Noname manuscript No.
(will be inserted by the editor)

Tristan Cazenave

Evolving Monte-Carlo Tree Search Algorithms

19 December 2007

Abstract Monte-Carlo tree search is a recent and powerful algorithm that has been applied with success to the game of Go. Combining it with heuristics for the early development of nodes with few samples makes it even stronger. We use Genetic Programming to evolve such heuristics. The heuristics discovered by Genetic Programming outperform UCT and RAVE. Our Genetic Programming system computes the fitness using a Swiss tournament, and it selects valid individuals.

Keywords Genetic Programming · Co-Evolution · Monte-Carlo Tree Search · Game of Go · Signature Heuristic

1 Introduction

Genetic Programming is a well established technique that has been successfully used for many problems [16]. Recent success in the field of games include the evolution of backgammon players using competition between players [2], of sumo-fighting robots [18], and of chess endgame players [14], these results are summarized in [19].

In this paper, we detail the application of Genetic Programming to the discovery of heuristics for developing Monte-Carlo search trees in the game of Go.

The second section deals with the game of Go from a programmer's point of view. The third section traces back the applications of Monte-Carlo algorithms to the game of Go. The fourth section explains the specificity of our Genetic Programming system. The fifth section describes the various components used in our individuals. The sixth section presents experimental results.

L.I.A.S.D. Dept Informatique, Université Paris 8
93526, Saint-Denis, France
E-mail: cazenave@ai.univ-paris8.fr

2 The game of Go

Writing a computer player for the game of Go is notoriously difficult [5]. The complexity comes from the high branching factor of the game and from the lack of a good and simple positional evaluation function. A recent and powerful development is Monte-Carlo Go, which evaluates a position by playing many pseudo-random games (playouts) from this position, and grows a search tree in an incremental and best first manner. In 9x9 Go all the best programs are Monte-Carlo based, and even in 19x19 Go MoGo and CrazyStone, which are Monte-Carlo programs, have finished first and second of the 2007 computer Olympiad. MoGo uses the RAVE algorithm [12] and CrazyStone uses pattern based progressive widening of the Monte-Carlo tree [11].

3 Monte-Carlo Tree Search

In this section we expose related works on Monte-Carlo Go. We first explain basic Monte-Carlo Go as implemented in Gobble in 1993. Then we address the combination of search and Monte-Carlo Go, the improvement of playouts, followed by the UCT and RAVE algorithms.

3.1 Monte-Carlo Go

The first Monte-Carlo Go program is Gobble [7]. It uses simulated annealing on a list of moves. The list is sorted by the mean score of the games where the move has been played. Moves in the list are switched with their neighbor with a probability dependent on the temperature. The moves are tried in the games in the order of the list. At the end, the temperature is set to zero for a small number of games. After all games have been played, the value of a move is the average score of the games it has been played in first. The only Go knowledge used by Gobble is to avoid eye filling moves during playouts, which is a very basic and simple, yet powerful Go knowledge. Gobble-like programs have a good global sense but lack of tactical knowledge. For example, they often play useless Atari, or try to save captured strings.

3.2 Search and Monte-Carlo Go

Research on Monte-Carlo Go resumed with the work of B. Bouzy and B. Helmstetter that experimentally verified that simple sampling and a depth one search was effective [6].

The combination with search was then tried in multiple ways, either by progressively pruning the moves according to their mean and variance [4], or by selecting move based on tactical search [3], or by computing statistics on goals instead of computing statistics only on moves [9].

Then, a very effective way to combine search with Monte-Carlo Go has been found by R. Coulom with his program Crazy Stone [10]. It consists in adding a leaf to the search tree for each playout. The choice of the move to develop at a node of the search tree depends on the comparison of the results of the previous playouts that went through this node, and of the results of the playouts that went through its sibling nodes.

3.3 Biasing playouts

It is possible to improve on completely random playouts. Crazy Stone uses high urgency to capture and save stones in Atari [10], Golois improves on this with the simple knowledge of playing the Atari that would provide the most liberties to the opponent [8].

MoGo successfully uses patterns to bias playouts, it only verifies some 3x3 patterns near the previous move of the playout and also gives a high urgency to capture [13]. Recent improvements consist in ranking patterns to bias playouts and UCT [11].

