MOVE ORDERING VS HEAVY PLAYOUTS: WHERE SHOULD HEURISTICS BE APPLIED IN MONTE CARLO GO?

Peter Drake
Lewis & Clark College
Department of Mathematical Sciences, MSC 110
0615 SW Palatine Hill Road
Portland, OR 97219
drake@lclark.edu

Steve Uurtamo
SUNY Buffalo State
317 Bishop Hall
1300 Elmwood Ave
Buffalo, NY 14222
uurtamo@gmail.com

ABSTRACT

Writing programs to play the classical Asian game of Go is considered one of the grand challenges of artificial intelligence. Techniques used in a strong Go program would likely be applicable to other games and to non-game domains. Traditional game tree search methods have failed to conquer Go because the search space is so vast and because static evaluation of board positions is extremely difficult. There has been considerable progress recently in using Monte Carlo sampling to evaluate moves. Such programs can be strengthened further by adding domain-specific heuristics. This paper addresses the question of where such heuristics are most effectively applied: to modify the order in which moves are considered or to bias the sampling used to evaluate those moves. Experimental results are presented for three heuristics. We conclude that the sampling "playouts" are the most effective place to apply heuristics.

KEYWORDS

artificial intelligence, Go game, Monte Carlo, UCT, heuristics

INTRODUCTION

Writing programs to play the classical Asian game of Go is widely considered to be a grand challenge of artificial intelligence (Bouzy and Cazenave 2001). While the best Chess programs are on a par with human world champions, the best Go programs still play at the amateur level, at least on the standard 19x19 board.

The traditional approach to computer game play has been minimax search with enhancements such as alpha-beta pruning (Russell and Norvig 2003). Since the number of possible games of Chess or Go is astronomically large (Bouzy and Cazenave 2001), it is not feasible to search the entire game tree. Chess programs therefore rely on domain-specific heuristics for move ordering (considering heuristically "good" moves first and often omitting others) and static evaluation (estimating the value of a board configuration before the game is over). This minimax approach has not been successful for Go. Because the branching factor is much higher in Go, there is a greater risk that a move ordering heuristic might omit good moves. More importantly, because "dead" stones can remain on the board for hundreds of moves before they are actually removed, static evaluation is extremely difficult.

Several computer Go programmers, including the authors, have recently begun exploring statistical sampling approaches called Monte Carlo Go (Brügmann 1993, Coulom 2006, Gelly *et al.* 2006, Drake and Uurtamo 2007). Monte Carlo Go programs respond to a given game situation by randomly completing the game thousands of times, then choosing the move that has led to the best average outcome.

While the Monte Carlo approach has not yet produced 19x19 Go programs at the master level, it has produced startlingly strong 9x9 programs. At the 19x19 level, Monte Carlo programs are now at the level of the strongest traditional programs. At the 12th Computer Olympiad, the top two finishers in the 19x19 competition (MoGo and Crazy Stone) were both Monte Carlo programs; neither lost a single game to any other program.

Because Go is such a rich domain, it is likely that techniques that fare well in Go will be applicable to other domains. Monte Carlo search techniques have been applied to other games (Chung *et al.* 2005) and to non-game domains (Chaslot et al, 2006).

This paper begins with a brief summary of the rules of Go. The Monte Carlo approach to Go is then described. We explain several places where heuristics might be applied within this framework. Three specific heuristics are then presented, followed by our experimental results. The paper closes with conclusions and a discussion of future work.

THE RULES OF GO

While the rules of Go are considerably simpler than those of Chess, space permits only a brief overview here. Interested readers are directed toward any of the many excellent tutorials available in print or on-line, such as (Baker 1986).

Go is a two-player game of perfect information. The board consists of a square grid of intersecting lines; traditionally the grid is 19x19, but smaller boards are sometimes used for teaching new players or for computer Go research. The two players, black and white, each have a supply of stones in their color. Starting with black, the players alternate taking turns. A turn consists of either passing or placing a stone on an unoccupied intersection of two lines. The game ends when both players pass consecutively. Each player scores one point for each intersection that is either occupied or surrounded by his or her stones.

