STA3431 Course Project - Self Directed

Author: Siyi Wei (2020 Master Student - Department of Statistical Science)

Email: siyia.wei@mail.utoronto.ca September 14, 2020

* My appreciation to Prof. Jeffrey S. Rosenthal and Ruyi Pan.

1 Monte Carlo Tree Search in Board Game

In this project, I want to investigate and implement one important technique which has been widely used in zero-sum games. The Monte Carlo Tree Search (MCTS). MCTS was first introduced in 2006, it is a heuristic search algorithm for decision processes. It combines knowledge from Tree Data Structure, Game Theory and the main topic I will focus, the Monte Carlo Sampling.
MCTS is highly similar to MinMax Tree Search. MinMax Tree Search is also a search algorithm that "Maximize the minimum gain". In short, it was designed for zero-sum games to maximize the minimum reward at every decisions. However, MinMax Tree Search is a greedy algorithm, it will search every possible end node of the game states, then justify who is the winner and return the status to the parents' nodes. MinMax could achieve a very high winning rate. But it also has one critical flaw. MinMax Tree has a quadratic time complexity $O(b^m)$. This flaw limits its usage in lots of board games like GO. To deal with this critical flaw, MCTS then been introduced to reduce time complexity. One of the most famous example, AlphaGO, combined both MCTS and deep learning to improve Monte Carlo Sampling and achieved astounding results.

2 How Monte Carlo Tree Search Works

The core idea of Monte Carlo Tree is obvious: Use Monte Carlo to reduce time complexity. Traditionally, MinMax Tree will expand every possible end state and evaluate all of them. However, we could improve this process using Monte Carlo Methods. Since at each incomplete game state, we could expand all the possible moves, then justify the winning rate by running simulations at each move. Then the estimated winning rate will converge to the true winning rate as the amount of simulations increase. Moreover, we could apply different Monte Carlo Methods at this step and compare the results.

2.1 Principle of Operation

We want to formulate the process of Monte Carlo Tree first. If we treat a Monte Carlo Tree search as a black box function. The input of this function will be the current game state. The output of this function will be the next move. Given an example of Tik-Tak-Toe. The input will be the game board and the output will be the estimated optimum move for the given game board. There are 4 main steps in Monte Carlo Tree Search:

1. **Selection Process:** The selection start from root node R. Then keep select the successive child nodes until one leaf node L is reached. We treat the root node R as the input (current game state) of the function. The leaf node L could be any possible successive child node where no simulations has been initialized.

2. **Expansion Process:** After we select the valid child nodes L. Unless L already be the end state, which either represent WIN=1, LOSE=-1 or a DRAW=0. We will expand node L by creating all possible child nodes of L. Then we randomly select one child node C represent the game state after any possible move from L.

3. **Simulation Process:** Once we randomly select a child node C. We want to get the estimated winning rate from the current game state C. We then run the Monte Carlo simulations on current game state until the game ends.

4. **Back Propagation:** We update the information C to R using the results of the simulation.

5. **Optimization:** We then return the move of current game state with the highest estimated winning rate.

As one may notice, the advantage of MCTS is due to the Simulation Process. By simulating the possible end state we do not need to traverse all the possible end states. Yet we still got a reasonable estimation of the winning probability of the moves. Also, we can improve MCTS even further by improving the simulation process. For example, AlphaGo applied deep learning to improve the simulation process, which could generate the possibility of winning using neural network, and significantly reduced the time in estimating the winning rate when play games. For the trade off, the neural network has to cost more time in training and simulating.

3 Monte Carlo Tree Search in Tik-Tak-Toe and its Variants

3.1 Introduction

I want to apply Monte Carlo Tree Search in a 4x4 Tik-tak-toe and a 6x6 board Gomoku. The possible end states could be at most 2^{16} and 2^{49}. So that it is infeasible for traditional greedy algorithms, like MinMax tree search to do such work. For example, play a 4x4 Tik-Tak-Toe using MinMax Tree Search with some optimizing technique like alpha-beta pruning. It still take more than 20 seconds in some high performance language like java, and much longer in python. People usually focus on the selection process. Which is the foundation of MCTS. The widely used UCT formula which balances expansion and exploration process. **However, In this project I want to explore other simulation possibilities rather than vanilla Monte Carlo sampling.** With even the simplest Monte Carlo method we can generate the estimated optimized move for 4x4 Tik-tok-toe in less than 3 seconds. With a descent winning rate and some game tricks founded.

