# THE LAST-GOOD-REPLY POLICY FOR MONTE-CARLO GO

*Peter Drake*[1]

Portland, Oregon, USA

## ABSTRACT

The dominant paradigm for computer Go players is Monte-Carlo tree search (MCTS). This algorithm builds a search tree by playing many simulated games (*playouts*). Each playout consists of a sequence of moves within the tree followed by many moves beyond the tree. Moves beyond the tree are generated by a biased random *sampling policy*. This paper presents a dynamic sampling policy that takes advantage of information from previous playouts. This policy makes moves that, in previous playouts, have been successful replies to immediately preceding moves. Experimental results show that this provides a large improvement in playing strength.

## 1. INTRODUCTION

Go is a deterministic, zero-sum, two-player game of perfect information (Baker, 2008). Writing programs to play Go well stands as a grand challenge of artificial intelligence (Cai and Wunsch, 2007). The strongest programs are only able to defeat professional human players with the aid of large handicaps, allowing the program to play several additional moves at the beginning of the game (Chaslot et al, 2009). Even this performance is the result of a recent breakthrough: before 2008, programs were unable to defeat professional players on the full 19x19 board despite enormous handicaps (Garlock, 2008; Müller, 2002; Wedd, 2009).

This breakthrough, which lies at the core of all strong, modern programs, is Monte-Carlo tree search (MCTS) (Kocsis and Szepesvári, 2006). This algorithm elegantly combines classical artificial intelligence (tree search) with statistical sampling. MCTS is described in section 2.

Being a global search, plain MCTS is unable to break the Go board down into relatively independent subproblems, something humans do with ease. Section 3 explores this local search problem.

Section 4 presents the new sampling policy. Section 5 provides experimental evidence that this policy improves both performance and local search. Section 6 closes the paper with discussion and future work.

[1] Lewis & Clark College, Dept. of Mathematical Sciences, email:drake@lclark.edu

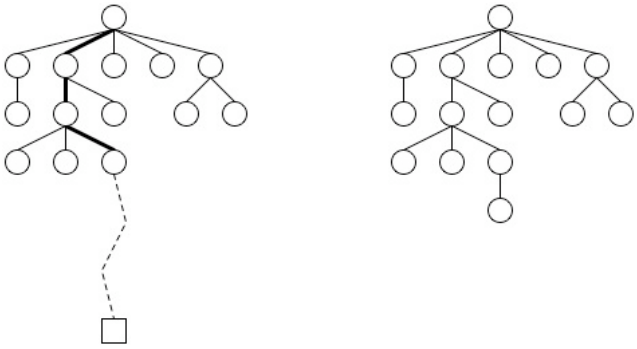

**Figure 1**: Monte-Carlo tree search.

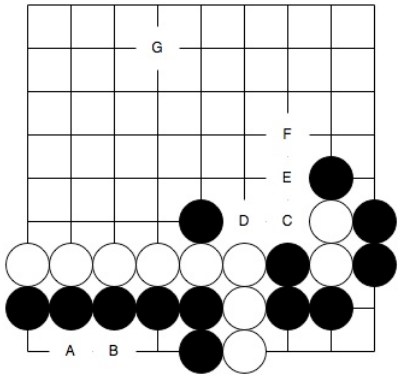


**Figure 2**: Local search.

## 2. MONTE-CARLO TREE SEARCH

Classical techniques such as alpha-beta search have not fared well in computer Go. One major problem is the difficulty of static evaluation: adding or removing a single stone can radically change the value of a board configuration. Monte-Carlo programs address this difficulty with statistical sampling: the value of a board for a player is the likelihood that a random game played from this board (a playout) will lead to a win for that player.

The simplest Monte-Carlo program would be a 1-ply search: for each legal move from the current position, take many samples from the resulting board. When search time runs out, choose the move with the highest win rate. The problem with this approach is that it assumes a naive opponent. If the opponent has one best reply to a move, it is unrealistic to assume that the opponent will respond randomly.

MCTS improves upon 1-ply search by growing a search tree during the sampling process as shown in Figure 1. Each playout consists of three phases. *Selection* (bold lines) chooses a sequence of moves within the tree. *Sampling* (dashed lines) completes the game with random moves. *Backpropagation* improves the tree to take into account the winner of the playout. This includes updating the win rate stored at each node and, as shown at the right of the figure, adding a node along the path taken in the playout.

The tree grows unevenly, with the more promising branches being explored more deeply. After thousands of playouts, the best move from the root (defined as, e.g., the move with the most wins) is returned.

