Two Online Learning Playout Policies in Monte Carlo *Go*: An Application of Win/Loss States

Jacques Basaldúa, Sam Stewart, J. Marcos Moreno-Vega, and Peter D. Drake

***Abstract*—Recently, Monte Carlo tree search (MCTS) has become the dominant algorithm in Computer Go. This paper compares two simulation algorithms known as playout policies. The base policy includes some mandatory domain-specific knowledge such as seki and urgency patterns, but is still simple to implement. The more advanced learning policy combines two different learning algorithms with those implemented in the base policy. This policy makes use of win/loss states (WLSs) to learn win rates for large sets of features. A very large experimental series of 7960 games includes results for different board sizes, in self-play and against a reference opponent: FUEGO. Results are given for equal numbers of simulations and equal central processing unit (CPU) allocation. The improvement is around 100 Elo points, even with equal CPU allocation, and it increases with the number of simulations. Analyzing the proportion of moves generated by each part of the policy and the individual impact of each part provides further insight on how the policy is learning.**

***Index Terms*—Knowledge discovery, Monte Carlo methods, statistical learning, stochastic systems.**

I. Introduction

In the last five years, Monte Carlo tree search (MCTS) [1] has become the dominating algorithm in Computer Go. This paper takes basic knowledge about the ancient Asian game of *Go* for granted. *Go* rules and basic concepts can be found on the Internet [2].

In MCTS, the simulation that follows the tree search once a leaf is reached is called a playout. The algorithm generating it is called a playout policy. When applying MCTS in complicated domains like *Go*, a uniform random policy is known to be ineffective, both because it misevaluates life and death more than 50% of the time in some simple cases [see Fig. 2(I)] and because it kills living groups without some appropriate rules for handling eye shape and seki. In Section II, we describe a simple but state-of-the-art playout policy named the base policy. The base policy includes some mandatory domain-specific knowledge like seki and urgency patterns, but is still simple to implement. It does not handle advanced topics such as semeai or ko. Then, we describe a modification to it known as the learning policy which combines two different algorithms: "node" moves and "edge" moves. This policy is not strong enough to beat a strong program such as FUEGO [3] on equal numbers of simulations, but wins enough games using FUEGO as a reference opponent given four times as many simulations (see Section IV). This policy makes use of win/loss states (WLSs) [4], [5] to learn win rates for large sets of features. The aim of this paper is proving the superiority of the learning policy over the base policy, not analyzing the efficiency of WLS. A very large experimental series includes results for different board sizes, in self-play and against a reference opponent: FUEGO. Results are given for equal numbers of simulations and equal central processing unit (CPU) allocation.

Online learning applied to playout policies in games and in *Go* is an active research area [6], [7]. Our implementation combines online learning with the efficiency of WLS and a set of automatically written assembly language functions named the hologram board system (HBS) [8], which makes the fast computation of local contexts (see Section III-B) possible. We have released HBS as free software to encourage replication studies and implement learning playouts in both research and commercial programs.

Manuscript received December 19, 2012; revised April 25, 2013 and July 14, 2013; accepted November 06, 2013. Date of publication November 26, 2013; date of current version March 13, 2014. This work was supported in part by the European Regional Development Fund and the Spanish Government under Project TIN2012-32608.

J. Basaldúa and J. M. Moreno-Vega are with the Departamento de Estadística IO y Computación, Universidad de La Laguna, La Laguna, Tenerife 38271, Spain (e-mail: jacques@dybot.com; jmmoreno@ull.es).

S. Stewart and P. D. Drake are with Lewis & Clark College, Portland, OR 97219 USA (e-mail: sstewart@lclark.edu; 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.2013.2292565

II. Base Playout Policy

This section describes the base policy. Sections II-A–II-E describe the individual steps in it. These steps are also included in the learning policy.

A. Stopping Rule

The board system manages move illegality according to standard *Go* rules (suicide is not allowed and immediate ko recapture is not allowed). The playout ends after two consecutive passes. Also, playing single-point eye filling moves is illegal in the playout as a necessary stopping rule. The standard one point eye shape definition is an empty intersection with all four immediate neighbors, either own color or off the board. When intersections at distances $(\pm 1, \pm 1)$ are cut twice or more by opponent stones or once if some neighbors are off the board, the resulting shape is not an eye.

B. Self-Atari and Seki

Each time a move is generated (all moves without exception no matter what part of the policy generated it), it is checked for any violation of self-atari policy $p_{\text{self-atari}}$. If violation has occurred, the move is temporarily made illegal, and the playout policy's move generator is called on again for another move

which will also be tested by the self-atari policy and either played or made illegal temporarily. Before a move is played, all temporarily illegal moves are made legal again to keep the board updated. The self-atari policy $p_{\text{self-atari}}$ used in this paper is: "Don't play self-atari moves when the resulting shape is not a known dead nakade shape like: 1-stone, 2-stone, straight 3, bent 3 (with one exception), squared 4, pyramid 4, crossed 5 or bulky 5." The exception to the bent 3 shape is: The shape only applies if the last stone is one of the extreme stones in the bent 3, otherwise this would destroy a known seki shape [see Fig. 2(III)].

C. Atari/Capture Tactical Moves

The board system tracks each time a move sets groups in atari. It also tracks when groups already in atari have increased their size. Both things may happen simultaneously. Additionally, a counter is increased each time any of these events is true and cleared when neither is true. This counter is named t. The policy is randomized to avoid deterministic behavior. This helps giving different priorities to ladder related events. It is usual in *Go* to play some moves from a losing ladder for a good reason, usually improving the shape of an adjacent group or destroying an opponent's eye shape, but it does not make sense to play a losing ladder to the bitter end. Keeping track of the number of consecutive atari tactical moves allows playing some moves without playing the whole sequence. But when enough moves have been played, the ladder must be played out to the end on every move to prevent bias to one or other of the players. This tactic includes capturing groups set into self-atari by the last move or groups that failed to escape from the previous atari. The probability of an atari/capture tactical move being played is stored in an array $p_{\text{AT}}[t]$. Candidate values for this parameter contain some initial probability increased linearly, becoming 1 for all moves above some value, resulting in only two degrees of freedom. (Values are given in Section IV-A.)