3.4 UCT

The UCT algorithm [15] develops the tree in a way similar to Crazy Stone, except that it uses another formula to select the moves to explore. It has been applied with success to Go in the program MoGo [13] among others.

When choosing a move to explore, there is a balance between exploitation (exploring the best move so far), and exploration (exploring other moves to see if they can prove better). The UCT algorithm addresses the exploration/exploitation problem. UCT consists in exploring the move that maximizes $val_i = \mu_i + c \times \sqrt{log(games)/games_i}$. The mean result of the games that start with the c_i move is μ_i, the number of games played in the current node is $games$, and the number of games that start with move c_i is $games_i$.

The c constant can be used to adjust the level of exploration of the algorithm. High values favor exploration and low values favor exploitation.

3.5 RAVE

The RAVE algorithm [12] improves on UCT by using a rapid estimation of the value of moves when the number of games of a node is low. It uses a constant k and a parameter β that progressively switches from the rapid estimation heuristic to the normal UCT value. The parameter β is computed using the formula: $\beta = \sqrt{\frac{k}{3 \times games + k}}$. β is then used to bias the evaluation of a move in the tree with the formula: $val_i = \beta \times heuristic + (1.0 - \beta) \times UCT$. Experiments with MoGo have found that k = 1,000 gives good results.

4 Principles of Evolution

We only use one population, so we do not use co-evolution as defined by the competition of individuals from two or more differing populations as described in [17] for example. However we evaluate the fitness of individuals making them compete against each other.

4.1 Swiss Tournament

In [1], the authors propose three competitive fitness functions, either full competition, or bipartite competition, or a direct elimination tournament.

Instead of playing a direct elimination tournament between individuals, we think that it is more accurate to run a Swiss tournament. An accurate fitness function would be to play many games against a fixed and strong opponent such as GNUGO for example. However, it would take much more time, and we have a balance between the time spent evaluating individuals and the advancement of the evolution. Morevoer, it does not provide variety of the opponents and may therefore prove less robust. A Swiss tournament is a good compromise between the time spent for the evaluation of individuals, and the quality of the resulting order of individuals. For example, in a direct elimination tournament, if the best individual meets another strong individual in the first round, the other one is discarded and will not reproduce. In a Swiss tournament, the defeated one plays additional rounds that will give it chances to recover from its initial loss.

In the first round of the tournament, every individual is randomly paired against another individual. In the second round, the individuals with one win play against the other individuals with one win, and the individuals with zero win play against the other zero win individuals. In subsequent rounds, individuals are matched with other individuals that have the same number of wins.

At the end of the tournament, the primary factor for sorting the individuals is the number of wins. The secondary factor is the sum of opponents scores (SOS), i.e. the sum of the number of wins of the other individuals met during the tournament. Therefore, the fitness of an individual at the end of the tournament is: $fitness = wins \times 100 + SOS$.

Individuals are sorted according to their fitness.

The number of rounds of the tournament is also a compromise between the time spent evaluating the individuals and the advancement of evolution. In our experiments we use a four rounds Swiss tournament.

4.2 Reproduction and Mutation

The first one-eighth portion of the sorted individuals are granted four reproductions. From one-eighth to one-quarter of the population, the individuals are granted two reproductions. From one-quarter to one-half of the population, individual are granted only one reproduction.

The mutation operator consists in replacing a leaf of an individual by another randomly chosen leaf with a given probability. The mutation operator is applied at every leaf of every individual of a new population, just after it has been created with reproduction.

4.3 Valid expressions

We have found that it helps genetic programming a lot to detect and destroy individuals that are clearly not appropriate. In the Monte-Carlo tree search application it is clear that an individual not containing any element related to the move under consideration cannot be relevant to choose between moves.

In order to constrain genetic programming, we use a function that assess the validity of individuals. Individual that are not valid are destroyed and replaced by randomly created and valid individuals.

5 Components of Expressions

The basic components we use to build individuals are the classic functions: +, -, *, /, logarithm, exponential, square root. We also use the constants 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 20.

