*Proceedings of the 2024 Winter Simulation Conference*
*H. Lam, E. Azar, D. Batur, S. Gao, W. Xie, S. R. Hunter, and M. D. Rossetti, eds.*

# A TUTORIAL FOR MONTE CARLO TREE SEARCH IN AI

Michael C. Fu[1], Daniel Qiu[2], and Jie Xu[3]

[1]Robert H. Smith School of Business & ISR, University of Maryland, College Park, MD, USA
[2]School of Computer Science, Carniege Mellon University, Pittsburg, PA, USA
[3]Dept. of Systems Engineering & Operations Research, George Mason University, Fairfax, VA, USA

## ABSTRACT

This tutorial serves as an introductory guide to Monte Carlo tree search (MCTS), a versatile methodology for sequential decision making under uncertainty through stochastic/Monte Carlo simulation. MCTS gained notoriety from its pivotal role in Google DeepMind's AlphaZero and AlphaGo, hailed as major breakthroughs in artificial intelligence (AI) due to AlphaGo defeating the reigning human world Go champion Lee Sedol in 2016 and the world's top-ranked Go player Ke Jie in 2017. AlphaZero, without requiring any domain-specific knowledge beyond the game rules (tabula rasa), achieved remarkable success by surpassing previous benchmarks in Go and outperforming leading AI opponents in chess (Stockfish) and shogi (Elmo) after just 24 hours of MCTS-driven reinforcement learning. We demonstrate the building blocks of MCTS and its performance through decision trees and the game of Othello, and provide an empirical simulation study for the latter.

## 1 INTRODUCTION

*Monte Carlo tree search* (MCTS) is a sequential decision-making algorithm based on a tree structure and randomized sampling. The name MCTS was first used by Rémi Coulom (Coulom 2006), who presented a randomized tree search algorithm for Crazy Stone, an artificial intelligence (AI) program for playing Go, the ancient Asian board game. This randomized approach, inspired by algorithms proposed in Chang, Fu, Hu, and Marcus (2005), represented a paradigm shift from previous Go-playing programs that had focused on deterministic search algorithms, e.g., alpha-beta pruning, widely used for combinatorial games such as chess (Newborn 2012).

MCTS was employed by Google DeepMind in its AlphaGo program to generate game play data used to train two deep neural networks (DNNs), one used as a policy network and the other used as a value network. AlphaGo claimed its first important victory over a professional human player when it defeated the reigning European Go champion Fan Hui in October 2015 with a 5-0 sweep. Less than a year later, AlphaGo defeated then reigning world Go champion Lee Sedol in March 2016 (4-1; Seoul, South Korea), which was followed by a decisive victory over the world's No. 1 ranked Go player Ke Jie in May 2017 (3-0; Wuzhen, China). The remarkable success of AlphaGo was hailed as an important milestone in AI development, and the fascinating stories of AlphaGo were documented in a short movie (Silvestri 2017).

MCTS played a dual role in the development of AlphaGo (Silver et al. 2016) and its successor AlphaGo Zero and AlphaZero (Silver et al. 2017). It was used as an algorithm to search for the best move but also as the computational engine that executed millions of *self-play* games to allow AlphaGo to use reinforcement learning (RL) to train its neural networks for predicting moves and values of a game state. Self-play facilitated a shift in the learning paradigm of the initial training stage used by AlphaGo, which initially trained on a library of recorded Go games played by human players, i.e., supervised learning, to initialize its neural networks. The new approach allowed AlphaZero to learn "from scratch" (tabula rasa), only being provided the rules of the game, enabling it to train/learn without requiring the benefit of expert domain knowledge. Even so, AlphaGo Zero was able to trounce its predecessor/parent AlphaGo 100 games to 0, as announced by DeepMind in the October 19, 2017 issue of *Nature*, Silver et al. (2017). No longer limited to just the game of Go, AlphaZero was then able to go on to soundly defeat the world's leading chess playing program, Stockfish, in December of 2017, using only MCTS-enabled reinforcement learning without any supervised domain-specific training or knowledge of classical chess openings or end-game strategies, which Stockfish and other chess-playing algorithms include as key components of their arsenal. However, it should be noted that there is some dispute as to whether the computational requirements of AlphaZero gave it an unfair advantage, even though MCTS guided AlphaZero to search orders of magnitudes fewer tree paths than Stockfish.

While MCTS gained its fame as a search algorithm for deterministic combinatorial games, MCTS also works smoothly with the more general setting of sequential decision making under uncertainty, which can be modeled using Markov decision processes (MDPs) (Chang, Fu, Hu, and Marcus 2007; Chang, Hu, Fu, and Marcus 2013). This tutorial will introduce MCTS under this more general setting, with an objective to maximize an expected reward over a finite time horizon from an given initial starting state, i.e., the root node of a game decision tree.

Solving a sequential decision making problem using tree search is not a new idea. For example, alpha-beta pruning of game decision tree was the core of IBM's Deep Blue chess program that defeated world chess champion Kasparov in 1997 (Newborn 2012). However, both the breadth (number of feasible actions that can be taken at a state) and depth (number of decision epochs) make a complete tree search impossible for many real-world decision problems. For example, the possible sequence of game plays in a combinatorial board game can be approximated as $b^d$, with $b$ and $d$ representing the average breadth (number of legal moves) and depth (number of total moves) in a game. For chess and Go, we have $b \approx 35, d \approx 80$ and $b \approx 250, d \approx 150$ (Silver et al. 2016). Because of the many orders of magnitude differences between chess and Go, applying a pruning strategy similar to what Deep Blue used is (currently) computationally impractical for the game of Go. MCTS circumvents this challenge by *sampling* an admissible action at each decision stage and *simulating* (more often referred to as *roll-out* in the MCTS literature) to the end of the game without any branching. Averages of simulation results are then used to evaluate the game state and different actions, e.g., the probability of winning the game from the current state and the action that will lead to the highest win probability. In this process, MCTS needs to balance exploration, i.e., to expand the tree to visit more states and sample more actions, and exploitation, i.e., to obtain more accurate estimates of promising actions to identify the best action for the current state.

The remainder of this tutorial is organized to provide both an overview of MCTS and selected recent research advances in MCTS. We start in Section 2 with a description of MCTS, including its historical roots and connection to the simulation-based adaptive multi-stage sampling algorithm of Chang et al. (2005), illustrated using simple decision tree examples, where a decision tree here refers to the older OR term from decision theory and not the more recent AI/ML technique. We then describe the main components of an MCTS algorithm in the context of AlphaGo and AlphaZero. In Section 3 we focus on the role of tree selection policy in MCTS and compare the tree policy used by AlphaGo and AlphaZero with MCTS-OCBA (Li et al. 2019; Li et al. 2022), an optimal computing budget allocation (OCBA) (Chen and Lee 2011) policy adapted to MCTS. In Section 4, we illustrate MCTS using Othello, a combinatorial board game with $b \approx 10, d \approx 58$, which has only been claimed to be weakly solved as of 2023 (Takizawa 2023), and we include empirical numerical test results using MCTS-OCBA.

