The Power of Forgetting: Improving the Last-Good-Reply Policy in Monte Carlo *Go*

Hendrik Baier and Peter D. Drake

***Abstract*—The dominant paradigm for programs playing the game of *Go* is Monte Carlo tree search. 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*. The recently published last-good-reply policy makes moves that, in previous playouts, have been successful replies to immediately preceding moves. This paper presents a modification of this policy that not only remembers moves that recently succeeded but also immediately forgets moves that recently failed. This modification provides a large improvement in playing strength. We also show that responding to the previous two moves is superior to responding to the previous one move. Surprisingly, remembering the win rate of every reply performs much worse than simply remembering the last good reply (and indeed worse than not storing good replies at all).**

***Index Terms*—Board games, *Go*, machine learning, Monte Carlo methods.**

I. Introduction

GO [2] is a deterministic, zero-sum, two-player game of perfect information. Writing programs to play *Go* well stands as a grand challenge of artificial intelligence [5]. 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. Even this performance is the result of a recent breakthrough: before 2008, programs were unable to defeat professional players on the full 19×19 board despite enormous handicaps [12], [17].

This breakthrough, which lies at the core of all strong, modern *Go* programs, is Monte Carlo tree search (MCTS) [16], [7]. This algorithm combines classical artificial intelligence (tree search) with statistical sampling.

The next section of this paper describes MCTS in detail. Various techniques for learning within MCTS are then reviewed. Last-good-reply policies, including Drake's original policy [9] and the new variants introduced in this paper, are covered in another section. Experimental results are followed by conclusions and future work.

Manuscript received June 25, 2010; revised October 06, 2010; accepted December 06, 2010. Date of publication December 20, 2010; date of current version January 19, 2011.

H. Baier is with the Institute of Cognitive Science, University of Osnabrück, Osnabrück 49069, Germany (e-mail: hbaier@uos.edu).

P. D. Drake is with the Department of Mathematical Sciences, Lewis & Clark College, Portland, OR 97219 USA (e-mail: drake@lclark.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCIAIG.2010.2100396

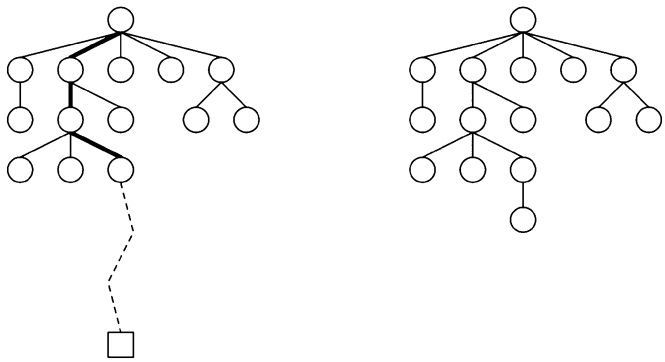

Fig. 1. Monte Carlo Tree Search.

II. Monte Carlo Tree Search

MCTS grows a search tree by repeatedly simulating complete games (Fig. 1). Each such simulation, called a playout, consists of three phases: selection, sampling, and backpropagation. Selection (bold lines) chooses a sequence of moves within the search 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 [16].

III. Learning in Monte Carlo Tree Search

MCTS can be viewed as an inductive machine learning technique in that its behavior changes based on the results of past playouts. Within each playout, the algorithm selects moves that have previously fared well in similar game states. Much hinges on how the sets of "similar" states are defined. If they are too small, very little information is available for each state. If they are too large, the information becomes extremely noisy.

In the discussion that follows, a state of the game is identified with the sequence of moves leading to that state. A state s is a

sequence of moves, s_i is the ith move in that sequence, and $|s|$ is the length of the sequence. Thus, $s_{|s|}$ is the last move in the sequence. $N(s)$ is the set of states considered similar to s, i.e., the neighborhood of s.

A. Learning for the Selection Phase

The narrowest definition of similarity has exactly one state in each neighborhood. Thus

$$N(s) = \{s\}.$$

This is the definition used by the pure MCTS algorithm described in the previous section. It is completely safe, but fails to take full advantage of each playout.

At the other extreme would be to have only one neighborhood

$$N(s) = S$$

where S is the set of all possible states.

This algorithm simply maintains win rates for each move regardless of context. It requires very little memory but is not very effective. The earliest work on Monte Carlo *Go* [4] introduced this approach, which is known as all-moves-as-first (AMAF).

A transposition table [19] ignores the history of game states, considering only the stones on the board

$$N(s) = \{r : c(r) = c(s)\}$$

where $c(s)$ is the configuration of the board in state s.