Components which are specific to Monte-Carlo tree search and which are related to the current node but not to the move under consideration are:

- PlayoutParent : the number of playouts already played from the current node.
- MeanParent : the mean result of the playouts already played from the current node.
- BestMean : the best mean among the children of the node.
- BestSumSquares : the sum of the squares of the results of the child with the best mean.
- BestGames : the number of games of the child with the best mean.
- BestSigma : the standard deviation of the child with the best mean.
- allGamesAMAF : the sum over all the children of the number of playouts that contained the move associated to the child.
- allPlayoutsSignature : the sum over all the node's legal moves of the number of playouts that contained the move with the same signature as the node's move.

Components which concern the move under consideration are:

- Mean : the mean result of the playouts that start from the current node with the move.
- Playouts : the number of playouts that start from the current node with the move.
- Sigma : the standard deviation of the results of the playouts that start from the current node with the move.
- AMAF : the mean result over all the playouts (including those that do not pass through the current node) of the playouts that contain the move.
- AMAFSignature : the mean result over all the playouts (including those that do not pass through the current node) of the playouts that contain the move played with the same signature as the move under consideration. The signature is a hash code that represents the four immediate neighbors of a move. Two moves have the same signature if they have the same four neighbors at the time they have been played.

Table 1 Performance of UCT with 10,000 playouts against GNUGO 3.6 for different constants.

c = 0.1	c = 0.2	c = 0.25	c = 0.3	c = 0.35	c = 0.4	c = 0.5
8.5%	22.0%	28.5%	40.0%	35.5%	28.0%	36.0%

– gamesAMAF : the number of playouts that have passed through the node and that contain the move.
– localAMAF : the mean result of the playouts that have passed through the node and that contain the move.

The AMAF heuristic has been described in GOBBLE [7]. The localAMAF heuristic has been used in MOGO to implement RAVE [12]. The AMAFSignature heuristic is specific to GOLOIS.

6 Experimental Results

In order to select the best constant for UCT, we played several 9x9 games against GNUGO 3.6. For each tested constant, 200 games were played between UCT with 10,000 playouts and GNUGO 3.6. The result given in table 1 is that the best constant we have found is 0.3. From now on, this is the constant we use in all our experiments.

The error of a 200 games experiment associated to 40% of wins is 3.46%. In the following experiments we systematically use 200 games experiments since the associated error is between 2.45% for 86% of wins and 3.54% for 50% of wins.

The version of Golois used in the experiments plays 17,000 playouts per second, on the empty 9x9 board, running on a single thread of an Intel Xeon 2.33 GHz. It biases playouts using MOGO's patterns around the previous move and high urgency for capturing and saving stones in Atari. The result of a playout is either 0 if it is lost or 1 if it is won, therefore the mean of a node represents the probability of winning the game. Using the probability of winning the games is better than trying to maximize the score since it makes the program play safe moves when it is ahead, and more threatening moves when it is behind.

The following experiments consist in evolving individuals that represent the formula used to select the move to try in the Monte-Carlo search tree.

Table 2 gives the performance of the three basic heuristics against UCT for 1,000 playouts on 7x7 boards. The scores are the percentage of wins of a 200 games match. localAMAF, the heuristic used in the original RAVE algorithm scores 59.5% and beats UCT. Note that in this experiment we do not use RAVE at all, but replace the UCT formula with the evaluation given by the heuristic. The best result for an heuristic alone is for AMAFSignature which beats UCT 72.5% of the time.

Figure 1 gives the percentage of wins against the original UCT for the best individual of each successive population, playing 1,000 7x7 playouts at each move. Each population is composed of 128 individuals. An individual

Table 2 Performance of the three basic heuristics against UCT for 1,000 playouts.

localAMAF	AMAF	AMAFSignature
59.5%	63.5%	72.5%

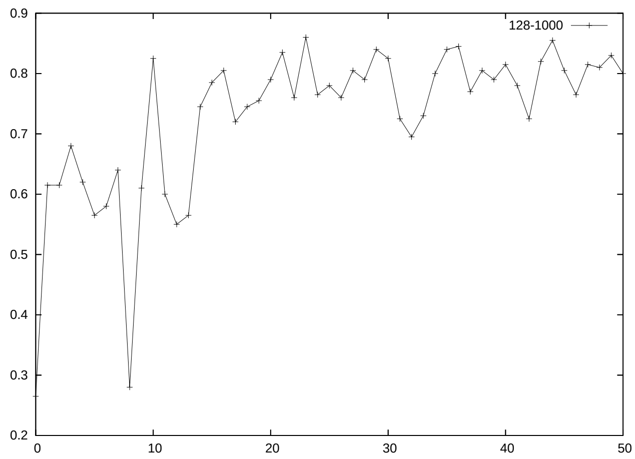

Fig. 1 Performance against UCT of the best individual of successive populations of 128 individuals playing 1,000 playouts, fitness is computed with 4 rounds.

