*Proceedings of the 2016 Winter Simulation Conference*
*T. M. K. Roeder, P. I. Frazier, R. Szechtman, E. Zhou, T. Huschka, and S. E. Chick, eds.*

# ALPHAGO AND MONTE CARLO TREE SEARCH: THE SIMULATION OPTIMIZATION PERSPECTIVE

Michael C. Fu

Robert H. Smith School of Business
Institute for Systems Research
University of Maryland
College Park, MD 20742, USA

## ABSTRACT

In March of 2016, Google DeepMind's AlphaGo, a computer Go-playing program, defeated the reigning human world champion Go player, 4-1, a feat far more impressive than previous victories by computer programs in chess (IBM's Deep Blue) and Jeopardy (IBM's Watson). The main engine behind the program combines machine learning approaches with a technique called Monte Carlo tree search. Current versions of Monte Carlo tree search used in Go-playing algorithms are based on a version developed for games that traces its roots back to the adaptive multi-stage sampling simulation optimization algorithm for estimating value functions in finite-horizon Markov decision processes (MDPs) introduced by Chang et al. (2005), which was the first use of Upper Confidence Bounds (UCBs) for Monte Carlo simulation-based solution of MDPs. We review the main ideas in UCB-based Monte Carlo tree search by connecting it to simulation optimization through the use of two simple examples: decision trees and tic-tac-toe.

## 1 INTRODUCTION

Go (weiqi in Mandarin pinyin) is a two-player board game tracing its origins to China more than 2,500 years ago. Since the size of the board is 19 by 19, as compared with 8 by 8 for a chess board, the number of board configurations for Go far exceeds that for chess, with estimates at around $10^{170}$ possibilities, putting it a googol (no pun intended) times beyond that of chess and exceeding the number of atoms in the universe (Google blog Jan.27, 2016 article published as https://googleblog.blogspot.nl/2016/01/alphago-machine-learning-game-go.html). Without trying to explain the rules of Go, the objective is to have "captured" the most territory by the end of the game, which occurs when neither player can move or wishes not to move. In short, the winner is the player with the highest score, which is calculated according to certain rules. Of course, a player can also resign prior to the end of the game.

Perhaps the closest game in the West to Go is Othello, which coincidentally (or maybe not) is played on an 8 by 8 board, the same size as chess (and checkers). The number of legal positions is estimated at less than $10^{28}$, which is nearly a googol and a half times less than the estimated number of possible Go board positions. As a result of this far more manageable number, traditional exhaustive game tree search can in principle handle the game of Othello, so that brute-force programs with enough computing power will easily beat any human. Of course, more intelligent programs can get by with far less computing (so that they can be implemented on a laptop or smartphone), but the point is that complete solvability is within the realm of computational tractability given today's available computating power, whereas such an approach is doomed to failure for the game of Go, merely due to the 19 by 19 size of the board. More importantly for our purposes, all of the Othello computer programs still incorporate traditional game tree search, which could include heuristic procedures such as genetic algorithms and other evolutionary approaches that could lead to for example pruning of the tree, but they do not incorporate sampling or simulation to generate the game tree itself.

Similarly, IBM Deep Blue's victory (by 3 1/2 to 2 1/2) over the chess world champion Garry Kasparov in 1997 was more of an example of sheer computational power than true artificial intelligence, as reflected by the program being referred to as "the primitive brute force-based Deep Blue" in the current Wikipedia account of the match. Again, traditional game tree search was employed, which becomes impractical for Go, as alluded to earlier. The realization that traversing the entire game tree was computationally infeasible for Go meant that computer science artificial intelligence (AI) researchers had to seek new approaches, leading to a fundamental paradigm shift. The main components of this new paradigm were 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. 2007/2013, Gosavi 2003), also studied under the following names:

- neuro-dynamic programming (Bertsekas and Tsitsiklis 1996);
- approximate dynamic programming (Powell 2010);
- adaptive dynamic programming (Werbos et al.);
- reinforcement learning (Sutton and Barto 1998, Gosavi 2003).

However, the setting for all of 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 adopted these frameworks by modeling the uncertain transitions — which could be viewed as the actions of "nature" (uncertain outcomes leading to a state transition for a given action by the decision maker) — as the action of the opposing player. Thus, to put the game setting into the MDP setting required modeling the state transition probabilities as a distribution over the actions of the opponent. Note that AlphaGo also uses a model-free form of reinforcement learning (see below) called $Q$-learning for training its neural networks.