D. Urgent Response Patterns

These patterns define urgent answers around the last move based on 3×3 patterns. The eight neighbors of the last move are checked to see if they can be classified in any of the patterns shown in Fig. 1. If this is the case, they are played with a probability p_{MP}. These patterns are sometimes known as Mogo-style patterns, since they are the same set created by Yizao Wang for the original Mogo program written by Sylvain Gelly [9] based on the idea first suggested by Bruegman [10], and first shown to be efficient by Bouzy [11].

E. Shape Improvement Moves

Each time a random move is generated, the candidate position is tested for the number of adjacent empty intersections in its four immediate neighbors. When the move has only one empty adjacent intersection, that intersection is played instead when it is legal and has more than one free adjacent intersection.

In all the cases shown in Fig. 2(I) and (II), the potentially right move dominates the wrong moves. For black, it fixes the shape and makes the wrong moves one point eyes and, therefore, not playable, while the wrong moves destroy the shape. For white,

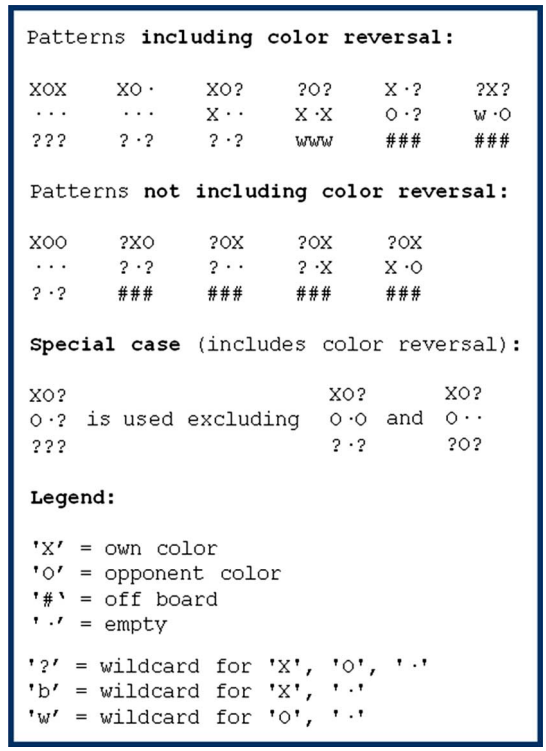


Fig. 1. All urgent response patterns supported by isGO.

the wrong moves achieve nothing, losing one opportunity to destroy the eye shape, while the potentially right move destroys the shape.

Playouts do not understand the urgency of playing the potentially right move. That would require analysis about other possible ways of living. Whether the potentially right move is urgent has to be decided in the tree.

F. Policy in Pseudocode

Fig. 3 describes the move generation in the base policy in pseudocode.

The policy generates moves for both players alternately, including the restrictions mentioned in Sections II-A and II-B, until two consecutive pass moves are generated. When finished, the board is evaluated considering all remaining stones alive. The base policy already includes all necessary elements of a strong program, at least stronger than the seminal policy implemented in MoGo which has influenced most strong programs.

III. Learning Playout Policy

A. Rationale: The "Loose Tree" Analogy

One of the weaknesses of Monte Carlo methods without a tree, that becomes a strength in MCTS, is handling answers to moves, i.e., understanding the difference between the board states in which a move is good and the board states in which the same move is bad. In MCTS, the playouts after a leaf is reached do not benefit from the knowledge in the tree. Ideally, we would like the playout to be driven by some of that knowledge, at least for some local context (a bitmap of neighboring stones, described in Section III-B).

