This article was downloaded by: [216.73.216.182] On: 04 June 2026, At: 01:07
Publisher: Institute for Operations Research and the Management Sciences (INFORMS)
INFORMS is located in Maryland, USA

INFORMS TutORials in Operations Research

Publication details, including instructions for authors and subscription information:
http://pubsonline.informs.org

Markov Decision Processes, AlphaGo, and Monte Carlo Tree Search: Back to the Future

Michael C. Fu

To cite this entry: Michael C. Fu. Markov Decision Processes, AlphaGo, and Monte Carlo Tree Search: Back to the Future. *In* INFORMS TutORials in Operations Research. Published online: 03 Oct 2017; 68-88.
https://doi.org/10.1287/educ.2017.0166

Full terms and conditions of use: https://pubsonline.informs.org/Publications/Librarians-Portal/PubsOnLine-Terms-and-Conditions

This article may be used only for the purposes of research, teaching, and/or private study. Commercial use or systematic downloading (by robots or other automatic processes) is prohibited without explicit Publisher approval, unless otherwise noted. For more information, contact permissions@informs.org.

The Publisher does not warrant or guarantee the article's accuracy, completeness, merchantability, fitness for a particular purpose, or non-infringement. Descriptions of, or references to, products or publications, or inclusion of an advertisement in this article, neither constitutes nor implies a guarantee, endorsement, or support of claims made of that product, publication, or service.

Copyright © 2017, INFORMS

Please scroll down for article—it is on subsequent pages

With 12,500 members from nearly 90 countries, INFORMS is the largest international association of operations research (O.R.) and analytics professionals and students. INFORMS provides unique networking and learning opportunities for individual professionals, and organizations of all types and sizes, to better understand and use O.R. and analytics tools and methods to transform strategic visions and achieve better outcomes.
For more information on INFORMS, its publications, membership, or meetings visit http://www.informs.org

TUTORIALS IN
OPERATIONS RESEARCH
INFORMS 2017

© 2017 INFORMS | ISBN 978-0-9906153-0-9
https://doi.org/10.1287/educ.2017.0166

Markov Decision Processes, AlphaGo, and Monte Carlo Tree Search: Back to the Future

Michael C. Fu
Robert H. Smith School of Business and Institute for Systems Research, University of Maryland, College Park, Maryland 20742, mfu@umd.edu

Abstract In 2016, a computer Go-playing program called AlphaGo stunned the (human) world by winning a match (4 games to 1) against the reigning human world champion, a feat more impressive than previous victories by computer programs in chess (Deep Blue) and the TV game show *Jeopardy*! (Watson). The main engine behind AlphaGo combines machine learning approaches in the form of deep neural networks with a technique called Monte Carlo tree search, whose roots can be traced back to an adaptive multistage sampling simulation-based algorithm for Markov decision processes (MDPs) published in *Operations Research* in 2005 [H. S. Chang, M. C. Fu, J. Hu, and S. I. Marcus. An adaptive sampling algorithm for solving Markov decision processes. *Operations Research* 53(1):126–139, 2005] (and introduced even earlier in 2002). This tutorial describes AlphaGo and the simulation-based MDP algorithm, as well as providing contextual and historical background material for both, and uses simple examples to illustrate the main ideas behind Monte Carlo tree search.

Keywords Markov decision processes; Monte Carlo; stochastic simulation; sampling; tree search; games

1. Introduction

The framework of Markov decision processes (MDPs) provides a very general framework for sequential decision making under uncertainty. When both the actions and states are discrete, MDPs can be simplified to a graphical decision tree representation that is easy to visualize and understand. When the uncertainty or randomness in the model, often generically referred to as "nature," is an opponent in a game, then decision trees become game trees. If the game itself is deterministic, then the complete game tree can be used to determine optimal strategies. However, in the case where the state space is far too large to generate the complete game tree, one must determine how to explore the possible games, which will frequently be referred to as a "simulated path" or "sample path" throughout. In facing this challenge, one faces a well-known exploration–exploitation trade-off, which arises in at least two forms in this context: the usual choice in pursuing actions that have been very promising in previous simulated paths versus relatively unexplored paths and how far the path itself should be simulated (i.e., all the way to the end of the game or just to a point where confidence is high about the eventual outcome). The MDP term "policy" in a game context ends up taking two forms: the player's best move for a given position (MDP state) and the opponent's move via sampling (state transition)—i.e., a randomized policy for the latter (which can also become deterministic once the game tree has been explored sufficiently for a given state).

This tutorial focuses on an approach called Monte Carlo tree search (MCTS), which is now used in the most successful computer Go-playing programs and was coined by Rémi Coulom in 2006 (Coulom [11]). This approach is a sampling/simulation-based approach to estimating the "value" of a move (e.g., the probability of leading to an eventual win) by generating

Downloaded from informs.org by [216.73.216.182] on 04 June 2026, at 01:07 . For personal use only, all rights reserved.

promising paths in the game tree, and it is based on an algorithm originally designed for MDPs by Chang et al. [7] called adaptive multistage sampling (AMS), which exploits the upper confidence bound (UCB) approach for multiarmed bandit models (Agrawal [1], Auer et al. [2]). After providing some historical background on Go and AlphaGo, I will describe the main ideas behind MCTS, illustrating the relationship to decision trees and using the game of tic-tac-toe to demonstrate concepts. Ultimately, the success of AlphaGo reflects the effectiveness of its two main engines (i.e., the two deep neural networks); MCTS merely provides an effective mechanism to train these nets. However, AMS is a very general approach for MDPs, and its incarnation as MCTS should prove successful in many other game settings (deterministic or stochastic) where the state space is too huge to explore exhaustively. Note that, similar to chess, Go is a deterministic game and unlike games such as backgammon, bridge, and poker. Randomness (Monte Carlo) is introduced for the purpose of "sampling" the opponent's possible moves according to a distribution when the optimal move is not yet known. Each such simulated path augments the game tree, which in turn leads to a better estimate of the "value function" in each state of the tree and an improved overall policy. However, MCTS should be equally applicable to games that are inherently stochastic. Similarly, AMS should be applicable to many sequential decision-making problems under uncertainty settings where, again, the state space is huge and the only way to generate paths is through a simulation model.

The remainder of this tutorial will cover the following: an introduction to Go and an overview of AlphaGo, motivation for MCTS using illustrative examples of decision trees and tic-tac-toe, background on multiarmed bandit models using UCBs and simulation-based algorithms for MDPs (Chang et al. [8, 10]), followed by the specific AMS algorithm in Chang et al. [7], which is then connected back to MCTS and AlphaGo.

2. Go, Go AlphaGo

Go is the most popular two-player board game in East Asia, tracing its origins to China more than 2,500 years ago. According to *Wikipedia*,[1] Go is also thought to be the oldest board game still played today. Since the size of the board is 19×19, compared with 8×8 for a chess board, the number of board configurations for Go far exceeds that for chess, with estimates at approximately 10^{170} possibilities, putting it a googol (no pun intended) times beyond that of chess and exceeding the number of atoms in the universe (Hassabis [14]). 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 like 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).

Perhaps the closest game in the Western world to Go is Othello: like chess, it is played on an 8×8 board, and like Go, the player with the black pieces moves first. Similar to Go, there is also a "flanking" objective but with far simpler rules. The number of legal positions is estimated at less than 10^{28}, nearly a googol and a half times less than the estimated number of possible Go board positions. As a result of the far smaller number of possibilities, traditional exhaustive game tree search (which could include heuristic procedures such as genetic algorithms and other evolutionary approaches leading to a pruning of the tree) can in principle handle the game of Othello, so that brute-force programs with enough computing power will easily beat any human. 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

[1] *Wikipedia*, s.v. "Go (Game)," last modified August 21, 2017, https://en.wikipedia.org/wiki/GO_(game).

Downloaded from informs.org by [216.73.216.182] on 04 June 2026, at 01:07 . For personal use only, all rights reserved.

available computing power, whereas such an approach is doomed to fail for the game of Go, merely because of the 19×19 size of the board.

Similarly, IBM Deep Blue's victory (by 3 1/2 to 21/2 points) 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.[2] 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 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. These components are the basis of simulation-based approaches to solving Markov decision processes (Chang et al. [8, 10], Gosavi [13]), also studied under the following names:

- neuro-dynamic programming (Bertsekas and Tsitsiklis [4]),
- approximate (or adaptive) dynamic programming (Powell [20]), and
- reinforcement learning (Gosavi [13], Sutton and Barto [22]).