Google DeepMind is a British AI company founded in 2010 and acquired by Google in 2014 for \$500M (Wikipedia). The company had built its reputation on the use of "deep" neural networks (trained by $Q$-learning) for AI applications, including video games. According to DeepMind's Web site, AlphaGo's neural networks employ 12 layers with millions of connections. In September of 2015, Google DeepMind's AlphaGo became the first computer Go-playing program to defeat a professional Go player, besting the European champion Fan Hui five games to none. In March of 2016 in Seoul, Korea, AlphaGo defeated the reigning human world champion Go player, Lee Sedol, winning four games to one. The main engine behind AlphaGo combines the machine learning techniques of deep neural networks and reinforcement learning with an approach called Monte Carlo tree search, a term coined by Rémi Coulom in his 2006 paper. Current versions of Monte Carlo tree search used in Go-playing algorithms are based on a version developed for games called UCT (Upper Confidence Bound 1 applied to trees), proposed by Kocsis and Szepesvári (2006), which addresses the well-known exploration-exploitation tradeoff that arises in multi-armed bandit problems by using upper confidence bounds (UCBs), a concept introduced to the machine learning community by Auer, Cesa-Bianchi, and Fischer (2002). We review the main ideas behind UCBs and UCT and show how UCT traces its roots back to the adaptive multi-stage sampling simulation optimization algorithm for estimating the value function in finite-horizon Markov decision processes (MDPs) introduced by Chang et al. (2005), which was the first to use UCBs for Monte Carlo simulation-based solution of MDPs.

Specifically, we illustrate the transition from the simulation-based MDP setting to the game setting through two simple examples: decision trees, and the game of tic-tac-toe. We finish by demonstrating the connection between the adaptive multi-stage sampling algorithm of Chang et al. (2005) and the UCT MCTS algorithm that underlies all current Go-playing programs, including AlphaGo; see also Chang et al. (2016). Along the way, we show how the MDP setting connects naturally to the game tree setting through the use of post-decision state variables promoted by Powell (2010) in his book. Depending on the context (decision tree, MDP, game), some equivalent classes of terms used throughout the paper are as follows: decision, action, move; outcome, state transition, opponent's move; position, state, board configuration.

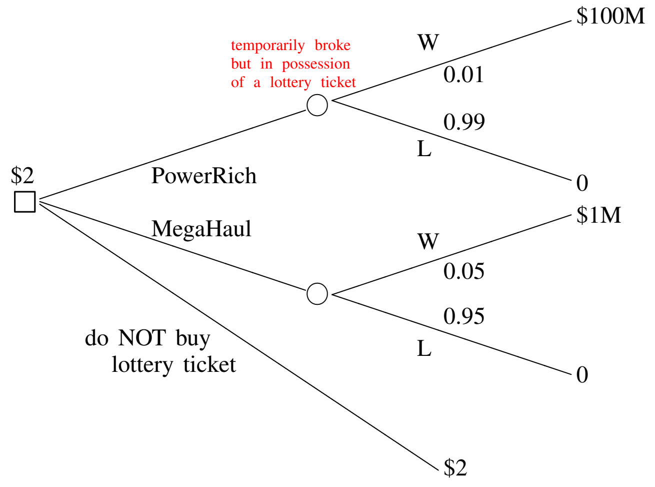

Figure 1: Decision tree for lottery choices.

## 2 SIMULATION OPTIMIZATION AND GAMES

### 2.1 Example: A Simple Decision Tree

We begin with a simple decision problem. Assume you have $2 in your pocket, and you are faced with the following three choices: 1) buy a PowerRich lottery ticket (win $100M w.p. 0.01; nothing otherwise); 2) buy a MegaHaul lottery ticket (win $1M w.p. 0.05; nothing otherwise); 3) do not buy a lottery ticket. The expected value of the three choices are given respectively by $1M, $50K, and $2.

In Figure 1, the problem is represented in the form of a decision tree, following the usual convention where squares represent decision nodes and circles represent outcome nodes. In this one-period decision tree, the initial "state" is shown at the only decision node, and the decisions are shown on the arcs going from decision node to outcome node. The outcomes and their corresponding probabilities are given on the arcs from the outcome nodes to termination (or to another decision node in a multi-period decision tree). Again, in this one-period example, the payoff is the value of the final "state" reached.

If the objective is to have the most money, which is the optimal decision?

