# LOCALIZING SEARCH IN MONTE-CARLO GO USING STATISTICAL COVARIANCE

*Seth Pellegrino*[1]*, Andrew Hubbard*[2]*, Jason Galbraith*[3]*, Peter Drake*[4]*, and Yung-Pin Chen*[5]

Lewis & Clark College, Portland, Oregon

ABSTRACT

The classical Asian game of Go presents a difficult challenge for artificial intelligence programmers because of its vast search space and the difficulty of static board evaluation. One of the techniques humans employ while playing Go is spatial localization – they direct their attention toward the important points on the board and ignore the unimportant areas. This describes using statistical covariance as a tool to execute spatial localization in our Go program, OREGO.

## 1. 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 only play at the level of a strong amateur.

The traditional approach to computer game play has been minimax search with enhancements such as alpha-beta pruning (Russell and Norvig, 2002). Since the number of possible states in a game of Chess or Go is $10^{46}$, (Chinchalkar, 1996), it is not feasible to search the entire game tree. Chess programs therefore rely on static evaluation (estimating the value of a board configuration rather than playing it out). Because Go has a much higher branching factor than Chess examining all legal moves is infeasible, so move heuristics can be used, but they present a risk of discarding good moves. Further, Go board configurations are very volatile – a single stone often changes a winning position into a losing one – so attempting a static evaluation is far from trivial.

Several computer Go programmers, including the authors, have recently begun exploring statistical sampling approaches called Monte-Carlo Go (Brügmann, 1993; Chaslot *et al.*, 2006; Coulom, 2006; Drake and Uurtamo, 2007; Yoshimoto *et al.*, 2006). Monte-Carlo Go programs search for the best move by randomly completing the game thousands of times, then choosing the move that has led to the best average outcome. One very successful program that utilizes this method is known as MOGO[6].

This approach has been surprisingly successful at 9x9 Go; Monte-Carlo programs have dominated computer Go tournaments in recent years. The technique has not been as successful on the 19x19 board, though this is largely due to the much larger search space. Monte-Carlo-based programs have been able to perform quite well in 19x19 games, but the technique is far from a magic bullet. However, the aforementioned MOGO was able to defeat a high-ranked professional with the help of a supercomputer and a sizable handicap (van den Herik, 2008).

After giving a brief introduction to Go in general (and Monte-Carlo Go in particular), we intend to describe our method for focusing a computer Go program's attention on a few crucial points. A point is important if it has a high covariance value, which can be calculated as described below. By focusing on points with high covariance values, our computer Go program, OREGO, became significantly stronger.

[1]email:sethp@lclark.edu
[2]email:hubba1@umbc.edu
[3]email:jason_galbraith@beavton.k12.or.us
[4]email:drake@lclark.edu
[5]email:ychen@lclark.edu
[6]More information about MOGO can be found here: http://www.lri.fr/∼gelly/MoGo.htm

### 1.1 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 (2008).

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 alternate either placing stones on unoccupied intersections or passing. 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[7].

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 own stones to be captured. Third, it is illegal to repeat a previous board position.

### 1.2 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 and then use sampling to evaluate the leaves. This evaluation is performed by playing random moves until the end of the game. With a sufficient number of these playouts, the portion of games won by each side gives a reasonable approximation to the value of the position.

In some cases, we can tell the result of a playout before the end of the game because one player has so many more stones left on the board than his opponent that our program assumes that this player has already won. This concept is called a "mercy threshold" (Hillis, 2006). However, quite often the mercy threshold either does not take effect or is reached after a high number of moves.

#### 1.2.1 UCT

A further refinement of Monte-Carlo Go is to focus the sampling on more promising moves. As the program has a limited amount of time to select a move, this focusing introduces a tradeoff between *exploration*, where we widen the tree by spending time evaluating different moves (hopefully to discover a good move we might have missed before), and *exploitation*, where we deepen the branches of the tree which have produced the best results so far. Many strong Monte-Carlo programs do this using the UCT (**u**pper **c**onfidence bounds applied to **t**rees) algorithm (Kocsis and Szepesvári, 2006; Gelly and Wang, 2006).