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.

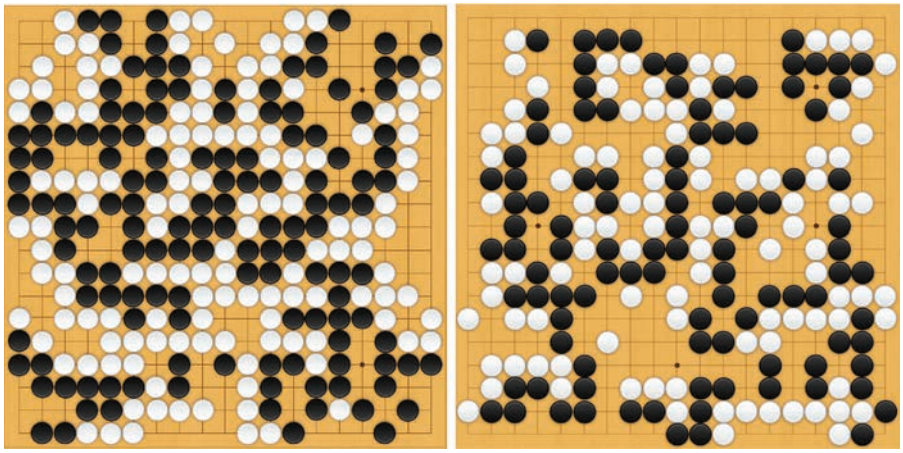
In March 2016 in Seoul, Korea, Google DeepMind's computer program AlphaGo defeated the reigning human world champion Go player Lee Sedol four games to one, representing yet another advance in artificial intelligence (AI) arguably more impressive than previous victories by computer programs in chess (IBM's Deep Blue) and *Jeopardy*! (IBM's Watson), because of the sheer size of the problem. A little over a year later, in May 2017 in Wuzhen, China, at the Future of Go Summit, a gathering of the world's leading Go players, AlphaGo cemented its reputation as the planet's best by defeating Chinese Go grandmaster and world number one (at only 19 years old) Ke Jie (three games to none). To get a flavor of the complexity of the game, the final board positions of games 1 and 3 are shown in Figure 1. In game 1, the only one of the three played to completion, AlphaGo won by just 0.5 points playing the white pieces, with a compensation (called komi) of 7.5 points. In game 3, Ke Jie playing white resigned after 209 moves.

As will be described later in more detail, AlphaGo's main engine consists of two deep neural networks that are trained using MCTS. Current versions of MCTS used in Go-playing algorithms are based on a version developed for games called UCT (upper confidence bound 1 applied to trees) (Kocsis and Szepesvári [18]), which traces its roots back to the AMS algorithm for estimating value functions in finite horizon MDPs introduced by Chang et al. [7], which was the first use of UCBs (Auer et al. [2]) for Monte Carlo simulation-based solution of MDPs. The UCB approach is one solution introduced in the machine learning community to address the exploration–exploitation trade-off that arises in multiarmed bandit problems. I will later briefly review the main idea of UCB as well as illustrate AMS and MCTS through the use of two simple examples: decision trees and tic-tac-toe—but first, a little more on DeepMind and AlphaGo.

[2] *Wikipedia*, s.v. "Deep Blue versus Garry Kasparov," last modified June 23, 2017, https://en.wikipedia.org/wiki/Deep_Blue_versus_Garry_Kasparov.

Downloaded from informs.org by [216.73.216.182] on 04 June 2026, at 01:07 . For personal use only, all rights reserved.

FIGURE 1. Final positions of Games 1 (left) and 3 (right) between AlphaGo and top-rated human Ke Jie; AlphaGo is white in Game 1 and black in Game 3 (Reproduced from the DeepMind website).



2.1. DeepMind

DeepMind is a London-based AI company founded in 2010 by Demis Hassabis, Shane Legg, and Mustafa Suleyman, and it was acquired by Google in 2014 for $500 million. The company built its reputation on the use of deep neural networks for AI applications, most notably video games. AlphaGo's neural networks employ 12 layers with millions of connections. In 2016, the DeepMind web page declared that AlphaGo is

> THE FIRST COMPUTER PROGRAM TO EVER BEAT A PROFESSIONAL PLAYER AT THE GAME OF GO, (https://deepmind.com/alpha-go.html, accessed May 11, 2016, emphasis in original)

referring to the October 2015 defeat of the reigning European champion, Fan Hui, by a stunningly dominant margin of 5 games to 0. Almost all AI experts had assumed that such an advance was decades away (see the next section).

Now more than a year later, the same web page begins,

> AlphaGo is the first computer program to defeat a professional human Go player, the first program to defeat a Go world champion, and arguably the strongest Go player in history. (accessed May 26, 2017)

The website description goes on to say that

> AlphaGo's 4-1 victory in Seoul, South Korea, in March 2016 was watched by over 200 million people worldwide. It was a landmark achievement that experts agreed was a decade ahead of its time, and earned AlphaGo a 9 dan professional ranking (the highest certification)—the first time a computer Go player had ever received the accolade. (accessed May 26, 2017)

Also added to the web page is a blog called *Innovations of AlphaGo*. It describes strategic and tactical innovations and playing insights generated by AlphaGo's play, which is somewhat specialized to the particular game of Go, but more broadly, the AI scientists at DeepMind concluded from scores of postgame analyses that

> AlphaGo's strategy embodies a spirit of flexibility and open-mindedness: a lack of preconceptions that allows it to find the most effective line of play.... [This] often leads AlphaGo to discover counterintuitive yet powerful moves. (Baker and Hui [3])

And hot off the press after the victory, DeepMind added a new blog on May 27 called *AlphaGo's Next Move* in which it announced that the company was publishing a special set of 50 games in which AlphaGo played against itself, "which we believe contain many new and interesting ideas and strategies" (Hassabis and Silver [15]).

Downloaded from informs.org by [216.73.216.182] on 04 June 2026, at 01:07 . For personal use only, all rights reserved.

In more general terms, DeepMind describes its AI research as follows:

> The algorithms we build are capable of *learning* for themselves directly from raw experience or data, and are *general* in that they can perform well across a wide variety of tasks straight out of the box. Our world-class team consists of many renowned experts in their respective fields, including but not limited to deep neural networks, reinforcement learning and systems neuroscience-inspired models. ... Our *Nature* paper ... describes the technical details behind a new approach to computer Go that combines Monte-Carlo tree search with deep neural networks that have been trained by supervised learning, from human expert games and by reinforcement learning from games of self-play. (accessed May 27, emphasis in original.)

2.2. The Roots of Monte Carlo Tree Search

As mentioned already in the introduction, Coulom appears to be the first to use the term "Monte Carlo tree search," introduced in a conference paper presented in 2006 (Coulom [11]).[3] Strikingly, in his paper, he writes the following (emphasis added):

> Our approach is similar to the algorithm of Chang, Fu and Marcus[4] [sic]. ... In order to avoid the dangers of completely pruning a move, it is possible to design schemes for the allocation of simulations that reduce the probability of exploring a bad move, without ever letting this probability go to zero. Ideas for this kind of algorithm can be found in ... n-armed bandit problems, ... [which] are the basis for the *Monte-Carlo tree search* algorithm of Chang, Fu and Marcus [sic]. (p. 73)

In other words, Coulom himself refers to the MDP algorithm in Chang et al. ([7]) as a Monte Carlo tree search algorithm. This algorithm was developed in 2002, presented at a Cornell University colloquium/seminar in the School of Operations Research and Industrial Engineering on April 30, and submitted to *Operations Research* shortly thereafter (in May).