is considered valid only if it contains at least two move related variables, it also has to contain the Playout variable. The percentages were computed completing 200 7x7 games between the best individual of each population and UCT. 100 games were played with black and 100 with white, the komi was set to 9.5 points for white as it is believed that 7x7 Go is a 9 points win for black. The best result is obtained for population 23 with 86% which is better than the best basic heuristic alone. After population 40, the best individual consistently scores better than 80%.

Figure 2 gives the percentage of wins against UCT for the best individual of each population of 256 individuals, playing 1,000 7x7 playouts at each move. The percentages were computed completing 200 7x7 games between the best individual of each population and UCT. The best individual of the first population beats UCT 55% of the time. Evolution improves to 75% in a few generations, which is close to the score of AMAFSignature alone. The experiment with 256 individuals gives slightly worse results than the experiment with 128 individuals, this is surprising and a possible explanation could be the inherent randomness in the evolution process, a more constructive explanation could be that with more individuals, the Swiss tournament needs more rounds to detect the best individuals.

The evolved heuristics beat UCT for a relatively small number of playouts, however when more playouts are allowed, UCT becomes better and beats the heuristics.

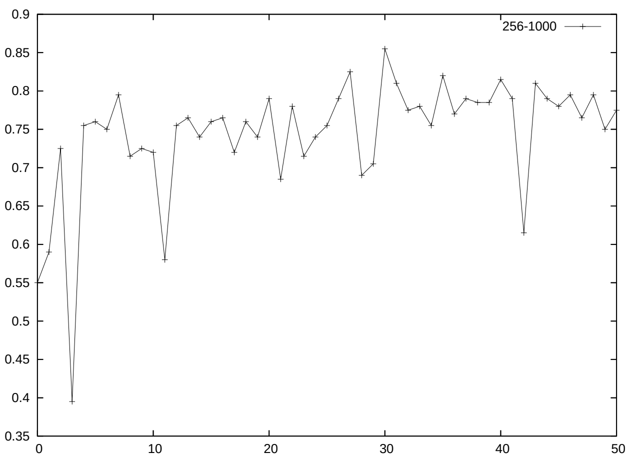

Fig. 2 Performance against UCT of the best individual of successive populations of 256 individuals playing 1,000 playouts, fitness is computed with 4 rounds.

Table 3 Performance of RAVE combined with the three heuristics against GNUGO 3.6 for 10,000 playouts.

RAVE(localAMAF)	RAVE(AMAF)	RAVE(AMAFSignature)
46.5%	50.5%	47.5%

In order to use the heuristics with more playouts, they were combined with the RAVE algorithm as described in [12]. The original implementation of RAVE adds $c \times \sqrt{log(allGamesAMAF)/gamesAMAF}$ to the localAMAF heuristic. Here we test the localAMAF heuristic alone without the added term. We expected that AMAF and AMAFSignature which perform better for one thousand playouts than localAMAF, would also perform better when combined with RAVE. Table 3 gives the results for the combination of RAVE with the three heuristics, playing 200 9x9 games against GNUGO 3.6. The k factor of RAVE was set to 1,000 and 10,000 playouts were performed at each move of each game. Surprisingly, the three heuristics give similar results even if AMAFSignature plays better alone for 1,000 playouts.