This tutorial makes generous use of previously published materials by some of the authors, especially the following: a recent expository journal article (Fu 2019), a 2017 INFORMS TutORial chapter (Fu 2017), a recent original research article (Li et al. 2022), and two previous WSC proceedings article (Fu 2016; Fu 2018). The October 2016 *OR/MS Today* article (Chang et al. 2016) provides more non-technical details of the history of MCTS and its connection to OR and simulation. Beyond MCTS, the recent INFORMS TutORial by (Peng et al. 2023) provides other connections between simulation optimization and AI.

## 2 OVERVIEW OF MCTS

### 2.1 Background

The term MCTS was first coined by Rémi Coulom (Coulom 2006) when he integrated a tree search algorithm using Monte Carlo roll-outs into his Go-program known as Crazy Stone. Then, Kocsis and Szepesvári (2006) introduced UCT (Upper Confidence Bound applied to trees) as a tree selection policy for tree-search algorithm for Markov Decision Process (MDP) problems, although Kocsis and Szepesvári (2006) did not use the term MCTS and instead referred to it as Monte Carlo planning. UCT was based on the upper confidence bound (UCB) policy from the multi-armed bandit (MAB) literature (Auer, Cesa-Bianchi, and Fischer 2002), and has been used in MCTS-based Go programs including AlphaGo and AlphaZero. Both Coulom (2006) and Kocsis and Szepesvári (2006) cite the main idea in Chang et al. (2005), which introduced an adaptive multi-stage simulation-based sequential decision-making algorithm for MDPs using a UCB policy, developed in 2002 (and presented at a Cornell University colloquium in the School of Operations Research and Industrial Engineering in April 2002). In fact, as noted in Kocsis and Szepesvári (2006), "When states are not likely to be encountered multiple times, our algorithm degrades to this algorithm."

### 2.2 MCTS Algorithm

A generic MCTS algorithm has four main "operators". We will use the term "operator" in the spirit of the description of MCTS itself as a "powerful policy improvement operator" by Silver et al. (2017). These four operators are the following: selection, expansion, simulation, and back propagation. Please refer to a survey of MCTS by Browne et al. (2012) for a complete description of these operators and various policies and techniques for these four operators that have been devised up to the time of writing of that survey. Also, it is more common in the MCTS literature to refer to the "simulation" operator as a "roll-out". Both terms refer to a depth first search from a leaf node using a Monte Carlo sampling policy until the end of the decision horizon or a state that requires no further evaluation. We used the term "simulation" to emphasize that roll-out is essentially a simulated sample path of an MDP. We briefly describe these four operators below and will illustrate them using a decision-tree example shortly:

- "Selection" corresponds to choosing a move at a node or an action in a decision tree in the traversal down the already grown tree to a leaf node. The selection operator is repeatedly invoked until a leaf node is reached for expansion of the tree.
- "Expansion" corresponds to the growth of decision tree from a leaf node, which corresponds to a state transitioned into after a move or action, for example, after an opponent's move in a game. It is modeled by a probability distribution that is a function of the state reached after the chosen move or action (corresponding to the transition probability in an MDP model).
- "Simulation/Evaluation" corresponds to returning the estimated value at a given node, which could correspond to the actual end of the horizon or game, or simply to a point where the current estimation may be considered sufficiently accurate so as not to require further simulation.
- "Backup" corresponds to the backwards dynamic programming algorithm employed in decision trees and MDPs.

### 2.3 Illustration of MCTS with a Simple Decision Tree

We now illustrate how a tree structure can be used to support decision making. Assume Bob has ten dollars and wants to use the ten dollars to buy a lottery ticket. There are two lottery tickets Bob is considering: 1) a PowerRich lottery ticket that would allow him to win \$100 million with a probability of 0.01 or nothing otherwise; and 2) a MegaHaul lottery ticket that would allow him to win \$1 million with a probability of 0.05 and nothing otherwise. The third option is not to buy a lottery ticket and keep the ten dollars.

Figure 1 depicts a classical decision tree to help Bob make a decision. Following the usual convention, squares represent decision nodes and circles represent outcome nodes. This is a one-period decision tree

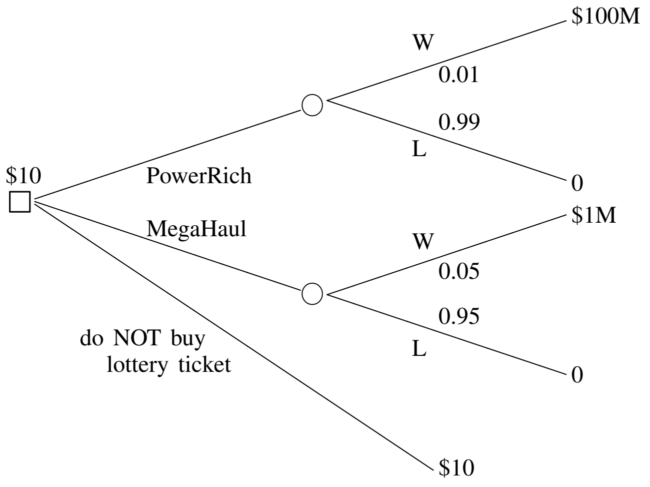

Figure 1: Decision tree for lottery choices.

with the initial state being the only decision node. The decisions are shown on the edges going from decision node to outcome node. The edges leaving the outcome node could be considered as a game move by "nature", and the corresponding probabilities are given on the edges. These edges lead to termination, because this is a one-period decision tree. The edges would lead to another decision node in a multi-period decision tree. The payoff is the money on hand at termination.

For simplicity, assume that Bob will take the decision that maximizes expected payoff, which in this case can be easily determined (buy MegaHaul). In reality, we know that human behavior generally does not follow the principle of maximizing expected value but it suffices for illustrative purposes.

However, what if Bob does not know the winning and losing probabilities of PowerRich and/or MegaHaul, or even not the payoff if he wins the lottery, or if he realizes that he can use any amount of money that he wins to buy more lottery tickets and thus there would be a sequence of decisions for him to make? When all of these complications need to be taken into account, Bob no longer knows, or at least easily, what would be the expected payoff associated with buying a PowerRich or MegaHaul lottery ticket. The decision tree now becomes the one in Figure 2, where the outcome nodes are replaced by two black boxes "BB1" and "BB2". The black boxes can be queried to return probabilistically an outcome of buying a PowerRich or MegaHaul lottery ticket, i.e., Bob can use the *simulation* operator of MCTS to simulate a realization of the random lottery payoff without knowing the probabilities of winning the lottery, or even the amount of the money he would win if it turns out to be his lucky day.

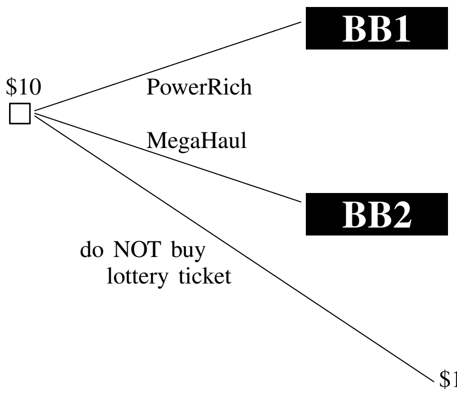

Figure 2: Decision tree for lottery choices with black boxes BB1 and BB2.