Also, as alluded to earlier, current versions of MCTS used in Go-playing algorithms are based on a version developed for games called UCT (Kocsis and Szepesvári [18]), which refers to the UCB simulation-based MDP algorithm in Chang et al. ([7]) as follows:

> Recently, Chang et al. also considered the problem of selective sampling in finite horizon undiscounted MDPs [citation omitted]. However, since they considered domains where there is little hope that the same states will be encountered multiple times, their algorithm samples the tree in a depth-first, recursive manner: At each node they sample (recursively) a sufficient number of samples to compute a good approximation of the value of the node. The subroutine returns with an approximate evaluation of the value of the node, but the returned values are not stored (so when a node is revisited, no information is present about which actions can be expected to perform better). Similar to our proposal, they suggest to propagate the average values upwards in the tree and sampling is controlled by upper-confidence bounds. They prove results similar to ours, though, due to the independence of samples the analysis of their algorithm is significantly easier. They also experimented with propagating the maximum of the values of the children and a number of combinations. These combinations outperformed propagating the maximum value. When states are not likely to be encountered multiple times, our algorithm degrades to this algorithm. On the other hand, when a significant portion of states (close to the initial state) can be expected to be encountered multiple times then we can expect our algorithm to perform significantly better. (p. 292)

In other words, the main difference is in terms of algorithmic implementation, specifically in that returned values are not stored.

[3] See also *Wikipedia*, s.v. "Monte Carlo tree search," last modified July 4, 2017, https://en.wikipedia.org/wiki/Monte_Carlo_tree_search.

[4] In this excerpt, Coulom is referring to Chang et al. [7].

Downloaded from informs.org by [216.73.216.182] on 04 June 2026, at 01:07 . For personal use only, all rights reserved.

A high-level summary of MCTS is given in the abstract of a 2012 survey article, "A Survey of Monte Carlo Tree Search Methods":

> Monte Carlo Tree Search (MCTS) is a recently proposed search method that combines the precision of tree search with the generality of random sampling. It has received considerable interest due to its spectacular success in the difficult problem of computer Go, but has also proved beneficial in a range of other domains. (Browne et al. [6], p. 1)

The same article later provides the following overview description of MCTS:

> The basic MCTS process is conceptually very simple ... A tree is built in an incremental and asymmetric manner. For each iteration of the algorithm, a tree policy is used to find the most urgent node of the current tree. The tree policy attempts to balance considerations of exploration (look in areas that have not been well sampled yet) and exploitation (look in areas which appear to be promising). A simulation is then run from the selected node and the search tree updated according to the result. This involves the addition of a child node corresponding to the action taken from the selected node, and an update of the statistics of its ancestors. Moves are made during this simulation according to some default policy, which in the simplest case is to make uniform random moves. A great benefit of MCTS is that the values of intermediate states do not have to be evaluated, as for depth-limited minimax search, which greatly reduces the amount of domain knowledge required. Only the value of the terminal state at the end of each simulation is required. (Browne et al. [6], p. 1–2)

In the case of AlphaGo, the value function approximation neural network can also be used to terminate the simulation (depth-first traversing of the tree) earlier. The survey also provides a historical timeline on MCTS, stating that "the development of MCTS is the coming together of numerous different results in related fields of research in AI" (Browne et al. [6], p. 7). However, there is a glaring gap between the 2002 reference (Auer et al. [2]) and the two 2006 conference proceedings papers (Coulom [11], Kocsis and Szepesvári [18]) when, clearly, the 2005 *Operations Research* article (Chang et al. [7]) (which, as was noted earlier, was actually submitted back in 2002) served as a direct link, since it is cited in both papers but was not included among the 240 references. Specifically, the AMS algorithm in Chang et al. [7] chooses to sample the action that maximizes the UCB. As described there,

> Based on recent results for multiarmed bandit problems, we propose an adaptive sampling algorithm that approximates the optimal value of a finite-horizon Markov decision process (MDP) with finite state and action spaces. The algorithm adaptively chooses which action to sample as the sampling process proceeds and generates an asymptotically unbiased estimator. ... [It] 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. (p. 126, emphasis in original)

2.3. Who Can Predict the Future? And ... Is Monte Carlo Tree Search a Brute-Force Algorithm?

Brexit and the 2016 U.S. presidential election are recent examples of the perils of predictions. Clearly, predicting outcomes, especially those in the future, can be a challenge. Here was a prediction made in an October 2007 *IEEE Spectrum* article by Feng-Hsiung Hsu, one of the scientists on the IBM Deep Blue project and author of the book *Behind Deep Blue: Building the Computer that Defeated the World Chess Champion*:

> I believe that a world-champion-level Go machine can be built within 10 years, based on the same method of intensive analysis—brute force, basically—that Deep Blue employed for chess. (Hsu [16], p. 51)

Downloaded from informs.org by [216.73.216.182] on 04 June 2026, at 01:07 . For personal use only, all rights reserved.

The first part of his prediction was spot on, as AlphaGo succeeded in doing so within the 10-year time frame (from 2007), and at the time, he was a minority voice in making such a prognostication. However, the second part of the prediction—"based on the same method . . . brute force, basically"—would seem on the surface not to have been correct, since AlphaGo clearly did not use the same methods as Deep Blue. However, later on in the same article, the author makes the following statement within a separate companion insert titled, "Can Monte Carlo Work on Go?":

> Some of the best Go programs today employ Monte Carlo methods, which play out move possibilities internally, in random games, then select the move with the best win/loss index. *It can be considered a brute-force technique.* (Hsu [16], p. 54, emphasis added)

Later on, he goes on to write,

> Monte Carlo techniques have recently had success in Go played on a restrictive 9-by-9 board. My hunch, however, is that they won't play a significant role in creating a machine that can top the best human players in the 19-by-19 game. Even so, Monte Carlo is worth keeping in mind for games and gamelike computing challenges of truly daunting complexity. (Hsu [16], p. 54)

So in the end, was he right or wrong in his prediction(s)? I guess it depends on how you look at it, or you could say it was a mixed bag. One moral of the story is that qualifying your predictions and statements is always a good idea.

3. Decision Trees and Monte Carlo Tree Search

To motivate MCTS, as well as connect it with MDPs, we use decision trees, which are simple to understand and easy to depict; they look similar to game trees if you rotate your head 90 degrees. We use simple examples for illustrative purposes, completely putting aside the notion of games at first, as the focus will be on (Monte Carlo) sampling and also on the choice of objectives (e.g., in the game context, should it be the probability of winning, the probability of not losing, or some other measure of strength for a given position in the game?). Then we move on to the simple two-person game of tic-tac-toe to connect decision trees (and hence, by implication, MDPs) back to MCTS. The next section will then describe the AMS MDP algorithm in more technical detail.

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:

- Buy a PowerRich lottery ticket.
- Buy a MegaHaul lottery ticket.
- Do not buy a lottery ticket.

Associated with the two different lotteries are the following corresponding outcomes:

- PowerRich: win \$100 M with probability (w.p.) 0.01; nothing otherwise.
- MegaHaul: win \$1 M w.p. 0.05; nothing otherwise.

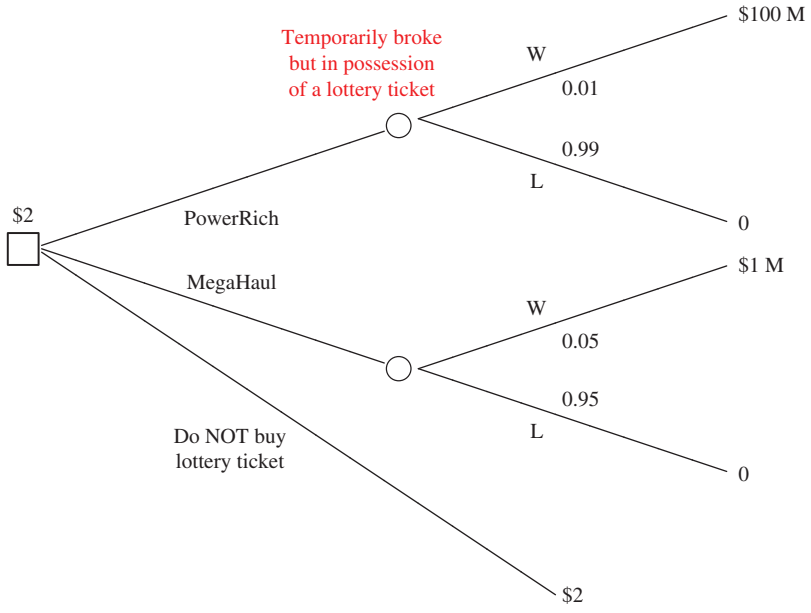
The respective expected values of the three choices are as follows:

- (Win) \$1 M.
- (Win) \$50 K.
- (Save) \$2.

In Figure 2, 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

Downloaded from informs.org by [216.73.216.182] on 04 June 2026, at 01:07 . For personal use only, all rights reserved.