The best individual of the 29th population of the 128-1000 experiment scores 84% against UCT for 1,000 playouts. We tested its inclusion in the RAVE algorithm playing 5,000 playouts and let it play a 200 games match against the original RAVE. The evolved individual won 64% of its games, so genetic programming evolved a formula that beats the original RAVE. The code of this individual is : * (/ (AMAFSignature , / (sqrt allPlayoutsSignature , + (/ (sqrt localAMAF , + (/ (9 , * (/ (sqrt + (Playouts , 4) , + (* (/ (/ (sqrt * (/ (AMAFSignature , AMAF) , * (allPlayoutsSignature , / (+ (* (* (PlayoutsParent , 20) , MeanParent) , 5) , + (log Sigma , BestSigma))))) , * (/ (BestGames , AMAFSignature) , * (allPlayoutsSignature , / (+ (gamesAMAF , sqrt 6) , * (5 , / (localAMAF , localAMAF)))))) , 5) , / (gamesAMAF , * (20 , + (log 7 , + (/ (log

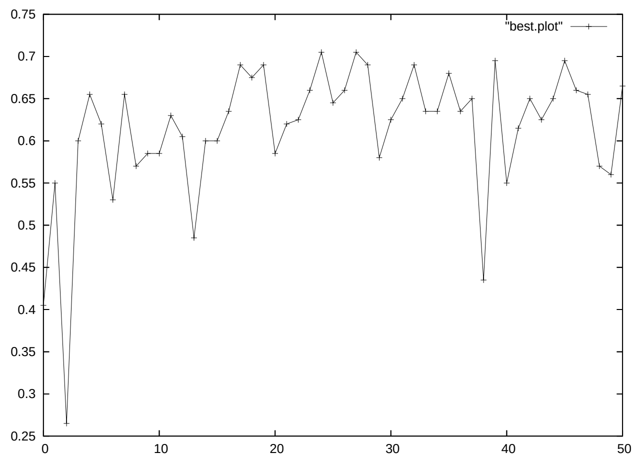

Fig. 3 Performance against RAVE of the best individual of successive populations of 128 individuals playing 5,000 playouts, fitness is computed with 4 rounds.

allPlayoutsSignature , BestSigma) , 6))))) , + (/ (sqrt 10 , + (/ (9 , * (/ (sqrt BestSumSquares , + (* (* (* (Playouts , MeanParent) , 5) , / (gamesAMAF , * (20 , + (log Playouts , / (/ (localAMAF , + (5 , + (log 4 , + (* (Playouts , / (* (localAMAF , 3) , 10)) , PlayoutsParent)))) , / (Playouts , + (AMAFSignature , 7)))))))) , 1)) , PlayoutsParent)) , 2)) , localAMAF))) , localAMAF)) , 20)) , localAMAF))) , allPlayoutsSignature).

The code of the individual playing the original RAVE algorithm has been entered by hand, it is : + (localAMAF , * (3 , / (sqrt / (log allGamesAMAF , gamesAMAF) , 10))).

However, some individuals that perform better for 1,000 playouts than the previous individual (scoring 86% for example) perform worse when included in the RAVE algorithm. Here again we observed that the level of the RAVE algorithm is not simply correlated with the playing strength of the heuristic alone.

Given this observation, we directly evolved a RAVE algorithm. Each individual of the population represents the heuristic to combine with UCT using RAVE. The k factor is set to 1,000. The number of playouts is set to 5,000. Valid expressions have to contain at least two move related components. The evolution of the best individual is given in figure 3.

RAVE(AMAFSignature) alone wins 58.5% of its games against RAVE for 5,000 playouts per move in a 200 games match.

The best individual of population 25 and the best individual of population 27 both score 70.5% against RAVE in a 200 games match as can be seen from figure 3. The code of the best individual of population 27 is much simpler than the code of the other individual, it is: + (+ (AMAFSignature , + (localAMAF , + (AMAFSignature , log MeanParent))) , + (/ (log 8 , 1) , + (/ (localAMAF , + (log 6 , gamesAMAF)) , + (BestMean , AMAF)))).

Table 4 Performance of UCT, RAVE and Genetic Programming against GNUGO 3.6 on 9x9 boards.

playouts	UCT	RAVE	Genetic Programming
10,000	40.0%	51.0%	65.5%
50,000	65.5%	73.5%	83.5%

We removed the useless components from the code of the best individual of population 27, and we tested its performance on 9x9 boards, with 10,000 playouts and 50,000 playouts against GNUGO 3.6. The komi was set to 7.5, and 200 games were played for each algorithm. The results are given in table 4. We also give the results for the UCT and the RAVE algorithms with the same number of playouts. We can see that the evolved individual outperforms both UCT and RAVE in both experiments. The code we used is: *2 * AMAFSignature + localAMAF + localAMAF / (child->gamesAMAF + log (6)) + AMAF.*

7 Conclusion

Genetic programming has evolved successful heuristics that efficiently develop Monte-Carlo trees in the game of Go. For 1,000 playouts, the best evolved heuristic beats the original UCT 86% of the time on 7x7 boards. Genetic programming also helps improve the RAVE algorithm which is a key component of the current best Go programs. The best individual found with genetic programming beats the original RAVE 70.5% of the time on 7x7 boards for 5,000 playouts.