In an informal and decidedly unscientific study at Cornell in a master's level course (April 25, 2016), this set of choices was posed. Surprisingly, 2 out of about 30 students opted for the not-playing-at-all option, a few opted for the lower payoff lottery, and the remaining vast majority (including the famous instructor) chose the first option, which had the highest expectation, an objective commonly assumed. When asked for an explanation as to why the non-playing option was chosen, one student said he/she did not believe in gambling; the other said that he/she would rather have the guarantee of the money in hand (even if only $2). For the choice of the lower payoff (and expected value) lottery, the consensus was that the lower amount was worth the significantly higher probability of winning.

This example highlighted the importance of objectives (also of concern for AlphaGo). Aside from the obvious expected value maximization, here are some that could be interpreted from the Cornell students' answers to justify their choices:

- Maximize the probability of having some money for lunch (class ended just before lunch).
- Maximize the probability of not being broke (tuition was due in a few days).
- Maximize the probability of becoming a millionaire (at least before taxes).
- Maximize a quantile.

It is also well known that human behavior does not follow the principle of maximizing expected value. We recall that a probability is also an expectation (of the indicator function of the event of interest), so the

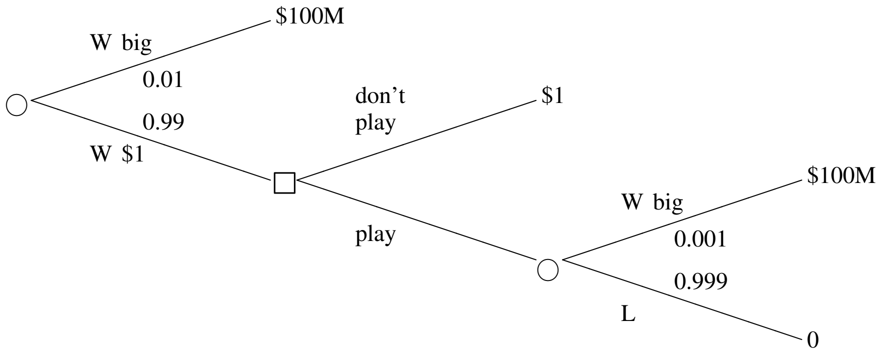

Figure 2: Decision tree for PowerRich lottery expanded once.

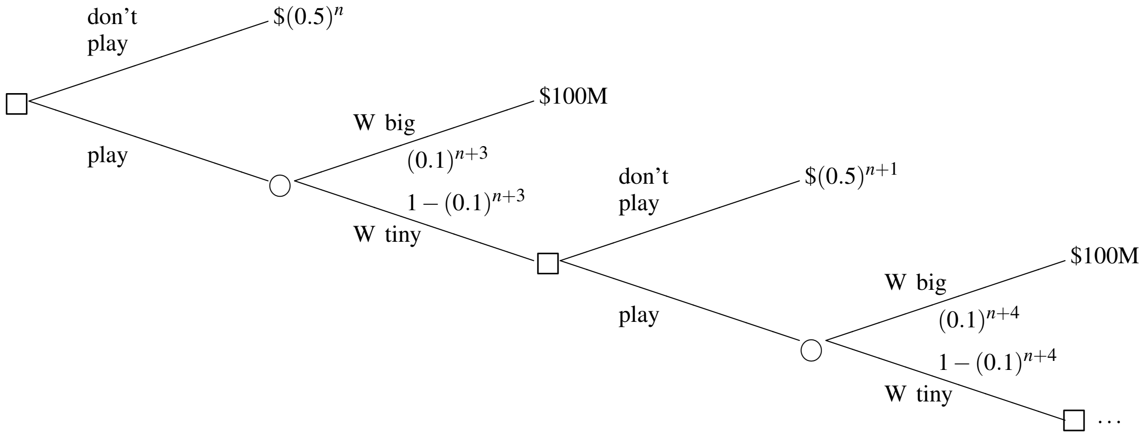

Figure 3: Decision tree for PowerRich lottery ad infinitum.

first three bullets fall into the same category, as far as this principle goes. Utility functions are often used to try and capture individual risk preferences, e.g., by converting money into some other units in a nonlinear fashion. Other approaches to modeling risk include the prospect theory of Kahneman and Tversky (1979), which demonstrated that humans often distort their probabilities, differentiate between gains and losses, and often anchor their decisions. Recent work (Prashanth et al. 2016) has applied cumulative prospect theory to MDPs.

We return to the simple decision tree example. Whatever the objective and whether or not we incorporate utility functions and distorted probabilities into the problem, it still remains easy to solve, because there are only two real choices to evaluate (the third is trivial), and these have outcomes that are assumed known with known probabilities. However, we now ask the question(s): What IF ...

