can play perfectly by looking up the right move in this table. The table is constructed by **retrograde** minimax search: start by considering all ways to place the KBNK pieces on the board. Some of the positions are wins for white; mark them as such. Then reverse the rules of chess to do reverse moves rather than moves. Any move by White that, no matter what move Black responds with, ends up in a position marked as a win, must also be a win. Continue this search until all possible positions are resolved as win, loss, or draw, and you have an infallible lookup table for all endgames with those pieces. This has been done not only for KBNK endings, but for all endings with seven or fewer pieces. The tables contain 400 trillion positions. An eight-piece table would require 40 quadrillion positions.

Retrograde

5.4 Monte Carlo Tree Search

The game of Go illustrates two major weaknesses of heuristic alpha–beta tree search: First, Go has a branching factor that starts at 361, which means alpha–beta search would be limited to only 4 or 5 ply. Second, it is difficult to define a good evaluation function for Go because material value is not a strong indicator and most positions are in flux until the endgame. In response to these two challenges, modern Go programs have abandoned alpha–beta search and instead use a strategy called **Monte Carlo tree search (MCTS)**.[3]

Monte Carlo tree search (MCTS)

The basic MCTS strategy does not use a heuristic evaluation function. Instead, the value of a state is estimated as the average utility over a number of **simulations** of complete games starting from the state. A simulation (also called a **playout** or **rollout**) chooses moves first for one player, then for the other, repeating until a terminal position is reached. At that point the rules of the game (not fallible heuristics) determine who has won or lost, and by what score. For games in which the only outcomes are a win or a loss, "average utility" is the same as "win percentage."

Simulation
Playout
Rollout

How do we choose what moves to make during the playout? If we just choose randomly, then after multiple simulations we get an answer to the question "what is the best move if both players play randomly?" For some simple games, that happens to be the same answer as "what is the best move if both players play well?," but for most games it is not. To get useful information from the playout we need a **playout policy** that biases the moves towards good ones. For Go and other games, playout policies have been successfully learned from self-play by using neural networks. Sometimes game-specific heuristics are used, such as "consider capture moves" in chess or "take the corner square" in Othello.

Playout policy

Given a playout policy, we next need to decide two things: from what positions do we start the playouts, and how many playouts do we allocate to each position? The simplest answer, called **pure Monte Carlo search**, is to do N simulations starting from the current state of the game, and track which of the possible moves from the current position has the highest win percentage.

Pure Monte Carlo search

For some stochastic games this converges to optimal play as N increases, but for most games it is not sufficient—we need a **selection policy** that selectively focuses the computational resources on the important parts of the game tree. It balances two factors: **exploration** of states that have had few playouts, and **exploitation** of states that have done well in past playouts, to get a more accurate estimate of their value. (See Section 17.3 for more on the

Selection policy
Exploration
Exploitation

[3] "Monte Carlo" algorithms are randomized algorithms named after the Casino de Monte-Carlo in Monaco.