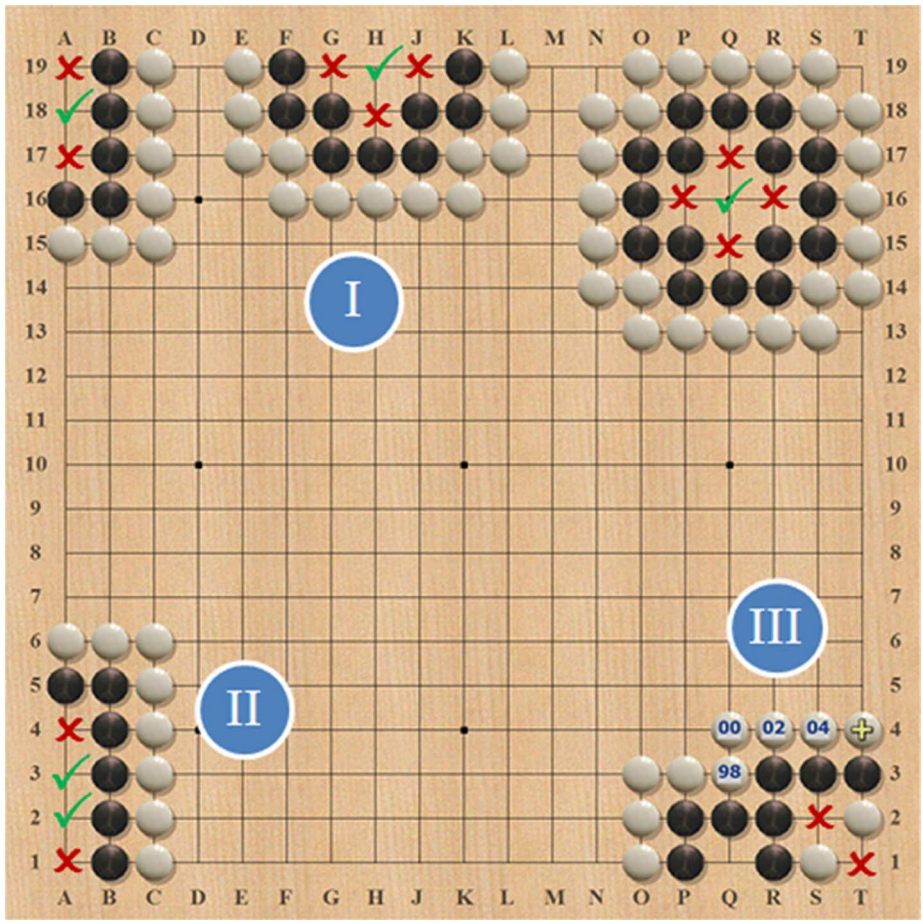


Fig. 2. Some examples of shapes. In (I), the wrong moves outnumber the potentially right moves. The shape improvement part of the policy plays the possibly right moves rather than the wrong moves in (I) and (II). (III) is an example of how $s2$ and $t1$ are both wrong for white even if they leave a dead nakade shape because they destroy the seki of the white stones $s1$ and $t2$.

Base policy (move generation)
{
1. If a legal **atari/capture** move exists, play it with probability $p_{AT}[t]$ where t is the number of consecutive **atari/capture** moves played.
2. If a legal **urgency** pattern move exists, play it with probability p_{MP}.
3. Play moves previously skipped by randomization in steps 1 and 2 before new random moves.
4. If a legal random move m exists, consider improving it to m' in case m is a shape destruction move. If improvement is possible and legal, play m' else play m.
5. Play a pass move

}

Fig. 3. Base playout policy.

The following ideas emerge naturally:

- keeping statistics of wins/losses for each move classified by a local context;
- keeping statistics of wins/losses for all answers to each possible previous move classified by the local contexts in both the previous move and the next move.

MCTS is based on a sound logic where each node represents a completely defined game state. On the other hand, the "loose tree" idea only aims at stochastically getting things "more times right than wrong." Learning playouts will not always find moves that drive the playout like the tree does in MCTS, but when it does, it is a way of propagating online learned knowledge, just like MCTS, but beyond the tree search resulting in more moves "driven by knowledge."

In the worst case, a local context may be the combination of states where the move is both good and bad. For example, a semeai with the same local pattern may be decided by the number of liberties at some distant location, making the pattern useless. But, in enough cases, the pattern will decide the difference between good and bad.

Following this "loose tree" analogy, we name the moves generated by the first idea "node" moves, because a move associated with a local context (for which we keep a WLS) acts like a node in the tree. The moves generated following the second idea are similarly named "edge" moves. Such a move acts like an edge of the tree: an answer to a previous move.

B. Local Context of a Move

A local context is a bitmap around each intersection with four possible values (empty, own color, opponent color, off board)

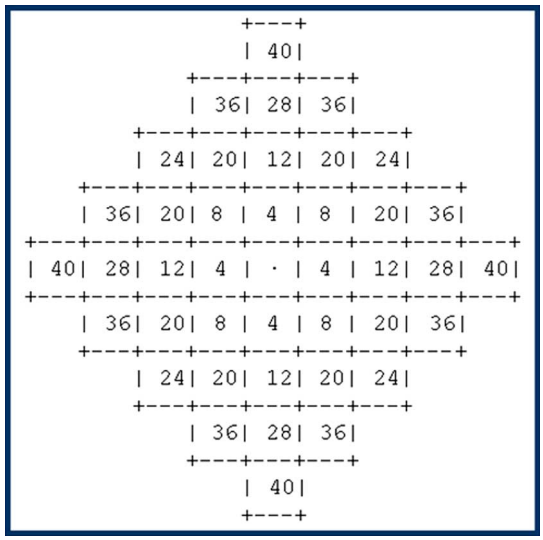


Fig. 4. All different local context sizes supported by the HBS board system. Each board size s contains all intersections numbered with integers $\leq s$ in the picture.

for each neighboring point. The HBS board system [8] keeps a Zobrist hash of the pattern for each empty intersection (legal or not). The size of the pattern is configurable at compilation time. The options available are 4, 8, 12, 20, 24, 28, 36, and 40, shown in Fig. 4. Zobrist hashes computed by the board system are 32-b unsigned hashes. Due to the large number of intersections considered per second, hash collisions are not "almost impossible" as in other applications, but only limited to some controllable probability of happening. In fact, for practical reasons, the implementation only considers a number of bits (12–20) from the whole hash value. The whole evaluation by simulation idea is stochastic and accepting the probability of a hash collision around 1/4096–1/1 048 576 is not unreasonable.

The pattern size shown in Figs. 5 and 6 is 36. The pattern size used in the experiments is 40, which corresponds to the worst case in terms of speed for the learning policy. From size 24 upwards, the improvement is small, and it would require a long series of experiments to find out the optimal size in terms of strength versus speed balance.

C. Node Moves: Move Generation

A WLS is an integer representing evidence that a proportion is below or above 1/2, where higher values represent more evidence of being above. A detailed description is recommended [4], [5].

Fig. 5 shows the basic idea of how node moves are played. The empty intersections around the last move are checked (for legality) in the local context as the board stands before any other move is played. The move with the highest WLS is played if the value is above some threshold.

The knowledge stored for the node moves is an array of WLS states *node_WLS*[ ] with three indices (i_c, i_m, i_l): black or white, next move, context (an integer in $\{0, \ldots, 2^{n-1}\}$) which is, as mentioned, the lower n bits of the local context of the (candidate) next move.

The threshold is $t_n \pm b_n$, the base threshold for *node moves* t_n plus/minus a balancing offset described in Section III-G. The move played (if any) is the legal move satisfying *node_WLS*$[i_c, i_m, i_l] \geq t_n \pm b_n$ with the highest value of *node_WLS*$[i_c, i_m, i_l]$.