The stated definition is not entirely safe in *Go* because of the ko rule forbidding full-board repetitions: the legality of a move may depend on how the current state was reached. Fortunately, keeping track of the simple ko point (if any) and the color to play catches most collisions. With this addition, the neighborhood of s is the set of states r that 1) have the same configuration of stones, 2) have the same simple ko point, and 3) have the same color to play. Formally

$$N(s) = \{r : c(r) = c(s) \wedge k(r) = k(s) \wedge |r| \equiv |s| (\mathrm{mod} 2)\}$$

where $k(s)$ is the simple ko point in state s (or null if there is no such point). The last conjunct avoids conflating states with different colors to play.

A broader and more powerful definition of similarity is used by rapid action value estimation (RAVE) [14]. In RAVE, the neighborhood of a state consists of its successors, i.e., those states (move sequences) r of which s is a prefix. Formally

$$N(s) = \{r : |r| \geq |s| \wedge \forall t \leq |s|, r_t = s_t\}.$$

This definition is asymmetric. Indeed, it is antisymmetric, i.e., $r \in N(s) \rightarrow s \notin N(r)$ for all $r \neq s$.

In RAVE, moves played directly from a state and from any subsequent states in that playout are taken into account. Because of the large neighborhoods, the data from RAVE are considerably noisier than the raw MCTS data; they are discounted accordingly. On the other hand, RAVE produces many more data per playout, quickly providing information as to which moves are most promising. A surprising side effect of RAVE is that the exploration term becomes unnecessary [6]; moves that have been temporarily abandoned by selection are still available in sampling.

B. Learning for the Sampling Phase

It is much rarer for learning to be applied during the sampling phase. Many programs use "heavy playout" policies that bias the sampling toward better moves. Such a policy may be constructed by hand or learned offline [3], [8], [14]. Dynamically updating the sampling policy is more difficult because of the need for speed, the need for diversity, and the large, complex, sparsely sampled state space.

Two techniques applied outside *Go* are the contextual Monte Carlo (CMC) and the predicate average sampling technique (PAST).

In CMC [18], the neighborhood depends on the current player's previous move $s_{|s|-1}$. Any state where that move was also played by the current player (although not necessarily at time $|s| - 1$) is part of the neighborhood

$$N(s) = \{r : \exists i : r_i = s_{|s|-1} \wedge i \equiv |s| - 1 (\mathrm{mod}\ 2)\}.$$

PAST [11] was applied to the general game-playing problem, where states are defined in predicate logic. The neighborhood consists of states sharing some true predicate with s

$$N(s) = \{r : \exists p : p(r) \wedge p(s)\}$$

where p is chosen from the set of predicates.

The last-good-reply-1 policy (LGR-1) [9], while not domain specific, has been successfully applied to *Go*. In this policy, the neighborhoods consist of all states with the same last move

$$N(s) = \{r : r_{|r|} = s_{|s|} \wedge |r| \equiv |s| (\mathrm{mod}\ 2)\}.$$

This can be thought of as a variation on PAST where the only predicates are of the form "the previous move was X."

The last-good-reply-2 (LGR-2) policy narrows the neighborhood to those states where the last two moves are the same

$$N(s) = \{r : r_{|r|} = s_{|s|} \wedge r_{|r|-1} = s_{|s|-1} \wedge |r| \equiv |s| (\mathrm{mod}\ 2)\}.$$

Similar neighborhoods were foreshadowed in [4], where the use of win rates across the neighborhood was suggested, as well as [15], where the neighborhoods have been used for move ordering in an alpha–beta context.

The experiments in this paper, based on the work of the first author in [1], focus on last-good-reply policies. These policies are described in more detail in Section IV.

IV. Last-Good-Reply Policies

A. Last-Good-Reply-1

Each move in a game (except the first) can be thought of as a reply to the previous move. A reply is successful if the player who makes the reply goes on to win the playout. LGR-1 maintains, for each move (including location and color), the last successful reply to that move. Only one reply is stored for each move.