Selection policies must strike a balance between exploring untried (or undersampled) branches and exploiting the information gathered so far. Specifically, simply choosing the move with the highest win rate at every point would lead to some moves being prematurely abandoned after a few "unlucky" playouts. The first successful MCTS programs resolved this issue by adding to the win rate an exploration term which encourages revisiting undersampled nodes (Kocsis and Szepesvári, 2006). The more recent Rapid Action Value Estimation (RAVE) technique (Gelly and Silver, 2007) propagates information back from each playout move to many preceding nodes. This encourages exploration of moves that fare well later in the playout; it also makes the exploration term unnecessary (Chaslot et al, 2009).

## 3. LOCAL SEARCH

A local situation on the board must often be explored redundantly by global MCTS. The subtree following move A in one branch of the search provides no information about the consequences of playing move A in another branch. This wastes information, as the correct replies in the two situations tend to be correlated.

Many moves in Go have correct local replies. In Figure 2, if white plays at A, black must play at B or all of the black stones in the lower left corner will die. A is therefore called a forcing move. Stones elsewhere on the board are irrelevant to the local correctness of this reply. (It is possible that some situation elsewhere, such as a ko fight, is so important that black can afford to ignore white's threat.)

In other situations, the correct reply can depend on the presence of other stones, even stones very far away. If white plays at C, black should respond at D. If white then plays at E, black should respond at F. This sequence, called a

ladder, will eventually result in the capture of the growing chain of white stones. Black should therefore always respond to white's attempts to escape. If, however, there is a white stone at G, the white chain can escape and black would be foolish to pursue it. Whether D is the correct reply to C depends on the presence or absence of a ladder-breaking stone at the distant point G.

Human players very quickly learn to read ladders to their conclusion. Such deep reading is often easy because there is only one move to consider at each ply. Global MCTS, on the other hand, would have to re-explore the ladder after any "distracting" moves, such as the A-B exchange.

In addition to wasting time, this redundant searching can cause MCTS to miss good moves. Consider a situation where an important local fight is very difficult for black assuming white responds correctly, but easy for black if both players play naively. MCTS playing as black might try to create a sequence of distracting forcing moves that push the resolution of the fight beyond the frontier of the search tree. This is a variation of the horizon effect (Russell and Norvig, 2003).

It can be argued that much of human Go-playing strength comes from the ability to perform local search. The partial independence of local situations makes Go fascinating both as a game and as a research problem. Many serious real-world problems share this feature. Consider the design of mass transit systems. The systems of two nearby cities are largely independent, but they interact for commuters who must connect with trains running between the cities.

# 4. A DYNAMIC SAMPLING POLICY

The term "policy" is used in different ways in the literature. Henceforth in this paper, "policy" will specifically refer to the sampling policy, as opposed to the selection or backpropagation policies.

## 4.1. Existing Policies

The simplest policy is to choose from among all legal moves with a uniform probability. It is standard to omit friendly eyelike points, i.e., points where all of the orthogonal neighbors are friendly stones, and at most one of the diagonal points (or none around an edge or corner) are enemy stones. Playing in friendly eyelike points is almost always a bad move. More crucially for Monte-Carlo simulation, playouts take vastly longer if these moves are not omitted because the game does not converge to a collection of live groups.

Many programs use heavier playouts, using domain-specific knowledge to bias the choice of random moves. A heavy policy can be designed by hand or with the use of a machine learning technique (Bouzy and Chaslot, 2006; Coulom 2007; Gelly and Silver, 2007).

In the limit, a perfect sampling policy would lead to perfect play, as only one playout would be needed for each move from the root. Counterintuitively, a sampling policy which chooses better moves can actually hurt overall performance (Chaslot et al, 2009). This can happen for three reasons: the policy may be so computationally intensive that the number of playouts per second is severely reduced; the policy may be so deterministic that the diversity of the sampling suffers; or the policy may be biased in a way that favors one player over the other (Silver and Tesauro, 2009).

To date, sampling policies have generally been *static*, in that the policy is not changed during play (although it may be modified off-line). The central contribution of this paper is to propose a *dynamic* policy incorporating information from previous playouts.

## 4.2. The Last-Good-Reply Policy

If a move white A was followed by black B and then black won the playout, it is reasonable for black to reply in the same way in future playouts. Here a "move" refers only to playing at a particular point on the board, without any context. Black B might not be a good reply in a subsequent playout where different moves are played before white A, but it is worth trying (assuming it is legal).

The *last-good-reply policy* maintains, for each color and each point on the board, the last successful reply. When it is time to choose a move (in sampling beyond the search tree), this policy looks up the last winning reply to the previous move. If there has been such a reply and it is legal, that move is played. If not, the default policy is used. After each playout, the winning player's stored replies are updated with moves made during that playout.

