Monte-Carlo Tree Search and Rapid Action Value Estimation in Computer Go

Sylvain Gelly[1]

Université Paris Sud, LRI, CNRS, INRIA, France

David Silver

University College London, UK

Abstract

A new paradigm for search, based on Monte-Carlo simulation, has revolutionised the performance of computer Go programs. In this article we describe two extensions to the Monte-Carlo tree search algorithm, which significantly improve the effectiveness of the basic algorithm. When we applied these two extensions to the Go program *MoGo*, it became the first program to achieve *dan* (master) level at 9×9 Go. In this article we survey the Monte-Carlo revolution in computer Go, outline the key ideas that led to the success of *MoGo* and subsequent Go programs, and provide for the first time a comprehensive description, in theory and in practice, of this extended framework for Monte-Carlo tree search.

1. Introduction

Monte-Carlo tree search [1] is a new paradigm for search, which has revolutionised computer Go [2, 3], and is rapidly replacing traditional search algorithms as the method of choice in challenging domains such as General Game Playing [4], Amazons [5], Lines of Action [6], multi-player card games [7, 8], and real-time strategy games [9].

The key idea is to simulate many thousands of random games from the current position, using self-play. New positions are added into a search tree, and each node of the tree contains a value that predicts who will win from that position. These predictions are updated by Monte-Carlo simulation: the value of a node is simply the average outcome of all simulated games that visit the position. The search tree is used to guide simulations along promising paths, by selecting the child node with the highest potential value [10]. This results in a highly selective search that very quickly identifies good move sequences.

The evaluation function of Monte-Carlo tree search depends only on the observed outcomes of simulations, rather than the handcrafted evaluation functions used in traditional search algorithms. The evaluation function continues to improve from additional

[1] Now at Google, Zurich

simulations; given infinite memory and computation, it will converge on the optimal search tree [10]. Furthermore, Monte-Carlo tree search develops in a highly selective, best-first manner, expanding promising regions of the search space much more deeply.

In this article we describe two major enhancements to Monte-Carlo tree search. The first extension, the *Rapid Action Value Estimation* (RAVE) algorithm, shares the value of actions across each subtree of the search tree. RAVE forms a very fast and rough estimate of the action value; whereas normal Monte-Carlo is slower but more accurate. The MC–RAVE algorithm combines these two value estimates in a principled fashion, so as to minimise the mean squared error.

The second extension, *heuristic Monte-Carlo tree search*, uses a heuristic function to initialise the values of new positions in the search tree. We demonstrate that an effective heuristic function can be learnt by temporal-difference learning and self-play; however, in general any heuristic can be provided to the algorithm.

We applied these two extensions to the Go program *MoGo*, achieving a significant improvement to its performance in 9×9 Go. The resulting program became the first program to achieve *dan* (master) level, and the first program to defeat a human professional player. This framework for Monte-Carlo tree search is now used in a wide variety of master-level Go programs, including the first programs to achieve *dan* level at 19×19 Go.

This article provides the first comprehensive description of this extended framework for Monte-Carlo tree search. It adds new theory, results, pseudocode, and discussion to the original presentation of heuristic MC–RAVE [11, 3, 12]. In addition, we include a survey of the strongest Go programs based on prior approaches, and the strongest current programs based on Monte-Carlo methods.

2. Simulation-Based Search

2.1. Two-Player Games

We consider the class of two-player, perfect-information, zero-sum games such as chess, checkers, backgammon and Go. Without loss of generality, we call the player to move first *Black* and the player to move second *White*. Black and White alternate turns, at each turn t selecting an *action* $a_t \in \mathcal{A}(s_t)$, where $s_t \in \mathcal{S}$ is the current *state*, $\mathcal{S}$ is a finite state space, and $\mathcal{A}(s)$ is a finite set of legal actions in state s. The game finishes upon reaching a terminal state with outcome z. Black's goal is to maximise z; White's goal is to minimise z.

We define a two-player *policy* $\pi(s, a) = Pr(a|s)$ to be a stochastic action selection strategy that determines the probability of selecting actions in any given state. It consists of both a Black policy $\pi_B(s, a)$ that is used for Black moves, and a White policy $\pi_W(s, a)$ that is used for White moves, $\pi = \langle \pi_B, \pi_W \rangle$. We define the *value function* $Q^\pi(s, a)$ to be the expected outcome after playing action a in state s, and then following policy π for both players until termination,[2]

[2]In two-player games a state is usually called a *position* and an action is usually called a *move*. The goodness of positions or moves is estimated by an *evaluation function*. We use these terms during informal discussions, but use *state*, *action* and *value function* in their precise sense.

$$Q^{\pi}(s,a) = \mathbb{E}_{\pi}[z|s_t = s, a_t = a] \qquad \forall s \in \mathcal{S}, a \in \mathcal{A}(s) \tag{1}$$

The *minimax value function* $Q^*(s,a)$ is the value function that maximises Black's action value and minimises White's action value, from every state and for every action,

$$Q^*(s,a) = \max_{\pi_B} \min_{\pi_W} Q^{\pi}(s,a) \qquad \forall s \in \mathcal{S}, a \in \mathcal{A}(s) \tag{2}$$

A *minimax policy* deterministically plays Black moves so as to maximise $Q^*(s,a)$, and plays White moves to minimise $Q^*(s,a)$. This is commonly called *perfect play*.

2.2. Simulation

The basic idea of *simulation-based search* [13] is to evaluate states online from simulated games. Each simulated game, which we call a *simulation*, starts from a root state s_0, and sequentially samples states and actions, without backtracking, until the game terminates. At each step t of simulation, a *simulation policy* $\pi(s,a)$ is used to select an action, $a_t \sim \pi(s_t, \cdot)$, and the rules of the game are used to generate the next state s_{t+1}. The outcome z of each simulated game is used to update the values of states or actions encountered during that simulation.

2.3. Monte-Carlo Simulation

Monte-Carlo simulation is a simple simulation-based search algorithm for evaluating candidate actions from a root state s_0. The search proceeds by simulating complete games from s_0 until termination, using a fixed simulation policy, for example selecting actions uniformly amongst all legal moves. The value of each action a from s_0, is estimated by the mean outcome of all simulations starting with candidate action a.

Monte-Carlo simulation provides a simple method for estimating the root value $Q^{\pi}(s_0,a)$. $N(s)$ complete games are simulated by self-play with policy π from state s. The *Monte-Carlo value* (MC value) $Q(s,a)$ is the mean outcome of all simulations in which action a was selected in state s,

$$Q(s,a) = \frac{1}{N(s,a)} \sum_{i=1}^{N(s)} \mathbb{I}_i(s,a) z_i, \tag{3}$$

where z_i is the outcome of the ith simulation; $\mathbb{I}_i(s,a)$ is an indicator function returning 1 if action a was selected in state s during the ith simulation, and 0 otherwise; and $N(s,a) = \sum_{i=1}^{N(s)} \mathbb{I}_i(s,a)$ counts the total number of simulations in which action a was selected in state s.

In its most basic form, Monte-Carlo simulation is only used to evaluate actions, but not to improve the simulation policy. However, the basic algorithm can be extended by progressively favouring the most successful actions, or by progressively pruning away the least successful actions [14, 15].

In some problems, such as backgammon [16], Scrabble [17], Amazons [5] and Lines of Action [6], it is possible to construct an accurate evaluation function. In these

cases it can be beneficial to stop simulation before the end of the game, and bootstrap from the estimated value at the time of stopping. This approach, known as *truncated Monte-Carlo simulation*, both increases the simulation speed, and also reduces the variance of Monte-Carlo evaluation. In more challenging problems, such as Go [15], it is hard to construct an accurate evaluation function. In this case truncating simulations usually increases the evaluation bias more than it reduces the evaluation variance, and so it is better to simulate until termination.

2.4. Monte-Carlo Tree Search

Monte-Carlo tree search (MCTS) uses Monte-Carlo simulation to evaluate the nodes of a search tree [1]. The values in the search tree are then used to select the best action during subsequent simulations. Monte-Carlo tree search is *sequentially best-first*: it selects the best child at each step of simulation. This allows the search to continually refocus its attention, each simulation, on the highest value regions of the state space. As the search tree grows larger, the values of the nodes approximate the minimax value, and the simulation policy approximates the minimax policy.

The search tree $\mathcal{T}$ contains one node, $n(s)$, corresponding to each state s that has been seen during simulations. Each node contains a total count for the state, $N(s)$, and an action value $Q(s,a)$ and count $N(s,a)$ for each action $a \in \mathcal{A}$.

Simulations start from the root state s_0, and are divided into two stages. When state s_t is represented in the search tree, $s_t \in \mathcal{T}$, a *tree policy* is used to select actions. Otherwise, a *default policy* is used to roll out simulations to completion. The simplest version of the algorithm, which we call *greedy MCTS*, selects the greedy action with the highest value during the first stage, $\operatorname{argmax}_a Q(s_t, a)$; and selects actions uniformly at random during the second stage.

Every state and action in the search tree is evaluated by its mean outcome during simulations. After each simulation $s_0, a_0, s_1, a_1, ..., s_T$ with outcome z, each node in the search tree, $\{n(s_t) | s_t \in \mathcal{T}\}$, updates its count, and updates its action value $Q(s_t, a_t)$ to the new MC value (Equation 3). This update can also be implemented incrementally, without reconsidering previous simulations, by incrementing the count and updating the value towards the outcome z.[3]

$$N(s_t) \leftarrow N(s_t) + 1 \tag{4}$$

$$N(s_t, a_t) \leftarrow N(s_t, a_t) + 1 \tag{5}$$

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \frac{z - Q(s_t, a_t)}{N(s_t, a_t)}, \tag{6}$$

In addition, each visited node is added to the search tree. In practice, to reduce memory requirements, new nodes are not added for every simulation. Typically, just one new node is added to the search tree in each simulation. The first state encountered, that is not already represented in the tree, is added into the search tree. If memory

[3]This incremental formulation may accumulate error, and in practice it usually requires double precision.

