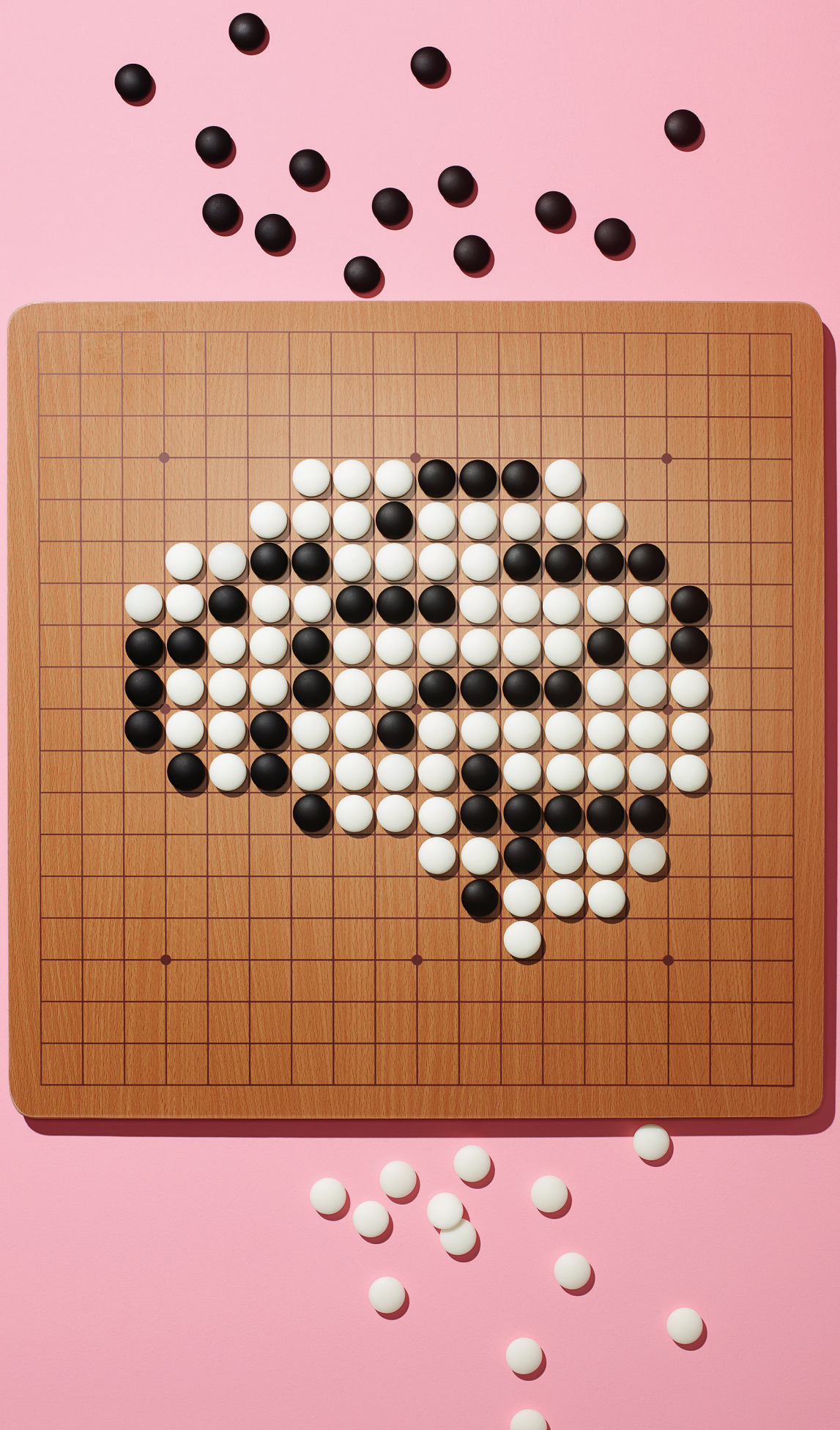


Go-bot, go

AIs may use randomness to finally master this ancient game of strategy

By **Jonathan Schaeffer**, **Martin Müller** & **Akihiro Kishimoto**
Photography by **Dan Saelinger**