- the probabilities are unknown?
- the outcomes are unknown?
- the terminal nodes keep going? (additional sequence(s) of action – outcome, etc.)
- "nature" is an opposing player?

By investigating each one of these in turn, the combination gives the essence of how Go-playing programs work. In addition, it could also be the case in many decision settings that the outcome structure itself is unknown, i.e., both the number of branches, which could be uncountable, and its associated probability distribution are unknown. This is generally not the case in a game setting, where the moves are known, but the probability distribution over the opponent's moves is not.

We begin with the first setting, where we assume that the probabilities are unknown but can be sampled from a black box simulator, i.e., there is a simulator for each outcome node. Then the problem essentially reduces to estimating the unknown probabilities, in this case sampling from Bernoulli distributions with known rewards. Initially, let's eliminate the MegaHaul lottery to simplify things even further. In this case, if we have a fixed simulation budget, we just simulate the one outcome node until the simulation budget is exhausted, because the other choice (do nothing) is known exactly. Conversely, we could pose the problem in terms of how many simulations it would take to guarantee some objective, e.g., with 99% confidence that we could determine whether or not the lottery has a higher expected value than doing nothing ($10); again there is no allocation as to where to simulate but just when to stop simulating. However, with the MegaMaul reinstated, there is now a choice as to which black box to simulate, or for a fixed budget how to allocate the simulation budget between the two simulators. Thus, we find ourselves somewhat in the domain of ranking and selection, but where the samples arise in a slightly different way.

The next twist is where the outcome values (rewards) themselves are also unknown, but again can be sampled from say the same black box simulator that produces the outcome (win or lose). Again, depending on the objective there could be many different approaches to determine the best way to allocate simulation replications. Moving to the next variation, the uncertain outcome value (reward) could also be the result of subsequent stages and uncertain outcomes. For example, in Figure 2, the total loss is replaced with a "win" of $1, for which there is the choice of stopping play or trying again, with the same jackpot but the probability decreased by an order of magnitude. Now the problem can be viewed as an optimal stopping problem, where again the question of where to allocate a simulate budget come into play at either of the two stages or if the MegaMaul option is added back into the mix. This can be repeated ad infinitum to make it an infinite horizon problem, as in Figure 3.

The final twist, the random outcomes being replaced by an opposing player, will be illustrated in the next example of tic-tac-toe. But first let's introduce the notion of a post-decision state using this example. The usual notion of state would have the decision maker with $2 at the beginning, and then after the decision, an outcome is realized to be in one of four different states: broke, same, millionaire, or $100M. However, in the case of two of the actions, buying either one of the lottery tickets, there is a post-decision state in which the decision maker is broke but holding a lottery ticket, as indicated in red in Figure 1 above the outcome node. In a game, that is the position that is reached after you make your move and before your opponent makes his/hers. In terms of your decision making, it really is irrelevant, because what matters is what the position of the game is *after* the opponent moves, because that is the state you will face. However, in order to employ Monte Carlo tree search, an essential ingredient is being able to model the probability distribution over opponent moves, which are expressed as a function of this post-decision state.

## 2.2 Example: Tic-Tac-Toe

Presumably, everyone is familiar with the game of tic-tac-toe, but just to quickly summarize in case there is a reader who has not played the game before. It is a two-player game on a 3 x 3 grid, in which the objective is to get 3 marks in a row, and players alternate between the "X" player who goes first and the "O" player. If each player follows an optimal policy, then the game should always end in a tie (cat's game). In theory there are something like 255K possible configurations that can be reached, of which only an order of magnitude less of these are unique after accounting for symmetries (rotational, etc.), and only 765 of these are essentially different in terms of actual moves for both sides. The optimal strategy for either side can be easily described in words in a short paragraph, and a computer program to play optimally requires under a hundred lines of code in most programming languages.

Assuming the primary objective (for both players) is to win and the secondary objective is to avoid losing, the following "greedy" policy is optimal (for both players):

*If a win move is available, then take it; else if a block move is available, then take it.*

(A win move is defined as a move that gives three in a row for the player who makes the move; a block move is defined here as a move that is placed in a row where the opposing player already has two moves,

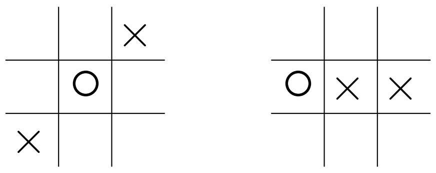

Figure 4: Two tic-tac-toe board configurations to be considered.

thus preventing three in a row.) This leaves still the numerous situations where neither a win move nor a block move is available. In traditional game tree search, these would be enumerated (although as we noted previously, it is easier to implement using further if–then logic). We instead illustrate the Monte Carlo tree search approach, which samples the opposing moves rather than enumerating them.