The number of neighbors (8 or 20) is configurable and was adjusted in the final experiments considering both improvement and speed. It was finally set to 20.

D. Node Moves: Updating the Information

At the end of each playout, the board is evaluated and the winner identified. The first n_{upd} moves played in the playout (or the entire playout, whichever is smaller) are updated by adding a win to all the moves of the winning player, in the context that was on the board before the move was played, and a loss to all the moves of the opponent. The parameter n_{upd} is justified by the assumption that the first moves in the playout have more influence on the outcome of the game than the final moves. It depends on the length of the playout (which depends on the board size and the current move number). Larger values should produce more updated information; smaller values should produce a higher quality of information. We did not intensively test this parameter. It was set to 80 by CLOP optimization [12] from a categorical variable with four possible values: 48, 64, 80, and 96.

The entire array *node_WLS*[ ] has to be cleared, at least when a new board is defined. We decided to keep the values from one move to the next without actually testing which of the two options is best: clearing it at each new move to keep the most up to date information or keeping the previous states so as to have more information.

E. Edge Move: Move Generation

Fig. 6 shows an example of edge move generation. The list of "best k" moves in the local context of the previous move and the color of the previous player are used to see what the possible answers are and in what context they are expected. If the moves are legal on the current board and their local context matches the expected local context, the move with the highest WLS is played if its value is above some threshold.

The knowledge stored for edge moves is an array of WLS states *edge_WLS*[ ] with four indices $(i_c, i_{pm}, i_{nm}, i_l)$: black or white (the previous move will be of the opposite color), previous move, next move, and context (an integer in $\{0, \ldots, 2^{m-1}\}$). The latter represents the lower m bits of the local context of the next move XOR, which is the local context of the previous move.

It would be very slow to search all possible legal moves as answers to the previous move. Since playout policies have to be fast, keeping a list of "best k" answers to each move in each local context (of the previous move) is a solution to this problem. In this case, only the "best k" answers kept of the previous (move, context) are going to be considered. These answers may even be illegal and may probably not fit the local context currently on the board for the next move. It does happen that the "best k" answers do not provide a move even when "somewhere in the *edge_WLS*[ ]" a move could be found, but that compromise is accepted for the higher speed of just considering the "best k." The optimal value was obtained, as described in Section IV-A.

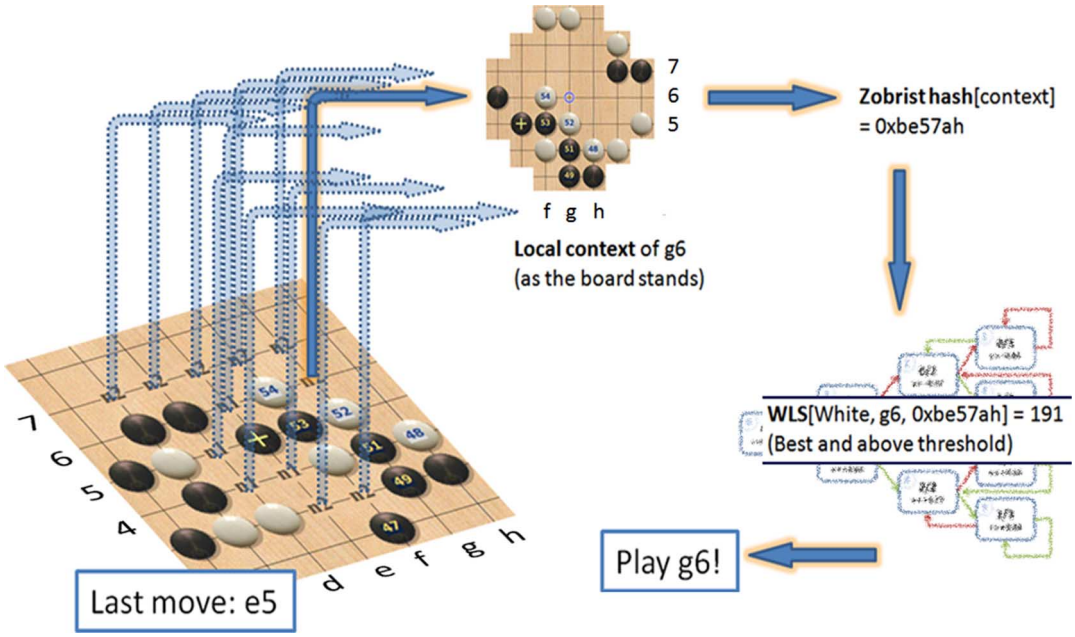


Fig. 5. "Node move" example.