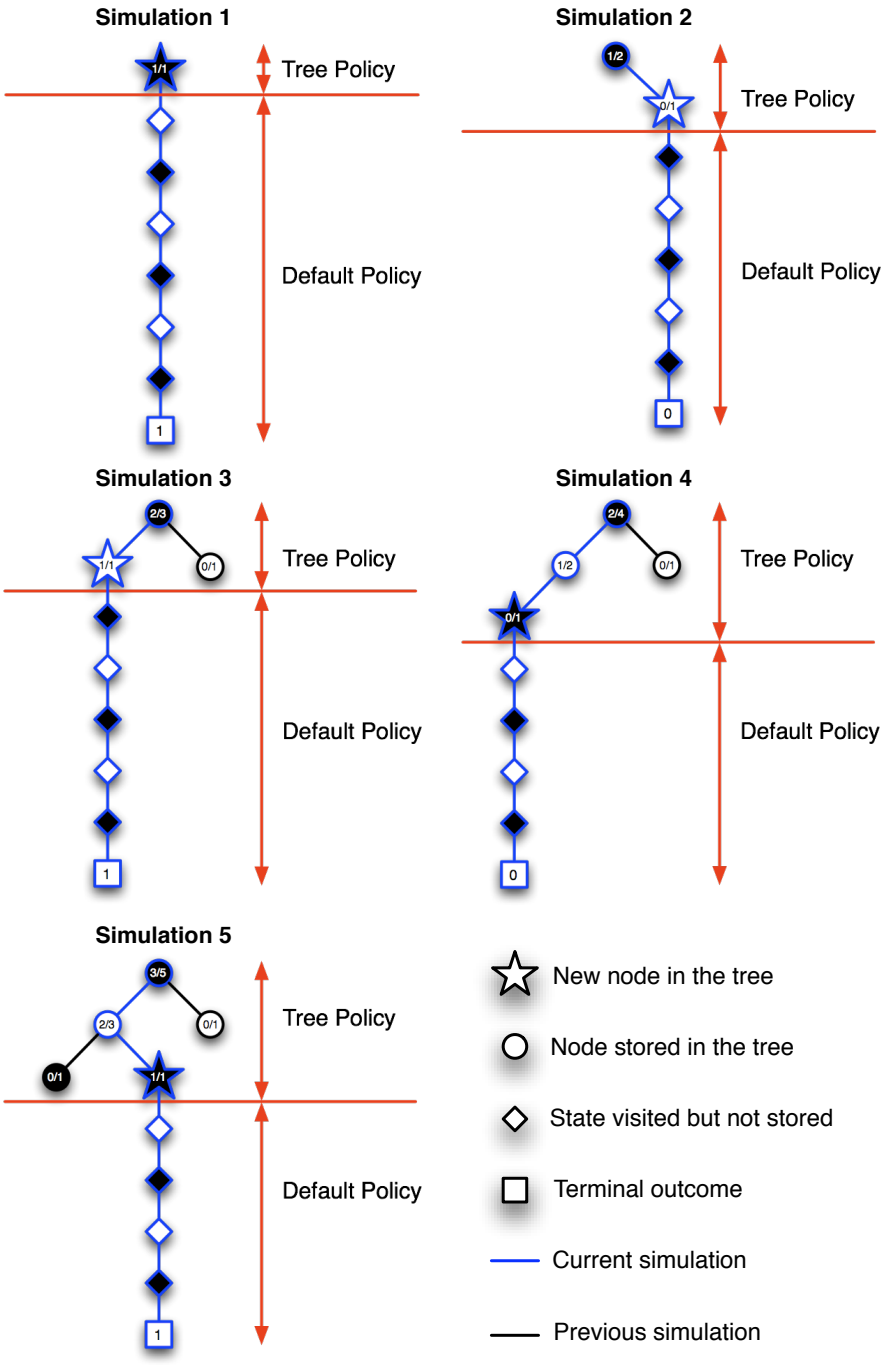

Figure 1: Five simulations of a simple Monte-Carlo tree search. Each simulation has an outcome of 1 for a black win or 0 for a white win (square). At each simulation a new node (star) is added into the search tree. The value of each node in the search tree (circles and star) is then updated to count the number of black wins, and the total number of visits (wins/visits).

limitations are still an issue, it is also possible to wait several simulations before adding a new node, or to prune old nodes as the search progresses. Figure 1 illustrates several steps of the MCTS algorithm.

It is also possible to compute other statistics by Monte-Carlo tree search, for example the max outcome, which may evaluate positions more rapidly but is also sensitive to outliers [15], or an intermediate statistic between mean and max outcome [1]. However, the mean outcome has proven to be the most robust and effective statistic in Go and other domains.

2.5. UCT

Greedy action selection can often be an inefficient way to construct a search tree, as it will typically avoid searching actions after one or more poor outcomes, even if there is significant uncertainty about the value of those actions. To explore the search tree more efficiently, the principle of *optimism in the face of uncertainty* can be applied, which favours the actions with the greatest *potential* value. To implement this principle, each action value receives a bonus that corresponds to the amount of uncertainty in the current value of that state and action.

The UCT algorithm applies this principle to Monte-Carlo tree search, by treating each state of the search tree as a multi-armed bandit, in which each action corresponds to an arm of the bandit [10].[4] The tree policy selects actions by using the *UCB1* algorithm, which maximises an upper confidence bound on the value of actions [18]. Specifically, the action value is augmented by an exploration bonus that is highest for rarely visited state-action pairs, and the tree policy selects the action a^* maximising the augmented value,

$$Q^{\oplus}(s,a) = Q(s,a) + c\sqrt{\frac{\log N(s)}{N(s,a)}} \tag{7}$$

$$a^* = \operatorname*{argmax}_a Q^{\oplus}(s,a) \tag{8}$$

where c is a scalar *exploration constant* and log is the natural logarithm. Pseudocode for the UCT algorithm is given in Algorithm 1.

UCT is proven to converge on the minimax action value function [10]. As the number of simulations N grows to infinity, the root values converge in probability to the minimax action values, $\forall a \in \mathcal{A}, \operatorname{plim}_{n\to\infty} Q(s_0,a) = Q^*(s_0,a)$. Furthermore, the bias of the root values, $\mathbb{E}[Q(s_0,a) - Q^*(s_0,a)]$, is $O(\log(n)/n)$, and the probability of selecting a suboptimal action, $Pr(\operatorname*{argmax}_{a\in\mathcal{A}} Q(s_0,a) \neq \operatorname*{argmax}_{a\in\mathcal{A}} Q^*(s_0,a))$, converges to zero at a polynomial rate.

The performance of UCT can often be significantly improved by incorporating domain knowledge into the default policy [19, 20]. The UCT algorithm, using a carefully

[4] In fact, the search tree is not a true multi-armed bandit, as there is no real cost to exploration during planning. In addition the simulation policy continues to change as the search tree is updated, which means that the payoff is non-stationary.

Algorithm 1 Two Player UCT

procedure UCTSEARCH(s_0)
  while time available **do**
    SIMULATE($board, s_0$)
  end while
  $board.SetPosition(s_0)$
  return SELECTMOVE($board, s_0, 0$)
end procedure

procedure SIMULATE($board, s_0$)
  $board.SetPosition(s_0)$
  $[s_0, ..., s_T]$ = SIMTREE($board$)
  z = SIMDEFAULT($board$)
  BACKUP($[s_0, ..., s_T], z$)
end procedure

procedure SIMTREE($board$)
  $c = exploration\ constant$
  $t = 0$
  while not $board.GameOver()$ **do**
    $s_t = board.GetPosition()$
    if $s_t \notin tree$ **then**
      NEWNODE(s_t)
      return $[s_0, ..., s_t]$
    end if
    a = SELECTMOVE($board, s_t, c$)
    $board.Play(a)$
    $t = t + 1$
  end while
  return $[s_0, ..., s_{t-1}]$
end procedure

procedure SIMDEFAULT($board$)
  while not $board.GameOver()$ **do**
    a = DEFAULTPOLICY($board$)
    $board.Play(a)$
  end while
  return $board.BlackWins()$
end procedure

procedure SELECTMOVE($board, s, c$)
  $legal = board.Legal()$
  if $board.BlackToPlay()$ **then**
    $a^* = \underset{a \in legal}{\text{argmax}} \left(Q(s,a) + c\sqrt{\frac{\log N(s)}{N(s,a)}} \right)$
  else
    $a^* = \underset{a \in legal}{\text{argmin}} \left(Q(s,a) - c\sqrt{\frac{\log N(s)}{N(s,a)}} \right)$
  end if
  return a^*
end procedure

procedure BACKUP($[s_0, ..., s_T], z$)
  for $t = 0$ **to** T **do**
    $N(s_t) = N(s_t) + 1$
    $N(s_t, a_t)$ ++
    $Q(s_t, a_t)$ += $\frac{z - Q(s_t, a_t)}{N(s_t, a_t)}$
  end for
end procedure

procedure NEWNODE(s)
  $tree.Insert(s)$
  $N(s) = 0$
  for all $a \in \mathcal{A}$ **do**
    $N(s, a) = 0$
    $Q(s, a) = 0$
  end for
end procedure

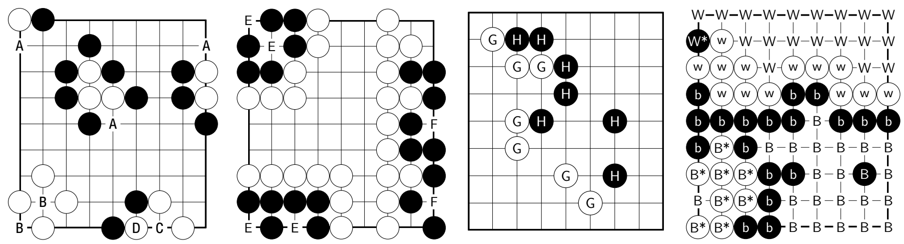

Figure 2: a) The White stones are in *atari* and can be captured by playing at the points marked A. It is illegal for Black to play at B, as the stone would have no liberties. Black may, however, play at C to capture the stone at D. It is illegal for White to recapture immediately by playing at D, as this would repeat the position - it is a *ko*. b) The points marked E are *eyes* for Black. The black groups on the left can never be captured by White, they are *alive*. The points marked F are *false eyes*: the black stones on the right will eventually be captured by White and are *dead*. c) *Groups* of loosely connected white stones (G) and black stones (H). d) A final position. Dead stones ($B*, W*$) are removed from the board. All surrounded intersections (B, W) and all remaining stones (b, w) are counted for each player. If *komi* is 6.5 then Black wins by 8.5 points in this example.

chosen default policy, has outperformed previous approaches to search in a variety of challenging games, including Go [19], General Game Playing [4], Amazons [5], Lines of Action [6], multi-player card games [7, 8], and real-time strategy games [9]. Much additional research in Monte-Carlo tree search has been developed in the context of computer Go, and is discussed in more detail in the next section.

3. Computer Go

For many years, computer chess was considered to be "the *drosophila* of AI",[5] and a "grand challenge task" [21]. It provided a sandbox for new ideas, a straightforward performance comparison between algorithms, and measurable progress against human capabilities. With the dominance of alpha-beta search programs over human players now conclusive in chess [22], many researchers have sought out a new challenge. Computer Go has emerged as the "new *drosophila* of AI" [21], a "task *par excellence*" [23], and "a grand challenge task for our generation" [24].

Go has more than 10^{170} states and up to 361 legal moves. Its enormous search space is orders of magnitude too big for the alpha-beta search algorithms that have proven so successful in chess and checkers. Although the rules are simple, the emergent complexity of the game is profound. The long-term effect of a move may only be revealed after 50 or 100 additional moves. Professional Go players accumulate Go knowledge over a lifetime; mankind has accumulated Go knowledge over several millennia. For the last 30 years, attempts to encode this knowledge in machine usable form have led to a positional understanding that is at best comparable to weak amateur-level humans.

Figure 3: Performance ranks in Go, in increasing order of strength from left to right.