If going first (as X), there are three unique first moves to consider – corner, side, middle. If going second (as O), the possible moves depend of course on what X has selected. If the middle was chosen, then there are two unique moves available (corner or non-corner side); if a side (corner or non-corner) was chosen, then there are five uniques moves (but different set of five for the two choices) available. However, even though the game has a relatively small number of outcomes compared to most other games (even checkers), enumerating them all is still quite a mess for illustration purposes, so we simplify further by viewing two different game situations that already have preliminary moves.

Assume henceforth that we are the "O" player. We begin by illustrating with two games that have already seen three moves, 2 by our opponent and 1 by us, so it is our turn to make a move. The two different board configurations are shown in Figure 4. For the first game, by symmetry there are just two unique moves for us to consider: corner or (non-corner) side. In this particular situation, following the optimal policy above leads to an easy decision: corner move leads to a loss, and non-corner move leads to a draw; there is a unique "sample path" in both cases so no need to simulate/sample. The trivial game tree and decision tree are given in Figures 5 and 6, respectively, with the game tree that allows non-optimal moves shown in Figure 7.

Now let's consider the other more interesting game configuration, the righthand side of Figure 4, where there are three unique moves available to us, and without loss of generality, they can be the set of moves on the top row. We need to evaluate the "value" of each of the three actions to determine which one to select. Turns out that going in the upper left corner leads to a draw, whereas the upper middle and upper right corner lead to 5 possible unique moves for the opponent. The way the Monte Carlo tree branching would work is depicted in Figure 8, where two computational decisions would need to be made:

- How many times should each possible move (action) be simulated?
- How far down should the simulation go (to the very end or stop short at some point)?

Note that these questions have been posed in a general framework, with tic-tac-toe merely illustrating how they would arise in a simple example.

## 3 ADAPTIVE MULTISTAGE SAMPLING (AMS) ALGORITHM

In this section, we relate MCTS to the adaptive multistage sampling (AMS) algorithm in Chang et al. (2005). Consider a finite horizon MDP with finite state space $S$, finite action space $A$, nonnegative 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 $X$. We denote the 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 us and our 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.

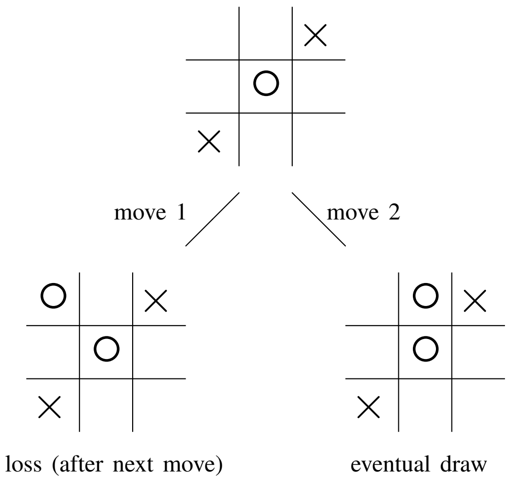

Figure 5: Game tree for 1st tic-tac-toe board configuration (assuming “greedy optimal” play). Note that if opponent’s moves were randomized, we would have an excellent chance of winning!

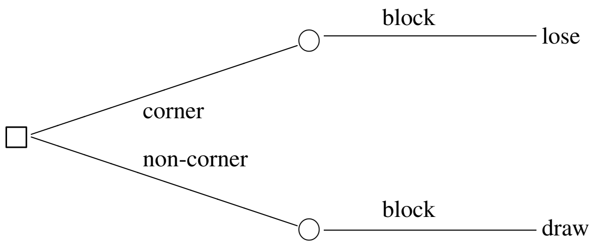

Figure 6: Decision tree for 1st tic-tac-toe board configuration of Figure 4.

Some questions/answers in a nutshell:

- How does AMS work?
  (1) UCB: selecting our (decision maker’s) actions to simulate throughout a sample path.
  (2) simulation/sampling: generating next state transitions (nature’s “actions”).
- How does MCTS work?
  (1) how to select our moves to follow in a game tree.
  (2) how to simulate opponent’s moves.
- How does MCTS fit into AlphaGo?
  (1) UCB: selecting our next moves in a simulated game tree path.
  (2) simulation/sampling: generating opponent’s next moves.