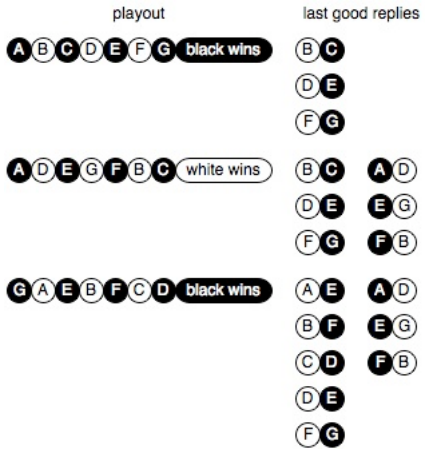


**Figure 3**: Updates in the last-good-reply policy.

Figure 3 illustrates how the last-good-reply table is updated. After the first playout, black learns that a good reply to white B is black C. Similarly, D leads to E and F leads G. After the second playout, white learns three good replies. After the third playout, black changes the best reply to B (replacing C with F) and learns replies to A and C.

A pass is never stored as the best reply. The tables are retained from one turn to the next, thus maintaining local search information.

A reply is dislodged from the table when some other reply to the same move wins a playout. Good replies, to which there are few alternatives, therefore tend to stay in the table longer. Forced replies should be very firmly entrenched.

This policy requires only a small amount of memory: 2*a* integers, where *a* is the area of the board (e.g., 361). All access to the table consists of atomic reads and writes, so this policy does not interfere with parallelism.

# 5. EXPERIMENTAL RESULTS

## 5.1. Improved Playing Strength

To test the effectiveness of the last-good-reply policy, experiments were run comparing plain RAVE (RAVE) with RAVE augmented by the new policy (LastGoodReply). In each condition, Orego version 7.00 played 500 games (250 as black, 250 as white) against GNU Go 3.7.11. All games used Chinese scoring and 7.5 points komi.

The experiments were run on Macs running OS 10.5.8. Orego was run using Java 1.6.0_15, using the command-line options -ea (enable assertion), -server (server mode, turning on the just-in-time compiler), and -Xmx1024 (allocating extra memory).

Orego contains a number of other features not described in this paper, such as a static playout policy similar to that described in Gelly et al (2006). (In the LastGoodReply condition, this policy only comes into play when the reply table does not provide a legal move.) The RAVE bias parameter was set to 0. All other features were left at their default values.

9x9 results are shown in Figure 4. 19x19 results are shown in Figure 5.

The last-good-reply policy provides no improvement on the 9x9 board. On the 19x19 board, on the other hand, it has a significantly higher win rate at 32000 and 64000 playouts per move (two-tailed z-test, $p < 0.002$ and 0.001 respectively).

Additional evidence of the value of the last-good-reply policy is Orego's rank on the KGS Go Server. This rank improved from 11 kyu to 8 kyu after the new policy was incorporated. In server games, Orego's pondering feature (thinking during the opponent's turn) and opening book were used.