When choosing a move to sample, the UCT algorithm picks the move which maximizes the quantity (1), where $W_j$ is the average win rate for playing move $j$ and $U_j$ reflects the uncertainty of $W_j$ (Gelly and Wang, 2006).

$$W_j + pU_j \tag{1}$$

The coefficient $p$ controls the relative importance of exploration and exploitation. If $p$ is high, the program explores all moves extensively before focusing on a few promising moves. If $p$ is low, the program quickly discards moves that fare poorly in early playouts.

Heavily-sampled tree nodes are expanded, so the search is deepest along the most promising line of play (allowing us to ensure that there is no "surprise countermove"). The algorithm is therefore able to make a confident statement about which move is really best to play in the present.

[7] In the sequel we use, for brevity, 'he' and 'his' whenever 'he or she' and 'his and her' are meant.

### 1.2.2 RAVE

Rapid Action Value Estimation (RAVE) is another refinement on the Monte-Carlo Go technique (Gelly and Silver, 2007). RAVE is based on the idea that good moves are temporally independent. That is to say, the RAVE heuristic attempts to calculate the value of playing a move at any point in the rest of the game, and then using that as a proxy for the current value. Thus, a RAVE player will not only tabulate the win rates for moves played at a particular board position, but also the win rates for moves played after that particular board position in a playout. Each playout therefore generates many more data points, and we are able to estimate the value of a move much more quickly. However, it is frequently necessary to play moves in a certain order, and RAVE assumes all permutations of a move sequence are equivalent. So, we see that the RAVE data will never be as accurate as the Monte-Carlo results will, given the same number of data points – the true benefit of RAVE is the speed with which we can generate data points.

The RAVE algorithm tries the move which maximizes (2), where $W_j^{mc}$ is the Monte-Carlo win rate for move $j$ and $W_j^{rave}$ is the win rate where $j$ was played at any point in the future. $\beta$ is a term designed to shift the focus from being initially RAVE-heavy to utilizing the Monte-Carlo data in such a way so as to reduce the mean squared error (Silver, 2008).

$$\beta W_j^{rave} + (1 - \beta) W_j^{mc} \tag{2}$$

## 2. LOCALIZING SEARCHES

One problem with Monte-Carlo sampling is that, particularly on a 19x19 board, the search space is far too large to obtain an accurate evaluation. Initially a computer Go program must consider 361 possible moves. Although this number decreases over time, the branching factor remains quite large. Further, because board positions are so complex, it is very difficult to evaluate the value of a board position (hence the Monte-Carlo simulations).

In order to make the most of our limited time, we would like to apply heuristics (such as the mercy threshold given above) to focus our playouts on the most promising moves. Such focusing we term "localizing," because we would ideally like to spot precisely the maximum number of playouts in the local area of the most important points. This mimics the way in which human players analyze Go positions, where they make the most of their limited thinking time by focusing their attention on a few points that, based on intuition and experience, seem most important.

### 2.1 Covariance as a UCT Localization Tool

Ideally, we would like to achieve our search localization with as few playouts as possible, thus allowing that localization to benefit us for a longer period of time. Control/win covariance is a good measure of point importance that can be reliably calculated quickly.

By the nature of Monte-Carlo Go, we are already generating a large corpus of possible end board positions (i.e. the board states at the end of playouts). We wanted to use this information to do two things: (1) locate the points that we want to control, and (2) further decide which points we can control. Covariance here is very helpful because in order for the win/control covariance to grow large, controlling a point must (a) be strongly correlated with winning the game and (b) vary between players among our corpus.