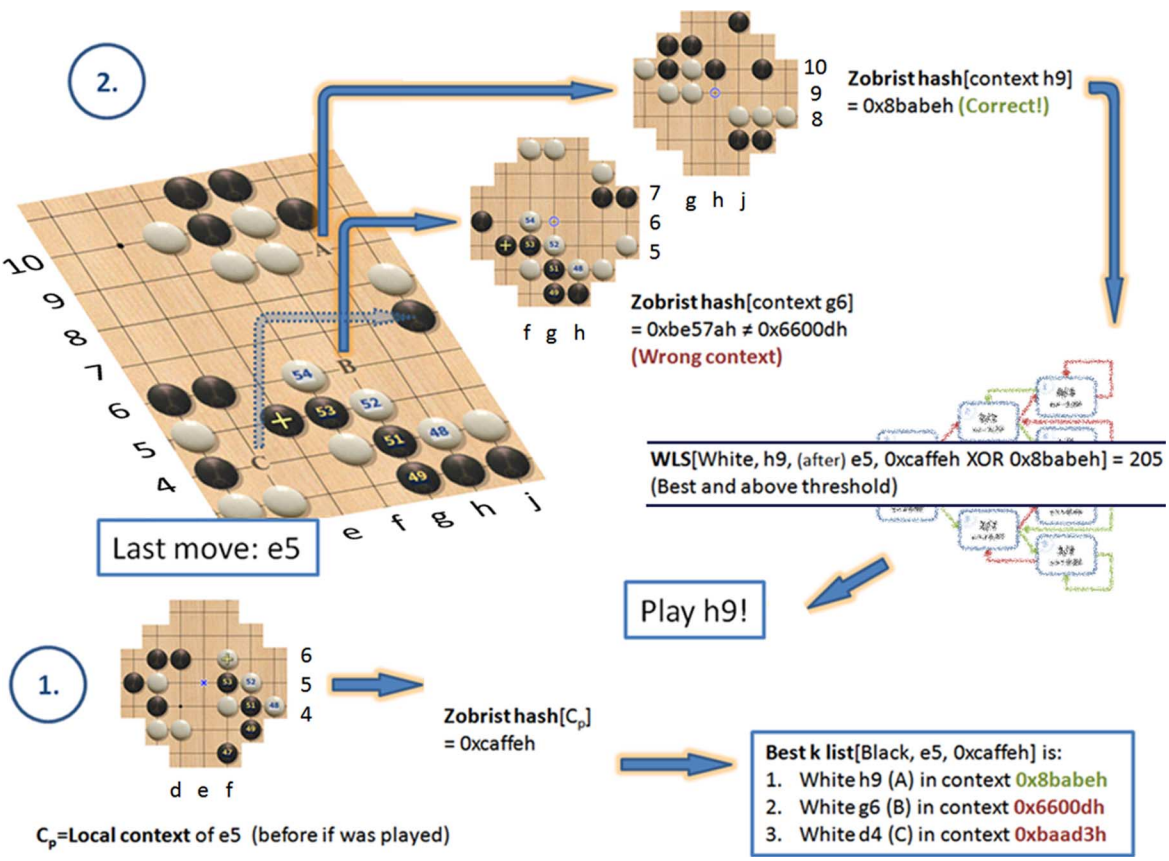


Fig. 6. "Edge move" example.

The "best k" lists are stored in an array of records *edge_bestk*[ ] with three indices (i_c, i_{pm}, i_{lpm}): black or white (color of previous move), previous move, local context of previous move (an integer in $\{0, \ldots, 2^{m-1}\}$), which is the lower m bits of the local context on the previous board before the previous move was played.

"Best k" records store a list of size k of best moves, each containing: the answer move, the next hash, and a counter tracking the illegality of each move to purge the list. While generating moves, the legality of the moves in the "best k" table is checked: if the move is legal, the counter is cleared; if not, it is increased.

F. Edge Moves: Updating the Information

The procedure to update edge move knowledge is similar to that which updates node knowledge, but it is done in move pairs from which neither is a pass. The winner is considered as the answering color. For example, if the playout was won by B, the pair WB will be updated as wins and the pairs BW will be updated as losses. The same n_{upd} parameter used for updating node moves applies to this case.

For each BW and WB pair of moves not including pass moves, the "best k" list is purged. If any element has an illegality counter above the parameter t_{ill}, the entry is removed. This is intended to allocate room on the list that may be used by old moves that are no longer legal (because a solid group might have been formed at that spot). After the purge, the lowest WLS in the list is noted to determine the level at which new elements can reach the "best k" list.

The array *edge_WLS*[ ] is updated as in the case of nodes (but with four indices), and the resulting states are checked as to whether they are above the threshold for entering the "best k" list. If above, the new element enters the "best k" list, replacing the previously lowest WLS found. A new lowest element and threshold is identified.

G. Balancing B/W Move Numbers

It was hypothesized that when applied to board positions where one player is clearly ahead, generating moves using a fixed WLS threshold level would produce more moves for one player than for the other, introducing bias in the evaluation. Since the learning policy generates moves that are, on average, better than random moves, this would introduce a bias because the system would be "helping the winner." Also, when the winner is getting 75% wins in the simulation, it may consider moves winning over 50% of the time as far from optimal. For the losing player who is winning 25% of the time, moves winning over 50% will be very rare. This player would consider moves winning 30% of the time interesting. The impact of balancing has been established already [13], [14] and confirmed by experiments (see Table III).

The condition for playing a node move with index i as black requires $WLS[i] \geq t_n + b_n$; as white with index j, it requires $\mathrm{WLS}[j] \geq t_n - b_n$, where t_n is the threshold for all node moves, and b_n is the balancing offset against black, initially zero. The program counts the number of moves each policy generates for black and white. Every 50 simulations, b_n is adjusted in the following way.

When the difference in these numbers is bigger than twice the standard error of the mean (the SD is estimated from the binomial distribution), a small correction in ± 1 steps is considered to adjust b_n toward the appropriate direction (increased when the policy is creating too many black moves and decreased when it is creating too few). If one number of moves exceeds the double of the other, a bigger step is considered (4 or 8) to make the initial correction faster. Results logged out by the program show that this simple procedure balances the move numbers well enough, resulting in no correction being issued most of the time and corrections being done in ± 1 steps. Similar parameters t_e and b_e exist for edge moves adjusted by the same procedure.

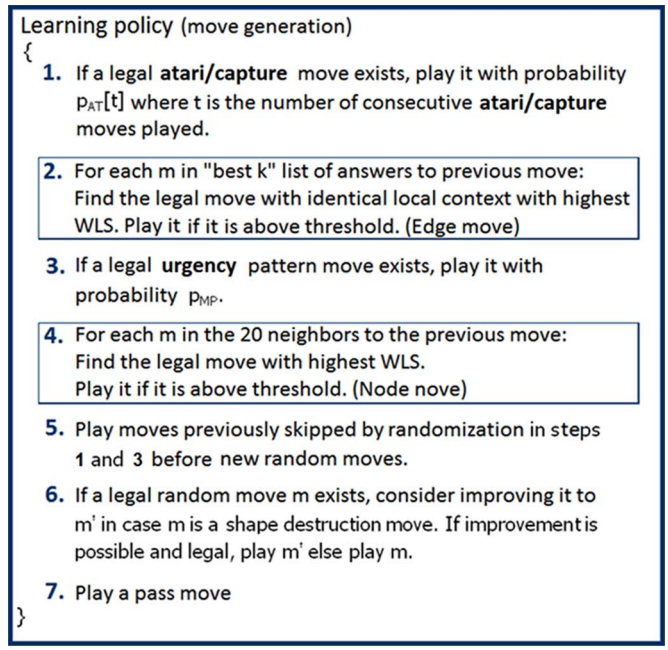


Fig. 7. Learning playout policy in pseudocode.