FIGURE 2. Decision tree for lottery choices.



termination (or to another decision node in a multiperiod 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 of about 30 students opted for the not-playing-at-all option, a few opted for the lower payoff lottery, and the large majority remainder chose the first option, which had the highest expectation, an objective commonly assumed. When asked for an explanation as to why the nonplaying option was chosen, the one student said he or she did not believe in gambling; the other said that he or 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.

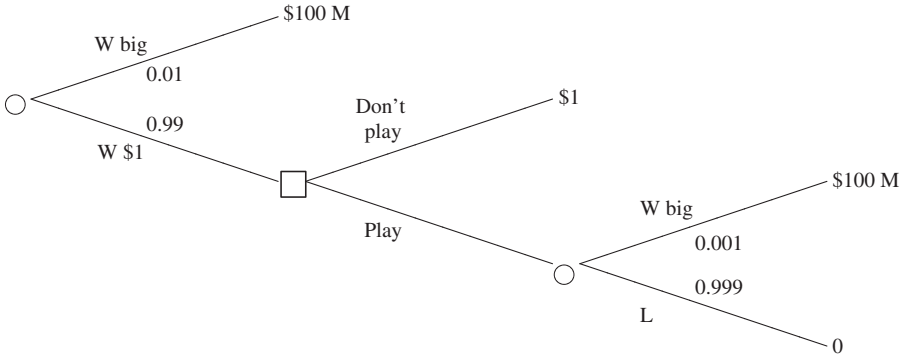
This example highlighted the importance of objectives (also of concern for AlphaGo). Aside from the obvious expected value maximization, here are some objectives that might be inferred from the Cornell students' answers to justify their choices:

- Maximize the probability of having some money for lunch (class ended just before lunch).
- Maximize the probability of not being broke (tuition is 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 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 [17], which demonstrate that humans often distort their probabilities, differentiate between gains and losses, and anchor their decisions. Recent work (Prashant et al. [19]) applies cumulative prospect theory to MDPs.

Downloaded from informs.org by [216.73.216.182] on 04 June 2026, at 01:07 . For personal use only, all rights reserved.

FIGURE 3. Decision tree for PowerRich lottery expanded once.



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 questions:

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

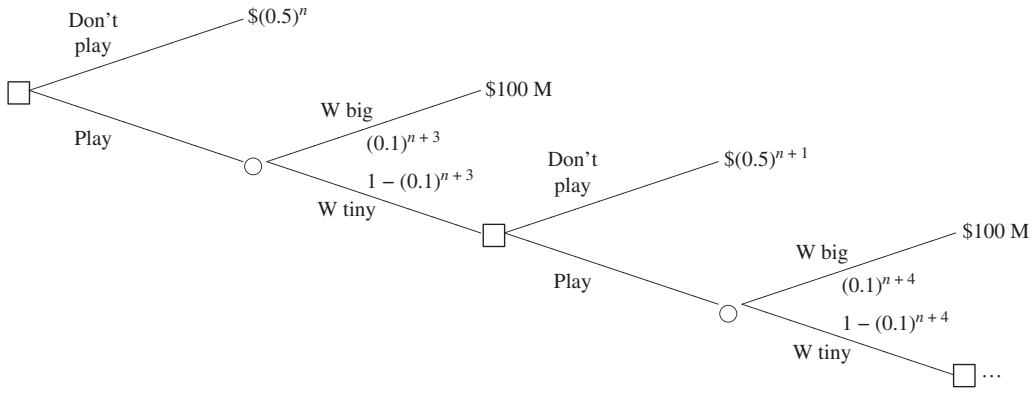
By investigating each one of these in turn, the combination gives the essence of how Go-playing programs work. In addition, it could also be the case in many decision settings that the outcome structure itself is unknown; i.e., both the number of branches, which could be uncountable, and its associated 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 us 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 ($2); again, there is no allocation as to where to simulate but just when to stop simulating. However, with the MegaHaul 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 statistical 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 3, 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 simulation

Downloaded from informs.org by [216.73.216.182] on 04 June 2026, at 01:07 . For personal use only, all rights reserved.

FIGURE 4. Decision tree for PowerRich lottery ad infinitum.



budget comes 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 4.

For 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 us 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 and outcome is realized be in one of four different states: broke, same, millionaire, or $100 M. 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. In a game, that is the position that is reached after you make your move and before your opponent makes his or 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, 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.

Example: Tic-Tac-Toe

The game of tic-tac-toe or "three in a row" is well known to most of us from childhood (at least to those of us born prior to the global domination of video games), but a quick summary will be provided for the benefit of readers unfamiliar with the game. It is a two-player game on a 3×3 grid, in which the objective is to get three marks in a row (horizontal, vertical, or diagonal, for a total of eight possible paths to victory), 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 (sometimes called a cat's game). In theory, there are something like 255 K possible configurations that can be reached, of which only an order of magnitude 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 fewer than a hundred lines of code in even the oldest standard computer programming languages.

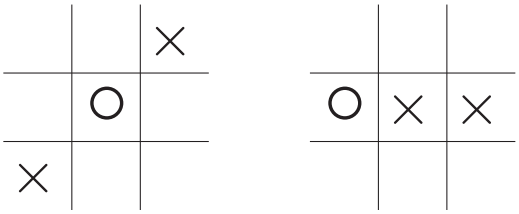
Assuming the primary objective (for both players) is to win and the secondary objective is to avoid losing, the following "greedy" policy is optimal (for both players):

- 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 as a move that is placed in a row where the opposing player already

Downloaded from informs.org by [216.73.216.182] on 04 June 2026, at 01:07 . For personal use only, all rights reserved.

FIGURE 5. Two tic-tac-toe board configurations to be considered.



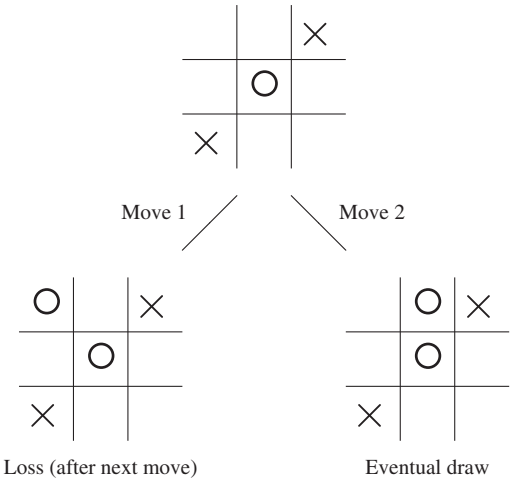
has two marks, thus preventing three in a row.) This leaves still the numerous situations where neither a win move nor a block move is available. In traditional game tree search, these would be enumerated (although as we noted previously, it is easier to implement using further if–then logic). We instead 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, and 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 unique moves (but a different set of five for the two choices) available. However, even though the game has a relatively small number of outcomes compared with 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, two by our opponent and one by us, so it is our turn to make a move. The two different board configurations are shown in Figure 5. 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: a corner move leads to a loss, and a 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 6 and 7, respectively, with the game tree that allows nonoptimal moves shown in Figure 8.

Now let us consider the other more interesting game configuration shown on the right-hand side of Figure 5, where there are three unique moves available to us, and without loss

FIGURE 6. Game tree for first tic-tac-toe board configuration (assuming "greedy optimal" play).



Note. If opponent's moves were randomized, we would have an excellent chance of winning!

Downloaded from informs.org by [216.73.216.182] on 04 June 2026, at 01:07 . For personal use only, all rights reserved.

FIGURE 7. Decision tree for first tic-tac-toe board configuration.