Bob's decision-making process would then involve querying these two black boxes repetitively to learn more about the expected payoff of PowerRich and MegaHaul. Each time Bob is going to run another query, Bob needs to decide if he should *select* PowerRich or MegaHaul for simulation. This is what the *selection* operator of an MCTS is for. It would need to balance the exploration and exploitation trade-off, e.g., to consider both the uncertainty in the estimated payoff for PowerRich and MegaHaul, as well as how much the expected payoff is.

Bob will then keep applying the selection operator until he uses up a simulation budget. Alternatively, instead of working with a fixed simulation budget, Bob may want to find out how many simulations it would take to achieve a fixed precision, e.g., with 99% confidence that Bob could determine which decision is the best subject to a small estimation error with an amount that he is indifferent to, e.g., a payoff different of one cent. In both cases, once all given or required simulations have finished, Bob will then compare the simulation results for PowerRich and MegaHaul to make the best decision he can using the noisy simulation estimates. This becomes the well-studied Ranking & Selection (R&S) problem in the simulation literature (Fu 2015).

This simple decision tree has just a single stage, with the three leaf nodes in the tree being terminal nodes in the decision process. If Bob wants to entertain the possibility of using any money he may win from the lottery ticket to buy more lottery tickets, he will then solve a sequential decision-making under uncertainty problem that would require an expansion of the tree to incorporate downstream decisions and outcomes as shown in Figure 3. Suppose one simulation of the black box "BB1" in the tree in Figure 2 returns an outcome of Bob winning $100M. This creates a new decision node with a state of $100M. The *expansion* operator of an MCTS then adds to this node three outcome nodes, with two of them again requiring black box simulations. Suppose Bob again chooses buy PowerRich using all of his $100M, which would be enough for 10M PowerRich tickets. The black box simulation now will return the random payoffs for all of these 10M PowrRich tickets. This outcome will then be used to update the payoff of the first decision to buy PowerRich using the *backup* operator of MCTS.

We note that the outcome of a decision in this example is assumed to be randomly generated by "nature" with (unknown) probabilities. When "nature" is the opposing player, we land at a game setting, for which MCTS is a popular algorithm for combinatorial board games like Othello and Go (Browne et al. 2012). In the game setting, the "selection" operator determines the action to evaluate from a decision node, whereas a "simulation" operator generates the opponent's subsequent move.

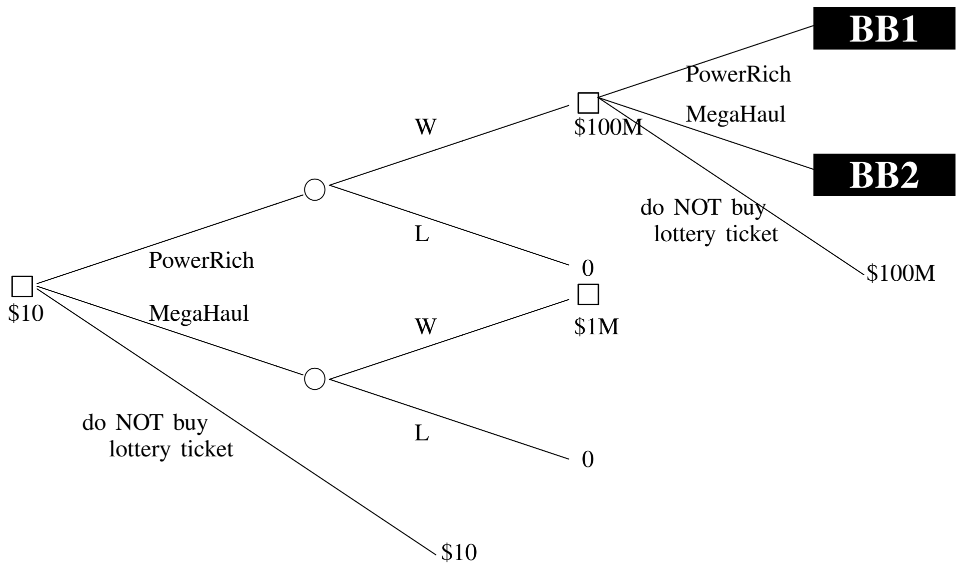

Figure 3: Expansion of the decision tree for two-stage lottery choices.

## 3 MCTS SELECTION OPERATOR

Reviewing the decision-tree example presented in the last section, we would recognize that among the four MCTS operators, the "selection" operator is the one that requires the most thought and determines how the tree would be expanded during the decision-making process. For problems with large decision space and state space, it is not possible to build a complete decision tree. Therefore, the selection operator is expected to have a major effect on the performance of MCTS for any given applications. Of course, the fidelity of the "simulation" operator is also vital to the performance of the MCTS algorithm. However, it is highly problem dependent, whereas a good "selection" operator may be applied to virtually all MCTS applications.

A widely used selection operator is UCT, which is based on UCB (Auer et al. 2002) and closely resembles the adaptive multistage sampling algorithm (AMS) for MDPs developed by Chang et al. (2005). For each possible action, UCT is a function of the current estimated value (e.g., probability of winning) plus a "fudge" factor. UCB is designed to minimize cumulative regret in an MAB setting. However, for MCTS applications such as AlphaGo, the objective is to select the action that would give the player the highest probability of winning a game, not any type of "cumulative regret" that the player may incur as the game goes on. Therefore, although UCT was used by AlphaGo, there is a misalignment between what UCT is designed for and what AlphaGo is trying to accomplish with the selection operator. In this section, we discuss two MCTS selection operators that are designed with different objectives in mind.

### 3.1 Adaptive Multistage Simulation Sampling Algorithm

Consider a finite-horizon MDP with finite state space $S$, finite action space $A$, non-negative bounded reward function $R$ such that $R : S \times A \to \mathscr{R}^+$, and transition function $P$ that maps a state and action pair to a probability distribution over state space $S$. We denote the feasible action set in state $s \in S$ by $A(s) \subset A$ and the probability of transitioning to state $s' \in S$ when taking action $a$ in state $s \in S$ by $P(s,a)(s')$. In terms of a game tree, the initial state of the MDP or root node in a decision tree corresponds to some point in a game where it is our turn to make a move. A simulation replication or sample path from this point is then a sequence of alternating moves between a player and the opponent, ultimately reaching a point where the final result is "obvious" (win or lose) or "good enough" to compare with another potential initial move, specifically if the value function is precise enough.

The AMS algorithm of Chang et al. (2005) chooses to sample in state $s$ an optimizing action in $A(s)$ according to the following:

$$\max_{a \in A(s)} \left( \hat{Q}(s,a) + \sqrt{\frac{2 \ln \bar{n}}{N_a^s}} \right), \tag{1}$$

where $N_a^s$ is the number of times action $a$ has been sampled thus far (from state $s$), $\bar{n}$ is the total number of samples thus far, and

$$\hat{Q}(s,a) = R(s,a) + \frac{1}{N_a^s} \sum_{s' \in S_a^s} \hat{V}(s'),$$

where $S_a^s$ is the set of sampled next states thus far ($|S_a^s| = N_{a,i}^s$) with respect to the distribution $P(s,a)$, and $\hat{V}$ is the value function estimate (as a function of the state). The argument in (1) is the UCB of action $a$. It combines the value ($Q$-function) of taking action $a$ at state $s$ with a bonus term that favors an action that has been less sampled. Therefore, AMS encourages sampling actions that have higher returns (larger $\hat{Q}(s,a)$) and/or are less frequently sampled (smaller $N_a^s$). Under the MAB setting, UCB has been shown to minimize the cumulative regret (Auer et al. 2002). Algorithms that focus on minimizing regret tend to discourage exploration as can be explained as follows. Suppose at some point an action was taken and the player received a small reward. To minimize regret, the player would be discouraged from taking this action again. However, the small reward could be due to the randomness in the reward distribution. Mathematically, Lai and Robbins (1985) showed that for MAB algorithms, the number of times the optimal action is taken is exponentially more than sub-optimal ones.