The adaptive multi-stage sampling (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

$$\hat{Q}(s,a)=R(s,a)+\frac{1}{N_a^s}\sum_{s'\in S_a^s}\hat{V}(s'), \tag{2}$$

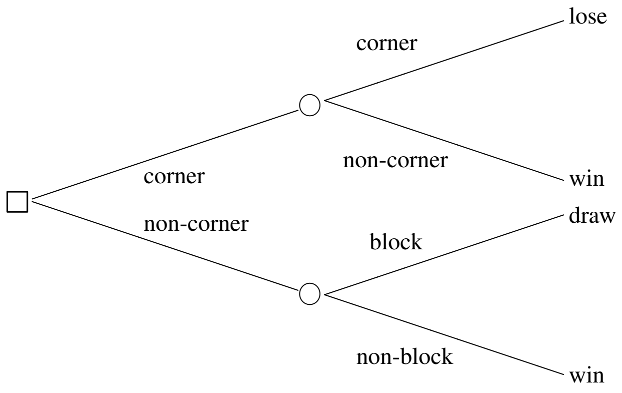

Figure 7: Decision tree for 1st tic-tac-toe board configuration with non-optimal greedy moves.

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 upper confidence bound (UCB). As described in Chang et al. (2005), AMS "approximates the optimal value to break the well-known *curse of dimensionality* in solving finite horizon Markov decision processes (MDPs). The algorithm is aimed at solving MDPs with large state spaces and relatively smaller action spaces. The approximate value computed by the algorithm not only converges to the true optimal value but also does so in an 'efficient' way. The algorithm adaptively chooses which action to sample as the sampling process proceeds, and the estimate produced by the algorithm is asymptotically unbiased."

The generation of Monte Carlo trees is also the centerpiece in the American-style option pricing algorithm of Broadie and Glasserman (1996), with the primary difference between AMS and their algorithm being that the option pricing setting is an optimal stopping problem similar to the extended multi-stage decision tree example considered earlier, so there is no need to choose which action to simulate, because there are only two actions – stop or continue – at each decision epoch, where further simulation is only required in the latter case. On the other hand, the general MDP setting gives rise to the simulation allocation problem, for which Chang et al. (2005) were the first to propose using UCB ideas from multi-armed bandit problems to balance exploitation and exploration.

We now return to the tic-tac-toe example to illustrate these concepts.

In MDP notation, we will model the state as an 9-tuple corresponding to the 9 locations, starting from upper left to bottom right, where 0 will correspond to blank, 1 = X, and 2 = O; thus, (0,0,1,0,2,0,1,0,0) and (0,0,0,2,1,1,0,0,0) correspond to the states of the two board configurations of Figure 4. Actions will simply be represented by the 9 locations, with the feasible action space the obvious remainder set of the components of the space that are still 0, i.e., $\{1,2,4,6,8,9\}$ and $\{1,2,3,7,8,9\}$ for the two board configurations of Figure 4. This is not necessarily the best state representation (e.g., if we're trying to detect certain structure or symmetries). If the learning algorithm is working how we'd like it to work ideally, it should converge to the degenerate distribution that chooses with probability one the optimal action, which can be easily determined in this simple game.

Implementations of Monte Carlo tree search face a basic tradeoff "between the *exploitation* of deep variants after moves with high average win rate and the *exploration* of moves with few simulations. The first formula for balancing exploitation and exploration in games, called UCT (Upper Confidence Bound 1 applied to trees), was introduced by Levente Kocsis and Csaba Szepesvári. UCT is based on the UCB1 formula derived by Auer, Cesa-Bianchi, and Fischer, and first applied to multi-stage decision making models (specifically, Markov Decision Processes) by Chang, Fu, Hu, and Marcus... Most contemporary implementations of Monte Carlo tree search are based on some variant of UCT." (Wikipedia entry, Monte Carlo tree search, accessed May 3, 2016)