Chou Chun-hsun,

one of the world's top players of the ancient game of Go, sat hunched over a board covered with a grid of closely spaced lines. To the untrained eye, the bean-size black and white stones scattered across the board formed a random design. To Chou, each stone was part of a complex campaign between two opposing forces that were battling to capture territory. The Go master was absorbed in thought as he considered various possibilities for his next move and tried to visualize how each option would affect the course of the game. Chou's strategy relied on a deep understanding of Go, the result of almost 20 years of painstaking study. • Although Chou looked calm, he knew he was in big trouble. It was 22 August 2009, and Chou was matched against a Go-playing computer running Fuego, an open-source program that we developed at the University of Alberta, in Canada, with contributions from researchers at IBM and elsewhere. The program was playing at the level of a grand master—yet it knew nothing about the game beyond the basic rules.

For decades, researchers have taught computers to play games in order to test their cognitive abilities against those of humans. In 1997, when an IBM computer called Deep Blue beat Garry Kasparov, the reigning world champion, at chess, many people assumed that computer scientists would eventually develop artificial intelligences that could triumph at any game. Go, however, with its dizzying array of possible moves, continued to stymie the best efforts of AI researchers.

But that climactic competition in 2009 showed that a computer might yet become a Go champion. In that match, an AI defeated a world-class human Go player in a no-handicap game for the first time in history. Although that game was played on a small board, not the board used in official tournaments, Fuego's win was seen as a major milestone.

Remarkably, the Fuego program didn't triumph because it had a better grasp of Go strategy. And although it considered millions of possible moves during each turn, it didn't come close to performing an exhaustive search of all the possible game paths. Instead, Fuego was a know-nothing machine that based its decisions on random choices and statistics.

THE RECIPE FOR BUILDING A SUPERHUMAN CHESS PROGRAM IS now well established. You start by listing all possible moves, the responses to the moves, and the responses to the responses, generating a branching tree that grows as big as computational resources allow. To evaluate the game positions at the end of the branches, the program needs some chess knowledge, such as the value of each piece and the utility of its location on the board. Then you refine the algorithm, say by "pruning" away branches that obviously involve bad play on either side, so that the program can search the remaining branches more deeply. Set the program to run as fast as possible on one or more computers and voilà, you have a grand master chess player. This recipe has proven successful not only for chess but also for such games as checkers and Othello. It is one of the great success stories of AI research.

Go is another matter entirely. The game has changed little since it was invented in China thousands of years ago, and millions around the world still enjoy playing it. Beginners often learn Go on a board composed of a grid of 9 lines by 9 lines before working their way up to the official board with its 19-by-19 grid. Game play sounds simple in theory: Two players take turns placing stones on the board to occupy territories and surround the opponent's stones, earning points for their successes. Yet the scope of Go makes it extremely difficult—perhaps impossible—for a program to master the game with the traditional search-and-evaluate approach.

For starters, the complexity of the search algorithm depends in large part on the branching factor—the number of possible moves at every turn. For chess, that factor is roughly 40, and a typical chess game lasts for about 50 moves. In Go, the branching factor can be more than 250, and a game goes on for about 350 moves. The proliferation of options in Go quickly becomes too much for a standard search algorithm.

There's also a bigger problem: While it's fairly easy to define the value of positions in chess, it's enormously difficult to do so on a Go board. In chess-playing programs, a relatively simple

THIS PAGE AND PREVIOUS PAGES, PROP STYLING BY **Dominique Baynes**

evaluation function adds up the material value of pieces (a queen, for example, has a higher value than a pawn) and computes the value of their locations on the board based on their potential to attack or be attacked.

Compared with that of chess pieces, the value of individual Go stones is much lower. Therefore the evaluation of a Go position is based on all the stones' locations, and on judgments about which of them will eventually be captured and which will stay safe during the shifting course of a long game. To make this assessment, human players rely on both a deep tactical understanding of the game and a clear-eyed appraisal of the overall board situation. Go masters consider the strength of various groups of stones and look at the potential to create, expand, or conquer territories across the board.