### 3.2 Optimal Computing Budget Allocation Algorithm

The exponential dominance of sub-optimal actions by the optimal action in AMS/UCT is optimal for minimizing cumulative regret. However, when the goal is to select the best action with high probability, as in the setting of board games such as Go, AMS/UCT may overly exploit while failing to explore sufficiently. Li et al. (2022) developed an MCTS selection operator based on a popular framework for fixed-budget R&S known as optimal computing budget allocation (OCBA) (Chen et al. 2000; Chen and Lee 2011), which aims to maximize the probability of correctly selecting the best action under a fixed sampling budget. Li et al. (2022) adapted OCBA to MCTS, and in doing so, also removed the known-and-bounded-support assumption on the reward distribution that is required by UCT.

In MCTS-OCBA, we denote by $\hat{a}^*(s)$ the action that has the highest $Q(s,a)$ value,

$$\hat{a}^*(s) = \arg\max_{a \in A(s)} \hat{Q}(s,a). \tag{2}$$

Let $\delta(s,a) = \hat{Q}(s,\hat{a}^*(s)) - \hat{Q}(s,a)$. Also, let $\sigma^2(s,a)$ be the sample variance of the simulation estimate of $Q(s,a)$, where with a slight abuse of notation, we use $Q(s,a)$ to denote the simulation samples that provide the information to calculate $\hat{Q}(s,a)$. Compared to AMS/UCT, a key difference of MCTS-OCBA is the role of $\sigma^2(s,a)$ in determining the sample size $N_a^s$, which needs to satisfy the following equations:

$$\frac{N_a^s}{N_{a'}^s} = \frac{\delta^2(s,a')/\sigma^2(s,a')}{\delta^2(s,a)/\sigma^2(s,a)}, \ \forall a,a' \in A(s), a,a' \neq \hat{a}^*(s) \tag{3}$$

$$N_{\hat{a}^*(s)}^s = \sigma^2(s,\hat{a}^*(s))\sqrt{\sum_{a\in A(s), a\neq \hat{a}^*(s)} \frac{N_a^{s2}}{\sigma^2(s,a)}}. \tag{4}$$

Equation (4) implies that OCBA samples non-optimal actions at a rate almost proportional to the sampling rate for the optimal action, which is a sharp contrast to the exponential dominance of sampling the optimal action over non-optimal actions in AMS/UCT. This increase in exploration translates to better performance of MCTS when applied to search for the best action at the root decision node as shown by Li et al. (2022).

## 4 MCTS APPLIED TO OTHELLO

### 4.1 Othello in a Nutshell

Othello is a classic two-player combinatorial board game played on an 8x8 board. It is also known as Reversi (Wikipedia 2024) to some players. Similar to Go, it is a deterministic zero-sum game with perfect information. It has been frequently used as a test case in the development of AI algorithms for game play (Browne et al. 2012). We demonstrate the differences in sampling allocation and performance of MCTS with AMS/UCT or OCBA selection operator when applied to Othello here. In Othello, the goal is to have the most tokens at the end of the game. To do this, players take turns making moves. Each move a player makes consists of placing a token in an empty space that outflanks opponent tokens, then flipping opponent tokens to your color. Outflanking opponent tokens means placing a token on the board such that a row, column or diagonal of the opponent's token are bordered at each end by a token of your color. Figure 4a shows a white token indicated by an arrow was placed strategically. This white token then join the three white tokens already on the board to outflank all black tokens in three different directions and flip them into white tokens as shown in Figure 4b. In addition, players are not allowed to skip over their own color

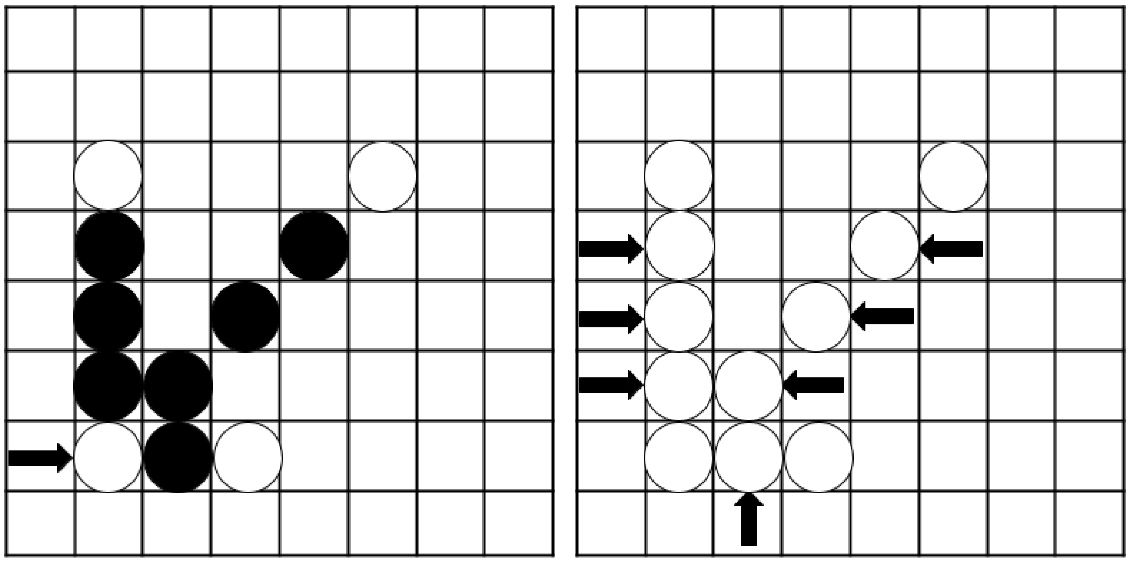
(a) A white token is placed indicated by an arrow. (b) The token placed flipped black tokens into white.

Figure 4: A strategically placed white token flipped all black tokens by outflanking them.

token to flip other tokens. In the example shown in Figure 5, the newly placed black token identified by the arrow will only flip one white token (the token marked by "1") from white to black.

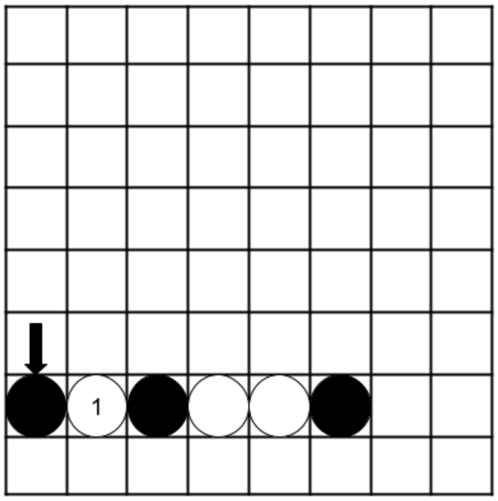

Figure 5: No skipping over own color token to flip other token.

