# Adversarial search of Monte-Carlo simulation for 3D Tic-Tac-Toe

Kohei Kawasaki
kawas009@umn.edu
Carlo Hernandez
herna463@umn.edu

May 11, 2016

## 1 Introduction

### 1.1 Brief Description of problem

As the game is getting more complex and deeper strategically, the heuristic function and cost function is getting useless due to absence of absolute evaluation of current state and forward step. That node will be expanded exponentially and it could be said impossible to use $A*$ search or $IDA*$ since both algorithms depends on the validity of evaluation function and strategically complexity causes evaluating a each move to difficult. In alpha-beta pruning, if we put bounds on the possible values of the utility function, the nwe can arrive at bounds for the average without looking at every number. Therefore Monte-Carlo simulation(Tree search) works an alternative evaluation of Alpah-Beta prunning or other search algorithm. From a start position the algorithm play thousands of games against itself, using randomly chosen move and evaluate each note by the statical win percentage. Monte Carlo Tree Search MCTS, does not rely on a positonal evaluation function, however this approach is a general algorithm and can be applied to mamy problems. The most promising result was the game of Go. In this project, I am going to work on 3D tic-tac-toe from the approach of monte carlo tree search. 3D tic-tac-toe is the three dimensional version of commonly played board game. The strategy of game expanded due the expansion of dimension. Assume N as the length of each side, the size of the board game $N^3$. Thus the maximum tree size is the$N^3!$. In terms only size of board game, from 5 side length, it is over the game of go. However because of the simplicity of the game rule, its branching factor still is inferior to game of go. For the similarity of game, I implement algorithm, which is commonly used Go, to 3D tic-tac-toe. Here is the categoris of problems: (1) Single player ploblems(called optimization

Table 1: Different types of problems.

| | One player | Two player | Multi Player |
|---|---|---|---|
| Deterministic | TSP, PMP | Go, Chess, Shogi | Chinese Chekers |
| Stochastic | Sailing Problem | Backgammon | Simplified Catan |

problems), (2) two opponent problems, and (3) problems with multiple opponents. Likewise Go, Chess or Shogi, 3D tic-tac-toe is also non-cooperative two player deterministic game.

### 1.2 Monte-Carlo Tree Search

Monte Carlo Tree Search is the best first search method that does not require a positional evaluation function based on a stochastic exploration. And piling the result of explorations, the algorithm gradually builds up a game tree in memory, and successively becomes to more accurate estimation of the variable moves. MCTS consists of four main parts. (1) Selection, (2) Expansion, (3) Simulation and (4) Back propagation. These part are repeated as long as there is time left. The steps are as follows. (1) From the root node, the selection strategy is applied recursively until unexplored node selected, (2) then one unexplored node is added to the tree, (3) one simulated game is played on the node. (4) Finally, the result of this game is back propagated in the tree.

## 2 Related works

The first Monte-Carlo simulation appeared on early 1990. B. Abramson developed (expected-outcome) model which returns expected value of the games outcome[1]. A standard model of static evaluators is Min-Max tree search. For the difficulty of setting of evaluation, however, the design is viewed as impossible. Moreover, it was too complicated to evaluate the accuracy due to the absence of standard evaluation to grasp a strategy in the entire game. From this perspective, Abramson defined as (Expected-Outcome) values as

$$EO(G) = \sum_{leaf=1}^{k} V_{leaf} P_{leaf}$$

where Node G is given by a players expected payoff over an infinite number of random completions o game, k is the number of leaves in the subtree, $P_{leaf}$ is the probability to be reached in a given random play. At the bottom line, Abramson tired to make a heuristic from the statical evaluation of random game plays. Due to limitation of computer architecture at those days, the test was operated 6x6 Othello. By running a program in a tons of random game, from its value, the player make a decision.

