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

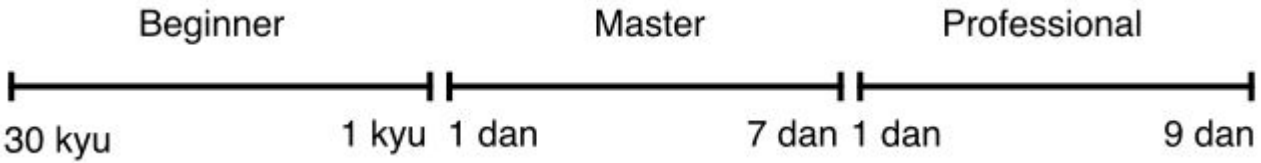
By Sylvain Gelly and David Silver
Presented by Caleb Nelson

The Game of Go

- Go is played on a 19x19, 13x13, or 9x9 grid
- Black and White take turns putting stones on intersections of the grid
- 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
- Blocks which cannot be captured are described as alive; blocks which will certainly be captured are described as dead
- The game ends when both players pass without placing a stone. Dead blocks are removed from the board and each player adds up their stones and intersections they surround, which becomes their final score.

Extra Go Terminology

- Blocks with just one remaining liberty are said to be in atari
- 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
- Since Black always goes first, White receives compensation, known as komi, which is added to White's score at the end of the game
- Rankings in Go come in 3 stages



Monte-Carlo Tree Search

- MCTS is an alternative to minimax that makes it much more feasible to search massive state spaces with a large number of state/action pairs
- Instead of mapping the whole game tree at once, MCTS chooses the moves that seem the most promising and simulates them until completion, reflecting the result in its evaluation of the move
- Can run indefinitely and be safely cut off at any time - once it is cut off, it simply chooses what it feels to be the best move available to it and can begin the search anew
- Given infinite time, MCTS will always converge to minimax.

UCT Algorithm

- MCTS will explore the nodes it considers most promising - but how does it factor uncertainty into its evaluation of nodes?
  - Namely, how does it balance exploration and exploitation?
- The UCB1-T (short for the Upper Confidence Bound 1 algorithm for Trees) or UCT algorithm encourages optimism in the face of uncertainty by replacing the Q-values with ones that contain a benefit for under-explored states