We define a point to be *controlled by a colour* when either that colour occupies that point or, if the point is vacant, the majority of stones surrounding the point are that colour.[8] Let $B_b$ be the number of times Black has controlled the point and won the playout, $B_w$ be the number of times Black has controlled the point but White won the playout, and so on. The covariance between point control and winning is given in (3).

$$\mathrm{Cov}(\mathit{Win}, \mathit{Control}) = \frac{B_b W_w - B_w W_b}{(B_b + B_w + W_b + W_w)^2} \tag{3}$$

[8] If both colours have an exactly equal number of stones surrounding the point, then we say the point is controlled by neither player

We examined to what extent we might collect data from all of our playouts into a single top-level table. The idea comes from the fact that it is expensive to keep tables for multiple board states. So, we found that the board states that are further in the future tended to have similar distributions to the top-level table, but with a significantly smaller sample size. Thus, we maintained one global table and applied it at multiple levels throughout the search.

Now that we have an estimation for how important each point is, we need a mechanism for focusing our attention on those points. Here it is good to remember that we already have a device which allows us to compare the relative worth of two points – Equation (1). Once we update the UCT formula to (4) (see below), we proceed as normal (running playouts through the points maximizing that value). Thus, we naturally tend to focus our efforts on those points which are critically important, and thus have a high covariance value.

$$W_j + pU_j + 2 \cdot \mathrm{Cov}(\mathit{Win}, \mathit{Control}) \tag{4}$$

## 3. RESULTS

Below we present two types of results, viz. those with UCT (3.1) and those with RAVE (3.2).

### 3.1 UCT

Win/control covariance actually produced surprisingly accurate representations of point importance after a relatively small number of playouts, as can be seen in Figure 1(b). It is immediately apparent to OREGO (as it would be to any human player) that it does not need to spend time considering anything outside of the lower left corner. As we can see in the Figures 1(a) and 2(a) , we arrive at an excellent approximation of importance on the empty board using a relatively small number of playouts. The points found on the 19x19 board are on the 3rd and 4th lines, as recommended by expert human players (Kim, 1998), even though the program has almost no knowledge of Go beyond the rules. Despite the fact that the strict number of playouts is the same for both boards, it takes much longer to come up with accurate covariance data on the 19x19 board because each individual playout takes much longer. However, it still takes only a fraction of the total amount of time needed to come up with a good move, which is exemplified in 2(b). Even after running for 50,000 playouts, plain UCT has yet to discern any meaningful patterns. We should clarify that, at present, the covariance values are only weakly reflected in OREGO's choice of moves on such a large, open board.

In order to determine how much our program increased in playing strength as a result of implementing covariance localization, we played an automated series of games against GNU Go.[9] As we did not know in advance exactly which values of $p$ [10] would be best, we allowed it to vary. Komi for each game was set at 7.5 and OREGO was allowed $16,000$ playouts per move (with $8,000$ of those going toward the covariance of the data collection). We matched each type of player against GNU Go version 3.7.11 for one thousand 9x9 games. The results of these matchings can be seen in Table 1.

**Table 1**: Low $p$ win percentages

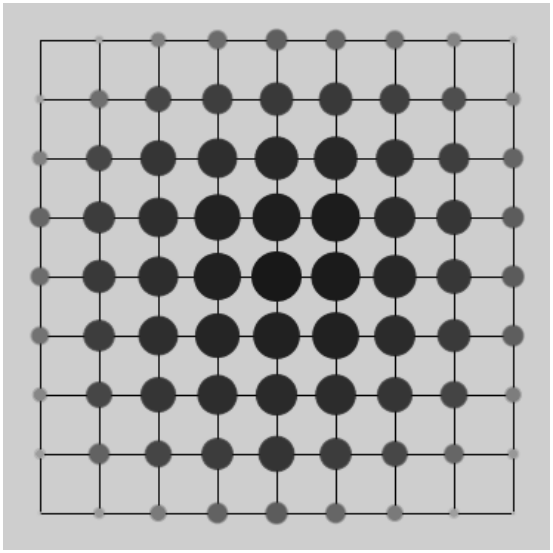
| $p$ | 0.3 | 0.6 | 0.9 | 1.2 | 1.5 | 1.8 |
|---|---|---|---|---|---|---|
| UCT Player | 34.0% | 30.7% | 28.6% | 23.8% | 25.5% | 22.2% |
| Covariance Player | 39.6% | 37.8% | 33.3% | 33.7% | 28.9% | 27.9% |