H. Policy in Pseudocode

Fig. 7 describes the learning policy in pseudocode. It is the same as the base policy but with the insertion of edge moves with a higher priority than the urgent response pattern moves and the node moves with a priority just below it.

IV. Experimental Results

All experiments in this section compare the performance between GoKnot-isGO [15] using the base policy and the learning policy, named isGO(base) and isGO(learning), respectively. In all cases, all other program settings are identical, and isGO(learning) does a fixed number of playouts per move. isGO(base) does either the same number of playouts or is assigned the same execution time used by isGO(learning) depending on the experiment.

Direct Experiments: We conducted six experiments playing isGO(base) versus isGO(learning) for three board sizes: 9×9, 13×13, and 19×19; and two settings: identical number of simulations per move and identical computing time. One thousand games were played in each experiment (600 in 19×19).

Test Against an Independent Third Program: As a confirmation, we included experiments against a fixed reference program Fuego [3]. We used a version from July 19, 2012. Fuego was set on a fixed number of simulations per move, four threads, no opening book, and pondering was disabled. Games were conducted for board size 13×13 and a number of simulations resulting in not too much advantage for Fuego: 4×1000 (four threads, 1000 simulations per thread) for Fuego versus 4×4000 for isGO. We conducted 300 games between Fuego and isGO(learning). Two different experiments were conducted with isGO(base): one with identical numbers of

TABLE I
VALUES OF PARAMETERS USED IN ALL EXPERIMENTS

	isGO(base)	isGO(learning)	Tuning method
$p_{AT}[t]$	pAT[1]=0.3405 pAT[8]=1 (linearly)	pAT[1]=0.3405 pAT[8]=1 (linearly)	(1)
k of "best k" lists	n.a.	8	(2)
Threshold for edges	n.a.	125/200 (closest integer to this proportion)	(4)
p_{MP}	0.4812	0.4812	(1)
Neighbors for nodes	n.a.	20	(3)
Threshold for nodes	n.a.	1/2	(4)
Neighbors for *local contexts*	n.a.	40	(3)
Balancing B/W	n.a.	ON	(5)
WLS end of scale	n.a.	21	(6)

Note: Tuning methods: (1) CLOP optimization of the *base* policy, same values used for both versions. (2) Individually tested, increasing further is slower and does not show measurable improvement. (3) Individually tested, the maximum showed improvement. (4) Only tuned individually in large steps due to the number of games required to assess the winner because of low sensitivity. (5) Tuned ON vs. OFF with clear superiority. (6) Not tested. This is the maximum number for an 8 bit WLS and influence was assessed in the WLS study. n.a. = not applicable.

simulations as isGO(learning), and the other with an identical running time as isGO(learning).

Test for Strong Engine Settings: To further confirm the results in a strong environment, we did two extra experiments with 4×8000 and $4 \times 32\,000$ simulations in board size 9×9. Again, these experiments were repeated for equal computing time.

Additional Testing: We also tested the individual impact of edge move and node move parts independently and the influence of the balancing heuristic.

For all results, we include winning percentage, Elo increase, and 95% confidence intervals for the Elo increase for both equal numbers of simulations and equal time.

A. Parameter Values Used in Experiments

Table I shows the values of the parameters described above used in the experiments. Note that common parameters were tuned by testing with the base policy and should be near optimal. The learning policy could further improve from different settings for these parameters.

B. Number of Moves Generated at Each Step

Table II shows the percentage of moves generated by the different steps in the policies taken from 60 extra games played for the different board sizes: 9×9, 13×13, and 19×19, 20 games each. The base policy plays moves found in the previous steps and discarded by the randomization rules, before playing a random move (19.04% of moves under "previously skipped").

TABLE II
PERCENTAGE OF MOVES PLAYED BY EACH PART OF THE POLICY

	isGO (base)	isGO (learning)	Learning / base
Total simulations	$30.2x10^6$	$30.2x10^6$	
Total moves	$7.62x10^9$	$8.1x10^9$	
Atari/capture (%)	7.60 %	6.72 %	0.88
Edge moves (%)	n.a.	3.98 %	n.a.
Urgent response (%)	17.66 %	17.43 %	0.99
Node moves (%)	n.a.	33.41 %	n.a.
Prev. skipped (%)	19.04 %	4.72 %	0.25
Total before random	*44.29 %*	*66.26 %*	*1.50*
Blind random (%)	47.96 %	28.69 %	0.60
Improved shape (%)	4.40 %	1.53 %	0.35
Pass (no more) (%)	3.35 %	3.53 %	1.05

Note: The complete dataset is based on 20 games in each category (self play 9x9 (4×2000), 13x13(4×2000), 19x19(4×2000)) merged in a single table. The proportions depend on configurable parameters: $p_{AT}[t]$, k of the "best k" lists, threshold for edges, p_{MP}, the number of neighbors for nodes, the threshold for nodes and the number of neighbors for *local contexts*. The values used are those described in table I. The values are the total numbers and the mean of the percentages for each board size. n.a. = not applicable.