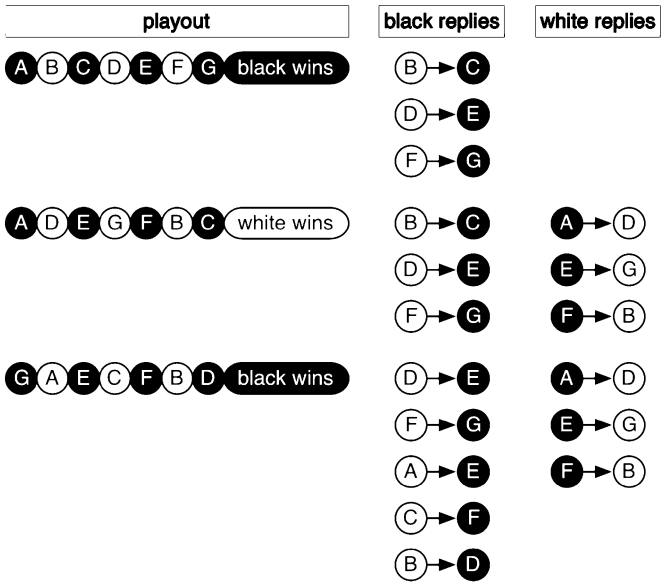

Fig. 2. Updating the reply table in LGR-1.

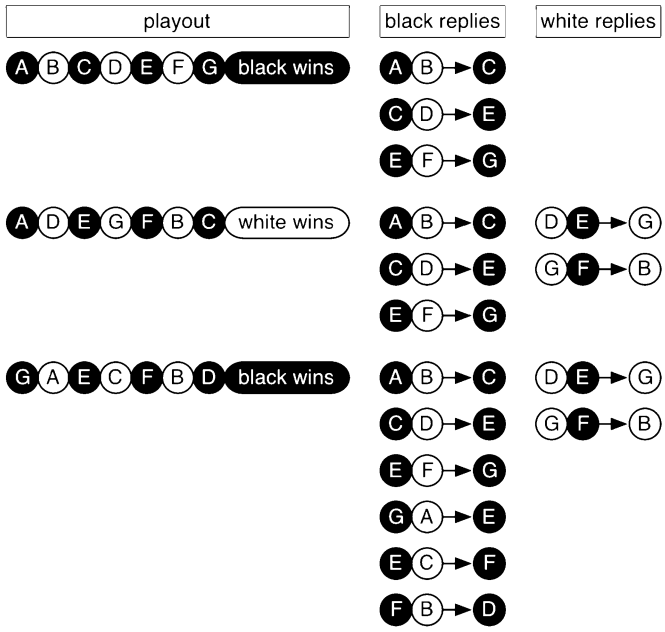

Fig. 3. Updating the reply table in LGR-2.

In the backpropagation phase at the end of each playout, each of the winner's moves is stored as a good reply to its predecessor. Any previously stored reply is overwritten.

The updating of the reply table is illustrated in Fig. 2. In the first playout, the program learns that a good reply to white B is black C. This is written Ⓑ→●C. Black also learns replies Ⓓ→●E and Ⓕ→●G. The second playout stores three good replies for white: ●A→ⒹⒺ→●G, and ●F→Ⓑ. The third playout adds black replies ⒶⒺ→●E and ⒸⒻ→●F and overwrites Ⓑ→●C with Ⓑ→●D.

In the sampling phase, when it is time to choose a move, the last good reply to the previous move is used if possible. It may be that no good reply has yet been stored. The stored reply may also be illegal because the current context differs from that where the reply was originally played. In either of these cases, the default sampling policy is used instead.

B. Last-Good-Reply-2

LGR-2 extends LGR-1 by additionally keeping track of the previous two moves instead of just the previous move. This is illustrated in Fig. 3.

By placing each reply in a more detailed context than LGR-1, LGR-2 narrows the neighborhoods of "similar" states. Move suggestions are therefore based on fewer, more relevant samples.

Whenever the two-move reply table provides no stored answer in the sampling phase, the policy defaults to the one-move reply table of LGR-1, which is maintained separately. These data are less accurate, but more readily available; one-move contexts create larger neighborhoods that are sampled more frequently. (See also the experiments in Section V-D.)

C. Forgetting

In the current work, we found that the last-good-reply policy could be improved with the addition of forgetting. Specifically, at the end of the playout, in addition to learning that the winner's replies were good, we delete any stored replies made by the loser. The version of LGR-1 with forgetting is abbreviated LGRF-1.

