Coevolving Partial Strategies for the Game of Go

Peter Drake and Yung-Pin Chen
Department of Mathematical Sciences
Lewis & Clark College
0615 SW Palatine Hill Road
Portland, OR 97219

Abstract

Writing programs to play the Asian game of Go is one of the grand challenges of artificial intelligence. A recent breakthrough has been the development of the UCT algorithm, which uses random sampling biased by the results of previous samples. Genetic algorithms can be described similarly. This paper examines the possibility of using genetic algorithms to find moves in Go. A representation of partial strategies is presented, as is a multithreaded, steady-state coevolution algorithm. Experimental results demonstrate that coevolution without domain-specific knowledge can discover the correct move, effectively performing distributed tree search, quickly eliminating unpromising moves and dividing exploratory computation across multiple search paths.

Keywords: Go, game play, tree search, genetic algorithms, coevolution

1. Introduction

Writing programs to play the Asian game of Go is one of the grand challenges of artificial intelligence [3]. While the best Chess programs now play at the world champion level, the best Go programs are still at the strong amateur level.

The rules of Go can be stated succinctly, although uninitiated readers are directed to less terse introductions [1, 12]. The board is a grid of lines. (19x19 is the normal size, but this can be changed without altering any other rules. A 9x9 board, common both for teaching new human players and for computer Go research, is used here.) There are two players, black and white, with black playing first. Players alternate turns, with a turn consisting of either placing a stone of one's color on an empty intersection or passing. If, after a turn, a contiguous block of the opponent's stones has no adjacent vacant intersections, those stones are removed. It is illegal to capture one's own stones or to repeat a previous board position. The game ends when both players pass consecutively. A point is scored for each intersection occupied or surrounded by friendly stones. (Some rule sets give slightly different, but generally equivalent, ways of counting the score.) The high score wins.

Why is Go so hard for computers to play? Specifically, why doesn't brute-force minimax search, which has worked so well for other games, produce a strong Go player? Two features of Go are frequently cited. First, it is a very large game. There are approximately 10^{160} board positions, compared with 10^{50} for Chess [3]. Second, estimating the value of a given board position is extremely difficult. The presence of a single stone can alter the capturability of an enormous block of stones, radically changing the outcome of the game.

Until recently, the strongest Go programs were painstakingly handcrafted by programmers who were themselves strong players [20]. In the last couple of years, however, enormous strides have been made using Monte-Carlo techniques [5, 10]. These techniques offer a surprisingly simple method of evaluating a board position: play random moves to the end of the game and see who wins. A position can be evaluated by counting the number of wins over many such random playouts.

The strongest Monte-Carlo programs are based on the UCT algorithm [14]. This algorithm builds a search tree, using Monte-Carlo playouts for position evaluation. UCT guides the search to spend more computational effort (more playouts) on more promising branches of the tree. With some enhancements, a UCT-based program has been able to achieve master-level play on the 9x9 board [9].

UCT works by random sampling, using past samples to bias future samples. This is also an apt description of genetic algorithms, which solve problems by simulated evolution [8].

In this paper, we explore the possibility of using genetic algorithms to play Go. While considerable work remains to be done, we believe this approach may eventually have two advantages over UCT. First, since UCT maintains a tree as a central data structure, there are limitations to parallelizing the algorithm [4]. Genetic algorithms have no such limitations. Second, since UCT is a global search, move sequences explored in one part of the tree must be redundantly explored if they appear elsewhere in the tree, even if the intervening moves do not affect the correctness of the sequence. Human Go players are able to read out spatially local sequences, largely independent of the

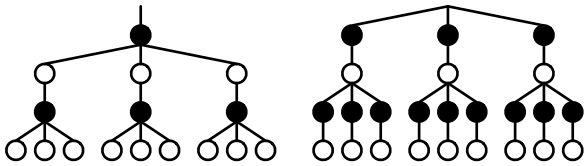

Figure 1: Strategies for black (left) and white (right) players in a simple game. In this game, each player makes two moves, black starting and play alternating after that. There are three options at each move. The color of a node indicates who played the move leading to that position.

rest of the board. The ability of genetic algorithms to combine partial solutions [11] may be helpful here.