## 4.2 Simulation Operator for Othello

We test two simulation policies: i) random play, ii) a heuristic policy. The heuristic policy builds upon Othello domain knowledge (Lee and Mahajan 1990) and uses the number of corners, X-squares (the squares directly diagonal to corners), number of moves, and potential mobility (p-mob, the total number of empty spaces next to opponent pieces) to estimate the desirability of a move using the equation below:

$$(\text{\# of your corners} - \text{\# of enemy corners}) * 400 + (\text{\# of enemy X-squares} - \text{\# of your X-squares}) * 100$$
$$+ (\text{\# of your moves})/(\text{\# of opponent moves}) * 50 + (\text{your p-mob})/(\text{enemy p-mob}) * 80.$$

We illustrate these key positions in Figure 6, where X squares are indicated with a red X. Corner squares are indicated with a red C. The potential mobility is indicated by the yellow squares (squares adjacent to the opponent token). In this case, black has a potential mobility of 13 because there are 13 yellow squares. It is worth pointing out that the heuristic simulation policy used here is a reasonable one, but certainly not the best. Neither does MCTS requires a strong simulation policy to achieve strong performance. Silver et al. (2016) described the performance improvement of using a weakly trained policy network as the simulation policy for AlphaGo. Silver et al. (2017) further explained how AlphaZero training was done with MCTS using just random play without any human knowledge and achieved superior performance for both Go and Chess.

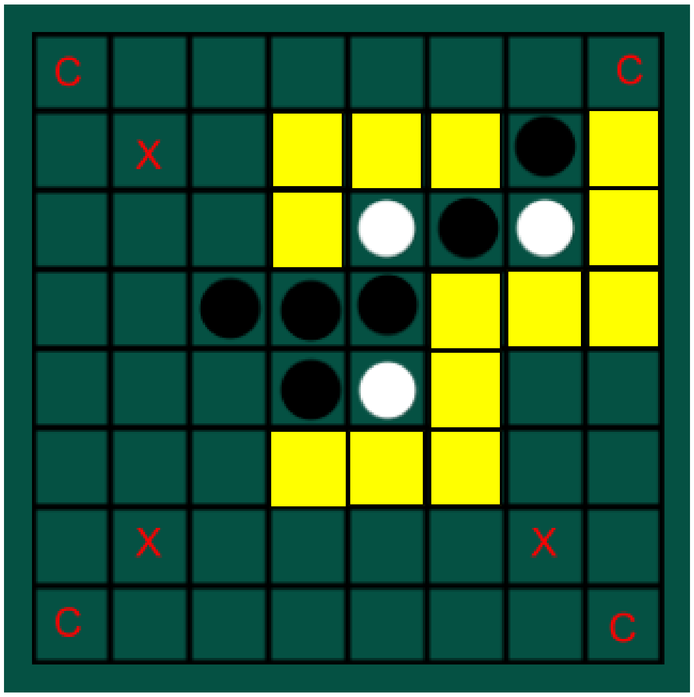

Figure 6: Key board positions considered in the heuristic simulation policy.

## 4.3 Results

We first compare how AMS/UCT and OCBA sample different moves using over 1000 different Othello positions. We ran MCTS with AMS/UCT or MCTS with OCBA to determine the move to take from these positions. Table 2 reported the percentage of simulation budget spent on the most visited five moves. Results showed that MCTS with OCBA tends to explore more while MCTS with AMS/UCT tends to spend much more budget on the most visited move at the expense of collecting more information from other promising moves.

To get a better understanding of how OCBA and AMS/UCT evaluate moves, we show a sample position with 3 legal moves in Figure 7. In the figure, X indicates black, O indicates white, and * indicates a legal move from that position. It is the turn for the player with white tokens to make a move.

Table 1: Percentage of budget spent on the top 5 most visited moves.

| Nth Most visited move | AMS/UCT | OCBA |
|---|---|---|
| Most visited move | 29.469% | 19.768% |
| 2nd most visited move | 19.917% | 14.891% |
| 3rd most visited move | 13.616% | 12.267% |
| 4th most visited move | 9.725% | 10.271% |
| 5th most visited move | 6.887% | 8.267% |

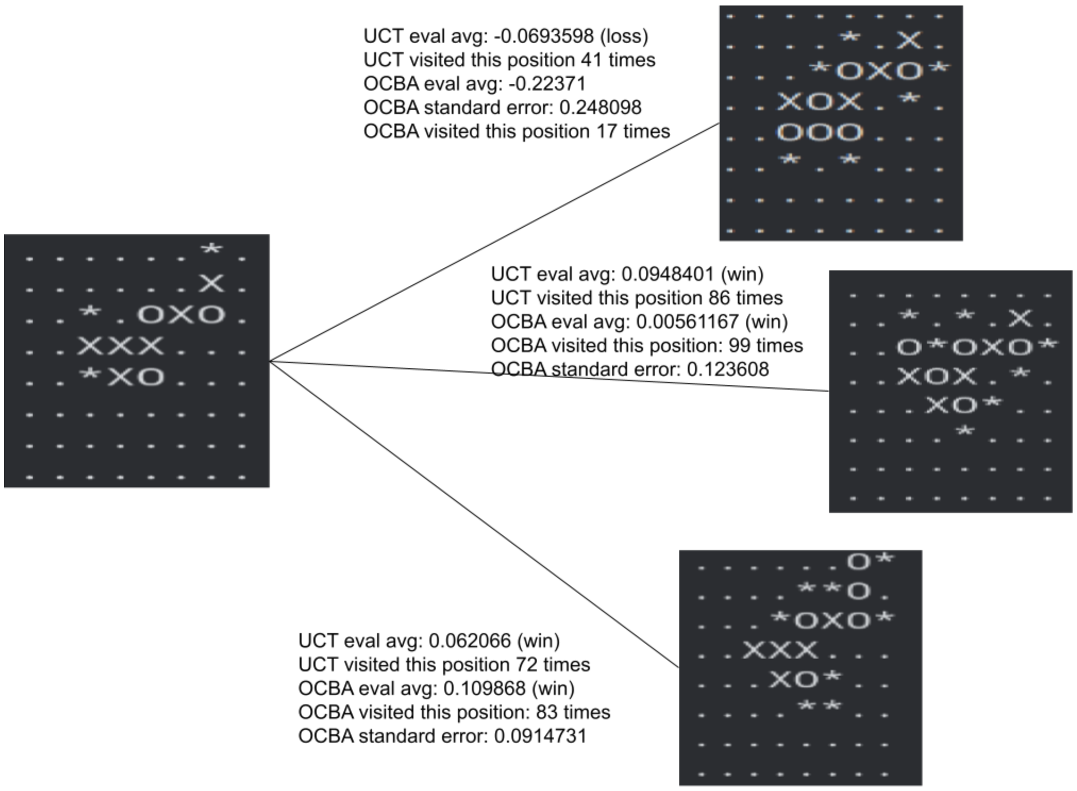
UCT eval avg: -0.0693598 (loss)
UCT visited this position 41 times
OCBA eval avg: -0.22371
OCBA standard error: 0.248098
OCBA visited this position 17 times
UCT eval avg: 0.0948401 (win)
UCT visited this position 86 times
OCBA eval avg: 0.00561167 (win)
OCBA visited this position: 99 times
OCBA standard error: 0.123608
UCT eval avg: 0.062066 (win)
UCT visited this position 72 times
OCBA eval avg: 0.109868 (win)
OCBA visited this position: 83 times
OCBA standard error: 0.0914731

Figure 7: Evaluations of moves from a sample Othello board position by MCTS with OCBA or AMS/UCT.