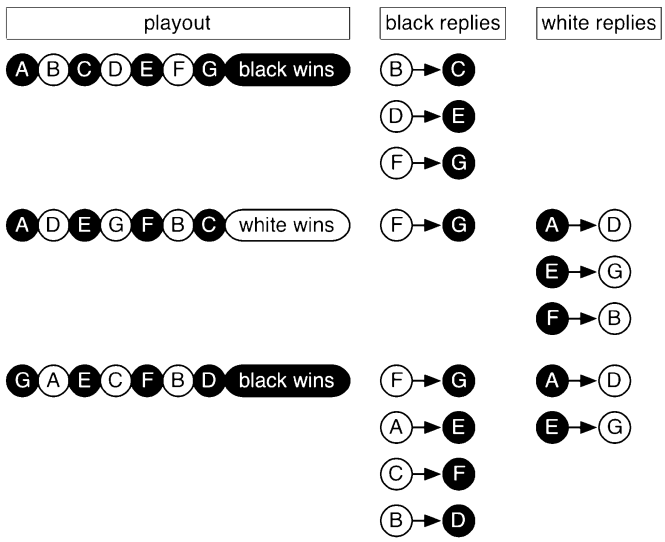

Fig. 4. Updating the reply table in LGRF-1.

This policy is illustrated in Fig. 4. In the second playout, black's replies Ⓑ→●C and Ⓓ→●E are deleted because they were played but black still lost. In the third playout, white's reply ●F→Ⓑ is deleted.

LGRF-2 is defined analogously to LGR-2. Replies to one-move and two-move contexts are stored independently whenever they appear in a winning playout, and deleted whenever they appear in a losing one. When no applicable reply to the previous two moves is available, LGRF-2 falls back to the single-move replies of LGRF-1.

D. Storing Win Rates

If storing the last good reply to the previous move is helpful, it seems reasonable to store the win rates of every possible

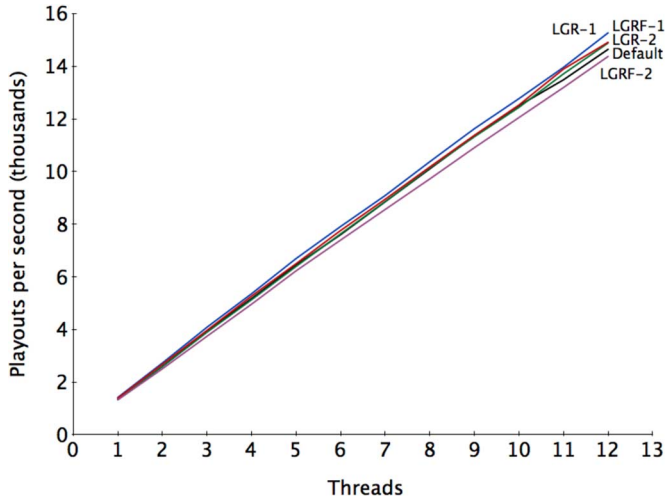

Fig. 5. Speed for various numbers of threads.

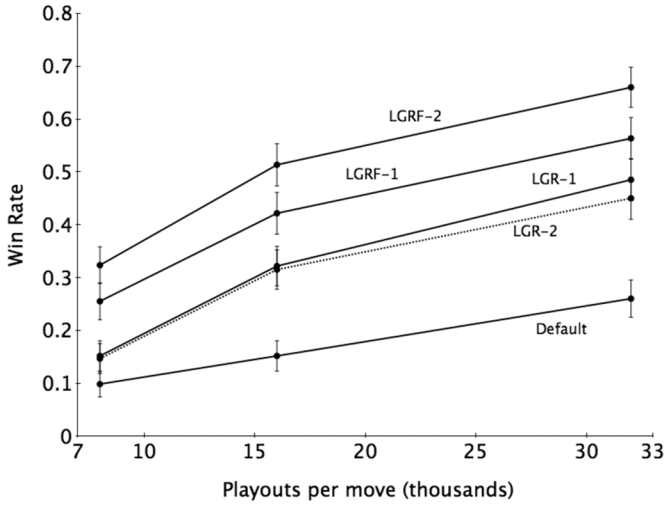

Fig. 6. 19 × 19 win rates versus GNU *Go*.

reply. The Softmax-1 policy maintains such rates. In sampling, Softmax-1 chooses a reply to the previous move according to a Gibbs (or Boltzmann) softmax policy: it chooses move a with probability

$$\exp(\text{winrate}(a)/T) \Big/ \sum_i \exp(\text{winrate}(i)/T)$$

where i sums over all available moves, and T is a temperature parameter.

Softmax-1 could have all of the benefits of a forgetting policy and also make use of more data from previous runs. On the other hand, paying too much attention to the past may limit this policy's ability to respond to changing conditions.

Softmax-2 could be defined analogously, but we did not experiment with such a policy.