Previous work on genetic algorithms and Go has attempted to evolve functions to suggest moves or to evaluate board positions. Evolved structures have included neural networks [15, 18, 25], cellular automata [22, 23], or combinations of handcrafted features [21, 24]. No strong Go programs have resulted from this work. A key difficulty has been the computational expense of playing vast numbers of games. One study used "nearly two processor-years of time to play over 40 million games of Go" [22].

Our plan is at once less and more ambitious. Rather than evolve a general function for playing Go from any board position, we will evolve partial strategies for the current position. On the other hand, we intend to do this in real time, during a game against a human or computer opponent. The idea of a Monte-Carlo playout allows two individuals to be compared extremely quickly. While further enhancements and optimizations will be necessary to reduce the response time to seconds, as is necessary for competitive play, the system described below simulates millions of steps of evolution in a matter of minutes.

We use coevolution, a form of evolution in which the fitness of an individual depends on other individuals in the same or another population [22, 26]. In the context of genetic algorithms, coevolution is beneficial when an objective fitness function is not available to provide evolutionary pressure. We use two-population coevolution, with one population of partial strategies for black and one of partial strategies for white. The hope is for the two populations to become embroiled in an evolutionary arms race, with improvements in each driving improvements in the other.

The next section clarifies our notion of what a partial strategy is and how it is represented in our system. A description of our steady-state, parallel coevolutionary algorithm follows. Experimental results are then presented, followed by conclusions.

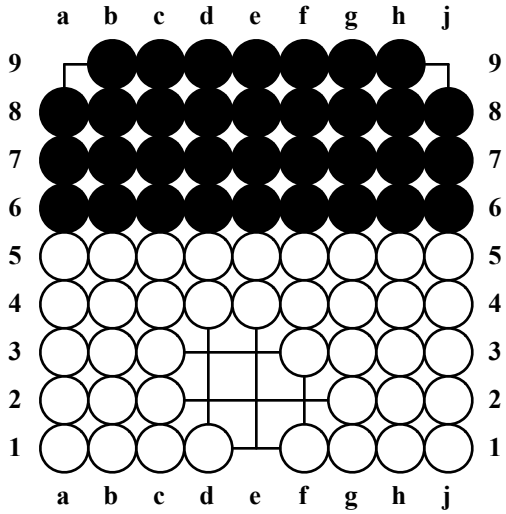

Figure 2: The flower six position, with black to play. The labeling of the last column as j, rather than i, is traditional, avoiding any confusion between i and 1

2. Representation of Partial Strategies

A *game* can be represented as a tree, with each node corresponding to a particular board position. The root node is the position at the beginning of the game. The children of each node are the positions reachable in one move.

A *strategy* for a player indicates how the player responds to any board position. This is a subset of the game tree, with at most one child for positions where it is the player's turn (Figure 1).

A complete strategy for Go, even on the 9x9 board, would be astronomically large. We instead choose to represent *partial strategies*. A partial strategy is part of a complete strategy, starting at the root but not continuing all the way to the leaves. When a game proceeds beyond the moves specified by the partial strategy, the player's moves are selected by some other method (e.g., randomly). A good partial strategy guides the game to a position that maximizes the chance of winning.

Techniques for evolving trees have been established in the context of genetic programming [16]. While we could attempt to evolve partial strategies as described above, such trees can be very fragile. As an example, consider the position shown in Figure 2.

In this position, the black stones cannot be captured because of the internal vacant points (*eyes*) at a9 and j9; white must occupy both of these points to capture black, but either move would be suicidal and hence illegal. White controls more of the board, so the outcome of the game hinges on whether the white stones can be captured. Specifically, can white divide the internal space around e2 into two separate eyes?