When the best individual is included in Golois, it outperforms both UCT and RAVE on 9x9 boards against GNUGO 3.6. It wins 65.5% of its games against GNUGO 3.6 with 10,000 playouts, and 83.5% of its games with 50,000 playouts.

These results were obtained playing individuals of the same population against each other running a Swiss tournament. A filter was also used to constrain genetic programming to deal with individuals that satisfy certain conditions such as containing a leaf that is related to the move under consideration for example.

References

1. Angeline, P.J., Pollack, J.B.: Competitive environments evolve better solutions for complex tasks. In: ICGA, pp. 264–270 (1993)
2. Azaria, Y., Sipper, M.: GP-Gammon: Genetically programming backgammon players. Genetic Programming and Evolvable Machines **6**(3), 283–300 (2005)
3. Bouzy, B.: Associating domain-dependent knowledge and Monte Carlo approaches within a go program. Information Sciences **175**(4), 247–257 (2005)
4. Bouzy, B.: Associating shallow and selective global tree search with Monte Carlo for 9x9 go. In: N.S.N. H. Jaap Herik Yngvi Bjornsson (ed.) Computers and Games: 4th International Conference, CG 2004, Volume 3846 of LNCS, pp. 67–80. Springer-Verlag, Ramat-Gan, Israel (2006)

5. Bouzy, B., Cazenave, T.: Computer Go: An AI-Oriented Survey. Artificial Intelligence **132**(1), 39–103 (2001)
6. Bouzy, B., Helmstetter, B.: Monte Carlo Go developments. In: Advances in computer games 10, pp. 159–174. Kluwer (2003)
7. Bruegmann, B.: Monte Carlo Go. white paper (1993)
8. Cazenave, T.: Playing the right atari. ICGA Journal **30**(1), 35–42 (2007)
9. Cazenave, T., Helmstetter, B.: Combining tactical search and Monte-Carlo in the game of go. In: CIG'05, pp. 171–175. Colchester, UK (2005)
10. Coulom, R.: Efficient selectivity and back-up operators in monte-carlo tree search. In: Computers and Games 2006, Volume 4630 of LNCS, pp. 72–83. Springer, Torino, Italy (2006)
11. Coulom, R.: Computing elo ratings of move patterns in the game of go. In: CGW 2007, pp. 113–124. Amsterdam, The Netherlands (2007)
12. Gelly, S., Silver, D.: Combining online and offline knowledge in UCT. In: ICML, pp. 273–280 (2007)
13. Gelly, S., Wang, Y., Munos, R., Teytaud, O.: Modification of UCT with patterns in monte-carlo go. Tech. Rep. 6062, INRIA (2006)
14. Hauptman, A., Sipper, M.: Emergence of complex strategies in the evolution of chess endgame players. Advances in Complex Systems **10**(1), 35–59 (2007)
15. Kocsis, L., Szepesvàri, C.: Bandit based monte-carlo planning. In: ECML, *Lecture Notes in Computer Science*, vol. 4212, pp. 282–293. Springer (2006)
16. Koza, J.: Genetic Programming. MIT Press, Cambridge, MA (1992)
17. Rosin, C.D., Belew, R.K.: New methods for competitive coevolution. Evolutionary Computation **5**(1), 1–29 (1997)
18. Sharabi, S., Sipper, M.: GP-Sumo: Using genetic programming to evolve sumobots. Genetic Programming and Evolvable Machines **7**(3), 211–230 (2006)
19. Sipper, M., Azaria, Y., Hauptman, A., Shichel, Y.: Designing an evolutionary strategizing machine for game playing and beyond. IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews **37**(4), 583–593 (2007)