Table 2 reports results from playing 250 games between MCTS with OCBA and MCTS with AMS/UCT. We experimented with different number of simulations used by each algorithm to evaluate moves in each play of the game. We did not report ties in the table. We observe that MCTS with OCBA consistently outperformed MCTS with AMS/UCT with random simulation policy, delivering 60.2% wins, 3.4% ties with a simulation budget of 50 simulations, 54.0% wins, 2.2% ties with a simulation budget of 100 simulations, and 52.0% wins, 3.8% ties for a simulation budget of 500 simulations.

OCBA also has better performance than AMS/UCT when the heuristic simulation policy is used, with 60.8% wins and 2.2% ties with 50 simulations budget, 59% wins and 1.4% ties with 100 simulations budget, and 50.8% wins and 3.6% ties with 500 simulations budget. OCBA's advantage is generally larger when simulation budget is relatively small or when using heuristic simulation policy.

Table 2: Number of wins by MCTS with OCBA vs. MCTS with AMS/UCT from 250 games between (ties not reported) with different simulation budget for each position evaluation.

| (White vs Black, 250 games) | No Heuristic | Heuristic |
|---|---|---|
| OCBA vs AMS/UCT (50 simulations) | 149 - 93 | 144 - 102 |
| AMS/UCT vs OCBA (50 simulations) | 89 - 152 | 83 - 160 |
| OCBA vs AMS/UCT (100 simulations) | 142 - 102 | 149 - 96 |
| AMS/UCT vs OCBA (100 simulations) | 117 - 128 | 102 - 146 |
| OCBA vs AMS/UCT (500 simulations) | 130 - 111 | 134 - 109 |
| AMS/UCT vs OCBA (500 simulations) | 110 - 130 | 119 - 120 |

## 5 MCTS IN GOOGLE DEEPMIND'S ALPHAGO AND ALPHAZERO

Go is the most popular two-player board game in East Asia, traing its history back to China more than 2,500 years ago. It is also thought to be the oldest board game still played today. Since the size of the board is 19×19, as compared to 8×8 for a chess board, the number of board configurations for Go far exceeds that for chess, with estimates at around $10^{170}$ possibilities, exceeding even the number of atoms in the universe (https://googleblog.blogspot.nl/2016/01/alphago-machine-learning-game-go.html).

Intuitively, the objective is to have "captured" the most territory by the end of the game, which occurs when both players are unable to move or choose not to move, at which point the winner is declared as the player with the highest score, calculated according to certain rules. Unlike chess, the player with the black (dark) pieces moves first in Go, but same as chess, there is supposed to be a slight first-mover advantage (which is actually compensated by a fixed number of points decided prior to the start of the game). As a result of the far smaller number of possibilities, traditional exhaustive game tree search that works for chess is doomed to failure for Go.

The realization that traversing the entire game tree was computationally infeasible for Go meant that new approaches were required, leading to a fundamental paradigm shift, the main components being Monte Carlo sampling (or simulation of sample paths) and value function approximation, which are the basis of simulation-based approaches to solving Markov decision processes (Chang et al. 2007; Chang et al. 2013), which are also addressed by neuro-dynamic programming (Bertsekas and Tsitsiklis 1996); approximate (or adaptive) dynamic programming (Powell 2007); and reinforcement learning (Sutton and Barto 2018). However, the setting for these approaches is that of a single decision maker tackling problems involving a sequence of decision epochs with uncertain payoffs and/or transitions. The game setting adapted these frameworks by modeling the uncertain transitions – which could be viewed as the actions of "nature" – as the action of the opposing player. As a consequence, to put the game setting into the MDP setting required modeling the state transition probabilities as a distribution over the actions of the opponent. Thus, as we shall describe later, AlphaGo employs two deep neural networks: one for value function approximation and the other for policy approximation, used to sample opponent moves.

AlphaGo's two neural networks are depicted in Figure 8. The value network estimates the "value" of a given board configuration (state), i.e., the probability of winning from that position. The policy network estimates the probability distribution of moves (actions) from a given position (state).

The subscripts $\sigma$ and $\rho$ on the policy network correspond to two different networks used, using supervised learning and reinforcement learning, respectively. The subscript $\theta$ on the value network represents the parameters of the neural net. AlphaGo's two neural networks employ 12 layers with millions of connections. AlphaGo Zero has a single neural network for both the policy network and value network.

In terms of MDPs and Monte Carlo tree search, let the current board configuration (state) be denoted by $s^*$. Then we wish to find the best move (optimal action) $a^*$, which leads to board configuration (state) $s$, followed by (sampled/simulated) opponent move (action) $a$, which leads to board configuration (new