3.2 Setting the Problem

Setting the problem for a board game is extremely hard. Unlike a static distribution for an image or a sequence of event. In a zero-sum game. The distribution of the winning rate are dynamic, which depends on your opponent's movement and your decision. In which case the game process is defined as a Markov Chain. So in this project I only tried modified importance sampling and compare its improvement with the vanilla MCTS method.
Our distribution parameters θ is a $n \times n$ Bernoulli random vector, which represent the game state. For example, $\beta_{1\times1} = 0$ represents O land on 1st row and 1st column. $\beta_{1\times2} = 1$ represents X land on 1st row 2nd column. However, it is hard to determine the distribution

of θ. Especially when θ is partially fulfilled. Then we need to estimate the marginal distribution θ^* from θ.

3.3 Vanilla Monte Carlo Simulation in MCTS

Given an example for vanilla monte carlo sampling, at rth step of the game. Given game state θ^r. We then want to choose parameter $\widehat{\theta}^{r+1} = \text{argmax}\, P(\theta^{r+1}|\theta^r)$. However, if θ^{r+1} is not the end state of the game, we then need to estimate $P(\theta^{r+1}|\theta^r) = \frac{\sum_{i=1}^{N} P(\theta_i^{r+2}|\theta^r)}{N}$ given N is the current samples we have obtained at step $r+2$, and recur until the end state reached. By induction we can see the sampler works, and to formulate this process.

Input: $\theta^r = [-1, 1, 0]^{n \times n}$ where θ^r is not an end state. (No winner nor draw)

Output: An updated tree structure so the selection process could gave a more accurate $\widehat{\theta}^{r+1} = \text{argmax}\, P(\theta_i^{r+1}|\theta^r) \ \ \forall i$ where i represent the possible positions.

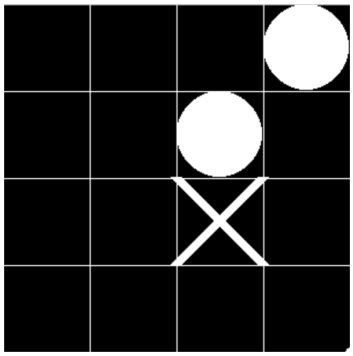
Choose i where i is the available land on board, Set $\theta^* = \theta_i^{r+1}$
while TRUE :
    if (θ^* is one of the end state) :
        Update all the winning probability from position r to current position $*$
        End While Loop
    else:
        Choose j (one of the available positions at step θ^{*+1})
        Set $\theta^* = \theta_j^{*+1}$

Algorithm: Optimize next step using vanilla Monte Carlo sampling in MCTS

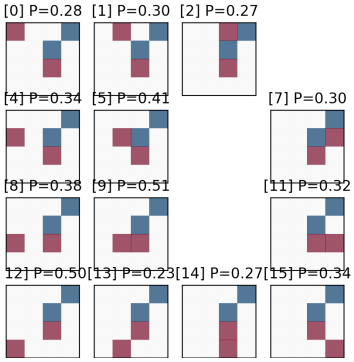
3.4 Defects of vanilla Monte Carlo Sampler

Even though we could save numerous time using vanilla Monte Carlo Sampling. However if we increase the grid size. The time complexity still increase quadratically if we want to keep the same winning rate. For example, Go has a 19x19 size grid and Gomoku has a 15x15 game grid. It would be extremely time consuming if we sample randomly using Monte Carlo Simulation and expected a descent winning rate since the simulation need more steps to get a end state. However, recall the game rules, a play win only if he has the grid length (assume Square grid) pieces in a line. Which implies the following conclusions for our games. If some placement position have empty neighbors, than it does not worth us to sample from this position. Since the possibility the player could win start from this position is significantly lower than the optimum positions. We could verify this result using the probability plot [Figure 1]. Someone may argue this phenomenon only holds for some specific games. However, among lots of games. We could always find rules which reveal some positions are more valuable than the others.

From the probability distribution plot we could find. There are 4 positions so called "empty positions". Which are 0, 4, 8 and 12. Among those 4 positions. Only position 12 has a relatively high winning possibility. The other three are relatively low comparing with the optimum position 9. Which has an estimated winning probability 51%. We could



((a)) Game Board



((b)) MC simulated winning probability

Figure 1: Current Board and corresponding winning probability

then naturally connect it with the knowledge we learned in Monte Carlo. Even though we cannot determine the distribution of an incomplete game state. We could still justify which positions could have a higher winning probability. Thus we could use the idea of Importance Sampling to improve the Monte Carlo process for some specific games. Importance sampling could regularize or prune less valuable tree branches in our example. I then implement the variant of simulation process and verify the improvements. There should be improvements on both efficiency and time complexity. Since Improved MC need less time to finish a simulation, but more reasonable than random playing.