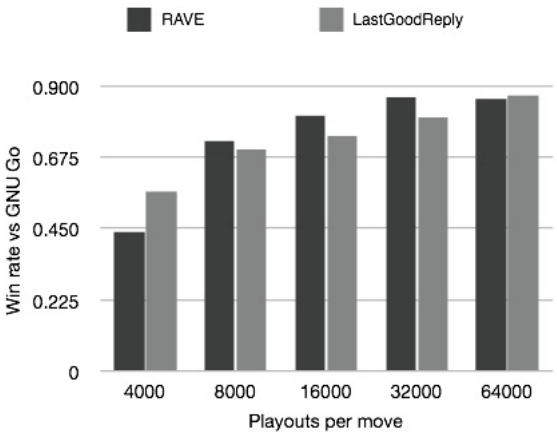


**Figure 4**: 9x9 performance against GNUGo.

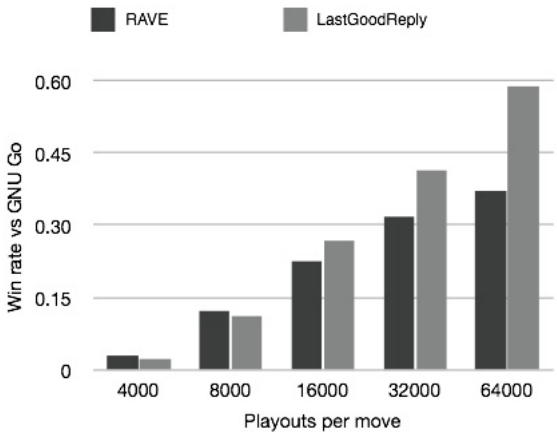


**Figure 5**: 19x19 performance against GNUGo.

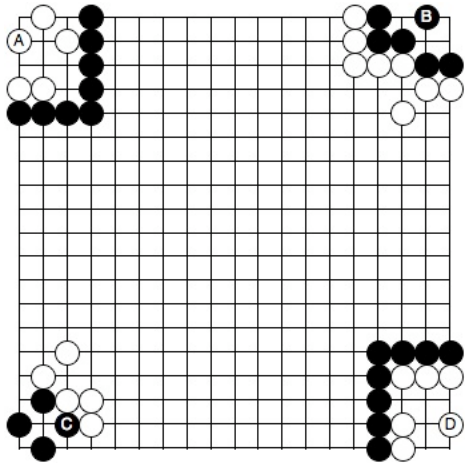


**Figure 6**: Local search.

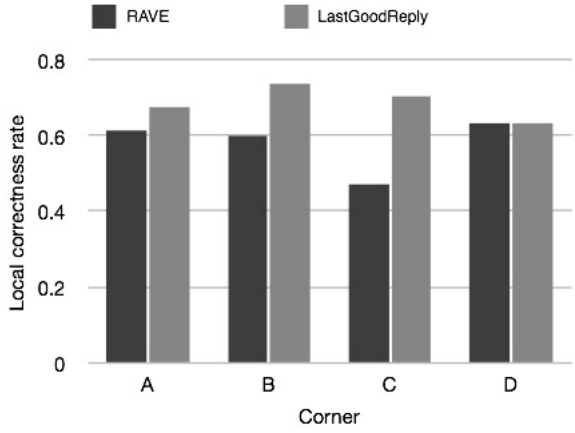


**Figure 7**: Local search performance.

### 5.2. Local Search

A second experiment explored whether the new policy really improves local search. The situation shown in Figure 6 includes four different life-and-death problems taken from Bozulich (2002). In each corner, the surrounded defender must play the indicated move to live. To the degree that a program is performing successful local search, it will play correctly in each corner during each playout. Correct local play was defined as either (a) the defender playing the indicated move before any other stone was played in the 3x3 area including the corner, or (b) the attacker playing in this 3x3 area before the defender. (This definition is slightly generous, as it includes a few unsuccessful attacking moves.)

In each run of this experiment, Orego was run for 4000 playouts on this position in each condition. The fraction of playouts in which each corner was played correctly was recorded. The results shown in Figure 7 are means over 50 runs. LastGoodReply outperforms RAVE in every corner. The results are statistically significant in all but corner D (one-tailed t-test, $p < 0.001$).

The percentage of correct plays is nearly identical for any number of playouts from 4000 to 64000.

## 6. DISCUSSION AND FUTURE WORK

This paper introduced a new dynamic forced-reply policy for MCTS. This policy stores successful local replies and uses them, whenever possible, during sampling beyond the search tree. This provides a significant improvement in

playing strength with large numbers of playouts. Each playout is more likely to be locally correct when using this policy.

In some ways, this work echoes the earliest Monte-Carlo Go work, predating MCTS (e.g., Brügmann, 1993). These early programs, with dynamic policies but no search trees, were not as strong as modern MCTS-based programs. The search tree both puts better moves into the reply table and protects the program against overgeneralizing from that table.

This policy is similar to the Predicate-Average Sampling Technique (PAST) (Finnsson and Björnsson, 2009). PAST was applied in the domain of general game-playing, where the board state is described only by predicate logic statements. The last-good-reply policy effectively pays attention only to "predicates" of the form "the previous move was X". PAST maintains a win rate for each predicate-move pair and uses Gibbs sampling to choose a move matching a predicate. The present policy simply chooses the last successful reply, assuming it is legal.

Both of these techniques are related to the history heuristic used in computer Chess (Schaeffer, 1989). That heuristic counts alpha-beta cutoffs rather than playouts won, but the key insight is the same: a move that worked well elsewhere merits consideration. PAST, the last-good-reply policy, and (within the MC search tree) RAVE all refine this idea to use information about a move's history when played in similar situations. In PAST, similar situations are those sharing a predicate. For the last-good-reply policy, similar situations are those having the same previous move. For RAVE, situations occurring later in a playout are considered similar.