TABLE III
INDIVIDUAL IMPACT OF THE POLICIES

	Winner Num (%)	Elo increase (95% CI)
Edge Moves (4x2000) vs. Node Moves (4x2000)	**Node moves** 315 of 500 (63%)	**92.5** (60.8—124.1)
Learning Policy: Balancing (4x2000) vs. No balancing (4x2000)	**Balancing** 300 of 500 (60%)	**70.4** (39.3—101.6)

Note: All tests were made for board size 9x9 with 4x2000 simulations each side and 500 games in all. Computing time differences measured were not statistically significant.

Even with that rule, it still plays only 44.29% of the moves from the "heuristics" with 47.96% of nonimproved random moves, plus 4.4% of improved random moves. On the other hand, the learning policy plays 50% more moves from heuristics than the base policy (66.26%), and it is also worth noting that the number of moves played by recovering skipped moves and improving random moves decreases significantly (19.04%–4.72% in the first case and 4.4%–1.53% in the second case).

C. Individual Impact of Parts

Table III shows the results of two extra experiments of 500 games each played in 9×9. Both sides are played by the learning policy with different options disabled. In the first experiment, edge moves are disabled for one player and node moves for the other. The side playing the node moves (edge disabled) is the winner.

TABLE IV
RESULTS OF ISGO(BASE) EXPERIMENT VERSUS ISGO(LEARNING) EXPERIMENT

	Won by learning equal simulations (equal CPU) of total %	Elo increase for equal number of simulations (95% CI)	Elo increase for equal CPU allocation (95% CI)
9x9 (4x2000)	663 (669) of 1000 66.3 (66.9) %	**117.6** (94.7—140.4)	**122.2** (99.3—145.2)
9x9 (4x8000)	147 (135) of 200 73.5 (67.5) %	**177.2** (122.3—232.3)	**127.0** (75.2—178.8)
9x9 (4x32000)	148 (138) of 200 74.0 (69.0) %	**181.7** (126.5—237.2)	**139.0** (86.6—191.5)
13x13 (4x2000)	685 (631) of 1000 68.5 (63.1) %	**135.0** (111.7—158.2)	**93.2** (70.9—115.6)
19x19 (4x2000)	382 (374) of 600 63.7 (62.3) %	**97.4** (68.5—126.4)	**87.5** (58.8—116.3)

Note: The number of simulations is given for each CPU thread times the number of threads. E.g., 4x2000 = 4 CPU threads running 2000 simulations each. In case of equal CPU allocation, this is the number of simulations played by the learning policy and the base policy does more. The confidence interval for the Elo increase is the Agresti-Coull confidence interval with 95% significance converted into Elo increase.

In the second experiment, one side is the complete isGO(learning) used in the other experiments and the other has black and white move balancing disabled. The full version is the winner.

D. isGO(Base) Experiment Versus isGO(Learning) Experiment

Table IV shows the improvement obtained by the learning policy for all board sizes and all CPU allocation settings. All experiments were repeated for equal CPU time allocation.

E. Experiments Against a Reference Opponent

The experiments played against FUEGO confirmed the size of the improvement in a series of 13×13 games shown in Table V. The number of simulations is 4×4000 instead of the 4×2000 in the previous case to further approach the strength of FUEGO. Again, two experiments were conducted: the first in which isGO(base) was given the same number of simulations as isGO(learning), and the second in which isGO(base) was given the same time as isGO(learning). In both experiments, isGO(base) achieved the same number of wins: 68 of 300. The comparison of isGO(learning) versus isGO(base) was computed as a three-player tournament, with isGO(learning) achieving 114 wins and isGO(base) achieving 68 (both times) against the same opponent without playing each other. The Elo increase is defined by the proportion $114/(114 + 68)$ with its confidence interval for a binomial proportion given in Elo points.

TABLE V
EXPERIMENTAL RESULTS IN PROPORTION OF WINS AGAINST FUEGO

	Won by isGO	% of wins (95% CI)	Elo increase (95% CI)
isGO(base) (4x4000)	68 of 300	**22.7** (18.2 – 27.8)	**-213.2** (-260.5 – -166.0)
isGO(base) (4x5480)	68 of 300	**22.7** (18.2 – 27.8)	**-213.2** (-260.5 – -166.0)
isGO(learning) (4x4000)	114 of 300	**38.0** (32.7 – 43.6)	**-85.0** (-125.7 – -44.4)
Comparison (learning vs. base)	114 of 182	**62.6** (55.3 – 69.4)	**90.0** (37.2 – 142.4)

Note: All results are for board size 13x13. The number of simulations is given for each CPU thread times the number of threads. 4 CPU threads running 4000 simulations each. For equal CPU allocation, the base policy was given more simulations (4x5480) resulting in identical time usage as the learning policy. Comparison isGO(learning) vs. isGO(base) is computed as a three player tournament of 900 games with 114 wins for isGO(learning), 68 wins for isGO(base) (same number of playouts), 68 wins for isGO(base) (same time) and the remaining wins for Fuego. The confidence interval for the proportion and the Elo increase is the Agresti-Coull confidence interval with 95% significance.

V. CONCLUSION AND FUTURE WORK

Besides the highly positive results of the experiments (7960 games in all), resulting in an around 100 Elo point improvement even with equal time settings, and improving with the number of simulations, other indicators show that the whole idea is working as expected.

The learning policy is finding the good moves automatically, as shown by the reduction in the number of moves produced by the "avoid shape destruction" step and the lower "recycling" of skipped moves. Playing less (35% of the number played by the base policy) "shape fixing" moves is, in part, a consequence of playing less random moves (60% of the number played by the base policy), but the reduction is much stronger in the former case than in the latter, the explanation being that the learning policy is learning the "good moves" and playing them before considering playing a random move.

The balancing of black and white moves described in Section III-G is contributing strongly to the strength of the learning playouts version. This is an indicator that both edge and node moves are better than random moves, and producing uneven numbers of moves for each player when the game becomes uneven introduces bias as the leading player gets more help than the trailing player.

