Applying coevolutionary genetic algorithms to computer Go

Lewis & Clark College, Portland OR
Gregory Aaronson ‘16, Andrea Dean ‘17, Dr. Peter Drake

Introduction

Go is an ancient Chinese board game that, despite its simple rules, is difficult to program computers to play well; the top Go programs still cannot beat the top human players. Current state of the art programs rely on Monte Carlo Tree Search: constructing search trees based on many simulated games, within which moves are chosen by a biased random process [1, 2]. A weakness of Monte Carlo Tree Search is that it only searches globally, whereas human players also search locally, breaking the game into semi-independent subproblems [2]. We propose an alternate approach: generate sets of replies, allowing a player to respond to the most recent moves regardless of the global context. We learn these sets of replies using a genetic algorithm, a search technique inspired by biological evolution [3].

Genetic algorithms

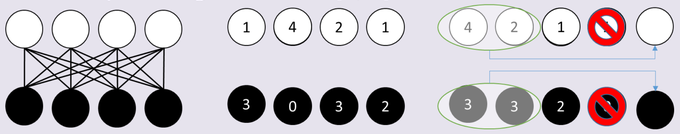
In a genetic algorithm, each individual represents one possible solution to the problem -- in our case, a set of replies. The initial population is composed of random individuals. Individuals are evaluated for fitness, a numerical rating of their strength. The more fit individuals are randomly recombined to produce a new generation, replacing the old population. Over time, the whole population becomes more fit, mimicking evolution.

Steady-state Coevolution

Our genetic algorithm is of the steady-state variant: instead of discrete generations, we replace individuals one at a time. This approach allows us to stop the evolution at any time and examine the state of the population. While plain genetic algorithms perform evolution given an externally-provided fitness measure, we perform coevolution, with two opposing populations. The fitness of each individual depends on its ability to compete with members of the other population.

Tournaments

We measure fitness by holding tournaments [2, 3]. *N* individuals from each population play games of Go against *N* individuals from the other population, starting from the current state of the board. The number of games each individual wins is recorded. The lowest scoring individual in each population is replaced with a combination of the two top-scoring individuals [2, 4].



Design

Representation

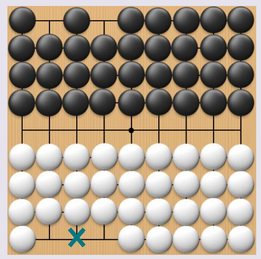
Our individual's "genetic information" comprises many three-move sequences; whenever the first two moves in a sequence are played, the individual tries to play the third move. Completely random move sequences are extremely unlikely to ever appear in games. To address this, the sequences in our initial population came from random simulated games played from the current board state. If an individual has multiple replies to the same two-move sequence, only the last one counts.

Voting among the population

After a period of evolution, each individual in the appropriate population is polled and “votes” for a move. The move with the most votes from the individuals is selected and played.

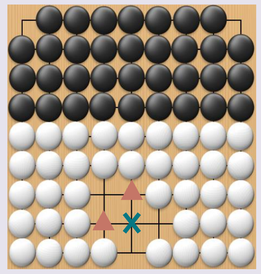
Experimental conditions

The following conditions were consistent for the experiments: Each individual was made of 100 move sequences and every population had 2000 individuals. The tournaments had 4 individuals from each population competing. The experiment was run on 32 threads using Google’s Cloud Computation. Detailed experiment code is in commit 7083486e8ac1e02637456dab62463735354b39c1 on the genetic branch at https://github.com/Orego/Orego.



Simple problem

We first set up a simple problem, in which a single move (‘X’) from black can ensure its victory. Our genetic algorithm consistently was able to solve this problem in a fraction of a second of evolution.

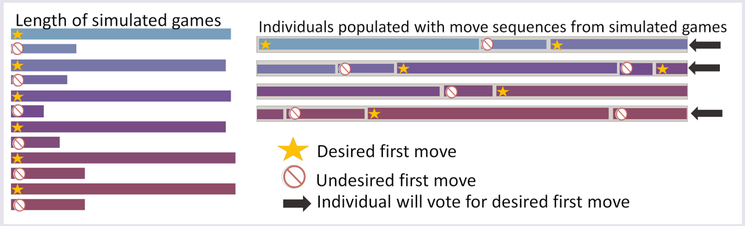


Complex problem

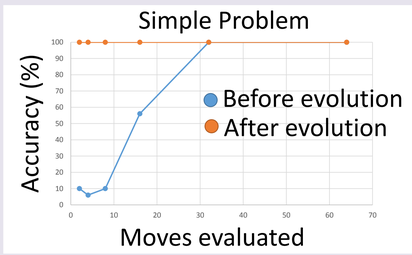
Our next problem was more complex: there is one desired move (‘X’), but for black to win it must also play at least one follow-up move (‘Δ’). Initially the genetic algorithms performed fairly well, but we noticed that the accuracy of our results decreased as evolution continued. Oddly, our program was most likely to find the correct answer with no evolution at all! We began investigating this conundrum.

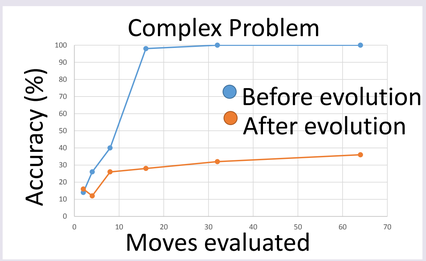
Investigations and solutions

We found a problem in generating our move sequences from random simulated games. More turns in a simulation make it more likely that the first move of that simulation will be the last reply to the particular preceding moves. This placement means that the move will be selected as an individual’s vote when it is polled. In our problems, the desirable moves lead to much longer games, so they are unfairly favored. To combat this problem, we have limited the number of moves per simulated game, taking only the first *k* moves for the individuals before starting the next simulation.



Final results





As shown in the graphs, as more moves are evaluated per simulated game, the higher accuracy the program has before any evolution has occurred. By limiting the number of moves evaluated, we can ensure that the evolution is improving the accuracy. The simple problem’s evolution is accurate 100% of the time, while the complex problem’s evolution still needs more investigation. Our problem of unfairly favoring the desirable move has been resolved.

Future Work

Our next steps are to investigate why follow-up moves present such a problem to our existing structure. Once we are confident that coevolutionary Monte Carlo is able to consistently solve complex problems, it can be incorporated into Orego, Dr. Drake’s Go playing program, replacing the tree search.

References

[1] C. Browne *et al.*, “A survey Of Monte Carlo Tree Search Methods,” in IEEE transactions on computational intelligence and AI in games, vol. 4, no. 1, pp. 1-49, March, 2012.
[2] P. Drake and Y.-P. Chen, “Coevolving partial strategies for the game of Go,” in *International Conference on Genetic and Evolutionary Methods*, Las Vegas, NV, 2008, pp. 312-318.
[3] A.E. Eiben and J.E. Smith, *Introduction to evolutionary computation*, Berlin, Germany: Springer, 2003.
[4] T. Miconi and A. Channon, “The N-Strikes-Out algorithm: A steady-state algorithm for coevolution,” in *Congress on Evolutionary Computation,* Vancouver, BC, 2006, pp. 1639-1946.
Image from <http://i.nextmedia.com.au/News/flickr-547944930-hd.jpg>

Acknowledgments

This work was funded by the John S. Rogers Science Research Program. Go problem images were created using Sen:te Goban [sic].