HEURISTICS IN MONTE CARLO GO

Peter Drake
Lewis & Clark College
Department of Mathematical Sciences
0615 SW Palatine Hill Road
MSC 110
Portland, OR 97219-7899

Steve Uurtamo
SUNY Buffalo State
Mathematics Department
317 Bishop Hall
1300 Elmwood Ave
Buffalo, NY 14222

ABSTRACT

Writing programs to play the classical Asian game of Go is considered one of the grand challenges of artificial intelligence. 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 select moves. This paper presents four heuristics used to bias the selection of moves during Monte Carlo sampling: the proximity heuristic (play near the last move), the avoid-the-first-two-lines heuristic (don't play at the edge of the board), the first-order history heuristic (play the move that has fared best elsewhere in the tree), and the second-order history heuristic (play the move that has fared best elsewhere in the tree in response to a particular move from the opponent). Experimental results show that the use of these heuristics significantly improves the ability of our program to defeat GNU Go, a widely-used Go program based on traditional, knowledge-intensive techniques.

KEYWORDS

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

1. Introduction

Writing programs to play the classical Asian game of Go is widely considered to be a grand challenge of artificial intelligence [1]. While the best Chess programs are on a par with human world champions, the best Go programs still play at the level of a moderately strong amateur.

The traditional approach to computer game play has been minimax search with enhancements such as alpha-beta pruning [2]. Since the number of possible games of Chess or Go is astronomically large [1], it is not feasible to search the entire game tree. Chess programs therefore rely on heuristics for move selection (only considering a few of the possible moves) and static evaluation (estimating the value of a board configuration before the game is over). This approach has not been successful for Go. Because the branching factor is much higher in Go, there is a greater risk that a move selection heuristic might discard 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 [3, 4, 5, 6]. 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 it has not yet produced strong 19x19 Go programs, this approach has been surprisingly successful at 9x9 Go. Monte Carlo programs have won all of the recent 9x9 KGS computer Go tournaments.

This paper begins with a brief summary of the rules of Go. The Monte Carlo approach to Go is then described. We explain our use of heuristics within Monte Carlo sampling and the specific heuristics we used. Our experimental results are then presented. The paper closes with conclusions and a discussion of future work.

2. 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 [7].

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.

3. Monte Carlo Go

The traditional approach to game playing, as presented in any introductory artificial intelligence course, is minimax search. This is a full depth-first search of the game space (Figure 1).

For any but the smallest games (e.g, Tic-Tac-Toe), the game tree is far too large to search exhaustively. The standard approach to dealing with this problem is to search to some depth and then apply a static evaluation function to estimate the value of the leaf nodes (Figure 2).

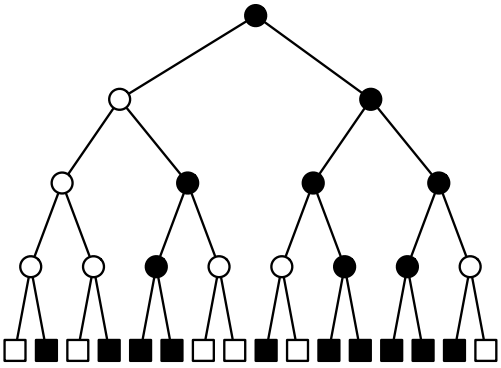

Figure 1: Full minimax game tree search. The value of each leaf node is the winner of that game. The value of each internal node is either the minimum or maximum (depending on the ply) of its children.

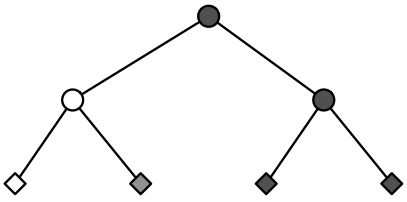

Figure 2: Minimax with static evaluation. The leaves are not end game states, so their value is determined by a static evaluation function.

