence bound method in multi-armed bandit problems, I implement a heuristic algorithm for a game tree search problem. Then, I present this program outperforms other algorithms by conducting simulated competitions.

1 Introduction

In March 2016, AlphaGo, a computer program that plays the board game Go beat Lee Sedol in a five-game match. This was the first time a computer Go program has beaten a 9-dan professional without handicap. AlphaGo is based on several techniques in [SHM+16], published in January 2016. The phenomenal success of AlphaGo and its successors heavily relies on the emerging development of deep convolutional neural networks and deep reinforcement learning techniques in the early 2010s, but another significant part of the breakthrough precedes them. Computer Go programs, in fact, could not beat even amateur players until Monte-Carlo Tree Search (MCTS) algorithm ([KS06], [Cou06]) was invented around 2006. Applying methods in multi-armed bandit problems to a tree search repeatedly, this algorithm effectively evaluates the state of the game looking into promising future situations and dismissing unlikely, less important situations. A survey describes many applications of MCTS to games and nongames, including Othello [BPW+12].

Board games like Othello, Chess, Shogi, and Go are classified as zero-sum games for two players with finitely many pure strategies. By Nash's Min-Max Theorem, it is indeed well-known that there exists an optimal strategy for the players in these games. Due to the enormous size of the policy spaces, however, calculation of the optimal policy is computationally difficult. The history of computer programs for such board games is, therefore, a journey to find effective heuristic approaches to find better policies. According to [Ito13], the sizes of the policy spaces are estimated as 10^{60} for Othello, 10^{120} for Chess, 10^{220} for Shogi, and 10^{360} for Go. These sizes are good indicators of the difficulties for computer programs, which are poor at dealing with human-like intuitive heuristics. This is evident from the history: computer programs beat top-level human players for the first time in 1997 for Othello, in 1997 for Chess, in 2013 for Shogi, and in 2016 for Go.

The purpose of this project is to study and evaluate the aforementioned MCTS algorithm without deep learning techniques by applying it to the relatively simple board game, Othello. In particular, we study MCTS with Upper Confidence Bound 1 (UCB1) algorithm. UCB1 in a multi-armed bandit problem is a method that balances the exploration-exploitation tradeoff by deterministically considering the sum of the estimated value of an arm and a decaying exploration bonus [ACBF02]. By conducting simulated competitions in Othello, we present this computer program outperforms other simple algorithms.

The rest of the article is structured as follows. First, a game of Othello is defined as a game tree. Then, after introducing an evaluation component for the game tree search program, we propose MCTS with UCB1, as well as two simpler algorithms. These algorithms are compared in a simulated competition. Lastly, the computer program's performance against a human player is discussed.

2 Othello Game Tree

In this article, we use the official rules of Othello. The game tree, illustrated in Fig. 1 is defined as follows. The game proceed with time $t = 0, 1, \ldots, 60$. A tree node corresponds to state s_t of the game at time t. At each node, the player chooses the place for their new disk as an action, and then the game state transits from s_t to s_{t+1}. This

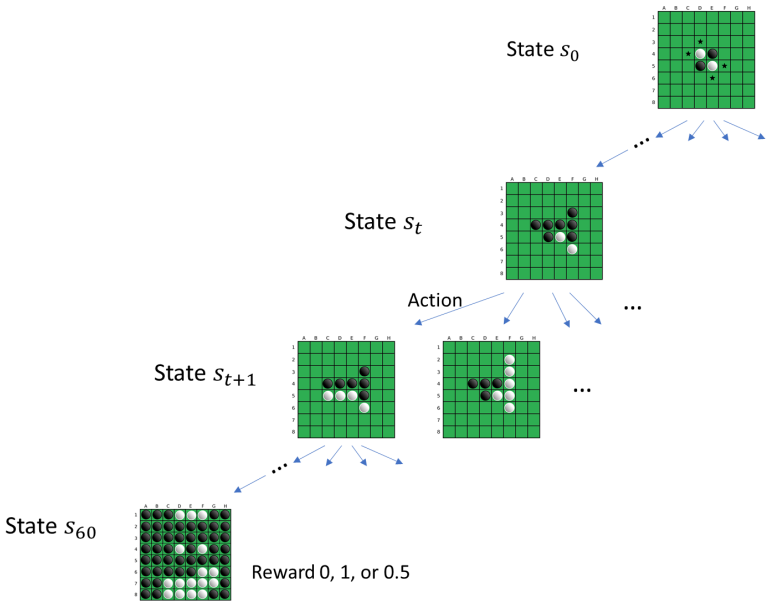

Figure 1: Illustration of Othello Game Tree.

