All-Moves-As-First Heuristics in Monte-Carlo Go

David P. Helmbold and Aleatha Parker-Wood
Department of Computer Science, University of California, Santa Cruz, California, USA

Abstract— *We present and explore the effectiveness of several variations on the All-Moves-As-First (AMAF) heuristic in Monte-Carlo Go. Our results show that:*

- *Random play-outs provide more information about the goodness of moves made earlier in the play-out.*
- *AMAF updates are not just a way to quickly initialize counts, they are useful after every play-out.*
- *Updates even more aggressive than AMAF can be even more beneficial.*

Keywords: Heuristic search, Computer Go, Monte-Carlo Tree Search, All-Moves-As-First, UCT

1. Introduction

Go is an ancient game, dating back to before the 5th century BC. It is very popular in China, Korea, and Japan with active professional organizations and big money tournaments in all three countries. Although the rules of Go are relatively simple, the combinatorial complexity of Go is very high, much greater than that of chess, and Computer Go has been identified as a grand challenge problem for AI [CW07].

Go is played by two players (black and white) on a rectangular grid. Each player has a supply of circular pieces, called *stones*, and players alternate placing stones on the intersections. Orthogonally adjacent stones of the same color are in the same *group*, and groups are captured (removed) if all the adjacent intersections are occupied by enemy stones. The goal of the game is to control the majority of the board's intersections either by occupying or surrounding them. Although most human games are played on 19×19 boards, 9×9 and 13×13 boards are also used, with many computer Go efforts concentrating on the smaller 9×9 board size. Even on this smaller board size, the combinatorial complexity and subtle strategy possibilities are such that computer Go programs rarely challenge experienced amateurs. However, some of the newer Monte-Carlo programs have done surprisingly well against human professionals in high-handicap 19×19 games.

In this paper we examine the effectiveness of the *All-Moves-As-First* (AMAF) heuristic in Monte-Carlo Go. In Monte-Carlo Go a large number of simulated play-outs are used to estimate the goodness of moves and positions. Assume that black is to move in some position. If the simulated play-outs starting with a particular move by black tend to results in wins for black, then the simulated play-outs provide some evidence that the move is a good one. The AMAF heuristic uses the play-out as some evidence that the other simulated moves made by black in a winning play-out are also good moves, and increases their estimated value.

In addition to a direct quantitative comparison of several AMAF variants, our results show that:

- Random play-outs provide more evidence about the goodness of moves made earlier in the play-out than moves made later.
- AMAF updates are not just a way to quickly initialize counts, they are useful after every play-out.
- Updates even more aggressive than AMAF can be even more beneficial.
- Combining heuristics can be more powerful than individual heuristics.

The next section introduces Monte-Carlo Go and the UCT algorithm. Section 3 describes the basic AMAF heuristic and the five parameterized variations for which we ran experiments. Section 4 presents our experimental results and concluding remarks are given in Section 5.

2. Monte-Carlo Tree Search

Many of today's strongest Go programs use Monte-Carlo techniques with one or more extensions. In its simplest form, Monte-Carlo Go programs evaluate positions by doing a large number of simulated *play-outs*. Each simulated play-out starts at the current game position and uses random play to generate moves until the play-out's winner can be determined. The goodness of a position for a particular player can be estimated by that player's win rate in the random play-outs. Similarly, those moves that tend to start winning random play-outs can be considered the good moves in the current game position. The accuracies of these estimates depend on the number of random play-outs and how well the two side's bad random plays tend to even out.

A popular and effective extension to this algorithm is Monte-Carlo tree search, which combines the Monte-Carlo random play-outs with a small dynamically built search tree (partial game tree) from the current game position. The UCT algorithm [KS06] for Monte-Carlo tree search uses an upper confidence bound exploration/exploitation technique. This upper confidence bound technique converges to the optimum solution for multi-armed bandit problems [ACBF02]. Related Monte-Carlo tree search techniques represent the current state-of-the-art in computer Go. We use Łucasz Lew's *Library of Effective Go Routines* (libEGO) [Lew09]

In *Proceedings of the 2009 International Conference on Artificial Intelligence*, ICAI'09/ISBN 1-60132-108-2, Editors: Arabnia, de la Fuente, Olivas, pp. 605-610, Los Vegas, USA, 2009.

implementation of the UCT algorithm to manage the search tree.