### 3.2 RAVE

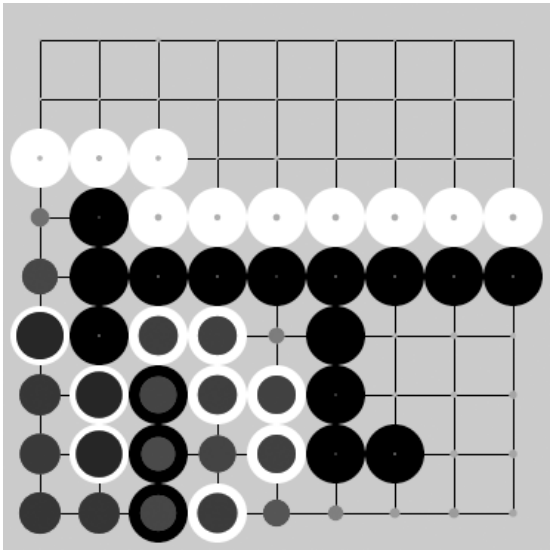
The covariance values remained a strong predictor of point importance, but we were unable to duplicate our successes using a RAVE-based player. When we would manually investigate positions, the covariance would still correctly identify the focal points on the board in only a few playouts. We ran similar experiments as with the UCT-based player, but in all cases our attempts at integrating the covariance data with the RAVE player led to

[9] GNU Go can be found online here: http://www.gnu.org/software/gnugo/

[10] The coefficient that controls the relative importance of exploration versus exploitation

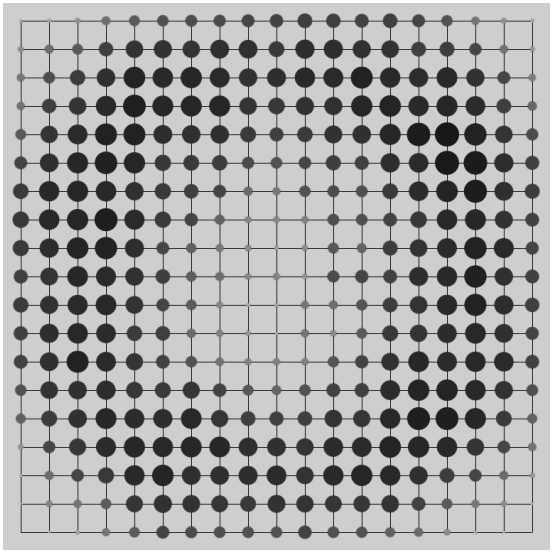


(a) Covariance values after about 50,000 playouts.

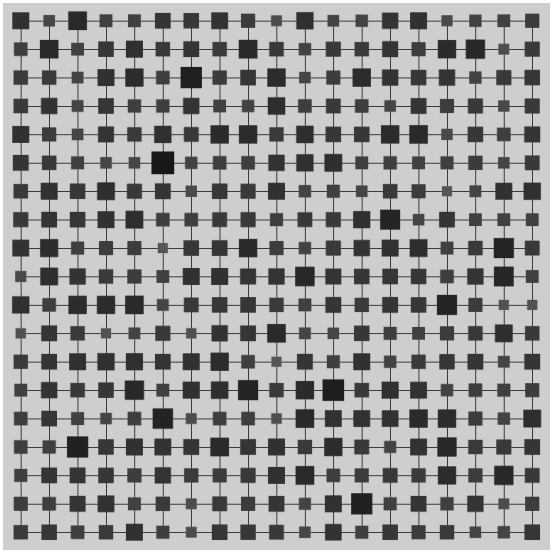


(b) After only 500 playouts.

**Figure 1**: Covariance values on two 9x9 boards. Larger, darker circles represent a higher covariance value.



(a) Covariance values on an empty 19x19 board after about 50,000 playouts.