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

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

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 ∉ 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* = argmax_{a∈legal}( Q(s,a) + c sqrt( log N(s) / N(s,a) ) )
    else
        a* = argmin_{a∈legal}( Q(s,a) - c sqrt( log N(s) / N(s,a) ) )
    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) += 1
        Q(s_t, a_t) += (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 ∈ A do
        N(s, a) = 0
        Q(s, a) = 0
    end for
end procedure
```

The All-Moves-As-First Heuristic

- The AMAF heuristic is a way to reduce the number of state/action pairs that need to be evaluated to make a good decision
- Essentially, the idea is that given a state, a certain action taken on the state will have the same value whether or not the move is played immediately or at any point during the game
- This means that, for any given state, each available action will only have one value associated with it for the rest of the game, which makes Monte-Carlo rollouts much faster

Rapid Action Value Estimation

- RAVE combines MCTS with the AMAF heuristic for faster evaluation
- When RAVE is choosing a move from the current state of the game, it values all available actions from the current state for the rest of the simulation, instead of re-evaluating actions at every simulated state

MC-RAVE

- Unfortunately, RAVE by itself doesn't work very well - the value of actions can be changed drastically by changes elsewhere on the board
- MC–RAVE overcomes this issue, by combining the rapid learning of the RAVE algorithm with the accuracy and convergence guarantees of MCTS
- MC-RAVE estimates the overall value of action a in state s by using a weighted sum Q*(s,a) of the MC value Q(s,a) and the AMAF value Q˜(s,a)
- β(s,a) is a weighting parameter that can change over time - usually change it from prioritizing AMAF value to MC value as more information is gathered

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

UCT-RAVE

- UCT-RAVE combines the exploration of UCT with MC-RAVE
- If β(s,a) trends to 0 for all states and actions, then the asymptotic behaviour of UCT-RAVE is equivalent to UCT
    - Remember that β(s,a) = 1 means only AMAF value is considered while β(s,a) = 0 means only MC value is considered

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

β(s,a) schedules

- Two different schedules for changing the parameter β(s,a) over time
- Hand-selected schedule where β(s,a) = 1/2 after some number of simulations k, chosen manually

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

- Minimum MSE schedule that selects β(s,a) as to minimize the MSE in the combined estimate Q*(s, a)

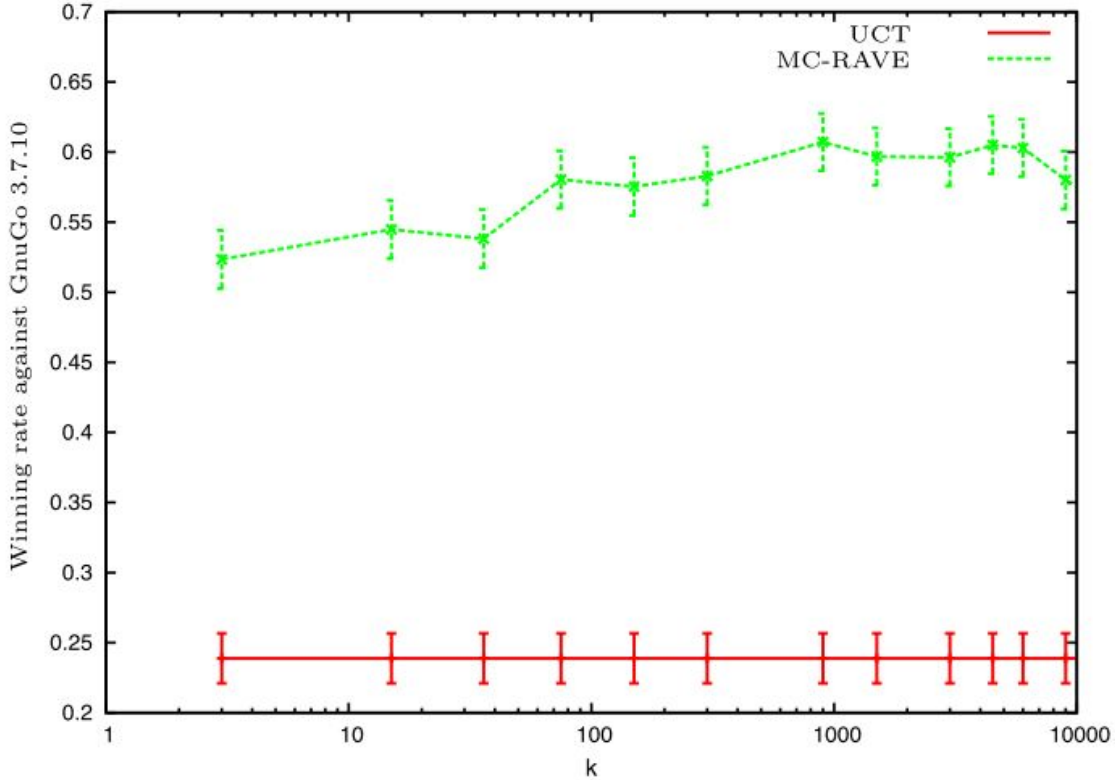


Fig. 5. Winning rate of MC–RAVE with 3000 simulations per move against GnuGo 3.7.10 (level 10) in 9×9 Go, for different settings of the equivalence parameter k. The bars indicate the standard error. Each point of the plot is an average over 2300 complete games.

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.

Schedule	Computation	Wins vs. GnuGo	CGOS rating
Hand-selected	3000 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*

Heuristic MCTS

- Since the state space for Go is so large, state/action pairs are rarely visited more than once, which makes MCTS unreliable
- In order to reduce the uncertainty for rarely encountered positions, 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)
- 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

Heuristic MC-RAVE

- Heuristic MC and MC-RAVE can be combined to form Heuristic MC-RAVE
- When a new node n(s) is added to the tree, and for all actions a ∈ A, initialise both the MC and AMAF values to the heuristic evaluation function, and initialise both counts to heuristic confidence functions C and C˜ respectively

$$
\begin{aligned}
Q(s,a) &\leftarrow H(s,a), \\
N(s,a) &\leftarrow C(s,a), \\
\tilde{Q}(s,a) &\leftarrow H(s,a), \\
\tilde{N}(s,a) &\leftarrow \tilde{C}(s,a), \\
N(s) &\leftarrow \sum_{a \in \mathcal{A}} N(s,a).
\end{aligned}
$$

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 ∉ 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 argmax_{a∈legal} EVAL(s, a)
    else
        return argmin_{a∈legal} EVAL(s, a)
    end if
end procedure

procedure EVAL(s, a)
    b = pretuned constant bias value
    β = Ñ(s,a) / (N(s,a)+Ñ(s,a)+4N(s,a)Ñ(s,a)b^2)
    return (1 − β) Q(s, a) + β Q̃(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) += 1
        Q(s_t, a_t) += (z − Q(s_t, a_t)) / N(s_t, a_t)
        for u = t to D step 2 do
            if a_u ∉ [a_t, a_{t+2}, ..., a_{u−2}] then
                Ñ(s_t, a_u) += 1
                Q̃(s_t, a_u) += (z − Q̃(s_t, a_t)) / Ñ(s_t, a_t)
            end if
        end for
    end for
end procedure

procedure NEWNODE(board, s)
    tree.Insert(s)
    for all a ∈ board.Legal() do
        N(s, a), Q(s, a), Ñ(s, a), Q̃(s, a) = HEURISTIC(board, a)
    end for
end procedure
```

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 3000 simulations per move (see Fig. 6).