This technique (with various enhancements) has fared well for Chess, where the branching factor is small enough to allow a fairly deep search and it is possible to construct a strong static evaluation function (e.g., counting the pieces on each side). Go is harder in both respects: there are more moves to choose from at each ply, and it is extremely difficult to statically evaluate the board since "dead" stones may remain on the board for hundreds of moves.

Traditional Go programs have used a great deal of domain-specific knowledge to hand-craft an evaluation function. The Monte Carlo approach, on the other hand, uses statistical sampling to evaluate the board state. Specifically, the game is completed by playing *random* moves. Many random completions are performed and the number of wins for each player is recorded. A move that leads to more wins in these randomly-completed sampling games is considered better (Figure 3).

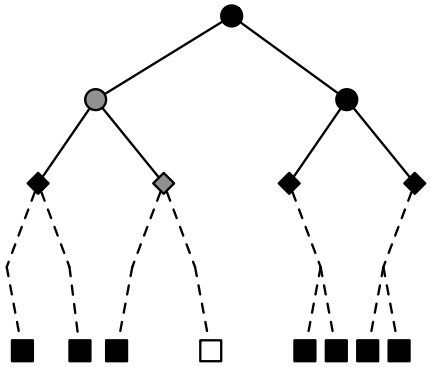

Figure 3: Monte Carlo game tree search. The tree is only fully searched up to some depth. Nodes at this depth (diamonds) are evaluated by sampling the games that might follow from there.

A further refinement of Monte Carlo Go is to focus the sampling on more promising moves. This introduces a tradeoff between *exploration* (sampling 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 (e.g., MoGo) use the UCT algorithm [8, 9]. 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 is thought to be good or a move that has not been sufficiently explored. As the samples proceed, the uncertainty shrinks and UCT spends most of its sampling time improving its estimates of the best moves. The algorithm is therefore able to make a confident statement about which of these moves is really best.

UCT is illustrated schematically in Figure 4. Technical details of the use of UCT in our program Orego are presented in [10].

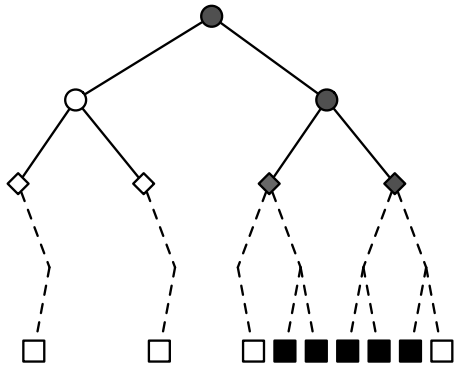

Figure 4: UCT is a variant of Monte Carlo search that focuses search on the most promising moves.

4. Heuristics Within Monte Carlo Go

In vanilla Monte Carlo sampling, the moves beyond the search tree are played purely randomly. (Most Monte Carlo Go programs include a very simple "don't play in your own eye" rule to prevent unreasonably long sampling games.) Orego biases the random moves with various heuristics. Specifically, each time a move is to be generated, it is either chosen randomly or chosen by one of the heuristics. There is a probability associated with each heuristic. These probabilities are treated as different-sized slices of a roulette wheel (Figure 5).

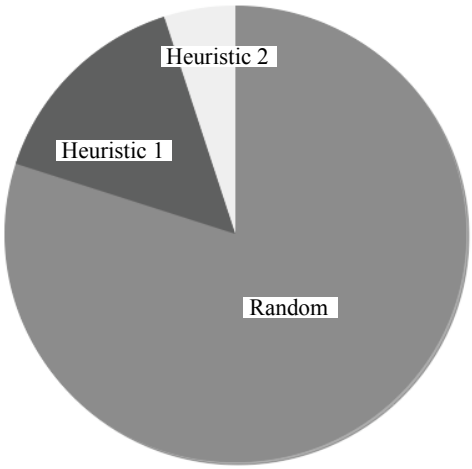

Figure 5: Moves are chosen according to a roulette wheel scheme. At most one heuristic is used to generate a given move.