3.5 Importance Monte Carlo Simulation in MCTS

The variant of Monte Carlo simulation I have claimed follows some idea I get from importance sampler. In short, we could use our prior knowledge to justify which positions could possibly have higher winning rate. Then instead distribute chess uniformly on all the valid positions. We will distribute the chess by our prior knowledge. Again, to formulate our algorithm. The pseudo code of our algorithm will be the following. The only difference is now we select j from optimum positions. In our case, the optimum positions are the empty position with at least 1 chess in its neighbor.

Input and Output: Same as above

Choose i where i is the available land on board, Set $\theta^* = \theta_i^{r+1}$
while (TRUE) :
    if (θ^* is one of the end state) :
        Update all the winning probability from position r to current position $*$
        End While Loop
    else:
        Choose j (one of the optimum positions at step θ^{*+1})
        Set $\theta^* = \theta_j^{*+1}$

Algorithm: Optimize next step using improved Monte Carlo sampling in MCTS

3.5.1 4x4 Tik-Tak-Toe

We could compare two simulation methods in a 4x4 Tik-Tak-Toe game. I have construct a self compete program for two methods and set the same parameters for both algorithms. Which is 500 simulations for each step. The maximum depth for search will be 20, which exceeds the maximum depth (16) for our game. Each algorithms could start first for half of the games. Meaning if x (represent Improved MC) start first for this game, then o (represent Naive MC) will start first for the next game. For 50 independent games, Improved MC win 16 of them. Naive MC win 5 of them and 29 games draw. Often the optimum result is draw. However we can see Improved MC win most of them since its simulations are more strategic. To verify the result I tried another 50 simulations. Where the Improved MC win 12 of them. Naive MC win 4 of them and 34 games draw. Overall I get the following result:

Improved MC win	Naive MC win	Draw
28	9	63

For the time complexity. We need to discuss by conditions. Theoretically, Improved MC need less steps for each simulation thus it should need less time compare to Naive MC. However, Improved MC need more time to calculate the possible positions. In another word, its distribution is more complicate thus need more time to construct it. So for the running time. Improved MC need more time than Naive MC in reality. But if we exclude the distribution construction time for both method. Improved MC need less time than Naive MC. [Figure 2] There are some cases Improved MC need more time than Naive MC. Those cases usually happen with Improved MC goes first. Thus need to expand more child nodes than next step.

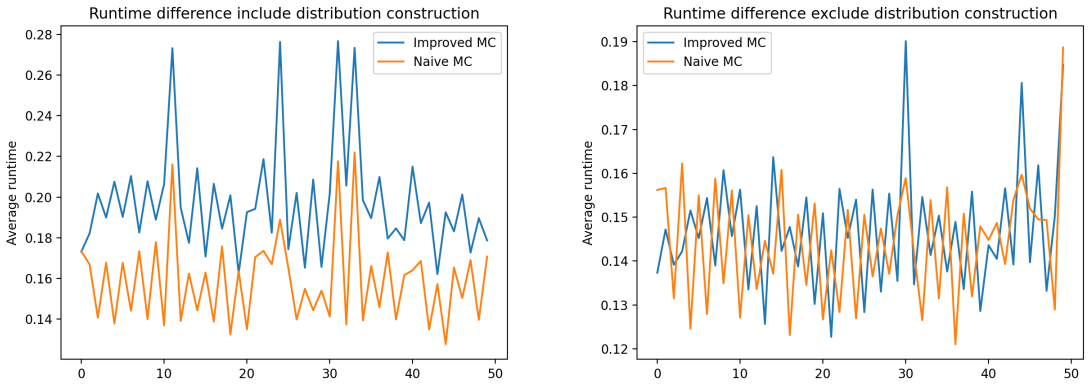


Figure 2: Compare the run time between different conditions in 4x4 Tik-Tak-Toe

3.5.2 6x6 Gomoku

Since the 6x6 Gomoku significantly increase the complexity and possibilities. We readjust the parameters where the simulations increase to 700 for each step and the depth stays 20. Due the huge time complexity, we reduced the amount of games to 60. The winner board and running time are displayed as below [Figure 3]:

Improved MC win	Naive MC win	Draw
22	15	23

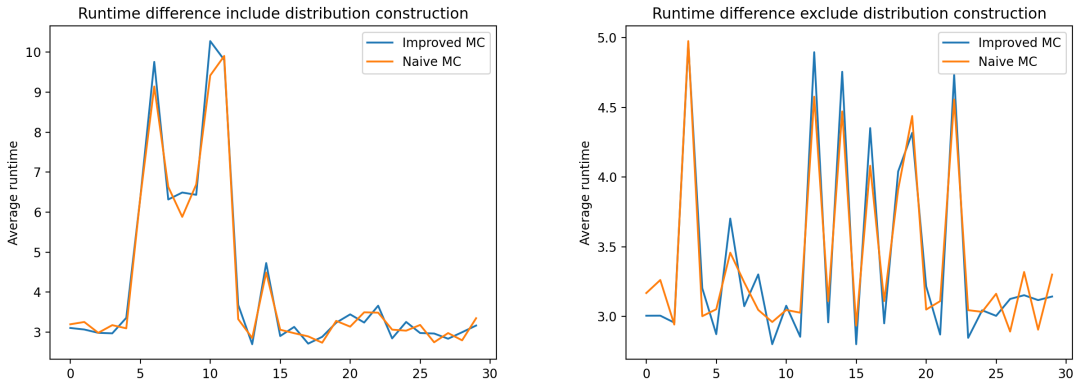


Figure 3: Compare the run time between different conditions in 4x4 Tik-Tak-Toe

We can see very little difference for their running time. However, the Improved MC still got a better winning rate even with a relative large simulations. If we increase the simulations to even larger. The Improved MC will have similar winning rate as Naive MC.

3.6 Defects of Improved Monte Carlo Sampler

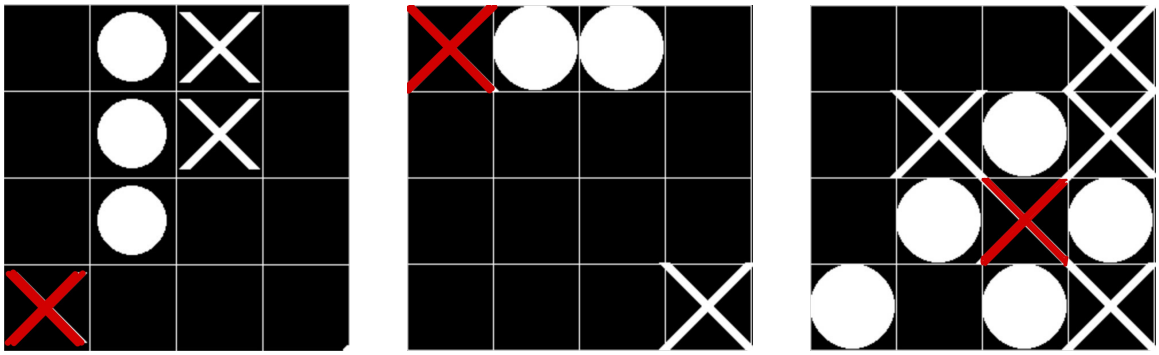
The defects of Importance Sampler is quite clear. To find the optimum position. We need to search the board to find all the optimum positions. This step usually cost more than uniform sampling along the empty positions. So instead decrease the time complexity. It increase the time complexity. The increment of running time decreases as the simulations increase. When we have less simulations. Improved Monte Carlo shows better performance in winning rate than Naive Monte Carlo. However, we want more simulations to increase winning probability. Instead of improving the Monte Carlo Simulations. We could further investigate how to reuse the simulations in unrelated plays.

3.7 Gaming tricks learned by sampling

In a 4x4 Tic-Tac-Toe. I observed by Monte Carlo Sampling method. The algorithm can generalize two important tricks called "Block". Means if we have O on some cross, the algorithm will try to block all possibilities I could win passing this O. This trick could usually learned with iteration more than 50 times. Even more, when the iterations larger than 100 times. I observed the algorithm will try to block me and grow his own Xs at the same time. Which demonstrate some interesting gaming tricks could be "learned" just using simulations [Figure 4]. For the following examples the red cross represents the algorithm final move.

4 Conclusion

In this project. I implement MCTS on two different games. Moreover, I have initialized a different sampling strategy: Importance Sampler. I argued such strategic sampler could be initialized in most of the zero-sum games. Such samplers could balance the trade off of time complexity and winning rates. The time complexity often increase due to a more complex sampling rule. But the winning rate will also increase due to a more valuable



((a)) iter = 10 ((b)) iter = 50 ((c)) iter = 100

Figure 4: Compare the game tricks between different sample sizes

simulation than uniformly sampler. Such trade off will only be balanced in small sample sizes. With the increase of samples the time complexity and winning rate will converge to the vanilla implementation. Moreover, I found gaming tricks could be learned via Monte Carlo simulation. Usually with more simulations more efficient game tricks could be found.

5 References

1. MCTS Algorithms `https://en.wikipedia.org/wiki/Monte_Carlo_tree_search`
2. Cameron.B, Edward.P, Daniel.W and others "A Survey of Monte Carlo Tree Search Methods"
3. Alexander.Z, Brent.H and Mark.R "Monte-Carlo Tree Search for Simulation-based Strategy Analysis"
4. KyushikMin "Game Visualization" `https://github.com/hayoung-kim/mcts-tic-tac-toe`