action and the transition forms an edge from node s_t to s_{t+1}. Considering all of the possible states and actions, the game tree is constructed from the initial state s_0 to a number of leaf nodes s_{60}. The players' decisions on actions lead the game from the root node to one of the leaf nodes. Then, a player gets a reward of 1 if the leaf node represents a win for them, 0 if it represents a loss, and 0.5 if the game result is a draw.

3 Game Tree Search Algorithms

In this section, we first introduce the Playout method, a key component for probabilistic evaluation of a given game state in the following tree search algorithms. Then, using the Playout method, we define three tree search algorithms, namely, Shallow Search, Min-Max Search, and MCTS with UCB1.

3.1 Playout

Playout, sometimes also referred to as Rollout, is an algorithm for the evaluation of a given game state. Alg. 3.1 shows the algorithm of Playout. In an evaluation of state s_t, this algorithm simulates the rest of the game by choosing an action of a player with the same probability for all of the available actions at time $t, t+1, \ldots, 59$, and returns the value of the reward.

The intuition behind this algorithm is that the return value presents a noisy observation of the advantage (or disadvantage) of the player at state s_t. By repeating Playout multiple times, one can evaluate the state with a relatively inexpensive computational cost without conducting a brute-force search.

Note that the mean evaluation by repeating multiple Playouts does not converge to the true value of the state for the optimal strategy. While Playout assumes that the players take any action with the same probability, the actual players choose (or attempt to choose) the best action that maximizes their own rewards.

3.2 Shallow Search

Shallow Search is a simple algorithm using the Playout method. Alg. 3.2 shows the algorithm of Shallow Search. To choose an action at state s_t, it repeats Playout n times for each of the possible s_{t+1}, where n is a given parameter. The action corresponding to s_{t+1} with the highest estimated value is considered to be the best action in this algorithm.

A drawback of Shallow Search is that the evaluation by repeated Playouts is quite inaccurate because of the large size of the sub-tree following s_{t+1}. The limitation of Playouts noted above also shows up evidently.

Algorithm 3.1. *Playout*

Input: state s_t

If the player wins at s_t **then**

    return 1

ElseIf the player loses at s_t **then**

    return 0

ElseIf the player draws at s_t **then**

    return 0.5

Else

    choose an action with probability $1/|A_t|$ from the set of available actions A_t

    calculate the next state s_{t+1}

    return $1 - Playout(s_{t+1})$

EndIf

Algorithm 3.2. *Shallow Search*

Input: state s_t, parameter n

For each s_{t+1}

    repeat $Playout(s_{t+1})$ n times and record the mean value

EndFor

return the action corresponding to s_{t+1} with the maximum recorded value

Even though the value of s_{t+1} is estimated to be high under the assumption of repeated Playouts, the opponent player's intentional choice of the action at s_{t+1} can lead to a disadvantageous situation.

3.3 Min-Max Search

To overcome the drawback of Shallow Search, Min-Max Search looks some steps ahead of the situation. Alg. 3.3 shows the algorithm of Min-Max Search. The algorithm evaluates each of s_{t+1} as follows. Firstly, it repeats Playouts n times for each of the child nodes. Secondly, it chooses s_{t+2} with the minimum value and *expand* it. The expansion of a node means that the algorithm prepares its child nodes for the next starting points of playouts. Thirdly, it repeats Playouts n times for each child of the expanded node. Fourthly, it chooses s_{t+3} with the maximum value and expands it. Similarly, it repeats Playouts and chooses the miminum and maximum nodes alternately for $s_{t+4}, s_{t+5}, \ldots, s_{t+k}$, where k is a given parameter. After choosing s_{t+k} for each s_{t+1} in this way, the algorithm returns the action for s_{t+1} with the highest value of s_{t+k}

An intuitive interpretation of Min-Max Search is that the algorithm looks k steps ahead of the game. In the algorithm, the player and the opponent are assumed to maximize their own rewards estimated by Shallow Search at each time in the future.