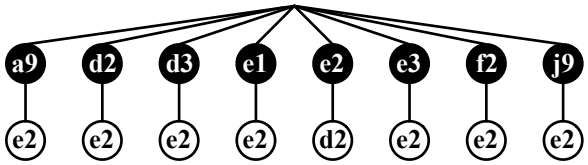

Figure 3: White's partial strategy for the flower six position. The moves at a9 and j9 are legal, if foolish, for black.

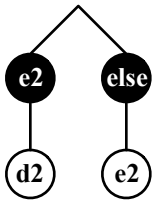

Figure 4: White's partial strategy represented as exceptions (respond to e2 with d2) and a default move (e2).

Even moderately strong amateur Go players will recognize the shape around e2 as the flower six [13], also known as the rabbitty six [6]. Black can kill the white group by starting at e2. Some followup is necessary: if white responds with e3, black must then play d2, and vice versa.

If black begins with e2, white should at least try playing either e3 or d2 and hope black makes a mistake. If black does not begin with e2, white should play there. This partial strategy is rather large, as indicated in Figure 3.

In order to play correctly, white's partial strategy must recommend the move e2 in no less than seven places. This is particularly problematic in a coevolutionary context, where the population of black players would be rewarded for playing a move to which white does not respond correctly. The tree is also very large for the amount of information it contains, making it difficult to learn and to understand.

We therefore chose a different representation where, at each decision point, a strategy specifies a number of exceptions and a default move. In this case, there is one exception: if black plays e2, play d2. The default move is e2. This tree is shown in Figure 4.

For purposes of evolution, we represent this tree as a binary tree. At nodes where the player is making a decision (e.g., the root), the left subtree is the beginning of the list of exceptions and the right subtree is the default move. Within the list of exceptions, the left subtree is the response to that exception and the right subtree is the next exception. Since every internal node has two children and can be represented by a single integer, and every leaf can be represented by some sentinel value, a prefix listing of the tree specifies it completely. This listing can be stored compactly in an array, as is common in genetic programming implementations.

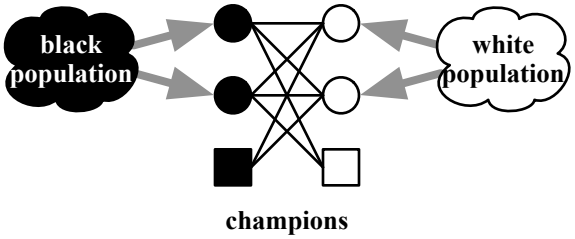

Figure 5: Fitness testing. Two individuals are chosen randomly from each population. Each plays against the two from the opposing population, as well as against an opposing champion. A total of eight games are played.

3. Coevolutionary Algorithm

We choose a steady-state coevolutionary algorithm [8] for two related reasons. First, a steady-state algorithm allows us to continuously vary the amount of computation performed; this will be important when the algorithm is incorporated into a full-blown player that must perform under tight time constraints. Second, we believe a steady-state algorithm will reduce the synchronization overhead when computation is spread over a cluster of several machines.

Our algorithm is also multithreaded, allowing us to take full advantage of a single multiprocessor (or multicore) machine.

Our basic algorithm is a variant of Miconi and Channon's N-strikes-out algorithm [19], with $N = 1$. Each thread repeatedly performs the following steps:

1. Randomly choose two individuals from each population.
2. Play the four possible games between members of opposing populations. Each individual therefore wins from 0-2 games. This is the fitness metric.
3. Within each population, the individual who won the most games (with ties broken randomly) is the winner; the other individual from the same population is the loser.
4. The loser is replaced by a new individual.

Preliminary experiments suggested that this simple algorithm suffered from *focusing*, a common problem in coevolution [26]. Focusing occurs when one population is taken over by individuals of one type; the other population then learns to do well only against that type of individual, ignoring others. In the flower six position, for example, suppose that the white population is dominated by the partial strategy in Figure 4. The black population might then learn to respond to d2 with e3, but be helpless against a white player that responded to e2 with e3.