Each simulated play-out from the current game position now has two parts: a navigation through the current search tree, followed by random moves to finish the game. The root of the UCT tree represents the current game position and each node in the tree represents a board position that can be reached from the current position in a few moves. Associated with each node are counts of how many winning and losing play-outs have gone through the node. These counts determine how the play-outs traverse the tree, how the tree grows, and eventually which single move is selected for play in the game.

Since examining the entire search space from most positions is prohibitively expensive, the partial search tree is kept relatively small. It is dynamically grown by expanding those moves which have proved promising during the previous Monte-Carlo play-outs. In particular, if a sufficiently large number of random play-outs have started from the same leaf in the tree, then that leaf's children are added to the partial search tree and start accumulating counts of their own.

The UCT tree assigns a *value* to each node based on the counts at the node. In particular, the value of a node is the win rate plus a variance term. The variance term is designed so that the current win-rate plus variance gives a high-confidence upper bound on the long-term win-rate of the position represented by the node. A play-out traverses the UCT tree by going to the child with the highest value. Using these upper bounds to traverse the tree provides a solution to the exploration/exploitation dilemma because nodes with high upper bounds have high variance (due to relatively few visits) and/or good win rates .

After reaching a leaf in the UCT tree, the play-out continues with a Monte-Carlo simulation. When the winner of the play-out is determined, the counts in the UCT tree nodes traversed by the play-out are updated. We call this the *standard update* for a play-out. The *standard UCT algorithm* combines UCT management of the Monte-Carlo search tree with standard updates after each simulated play-out.

In the example of Figure 1, the two tree nodes in bold (black C2 and white A1) are the only ones updated by the standard update for the indicated play-out. The various AMAF algorithms make additional updates and/or keep additional counts.

In addition to keeping a search tree, some programs guide the random parts of the play-outs to reduce blunders. Rather than using a uniform distribution over the legal moves, the program uses a distribution created by an agent with some Go knowledge. For example, the very successful MoGo [CCF$^+$] program directs its random play-outs using a hand-crafted hierarchy of priorities and heuristics.

Increasing the “smarts" in random play-outs is usually helpful, and minimal constraints (like not filling in one's own eyes) appear essential for play-out termination. The libEGO routines implement this minimal functionality. Surprisingly, increasing the bias in the random play-outs can occasionally weaken the strength of a program using the UCT algorithm *even when the bias is correlated with Go playing strength*. One instance of this was reported by Gelly and Silver [GS07], and our group observed a drop in strength when the random play-outs were encouraged to form patterns commonly occurring in computer Go games [Fly08].

3. All Moves As First (AMAF)

Brügmann [Brü93] gives the first description of the AMAF heuristic in Monte-Carlo computer Go and Gelly and Silver [GS07] describe the first effort to combine the AMAF heuristic with UCT that we are aware of. This heuristic increases the knowledge extracted from a play-out at the cost of including knowledge that may be biased or less relevant.

In the context of Monte-Carlo tree search, the *AMAF update* (see Figure 1) updates not only the counts at nodes through which the play-out passes, but also many siblings of those nodes. Assume a simulated play-out has a player making move m that goes from some parent node to a child c in the tree. For each other move m' made by the same player later in the play-out, the sibling of c corresponding to move m' could also be updated with the result of the play-out. The AMAF update does this additional sibling updating for each node c traversed by the play-out. Since play-outs tend to end with each player occupying almost half the intersections, many siblings tend to have their counts updated by the AMAF update.

The *basic AMAF algorithm* combines UCT with the AMAF update after each play-out. This algorithm rapidly grows the counts at the nodes in the UCT tree, and thus increases the algorithm's confidence in the win rates. On the other hand, the counts at nodes are increased not because the move was made in the position represented by the parent, but because it was made in a (perhaps very) different context. This use of information from other contexts makes the counts from the AMAF update immediately suspect.