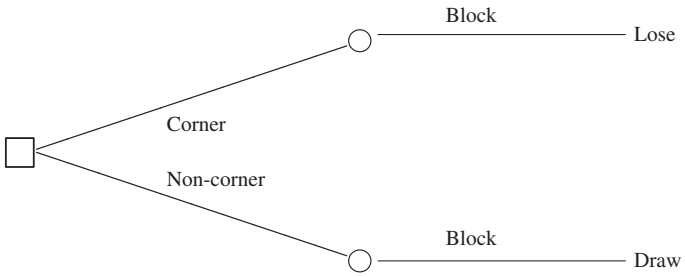
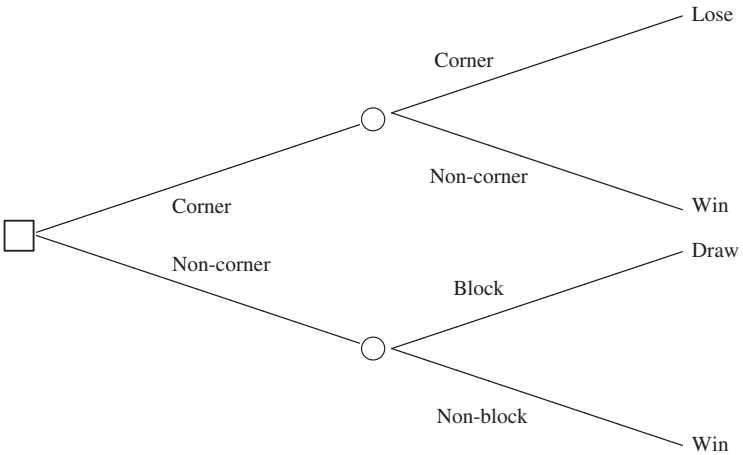


FIGURE 8. Decision tree for first tic-tac-toe board configuration with non-optimal greedy moves.



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. It turns out that going in the upper left corner leads to a draw, whereas the upper middle and upper right corner lead to five possible unique moves for the opponent. The way the Monte Carlo tree branching would work is depicted in Figure 9, 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.

4. Adaptive Multistage Sampling Algorithm

This section attempts to address the following questions:

- What is a multi-armed bandit?
- What is UCB?
- How does AMS work?
- How does MCTS work?
- How does MCTS fit into AlphaGo?

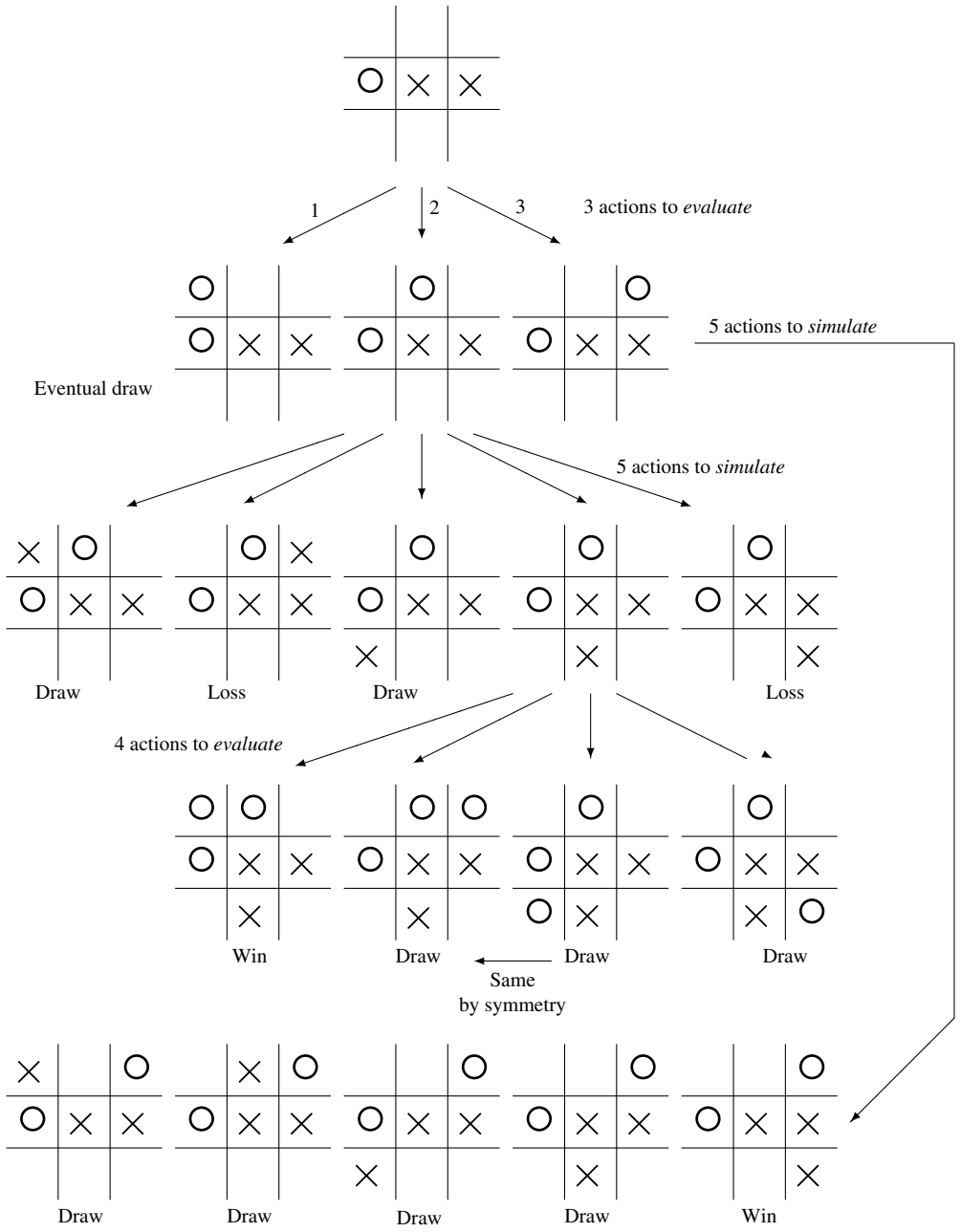
After introducing the multi-armed bandit problem and UCB index policies, the answers to the last three questions can be briefly summarized as follows:

- AMS

1. UCB: selecting our (decision maker's) actions to simulate throughout a sample path, and

Downloaded from informs.org by [216.73.216.182] on 04 June 2026, at 01:07 . For personal use only, all rights reserved.

FIGURE 9. Monte Carlo game tree for second tic-tac-toe board configuration.



2. simulation/sampling: next state transition (nature's "action").

- MCTS
  1. how to select our next moves to follow in the game tree, and
  2. how to select the opponent's move.
- MCTS in AlphaGo
  1. UCB guides AlphaGo's next move to follow in simulated game tree path, and
  2. opponent's next move sampled using policy network (probability distribution).

Downloaded from informs.org by [216.73.216.182] on 04 June 2026, at 01:07 . For personal use only, all rights reserved.

Downloaded from informs.org by [216.73.216.182] on 04 June 2026, at 01:07 . For personal use only, all rights reserved.

4.1. Multi-Armed Bandit Problems

Multi-armed bandit problems are a class of sequential decision-making problems under uncertainty, whereby a single decision maker must decide which arm to pull at each decision epoch. An arm when pulled provides a reward to the decision maker, but the reward distribution for each arm is unknown. Think of slot machines in a casino, which serve as the original motivation for this class of problems. *Index policies* assign a (generally dynamic) value to each possible arm, and the decision maker selects the arm with the current highest index. Index policies are particularly convenient because each arm's index is generally calculated based on observations specific to that particular arm. Under certain conditions, index policies can be proven to be optimal in some sense. One of the earliest and most well-known index policies is the Gittins index. The one that we will discuss in more detail is the UCB. In addition to index policies, another approach is Bayesian learning, where a (prior) distribution is assumed on the set of arms, and this distribution is updated with each observation. At any decision epoch, the decision maker chooses the action that maximizes the objective under the current (posterior) distribution. One of the most popular instantiations of this approach is Thompson sampling. All of these approaches are intended to capture the classic *exploration–exploitation trade-off* that arises in many settings, e.g., global optimization, where it signifies the balancing between local search in a promising region that has been previously explored versus considering an entirely new region. In the multi-armed bandit problem, the trade-off is between cashing in on an arm that has proved reliable in the past with trying new arms that have not done so well up to now but due to large uncertainty still have the potential to provide even higher rewards.

Again, What Is the Objective?

What does it mean to be optimal? In the multi-armed bandit problem, the most commonly used objective is expected cumulative regret, which represents the sum of expected "losses" (hence, regret) from playing a non-optimal arm up to the current time. The notion of regret and the associated regret analysis dominates the multi-armed bandit research literature. However, focusing on this objective may not be the most appropriate in many settings. Two deviations from this objective have to do with (i) the use of *expected* rewards and (ii) the use of *cumulative* rewards. An obvious alternative in the case of (i) might be the probability of exceeding some threshold (as in the classic gambler's ruin setting). In the case of (ii), one might be interested in a finite horizon problem in which all that matters is the last choice; i.e., all of the previous arm pulls are just "practice" for the real thing. Although this does not fit the original casino scene, it does correspond to the classical setting of statistical ranking and selection, which has been around for over half a century and only recently been explored in the multi-armed bandit community under the name of best-arm identification.