Three additional rules give the game its profound strategic depth. First, a block of contiguous stones is captured (removed from the board) if it is tightly surrounded, i.e., if there are no vacant intersections adjacent to it. Second, a player may not directly cause his or her own stones to be captured. Third, it is illegal to repeat a previous board position.

MONTE CARLO GO

In general, Monte Carlo approaches to game tree search are similar to traditional minimax search, but with statistical sampling used for position evaluation. For example, a program might do a full search to some fixed depth, with the leaves of the search tree evaluated by sampling. To evaluate a leaf position, random moves are played until the end of the game and the score is recorded. The board is then reset to the leaf position and random moves are again played to the end of the game. After many of these playouts, the portion of games won by each side gives a reasonable approximation to the value of the position.

A further refinement of Monte Carlo Go is to focus the sampling on more promising moves. This introduces a tradeoff between *exploration* (sampling the result of each potential move enough times to have an accurate estimate of its value) and *exploitation* (spending the most samples on the moves that have been most successful so far). The strongest extant Monte Carlo programs use the UCT algorithm (Kocsis and Szepesvári 2006, Gelly *et al*. 2006). UCT stands for "Upper Confidence bounds applied to Trees". When choosing a move to sample, UCT chooses the move with the greatest sum of value (portion of won games) and uncertainty. It will therefore always choose either a move that has a good winning record or a move that has not been sufficiently explored. As the sampling proceeds, the uncertainty shrinks and UCT spends most of its time improving its estimates of the best moves. Heavily-sampled tree nodes are expanded, so the search is deepest along the most promising line of play. The algorithm is therefore able to make a confident statement about which move is really best.

APPLYING HEURISTICS

A Monte Carlo Go program can be made even stronger by adding domain-specific knowledge. A number of papers have been published showing the benefits of various heuristics, some of them quite sophisticated. There is some variety in where the heuristics are applied. The options may be divided into two major categories:

- *Move ordering* uses a heuristic to decide the order in which moves are considered within (or at the fringe of) the search tree.
- *Heavy playouts* use a heuristic to bias the "random" moves in the sampling playouts, so that the results will more closely approximate a realistic game.

It is not obvious *a priori* which of these approaches is superior. Move ordering has its effect closer to the root of the tree, which is more relevant to the decision at hand. Furthermore, move ordering only requires computing the heuristics for a very small number of board positions. Heavy playouts require that the heuristics be evaluated for every move of the game. On the other hand, having more accurate playouts may be worth the price of completing fewer playouts in the allotted time. Heavy playouts certainly are better asymptotically, in that if the heuristic were perfect (suggesting only the single best move), only one sample would be needed from each tree node. At the same time, an imperfect heuristic may destroy the benefit of Monte Carlo sampling by exploring an unreasonable subset of the space of possible game completions.

Heuristic Application in Previous Work

Chaslot *et al*. 2007 use move ordering, modifying the UCT formula to take heuristic values into account for sparsely-sampled moves. They also delay exploration of moves with very low heuristic values, which they refer to as "progressive unpruning".

Coulom 2007 uses a very similar technique under the name of "progressive widening". Coulom also uses heavy playouts, producing a probability distribution over legal moves.

Drake and Uurtamo 2007 use heavy playouts. For each move in a playout, a heuristic is chosen with some probability; otherwise a random move is chosen.

Gelly *et al*. 2006 use a different version of heavy playouts, choosing a heuristicially-suggested move if there is one and playing randomly otherwise. The heuristics therefore allow "urgent" moves to be pounced on without slowing down the playouts too much. They also use move ordering and another progressive widening scheme.

Gelly and Silver 2007 use heavy playouts with a probability distribution, but with noise introduced in a variety of ways. They also use move ordering by initializing the value of a new leaf according to heuristics, effectively pretending that the leaf had already been sampled.

Cazenave 2007 uses heavy playouts with a probability distribution.

Our Heuristic Application Techniques

In the experiments in this paper, we apply heuristics in three different ways: move ordering, best-of-*n*, and first-*n*.