There is an argument that counts from the AMAF updates are relevant when strict Monte-Carlo play-outs are used. If moves are made uniformly at random from the legal plays and there are no captures, then either player's moves in the play-out can be permuted to create another play-out reaching the same final position. Furthermore, both the original and permuted play-outs have roughly the same probability of being generated[1] by the Monte-Carlo process. Therefore it is reasonable to update the counts at nodes traversed by (player-preserving) permutations of the play-out as well as the counts at the nodes traversed by the original simulated play-out. Our experiments show that this extended version,

[1]Because self-capture possibilities may arise and disappear, moves may be sampled from slightly different sets in the original and permuted play-out leading to slightly different probabilities.

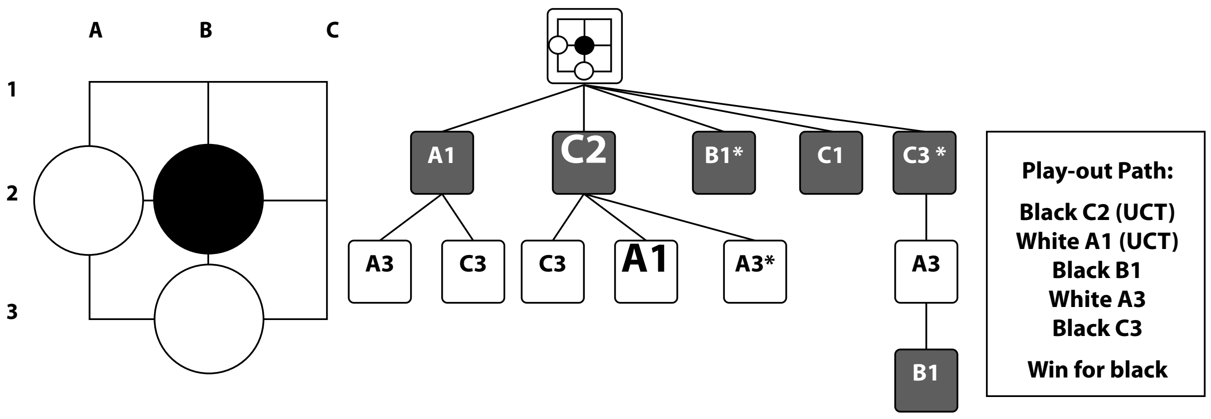

Fig. 1: An artificial example illustrating a play-out, the standard update, and the AMAF update. The current game position on a 3×3 board is given at the left and is represented by the root of the Monte-Carlo search tree. After a number of play-outs the Monte-Carlo tree has grown and part of it is shown in the middle of the figure. Each non-root node is labeled with the move made to enter it from its parent. The next simulated play-out is shown at the right. The tree part of this play-out goes through nodes Black C2 and White A1 (in bold), and the standard update uses the result of the play-out to update the counts at those two nodes (only). In the AMAF update, if the play-out passes through a node, then that node's siblings that are labeled with moves made later in the play-out are also updated. For this play-out, the additional nodes updated by the AMAF update are: Black B1, Black C3, and White A3, and these nodes are indicated by asterisks ($*$) in the figure.

which we call Permutation-AMAF, can be beneficial (see Section 4).

On the other hand, this argument breaks down when specialized Go knowledge is used to direct the random play-outs.

Recall that the *standard update* adjusts the counts only in those UCT nodes traversed by the original simulated play-out while the *AMAF update* adjusts the counts at those UCT nodes as well as all siblings of those nodes that correspond to a move made by the same player later in the play-out (see Figure 1). In the UCT method, each kind of count is combined with a variance term to provide an optimistic estimate on the value or goodness of the node. In this paper we consider the following algorithms which are parameterized variants of the AMAF heuristic.