B. Brugmann implemented simulated annealing to the Monte Carlo simulation as their project GOBBLE[3]. At the first glance because of the difference between traveling salesman problem and tree search Game like Go however, the two competing parties playing the game

in hostile way and have to optimise two sequences of moves. It seemed simulated annealing cannot be to find local variable due to the discontinuous function of the list of moves. As one of possible solution, through the simulation of random games, algorithm itself is learned. And here is the 2 point of improvement. Evaluation of the moves. at any stage of the game, moves were evaluated by the average score of the game. Selection of the moves. Simulated annealing was used to control the probability to good moves are more likely to be played first.

Boozy and Helmstetter developed two Go programs based on same basic idea[2]. That derives from their two hypothesis. First if it is possible to terminate search process of game, the process provides the best move and this process does not necessarily need domain dependent knowledge, however, its cost is exponential by the search depth. Second, knowledge-based go programs seemed impossible to improve. Thus, their paper explored a hybrid solution of these two idea, which is little knowledge based. As an domain-dependent knowledge part, they defined of an eye, and in the eye random programs not to play. Next thing is evaluation of terminal position for a random game. There are two two ways to evaluate it,for the first move of the game only, or for all moves played in the as if they were the first to be played. In their project latter one is implemented. The Advantage is the one random game helps evaluate almost all possible moves at the root, yet as a drawback, move order is ignored which is crucial to this game.

The basis of Monte Carlo Simulation came out of the limitations of other Artificial Intelligence. Classic approaches from the past relied on on a game dependent heuristic to give an AI an estimate of the current game state. However, making making mid state evaluators that reliably evaluated heuristic values are too complex and depend too much on the nature of the game. [5] Monte Carlo Tree Search does not have this problem because it does not rely on heuristics to get a guess of the game state. Rather, it builds simulations of the game and then builds statistics to choose the next action with the highest win rate of the simulations. This is done using four stages: Selection finds a state in the tree and decides based on the statistics aggregated which node to explore. There is a balance that needs to be achieved such that it does not keep picking a local optimum to allow the exploration of a possible better game state. Expansion is when the game reaches a node that is currently not in the search tree, and a new node is added. From there, Simulation takes over, making random moves for both the parties. Backpropagation happens when the simulation reaches a terminal node, and the result of that simulation is sent upwards throughout the path that the selection phase took, updating the win rates of those nodes.

In the Monte Carlo Tree search, the selection step works works in the following way. From the top root node, it select randomly until it reaches hit the node witch has no children yet. The selection strategy controls the balance between exploitation and exploration. The pruning the step witch is less likely to win is important (exploitation), on the other hand, the less promising moves still have a room to consider if its a critical way to open up a new horizon(exploration). Several selection strategies have been designed.

## 2.1 UCT

Upper Confidence bounds to applied to Trees strategy(UCT) was advocated and developed by Kocsis and Szepesvari(2006) in order to find a near optimal action in large state-space Markovian Decision Problems(MDPs)[8]. The previous approach to this problem was And the strategy is used widely nowadays in selection of each node to be expanded from its easiness to implement. From the two primary aspects of Monte Carlo planning algorithms, which is (1) small error of probability even if the algorithms is halted prematurely, and the returning the best action if enough time is given. The UCT Algorithm outline is here, In a bandit problem, K actions are defined in accordance with the set of random payoffs $X_{it}, i = 1, ...K, t \geq 1$, where each i is the index of a game. And this algorithm, keeps track of the average rewards $X_{i,Ti(t-1)}$ for all the arms and the distribution function is bounded by the upper confidence.

$$I_t = argmax_{i \in i,..K} X_{i,Ti(t-1)} + c_{t-1,Ti(t-1)}$$

where $C_{t,s}$ is a bias sequence.

$$C_{t,s} = \sqrt{\frac{2ln(t)}{s}}$$

Even though, their theoretical analysis is consistent in the optimal action as the sample number grows to infinity, in the experiment of sailing domain they found out the significant error level.