Our *move ordering* technique treats untried moves at the fringe of the tree as having a UCT value of $1.0 + ch$, where c is a small constant and h is the heuristic value of the move. The untried moves are therefore tried in the order suggested by the heuristic. An untried move is preferred to a move that has been tried once unless the tried move has a very high UCT value, e.g., it has led to a win every time it has been tried. This is consistent with setting the first-play urgency to 1.0 as described in Gelly *et al*. 2006.

Our *best-of-n* technique is a heavy playout variation that attempts both to remedy the high cost of computing heuristics during playouts and to avoid over-biasing the sampling. At each move during a playout a fixed number (*n*) of random legal moves are chosen. The move with the highest heuristic rating is chosen.

Our *first-n* technique is another heavy playout variation where, for the first *n* moves beyond the fringe of the tree, the move rated highest by the heuristic is chosen. After this point the playout is completed randomly.

Three Heuristics

Our experiments involve three different heuristics: the capture heuristic, the fighting heuristic, and the pattern heuristic.

The *capture* heuristic simply attempts to capture enemy stones whenever possible. Moves that capture more stones are rated higher.

The *fighting* heuristic, at an abstract level, plays moves in areas of the board where fights are taking place. Recall that a block of stones is removed if there are no vacant points adjacent to it. These vacant points are called liberties. We define a pseudo-liberty to be a graph edge leading from a stone to an adjacent liberty. The number of pseudo-liberties a block has is at least equal to the number of liberties it has, but may be slightly higher if there are multiple edges leading to the same liberty. Since counting pseudo-liberties is easier than counting liberties, our board-handling code uses pseudo-liberties to determine when a block has been captured. Pseudo-liberty counts are therefore available for use in a heuristic without additional computation. A group with a small number of pseudo-liberties is in some danger of being captured, i.e., is in a fight. The fighting heuristic prefers moves adjacent to blocks (friendly or enemy) that have few pseudo-liberties.

The *pattern* heuristic involves a table of patterns. Each pattern represents a 3x3 region of the board, with each point marked as either black, white, vacant, or off-board (which happens if the center point is at the edge or corner of the board). Associated with each pattern is a weight indicating the value of playing in the center.

Our patterns were automatically extracted from a database of 3199 9x9 games provided by Nici Schraudolph. Since the players of these games were much stronger than random players, we wanted our patterns to suggest playing similar moves. Each game was run through in each of the eight possible rotations and reflections. For each move in each game in the database, the pattern around the move was given a “win”. Each other pattern that appeared elsewhere on the board was given a “loss”. Any pattern with less than 100 wins was discarded, leaving 1639 patterns. The value of each pattern was set to the ratio of wins to (wins + losses) for that pattern.

RESULTS

Experiment 1: Direct Comparison of Heuristics

As a quick check of the quality of our heuristics, we ran a tournament between different heuristics. Every move in these games was decided by direct application of the heuristics, with ties between heuristic values decided randomly. Orego version 5.05 was used.

There were four contestants: random (no heuristic), capture, fighting, and patterns. Each contestant played 1000 9x9 games against each other contestant, 500 as black and 500 as white. Komi was set at 7.5.

Random won 739 games, capture 1224, fighting 1514, and patterns 2523. This suggests that all three heuristics are useful, but patterns is the strongest and capture the weakest. This experiment did not, however, take into account the time required to compute the various heuristics. The next experiment addresses that issue.

Experiment 2: Time Test

In this experiment, we used four instances of Orego, each using as a heuristic one of the four contestants from the previous experiment. The heuristic was applied using best of 4 heavy playouts. This experiment was run on a Mac Pro with two dual-core 3 GHz Intel Xeon processors and 2 GB of RAM.

Orego was given 10 seconds to search for the best move on a 9x9 board. The random heuristic completed 308,112 playouts, the capture heuristic 225,483, the fighting heuristic 269,760, and the pattern heuristic 235,155.

We repeated this experiment with best of 81 heavy playouts. The capture heuristic completed 77,054 playouts, the fighting heuristic 85,360, and the pattern heuristic 59,956. (The random heuristic would not be affected by this change.)

In both cases the fighting heuristic is the fastest, but the order of the other two is not consistent. No heuristic is twice as fast as another, so we do not expect the speed of the heuristics to be a major factor in the next experiment.