Upper Confidence Bounds

The idea of using upper confidence bounds for the index policies dates back to Agrawal [1], but the paper most cited in the machine learning community is Auer et al. [2], which used the assumption of bounded rewards to derive a UCB algorithm that achieves finite-time logarithmic regret rather than asymptotic results more common in the statistics literature. The AMS algorithm is a recursive extension of the UCB1 algorithm in Auer et al. [2] in the context of the MDP framework. Thus, it is an index-based policy where the index for an action is given by the sum of the current estimate of the true Q-value for the action plus a term that relates the size of the upper confidence bound.

The key aspects of the AMS MDP algorithm in Chang et al. [7] that make it particularly suitable to the game environment, and differentiate it from the traditional multi-armed bandit problem settings, are the following:

- finite horizon (terminating);
- nonstationarity.

Although in principle each of these could be handled separately, putting them together in an easy-to-implement form was not present in the bandit literature in 2005.

Other Alternatives to UCB. UCB algorithms are simple to implement, which may explain why UCB-based sampling formed the basis for Monte Carlo tree search. However, they are clearly not the only approach and not even necessarily a logical approach, since the objective in a game setting is generally to win, so it really does not matter how much is "spent" along the way in trying different actions and simulating paths. As alluded to earlier, best-arm identification has recently become an area of research in the machine learning community analogous to statistical ranking and selection—the setting is the same, but the types of results look very different.

4.2. Markov Decision Processes

Consider a finite horizon MDP with state space S, finite action space A, set of admissible actions $A(s) \subseteq A$ in state $s \in S$, nonnegative bounded reward function R: $S \times A \to \mathcal{R}^+$ (i.e., $R(s,a)$ is the reward obtained when in state $s \in S$ and action $a \in A(s)$ is taken), and transition probability function P that maps a state and action pair to a probability distribution over S (i.e., $P(s,a)(s')$ is the probability of transitioning to state $s' \in S$ when taking action $a \in A(s)$ in state $s \in S$). As in the setting of Chang et al. [7], an explicit form for the transition probability P is assumed to be unavailable, but samples of the transition can be obtained. For an MDP model, such samples are realizations of nature or a simulation model, whereas in the game setting, they represent moves by the opposing players. 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.

Let s_t denote the state in period (or stage) t, and a_t the action taken in period t. Assume that the objective is to maximize expected total reward over a finite horizon of (deterministic finite) length H; i.e., $E[\sum_{i=0}^{H-1} R(s_t, a_t)]$. Define a decision rule for period t by the mapping of states to actions π_t: $S \to A, t \geq 0$ and a policy as the set of decision rules over the horizon; i.e., $\pi = \{\pi_t, t = 0, \ldots, H-1\}$. The algorithm in Chang et al. [7] estimates the optimal total reward (thereby obtaining an approximate optimal policy) for horizon length H.

The algorithm works by recursively estimating the optimal reward-to-go value function for state s in period i defined by

$$V_i^*(s) = \sup_{\pi} E\left[\sum_{t=i}^{H-1} R(s_t, \pi_t(s_t)) \,\middle|\, s_i = s\right], \quad s \in S,\ i = 0, \ldots, H-1,$$

with $V_H^*(s) = 0$ for all $s \in S$, and satisfying the well-known (Bellman) optimality equation,

$$V_i^*(s) = \max_{a \in A} Q_i^*(s,a),$$
$$Q_i^*(s,a) = R(s,a) + E[V_{i+1}^*(S'(s,a))],$$

written in a form using Q-functions and next state (random variable) $S' \sim P(s,a)$.

The algorithm in Chang et al. [7] estimates $Q_i^*(s,a)$ by a sample mean estimate $\hat{Q}_i$ given by (3), initialized by (1), and chooses to sample an action a^* that achieves

$$\max_{a \in A} (\hat{Q}_i(s,a) + \text{"fudge factor"}),$$

where the quantity being maximized is the UCB. The resulting AMS algorithm is given in Figure 10, where the specific form of the UCB is given by the quantity in (2), which assumes

Downloaded from informs.org by [216.73.216.182] on 04 June 2026, at 01:07 . For personal use only, all rights reserved.

FIGURE 10. Adaptive multistage sampling algorithm in Chang et al. [7] (undiscounted version).

Adaptive Multistage Sampling (AMS)

- **Input:** state $s \in S$, $N_i \geq |A|$, and period i. **Output:** $\hat{V}_i^{N_i}(s)$.
- **Initialization:** Sample each action $a \in A$ sequentially once in state s and set

$$\hat{Q}_i(s,a) = \begin{cases} 0 & \text{if } i = H \text{ and go to } \textbf{Exit}, \\ R(s,a) + \hat{V}_{i+1}^{N_{i+1}}(S') & \text{if } i \neq H, \end{cases} \tag{1}$$

where $S' \sim P(s,a)$, and set $\bar{n} = |A|$.

- **Loop:** Sample an action a^* that achieves

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

where $N_{a,i}^s$ is the number of times action a has been sampled so far, and $\bar{n}$ is the total number of samples thus far for this period, and

$$\hat{Q}_i(s,a) = R(s,a) + \frac{1}{N_{a,i}^s} \sum_{s' \in S_a^s} \hat{V}_{i+1}^{N_{i+1}}(s'), \tag{3}$$

where S_a^s is the set of sampled next states so far with $|S_a^s| = N_{a,i}^s$ w.r.t. distribution $P(s,a)$.

· Update $N_{a^*,i}^s \leftarrow N_{a^*,i}^s + 1$ and $S_{a^*}^s \leftarrow S_{a^*}^s, \cup \{s'\}$, where s' is the newly sampled next state by a^*.

· Update $\hat{Q}_i(s,a^*)$ with the $\hat{V}_{i+1}^{N_{i+1}}(s')$ value.

· $\bar{n} \leftarrow \bar{n} + 1$. If $\bar{n} = N_i$, then exit **Loop**.

- **Exit:** Set $\hat{V}_i^{N_i}(s)$ such that

$$\hat{V}_i^{N_i}(s) = \begin{cases} \sum_{a \in A} \frac{N_{a,i}^s}{N_i} \hat{Q}_i(s,a) & \text{if } i = 0, \ldots, H-1, \\ 0 & \text{if } i = H, \end{cases} \tag{4}$$

and return $\hat{V}_i^{N_i}(s)$.

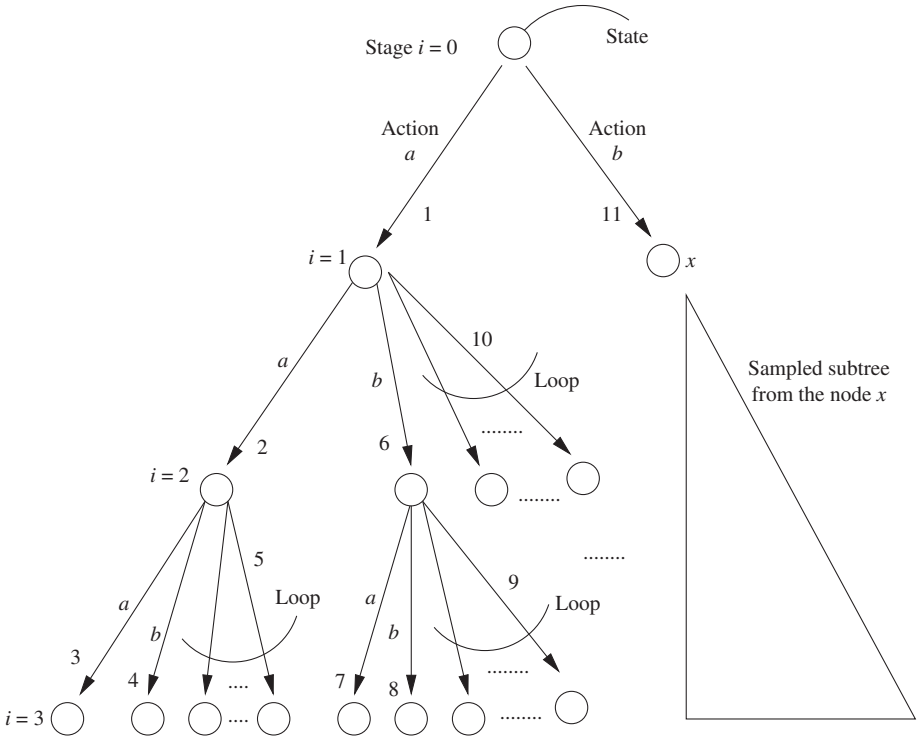
that one-period rewards are bounded by $1/H$; else, another scaling factor must multiply the fudge factor. (The original machine learning UCB algorithm in Auer et al. [2] assumes bounded rewards.) The final output of the AMS algorithm is an estimate of $V_i^*(s)$ given by (4), recursively using estimates for V_{i+1}^*.