In generational coevolution, a common remedy to focusing is to maintain a "Hall of Fame" of highly-fit individuals from previous generations [22]. Analogously, we have each thread maintain a *champion* for each population. In step 2 of the

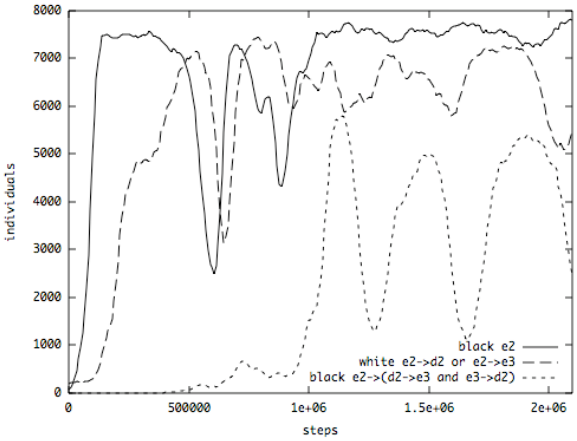

Figure 6: A single run. A step is the evaluation of two individuals from each population, as in Figure 5. See text for discussion.

algorithm above, each individual now plays three games: two against randomly-chosen individuals from the other population and one against the champion (for this thread) from the other population (Figure 5). An additional step is added:

5. If the winner won all three games, it replaces the champion (for this thread) for this population with some probability.

4. Experimental Setup

All games played are 9x9. Komi (the compensation granted to white to make up for the disadvantage of moving second) is set at 7.5 points.

Games are played by following the individuals' partial strategies as described previously. At points where a strategy has no suggestion, a random move is played. A random move is also played if the partial strategy selects an illegal move. We had considered declaring a loss for any individual that selected an illegal move, but we believed this would encourage a tactic of playing strange moves to "trick" the opponent into making an illegal move. Furthermore, the irrelevant "junk DNA" contained in unreachable branches of the tree may be beneficial to evolution [27].

As is standard in Monte-Carlo Go, one very simple heuristic is incorporated into the random moves: do not move into a possible friendly eye. This is done not so much to improve the quality of play as to save computation time by avoiding extremely long playouts where both players repeatedly destroy their own eyes. This heuristic is the only Go-specific knowledge provided to the program.

Trees are generated to a height of 4, with each node equally likely to be any point on the board, the pass move, or the null sentinel. This is the "grow" method described by Koza [16].

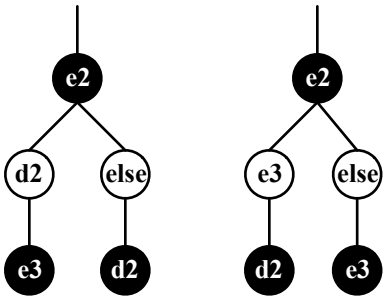

Figure 7: Two correct black partial strategies for the flower six position.

When a new individual is to be generated, it is either a cross between two random individuals (probability 0.9) or a clone of a random individual (p=0.1). In either case, the new individual is then mutated with probability 0.1.

Mutation and crossover occur as per Koza. Tree points were chosen from among internal nodes (if any) (p=0.9) and from among leaves (p=0.1). If a mutation or crossover would produce a tree with more than 127 nodes (enough for a full tree of height 6), the parent is simply cloned instead.

The probability of champion replacement (within each population) was chosen so that, on average, one thread's champion would be replaced every 8192 times an individual won all three of its games.

5. A Single Run

Figure 6 shows the results of a single run, chosen neither for its typicality nor its excellence, but for its clear evidence of frequently-occurring phenomena. This run evolved solutions to the flower six position, with 8192 individuals in each population and 64 threads.

The solid line (on top during most of the run) shows the number of individuals in the black population that start with e2. The black population is quickly dominated by such individuals because, with subsequent random play, black is much more likely to win with this stone in place.