V. Experimental Results

A. Methods

All experiments were run using Orego [10] version 7.06. Orego was run on a CentOS Linux cluster of five nodes, each using 8-GB RAM and two 6-core AMD Opteron 2427 processors running at 2.2 GHz, giving the cluster a total of 60 cores. The program was run with Java 1.6, using the command-line options -ea (enabling assertions), -server (server mode, turning on the just-in-time compiler), and -Xmx1024M (allocating extra memory).

The default version of Orego contains a number of other features not described in detail here, including a transposition table, RAVE, and a sampling policy similar to that described in [13].

B. Speed

In order to examine the effect of the last-good-reply policies on the runtime speed of Orego, the speed (in thousands of playouts per second) was measured in a 10-s run starting from an empty board. Fig. 5 shows the averages of ten such runs for different numbers of threads. Since the reply tables are accessed atomically, all threads share the same tables.

Since no tested variant of LGR slowed down Orego by any significant amount, the experiments in the following sections were conducted with constant playouts per move instead of constant time per move.

C. Playing Strength

To test the effectiveness of the various policies described in Section IV, we pitted Orego (using various policies) against GNU *Go* 3.8 running at its default level of 10. All games used Chinese (area) scoring and 7.5 points komi. Both 19 × 19 and 9 × 9 experiments were conducted.

On the 19 × 19 board, 600 games per condition were played by Orego (300 as black, 300 as white). Policies included default Orego, LGR-1, LGR-2, LGRF-1, and LGRF-2. Each policy was run for 8000, 16 000, and 32 000 playouts per move. (These playouts were divided over two threads.) The results are shown in Fig. 6. Error bars indicate 95% confidence intervals.

At all numbers of playouts, the following strength relations are significant ($p < 0.05$, two-tailed z-test):

$$\text{Default} < \{\text{LGR-1}, \text{LGR-2}\} < \text{LGRF-1} < \text{LGRF-2}.$$

The Softmax-1 policy has only been tested with 8000 playouts per move. Across a variety of temperature parameters tested (0.001, 0.01, 0.1, and 1.0), $T = 0.1$ had the most success with only three games won out of 600. As a result, experiments with Softmax were abandoned.

The results are quite clear: forgetting provides a large improvement. Responding to the previous two moves is better than responding to the previous move, but only if forgetting is used. The Softmax-1 policy fails miserably.

On the 9 × 9 board, 1200 games per condition were played by Orego (600 as black, 600 as white). Policies used and playouts per move were identical to the 19 × 19 conditions. The results are shown in Fig. 7.

On the small board, the LGR policies without forgetting are no improvement to the default policy; for 8000 and 16000 playouts, LGR-2 is significantly weaker than even default Orego. LGRF-1 and LGRF-2 do not differ significantly in strength for any tested number of playouts. With the one exception of

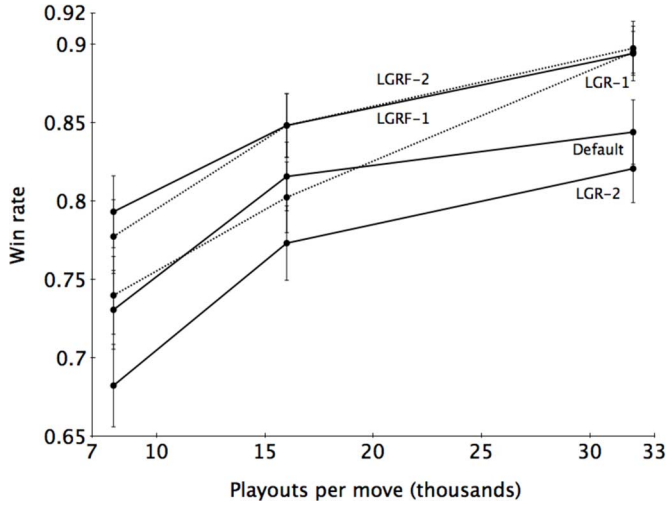

Fig. 7. 9×9 win rates versus GNU *Go*.

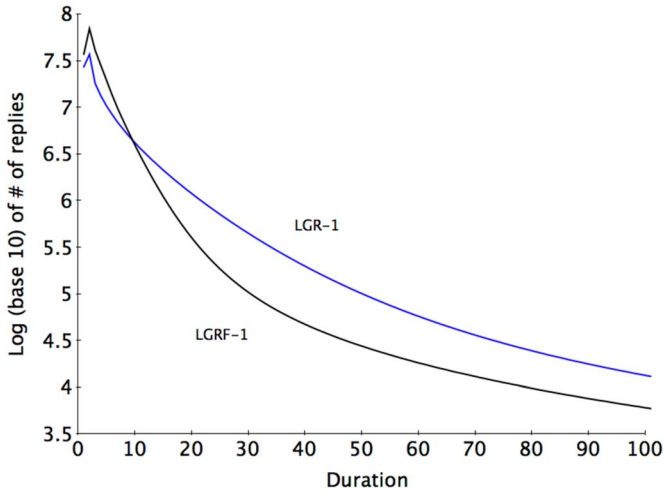

Fig. 8. Number of replies as a function of duration in the reply table.