3.1. The Rules of Go

The game of Go is usually played on a 19×19 grid, with 13×13 and 9×9 as popular alternatives. Black and White play alternately, placing a single stone on an intersection of the grid. Stones cannot be moved once played, but may be captured. Sets of adjacent, connected stones of one colour are known as *blocks*. The empty intersections adjacent to a block are called its *liberties*. If a block is reduced to zero liberties by the opponent, it is captured and removed from the board (Figure 2a, A). Stones with just one remaining liberty are said to be in *atari*. Playing a stone with zero liberties is illegal (Figure 2a, B), unless it also reduces an opponent block to zero liberties. In this case the opponent block is captured, and the player's stone remains on the board (Figure 2a, C). Finally, repeating a previous board state is illegal.[6] A situation in which a repeat could otherwise occur is known as *ko* (Figure 2a, D).

A connected set of empty intersections that is wholly enclosed by stones of one colour is known as an *eye*. One natural consequence of the rules is that a block with two eyes can never be captured by the opponent (Figure 2b, E). Blocks which cannot be captured are described as *alive*; blocks which will certainly be captured are described as *dead* (Figure 2b, F). A loosely connected set of stones is described as a *group* (Figure 2c, G, H). Determining the life and death status of a group is a fundamental aspect of Go strategy.

The game ends when both players pass. Dead blocks are removed from the board (Figure 2d, $B*, W*$). In Chinese rules, all alive stones, and all intersections that are enclosed by a player, are counted as a point of *territory* for that player (Figure 2d, B, W).[7] Black always plays first in Go; White receives compensation, known as *komi*, for playing second. The winner is the player with the greatest territory, after adding *komi* for White.

3.2. Go Ratings

Human Go players are rated on a three-class scale, divided into *kyu* (beginner), *dan* (master), and *professional dan* ranks (see Figure 3). *Kyu* ranks are in descending order of strength, whereas *dan* and *professional dan* ranks are in ascending order. At amateur level, the difference in rank corresponds to the number of handicap stones required by the weaker player to ensure an even game.[8]

[5] *Drosophila* is the fruit fly, the most extensively studied organism in genetics research.
[6] The exact definition of repeating differs subtly between different rule sets.
[7] The Japanese scoring system is somewhat different, but usually has the same outcome.
[8] The difference between 1 *kyu* and 1 *dan* is normally considered to be 1 stone.

The majority of computer Go programs compete on the Computer Go Server (CGOS). This server runs an ongoing rapid-play tournament of 5 minute games for 9×9 and 20 minute games for 19×19 boards. The Elo rating of each program on the server is continually updated. The Elo scale on CGOS assumes a logistic distribution with winning probability $Pr(A \text{ beats } B) = \frac{1}{1+10^{\frac{\mu_B - \mu_A}{400}}}$, where μ_A and μ_B are the Elo ratings for player A and player B respectively. On this scale, a difference of 200 Elo corresponds to a 75% winning rate for the stronger player, and a difference of 500 Elo corresponds to a 95% winning rate. Following convention, the open source Go program *GnuGo* (level 10) anchors this scale with a rating of 1800 Elo.

3.3. Handcrafted Heuristics

In many other classic games, handcrafted heuristic functions have proven highly effective. Basic heuristics such as *material count* and *mobility*, which provide reasonable estimates of goodness in checkers, chess and Othello [25], are next to worthless in Go. Stronger heuristics have proven surprisingly hard to design, despite several decades of endeavour [26].

Until recently, most Go programs incorporated very large quantities of expert knowledge, in a *pattern database* containing many thousands of manually inputted patterns, and typically including expert knowledge such as *fuseki* (opening patterns), *joseki* (corner patterns), and *tesuji* (tactical patterns). Traditional Go programs use these databases to generate plausible moves that match one or more patterns. The pattern database accounts for a large part of the development effort in a traditional Go program, sometimes requiring many man-years of effort from expert Go players.

The Many Faces of Go[9] uses local alpha-beta searches to determine the life or death status of blocks and groups. A global alpha-beta search is used to evaluate full-board positions, using a heuristic function of the local search results. Pattern databases are used to generate moves in both the local and global searches. The program *GnuGo*[10] uses pattern databases and specialised search routines to determine local subgoals such as capture, connection, and eye formation. The local status of each subgoal is used to estimate the overall benefit of each legal move.

3.4. Reinforcement Learning in Go

Reinforcement learning can be used to train a value function that predicts the eventual outcome of the game from a given state. The learning program can be rewarded by the score at the end of the game, or by a reward of 1 if Black wins and 0 if White wins. Surprisingly, the less informative binary signal has proven more successful [1], as it encourages the agent to favour risky moves when behind, and calm moves when ahead. Expert Go players will frequently play to minimise the uncertainty in a position once they judge that they are ahead in score; this behaviour cannot be replicated by simply maximising the expected score. Despite this shortcoming, the final score has been widely used as a reward signal [27, 28, 29, 30].

[9] http://www.smart-games.com/manyfaces.html
[10] http://www.gnu.org/software/gnugo

Schraudolph et al. [27] exploit the symmetries of the Go board in a convolutional neural network. The network predicts the final territory status of a particular target intersection. It receives one input from each intersection ($-1, 0$ or $+1$ for White, Empty and Black respectively) in a local region around the target, and outputs the predicted territory for the target intersection. The global position is evaluated by summing the territory predictions for all intersections on the board. Weights are shared between rotationally and reflectionally symmetric patterns of input features, and between all target intersections. They train their multilayer perceptron using $TD(0)$, using a reward signal corresponding to the final territory value of the intersection. The network outperformed a commercial Go program, *The Many Faces of Go*, when set to a low playing level in 9×9 Go, after just 3,000 self-play training games.

Dahl's *Honte* [29] and Enzenberger's *NeuroGo III* [30] use a similar approach to predicting the final territory. However, both programs learn intermediate features that are used to input additional knowledge into the territory evaluation network. *Honte* has one intermediate network to predict local moves and a second network to evaluate the life and death status of groups. *NeuroGo III* uses intermediate networks to evaluate connectivity and eyes. Both programs achieved single-digit *kyu* ranks; *NeuroGo* won the silver medal at the 2003 9×9 Computer Go Olympiad.

RLGO 1.0 [31] uses a simpler but more computationally efficient approach to reinforcement learning. It uses a million *local shape features* to enumerate all possible 1×1, 2×2 and 3×3 configurations of Black, White and empty intersections, at every possible location on the board. The value of a state is estimated by a linear combination of the local shape features that are matched in that state. The weights of these features are trained offline by temporal-difference learning from games of self-play, and sharing weights between symmetric local shape features. The basic version of *RLGO* was rated at 1350 Elo on the 9×9 Computer Go Server.

RLGO 2.4 [32, 13] applies the same reinforcement learning approach *online*. It applies temporal-difference learning to simulated games of self-play that start from the current state: a form of simulation-based search. At every move, the value function is re-trained in real-time, specialising on the tactics and strategies that are most relevant to the current position. This approach boosted *RLGO's* rating to 2100 Elo on CGOS, outperforming traditional Go programs and resulting in the strongest 9×9 Go program not based on Monte-Carlo tree search.

3.5. Monte Carlo Simulation in Go

In contrast to traditional search methods, Monte-Carlo simulation evaluates the current position dynamically, rather than storing knowledge about all positions in a static evaluation function. This makes it an appealing choice for Go, where, as we have seen, the number of possible positions is particularly large, and position evaluation is particularly challenging.

The first Monte-Carlo Go program, *Gobble* [33], simulated many games of self-play from the current state s. It combined Monte-Carlo evaluation with two novel ideas: the *all-moves-as-first* heuristic, and *ordered simulation*. The all-moves-as-first heuristic assumes that the value of a move is not significantly affected by changes elsewhere on the board. The value of playing action a *immediately* is estimated by the

average outcome of all simulations in which action a is played *at any time*. We formalise this idea more precisely in Section 4.1. *Gobble* also used ordered simulation to sort all moves according to their estimated value. This ordering is randomly perturbed according to an annealing schedule that cools down with additional simulations. Each simulation then plays out all moves in the prescribed order. *Gobble* itself played weakly, with an estimated rating of around 25 *kyu*.

Bouzy and Helmstetter developed the first competitive Go programs based on Monte-Carlo simulation [15]. Their basic framework simulates many games of self-play from the current state s, for each candidate action a, using a uniform random simulation policy; the value of a is estimated by the average outcome of these simulations. The only domain knowledge is to prohibit moves within eyes; this ensures that games terminate within a reasonable timeframe. Bouzy and Helmstetter also investigated a number of extensions to Monte-Carlo simulation, several of which are precursors to the more sophisticated algorithms used now:

1. *Progressive pruning* is a technique in which statistically inferior moves are removed from consideration [34].
2. The *all-moves-as-first heuristic*, described above.
3. The *temperature* heuristic uses a softmax simulation policy to bias the random moves towards the strongest evaluations. The softmax policy selects actions with a probability $\pi(s,a) = \frac{e^{Q(s,a)/\tau}}{\sum_{b \in legal} e^{Q(s,b)/\tau}}$, where τ is a constant temperature parameter controlling the overall level of randomness.[11]
4. The *minimax enhancement* constructs a full width search tree, and separately evaluates each node of the search tree by Monte-Carlo simulation. Selective search enhancements were also tried [35].

Bouzy also tracked statistics about the final territory status of each intersection after each simulation [36]. This information is used to influence the simulations towards disputed regions of the board, by avoiding playing on intersections which are consistently one player's territory. Bouzy also incorporated pattern knowledge into the simulation player [20]. Using these enhancements his program *Indigo* won the bronze medal at the 2004 and 2006 19×19 Computer Go Olympiads.

It is surprising that a Monte-Carlo technique, originally developed for stochastic games such as backgammon [16], Poker [14] and Scrabble [17] should succeed in Go. Why should an evaluation that is based on random play provide any useful information in the precise, deterministic game of Go? The answer, perhaps, is that Monte-Carlo methods successfully manage the uncertainty in the evaluation. A random simulation policy generates a broad distribution of simulated games, representing many possible futures and the uncertainty in what may happen next. As the search proceeds and more information is accrued, the simulation policy becomes more refined, and the distribution of simulated games narrows. In contrast, deterministic play represents perfect confidence in the future: there is only one possible continuation. If this confidence is

[11] Gradually reducing the temperature, as in simulated annealing, was not beneficial.

misplaced, then predictions based on deterministic play will be unreliable and misleading. Abramson [37] was the first to demonstrate that the expected value of a game's outcome under random play is a powerful heuristic for position evaluation in deterministic games.

3.6. Monte-Carlo Tree Search in Go

Monte-Carlo tree search was first introduced in the Go program *Crazy Stone* [1]. The Monte-Carlo value of each action is assumed to be normally distributed about the minimax value, $Q(s,a) \sim \mathcal{N}(Q^*(s,a), \sigma^2(s,a))$. During the first stage of simulation, the tree policy selects each action according to the estimated probability that its minimax value is better than the Monte-Carlo value of the best action a^*, $\pi(s,a) \approx Pr(Q^*(s,a) > Q(s,a^*))$. During the second stage of simulation, the default policy selects moves with a probability proportional to a handcrafted *urgency* heuristic. Using these techniques, *Crazy Stone* exceeded 1800 Elo on CGOS, achieving equivalent performance to traditional Go programs such as *GnuGo* and *The Many Faces of Go*. *Crazy Stone* won the gold medal at the 2006 9×9 Computer Go Olympiad.