To illustrate the allocation rule, consider first the one-stage approximation; i.e., assume that $V_1^*(s)$ for all $s \in S$ is known. To estimate $V_0^*(s)$ requires estimating $Q_0^*(s,a^*)$, where $a^* \in \arg\max_{a \in A} Q_0^*(s,a)$. The choice of a^* corresponds to the search for the best arm in the multi-armed bandit problem. Start by sampling each possible action once at the given state s, which leads to the next state according to $P(s,a)$. The iteration (see **Loop** in Figure 10) proceeds by sampling the next action achieving the maximum UCB using the current estimates of $Q_0^*(s,a)$ (see Equation (2)), where the estimate $\hat{Q}_0(s,a)$ is given by the immediate reward $R(s,a)$ plus the *sample mean* of V_1^*-values at the *sampled next states that have been sampled thus far* (see Equation (3)).

Among the N_0 samples for state s, $N_{a,0}^s$ denotes the number of samples using action a. If the sampling is done well, $N_{a,0}^s/N_0$ provides a good estimate of the likelihood that action a is optimal in state s, because in the limit as $N_0 \to \infty$, the sampling scheme should lead to $N_{a^*,0}^s/N_0 \to 1$ if a^* is the unique optimal action, or if there are multiple optimal actions—say, a set A^*—then $\sum_{a \in A^*} N_{a,0}^s/N_0 \to 1$; i.e., $\{N_{a,0}^s/N_0\}_{a \in A}$ should converge to a probability distribution concentrated on the set of optimal actions. For this reason, a weighted (by $N_{a,0}^s/N_0$) sum of the currently estimated value of $Q_0^*(s,a)$ over A is used to approximate $V_0^*(s)$ (see Equation (4)). Ensuring that the weighted sum concentrates on a^* as the sampling proceeds ensures that in the limit the estimate of $V_0^*(s)$ converges to $V_0^*(s)$. Alternatives to this weighted sum that often perform better in numerical experiments can be found in Chang et al. [7].

Downloaded from informs.org by [216.73.216.182] on 04 June 2026, at 01:07 . For personal use only, all rights reserved.

FIGURE 11. Graphical illustration of the sequence of the recursive calls made in **Initialization** of the AMS algorithm. Each circle corresponds to a state and each arrow with a noted action that signifies a sampling (and a recursive call). The boldface number near each arrow is the sequence number for the recursive calls made. For simplicity, the entire **Loop** process is signified by one call number (taken from Chang et al. [7]).



To illustrate how the recursive calls are made sequentially, Figure 11 graphically depicts the sequence of calls with two actions and $H = 3$ for the **Initialization** portion.

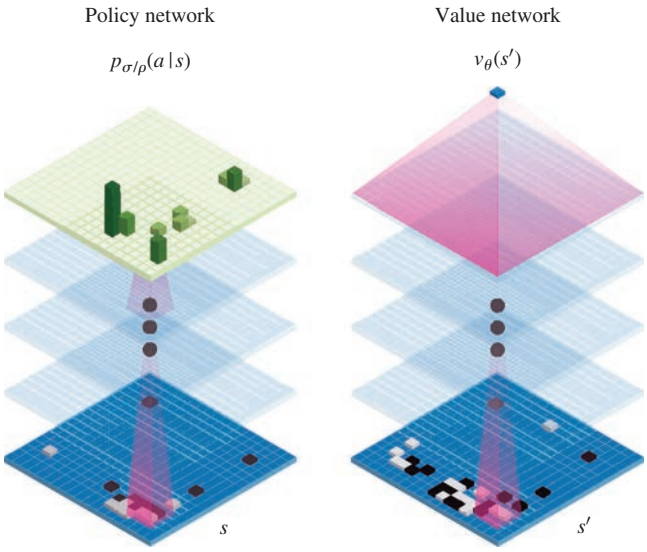
The generation of Monte Carlo trees is also the centerpiece in the American-style option pricing algorithm of Broadie and Glasserman [5], with the primary difference between AMS and their algorithm being that the option pricing setting is an optimal stopping problem, so there is *no need to choose which 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. In Monte Carlo tree search, the decision as to which action to simulate is central to its implementation. Thus, the general MDP setting gives rise to the simulation allocation problem, for which Chang et al. [7] first proposed using UCB ideas from multiarmed bandit problems, which attempts to balance exploitation and exploration.

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

In MDP notation, model the state as an 9-tuple corresponding to the nine locations, starting from upper left to bottom right, where 0 will correspond to blank, $1 = \mathrm{X}$, and $2 = \mathrm{O}$; thus, $(0,0,1,0,2,0,1,0,0)$ corresponds to the state of the right-hand side game of Figure 5. Actions will simply be represented by the nine locations, with the feasible action space the obvious remainder set of the components of the space that are still 0. This is not necessarily the best state representation (e.g., if we are trying to detect certain structure or symmetries). If the learning algorithm is working in an ideal manner, it should converge to the degenerate distribution that chooses with probability 1 the optimal action, which can be easily determined in this simple game.

Downloaded from informs.org by [216.73.216.182] on 04 June 2026, at 01:07 . For personal use only, all rights reserved.

FIGURE 12. AlphaGo's two deep neural networks (Adapted by permission from Macmillan Publishers Ltd.: *Nature* (Figure 1b, Silver et al. [21]), copyright 2016).



4.3. Back to AlphaGo and Monte Carlo Tree Search

Now we return to AlphaGo, whose two main components are depicted in Figure 12:

- *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 board configuration (state).

The subscripts σ and ρ on the policy network correspond to two different networks used, using supervised learning and reinforcement learning, respectively. The subscript θ 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 opponent move (action) a, which leads to board configuration (new "post-decision" state) s'; i.e., a sequence of a pair of moves can be modeled as

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

using the notation consistent with Figure 12.

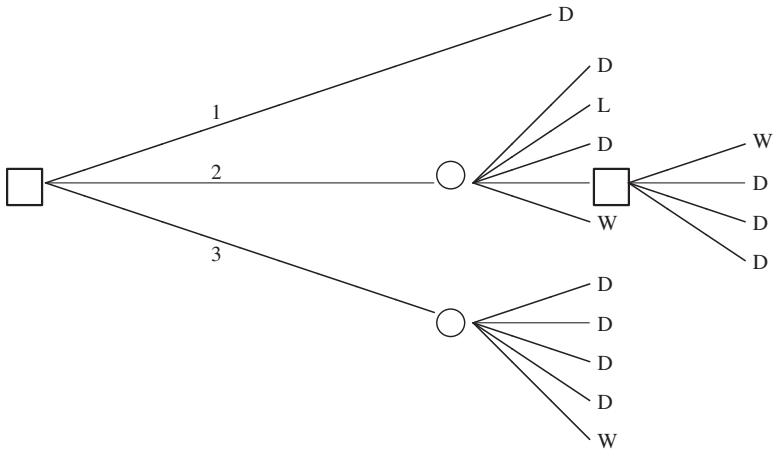
As an example, consider again the tic-tac-toe example, the right-hand side game of Figure 5, for which we derived the Monte Carlo game tree as in Figure 9. The corresponding decision tree is shown in Figure 13; note that the probabilities are omitted, since these are unknown. In practice, the "outcomes" 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.

The 2016 *Nature* article by Silver et al. [21] describes Monte Carlo tree search as follows:

> 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. (p. 484)

Downloaded from informs.org by [216.73.216.182] on 04 June 2026, at 01:07 . For personal use only, all rights reserved.