The dashed line shows the number of white individuals that respond to e2 with either d2 or e3. This also grows fairly quickly once black is playing e2.

Around 500k steps, the black e2 population abruptly crashes. We believe this happens because white has discovered a strategy of playing d2 and then e3 (or vice versa) and black has not yet found a response. With black not playing e2, there is less pressure for white to respond to this move, so the white population also crashes. This improves the value of black e2, so the black population recovers, followed by the white population. Less severe oscillations follow.

The dotted line shows the number of black individuals with one of the partial strategies shown in Figure 7. (These individuals may have other

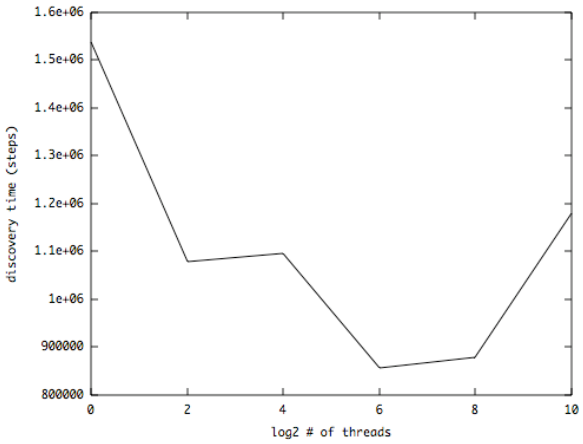

Figure 8: First discovery time of e2 in the flower six position as a function of the number of threads. The horizontal scale is logarithmic. Results are means over 20 runs.

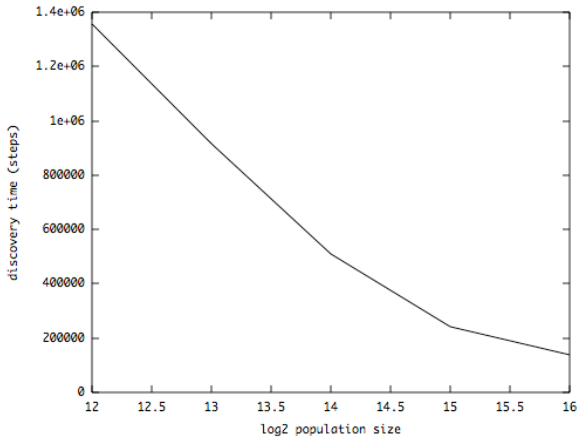

Figure 9: First discovery time of e2 in the flower six position as a function of the population size. The horizontal scale is logarithmic.

exceptions; we only checked their response to white d2 and e3.) While there is some oscillation, these individuals maintain a healthy presence in the population once they appear.

Two phenomena deserve emphasis. First, when a move isn't working, coevolution explores alternatives. Second, as coevolution proceeds, exploration occurs more deeply in the game tree.

6. Experiment 1: Number of Threads

Our first experiment examined the effect of changing the number of threads the program runs. We adjusted the number of evaluation steps performed per thread so that the total amount of computation is the same in each condition. Since the amount of synchronization between threads is minimal (they only have to be careful not to select the same individuals from the populations), the wall clock time is also equivalent in almost all conditions. The exceptions are those where the number of threads is less than the number of processor cores. For example, on the dual-core processor on which these experiments were run, the one-thread condition took twice as long as any other condition.

With computation equalized, the main effect of changing the number of threads is to change the size of the virtual Hall of Fame represented by the champions. If the number of threads is too small, the champions provide no significant memory, making the coevolution unstable. If the number of threads is too large, the champions may provide so much inertia as to stifle innovation.