## 2.2 PBBM

As one of the other selection part strategy of Monte Carlo, in 2006 Coulom developed PBBM(Probability to be Better than Best Move)[7]. In their paper their attempt of a new framework is combining Min-max tree search algorithm into it by not backing up the min-max value close to the root, the average value at each depth like a Min-Max, but by general backup operator, which is defined below. Most of the monte carlo algorithm rely on the central limit theorem that is approached a normal distribution with mean $\mu$ and variance. In their progressive purning, the standard deviation of the best move is taken into account. However, it is very insecure in tree search, because the payback of random simulations are not identically distributed as the search tree expanded, move probabilities are changed. For the sake of dodging the dangers of completely pruning a move, it must be considered to allocate the reduced probability of exploring a bad move, instead of cutting off the move which is supposed to bad move at this moment.

$$u_i = exp(-2.4 \frac{\mu_0 - \mu_i}{\sqrt{2(\sigma_0^2 - \sigma_i^2)}}$$

where $\mu_0$ and $\sigma_0$ are the value and standard deviation of the best move, respectively. Similarly, $\mu_i$ and $\sigma_i$ are the value and the standard deviation of the move under consideration.

### 2.3 Objective Monte Carlo

Objective Monte Carlo developed in 2006 by Chaslot et al[4]. Objective Monte Carlo consists of two part, the first one is a move selection strategy and the next one is a new back propagation strategy.
In the first part, suppose Vm is the current evaluation of the move m, and $\sigma$ is the standard deviation of Vm. They defined the probability of the move m to be superior to $O_{bj}$ as

$$U_m(O_{bj}) = \frac{\sum_{i=O_{bj}}^{\infty} D_m(x)}{\sum_{i=-\infty}^{\infty} D_m(x)}$$

where,

$$D_m(S) = N(V_m, \frac{\sigma}{\sqrt{n}})$$

In the second back propagation part, their strategy returns the value relying of standard deviation of the move m to the value of move m. the Min-Max values measured by all child node in a max of random number, so it is overestimated in a sence. To avoid this error, the value returned by the $U_m(O_{bj})$ represent the urgency of the a move, the value estimated by the back propagation strategy should be close to the average value of the child node in the beginning of the experiments. As a consequence it does not require any evaluation function in the usage of this selection strategy and the backpropagation strategy. Therefore, this is applicable to any game where it is difficult or impossible to create an evaluation function without parameter tuning.

## 3 Concise description of approach to solve problem

The problem to solve is to find the most reliable monte carlo variant. Four candidates were put head to head to find the most viable candidate. These four candidates rose from ideas of defense, offense, and a combination of both. To facilitate a better illustration of how the modifications were made, the AI was written in an object oriented format. Each variant extends from the vanilla monte carlo search or one of its other children, ensuring that the variants can be traced back to have insights on what inspired the design or what went into making the variant how was.

The first implementation just simply statically analyse the probability of each step in a set of test. This is the basis for the rest of the other implementations and uses vanilla monte carlo to pick the actions with the most win rate. This is done by sampling the possible actions randomly, simulating the rest of the game for that action, and then updating the winrate of that action. This is done for any action that is randomly selected during selection.

The second implementation applies a learning system where the distribution of picking new moves will be adjusted to match the distribution of the last few simulations. This means that if a move had a high win rate in the last 100 simulations, this move will be more likely to be picked in the next 100 simulations. This is done by modifying the selection section of the code and making a separate list of winrates. These seperate lists would be normalized to

positive values using the minimum value of the old winrates, and then squared. The range is calculated using the maximum and minimum and squared. This squared range is used to obtain a random number to which the coordinates for the squared winrate closest to it can be found. These coordinates are then selected for simulation.

The third implementation would be a full monte carlo simulation that favors the use of lookup tables. For this experiment, only one lookup will be considered and this will be the win condition of the opponent. To implement this, the selection code is modified to arbitrarily raise the winrate of the node that would prevent the opponents win condition. The selection code will still use the vanilla monte carlo selection, but with the modified winrates based on whether or not the opponent win condition is detected. This would appear as random into suddenly defensive gameplay.