The **α-AMAF** algorithm blends the value estimates from the standard updates and the AMAF updates. It keeps two sets of counts at each node, one updated with the standard update and the other updated with the AMAF update. α-AMAF estimates the value of a node as α times the estimate from the AMAF updated counts plus $(1-\alpha)$ times the estimate from the standard update counts (recall that each estimate includes a different variance term). Thus $\alpha = 0$ corresponds to the standard UCT algorithm and $\alpha = 1$ corresponds to the basic AMAF algorithm.

The **Some-First** algorithm uses a more restrictive version of the AMAF update that updates fewer nodes. Once the result of a simulated play-out has been determined, the play-out is truncated by deleting moves made after the first m random moves, and then the AMAF update is applied with respect to the moves in the truncated play-out. When m is larger than the play-out length, this has the same effect as the AMAF update. When $m = 0$, the only nodes that get their counts updated are those that are labeled by moves occurring within the UCT tree part of the play-out.

The **Cutoff** algorithm initially uses the AMAF update to "warm up" the counts in the tree and then switches over to the (presumably) more precise standard update to refine the estimates. It uses a single set of counts at the nodes, but the counts are updated with the AMAF update for the first k play-outs (out of 100,000) and updated using the standard update for the remaining play-outs. When $k = 100,000$ this is the basic AMAF algorithm, and when $k = 0$ this becomes the standard UCT algorithm.

RAVE stands for "Rapid Action Value Estimation" and is another way to help warm-up the tree that is used by the successful MoGo program [GS07]. The RAVE algorithm is like α-AMAF except that each node has its own α value that starts at 1 and decreases as simulated play-outs go through the node. There is a parameter that controls this decrease;

the α value used for a node is

$$\frac{\text{parameterVal} - (\text{play-outs through node})}{\text{parameterVal}}$$

or 0 if this expression is negative.

The **Permutation-AMAF** algorithm keeps two sets of counts, one using the standard update and a second using an update even more aggressive than the AMAF update. The AMAF update modifies the counts at nodes that can be reached using a prefix of the simulated play-out followed by a single move made later in the play-out. The *permutation update* modifies the counts at every node that can be reached by permuting the moves in the play-out while preserving stone colors and player alternation. In the example of Figure 1, Permutation-AMAF would not only update the bold and starred nodes updated by the AMAF update, but also the White A3 and Black B1 nodes under the starred Black C3 node since there is a color-preserving permutation of the play-out's moves that starts: Black C3, White A3, Black B1. Permutation-AMAF's updates are more expensive than the AMAF update because it requires a (partial) recursive tree traversal while the AMAF update need only examine the nodes traversed by the simulated play-out and their children. Like α-AMAF, this algorithm has an α-parameter that combines the UCT values from the standard update counts and the aggressively updated ones.

4. Experiments

We implemented the 5 variants of AMAF described in the previous section using Łukasz Lew's libEGO [Lew09][2] implementation of the UCT algorithm. Since the additional cost of performing the AMAF updates and associated bookkeeping is modest[3] almost all variants were set to make 100,000 play-outs per game move. The exception was the more expensive Permutation-AMAF which used 50,000 play-outs to make its time used roughly comparable to that of the other variants. In general, the AMAF variants performed much better than UCT without the AMAF heuristic.

The twogtp program was used to play matches between the AMAF variants and two baseline programs: the standard UCT algorithm and the basic AMAF algorithm. Matches were 1,000 games (500 with each color) and used Chinese scoring with a komi of 5.5 points. The error bars in the plots indicate 95% confidence intervals using the normal approximation.

Figure 2 shows α-AMAF's win rate against standard UCT and the basic AMAF algorithm. α-AMAF does best against basic AMAF when the α parameter is 0.4. α-AMAF does slightly better against standard UCT when α is 0.1,

[2]Version 0.110, with `explore_rate` set to 0.2

[3]In our experiments the AMAF variants usually take from 0% to 35% more time than the standard UCT algorithm (as reported by twogtp) with 20% being typical. Our implementation of the RAVE variant was a little slower, usually taking around 50% more time than standard UCT.