TABLE I
RESULTS OF LOOKUPS FROM LAST-GOOD-REPLY TABLES

	LGR-1	LGR-2	LGRF-1	LGRF-2
Legal (2)		45.2%		27.7%
Illegal (2)		54.7%		43.5%
None (2)		0.1%		28.8%
Legal	44.5%	23.5%	27.1%	24.0%
Illegal	55.5%	76.5%	43.7%	52.8%
None	0.0%	0.0%	29.3%	23.2%

LGR-1 at 32 000 playouts, both forgetting policies are significantly stronger than default Orego, LGR-1, and LGR-2 at all numbers of playouts.

D. Reply Table Contents and Usage

To better understand the effects of forgetting, several experiments on the contents of the reply tables were conducted.

In the first experiment, we determined the proportion of table accesses that return a playable reply, as opposed to an illegal move or no stored reply. Table I shows the results averaged over 100 19×19 games of Orego against itself, playing with 8000 playouts per move. For LGR-2 and LGRF-2, the first three rows of the table show the results of probing the two-move reply tables. For those times when Orego found an illegal or no move there, the last three rows show the results of accessing the one-move reply tables as fallback. (This is why those columns do not add up to 100%.) For example, in the LGRF-2 policy, 27.7% of the table lookups resulted in legal moves. Of those that did not, 24.0% found legal moves in the fallback LGRF-1 policy.

LGRF policies return fewer legal moves than LGR policies and fall back to the default policy more often. The reason is that there are more empty slots in the reply tables, which can only appear at the very beginning of the search in nonforgetting policies. Forgetting reduces the content of the reply tables to more recently successful move answers. Falling back to the default policy allows static knowledge from that policy to affect the reply tables.

In the second experiment, the average duration of moves in the reply tables was measured. In 100 games of self-play (as above), we found average durations of 8.639 (LGR-1), 4.185 (LGRF-1), 2544.958 (LGR-2), and 1700.780 (LGRF-2) playouts. The durations for the policies remembering two moves are much larger because pairs of consecutive moves occur less often than individual moves, offering fewer opportunities to update the reply tables.

The third experiment examined the distribution of durations for LGR-1 and LGRF-1 in more detail. Over 100 games of self-play, we gathered data on how many replies survived for various numbers of playouts. The results are shown in Fig. 8. Note that the vertical axis is logarithmic.

The results show two things. First, most replies are removed from the table very quickly. Second, forgetting sharpens this distribution.

Taken together, these experiment demonstrate that forgetting increases the fluctuation of table entries. Unless a reply *consistently* wins playouts in which it is played, it is more quickly deleted. (It may reappear later if generated again by the default policy.) This increases the algorithm's exploration of alternate replies.

E. Locality

In [9], Drake argued that the LGR-1 policy allows MCTS to perform local search. To further test this hypothesis, we gathered data on the Euclidean distance between each reply and its predecessor as a function of the reply's duration. The results, averaged over 100 games of self-play, are shown in Fig. 9.

Replies that survive longer in the table are more local, i.e., they are closer to their predecessors than more short-lived replies. This effect is stronger with forgetting. Furthermore, forgetting reduces the mean distance at all durations.

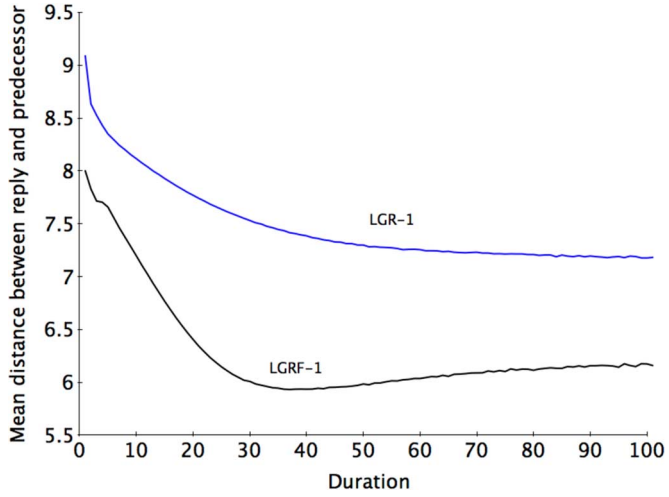

Fig. 9. Distance as a function of duration in the reply table.