Completing as many sampling games as possible in the allotted time is crucial to the success of Monte Carlo Go. Since computing heuristics is generally more expensive than randomly generating a move, it is not feasible to apply all of the heuristics to produce a probability distribution for every move. Orego's roulette wheel scheme ensures that at most one heuristic is applied for a given sampling move.

Note that, if a move is good for several reasons, it might be generated by several different heuristics. Since such a move occupies more space on the roulette wheel, it is more likely to be chosen than a move that is suggested by only one heuristic.

We now describe the particular heuristics used in the present research. While our present experiments are run on the 9x9 board, we are hopeful that these heuristics will be even more valuable on the full 19x19 board.

The Proximity Heuristic

The proximity heuristic [10] chooses a move within three points Manhattan distance of the last move played. Since the last move played is likely to have destabilized the local area of the board, a nearby response is often a good move. Put another way, this allows the program to "pay attention to" the current situation. Without some such attentional mechanism, Monte Carlo programs tend to play all over the board (especially in 19x19 Go), ignoring urgent responses. Intriguingly, beginning human players often suffer from the opposite defect of "tunnel vision", focusing intently on one corner of the board even when it would be better to step back and make a move elsewhere.

Our implementation of the proximity heuristic chooses a random point within three points Manhattan distance of the last move played. If this move is legal (and is not an eye-filling move), it is used. Otherwise, a random move is chosen.

The Avoid-The-First-Two-Lines Heuristic

A common bad move played by Monte Carlo programs is a stone played on the very edge of the board (the first line) or just above the edge (the second line). While such moves are correct in some corner situations and at the end of the game, they are usually counterproductive early in the game. Such a low stone is vulnerable to attack but does not stake out much territory.

As with the proximity heuristic, our implementation of the avoid-the-first-two-lines heuristic chooses a random point not on the first two lines and tries to play it. If this is unsuccessful, a random move is chosen instead. This approach has the advantage that near the end of the game, when most of the board is occupied, the heuristic will more often be ignored even if it is chosen by the roulette wheel.

The First-Order History Heuristic

The idea of the history heuristic is to choose a move that has proven to be successful elsewhere in the search tree [11]. If a move appears to be good along several branches of the search tree, it is probably accomplishing some local goal and is likely to be good along several other branches.

Our implementation maintains a table of the frequency of each move. Every time a move is selected by UCT as most promising, the count for that move is increased. When the first-order history heuristic is chosen, the most-frequent playable move is played.

The Second-Order History Heuristic

In Go, there are many moves that are not necessarily good to play overall, but are important responses to particular "forcing" moves by the opponent. For example, if a group of stones could be captured in two moves, connecting to another friendly group is an important response to an attack, but may not be worthwhile until such an attack occurs.

The second-order history heuristic is to choose a move that has proven to be successful *in response to a particular move from the opponent*. Our implementation maintains, for each move by each player, a table of frequencies like that maintained for the first-order history heuristic. When the second-order history heuristic is chosen, the most-frequent playable response to the opponent's last move is played.

5. Experimental Results

We ran experiments comparing various settings for the probabilities of our four heuristics. The effectiveness of each parameter setting was measured by having Orego (version 4.04) play games against GNU Go, a widely-used Go program based on traditional, knowledge-intensive techniques. All experiments were run on 9x9 boards. Orego was allowed 30 seconds per move, using two sampling threads on on 4-CPU Xeon 3.2GHz machines with 2GB of RAM running Linux. These machines are able to complete approximately 4000 sampling games per second under these conditions. In each experiment, half of the games were played with Orego as black, half with Orego as white. Komi was set at 7.5.

The parameter search was conducted by first computing a quadratic design for the 4-parameter system with 19 test points [12]. 152 games were played at each of the parameter settings represented by these 19 test points.

After obtaining the win percentage for each of these points, the resulting equation for the surface over the parameter space was calculated and a Monte Carlo search was performed to find the maximum point on that surface inside the constraint space.