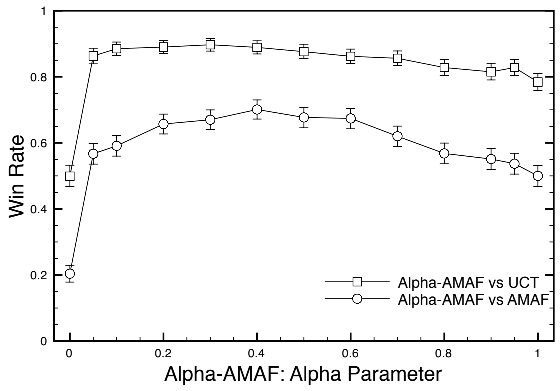

Fig. 2: α-AMAF algorithm versus benchmarks

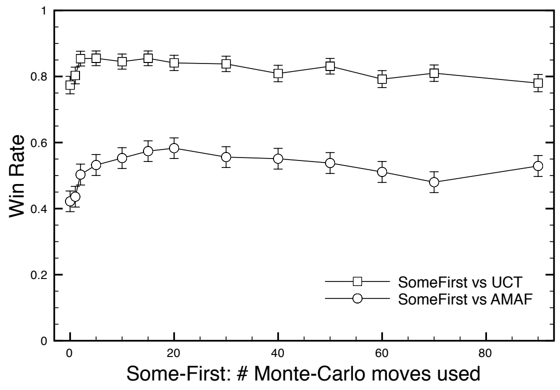

Fig. 3: SomeFirst algorithm versus benchmarks

0.2, and 0.3, but the differences are slight. Note that α-AMAF's performance against both benchmarks decreases as α approaches the extreme values 0 (equivalent to standard UCT) and 1 (equivalent to basic AMAF). This indicates that combining the two benchmark methods is better than either method alone.

Figure 3 shows the results for the Some-First algorithm (which maintains only a single set of counts at each node). The moves=0 parameter setting means only those moves selected within the UCT tree part of the play-out are used to make AMAF updates. Comparing with the α-AMAF results we see that much of the improvement of AMAF updating comes from those moves made within the tree. Using the first few randomly selected moves for the AMAF updates increases playing strength against the benchmarks, but with diminishing returns. Using more than 2-5 moves against UCT or more than 20 moves against AMAF starts diminishing performance. These results indicate that winning (or losing) a Monte-Carlo play-out provides more evidence about the goodness (or badness) of random moves made early in the play-out than those made later in the play-out.

The Cutoff algorithm uses the AMAF update only after

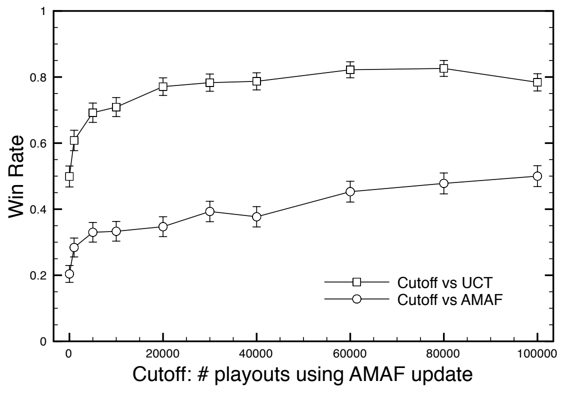

Fig. 4: Cutoff algorithm versus benchmarks

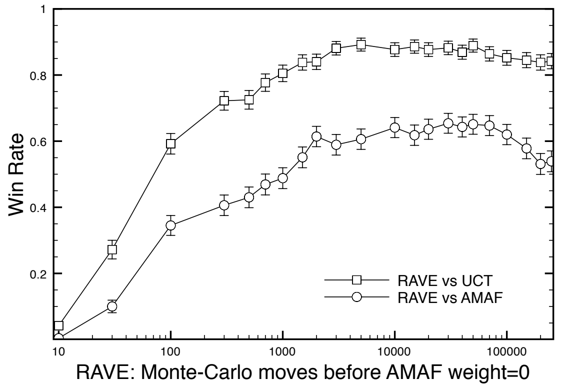

Fig. 5: RAVE algorithm versus benchmarks, unlike the other plots, the RAVE parameter is plotted on a log-scale.

the initial play-outs and then reverts to the standard update. The results for this method are in Figure 4. We expected the AMAF updates to rapidly initialize a UCT with reasonable values that would be refined by the later standard updates. The plot indicates that it is better to use AMAF updates throughout (although performance decreases slightly at the end against standard UCT, it continues to improve against basic AMAF).