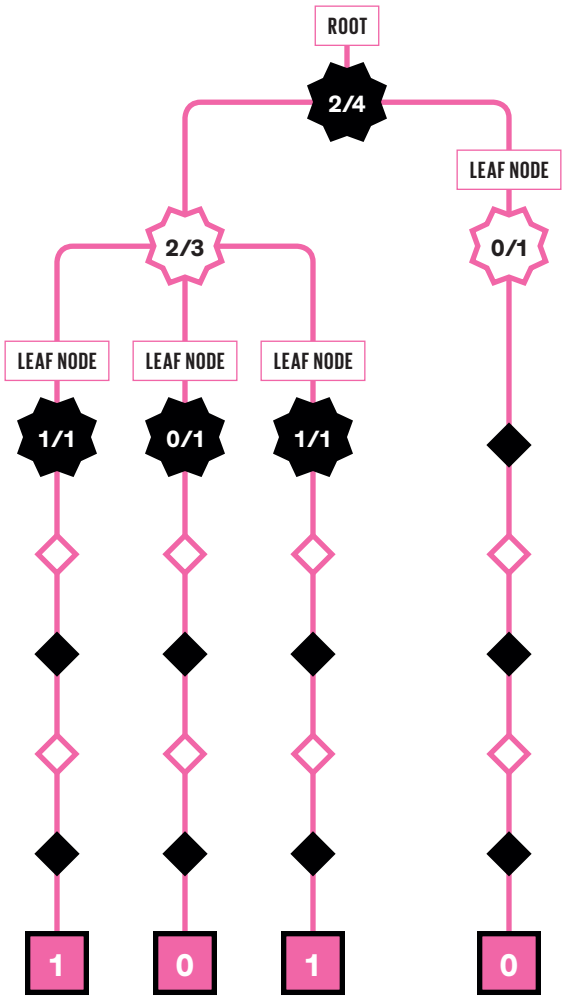
Rather than try to teach a Go-playing program how to perform this complex assessment, we've found that the best solution is to skip the evaluation process entirely. Over the past decade, several research groups have pioneered a new search paradigm for games, and the technique actually has a chance at cracking Go. Surprisingly, it's based on sequences of random moves. In its simplest form, this approach, called Monte Carlo tree search (MCTS), eschews all knowledge of the desirability of game positions. A program that uses MCTS need only know the rules of the game.

From the current configuration of stones on the board, the program simulates a random sequence of legal moves (playing moves for both opponents) until the end of the game is reached, resulting in a win or loss. It automatically does this over and over. The magic comes from the use of statistics. The evaluation of a position can be defined as the frequency with which random move sequences originating in that position lead to a win. For instance, the program might determine that when move A is played, random sequences of moves result in a win 73 percent of the time, while move B leads to a win only 54 percent of the time. It's a shockingly simple metric.

It may seem counterintuitive to try to win a deeply strategic game with a program that uses random moves to evaluate its different choices. But there are lots of precedents that show the efficacy of this statistical approach. For example, most Internet search engines do not attempt to analyze a query to try to understand the semantics of what is being asked for—they just apply some simple numerical schemes to rank results. Monte Carlo methods are also standard in disciplines such as particle physics, weather forecasting, chemistry, and finance. They are often the best approach for solving complex problems in which problem-specific knowledge is hard to formalize.

A GO-PLAYING AI CAN REPEATEDLY APPLY ITS MCTS ALGORITHM until resources—time or memory—run out. Like many other search methods, MCTS constructs a game tree, in which each possible move creates branches of new possible moves, which are conventionally drawn pointing downward. For a basic example of this algorithm, imagine that a Go program is trying to decide on its next move. It would therefore repeat these four steps:

1. Tree descent: From the existing board position (the root



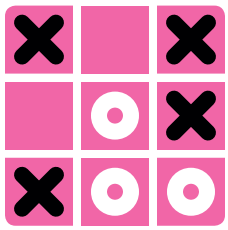
GROWING the Tree

In these four simulations of a simple Monte Carlo tree search, the program, playing as black, evaluates the winning potential of possible moves. Starting from a position to be evaluated (the leaf node), the program plays a random sequence of legal moves, playing for both black and white. It plays to the end of the game and then determines if the result is a win (1) or a loss (0). Then it discards all the information about that move sequence except for the result, which it uses to update the winning ratio for the leaf node and the nodes that came before it, back to the root of the game tree.

The GAMES Computers Play

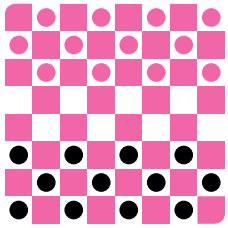
In two-player games without chance or hidden information, AIs have achieved remarkable success. However, 19-by-19 Go, with its staggering array of possible game positions, remains a challenge.