The Go program *MoGo* introduced the UCT algorithm to computer Go [19, 38]. Instead of the Gaussian approximation used in *Crazy Stone*, *MoGo* treats each state in the search tree as a multi-armed bandit. There is one arm of the bandit for each legal move, and the payoff from an arm is the outcome of a simulation starting with that move. During the first stage of simulation, the tree policy selects actions using the UCB1 algorithm. During the second stage of simulation, *MoGo* uses a default policy based on specialised domain knowledge. Unlike the enormous pattern databases used in traditional Go programs, *MoGo*'s patterns are extremely simple. Rather than suggesting the best move in any situation, these patterns are intended to produce local sequences of plausible moves. They can be summarised by applying four prioritised rules after any opponent move a:

1. If a put some of our stones into atari, play a saving move at random.
2. Otherwise, if one of the 8 intersections surrounding a matches a simple pattern for cutting or *hane*, randomly play one.
3. Otherwise, if any opponent stone can be captured, play a capturing move at random.
4. Otherwise play a random move.

The default policy used by *MoGo* is handcrafted. In contrast, a second version of *Crazy Stone* uses supervised learning to train the pattern weights for its default policy [2]. The relative strength of patterns is estimated by assigning an Elo rating to each pattern, much like a tournament between games players. In this approach, the pattern selected by a human player is considered to have won against all alternative patterns. *Crazy Stone* uses the minorisation-maximisation algorithm to estimate the Elo rating of simple 3×3 patterns and features. The default policy selected actions with a probability proportional to the matching pattern strengths. A more complicated set of 17,000 patterns, harvested from expert games, was used to progressively widen the search tree.

Using the UCT algorithm, *MoGo* and *Crazy Stone* significantly outperformed all previous 9×9 Go programs, and beginning a new era in computer Go.

4. Rapid Action Value Estimation

Monte-Carlo tree search separately estimates the value of each state and each action in the search tree. As a result, it cannot generalise between related positions or related moves. To determine the best move, many simulations must be performed from all states and for all actions. The RAVE algorithm uses the *all-moves-as-first* heuristic, from each node of the search tree, to estimate the value of each action. RAVE provides a simple way to share knowledge between related nodes in the search tree, resulting in a rapid, but biased estimate of the action values. This biased estimate can often determine the best move after just a handful of simulations, and can be used to significantly improve the performance of the search algorithm.

4.1. All-Moves-As-First

In incremental games such as computer Go, the value of a move is often unaffected by moves played elsewhere on the board. The underlying idea of the *all-moves-as-first* (AMAF) heuristic [33] (see Section 3.5) is to have one general value for each move, regardless of when it is played. We define the AMAF value function $\tilde{Q}^\pi(s,a)$ to be the expected outcome z from state s, when following joint policy π for both players, given that action a was selected at some subsequent turn,

$$\tilde{Q}^\pi(s,a) = \mathbb{E}_\pi[z|s_t = s, \exists u \geq t \text{ s.t. } a_u = a] \tag{9}$$

The AMAF value function provides a biased estimate of the true action value function. The level of bias, $\tilde{B}(s,a)$, depends on the particular state s and action a,

$$\tilde{Q}^\pi(s,a) = Q^\pi(s,a) + \tilde{B}(s,a) \tag{10}$$

Monte-Carlo simulation can be used to approximate $\tilde{Q}^\pi(s,a)$. The *all-moves-as-first value* $\tilde{Q}(s,a)$ is the mean outcome of all simulations in which action a is selected *at any turn* after s is encountered,

$$\tilde{Q}(s,a) = \frac{1}{\tilde{N}(s,a)} \sum_{i=1}^{N(s)} \tilde{\mathbb{I}}_i(s,a) z_i, \tag{11}$$

where $\tilde{\mathbb{I}}_i(s,a)$ is an indicator function returning 1 if state s was encountered at any step t of the ith simulation, and action a was selected at any step $u >= t$, or 0 otherwise; and $\tilde{N}(s,a) = \sum_{i=1}^{N(s)} \tilde{\mathbb{I}}_i(s,a)$ counts the total number of simulations used to estimate the AMAF value. Note that Black moves and White moves are considered to be distinct actions, even if they are played at the same intersection.

In order to select the best move with reasonable accuracy, Monte-Carlo simulation requires many simulations from every candidate move. The AMAF heuristic provides orders of magnitude more information: every move will typically have been tried on several occasions, after just a handful of simulations. If the value of a move really is unaffected, at least approximately, by moves played elsewhere, then this can result in a much faster rough estimate of the value.

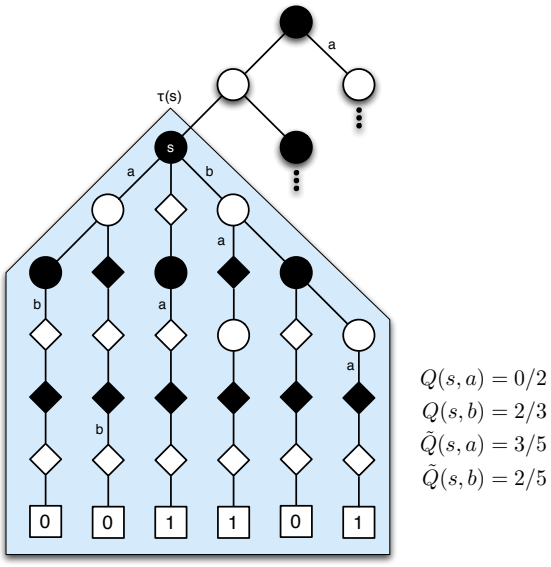

Figure 4: An example of using the RAVE algorithm to estimate the value of Black moves a and b from state s. Six simulations have been executed from state s, with outcomes shown in the bottom squares. Playing move a immediately led to two losses, and so Monte-Carlo estimation favours move b. However, playing move a at *any* subsequent time led to three wins out of five, and so the RAVE algorithm favours move a. Note that the simulation starting with move a from the root node does not belong to the subtree $\tau(s)$ and does not contribute to the AMAF estimate $\tilde{Q}(s,a)$.

4.2. RAVE

The RAVE algorithm combines Monte-Carlo tree search with the all-moves-as-first heuristic. Instead of computing the MC value (Equation 3) of each node of the search-tree, $(s,a) \in \mathcal{T}$, the AMAF value (Equation 11) of each node is computed.

Every state in the search tree, $s \in \mathcal{T}$, is the root of a subtree $\tau(s) \subseteq \mathcal{S}$. If a simulation visits state s_t at step t, then all subsequent states visited in that simulation, s_u such that $u \geq t$, are in the subtree of s_t, $s_u \in \tau(s_t)$. This includes all states $s_u \notin \mathcal{T}$ visited by the default policy in the second stage of simulation.

The basic idea of RAVE is to generalise over subtrees. The assumption is that the value of action a in state s will be similar from all states within subtree $\tau(s)$. Thus, the value of a is estimated from all simulations starting from s, regardless of exactly when a is played.

When the AMAF values are used to select an action a_t in state s_t, the action with maximum AMAF value in subtree $\tau(s_t)$ is selected, $a_t = \operatorname{argmax}_b \tilde{Q}(s_t, b)$. In principle it is also possible to incorporate the AMAF values, $\tilde{Q}(s_k, \cdot)$, from ancestor subtrees, $\tau(s_k)$ such that $k < t$. However, in our experiments, combining ancestor AMAF values did not appear to confer any advantage.

RAVE is closely related to the *history heuristic* in alpha-beta search [39]. During

the depth-first traversal of the search tree, the history heuristic remembers the success[12] of each move at various depths; the most successful moves are tried first in subsequent positions. RAVE is similar, but because it is a best-first not depth-first search, it must store values for each subtree. In addition, RAVE takes account of the success of moves made outside of the search tree by the default policy.

4.3. MC–RAVE

The RAVE algorithm learns very quickly, but it is often wrong. The principal assumption of RAVE, that a particular move has the same value across an entire subtree, is frequently violated. There are many situations, for example during tactical battles, in which nearby changes can completely change the value of a move: sometimes rendering it redundant; sometimes making it even more vital. Even distant moves can significantly affect the value of a move, for example playing a ladder-breaker in one corner can radically alter the value of playing a ladder in the opposite corner.

The *MC–RAVE* algorithm overcomes this issue, by combining the rapid learning of the RAVE algorithm with the accuracy and convergence guarantees of Monte-Carlo tree search.

There is one node $n(s)$ for each state s in the search tree. Each node contains a total count $N(s)$, and for each $a \in \mathcal{A}$, an MC value $Q(s,a)$, AMAF value $\tilde{Q}(s,a)$, MC count $N(s,a)$, and AMAF count $\tilde{N}(s,a)$.

To estimate the overall value of action a in state s, we use a weighted sum $Q_\star(s,a)$ of the MC value $Q(s,a)$ and the AMAF value $\tilde{Q}(s,a)$,

$$Q_\star(s,a) = (1 - \beta(s,a))Q(s,a) + \beta(s,a)\tilde{Q}(s,a) \tag{12}$$

where $\beta(s,a)$ is a weighting parameter for state s and action a. It is a function of the statistics for (s,a) stored in node $n(s)$, and provides a schedule for combining the MC and AMAF values. When only a few simulations have been seen, we weight the AMAF value more highly, $\beta(s,a) \approx 1$. When many simulations have been seen, we weight the Monte-Carlo value more highly, $\beta(s,a) \approx 0$.

As with Monte-Carlo tree search, each simulation is divided into two stages. During the first stage, for states within the search tree, $s_t \in \mathcal{T}$, actions are selected greedily, so as to maximise the combined MC and AMAF value, $a = \operatorname{argmax}_b Q_\star(s_t, b)$. During the second stage of simulation, for states beyond the search tree, $s_t \notin \mathcal{T}$, actions are selected by a default policy.

After each simulation $s_0, a_0, s_1, a_1, ..., s_T$ with outcome z, both the MC and AMAF values are updated. For every state s_t in the simulation that is represented in the search tree, $s_t \in \mathcal{T}$, the values and counts of the corresponding node $n(s_t)$ are updated,

[12] A successful move in alpha-beta either causes a cut-off, or has the best minimax value.

$$N(s_t) \leftarrow N(s_t) + 1 \tag{13}$$

$$N(s_t, a_t) \leftarrow N(s_t, a_t) + 1 \tag{14}$$

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \frac{z - Q(s_t, a_t)}{N(s_t, a_t)} \tag{15}$$

In addition, the AMAF value is updated for every subtree. For every state s_t in the simulation that is represented in the tree, $s_t \in \mathcal{T}$, and for every subsequent action of the simulation a_u with the same colour to play, *i.e.* $u \geq t$ and $t = u \mod 2$, the AMAF value of (s_t, a_u) is updated according to the simulation outcome z,

$$\tilde{N}(s_t, a_u) \leftarrow \tilde{N}(s_t, a_u) + 1 \tag{16}$$

$$\tilde{Q}(s_t, a_u) \leftarrow \tilde{Q}(s_t, a_u) + \frac{z - \tilde{Q}(s_t, a_u)}{\tilde{N}(s_t, a_u)} \tag{17}$$

If multiple moves are played at the same intersection during a simulation, then this update is only performed for the first move at the intersection. If an action a_u is legal in state s_u, but illegal in state s_t, then no update is performed for this move.