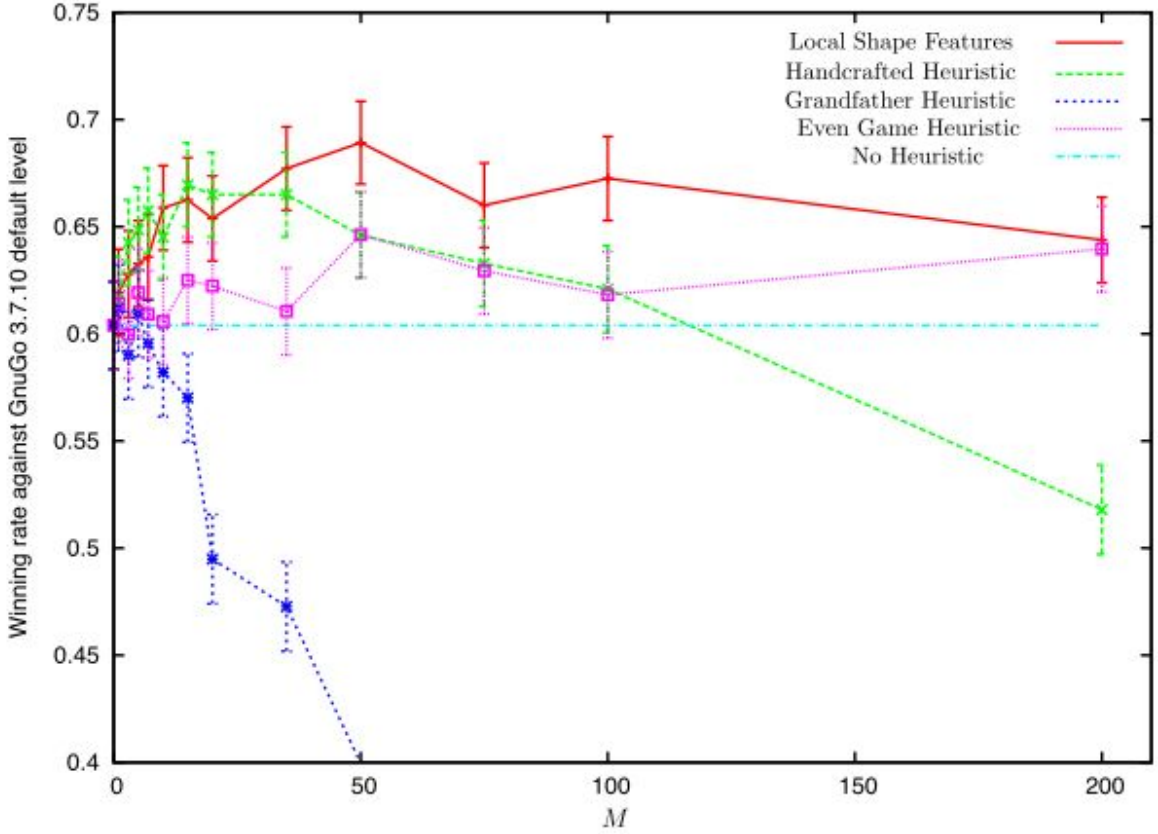
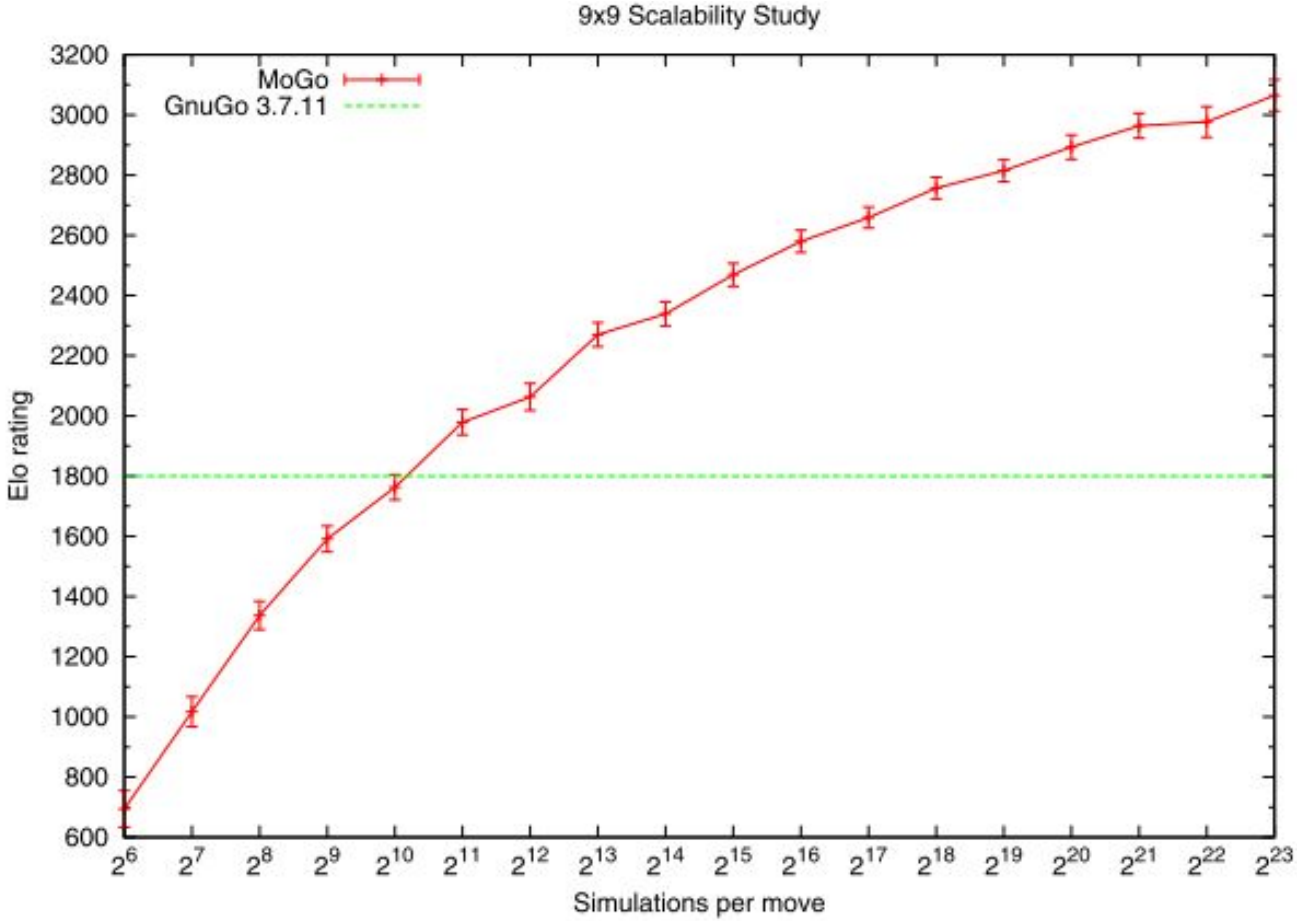