Experiment 3: Heuristic Application Within UCT

This is the central experiment of this paper. We pitted Orego against GNU Go (version 3.6), a widely-available Go program with a conventional (non-Monte-Carlo) architecture. Each condition involves a particular heuristic applied in a particular way. In each condition, 320 9x9 games were played against GNU Go (160 as black, 160 as white). Komi was set at 7.5. These experiments were run on Athlon64 3200+ 800FSB RT CPUs running Linux, with Orego allowed 15 seconds to think for each move.

Figure 1 shows the results for the capture heuristic. This heuristic provides its greatest benefit when applied in best of 2 heavy playouts. Best-of-*n* for large values of *n* is worse than no heuristic at all, presumably because the increase in the quality of the playouts does not make up for the decrease in the number of playouts.

The fighting heuristic (Figure 2) again performs best in the best-of-*n* mode. Intriguingly, it performs better with larger values of *n*. With first-*n*, we again see a decreasing trend. These two trends are somewhat contradictory, since for sufficiently

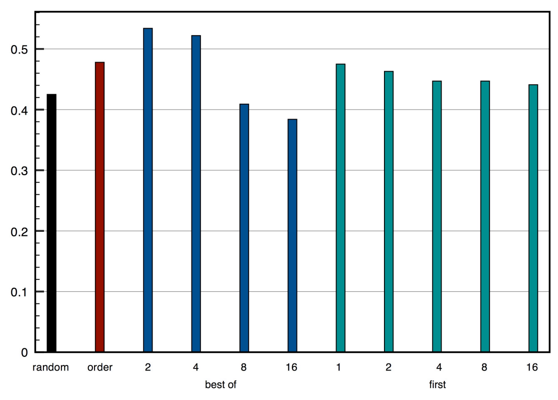

Figure 1: Results for the capture heuristic. The vertical axis shows the portion of games won against GNU Go in each condition. The best-of-2 and best-of-4 performances are significantly better than random (no heuristic) playouts according to a one-tailed T-test, *p* < 0.01 and *p* < 0.02 respectively.

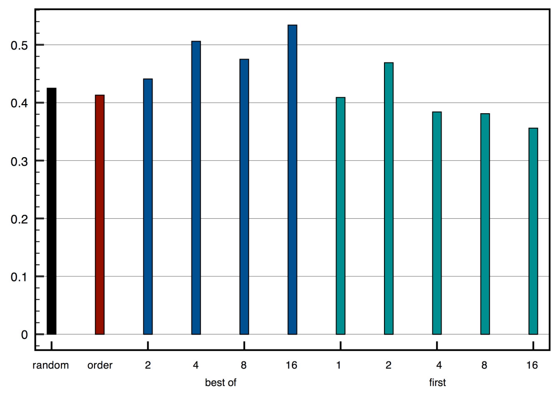

Figure 2: Results for the fighting heuristic. Best-of-4 and best-of-16 are significantly better than random playouts, *p* < 0.05 and *p* < 0.01 respectively.

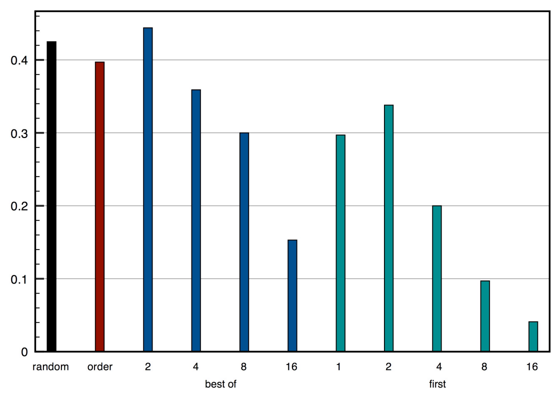

Figure 3: Results for the pattern heuristic. None of the conditions are significantly better than random playouts.

large values of *n* these techniques are identical (on every move in the playout, choose the move with the highest heuristic value).

Given the results of the previous experiments, we were surprised to see that the pattern heuristic weakens Orego in almost every condition (Figure 3). These results do, however, provide further evidence that heavy playouts (specifically the best-of-*n* technique) are the best use of a heuristic.