4.4. UCT–RAVE

The UCT algorithm extends Monte-Carlo tree search to use the optimism-in-the-face-of-uncertainty principle, by incorporating a bonus based on an upper confidence bound of the current value. Similarly, the MC–RAVE algorithm can also incorporate an exploration bonus,

$$Q^{\oplus}_{\star}(s, a) = Q_{\star}(s, a) + c\sqrt{\frac{\log N(s)}{N(s, a)}}, \tag{18}$$

Actions are then selected during the first stage of simulation to maximise the augmented value, $a = \operatorname{argmax}_b Q^{\oplus}_{\star}(s, b)$. We call this algorithm *UCT–RAVE*.[13]

If the schedule decreases to zero in all nodes, $\forall s \in \mathcal{T}, a \in \mathcal{A}$, $\lim_{N \to \infty} \beta(s, a) = 0$, then the asymptotic behaviour of UCT–RAVE is equivalent to UCT. The asymptotic convergence properties of UCT (see Section 2) therefore also apply to UCT–RAVE. We now describe two different schedules which have this property.

4.5. Hand-Selected Schedule

One hand-selected schedule for MC–RAVE uses an *equivalence parameter* k,

[13] The original UCT-RAVE algorithm also included the RAVE count in the exploration term [11]. However, it is hard to justify explicit RAVE exploration: many actions will be evaluated by AMAF, regardless of which action is actually selected at turn t.

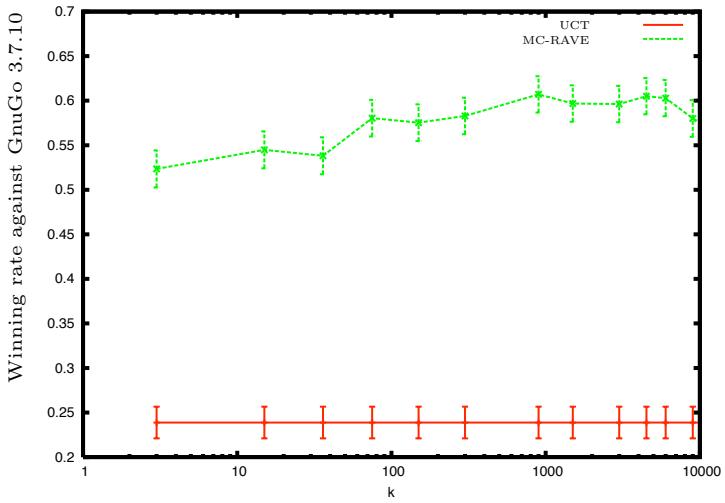

Figure 5: Winning rate of MC–RAVE with 3,000 simulations per move against GnuGo 3.7.10 (level 10) in 9×9 Go, for different settings of the equivalence parameter k. The bars indicate the standard error. Each point of the plot is an average over 2300 complete games.

$$\beta(s,a) = \sqrt{\frac{k}{3N(s)+k}} \tag{19}$$

where k specifies the number of simulations at which the Monte-Carlo value and the AMAF value should be given equal weight, $\beta(s,a) = \frac{1}{2}$,

$$\frac{1}{2} = \sqrt{\frac{k}{3N(s)+k}} \tag{20}$$

$$\frac{1}{4} = \frac{k}{3N(s)+k} \tag{21}$$

$$k = N(s) \tag{22}$$

We tested MC–RAVE in the Go program *MoGo*, using the hand-selected schedule in Equation (19) and the default policy described in [19], for different settings of the equivalence parameter k. For each setting, we played a 2300 game match against GnuGo 3.7.10 (level 10). The results are shown in Figure 5, and compared to Monte-Carlo tree search, using 3,000 simulations per move for both algorithms. The winning rate using MC–RAVE varied between 50% and 60%, compared to 24% without RAVE. Maximum performance is achieved with an equivalence parameter of 1,000 or more, indicating that the rapid action value estimate is more reliable than standard Monte-Carlo simulation until several thousand simulations have been executed from state s.

4.6. Minimum MSE Schedule

The schedule presented in Equation 19 is somewhat heuristic in nature. We now develop a more principled schedule, which selects $\beta(s,a)$ so as to minimise the mean squared error in the combined estimate $Q_\star(s,a)$.

4.6.1. Assumptions

To derive our schedule, we make a simplified statistical model of MC–RAVE. Our first assumption is that the policy π is held constant. Under this assumption, the outcome of each Monte-Carlo simulation, when playing action a from state s, is an independent and identically distributed (i.i.d.) Bernoulli random variable. Furthermore, the outcome of each AMAF simulation, when playing action a at any turn following state s, is also an i.i.d. Bernoulli random variable,

$$Pr(z=1|s_t=s,a_t=a) = Q^\pi(s,a) \tag{23}$$
$$Pr(z=0|s_t=s,a_t=a) = 1-Q^\pi(s,a) \tag{24}$$
$$Pr(z=1|s_t=s,\exists u \geq t \text{ s.t. } a_u=a) = \tilde{Q}^\pi(s,a) \tag{25}$$
$$Pr(z=0|s_t=s,\exists u \geq t \text{ s.t. } a_u=a) = 1-\tilde{Q}^\pi(s,a) \tag{26}$$

It follows that the total number of wins, after $N(s,a)$ simulations in which action a was played from state s, is binomially distributed. Similarly, the total number of wins, after $\tilde{N}(s,a)$ simulations in which action a was played at any turn following state s, is binomially distributed,

$$N(s,a)Q(s,a) \sim Binomial(N(s,a), Q^\pi(s,a)) \tag{27}$$
$$\tilde{N}(s,a)\tilde{Q}(s,a) \sim Binomial(\tilde{N}(s,a), \tilde{Q}^\pi(s,a)) \tag{28}$$

Our second assumption is that these two distributions are independent, so that the MC and AMAF values are uncorrelated. In fact, the same simulations used to compute the MC value are also used to compute the AMAF value, which means that the values are certainly correlated. Furthermore, as the tree develops over time, the simulation policy changes. This means that outcomes are not i.i.d. and that the total number of wins is not in fact binomially distributed. Nevertheless, we believe that these simplifications do not significantly affect the performance of the schedule in practice.

4.6.2. Derivation

To simplify our notation, we consider a single state s and action a. We denote the number of Monte-Carlo simulations by $n = N(s,a)$ and the number of simulations used to compute the AMAF value by $\tilde{n} = \tilde{N}(s,a)$, and abbreviate the schedule by $\beta = \beta(s,a)$. We denote the estimated mean, bias (with respect to $Q^\pi(s,a)$) and variance of the MC, AMAF and combined values respectively by $\mu, \tilde{\mu}, \mu_\star$; $b, \tilde{b}, b_\star$ and $\sigma^2, \tilde{\sigma}^2, \sigma_\star^2$, and the mean squared error of the combined value by $e_\star^2$,

$$\mu = Q(s,a) \tag{29}$$
$$\tilde{\mu} = \tilde{Q}(s,a) \tag{30}$$
$$\mu_\star = Q_\star(s,a) \tag{31}$$
$$b = Q^\pi(s,a) - Q^\pi(s,a) = 0 \tag{32}$$
$$\tilde{b} = \tilde{Q}^\pi(s,a) - Q^\pi(s,a) = \tilde{B}(s,a) \tag{33}$$
$$b_\star = Q^\pi_\star(s,a) - Q^\pi(s,a) \tag{34}$$
$$\sigma^2 = \mathbb{E}[(Q(s,a) - Q^\pi(s,a))^2 | N(s,a) = n] \tag{35}$$
$$\tilde{\sigma}^2 = \mathbb{E}[(\tilde{Q}(s,a) - \tilde{Q}^\pi(s,a))^2 | \tilde{N}(s,a) = \tilde{n}] \tag{36}$$
$$\sigma^2_\star = \mathbb{E}[(Q_\star(s,a) - Q^\pi_\star(s,a))^2 | N(s,a) = n, \tilde{N}(s,a) = \tilde{n}] \tag{37}$$
$$e^2_\star = \mathbb{E}[(Q_\star(s,a) - Q^\pi(s,a))^2 | N(s,a) = n, \tilde{N}(s,a) = \tilde{n}] \tag{38}$$

We start by decomposing the mean squared error of the combined value into the bias and variance of the MC and AMAF values respectively, making use of our second assumption that these values are independently distributed,

$$e^2_\star = \sigma^2_\star + b^2_\star \tag{39}$$
$$= (1-\beta)^2\sigma^2 + \beta^2\tilde{\sigma}^2 + (\beta\tilde{b} + (1-\beta)b)^2 \tag{40}$$
$$= (1-\beta)^2\sigma^2 + \beta^2\tilde{\sigma}^2 + \beta^2\tilde{b}^2 \tag{41}$$

Differentiating with respect to β and setting to zero,

$$0 = 2\beta\tilde{\sigma}^2 - 2(1-\beta)\sigma^2 + 2\beta\tilde{b}^2 \tag{42}$$
$$\beta = \frac{\sigma^2}{\sigma^2 + \tilde{\sigma}^2 + \tilde{b}^2} \tag{43}$$

We now make use of our first assumption that the MC and AMAF values are binomially distributed, and estimate their variance,

$$\sigma^2 = \frac{Q^\pi(s,a)(1 - Q^\pi(s,a))}{N(s,a)} \approx \frac{\mu_\star(1-\mu_\star)}{n} \tag{44}$$
$$\tilde{\sigma}^2 = \frac{\tilde{Q}^\pi(s,a)(1 - \tilde{Q}^\pi(s,a))}{\tilde{N}(s,a)} \approx \frac{\mu_\star(1-\mu_\star)}{\tilde{n}} \tag{45}$$
$$\beta = \frac{\tilde{n}}{n + \tilde{n} + n\tilde{n}\tilde{b}^2/\mu_\star(1-\mu_\star)} \tag{46}$$

In roughly even positions, $\mu_\star \approx \frac{1}{2}$, we can further simplify the schedule,

$$\beta = \frac{\tilde{n}}{n + \tilde{n} + 4n\tilde{n}\tilde{b}^2} \tag{47}$$

This equation still includes one unknown constant: the RAVE bias $\tilde{b}$. This can either be evaluated empirically (by testing the performance of the algorithm with various constant values of $\tilde{b}$), or by machine learning (by learning to predict the error between the AMAF value and the MC value, after many simulations). The former method is simple and effective; but the latter method could allow different biases to be identified for different types of position.

4.6.3. Results

We compared the performance of MC–RAVE using the minimum MSE schedule, using the approximation in Equation 47, to the hand-selected schedule in Equation 19. For the minimum MSE schedule, we first identified the best constant RAVE bias in empirical tests. On a 9×9 board, the performance of MoGo using the minimum MSE schedule increased by 80 Elo (see Table 1). On a 19×19 board, the improvement was more than 100 Elo.

5. Heuristic Prior Knowledge

We now introduce our second extension to Monte-Carlo tree search, *heuristic MCTS*. If a particular state s and action a is rarely encountered during simulation, then its Monte-Carlo value estimate is highly uncertain and very unreliable. Furthermore, because the search tree branches exponentially, the vast majority of nodes in the tree are only experienced rarely. The situation at the leaf nodes is worst of all: by definition each leaf node has been visited only once (otherwise a child node would have been added).