The test sidelength was reduced by a factor of 4 for each side, and a new packing design was computed for the smaller constrained set of parameters centered at this predicted maximal setting. 156 games were played at each of 19 new test points.

Another quadratic surface was computed using these values, and a new Monte Carlo search performed. The maximum found this way was compared with the unaltered code (which is represented by the (0,0,0,0) point in parameter space). A two-tailed Z test showed a statistically significant increase in playing strength ($p < 0.05$) with the new parameters.

Orego's win rate for the unaltered code was 50.0%; the win rate for the best setting was 61.1%. This best point sets the probabilities at 0.0625 (proximity), 0.12534 (avoid-the-first-two-lines), 0.03013 (first-order history), and 0.03014 (second-order history).

6. Conclusions and Future Work

This paper has presented a technique for incorporating multiple heuristics into Monte Carlo Go. We have presented four particular heuristics: the proximity heuristic, the avoid-the-first-two-lines heuristic, the first-order history heuristic, and the second-order history heuristic. Finally, we have presented experimental results showing that the use of these heuristics (with appropriate probability settings) significantly improves Orego's ability to defeat GNU Go.

We might judge the relative importance of the different heuristics by the probabilities they are given in the optimal parameter sets. Such a judgment should be taken with a grain of salt because Orego is designed to spend a certain *amount of time* per move rather than to do a certain *number of sampling runs* per move. Consequently, frequent use of an expensive heuristic will reduce the number of sampling runs completed. It may be that the avoid-the-first-two-lines heuristic receives the highest probability because it is so cheap to compute that it is worth using often.

Two of our heuristics can be generalized. We might include versions of the proximity heuristic for playing within d points Manhattan distance of the last move, for different values of d. The avoid-the-first-two-lines heuristic could be generalized to suggest play between lines a and b, inclusive. For example, conventional wisdom suggests that many of the first 5-10 moves in 19x19 Go should be on the 3rd or 4th line. We do not believe that creating a third-order history heuristic (playing the best known response to a particular two-

move sequence) is likely to be worth the computation time.

In the future, we hope to extend our work to 19x19 Go. Monte Carlo programs still perform poorly at this board size, where the space to be sampled is simply too vast. Despite this, humans are able to play 19x19 Go well (better than current programs, at least!). Our hypothesis is that humans do this by breaking the game down into local subproblems and searching out the subproblems semi-independently. Three of our heuristics (all but the avoid-the-first-two-lines heuristic) should help Orego in this realm. We have particularly high hopes for the second-order history heuristic.

As currently written, UCT treats each potential move as having an independent distribution. In Go, there are often large areas of the board that should be avoided because they already belong to one player or another. We may be able to get Orego to spend less time sampling these areas by modifying the UCT algorithm to take into account the sampling results of nearby points.

Acknowledgments

We thank the members of the Computer Go Mailing List for their helpful comments.

References:

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

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

[3] Brügmann, B. 1993. Monte Carlo Go. Unpublished technical report.

[4] Chaslot, G., Saito, J.-T., Uiterwijk, J., Bouzy, B., and van den Herik, H. 2006. Monte-Carlo Strategies for Computer Go. Forthcoming.

[5] 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.

[6] Yoshimoto, H., Yoshizoe, K., Kaneko, T., Kishimoto, A., and Taura, K. 2006. Monte Carlo Has a Way to Go. In *Proceedings of the Twenty-First National Conference on Artificial Intelligence (AAAI-06)*.

[7] Baker, K. The Way to Go. 1986. New York, NY: American Go Association.

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

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

[10] Drake, P., Pouliot, A., Schreiner, N., and Vanberg, B. 2007. The Proximity Heuristic and an Opening Book in Monte Carlo Go. Submitted.

[11] Schaeffer, J. 1989. The History Heuristic and Alpha-Beta Search Enhancements in Practice. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 11(11):1203-1212,

[12] Hardin, R. H. and Sloane, N. J. A. 1993. A New Approach to the Construction of Optimal Designs. *Journal of Statistical Planning and Inference* 37:339-369.