The RAVE variation's performance is plotted in Figure 5 (this plot uses log-scale for the X-axis to accommodate the large range of parameter values tested). Our initial tests with smaller parameter values seemed to indicate that its performance characteristics were are similar to the Cutoff algorithm, and converged (from below) to the AMAF algorithm's performance. This was somewhat surprising since some of MoGo's success is attributed to the RAVE updates [GS07]. They use a hierarchy of hand-crafted rules to help direct the Monte-Carlo play-outs (as opposed to our more random play) and suggested RAVE values on the order of 1,000.

However, in further tests we found that even larger RAVE

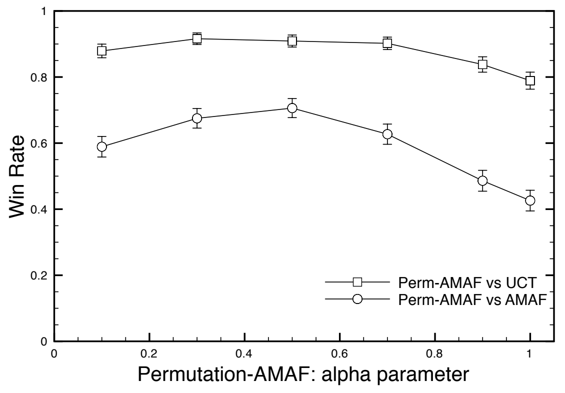

Fig. 6: Permutation-AMAF algorithm versus benchmarks

parameters were much more effective. When using a parameter between 5,000 and 100,000 the win rate against against the AMAF benchmark remains above 60%. The best win rate in our tests was 65.4% at parameter value 30,000. Although not quite as good as the 70% win rate for the $\alpha = 0.4$ AMAF version, this is still a very significant improvement over the ($\alpha = 1$) AMAF benchmark.

Our experiments do $100,000$ play-outs to determine which move to make, and only a few nodes in the tree get more than $10,000$ play-outs passing through them. The RAVE algorithm with a parameter value around $30,000$ essentially relies on values from the AMAF updates at the lower levels of the tree while blending in values from UCT updates at the most promising (and thus most explored) nodes at the top. As the parameter values increased further (to 150,000+) the performance decreases, and as the parameter approaches infinity the RAVE algorithm converges to the AMAF benchmark (assuming the number of play-outs is held constant).

It is peculiar that the RAVE variation with a small parameter value worked so poorly, far worse than the standard UCT baseline. Although we do not yet fully understand this, one possible explanation is that adequate moves can starve better alternatives for the current game situation. If the AMAF value estimate of the adequate move is initially higher than the AMAF value estimate of the better one then the play-outs will tend to pass through the adequate move. When the parameter value is small, this causes the evaluation function to quickly switch over over to the standard UCT one, and the smaller sample size gives it a larger variance term. Meanwhile the better alternative is still getting a large number of low-quality AMAF updates keeping its value (win rate plus variance term) below that of the adequate alternative.