In order to reduce the uncertainty for rarely encountered positions, we incorporate prior knowledge by using a *heuristic evaluation function* $H(s, a)$ and a *heuristic confidence function* $C(s, a)$. When a node is first added to the search tree, it is initialised according to the heuristic function, $Q(s, a) = H(s, a)$ and $N(s, a) = C(s, a)$. The confidence in the heuristic function is measured in terms of *equivalent experience*: the number of simulations that would be required in order to achieve a Monte-Carlo value of similar accuracy to the heuristic value.[14] After initialisation, the value and count are updated as usual, using standard Monte-Carlo simulation.

5.1. Heuristic MC–RAVE

The heuristic Monte-Carlo tree search algorithm can be combined with the MC–RAVE algorithm, described in pseudocode in Algorithm 2. When a new node $n(s)$ is added to the tree, and for all actions $a \in \mathcal{A}$, we initialise both the MC and AMAF values to the heuristic evaluation function, and initialise both counts to heuristic confidence functions C and $\tilde{C}$ respectively,

[14] This is equivalent to a beta prior when binary outcomes are used.

Algorithm 2 Heuristic MC–RAVE

procedure MC–RAVE(s_0)
  while time available **do**
    SIMULATE($board, s_0$)
  end while
  $board.SetPosition(s_0)$
  return SELECTMOVE($board, s_0, 0$)
end procedure

procedure SIMULATE($board, s_0$)
  $board.SetPosition(s_0)$
  $[s_0, a_0, ..., s_T, a_T]$ = SIMTREE($board$)
  $[a_{T+1}, ..., a_D], z$ = SIMDEFAULT($board, T$)
  BACKUP($[s_0, ..., s_T], [a_0, ..., a_D], z$)
end procedure

procedure SIMDEFAULT($board, T$)
  $t = T + 1$
  while not $board.GameOver()$ **do**
    a_t = DEFAULTPOLICY($board$)
    $board.Play(a_t)$
    $t = t + 1$
  end while
  $z = board.BlackWins()$
  return $[a_{T+1}, ..., a_{t-1}], z$
end procedure

procedure SIMTREE($board$)
  $t = 0$
  while not $board.GameOver()$ **do**
    $s_t = board.GetPosition()$
    if $s_t \notin tree$ **then**
      NEWNODE(s_t)
      a_t = DEFAULTPOLICY($board$)
      return $[s_0, a_0, ..., s_t, a_t]$
    end if
    a_t = SELECTMOVE($board, s_t$)
    $board.Play(a_t)$
    $t = t + 1$
  end while
  return $[s_0, a_0, ..., s_{t-1}, a_{t-1}]$
end procedure

procedure SELECTMOVE($board, s$)
  $legal = board.Legal()$
  if $board.BlackToPlay()$ **then**
    return $\underset{a \in legal}{\text{argmax}}$ EVAL(s, a)
  else
    return $\underset{a \in legal}{\text{argmin}}$ EVAL(s, a)
  end if
end procedure

procedure EVAL(s, a)
  $b = pretuned\ constant\ bias\ value$
  $\beta = \frac{\tilde{N}(s,a)}{N(s,a)+\tilde{N}(s,a)+4N(s,a)\tilde{N}(s,a)b^2}$
  return $(1 - \beta)Q(s, a) + \beta\tilde{Q}(s, a)$
end procedure

procedure
BACKUP($[s_0, ..., s_T], [a_0, ..., a_D], z$)
  for $t = 0$ **to** T **do**
    $N(s_t, a_t)$ ++
    $Q(s_t, a_t)$ += $\frac{z - Q(s_t, a_t)}{N(s_t, a_t)}$
    for $u = t$ **to** D **step** 2 **do**
      if $a_u \notin [a_t, a_{t+2}, ..., a_{u-2}]$ **then**
        $\tilde{N}(s_t, a_u)$ ++
        $\tilde{Q}(s_t, a_u)$ += $\frac{z - \tilde{Q}(s_t, a_t)}{\tilde{N}(s_t, a_t)}$
      end if
    end for
  end for
end procedure

procedure NEWNODE($board, s$)
  $tree.Insert(s)$
  for all $a \in board.Legal()$ **do**
    $N(s, a), Q(s, a), \tilde{N}(s, a), \tilde{Q}(s, a)$
      = HEURISTIC($board, a$)
  end for
end procedure

$$Q(s,a) \leftarrow H(s,a) \tag{48}$$
$$N(s,a) \leftarrow C(s,a) \tag{49}$$
$$\tilde{Q}(s,a) \leftarrow H(s,a) \tag{50}$$
$$\tilde{N}(s,a) \leftarrow \tilde{C}(s,a) \tag{51}$$
$$N(s) \leftarrow \sum_{a \in \mathcal{A}} N(s,a) \tag{52}$$

We compare four heuristic evaluation functions in 9×9 Go, using the heuristic MC–RAVE algorithm in the program *MoGo*.

1. The *even-game* heuristic, $Q_{even}(s,a) = 0.5$, makes the assumption that most positions encountered between strong players are likely to be close.
2. The *grandfather* heuristic, $Q_{grand}(s_t,a) = Q(s_{t-2},a)$, sets the value of each node in the tree to the value of its grandfather. This assumes that the value of a Black move is usually similar to the value of that move, last time Black was to play.
3. The *handcrafted* heuristic, $Q_{mogo}(s,a)$, is based on the pattern-based rules that were successfully used in *MoGo's* default policy. The heuristic was designed such that moves matching a "good" pattern were assigned a value of 1, moves matching a "bad" pattern were given value 0, and all other moves were assigned a value of 0.5. The good and bad patterns were identical to those used in *MoGo*, such that selecting moves greedily according to the heuristic, and breaking ties randomly, would exactly produce the default policy π_{mogo}.
4. The *local shape* heuristic, $Q_{rlgo}(s,a)$, is computed from the linear combination of local shape features used in *RLGO 1.0* (see Section 3.4). This heuristic is learnt offline by temporal difference learning from games of self-play.

For each heuristic evaluation function, we assign a heuristic confidence $\tilde{C}(s,a) = M$, for various constant values of equivalent experience M. We played 2300 games between *MoGo* and GnuGo 3.7.10 (level 10). The MC–RAVE algorithm executed 3,000 simulations per move (see Figure 6).

The value function learnt from local shape features, Q_{rlgo}, outperformed all the other heuristics and increased the winning rate of *MoGo* from 60% to 69%. Maximum performance was achieved using an equivalent experience of $M = 50$, which indicates that Q_{rlgo} is worth about as much as 50 simulations using all-moves-as-first. It seems likely that these results could be further improved by varying the heuristic confidence according to the particular position, based on the variance of the heuristic evaluation function.

5.2. Exploration and Exploitation

The performance of Monte-Carlo tree search is greatly improved by carefully balancing exploration with exploitation. The UCT algorithm significantly outperforms a greedy tree policy in computer Go [19]. Surprisingly, this result does not appear

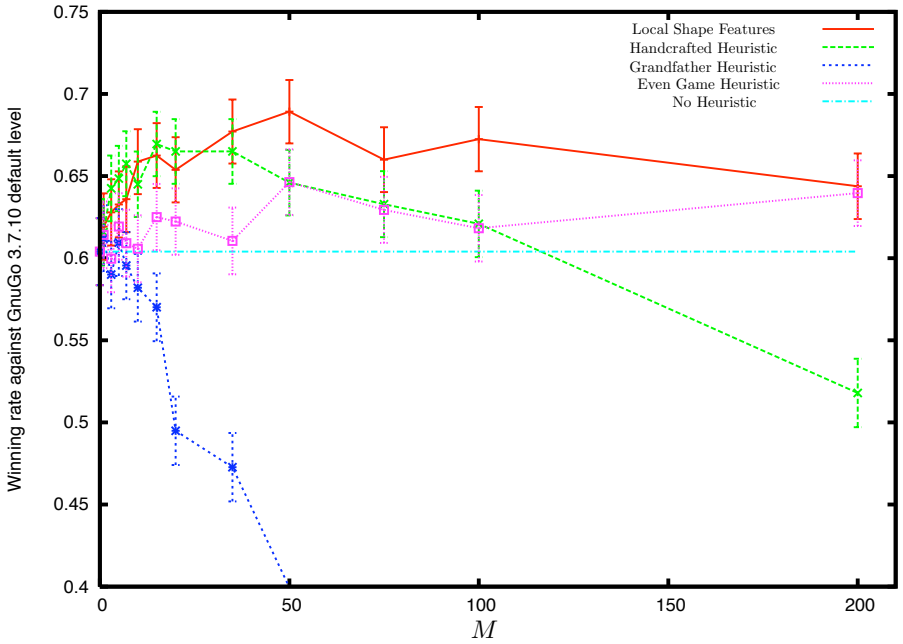

Figure 6: Winning rate of *MoGo*, using the heuristic MC–RAVE algorithm, with 3,000 simulations per move against GnuGo 3.7.10 (level 10) in 9×9 Go. Four different forms of heuristic function were used (see text). The bars indicate the standard error. Each point of the plot is an average over 2300 complete games.

to extend to the heuristic UCT–RAVE algorithm: the optimal exploration rate in our experiments was zero, i.e. greedy MC–RAVE with no exploration in the tree policy.

We believe that the explanation lies in the nature of the RAVE algorithm. Even if an action a is not selected immediately from position s, it will often be played at some later point in the simulation. This greatly reduces the need for explicit exploration, because the values for all actions are continually updated, regardless of the initial move selection.

However, we were only able to run thorough tests with tens of thousands of simulations per move. It is possible that exploration again becomes important when MC–RAVE is scaled up to millions of simulations per move. At this point a substantial number of nodes will be dominated by MC values rather than RAVE values, so that exploration at these nodes should be beneficial.

5.3. Soft Pruning

Computer Go has a large branching factor and several pruning techniques, such as selective search and progressive widening (see Section 3), have been developed to reduce the size of the search space [40]. Heuristic MCTS and MC–RAVE can be viewed as *soft pruning* techniques that focus on the highest valued regions of the search space without permanently cutting off any branches of the search tree.

Schedule	Computation	Wins vs. GnuGo	CGOS rating
Hand-selected	3,000 sims per move	69%	1960
Hand-selected	10,000 sims per move	82%	2110
Hand-selected	10 minutes per game	92%	2320
Minimum MSE	10 minutes per game	97%	2480*

Table 1: Winning rate of *MoGo* against GnuGo 3.7.10 (level 10) when the number of simulations per move is increased. *MoGo* competed on CGOS, using heuristic MC–RAVE and the hand-selected schedule, in February 2007. The versions using 10 minutes per game modify the simulations per move according to the available time, from $300,000$ games in the opening to $20,000$ in the endgame. The asterisked version competed on CGOS in April 2007 using the minimum MSE schedule and additional parameter tuning.