Game positions:
Computer strength:

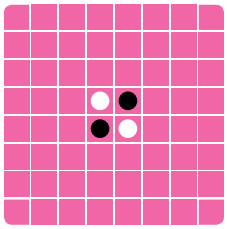


TIC-TAC-TOE
10^4
PERFECT

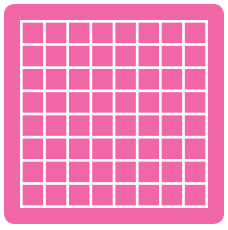
OWARE
10^{11}
PERFECT



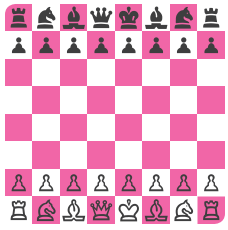
CHECKERS
10^{20}
PERFECT



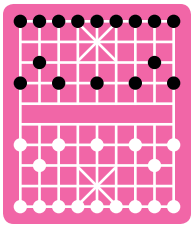
OTHELLO
10^{28}
SUPERHUMAN



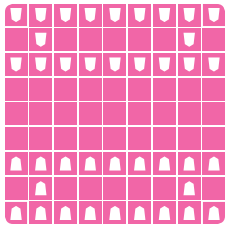
9-BY-9 GO
10^{38}
BEST PROFESSIONAL



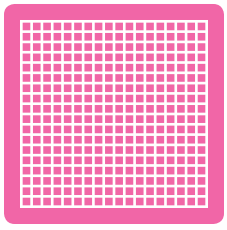
CHESS
10^{45}
SUPERHUMAN



XIANGQI
(CHINESE CHESS)
10^{48}
BEST PROFESSIONAL



SHOGI
(JAPANESE CHESS)
10^{70}
STRONG PROFESSIONAL



19-BY-19 GO
10^{172}
STRONG AMATEUR

node of the search tree), select a candidate move (a leaf node) for evaluation. At the very beginning of the search, the leaf node is directly connected to the root. Later on, as the search deepens, the program follows a long path of branches to reach the leaf node to be evaluated.

2. Simulation: From the selected leaf node, choose a random sequence of alternating moves until the end of the game is reached.

3. Evaluation and back propagation: Determine whether the simulation ends in a win or loss. Use that result to update the statistics for each node on the path from the leaf back to the root. Discard the simulation sequence from memory—only the result matters.

4. Tree expansion: Grow the game tree by adding an extra leaf node to it.

An MCTS-based program needs some intelligent way to select which branches of the game tree to grow. Good policies for doing that strike a balance between exploration (branching off nodes with few simulations and therefore high uncertainty about their prospects for leading to a win) and exploitation (pursuing moves that branch off the most promising nodes).

The best policies for expanding the tree also rely on a decision-making shortcut called rapid action value estimation (RAVE). The RAVE component tells the program to collect another set of statistics during each simulation. If the random sequence of moves results in a win, every grid point where the program placed one of its stones (thus roughly half the locations on the board) is given a numerical bonus. In this quick and dirty method, each board location accumulates a RAVE statistic as simulations are played out. Then, when the program is considering a move, it can look at both the win-loss statistic for that move as well as the RAVE statistic for that location.

These policies control the selective growth of the game tree. In typical MCTS programs, this growth is uneven: Promising lines of play are explored much more deeply than other lines. Because the search tree is grown one node at a time, the algorithm can be stopped at any time, and it will return the best move found so far.

Determining the best move is tricky, however. The most natural approach would be to pick the move with the highest

probability of leading to a win. But this is usually too risky. For example, a move with 7 wins out of 10 trials may have the highest odds of winning (70 percent), but because this number comes from only 10 trials, the uncertainty is high. A move with 65,000 wins out of 100,000 trials (65 percent) is a safer bet. This suggests a different strategy: Choose the move with the largest number of wins. And this is indeed the standard approach.