When these techniques are incorporated into a real player, the player must choose a move after coevolution has continued for some time. This move will be the move suggested by the most individuals in the population. For this experiment, we therefore chose to measure the time (number of steps) at which the

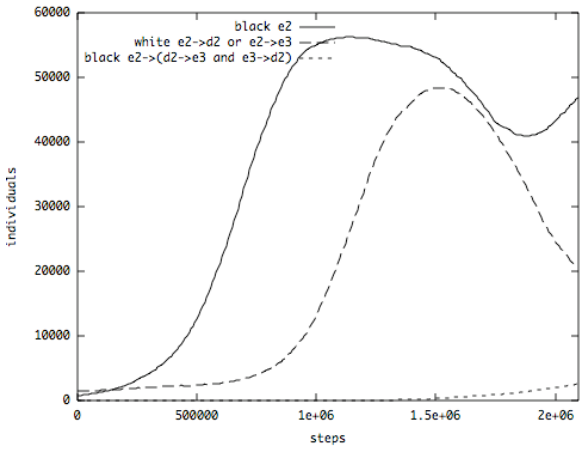

Figure 10: With a very large population, the black partial strategy that responds to both white responses only begins to appear at the end of the run.

correct move gained such a plurality. Because this plurality may occasionally be lost (e.g., around 600k steps in Figure 6), we chose to measure the the last *permanent* discovery time, i.e., the time at which e2 gained plurality and kept it until the end of the run (2^{21} steps).

The results of this experiment are shown in Figure 8. We conclude that 64 threads works best.

7. Experiment 2: Population Size

We next examined population size. A larger population should protect diversity, preventing the loss of strong individuals. Because of the steady-state nature of our algorithm, increasing the population size does not directly increase computation time, although it does increase the number of steps required for one type of individual to take over the population.

While the results (Figure 9) suggest that the population should be very large, examination of

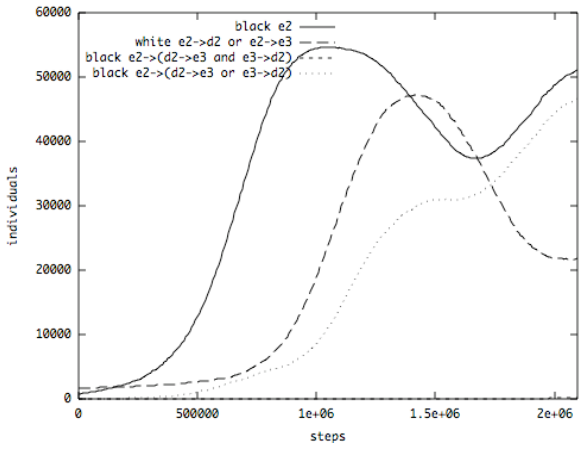

Figure 11: A typical run, showing the number of black individuals that responded correctly to at least one of the white responses to e2 (fourth line).

individual runs gave us pause. Figure 10 shows a typical run with the largest population size. Since the partial strategies from Figure 7 only begin to appear at the end of the run, we worried that the runs with larger populations may not be exploring the game tree as deeply as other runs.

To allay our fears, we did some more runs that plotted an additional metric: the number of black individuals that started with e2 and then responded correctly to at least one of the white responses. If such individuals are not present, the tree is not being explored.

Such individuals are, in fact, consistently present. A typical run for the largest population size is shown in Figure 11. This is a fascinating result: even though there are few if any individuals containing all of the required partial strategy, the strategy is present in the population in distributed form.

8. Experiment 3: The Multiple Cut Position

For our final experiments, we examined a different board position, shown in Figure 12. While not technically invulnerable, the white stones at the top and the black stones in the center cannot be captured short of majestically bad play on the part of the defenders. The action therefore centers on the white stones at the bottom of the board.

Taking into account the 7.5 point komi, black needs to gain three points to win. Capturing any of the white stones would do the trick, so white must respond to any of the six moves c1, c2, e1, e2, g1, and g2 with the other move in the same column. If white plays correctly, black can still win by playing c2, e2, and g2 in any order.

We believe this is a more difficult position because there is more than one correct move and because there are many more legal but incorrect moves (e.g., f5).