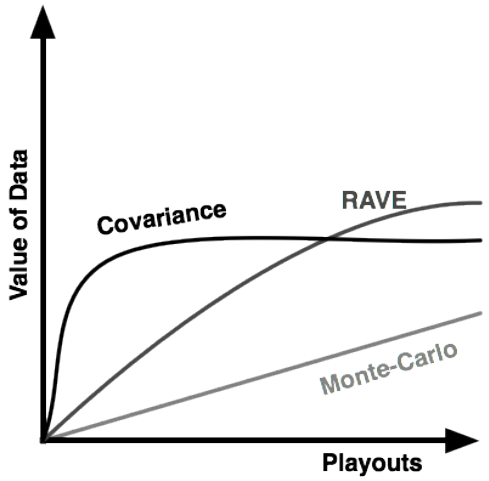


(b) Number of playouts on an empty 19x19 board after about 50,000 playouts.

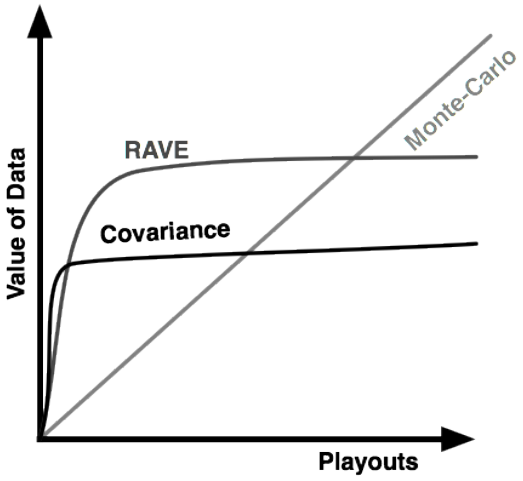
**Figure 2**: Comparison between covariance data and simple UCT playouts

failure. The first of these was to treat it as a stand-in for RAVE information. We would periodically give bonuses to the RAVE win rate of a move based on the covariance value of the point.

However, when looking more closely, we saw that board positions where the search space was small (i.e., near the end of the game) led to a reduction in the helpfulness of the covariance data. This is true because it would, by design, encourage exploration of the moves in the ”high covariance” area. Yet, when the search space was small, the RAVE player would rather quickly identify the correct move, and then the covariance values would act to distract the RAVE search from expanding that branch of the tree by attempting to encourage unnecessary exploration. After further investigation, we discovered the apparent relationship between the various types of data which we can collect. It is presented in Figures 3(a) and (b).




(a) Estimated relative usefulness of data with a large search space.




(b) Estimated relative usefulness of data with a small search space.

**Figure 3**: Estimated data usefulness.

We believe that the covariance data provides a window of usefulness before it is obscured by the RAVE data. To try and capture this helpfulness, we restricted the bonuses which the RAVE win rates of high-covariance moves to a one-time event when we began considering a particular board position. This idea was meant to reduce the amount of unnecessary exploration that the covariance data caused, but we found that it too failed to produce positive results. We believe our technique was not sufficiently powerful – it did not localize the search sufficiently precisely to compensate for the added expense of collecting/computing the covariance data.

We also tried various techniques in an attempt to maximize our use of the covariance data, but we had a difficult time in trying to balance, using the data too little (where we do not recoup the cost of computing it) and too much (where we distract the search). A major reason for this is that we assumed the best way to control a point was to play on that point, but this is not necessarily true. Perhaps the best method is to incorporate the data along with other heuristics into a move prediction engine, as Rémi Coulom (2009) has done in his similar research.

## 4. CONCLUSIONS AND FUTURE WORK

Since, on a 19x19 Go board, computer programs have so many legal moves that there is no time to evaluate all of them accurately, we decided to restrict the number of moves that our program would analyze deeply. This prevented our program from wasting valuable run time on moves we could determine as not important. Instead, our program could give more attention on just a few moves which it expected to be good, improving its efficiency by mimicking the thought processes of human Go players.