The fourth implementation would be a combination of the second and third implementations. It will incorporate the third implementation ability to detect the opponents win condition while still keeping track of the overall distribution of the winrates as done by implementation two. This will hopefully give a balanced effect such that the artificial intelligence would balance the notions of offense and defense. Experiment Description

# 4 Experiment

The experiments involves having these algorithms fight it out on a 3d tic tac toe board. The experiments were carried out in a consistent environment where variables were controlled. Each algorithm was put head to head against each other, resulting in 6 one on ones. These one on ones were carried out on CSE lab machines with 32gb memory and 8 core processors. To keep the performances of each algorithm against each other consistent, variables were kept constant so as to not affect the overall results. These variables were runtime, sampling size, and board size. The board sizes are kept at a four by four by four to ensure that at least one player will win the game; no ties will exist to better hash out which algorithm is better. The sampling sizes were kept at a constant 1000 so that no algorithm had the upper hand. Statistically speaking, the more samples any algorithm takes, the better its ability to evaluate the game state. Thus this was kept constant to make sure it is the ability of the algorithm that makes the algorithm better. The runtimes were kept limited as well. Since these are learning algorithms, previous experience would accumulate, to the point where one algorithm might dominate in the end game, showing not so interesting results. Rather, we kept the runtime limited to 30 seconds to ensure some randomness in the experiment.

# 5 Conclusion

# 6 Future work

## 6.1 UCT

As one of the possible attempt to the future work, Upper Confidence bound Tree search UCT could be enumerated[8]. In a MC search, evaluation of the finished random game is then used to update the statistics of all moves in the tree that were part of that game. The UCT method solves the problem of selecting moves in the tree such that important moves are searched more often than moves that appear to be bad. One way of doing this would be to select the node with the specific win rate of the time otherwise choose a random move. Whenever UCT simulation explores other moves, adding a number to the win rate for each candidate move that goes down every time the node has been visited. From the root, UCT searches a path of moves through the tree by summing up a value for each candidate position depending on the win rate and the each nodes number of visited times. The UCT value is defined as

$$UCT_{value}(parent, n) = winrate + \sqrt{\frac{\ln parent.visits}{5 * n.nodevisit}}$$

The UCT is a commonly used selection strategy and it could be got together with Convolution Neural Network which is used for a Alpha Go[9]. After studying Machine Learning, I would like to go back and combine of those works.

## 6.2 Parallelize

To be added one thing, Parallel Monte Carlo is also works as speeding up simulation. with the advent of new hardware paradigms, implementations of Monte Carlo can take advantage of these paradigms. One paradigm that Monte Carlo is set to take advantage of is parallelism in hardware. From this, four implementations Monte Carlo Search have arisen. Of the four, the two that give insight are Leaf and Root Parallelization.[6] Leaf is one way to apply parallelism to the simulation phase of MCTS. That is, multiple threads are assigned the task of carrying out simulations. This allows for multiple samples on one selection phase, giving a more accurate distribution of win rates across the search nodes. Root on the other hand seeks to parallelize from the root up. This means that full phase of MCTS are parallelized and the grand total of each search nodes are totalled up and the best action is selected based on that.

# References

[1] Bruce Abramson. Expected-outcome: A general model of static evaluation. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 12(2):182–193, 1990.

[2] Bruno Bouzy and Bernard Helmstetter. Monte-carlo go developments. In *Advances in computer games*, pages 159–174. Springer, 2004.

[3] Bernd Brügmann. Monte carlo go. Technical report, Citeseer, 1993.

[4] GMJB Chaslot, Jahn-Takeshi Saito, Bruno Bouzy, JWHM Uiterwijk, and H Jaap Van Den Herik. Monte-carlo strategies for computer go. In *Proceedings of the 18th BeNeLux Conference on Artificial Intelligence, Namur, Belgium*, pages 83–91, 2006.

[5] Guillaume Chaslot, Sander Bakkes, Istvan Szita, and Pieter Spronck. Monte-carlo tree search: A new framework for game ai. In *AIIDE*, 2008.

[6] Guillaume MJ-B Chaslot, Mark HM Winands, and H Jaap van Den Herik. Parallel monte-carlo tree search. In *Computers and Games*, pages 60–71. Springer, 2008.

[7] Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *Computers and games*, pages 72–83. Springer, 2006.

[8] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *Machine Learning: ECML 2006*, pages 282–293. Springer, 2006.

[9] 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.