A heuristic function provides a principled way to use prior knowledge to reduce the effective branching factor. Moves favoured by the heuristic function will be initialised with a high value, and tried much more often than moves with a low heuristic value. However, if the heuristic evaluation function is incorrect, then the initial value will drop off at a rate determined by the heuristic confidence function, and other moves will then be explored.

The MC–RAVE algorithm also significantly reduces the effective branching factor. RAVE forms a fast, rough estimate of the value of each move. Moves with high RAVE values will quickly become favoured over moves with low RAVE values, which are soft pruned from the search tree. However, the RAVE values are only used initially, so that MC–RAVE never cuts branches permanently from the search tree.

Heuristic MC–RAVE can often be wrong. The heuristic evaluation function can be inaccurate, and/or the RAVE estimate can be misleading. In this case, heuristic MC–RAVE will prioritise the wrong moves, and the best moves can be soft pruned and not tried again for many simulations. There are no guarantees that these algorithms will help performance. However, in practice they help more than they hurt, and on average over many positions, they provide a very significant performance advantage.

5.4. Performance of heuristic MC–RAVE in MoGo

Our two extensions to MCTS, heuristic MCTS and MC–RAVE, increased the winning rate of *MoGo* against GnuGo, from 24% for UCT, up to 69% using heuristic MC–RAVE. However, these results were based on executing just 3,000 simulations per move, using the hand-selected schedule in Equation 19. When the number of simulations was increased, the overall performance of *MoGo* improved correspondingly. Table 1 shows how the performance of heuristic MC–RAVE scales with additional computation.

The 2007 release version of *MoGo* used the heuristic MC–RAVE algorithm, the minimum MSE schedule in Equation 47, and an improved, handcrafted heuristic function.[15] The scalability of the release version is shown in Figure 7, based on the results of a combined study over many thousands of computer hours [41]. This version of

[15]Local shape features were not used in the release version.

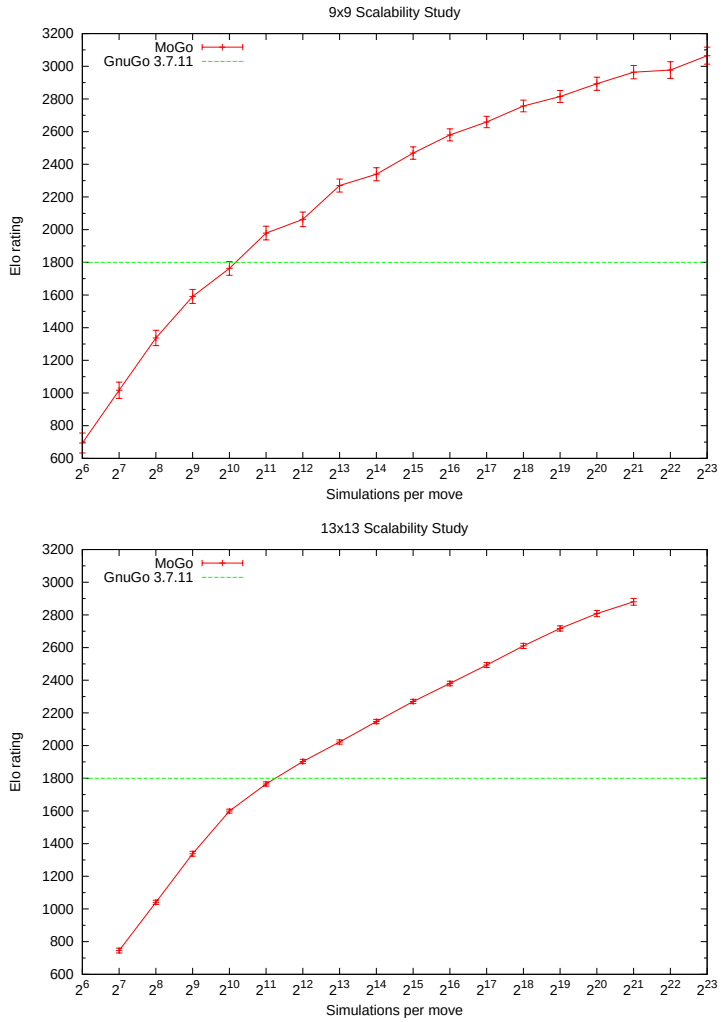

Figure 7: Scalability of *MoGo* (2007 release version), using data collected by members of the Computer Go mailing list [41]. Elo ratings were computed from a large tournament, consisting of several thousand games for each version of *MoGo*, using successive doublings of the number of simulations per move. Error bars indicate 95% confidence intervals in the Elo rating.

MoGo became the first program to achieve *dan* level at 9×9 Go; the first program to beat a professional human player at 9×9 Go; the highest rated program on the Computer Go Server for both 9×9 and 19×19 Go; the gold medal winner at the 2007 19×19 Computer Go Olympiad; and achieved a rating of 2 *kyu* at 19×19 Go against human players on the Kiseido Go Server.

6. Survey of Subsequent Work

The results in the previous section were achieved by *MoGo* in 2007. We briefly survey subsequent work on heuristic MC–RAVE in a variety of other strong Go programs.

The heuristic function of *MoGo* was substantially enhanced by initialising $H(s,a)$, $C(s,a)$, and $\tilde{C}(s,a)$ to hand-tuned values based on handcrafted rules and patterns [42]. Supervised learning was also used to bias move selection towards patterns favoured in expert games. In addition, the handcrafted default policy was modified to increase the diversity of simulations, by playing in empty regions of the board; and to fix a known issue with life-and-death, by playing in the key point of simple dead shapes known as *nakade*. Using 100,000 simulations, the improved version of *MoGo* achieved a winning rate of 55% on 9×9 boards, and 53% on 19×19 boards, against the 2007 release version of *MoGo*.

MoGo was also modified by massively parallelising the MC–RAVE algorithm to run on a cluster [43]. In order to avoid huge communication overheads, memory was only shared between the shallowest nodes in the search tree. The massively parallel version of *MoGo Titan* was run on 800 processors of Huygens, the Dutch national supercomputer. *MoGo Titan* defeated a 9 *dan* professional player, Jun-Xun Zhou, in 19×19 Go with 7 stones handicap.

The program *Zen* has successfully combined MC–RAVE with more sophisticated domain knowledge. *Zen* became the first program to sustain a *dan* rank, on full-size 19×19 boards, against human players on the Kiseido Go Server (KGS). It is currently ranked at 4 *dan*, placing it within the top 5% of the 30,000 ranked human players on KGS.

Several of the strongest traditional Go programs now combine their existing tactical and pattern knowledge with the heuristic MC–RAVE framework, including *The Many Faces of Go*, currently ranked at 2 *dan* on KGS and *Aya*, currently ranked at 1 *dan*.

The open source program *Fuego* [44] extends the MC–RAVE algorithm to use additional rapid value estimates, using a variant of the minimum MSE schedule (see Equation 46). A parallelised version of *Fuego* defeated a 9 *dan* professional player in an even 9×9 game, and defeated a 6 *dan* amateur player with 4 stones handicap on a full size board.[16] The latest versions of *MoGo*, *Crazy Stone* and the *The Many Faces of Go* have also achieved impressive victories against professional players on full size boards.

Most recently, the program *Erica* combined heuristic MC–RAVE with a new technique, known as *simulation balancing* [45], to automatically tune the parameters of its

[16] See *Human-Computer Go Challenges*, *http://www.computer-go.info/h-c/index.html*.

Year	*Program*	*Description*	*Elo*
2006	*Indigo*	Pattern database, Monte-Carlo simulation	1400
2006	*GnuGo*	Pattern database, alpha-beta search	1800
2006	*Many Faces*		1800
2006	*NeuroGo*	Temporal-difference learning, neural network	1850
2007	*RLGO*	Temporal-difference search	2100
2007	*MoGo*	Variants of heuristic MC–RAVE	2500
2007	*Crazy Stone*		2500
2009	*Fuego*		2700
2010	*Many Faces*		2700
2010	*Zen*		2700

Table 2: Approximate Elo ratings, on the Computer Go Server, of 9×9 Go programs discussed in the text.

default policy [46]. Previous machine learning approaches have focused on optimising the strength of the default policy, under the assumption that a stronger policy will perform better in a Monte-Carlo search [1]. Unfortunately, in practice this assumption is often incorrect [11], and in general it can be difficult to find a default policy that performs well in Monte-Carlo search. The key idea of simulation balancing is to minimise the error between the Monte-Carlo value $Q(s, a)$, and an oracle value computed by deep search. Erica used simulation balancing to train 2000 parameters of its default policy for 9×9 Go. Erica also won the gold medal in the 2010 19×19 Computer Go Olympiad, and is currently ranked at 3 dan on KGS.

We provide a summary of the current state of the art in computer Go, based on ratings from the Computer Go Server (see Table 2) and the Kiseido Go Server (see Figure 8). Several of the programs described in Section 3 are included for comparison.

7. Conclusions

For the last 30 years, computer Go programs have evaluated positions by using handcrafted heuristics that are based on human expert knowledge of shapes, patterns and rules. However, professional Go players often play moves according to intuitive feelings that are hard to express or quantify. Precisely encoding their knowledge into machine-understandable rules has proven to be a dead-end: a classic example of the knowledge acquisition bottleneck. Furthermore, traditional search algorithms, which are based on these handcrafted heuristics, cannot cope with the enormous state space and branching factor in the game of Go, and are unable to make effective use of additional computation time. This approach has led to Go programs that are at best comparable to weak amateur-level humans [26, 47].

In contrast, Monte-Carlo tree search requires no human knowledge in order to understand a position. Instead, positions are evaluated from the outcome of thousands of simulated games of self-play from that position. These simulated games are progressively refined to prioritise the selection of positions with promising evaluations. Over the course of many simulations, attention is focused selectively on narrow regions of the search space that are correlated with successful outcomes. Unlike traditional search

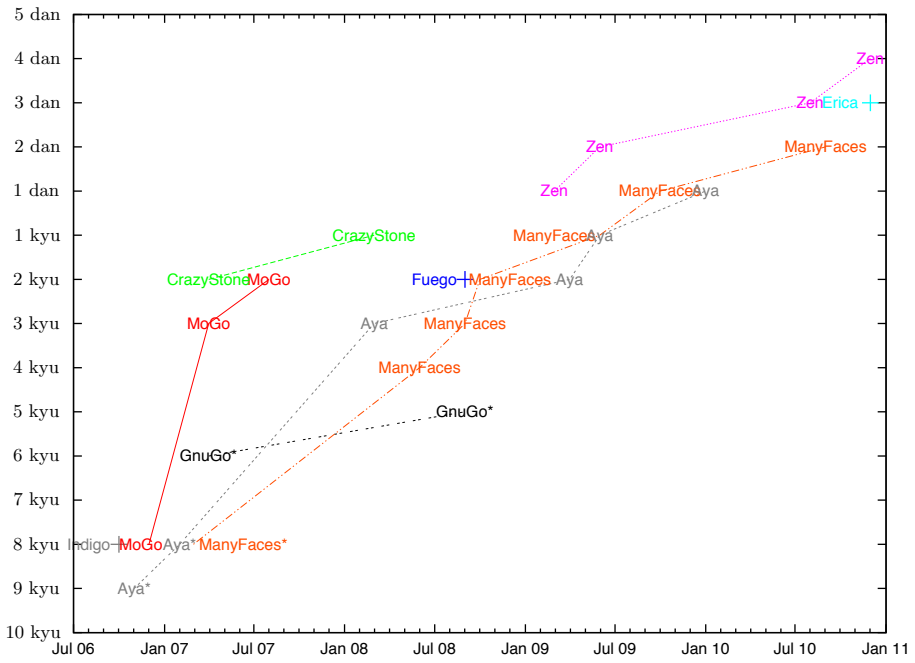

Figure 8: Ranks of various Go programs discussed in the text on the Kiseido Go Server (KGS). Each point represents the first date at which a program held the given rank for 20 consecutive games on KGS. Note that each program plays with different time controls, which may cause variations in rank; and that some programs play more regularly than others, which may cause variations in date. See http://senseis.xmp.net/?KGSBotRatings. *Version of *GnuGo, Aya, Many Faces of Go* based on traditional search with no Monte-Carlo.

algorithms, this approach scales well both with the size of the state space and branching factor, and also scale well with additional computation time. In practice, the strongest programs do make extensive use of expert human knowledge: both to improve the default policy and to define the prior knowledge. This knowledge accelerates the progress of the search, but does not affect its asymptotic optimality.