In conclusion, we were able to extract the above mentioned information from a relatively small number of Monte-Carlo playouts by determining the covariance between controlling a point and winning the game. Yet, using this data in a state-of-the-art algorithm such as RAVE presents challenges. Intuitively, we see that using covariance data allows us to build an importance map even before RAVE can decide on a move, but using that data may be

a nontrivial problem. Computing how to control a point, even if we know it is important to winning the game, is not a straightforward problem that may require an expensive search on its own.

## 5. REFERENCES

Baker, K. (2008). *The Way to Go.* American Go Foundation. http://www.usgo.org/usa/waytogo/.

Bouzy, B. and Cazenave, T. (2001). Computer Go: An AI Oriented Survey. *Artificial Intelligence*, Vol. 132, pp. 39–102.

Brügmann, B. (1993). Monte Carlo Go. Technical report, Syracuse University.

Chaslot, G., Saito, J.-T., Bouzy, B., Uiterwijk, J. W. H. M., and Herik, H. J. van den (2006). Monte-Carlo Strategies for Computer Go. *Proceedings of the 18th BeNeLux Conference on Artificial Intelligence, Namur, Belgium* (eds. P.-Y. Schobbens, W. Vanhoof, and G. Schwanen), pp. 83–91. http://www.cs.unimaas.nl/g.chaslot/papers/mcscg.pdf.

Chinchalkar, S. (1996). An Upper Bound for the Number of Reachable Positions. *ICGA Journal*, Vol. 19, No. 3, pp. 181–183.

Coulom, R. (2006). Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. *Proceedings of the 5th International Conference on Computer and Games* (eds. H. J. van den Herik, P. Ciancarini, and H. H. L. M. Donkers), Vol. 4630/2007 of *Lecture Notes in Computer Science*, pp. 72–83, Springer, Heidelberg, Germany. http://remi.coulom.free.fr/CG2006/CG2006.pdf.

Coulom, R. (2009). Criticality: a Monte-Carlo Heuristic for Go Programs Criticality: a Monte-Carlo Heuristic for Go Programs. Invited Talk. http://remi.coulom.free.fr/Criticality/.

Drake, P. and Uurtamo, S. (2007). Move Ordering vs Heavy Playouts: Where Should Heuristics Be Applied in Monte Carlo Go? *Proceedings of the 3rd North American Game-On Conference.* https://webdisk.lclark.edu/drake/publications/GAMEON-07-drake.pdf.

Gelly, S. and Silver, D. (2007). Combining online and offline knowledge in UCT. *ICML '07: Proceedings of the 24th international conference on Machine learning*, pp. 273–280, ACM, New York, NY, USA. ISBN 9781595937933. http://dx.doi.org/10.1145/1273496.1273531.

Gelly, S. and Wang, Y. (2006). Exploration exploitation in Go: UCT for Monte-Carlo Go. http://eprints.pascal-network.org/archive/00002713/01/nips_exploration_exploitation.pdf.

Herik, J. van den (2008). Go for a Breakthrough. *ICGA Journal*, Vol. 31, No. 3, pp. 129–130.

Hillis, D. (2006). Fast Board implementation. http://computer-go.org/pipermail/computer-go/2006-December/007478.html.

Kim, J. (1998). *Learn to Play Go: Volume II: The Way of the Moving Horse.* Good Move Press.

Kocsis, L. and Szepesvári, C. (2006). Bandit based Monte-Carlo Planning. *ECML-06.* http://zaphod.aml.sztaki.hu/papers/ecml06.pdf.

Russell, S. J. and Norvig, P. (2002). *Artificial Intelligence: A Modern Approach (2nd Edition).* Prentice Hall. http://aima.cs.berkeley.edu/.

Silver, D. (2008). David Silver's Explaination of the RAVE Formula. http://computer-go.org/pipermail/computer-go/2008-February/014093.html.

Yoshimoto, H., Yoshizoe, K., Kaneko, T., Kishimoto, A., and Taura, K. (2006). Monte Carlo Has a Way to Go. *Twenty-First National Conference on Artificial Intelligence (AAAI-06)*, AAAI Press. http://www.fun.ac.jp/~kishi/pdf_file/AAAI06YoshimotoH.pdf.