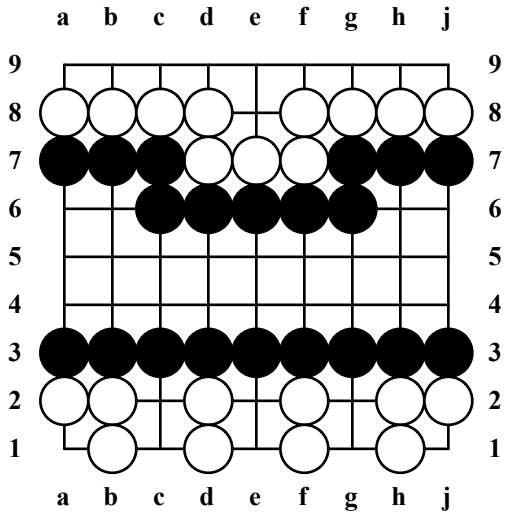

Figure 12: The multiple cut position, with black to play.

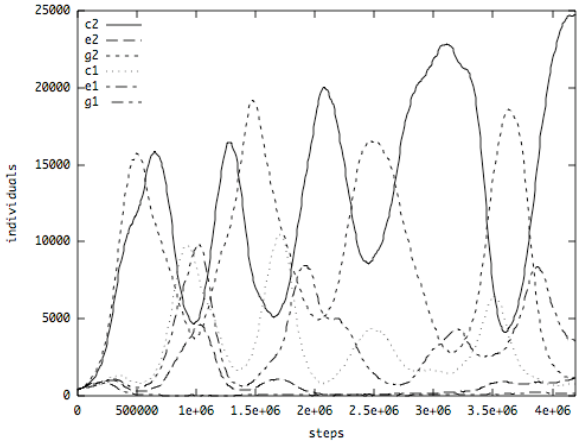

Figure 13: One run on the multiple cut position. The number of black individuals opening with each of the six key points are plotted. Population size was 32768. Note that the total run time is twice as long as in previous experiments.

One run is shown in Figure 13. The population is quickly dominated by the six moves mentioned above, despite the availability of many other legal moves. Through most of the run, the population oscillates between two of the correct moves, c2 and g2. Coevolution is thus exploring both strong moves, exhibiting behavior similar to UCT.

The moves e1 and e2 fare poorly in all runs because, if all subsequent moves are random, these are not as strong as the moves in the c and g columns. In some runs, either c2 or g2 became permanently dominant. In others, c1 or g1 occasionally gained temporary plurality even late in the run.

9. Conclusions

This paper has explored coevolution as a technique for finding the correct move in the game of

Go. We have presented a way of representing partial strategies and a multithreaded, steady-state coevolutionary algorithm.

Experimental results have demonstrated that coevolution can perform tree search. Experiment 1 explored the optimal number of threads to use in our algorithm. Experiment 2 examined the effect of population size and showed that a partial strategy can be present in the population in distributed form. Experiment 3 challenged the system with a considerably more difficult problem, demonstrating that the system was able to quickly eliminate unpromising moves and divide exploratory computation across more than one search path.

Considerable future work remains to be done. Drawing on the UCT research, we might heuristically bias the random moves, performing "heavy playouts" [7]. We might also bias the distribution of points chosen in the initial trees and in mutations.

Within the genetic algorithm context, alternative mutation and crossover operations may be helpful. For example, new mutation operators could add or remove exceptions at a move. Alternate representations may also be explored.

A genetic approach offers the possibility of combining useful building blocks. Koza's Automatically Defined Functions [17] may be beneficial here. In Go, building blocks correspond to local search. Some work has been done on combining local searches at the very end of the game when the board can be broken down into several completely independent situations [2]. The truly fascinating and challenging part of the game, though, is the early middle game, when there are several semi-independent situations. Humans still outperform computers at this kind of soft decomposition, but genetic algorithms just might be the right approach for this kind of problem.