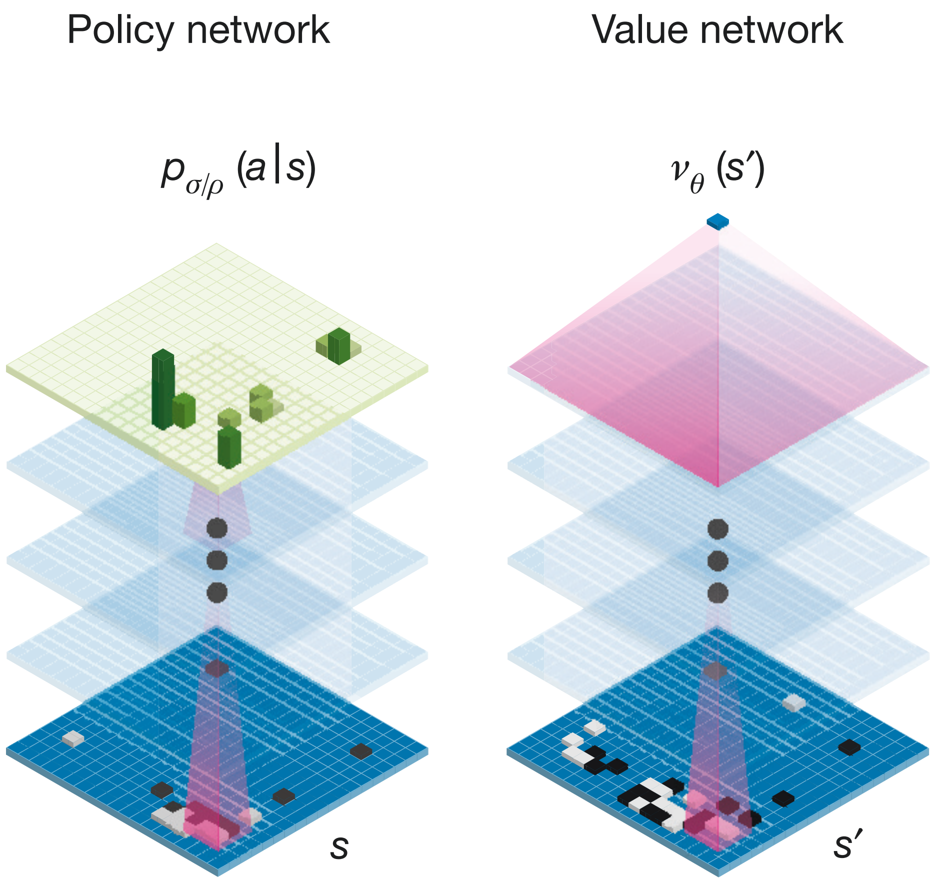

Figure 8: AlphaGo's two deep neural networks (Adapted by permission from Macmillan Publishers Ltd.: *Nature*, Figure 1b, Silver et al. 2016, copyright 2016).

"post-decision" state) $s'$, i.e., a sequence of a pair of moves can be modeled as

$$s^* \xrightarrow{a^*} s \xrightarrow{a} s',$$

using the notation consistent with Figure 8.

AlphaGo's use of MCTS is described on the 1st page of their 2016 *Nature* article in the following three excerpts (Silver et al. 2016, p.484):

> "Monte Carlo tree search (MCTS) uses Monte Carlo rollouts to estimate the value of each state in a search tree. As more simulations are executed, the search tree grows larger and the relevant values become more accurate. The policy used to select actions during search is also improved over time, by selecting children with higher values. Asymptotically, this policy converges to optimal play, and the evaluations converge to the optimal value function. The strongest current Go programs are based on MCTS...
>
> "We pass in the board position as a $19 \times 19$ image and use convolutional layers to construct a representation of the position. We use these neural networks to reduce the effective depth and breadth of the search tree: evaluating positions using a value network, and sampling actions using a policy network.

"Our program AlphaGo efficiently combines the policy and value networks with MCTS."

Figure 8 shows the corresponding $p(a|s)$ for the policy neural network that is used to simulate the opponent's moves and the value function $v(s')$ that estimates the value of a board configuration (state) using the value neural network. The latter could be used in the following way: if a state is reached where the value is known with sufficient precision, then stop there and start the backwards dynamic programming; else, simulate further by following the UCB prescription for the next move to explore.

The objective is to effectively and efficiently (through appropriate parametrization) approximate the value function approximation and represent the opposing player's moves as a probability distribution, which degenerates to a deterministic policy if the game is easily solved to completion assuming the opponent plays "optimally." However, at the core of these lies the Monte Carlo sampling of game trees or what our community would call the simulation of sample paths. Although the game is completely deterministic, unlike other games of chance such as those involving the dealing of playing cards (e.g., bridge and poker), the sheer astronomical number of possibilities precludes the use of brute-force enumeration of all possibilities, thus leading to the adoption of randomized approaches. This in itself is not new, even for games, but the Go-playing programs have transformed this from simple coin flips for search (which branch to follow) to the evaluation of a long sequence of alternating moves, with the opponent modeled by a probability distribution over possible moves in the given reached configuration. In the framework of MDPs, Go can be viewed as a finite-horizon problem where the objective is to maximize an expected reward (the territorial advantage) or the probability of victory, where the randomness or uncertainty comes from two sources: the state transitions, which depend on the opposing player's move, and the single-stage reward, which could be territorial (actual perceived gain) or strategic (an increase or decrease in the probability of winning), as also reflected in the estimated value after the opponent's move.

For chess, AlphaZero uses MCTS to search 80K positions per second, which sounds like a lot. However, Stockfish, considered in 2017 by many to be the best computer chess-playing program (way better than any human that has ever lived) considers about 70M positions per second, i.e., nearly three orders of magnitude more, and yet AlphaZero did not lose a single game to Stockfish in their December 2017 100-game match (28 wins, 72 draws). "Instead of an alpha-beta search with domain-specific enhancements, AlphaZero uses a general-purpose Monte-Carlo tree search (MCTS) algorithm." (Silver et al. 2017, p.3)

## 6 CONCLUSIONS

MCTS provides the foundation for training the DNNs of AlphaGo, as well as the single DNN engine for its successor AlphaZero. The roots of MCTS are contained in the more general AMS algorithm for MDPs published by Chang et al. (2005) in *Operations Research*, where dynamic programming (backward induction) is used to estimate the value function in an MDP. A game tree can be viewed as a decision tree (simplified MDP) with the opponent in place of "nature" in the model. In AlphaZero, both the policy and value networks are contained in a single DNN, which makes it suitable to play against itself when the net is trained for playing both sides of the game. Thus, an important takeaway message to be conveyed is that OR played an unheralded role, as the MCTS algorithm is based on the UCB algorithm for MDPs, with dynamic programming (backward induction) used to calculate/estimate the value function in an MDP (game tree, viewed as a decision tree with the opponent in place of "nature" in the model).

We have also shown that research originating from the simulation literature, and specifically the R&S area, has offered valuable ideas to improve the algorithmic performance of MCTS, which serves to complement the ML/AI community's focus on the DNN models used as the policy network and value network, or the single DNN trained using MCTS generated data as in AlphaZero. The OCBA selection operator discussed in Section 3.2 provides such an example, with the improved performance of MCTS demonstrated using the game of Othello. OCBA also removes the bounded and known support assumption for the reward distribution, enlarging the applicability of MCTS to more problems. Another important insight from OCBA is variance information is as important as estimating the average performance of a move.

In terms of future research, an obvious direction in AI would be to consider MCTS for games that are inherently stochastic. Both Go and chess are deterministic games, whereas games such as bridge and poker involve uncertainty both in terms of future outcomes (how an opponent plays) and imperfect information (not knowing what cards the opponents hold). The latter is what makes both bridge and poker more challenging; otherwise, they would be relatively easier to solve. Thus, in these settings, the inherent uncertainty dominates. Even so, in January 2017, an AI system called Libratus, developed by Tuomas Sandholm and his PhD student Noam Brown at CMU, bested four of the world's top human players, apparently deriving strategies using more traditional AI approaches and not MCTS, so an interesting question is whether or not using MCTS could improve the performance. As mentioned in the introduction, other connections between simulation optimization and AI are explored in the recent INFORMS TutORial by (Peng et al. 2023).

## ACKNOWLEDGMENTS

This work was supported in part by the Air Force of Scientific Research (AFOSR) under Grant FA95502010211; by the National Science Foundation (NSF) under Grants CMMI-2146067, IIS-2123684 and SBE-2228603; and by UChicago/Argonne LLC under Grant 1F-60250.

## REFERENCES

Auer, P., N. Cesa-Bianchi, and P. Fischer. 2002. "Finite-time Analysis of the Multiarmed Bandit Problem". *Machine Learning* 47:235–256.

Bertsekas, D. and J. N. Tsitsiklis. 1996. *Neuro-Dynamic Programming*. Athena Scientific.