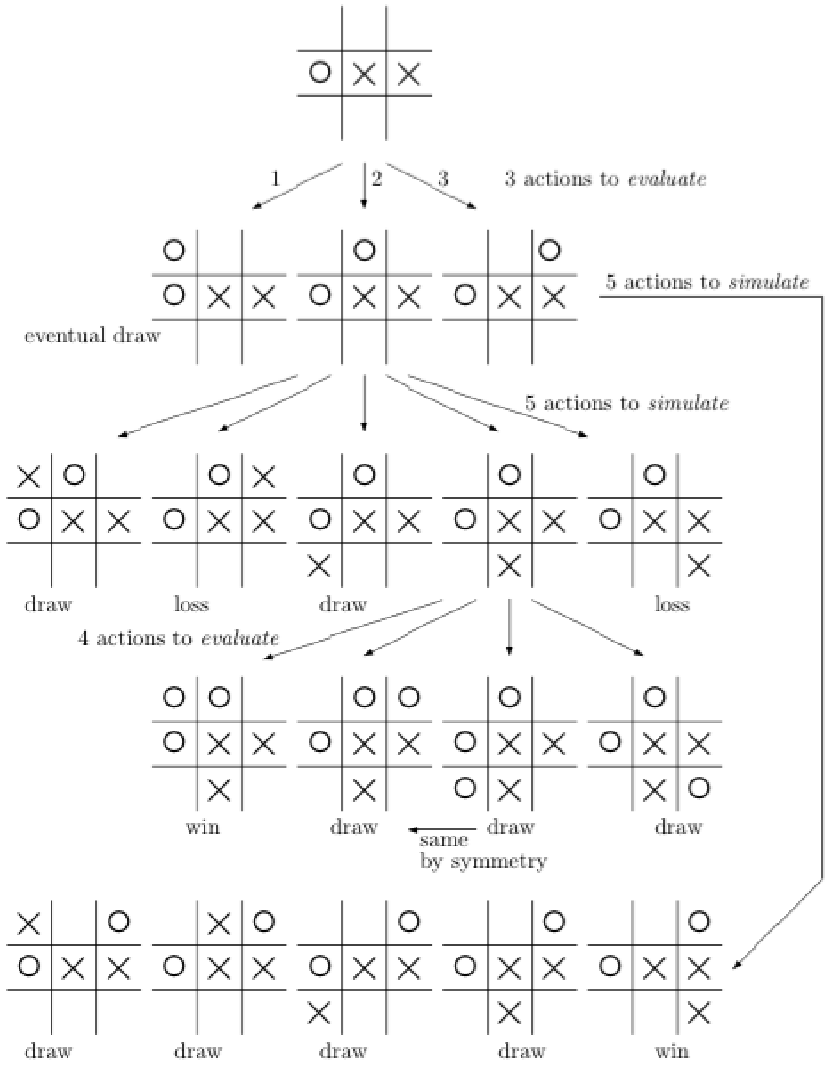

Figure 8: Monte Carlo game tree for 2nd tic-tac-toe board configuration of Figure 4.

## 3.1 AlphaGo and Monte Carlo Tree Search

Now we turn to AlphaGo, whose two main components are depicted in Figure 9:

- value network: estimate the "value" of a given board configuration (state), i.e., the probability of winning from that position;
- policy network: estimate 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.

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

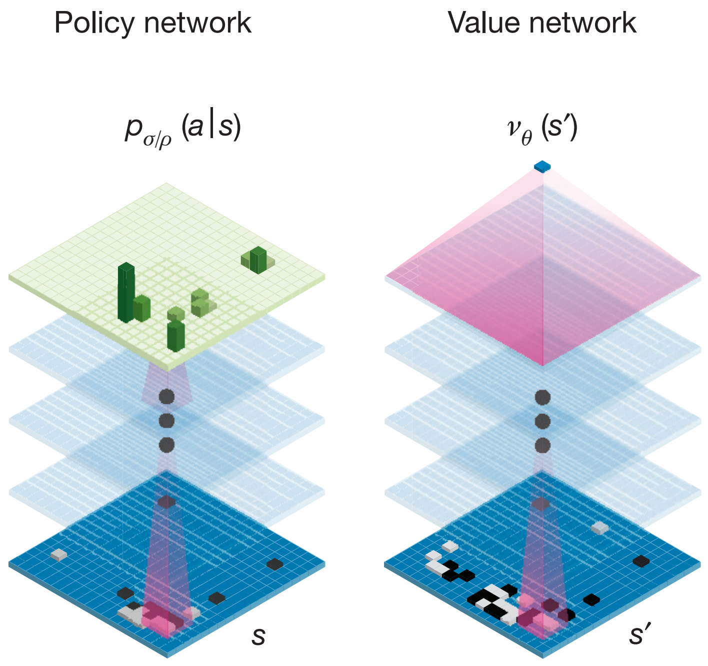

Figure 9: Alpha Go's two deep neural networks, Figure 1b taken from Silver et al. (2016) *Nature* article.

"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 9.