FIGURE 13. Decision tree for second tic-tac-toe board configuration.



The neural networks are then described:

> We pass in the board position as a 19×19 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. (p. 484)

And finally, as described previously,

> Our program AlphaGo efficiently combines the policy and value networks with MCTS. (p. 484)

Figure 12 shows the corresponding $p(a \mid s)$ 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.

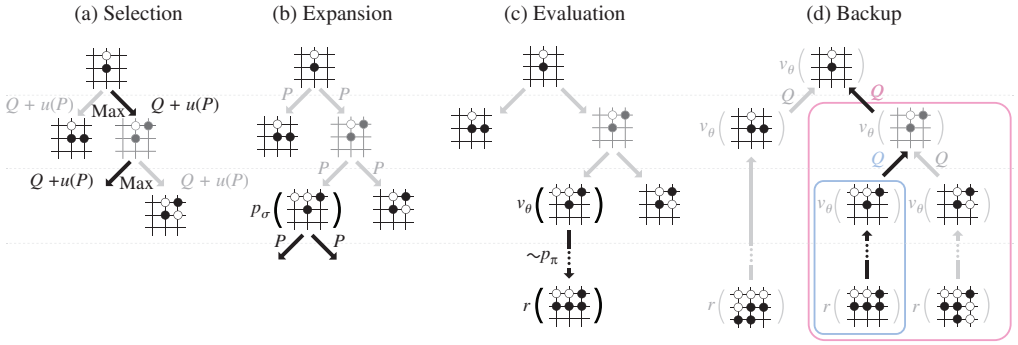
We end by parsing each of the four parts of Figure 14 (taken from Silver et al. [21]), which summarizes the main "operators" in AlphaGo's Monte Carlo tree search, in the context of our previous examples and the simulation-based adaptive multistage MDP algorithm:

- "Selection" corresponds to an action node in a decision tree, and the choice is based on the UCB for each possible action, which is a function of the current estimated Q-value plus a "fudge" factor, just as in (2) 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 value function approximation 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.

Note that due to the relative symmetry of the black and white positions (as a result of the large size of the board, 19×19, something that does not hold for the 3×3 tic-tac-toe board), one could essentially use selection for both sides or expansion for both sides, although most likely not for the initial move in the latter case. In fact, as mentioned previously,

Downloaded from informs.org by [216.73.216.182] on 04 June 2026, at 01:07 . For personal use only, all rights reserved.

FIGURE 14. Monte Carlo tree search (Adapted by permission from Macmillan Publishers Ltd.: *Nature* (Figure 3, Silver et al. [21]), copyright 2016).



AlphaGo exploits this in training by playing against itself, improving both the value and policy networks simultaneously.

5. Back to the Future

AlphaGo indeed serves as a showcase of AI, combining the engineering of machine learning (with two deep neural networks) with the sampling/simulation approach of MCTS, or Monte Carlo tree search, where the latter samples an opponent's next action and simulates the game tree (sample paths) to improve both neural nets (learning). One takeaway message is that OR played an unheralded role, as the MCTS algorithm is based on the UCB algorithm for MDPs published in *Operations Research* in 2005 (Chang et al. [7]), 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). The optimization arm of OR has already played (and continues to play) a significant role in machine learning, but the stochastic side of OR—namely, the applied probability and stochastic simulation communities—has not been similarly engaged to the same extent, when there are clearly many other ideas, approaches, and algorithms that can be brought to bear on important and challenging problems in AI. The aim of the National Science Foundation (NSF)-sponsored workshop (held at Rutgers University, May 30 to June 1, 2012), "A Conversation Between AI and OR on Sequential Decision Making" (led by Warren Powell of Princeton University and Satinder Singh of the University of Michigan), was to bring together researchers in reinforcement learning, approximate dynamic programming, simulation optimization, and stochastic programming to exchange ideas in the hopes of establishing fruitful research collaborations. The future of AI would benefit greatly from more OR ideas (and vice versa).

Acknowledgments

This work was supported in part by the National Science Foundation (NSF) [Grants CMMI-1362303 and CMMI-1434419] and by the Air Force of Scientific Research (AFOSR) [Grant FA9550-15-10050]. Much of this material is borrowed (verbatim at times, along with the figures) from the author's own previous exposition presented at the 2016 Winter Simulation Conference (Fu [12]) and the October 2016 coauthored article in *OR/MS Today* (Chang et al. [9]).

References

[1] R. Agrawal. Sample mean based index policies with $o(\log n)$ regret for the multi-armed bandit problem. *Advances in Applied Probability* 27(4):1054–1078, 1995.

[2] P. Auer, N. Cesa-Bianchi, and P. Fisher. Finite-time analysis of the multiarmed bandit problem. *Machine Learning* 47(2–3):235–256, 2002.

Downloaded from informs.org by [216.73.216.182] on 04 June 2026, at 01:07 . For personal use only, all rights reserved.

[3] L. Baker and F. Hui. Innovations of AlphaGo. *DeepMind* (blog), April 10, https://deepmind.com/blog/innovations-alphago/, 2017.

[4] D. P. Bertsekas and J. N. Tsitsiklis. *Neuro-Dynamic Programming*. Athena Scientific, Nashua, NH, 1996.

[5] M. Broadie and P. Glasserman. Estimating security price derivatives using simulation. *Management Science* 42(2):269–285, 1996.

[6] C. Browne, E. Powley, D. Whitehouse, S. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A survey of Monte Carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games* 4(1):1–43, 2012.

[7] H. S. Chang, M. C. Fu, J. Hu, and S. I. Marcus. An adaptive sampling algorithm for solving Markov decision processes. *Operations Research* 53(1):126–139, 2005.

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

[9] H. S. Chang, M. C. Fu, J. Hu, and S. I. Marcus. Google DeepMind's AlphaGo. *OR/MS Today* 43(5):24–29, 2016.

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

[11] R. Coulom. Efficient selectivity and backup operators in Monte-Carlo tree search. H. H. L. M. Donkers, H. Jaap van den Herik, and P. Ciancarini, eds. *Proceedings of the 5th International Conference on Computers and Games, CG 2006*. Springer, Berlin, 72–83, 2006.

[12] M. C. Fu. AlphaGo and Monte Carlo tree search: The simulation optimization perspective. *Proceedings of the* 2016 *Winter Simulation Conference*. IEEE, Piscataway, NJ, 659–670, 2016.

[13] A. Gosavi. *Simulation-Based Optimization: Parametric Optimization Techniques and Reinforcement Learning*. Kluwer Academic Publishers, Boston, 2003.

[14] D. Hassabis. AlphaGo: Using machine learning to master the ancient game of Go. *The Keyword* (blog), January 27, https://www.blog.google/topics/machine-learning/alphago-machine-learning-game-go/, 2016.

[15] D. Hassabis and D. Silver. AlphaGo's next move. *DeepMind* (blog), May 27, https://deepmind.com/blog/alphagos-next-move/, 2017.

[16] F.-H. Hsu. Cracking Go. *IEEE Spectrum* 44(10):50–55, 2007.

[17] D. Kahneman and A. Tversky. Prospect theory: An analysis of decision under risk. *Econometrica* 47(2):263–292, 1979.

[18] L. Kocsis and C. Szepesvári. Bandit based Monte-Carlo planning. M. Spiliopoulou, J. Fürnkranz, and T. Scheffer, eds. *Proceedings of the 17th European Conference on Machine Learning, ECML 2006* . Lecture Notes in Computer Science, Vol. 4212. Springer, New York, 282–293, 2006.

[19] L. A. Prashant, C. Jie, M. C. Fu, S. I. Marcus, and C. Szepesvári. Cumulative prospect theory meets reinforcement learning: Prediction and control. M. F. Balcan and K. Q. Weinberger, eds. *Proceedings of the* 33*rd International Conference on Machine Learning*, JMLR.org, 1406–1415, 2016.

[20] W. B. Powell. *Approximate Dynamic Programming: Solving the Curses of Dimensionality*, 2nd ed. John Wiley & Sons, New York, 2010.

[21] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, et al.. Mastering the game of Go with deep neural networks and tree search. *Nature* 529(7587):484–503, 2016.

[22] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 1998.

Downloaded from informs.org by [216.73.216.182] on 04 June 2026, at 01:07 . For personal use only, all rights reserved.