Browne, C. B., E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. 2012. "A Survey of Monte Carlo Tree Search Methods". *IEEE Transactions on Computational Intelligence and AI in Games* 4(1):1–43.

Chang, H. S., M. C. Fu, J. Hu, and S. I. Marcus. 2005. "An Adaptive Sampling Algorithm for Solving Markov Decision Processes". *Operations Research* 53(1):126–139.

Chang, H. S., M. C. Fu, J. Hu, and S. I. Marcus. 2007. *Simulation-Based Algorithms for Markov Decision Processes*. Springer.

Chang, H. S., M. C. Fu, J. Hu, and S. I. Marcus. 2016. "Google DeepMind's AlphaGo: Operations Research's Unheralded Role in the Path-breaking Achievement.". *OR/MS Today* 43(5):24–30.

Chang, H. S., J. Hu, M. C. Fu, and S. I. Marcus. 2013. *Simulation-based Algorithms for Markov Decision Processes*. 2nd ed. Springer.

Chen, C.-H. and L. H. Lee. 2011. *Stochastic Simulation Optimization: An Optimal Computing Budget Allocation*. World Scientific.

Chen, C.-H., J. Lin, E. Yücesan, and S. E. Chick. 2000. "Simulation Budget Allocation for Further Enhancing the Efficiency of Ordinal Optimization". *Discrete Event Dynamic Systems: Theory and Applications* 10:251–270.

Coulom, R. 2006. "Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search". In *International Conference on Computers and Games*, 72–83. Springer.

Fu, M. C. (Ed.) 2015. *Handbook of Simulation Optimization*. Springer.

Fu, M. C. 2016. "AlphaGo and Monte Carlo Tree Search: The Simulation Optimization Perspective". In *2016 Winter Simulation Conference (WSC)*, 659–670 https://doi.org/https://doi.org/10.1109/WSC.2016.7822130.

Fu, M. C. 2017. "Markov Decision Processes, AlphaGo, and Monte Carlo Tree Search: Back to the Future". In *INFORMS TutORials in Operations Research*, edited by R. Batta and J. Peng, 68–88. INFORMS.

Fu, M. C. 2018. "Monte Carlo Tree Search: A Tutorial". In *2018 Winter Simulation Conference (WSC)*, 222–236 https://doi.org/https://doi.org/10.1109/WSC.2018.8632344.

Fu, M. C. 2019. "Simulation-based Algorithms for Markov Decision Processes: Monte Carlo Tree Search from AlphaGo to AlphaZero". *Asia-Pacific Journal of Operational Research* 36(06):1940009.

Kocsis, L. and C. Szepesvári. 2006. "Bandit based Monte-Carlo Planning". In *European Conference on Machine Learning*, 282–293. Springer.

Lai, T. L. and H. Robbins. 1985. "Asymptotically Efficient Adaptive Allocation Rules". *Advances in Applied Mathematics* 6(1):4–22.

Lee, K. and S. Mahajan. 1990. "The Development of a World Class Othello Program". *Artificial Intelligence* 43:21–35.

Li, Y., M. Fu, and J. Xu. 2019. “Monte Carlo Tree Search with Optimal Computing Budget Allocation”. In *2019 IEEE 58th Conference on Decision and Control (CDC)*, 6332–6337. IEEE.

Li, Y., M. C. Fu, and J. Xu. 2022. “An Optimal Computing Budget Allocation Tree Policy for Monte Carlo Tree Search”. *IEEE Transactions on Automatic Control* 67(6):2685–2699.

Newborn, M. 2012. *Kasparov versus Deep Blue: Computer Chess Comes of Age*. Springer Science & Business Media.

Peng, Y., C.-H. Chen, and M. C. Fu. 2023. “Simulation Optimization in the New Era of AI”. In *INFORMS TutORials in Operations Research*, 82–108. INFORMS.

Powell, W. B. 2007. *Approximate Dynamic Programming: Solving the Curses of Dimensionality*, Volume 703. John Wiley & Sons.

Silver, D., A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, and M. Lanctot. 2016. “Mastering the Game of Go with Deep Neural Networks and Tree Search”. *Nature* 529(7587):484–489.

Silver, D., J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, and A. Bolton. 2017. “Mastering the Game of Go without Human Knowledge”. *Nature* 550(7676):354–359.

Silvestri, Allie 2017. “AlphaGo Movie”. Accessed 16$^{\text{th}}$ April 2024.

Sutton, R. S. and A. G. Barto. 2018. *Reinforcement Learning: An Introduction*. MIT press.

Takizawa, H. 2023. “Othello is Solved”. *arXiv preprint arXiv:2310.19387*.

Wikipedia 2024. “Reversi”. https://en.wikipedia.org/wiki/Reversi, accessed 05.03.2024.

## AUTHOR BIOGRAPHIES

**MICHAEL C. FU** holds the Smith Chair of Management Science in the Robert H. Smith School of Business, with a joint appointment in the Institute for Systems Research and affiliate faculty appointment in the Department of Electrical and Computer Engineering, A. James Clark School of Engineering, all at the University of Maryland, College Park, where he has been since 1989. His research interests include stochastic gradient estimation, simulation optimization, and applied probability, with applications to manufacturing, supply chain management, and healthcare. He has a Ph.D. in applied math from Harvard and degrees in math and EECS from MIT. He is the co-author of the books, *Conditional Monte Carlo: Gradient Estimation and Optimization Applications*, which received the INFORMS Simulation Society’s 1998 Outstanding Publication Award, and *Simulation-Based Algorithms for Markov Decision Processes*, and has also edited/co-edited four volumes: *Perspectives in Operations Research*, *Advances in Mathematical Finance*, *Encyclopedia of Operations Research and Management Science* (3rd edition), and *Handbook of Simulation Optimization*. He attended his first WSC in 1988 and served as Program Chair for the 2011 WSC. He received the INFORMS Simulation Society Distinguished Service Award in 2018, the INFORMS Saul Gass Expository Writing Award in 2021, and the INFORMS George E. Kimball Medal in 2022. He served as Program Director for the NSF Operations Research Program, Stochastic Models and Simulation Department Editor for *Management Science*, and Simulation Area Editor for *Operations Research*. He is a Fellow of INFORMS and IEEE. His e-mail address is mfu@umd.edu, and his Web page is https://www.rhsmith.umd.edu/directory/michael-fu.

**DANIEL QIU** is a student in the School of Computer Science, Carnegie Mellon University. His research interests are artificial intelligence and machine learning. His research interests include game theory and stochastic simulation and optimization. He has competed and placed highly in multiple Computer Science and AI competitions such as first place at MIT’s Battlecode competition. His email address is scienceqiu@gmail.com.

**JIE XU** is an associate professor in the Department of Systems Engineering and Operations Research at George Mason University. He received the M.S. degree in computer science from the State University of New York, Buffalo, in 2004, and the Ph.D. degree in industrial engineering and management sciences from Northwestern University, Evanston, IL, in 2009. His research interests are data analytics, dynamic data driven application systems, stochastic simulation and optimization, with applications in cloud computing, energy systems, electric vehicles, health care, manufacturing, and transportation. His email address is jxu13@gmu.edu and his website is https://mason.gmu.edu/~jxu13/.