On the Computer Go Server, using 9×9, 13×13 and 19×19 board sizes, traditional search programs are rated at around 1800 Elo, whereas Monte-Carlo programs, enhanced by RAVE and heuristic knowledge, are rated at over 2500 Elo using standard hardware[17] (see Table 2). On the Kiseido Go Server, on full-size boards against human opposition, traditional search programs have reached 5 *kyu*, whereas the best Monte-Carlo programs are rated at 4 *dan* (see Figure 8). The top programs are now competitive with top human professionals at 9×9 Go, and are winning handicap games against top human professionals at 19×19 Go.

In the Go program *Mogo*, every doubling in computation power led to an increase

[17] A difference of 700 Elo corresponds to a 99% winning rate.

in playing strength of approximately 100 Elo points in 13×13 Go (see Figure 7) and perhaps even more in 19×19 Go. The strongest programs still lag far behind the strongest humans, but they are improving rapidly. Figure 8 shows that, after the initial jump in performance achieved by the first Monte-Carlo programs, computer Go programs have continued to improve by more than one rank every year.

This new framework for Monte-Carlo tree search also extends beyond Go. Variants of heuristic MC–RAVE have outperformed previous search algorithms in other challenging games, such as Hex [48] and Havannah [49]. The challenging properties of Go are also characteristic of many of the hardest search, planning and decision-making problems. Immediate actions often have delayed, long-term consequences, leading to surprising complexity and enormous search spaces that are intractable to traditional search algorithms. Variants of heuristic MCTS and MC–RAVE are now outperforming previous approaches in challenging search spaces such as feature selection [50], POMDP planning [51], and natural language phrase generation [52]. Understanding how to achieve high performance in Go is opening up new possibilities for high performance AI in a wide variety of challenging problems.

References

[1] R. Coulom, Efficient selectivity and backup operators in Monte-Carlo tree search, in: 5th International Conference on Computer and Games, pp. 72–83.

[2] R. Coulom, Computing Elo ratings of move patterns in the game of Go, International Computer Games Association Journal 30 (2007) 198–208.

[3] S. Gelly, D. Silver, Achieving master level play in 9 x 9 computer Go, in: 23rd Conference on Artificial Intelligence, pp. 1537–1540.

[4] H. Finnsson, Y. Björnsson, Simulation-based approach to general game playing, in: 23rd Conference on Artificial Intelligence, pp. 259–264.

[5] R. Lorentz, Amazons discover Monte-Carlo, in: 6th International Conference on Computers and Games, pp. 13–24.

[6] M. Winands, Y. Y. Björnsson, Evaluation function based Monte-Carlo LOA, in: 12th Advances in Computer Games Conference, pp. 33–44.

[7] J. Schäfer, The UCT algorithm applied to games with imperfect information, Diploma Thesis. Otto-von-Guericke-Universität Magdeburg, 2008.

[8] N. Sturtevant, An analysis of UCT in multi-player games, in: 6th International Conference on Computers and Games, pp. 37–49.

[9] R. Balla, A. Fern, UCT for tactical assault planning in real-time strategy games, in: 21st International Joint Conference on Artificial Intelligence, pp. 40–45.

[10] L. Kocsis, C. Szepesvari, Bandit based Monte-Carlo planning, in: 15th European Conference on Machine Learning, pp. 282–293.

[11] S. Gelly, D. Silver, Combining online and offline learning in UCT, in: 17th International Conference on Machine Learning, pp. 273–280.

[12] S. Gelly, A Contribution to Reinforcement Learning; Application to Computer Go, Ph.D. thesis, University of South Paris, 2007.

[13] D. Silver, Reinforcement Learning and Simulation-Based Search in the Game of Go, Ph.D. thesis, University of Alberta, 2009.

[14] D. Billings, L. P. Castillo, J. Schaeffer, D. Szafron, Using probabilistic knowledge and simulation to play poker, in: 16th National Conference on Artificial Intelligence, pp. 697–703.

[15] B. Bouzy, B. Helmstetter, Monte-Carlo Go developments, in: 10th Advances in Computer Games Conference, pp. 159–174.

[16] G. Tesauro, G. Galperin, On-line policy improvement using Monte-Carlo search, in: Advances in Neural Information Processing 9, pp. 1068–1074.

[17] B. Sheppard, World-championship-caliber Scrabble, Artificial Intelligence 134 (2002) 241–275.

[18] P. Auer, N. Cesa-Bianchi, P. Fischer, Finite-time analysis of the multi-armed bandit problem, Machine Learning 47 (2002) 235–256.

[19] S. Gelly, Y. Wang, R. Munos, O. Teytaud, Modification of UCT with Patterns in Monte-Carlo Go, Technical Report 6062, INRIA, 2006.

[20] B. Bouzy, Associating domain-dependent knowledge and Monte Carlo approaches within a Go program, Information Sciences, Heuristic Search and Computer Game Playing IV 175 (2005) 247–257.

[21] J. McCarthy, AI as sport, Science 276 (1997) 1518–1519.

[22] D. McClain, Once again, machine beats human champion at chess, New York Times, December 5th (2006).

[23] A. Harmon, Queen, captured by mouse; more chess players use computers for edge, New York Times, February 6th (2003).

[24] D. Mechner, All Systems Go, The Sciences 38 (1998).

[25] J. Schaeffer, The games computers (and people) play, Advances in Computers 50 (2000) 189–266.

[26] M. Müller, Computer Go, Artificial Intelligence 134 (2002) 145–179.

[27] N. Schraudolph, P. Dayan, T. Sejnowski, Temporal difference learning of position evaluation in the game of Go, in: Advances in Neural Information Processing 6, pp. 817–824.

[28] M. Enzenberger, The integration of a priori knowledge into a Go playing neural network, http://www.cs.ualberta.ca/ emarkus/neurogo/neurogo1996.html, 1996.

[29] F. Dahl, Honte, a Go-playing program using neural nets, in: Machines that learn to play games, Nova Science, 1999, pp. 205–223.

[30] M. Enzenberger, Evaluation in Go by a neural network using soft segmentation, in: 10th Advances in Computer Games Conference, pp. 97–108.

[31] D. Silver, R. Sutton, M. Müller, Reinforcement learning of local shape in the game of Go, in: 20th International Joint Conference on Artificial Intelligence, pp. 1053–1058.

[32] D. Silver, R. Sutton, M. Müller, Sample-based learning and search with permanent and transient memories, in: 25th International Conference on Machine Learning, pp. 968–975.

[33] B. Bruegmann, Monte-Carlo Go, http://www.cgl.ucsf.edu/go/Programs/Gobble.html, 1993.

[34] B. Bouzy, Move pruning techniques for Monte-Carlo Go, in: 11th Advances in Computer Games Conference, pp. 104–119.

[35] B. Bouzy, Associating shallow and selective global tree search with Monte Carlo for 9x9 Go, in: 4th International Conference on Computers and Games, pp. 67–80.

[36] B. Bouzy, History and territory heuristics for Monte-Carlo Go, New Mathematics and Natural Computation 2 (2006) 1–8.

[37] B. Abramson, Expected-outcome: A general model of static evaluation, IEEE Transactions on Pattern Analysis and Machine Intelligence 12 (1990) 182–193.

[38] Y. Wang, S. Gelly, Modifications of UCT and sequence-like simulations for Monte-Carlo Go, in: IEEE Symposium on Computational Intelligence and Games, Honolulu, Hawaii, pp. 175–182.

[39] J. Schaeffer, The history heuristic and alpha–beta search enhancements in practice, IEEE Transactions on Pattern Analysis and Machine Intelligence PAMI-11 (1989) 1203–1212.

[40] G. Chaslot, M. Winands, J. Uiterwijk, H. van den Herik, B. Bouzy, Progressive strategies for Monte-Carlo tree search, New Mathematics and Natural Computation 4 (2008) 343–357.

[41] D. Dailey, 9x9 scalability study, http://cgos.boardspace.net/study/index.html, 2008.

[42] G. Chaslot, L. Chatriot, C. Fiter, S. Gelly, J. Hoock, J. Perez, A. Rimmel, O. Teytaud, Combining expert, online, transient and online knowledge in Monte-Carlo exploration, in: 8th European Workshop on Reinforcement Learning.

[43] S. Gelly, J. Hoock, A. Rimmel, O. Teytaud, Y. Kalemkarian, The parallelization of Monte-Carlo planning, in: 6th International Conference in Control, Automation and Robotics, pp. 244–249.

[44] M. Müller, M. Enzenberger, Fuego – An Open-source Framework for Board Games and Go Engine Based on Monte-Carlo Tree Search, Technical Report TR09-08, University of Alberta, Dept. of Computing Science, 2009.

[45] D. Silver, G. Tesauro, Monte-Carlo simulation balancing, in: 26th International Conference on Machine Learning, pp. 119–126.

[46] S. Huang, R. Coulom, S. Lin, Monte-Carlo simulation balancing in practice, in: 7th International Conference on Computers and Games, pp. 119–126.

[47] B. Bouzy, T. Cazenave, Computer Go: an AI-oriented survey, Artificial Intelligence 132 (2001) 39–103.

[48] B. Arneson, R. Hayward, P. Henderson, MoHex wins Hex tournament, International Computer Games Association Journal 32 (2009) 114–116.

[49] F. Teytaud, O.Teytaud, Creating an Upper Confidence Tree program for Havannah, in: 12th Advances in Computer Games Conference, pp. 65–74.

[50] R. Gaudel, M. Sebag, Feature selection as a one-player game, in: 27th International Conference on Machine Learning, pp. 359–366.

[51] D. Silver, J. Veness, Online Monte-Carlo planning in large POMDPs, in: Advances in Neural Information Processing Systems 24.

[52] J. Chevelu, T. Lavergne, Y. Lepage, T. Moudenc, Introduction of a new paraphrase generation tool based on Monte-Carlo sampling, in: 47th Annual Meeting of the Association for Computational Linguistics, pp. 249–252.