References

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

[2] Berlekamp, E. and Wolfe, D. *Mathematical Go: Chilling Gets the Last Point*. Wellesley, Massachusetts: A K Peters, 1994.

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

[4] Cazenave, T. and Jouandeau, N. On the Parallelization of UCT. *Computer Games Workshop*, 93-101, 2007.

[5] Coulom, R. Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. *Lecture Notes in Computer Science*, Vol. 4630, 72-83, Springer, 2007.

[6] Davies, J. *Life and Death*. Tokyo, Japan: The Ishi Press, 1975.

[7] Drake, P. and Uurtamo, S. Move Ordering vs Heavy Playouts: Where Should Heuristics Be Applied in Monte Carlo Go? In *Proceedings of the 3rd annual North American Game-On Conference*, 2007.

[8] Eiben, A.E. and Smith, J.E. *Introduction to Evolutionary Computing*. Springer Berlin/Heidelberg, 2003.

[9] Gelly, S. and Silver, D. Combining Online and Offline Knowledge in UCT. *International Conference of Machine Learning* (Corvallis, Oregon), 273-280, 2007.

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

[11] Holland, J. *Adaptation in Natural and Artificial Systems*. Cambridge, Massachusetts: The MIT Press, 1992.

[12] Kim, J. and Jeong, S. *Learn to Play Go, Vol. 1*, 2nd ed. Sheboygan, WI: Good Move Press, 1997.

[13] Kim, J. and Jeong, S. *Learn to Play Go, Vol. 2*, 2nd ed. Denver, CO: Good Move Press, 1998.

[14] Kocsis, L. and Szepesvari, C. Bandit Based Monte-Carlo Planning. In *15th European Conference on Machine Learning*, 282-293, 2006.

[15] Konidaris, G., Shell, D., and Oren, N. Evolving Neural Networks for the Capture Game. In *Proceedings of the SAICSIT Postgraduate Symposium* (Port Elizabeth, South Africa), 2002.

[16] Koza, J. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, Massachusetts: The MIT Press, 1992.

[17] Koza, J. *Genetic Programming II: Automatic Discovery of Reusable Programs*. Cambridge, Massachusetts: The MIT Press, 1994.

[18] Mayer, H. and Maier, P. Coevolution of Neural Go Players in a Cultural Environment. *IEEE Congress on Evolutionary Computation*, Vol. 2, 1017-1024, 2005.

[19] Miconi, T. and Channon, A. The N-Strikes-Out Algorithm: A Steady-State Algorithm for Coevolution. *IEEE Congress on Evolutionary Computation*, 1639-1646, 2006.

[20] Müller, M. Computer Go. *Artificial Intelligence*, 134 (1-2):145-179, 2002.

[21] Pratola, M. and Wolf, T. Optimizing GoTools' Search Heuristics Using Genetic Algorithms. *ICGA Journal*, Vol. 26, 28-48, 2003.

[22] Rosin, C. *Coevolutionary Search Among Adversaries*. Ph.D. thesis, University of California, San Diego, 1997.

[23] Rosin, C. and Belew, R. Methods for Competitive Coevolution: Finding Opponents Worth Beating. *Proceedings of the 6th International Conference on Genetic Algorithms*, 373-381, 1995.

[24] Rutquist, P. Evolving an Evaluation Function to Play Go. Technical report, Ecole Polytechnique, 2000.

[25] Stanley, K and Miikkulainen, R. Evolving a Roving Eye for Go. *Proceedings of the Genetic and Evolutionary Computation Conference*, New York, NY: Springer-Verlag, 2004.

[26] Watson, R. and Pollack, J. Coevolutionary Dynamics in a Minimal Substrate. *Proceedings of the Genetic and Evolutionary Computation Conference*, 702-709, 2001.

[27] Wu, A. and Lindsay, R. Empirical Studies of the Genetic Algorithms with Non-coding Segments. *Evolutionary Computation*, Vol. 3, Issue 2, 121-147, 1995.