One might be concerned that this policy would lead to a catastrophic loss of playout diversity, with the same playout being repeated over and over again. (This is presumably why Finnsson and Björnsson (2009) used Gibbs sampling.) Loss of diversity is not a problem for three reasons. First, if one branch of the search tree is always losing, the losing player will switch to a different branch. Second, it is unlikely that every move in a playout will be found in the reply table; once any random move is played, the sequence cannot be followed in exactly the same way. Third, keeping track of only the most recent winning reply adds some useful noise to the system.

Empirically, when Orego is run from the empty board, about 40-45% of the moves in the first 100,000 playouts are stored replies; the rest come from the static policy. Further examination of the updating and use of the best reply table is in order.

The last-good-reply policy seems quite robust to minor variations in implementation. Taking care to store a move only the first time it is encountered in a playout has no perceivable effect on performance. A policy of storing the last good reply to the last *two* moves also does not improve performance. Storing only the moves that occur in the search tree (which have been more thoroughly vetted) results in a slightly weaker player.

It would be possible to devise a version of this policy maintaining a win rate for each reply, rather than the last winning reply. The policy could then choose the move with the highest win rate. Even if the considerable time and memory costs of such bookkeeping could be overcome, there might be a price in sampling diversity: any reply that fared badly in the first few playouts where it was tried might never be tried again. Some way of avoiding this problem would be needed, such Gibbs sampling (Finnsson and Björnsson, 2009) or an epsilon-greedy strategy (Sutton and Barto, 1998).

## 7. ACKNOWLEDGMENTS

The author would like to thank Yung-Pin Chen for his help with statistical tests and the anonymous reviewers for their many helpful comments.

## 8. REFERENCES

Baker, K. (2008). *The Way to Go, Revised Seventh Edition*. American Go Association: New York, NY. http://www.usgo.org/usa/waytogo/index.html

Bouzy, B. and Chaslot, G. (2006). Monte-Carlo Go Reinforcement Learning Experiments. In *IEEE 2006 Symposium on Computational Intelligence in Games*, Reno, NV, USA, pp. 187-194.

Bozulich, R. (2002). *One Thousand and One Life-and-Death Problems.* Kiseido: Tokyo. ISBN 4-906574-72-6.

Brügmann, B. (1993). Monte carlo go. http://www.ideanest.com/vegos/MonteCarloGo.pdf

Cai, X. and Wunsch, D. (2007). Computer go: A grand challenge to AI. *Studies in Computational Intelligence*, Vol 63, pp. 443-465.

Chaslot, G. et al. (2009). Adding expert knowledge and exploration in Monte-Carlo Tree Search. In *Proceedings of the Twelfth Annual Advances in Computer Games Conference* (ACG'09), Pamplona, Spain, May 11-13.

Coulom, R. (2007). Computing Elo Ratings of Move Patterns in the Game of Go. In H. J. van den Herik et al, ed., *Proceedings of the Computer Games Workshop 2007 (CGW 2007)*, pp. 113-124.

Finnsson, H. and Björnsson, Y. (2009). Simulation Control in General Game Playing Agents. In *The IJCAI Workshop on General Game Playing (GIGA '09)*, pp. 21-26.

Garlock, C. (2008). Computer Beats Pro at U.S. Go Congress. *American Go E-Journal*, Vol. 9, No. 40. http://www.usgo.org/EJournal/archive/20080807/20080807.htm

Gelly, S. et al. (2006). Modification of UCT with Patterns in Monte-Carlo Go. Technical Report 6062. INRIA, France.

Gelly, S. and Silver, D. (2007). Combining Offline and Online Knowledge in UCT. In *ICML'07: Proceedings of the 24th International Conference on Machine Learning*, pp. 273-280. Association for Computing Machinery.

Kocsis, L. and Szepesvári, C. (2006). Bandit Based Monte-Carlo Planning. In *15th European Conference on Machine Learning*, pp. 282-293.

Müller, M. (2002). Computer go. *Artificial Intelligence*, Vol. 134, No. 1-2, pp. 145-179.

Russell, S. and Norvig, P. (2003). *Artificial Intelligence: A Modern Approach, Second Edition*. Prentice Hall: Upper Saddle River, NJ. ISBN 0-13-790395-2.

Schaeffer, J. The history heuristic and alpha-beta search enhancements in practice. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11:1203-1212.

Silver, D. and Tesauro, G. (2009). Monte-Carlo Simulation Balancing. In *Proceedings of the 26 Annual International Conference on Machine Learning*, Montreal, Quebec, Canada, pp. 954-852.

Sutton, R. and Barto, A. (1998). *Reinforcement Learning: An Introduction*. MIT Press: Cambridge, MA. ISBN 978-0262193986.

Wedd, N. (2009). Human-Computer Go Challenges. http://www.computer-go.info/h-c/index.html