The drawback of this algorithm is two-fold. For one thing, the prediction of future actions is not robust. Even though the evaluation by repeated Playouts is erratic, only the most likely state is expanded at each layer. The inaccurate prediction in an early stage can thus direct the subsequent tree search to a wrong sub-tree and eventually mislead the entire evaluation. For another, this algorithm can become computationally expensive because it internally conducts Shallow Search at each layer. Every time it conducts Shallow Search internally, it needs n Playouts for each node. Rather than trying the same number of Playouts for these nodes, the limited computational resource should be focused on evaluations of promising states.

3.4 Monte-Carlo Tree Search with Upper Confidence Bound 1

The drawbacks of Shallow Search and Min-Max Search can be overcome by MCTS.

Alg. 3.4 shows the algorithm of MCTS with UCB1. UCB1 for the multi-armed bandit problem is a method that balances the exploration-exploitation tradeoff by considering the current estimated value of the arms as

Algorithm 3.3. *Min-Max Search*

Input: state s_t, parameter n, k

For each s_{t+1}

  expand s_{t+1}

  For each i in $2, 3, \ldots, k$

    repeat $Playout(s_{t+i})$ n times on each of the child nodes s_{t+i} and record the mean value

    If i is even **then**

      expand the child node with the minimum value

    Else

      expand the child node with the maximum value

    EndIf

  EndFor

EndFor

return the action corresponding to s_{t+1} with maximum value at the expanded node s_{t+k}.

Table 1: The average reward in the simulated competition.

	vs Shallow Search	vs Min-Max Search	vs MCTS with UCB1
Shallow Search	-	0.5875	0.325
Min-Max Search	0.4125	-	0.25
MCTS with UCB1	**0.675**	**0.75**	-

well as a deterministically decaying exploration bonus. In MCTS with UCB1 algorithm, the UCB1 method is applied at each layer of the future prediction to choose which sub-tree to be further inspected. Once a node tries n Playouts, it is expanded. After conducting N playouts in total, the algorithm chooses the action corresponding to the state s_{t+1} on which the most Playouts are conducted.

This algorithm moderately incorporates the advantage of Min-Max Search: at an odd layer, the UCB1 method is more likely to choose a child node with a high reward for the computer program, whereas at an even layer, it chooses a child node with a high reward for the opponent. The multi-layered UCB1 method with variable r switching players' viewpoints at odd or even layers enables the robust and cost-effective search in the game tree.

4 Simulation

We compare Shallow Search, Min-Max Search, and MCTS with UCB1 by conducting a simulated competition. In each matching, both computer programs play 40 games as the first player and 40 games as the second. The performance of computer programs depends on the parameters. In particular, MCTS with UCB1 becomes very strong with sufficiently large N. The method is theoretically proven to find the optimal arm after sufficient time for a (single-layered) multi-armed bandit problem [ACBF02]. For a fair comparison, we set parameters so that each game takes roughly seven or eight minutes on average. The parameters are $n = 100$ for Shallow Search, $n = 10, k = 3$ for Min-Max Search, and $n = 50, N = 1200$ for MCTS with UCB1.

Table. 1 presents the result of the simulation. MCTS with UCB1 outperforms the other algorithms. This implies that it actually overcomes the simple algorithms' drawbacks explained in the previous section under the practical limitation of the computational resources. The average time for a game is 6.9 minutes for Shallow Search, 8.1 minutes for Min-Max Search, and 7.4 minutes for MCTS with UCB1.

Shallow Search plays better than Min-Max Search despite the fact that we designed the latter to overcome the drawback of the former. The reason for this is explained by the given time limitation and the parameter sets. Since Min-Max Search is computationally inefficient, it cannot repeat Playouts sufficiently many times at each layer. Thus its deep search in the game tree tends to end up being inaccurate, in comparison with the Shallow Search, which repeats Playouts for fewer nodes. In this regard, MCTS with UCB1 mitigates inaccurate estimation by striking a balance of concentration of computational resources on important nodes and deeper prediction in the game tree.