The plot in Figure 6 shows Permutation-AMAF's performance against the baselines. Although Permutation-AMAF uses fewer play-outs to generate each move, each play-out updates many more nodes in the tree. Like α-AMAF, we

Algorithm	α-AMAF			Cutoff		RAVE			SomeFirst		Permutation	
Parameter	0.0	0.4	1.0	1	80k	10	100	1000	0	20	0.5	1.0
Wins vs Gnu Go	254	374	293	245	306	51	277	367	374	332	416	315
Rank vs Gnu Go	10	2(t)	8	11	7	12	9	4	2(t)	5	1	6
Wins vs AMAF	204	701	500	204	478	4	345	488	422	583	706	426
Rank vs AMAF	10(t)	2	4	10(t)	6	12	9	5	8	3	1	7

Fig. 7: Number of wins and rank (best to worst) for selected variants against AMAF and Gnu Go.

use an α parameter to combine the AMAF values with the standard UCT values. With $\alpha = 0.5$ we get some of the best performances against the baseline. On the other hand, when $\alpha = 1$ we get worse performance than basic AMAF. An interesting aspect of these Permutation-AMAF experiments is that they indicate the potential benefit of extracting even more information from a play-out than is done by the AMAF updates.

Using related computer programs to estimate each other's playing strength has the drawback that a version that exploits common weaknesses of the group can appear to be much stronger than it really is. We validated our results by running 1000 game matches between selected versions and Gnu Go (version 3.7 at level 10), a relatively strong program implemented with entirely different techniques. The table in Figure 7 shows that comparing against Gnu Go gives nearly the same strength ordering as our AMAF baseline. The major differences are that α-AMAF with $\alpha = 1$ drops from 4th to 8th, and SomeFirst with parameter 0 moves up from 8th into a tie for second.

5. Conclusions and Further Work

The AMAF heuristic has the potential of multiplying the information gained by a Monte-Carlo play-out but it also makes the potentially dangerous assumption that moves occurring later in a play-out could equally well occur earlier. Our results show that AMAF updates greatly increase the strength of UCT Monte-Carlo methods when the play-outs are random. We observed that although moves early in the play-out provide much of this benefit, using AMAF updates for all play-outs is still better than using AMAF updates simply to "warm-up" the tree.

As mentioned in Section 2, biasing the random play-outs with Go knowledge can lead to counter-intuitive results. Although we anticipate similar results when Mogo-like heuristics such as "save stones in atari" and "rarely play on the edge" are used in the random play-outs, that remains to be verified.

Although AMAF updates are superior to the standard UCT updates, combining the two creates an even stronger player. The benefit of this combining is even greater for the more aggressive Permutation-AMAF update. We are starting to explore other combinations of the heuristics. For example, adding an α-parameter like that in the α-AMAF algorithm to the SomeFirst algorithm looks like a promising way to get additional improvement.

Acknowledgements

We thank Łucasz Lew for making his libEGO system available and for helpful comments on an early draft of the paper; Charlie McDowell, David Doshay, and the UCSC SlugGo group for many helpful discussions; and the Hierarchical Systems Research Foundation for the use of their server.

References

[ACBF02] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Mach. Learn.*, 47(2-3):235–256, 2002.

[Brü93] Bernd Brügmann. Monte carlo go, October 1993.

[CCF$^+$] Guillaume Chaslot, Louis Chatriot, C. Fiter, Sylvain Gelly, Jean-Baptiste Hoock, Julien Perez, Arpad Rimmel, and Olivier Teytaud. Combining expert, offline, transient and online knowledge in monte-carlo exploration. http://www.lri.fr/ teytaud/eg.pdf.

[CW07] Xindi Cai and Donald C. Wunsch II. *Computer Go: A Grand Challenge to AI*, volume 63 of *Studies in Computational Intelligence (SCI)*, pages 443–465. Springer, 2007.

[Fly08] Jennifer Flynn. Independent study quarterly reports. http://users.soe.ucsc.edu/~charlie/projects/SlugGo/, 2008.

[GS07] Sylvain Gelly and David Silver. Combining online and offline knowledge in uct. In *ICML '07: Proceedings of the 24th Internatinoal Conference on Machine Learning*, pages 273–280. ACM, 2007.

[KS06] L. Kocsis and Cs. Szepesvári. Bandit based monte-carlo planning. In *Machine Learning: ECML 2006*, number 4212 in LNCS, pages 282–293. Springer, 2006.

[Lew09] Łukasz Lew. Library of effective go routines (libEGO). http://www.mimuw.edu.pl/~lew/, 2009.