Fig. 6. Winning rate of *MoGo*, using the heuristic MC–RAVE algorithm, with 3000 simulations per move against GnuGo 3.7.10 (level 10) in 9×9 Go. Four different forms of heuristic function were used (see text). The bars indicate the standard error. Each point of the plot is an average over 2300 complete games.

Results of Heuristic MC-RAVE

- When executing 3000 simulations per move and using the hand-selected schedule, Heuristic MC-RAVE increased the winning rate of MoGo against GnuGo, from 24% for UCT, up to 69%
    - With more simulations, improvements increase even further
- 2007 release version of MoGo used the heuristic MC–RAVE algorithm, the minimum MSE schedule, and an improved, handcrafted heuristic function to become the first program to achieve master dan level in 9 x 9 go and 2 kyu in 19 x 19 Go


9x9 Scalability Study
MoGo
GnuGo 3.7.11
Elo rating
Simulations per move
3200
3000
2800
2600
2400
2200
2000
1800
1600
1400
1200
1000
800
600
2^6 2^7 2^8 2^9 2^{10} 2^{11} 2^{12} 2^{13} 2^{14} 2^{15} 2^{16} 2^{17} 2^{18} 2^{19} 2^{20} 2^{21} 2^{22} 2^{23}

Improvements to Heuristic MC-RAVE

- The heuristic function of MoGo was substantially enhanced by initialising H(s,a), C(s,a), and C˜(s,a) to hand-tuned values based on handcrafted rules and patterns
- Supervised learning was also used to bias move selection towards patterns favored by expert players
- MoGo was also modified by massively parallelising the MC–RAVE algorithm to run on a cluster
    - This parallel version, while running on Huygens, the Dutch national supercomputer, was able to defeat 9-dan player Jun-Xun Zhou in 19 x 19 Go with 7 handicap stones, giving it an effective rank of 2-dan

Source

- All information, images, and excerpts are from http://www.ics.uci.edu/~dechter/courses/ics-295/winter-2018/papers/mcts-gelly-silver.pdf

Thank you for listening!
Questions?