Algorithm 3.4. *Monte-Carlo Tree Search with Upper Confidence Bound 1*

Input: state s_t, parameter n, N

Data Structure: Node has variable m, r // m denotes #playouts and r denotes #wins in playouts

Make a tree graph G with a root node s_t and its child nodes s_{t+1} with variable $m = 0,\ r = 0$

For each i in $1, 2, \ldots, N$

  set current node c $\leftarrow s_t$

  While c is not a leaf node of G **do**

    let t be the sum of c's child nodes' m

    for each of the child nodes of c, calculate UCB1 score $\frac{r}{m} + \sqrt{\frac{2 \log t}{m}}$ (If $m = 0$, this is ∞.)

    set c to a child node with the maximum UCB1 score

  EndWhile

  let $r' \leftarrow Playout(c)$

  While c is not a root node of G **do**

    for variable of node c, let $m \leftarrow m + 1$

    If $m = n$ **then** expand c and add the child nodes with $m = 0,\ r = 0$ to G **EndIf**

    If c is at an odd layer of G, **then** for variable of node c, let $r \leftarrow r + r'$ **EndIf**

    If c is at an even layer of G, **then** for variable of node c, let $r \leftarrow r + 1 - r'$ **EndIf**

    set c to the parent node of c

  EndWhile

EndFor

return the action corresponding to s_{t+1} with the largest m.

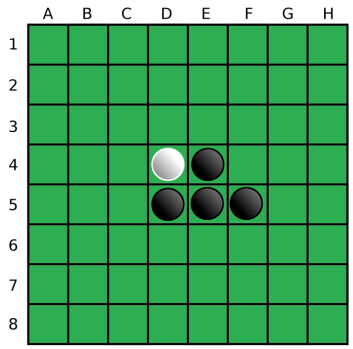

Figure 2: A common state at an early stage of a game. Action 4F is known to be a bad choice for player white.

5 Discussion and Conclusion

In this article, we studied computer programs that search the Othello game tree for a competitive strategy. We defined and implemented MCTS with UCB1 algorithm that robustly controls computational resources on inspection of important future situations. In the simulated competition, we showed this program performs well against other algorithms even with practical time limitations.

This may lead one's interest into a new question: how it performs against a human player? We provide a few, somewhat subjective, remarks on this matter. An experimental competition with some human players (undergraduate and graduate students including me) results in some wins and losses. With this outcome, the program is considered to be neither very strong nor weak. Our observation is that it plays poorly for the first actions but becomes stronger toward the end of the game. For computer programs in this article, it is difficult to find good action at an early stage since the size of the game tree to be searched is large. On the other hand, a human player can take advantage of tips such as "choose an action that reduces the opponent's available actions" or "do not choose action 4F in the situation of Fig. 2." As the game proceeds, the optimization problem becomes easy for computer programs since the size of the sub-tree shrinks, whereas a human player should deal with less-studied specific situations, rather than utilize tips for common situations.

Another direction for future study is to apply deep learning techniques to this problem. Instead of Playout in our program, a convolutional neural network is used to evaluate the state, and it can be trained with deep reinforcement learning techniques. Since computer Othello programs can become stronger than human players, future research should pay attention to finding useful tips for humans and supporting human players' training.

6 Acknowledgements

Although I did not cite it in the main text, I studied with a book about AlphaGo. [Ots18]

References

[ACBF02] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2):235–256, 2002.

[BPW+12] 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.

[Cou06] Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *International conference on computers and games*, pages 72–83. Springer, 2006.

[Ito13] Tsuyoshi Ito. The frontier of computer go – from the perspective of game informatics. In *Journal of Information Processing Society of Japan*, volume 576, pages 234–237, 2013.

[KS06] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *European conference on machine learning*, pages 282–293. Springer, 2006.

[Ots18] Tomofumi Otsuki. *Saikyo Igo AI, AlphaGO Kaitaishinsho*. Shueisha, 2018.

[SHM+16] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.