As an example, consider again the tic-tac-toe example, righthand side game of Figure 4, for which we derived the Monte Carlo game tree as Figure 8. The corresponding decision tree is shown in Figure 10, where note that the probabilities are omitted, since these are unknown. In practice the "outcomes" would also be estimated. In this particular example, action 3 dominates action 1 regardless of the probabilities, whereas between actions 2 and 3, it is unclear which is better without the probabilities.

AlphaGo's use of Monte Carlo tree search is described in the 2016 *Nature* article:
"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

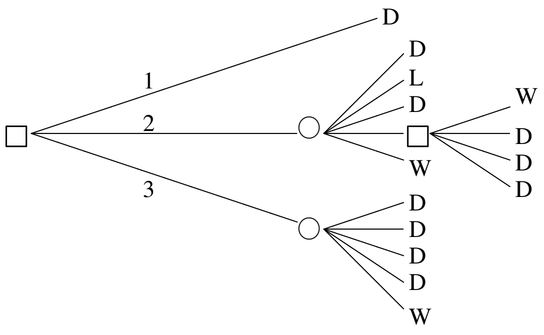

Figure 10: Decision tree for 2nd tic-tac-toe board configuration of Figure 4.

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 9 shows the corresponding $p(a|s)$ for the policy neural network that is used to simulate/sample 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.

We end by parsing each of the four main "operators" described in AlphaGo's Monte Carlo tree search, in the context of our previous examples and the simulation-based adaptive multi-stage MDP algorithm:

- "Selection" corresponds to an action node in a decision tree, and the choice is based on the upper confidence bound (UCB) for each possible action, which is a function of the current estimated $Q$-value plus a "fudge" factor, just as in (1) from AMS.
- "Expansion" corresponds to an outcome node in a decision tree, which is an opponent's move in a game, and it is modeled by a probability distribution that is a function of the "post-decision" state after the decision maker's action, corresponding to the transition probability in an MDP model.
- "Evaluation" corresponds to returning the estimated $Q$-value for a given action at a "terminal" node, which could correspond to the actual end of the horizon or simply 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.

Due to the relative symmetry of the black and white positions in Go, one could actually use selection for both sides or expansion for both sides, though most likely not for the initial move in the latter case.

## 4 CONCLUSIONS

The success of AlphaGo is due ultimately to the two "deep" neural networks for effectively and efficiently (through appropriate parametrization) approximating the value function approximation and representing 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.

## ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation (NSF) under Grants CMMI-1362303, CMMI-1434419, and by the Air Force of Scientific Research (AFOSR) under Grant FA9550-15-10050.

## REFERENCES

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

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

Broadie, M., and P. Glasserman. 1996. “Estimating Security Price Derivatives Using Simulation”. *Management Science* 42 (2): 269–285.

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-Verlag (2nd edition 2013).

Chang, H.S., M.C. Fu, J. Hu, and S.I. Marcus. 2016. “Google Deep Mind’s AlphaGo”. *OR/MS Today*.

Coulom, R. 2006. “Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search”. Computers and Games, 5th International Conference, CG.

Gosavi, A. 2003. *Simulation-Based Optimization: Parametric Optimization Techniques and Reinforcement Learning*, Kluwer (2nd edition 2015).

Kahneman, D., and A. Tversky. 1979. “Prospect Theory: An Analysis of Decision Under Risk”. *Econometrica* 47 (2): 263–291.

Kocsis, L., and C. Szepesvári. 2006. “Bandit based Monte-Carlo Planning”. *Proceedings of the 17th European Conference on Machine Learning*, Berlin, Germany: Springer, 282–293.

Powell, W.B. 2010. *Approximate Dynamic Programming*. 2nd ed. New York: Wiley.

Prashanth, L.A., C. Jie, M.C. Fu, S.I. Marcus, and C. Szepesvári. 2016. “Cumulative Prospect Theory Meets Reinforcement Learning: Prediction and Control”. *Proc. 33rd ICML*, New York.

Silver, D., A. Huang, Aja, C. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, D. Hassabis. 2016. ”Mastering the Game of Go with Deep Neural Networks and Tree Search”. *Nature* 529 (28 Jan): 484–503.

## AUTHOR BIOGRAPHY

**MICHAEL C. FU** is Ralph J. Tyser Professor 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. His research interests include simulation optimization and applied probability, with applications to manufacturing, supply chain management, and financial engineering. 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 served as WSC2011 Program Chair, NSF Operations Research Program Director, *Management Science* Stochastic Models and Simulation Department Editor, and *Operations Research* Simulation Area Editor. He is a Fellow of **INFORMS** and **IEEE**. His email address is mfu@umd.edu.