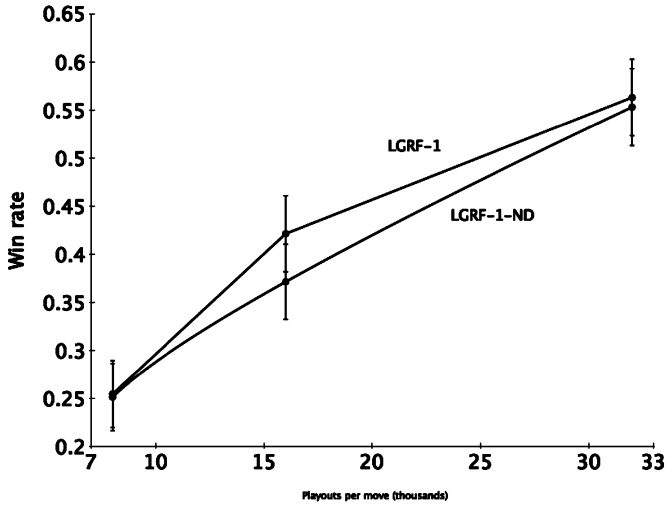

Fig. 10. Effect of no duplicate variant of LGRF-1.

F. Repeated Moves

In most playouts, many moves appear repeatedly. The LGRF policies as described above store every move played by the winning player as a good reply. Thus, if a move appears more than once in a playout (due to capturing), only the last reply in the playout remains in the table. This is conceivably harmful, as the board will have changed so much by the end of a playout that replies found then will be less relevant to other playouts than replies found at the beginning.

In the last experiment described in this paper, an alternative version of LGRF-1 was tested that saves replies only to the first appearance in a playout of any given move. The policy is called LGRF-1-ND (no duplicates). Fig. 10 compares its success to that of LGRF-1 (over 600 games, as in Section V-C) as described above.

No significant difference in performance could be found.

VI. Conclusion and Future Work

This paper has reviewed learning in MCTS, including learning within the tree in the selection phase (transposition tables and RAVE) and beyond the tree in the sampling phase (CMC, PAST, and last-good-reply policies). All of these can be described as using information from previous playouts when a given move was played in "similar" situations, using different definitions of "similar."

We have presented three new policies: forgetting versions of last-good-reply-1 (LGRF-1) and last-good-reply-2 (LGRF-2) and a policy that maintains win rates for each reply (Softmax-1). The first two policies offered significant improvements, with LGRF-2 being stronger in 19×19 *Go*. The Softmax-1 policy was significantly weaker than a default policy, which uses no learning in the sampling phase.

We believe the addition of forgetting to last-good-reply policies makes them stronger for two reasons. First, forgetting preserves diversity. A "forced" move will only persist as a stored reply if it consistently wins; in any other situation, exploration is preferable. Second, forgetting allows the algorithm to adapt more rapidly to changing conditions. If a given reply fares well in one context, the opponent will likely make different moves earlier in each playout, thus changing the context. With forgetting, the reply only persists if it is also good in this new context.

Both of these effects of forgetting—preserving diversity and adapting to changing contexts—also explain the very poor performance of the Softmax-1 policy.

In 9×9 *Go*, the addition of forgetting allows the last-good-reply policies to become beneficial. Still, their effect is much weaker than on the 19×19 board. One possible explanation is that the locally and tactically oriented heuristics of Orego's default policy (capturing, escaping capture, and 3×3 patterns) are more relevant in 9×9 than in 19×19 *Go*, making them more effective than reacting to the opponent's last move. Another hypothesis is that given a good (often local) reply, the 19×19 board allows for a wide variety of positions in which the reply is useful, because many distant moves do not affect its validity. On the 9×9 board, in contrast, many moves are likely to either invalidate a reply (because they are nearby) or render it irrelevant (because they greatly affect the score).

Our results strongly suggest that the ideal neighborhood of "similar" states from which to draw information is neither minimal (only precisely the current state) nor maximal (all states). Specifically, two previous moves of context are more effective than one. It may be that three moves of context would fare even better, although this would potentially require storing replies for $2 \times 361^3 > 94$ million different contexts. Alternately, it may be that situations where the antepenultimate move is important are so rare as to be irrelevant.

Acknowledgment

The authors would like to thank the anonymous reviewers for their many helpful comments and suggestions.