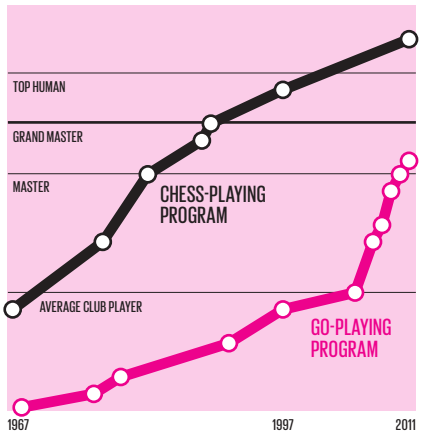
SINCE METHODS BASED ON MCTS REPLACED THE TRADITIONAL knowledge-based approaches, we have seen amazing improvements in the playing strength of Go programs. On the 9-by-9 board, top programs are on a par with the best human players. On the standard 19-by-19 board, a program called Crazy Stone has convincingly defeated a top professional while playing with a handicap of only four stones, indicating that the program plays as well as a very strong amateur.

The most basic Go-playing program using MCTS would employ only minimal knowledge of the game—namely, which moves are legal and who wins at the end of the game. This produces surprisingly successful Go-playing programs. But the latest research indicates that a little bit more knowledge can boost the performance of MCTS programs.

At the University of Alberta, we are finding ways to include some game-specific knowledge to give the program certain tendencies as it chooses its random moves. For example, a program can be biased so that its random move sequences aren't really so random. Instead, they often incorporate moves that would naturally follow from the opponent's previous move. Such obvious actions would include a move that would defend the program's stones from immediate capture, and a move that would seize an immediate opportunity to capture an opponent's stones.

The program can also be given some pieces of knowledge that can be applied without requiring it to perform true evaluations of game positions. For example, a program may have a database of simple patterns of stones that can occur within a 3-by-3 region of lines. After an opponent's move, the program checks the areas around that stone to see whether the resultant configurations match any of the stored patterns. If it does find a match, it plays the next move associated with that pattern in its database. If it finds several matching patterns, it chooses among the associated next moves at random.

When AI researchers first applied Monte Carlo methods to Go around 2005, computer Go programs improved dramatically and rapidly. Over the past few years, progress has been slower, but the research community is still optimistic. If we continue to refine our programs, enhancing the power of randomness with a dash of knowledge, we believe our AIs will eventually perform as well as Go's human grand masters.



GOING FOR IT: Chess-playing programs bested human grand masters more than a decade ago, but Go-playing programs weren't contenders until their coders embraced Monte Carlo tree search techniques in the late 2000s.

IN THE EARLY DAYS OF DEVELOPING CHESS-PLAYING PROGRAMS, researchers tried to get computers to play chess the way people do. Very quickly, it became clear that chess AIs couldn't efficiently learn and apply enough strategic knowledge to be successful. Programmers then adopted a search-intensive approach that required only enough knowledge to understand the rules and to evaluate the strength of a given board configuration. MCTS takes this one step further by questioning the need for making any such evaluations. It may seem paradoxical, but we're already seeing the benefits of such intelligence-free artificial intelligence in the game of Go—and that may be just the beginning.

In recent years, AI researchers have been trying to develop a program that can learn to play any game well—Go, tic-tac-toe, chess, whatever—given only the rules of the game as input. Historically, all the strong game-playing programs have been able to play only one specific game. They were "idiot savants" that could do one thing very well, but nothing else. If AI researchers can develop a program capable of more general learning, however, we might create a more flexible kind of computer intelligence. This would be a big step toward the real goal of artificial intelligence research: fashioning a general-purpose learner.

The AI community has been able to gauge progress in this area at the General Game Playing (GGP) competition, held at the annual conference of the Association for the Advancement of Artificial Intelligence. There, programs are given only the rules of a game and then have to play it in a tournament. From the rules, a GGP program can usually infer the appropriate search algorithm to find suitable moves. But these programs quickly run into trouble as they try to learn the game-specific knowledge that will allow them to make evaluations. One program might try to make deductions based on the rules of the game. Another might learn by playing against itself and making inferences. Yet neither strategy has proven effective. To date, there have been no truly successful approaches to machine learning in this sphere.

Instead, in recent tournaments virtually all the GGP programs have used a variation of MCTS to avoid the knowledge-acquisition problem altogether. These programs still have a long way to go. But there may come a day soon when an AI will be able to conquer any game we set it to, without a bit of knowledge to its name. If that day comes, we will raise a wry cheer for the triumph of ignorance. ■

POST YOUR COMMENTS at http://spectrum.ieee.org/go0714