Improvement increases as the program becomes stronger. This is consistent with the online learned information becoming more accurate by larger sampling sizes. Since the learning policy is automatically learning correct answers to tactical fights like "2 liberty semeai" [14], which are not coded in the base policy, a stronger base policy may result in less improvement. Also, the somewhat smaller improvement for board size 19×19 may be due to the number of playouts (4×2000) not feeding back enough information to the WLS arrays. More playouts (and maybe specific 19×19 tuning) should result in more improvement.

It is also worth noting that the policy combines well with any other policy and may be a successful replacement for more complicated tactical *Go* specific heuristics. Also, since efficiently implementing (in automatically written assembly language, as done in our program) local contexts of moves (see Section III-B) to get as little as a 37% slowdown while computing 40 neighbor local contexts of all empty intersections is a huge software development task, we have decided to make our HBS assembly language pattern management functions free software [8].

An idea for future work is implementing another layer of WLS that generates the moves, just as described in Sections III-C and III-E, but it is not updated in the playout. The values in the array of WLS are updated, while the engine is pondering (the opponent is thinking) the knowledge in the tree from the previous search. If this idea were to work, it would be a way of propagating knowledge learned in the tree to the playouts, a long time aspiration in Computer Go, in which no success has been reported yet.

REFERENCES

[1] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A survey of Monte Carlo tree search methods," *IEEE Trans. Comput. Intell. AI Games*, vol. 4, no. 1, pp. 1–43, Mar. 2012.

[2] J. Basaldúa, "Links to Go rule sets, tutorials and learning materials," 2012 [Online]. Available: http://www.dybot.com/research/doku.php?id=go_rules

[3] M. Enzenberger, M. Müller, B. Arneson, and R. Segal, "FUEGO: An open-source framework for board games and Go engine based on Monte Carlo tree search," *IEEE Trans. Comput. Intell. AI Games*, vol. 2, no. 4, pp. 259–270, Dec. 2010.

[4] J. Basaldúa and J. M. Moreno-Vega, "Win/loss states: An efficient model of success rates for simulation-based functions," in *Proc. IEEE Conf. Comput. Intell. Games*, Granada, Spain, 2012, pp. 41–46.

[5] J. Basaldúa, "WLS: Open source implementation and complementary materials," 2012 [Online]. Available: http://www.dybot.com/research/doku.php?id=wls_library

[6] H. Baier and P. Drake, "The power of forgetting: Improving the last-good-reply policy in Monte Carlo Go," *IEEE Trans. Comput. Intell. AI Games*, vol. 2, no. 4, pp. 303–309, Dec. 2010.

[7] P. Baudis, "MCTS with information sharing," M.S. thesis, Dept. Theor. Comput. Sci. Math. Logic, Charles Univ. Prague, Prague, Czech Republic, 2011, Ch. 6, "Liberty Maps".

[8] J. Basaldúa, "The hologram board system (HBS) assembly language functionality," 2012 [Online]. Available: http://www.dybot.com/research/doku.php?id=hbs_library

[9] S. Gelly, Y. Wang, R. Munos, and O. Teytaud, "Modification of UCT with patterns in Monte-Carlo Go," Tech. Rep. INRIA RR-6062, 2006.

[10] B. Brügmann, "Monte Carlo Go," white paper, 1993.

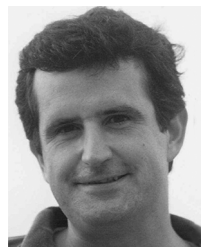
[11] B. Bouzy, "Associating domain-dependent knowledge and Monte Carlo approaches within a Go program," *Inf. Sci.*, vol. 175, no. 4, pp. 247–257, 2005.

[12] R. Coulom, "CLOP: Confident local optimization for noisy black-box parameter tuning," in *Advances in Computer Games*, ser. Lecture Notes in Computer Science. Berlin, Germany: Springer-Verlag, 2012, vol. 7168, pp. 146–157.

[13] D. Silver and G. Tesauro, "Monte-Carlo simulation balancing," in *Proc. 26th Annu. Int. Conf. Mach. Learn.*, 2009, pp. 945–952.

[14] S.-C. Huang, R. Coulom, and S.-S. Lin, "Monte-Carlo simulation balancing in practice," in *Computers and Games*, ser. Lecture Notes in Computer Science. Berlin, Germany: Springer-Verlag, 2011, vol. 6515, pp. 81–92.

[15] J. Basaldúa, "GoKnot," 2002–2012 [Online]. Available: http://www.dybot.com/research/doku.php?id=goknot



Jacques Basaldúa received the Ph.D. degree in Monte Carlo tree search applied to Computer Go and human genetics from the University of La Laguna, La Laguna, Tenerife, Spain, in 2013.

He is a Statistician and Computer Scientist involved in Computer Go development since 2002. He carried out Computer Go research at the Department of Statistics, Operations Research and Computation, University of La Laguna, Currently, he develops big data analytics software.

Sam Stewart is currently working toward the B.A. degree in mathematics and computer science at Lewis & Clark College, Portland, OR, USA.

He worked on the Orego project with Prof. P. D. Drake during summer 2012.

J. Marcos Moreno-Vega received the M.Sc. degree in mathematics and the Ph.D. degree in computer science from the University of La Laguna, La Laguna, Tenerife, Spain, in 1990 and 1997, respectively.

Currently, he is an Associate Professor at the Department of Statistics, Operations Research and Computation, University of La Laguna, in the area of computer science and artificial intelligence. His research interests include soft computing, metaheuristics, and data mining applied to games and logistic problems. In particular, he is interested in the optimization techniques Monte Carlo tree search, tabu search, scatter search, path relinking, and variable neighborhood search. At the moment, he is involved in a research project devoted to developing an intelligent system for the management of the freight transportation in ports.

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

Currently, he is an Associate Professor of Computer Science at Lewis & Clark College, Portland, OR, USA. 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.