References

[1] H. Baier, "Adaptive playout policies for Monte Carlo Go," M.S. thesis, Inst. Cogn. Sci., Osnabrück Univ., Osnabrück, Germany, 2010.
[2] K. Baker, *The Way to Go*, 7th ed. New York: American Go Association, 2008.
[3] B. Bouzy and G. Chaslot, "Monte-Carlo Go reinforcement learning experiments," in *Proc. IEEE Symp. Comput. Intell. Games*, Reno, NV, 2006, pp. 187–194.
[4] B. Brügmann, "Monte Carlo Go," 1993 [Online]. Available: http://www.ideanest.com/vegos/MonteCarloGo.pdf

[5] X. Cai and D. Wunsch, "Computer Go: A grand challenge to AI," *Studies in Comput. Intell.*, vol. 63, pp. 443–465, 2007.
[6] G. M. J.-B. Chaslot, C. Fiter, J.-B. Hoock, A. Rimmel, and O. Teytaud, "Adding expert knowledge and exploration in Monte-Carlo tree search," in *Proc. 12th Annu. Adv. Comput. Games Conf.*, Pamplona, Spain, May 11–13, 2009, pp. 1–13.
[7] R. Coulom, "Efficient selectivity and backup operators in Monte-Carlo tree search," in *Lecture Notes in Computer Science*. Berlin, Germany: Springer-Verlag, 2007, vol. 4630, pp. 72–83.
[8] R. Coulom, "Computing Elo ratings of move patterns in the game of Go," in *Proc. Comput. Games Workshop*, H. J. van den Herik, M. Winands, J. Uiterwijk, and M. Schadd, Eds., 2007, pp. 113–124.
[9] P. Drake, "The last-good-reply policy for Monte-Carlo Go," *Int. Comput. Games Assoc. J.*, vol. 32, no. 4, pp. 221–227, Dec. 2009.
[10] P. Drake, A. Hersey, and N. Sylvester, "Orego Go Program," [Online]. Available: http://legacy.lclark.edu/~drake/Orego.html
[11] H. Finnsson and Y. Björnsson, "Simulation control in general game playing agents," in *Proc. IJCAI Workshop General Game Playing*, 2009, pp. 21–26.
[12] C. Garlock, "Computer beats pro at U.S. Go Congress," *Amer. Go E J.*, vol. 9, no. 40, 2008.
[13] S. Gelly, Y. Wang, R. Munos, and O. Teytaud, "Modification of UCT with patterns in Monte-Carlo Go," INRIA, France, Tech. Rep. 6062, 2006.
[14] S. Gelly and D. Silver, "Combining offline and online knowledge in UCT," in *Proc. 24th Int. Conf. Mach. Learn.*, 2007, pp. 273–280.
[15] J. Hashimoto, T. Hashimoto, and H. Iida, "Context killer heuristic and its application to computer shogi," in *Proc. Comput. Games Workshop*, H. J. van den Herik, M. Winands, J. Uiterwijk, and M. Schadd, Eds., 2007, pp. 39–48.
[16] L. Kocsis and C. Szepesvári, "Bandit based Monte-Carlo planning," in *Proc. 15th Eur. Conf. Mach. Learn.*, 2006, pp. 282–293.
[17] M. Müller, "Computer Go," *Artif. Intell.*, vol. 134, no. 1–2, pp. 145–179, 2002.
[18] A. Rimmel and F. Teytaud, "Multiple overlapping tiles for contextual Monte Carlo tree search," in *Lecture Notes in Computer Science*. Berlin, Germany: Springer-Verlag, 2010, vol. 6024, pp. 201–210.
[19] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed. Upper Saddle River, NJ: Pearson, 2010, pp. 170–171.

Hendrik Baier received the B.Sc. degree in computer science from Darmstadt Technical University, Darmstadt, Germany, 2006. Currently, he is working towards the M.S. degree in cognitive science at Osnabrück University, Osnabrück, Germany.

Peter D. Drake received the B.A. degree in English from Willamette University, Salem, OR, in 1993, the M.S. degree in computer science from Oregon State University, Corvallis, in 1995, and the Ph.D. degree in computer science and cognitive science from Indiana University, Bloomington, in 2002.

Currently, he is an Associate Professor of Computer Science at Lewis & Clark College, Portland, OR. He is the author of *Data Structures and Algorithms in Java* (Upper Saddle River, NJ: Prentice-Hall, 2006). He is the leader of the Orego project, which has explored machine learning approaches to *Go* since 2002.

Dr. Drake is a member of the Association for Computing Machinery.