CONCLUSIONS AND FUTURE WORK

We have reviewed previous research on applying heuristics to UCT Go programs, with an emphasis on where heuristics are applied. Two major categories of heuristic application were delineated: move ordering and heavy playouts. Our experiments, using three different heuristics, showed that heuristics are most effectively applied in heavy playouts. More specifically, a best-of-*n* heavy playout technique seems more effective than a first-*n* technique.

Two of our heuristics (capturing and fighting) proved successful, while the other (patterns) did not. We speculate that the pattern heuristic helps a player *get into* a good position, while the others help a player *take advantage of* a good position. In experiment 1, the capture and fighting heuristics were not at their strongest because they didn't have good positions to take advantage of. In experiment 3, UCT achieved a fairly good position before the playouts began, so capturing and fighting had something to pounce on while the patterns didn't help much. In any case, direct games are not a reliable way of comparing heuristics.

Our results should prove useful in future Monte Carlo programs for Go and for other domains. Knowing where heuristics are best applied should save time in evaluating new heuristics. It may also inform architectural decisions, e.g., encouraging incremental updating of heuristic evaluations so that they may be used throughout the long playouts.

ACKNOWLEDGMENTS

We thank Nici Schraudolph for providing us with recorded game data and the members of the Computer Go Mailing List for their many helpful comments.

REFERENCES

Baker, K. *The Way to Go*. 1986. New York, NY: American Go Association.

Bouzy, B., and Cazenave, T. 2001. Computer Go: an AI Oriented Survey. *Artificial Intelligence* 132(1):39-103.

Brügmann, B. 1993. Monte Carlo Go. Unpublished technical report.

Cazenave, T. 2007. Playing the Right Atari. International Computer Games Association Journal 30(1):35-42.

Chaslot, G., De Jong, S., Saito, J.-T., and Uiterwijk, J. W. H. M. 2006. Monte-Carlo Tree Search in Production Management Problems. In Pierre-Yves Schobbens, Wim Vanhoof, and Gabriel Schwanen, editors, *Proceedings of the 18th BeNeLux Conference on Artificial Intelligence*, Namur, Belgium, pages 91-98.

Chaslot, G.M.J.B., Winands, M.H.M., Uiterwijk ,J.W.H.M., van den Herik, H.J., and Bouzy, B. 2007. Progressive strategies for

Monte-Carlo tree search. Draft, submitted to JCIS workshop 2007.

Chung, M., Buro, M., and Schaeffer, J. 2005. Monte Carlo Planning in RTS Games, *CIG* 2005, Colchester,

Coulom, R. 2006. Efficient selectivity and backup operators in Monte-Carlo tree search. In Proceedings of the 5th International Conference on Computers and Games. Turin, Italy.

Coulom, R. 2007. Computing Elo ratings of move patterns in the game of Go. Draft, submitted to ICGA Computer Games Workshop 2007.

Drake, P., and Uurtamo, S. 2007. Heuristics in Monte Carlo Go. In *Proceedings of the 2007 International Conference on Artificial Intelligence*, CSREA Press.

Gelly, S. and Silver, D. 2007. Combining online and offline knowledge in UCT. In International Conference on Machine Learning, ICML 2007.

Gelly, S., Wang, Y., Munos, R., and Teytaud, O. 2006. Modification of UCT with patterns in Monte-Carlo Go. Technical Report 6062, INRIA, France.

Kocsis, L. and Szepesvári, Cs. 2006. Bandit based Monte-Carlo Planning. In *Proceedings of the 17th European Conference on Machine Learning*. Springer-Verlag.

Russell, S., and Norvig, P. 2003. *Artificial Intelligence: A Modern Approach*, second edition. Upper Saddle River, NJ: Prentice Hall.

BIOGRAPHY

Peter Drake received a BA in English from Willamette University in Salem, Oregon, a MS in Computer Science from Oregon State University, and a PhD in Computer Science and Cognitive Science from Indiana University. He is Assistant Professor of Computer Science at Lewis & Clark College in Portland, Oregon. He is the author of *Data Structures and Algorithms in Java*.