Automatically Learning Heuristic Patterns for Computer Go

Emily Amundson (Trinity University, '13), Professor Yung-Pin Chen (Lewis & Clark College), Professor Peter Drake (Lewis & Clark College),
Kyle Evitts (Linfield College, '14), Jason Galbraith (Instructor at Sunset High School),
Jet'aime Mullins (Lewis & Clark College, '13), Sam Stewart (Lewis and Clark College, '15), Nick Sylvester (Lewis and Clark College, '13),
Ryan Takahashi (Harvey Mudd College, '13), Vincent Zhuang (Westview High School, '14)

Abstract

Go is an ancient two-player board game that originated in China over three thousand years ago [1]. It plays an important role in artificial intelligence research, being one of the few remaining games in which human players still dominate the top computer programs. Top Go programs all use Monte-Carlo sampling [2]. Many simulated games are played, after which the move most often leading to a win is selected. The moves within the simulated games are semi-random but biased by both the results of previous simulations and heuristic knowledge.

This paper focuses on a pattern-recognition heuristic. Certain local configurations of stones are recognized as good (or bad) by human Go players. We extracted patterns from a collection of games played by human experts. Playing moves suggested by our automatically-learned good patterns performed as well as expert hand crafted patterns. Avoiding moves suggested by bad patterns does not strengthen performance.

Heuristics in Monte Carlo Go

Early moves in each Monte Carlo simulation are biased by the results of previous simulations using a Monte Carlo search tree [2]. Beyond the search tree (and to a degree within the search tree) moves are biased by heuristic knowledge. Orego uses three heuristics: escaping when threatened, matching patterns, and capturing when possible. To select a simulated move, Orego uses the following algorithm:

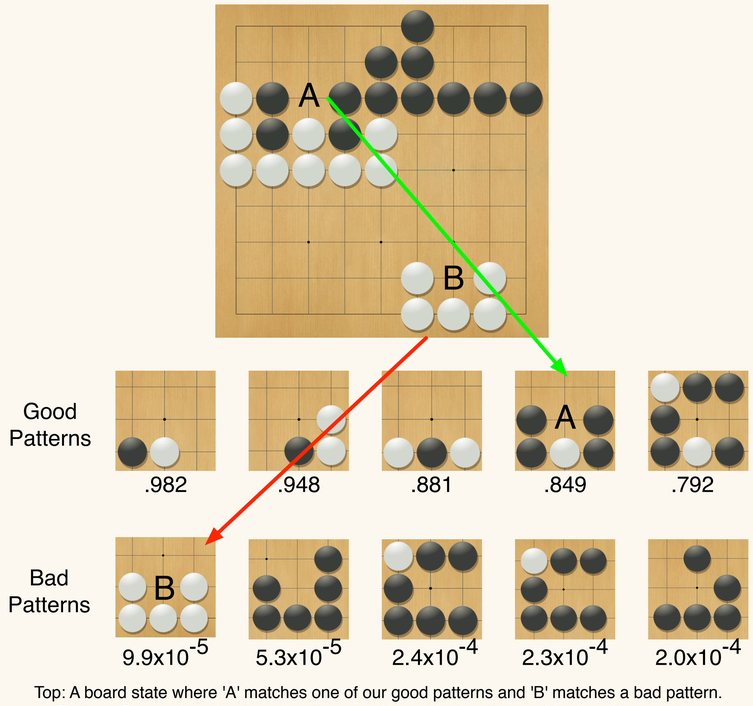
1. If any heuristic classifies a move as good, play it.
2. If there are no good moves, choose a random move that is not classified as bad by any heuristic.
3. If there are no non-bad moves, randomly choose a bad move to play.

Patterns

As in the MoGo program [3], our patterns are 3×3 regions surrounding a potential move (see figure). In searching for *good* moves, we examine only the eight points around the most recent move. Searching the entire board would be too time-consuming. We only have to reject *bad* moves one at a time (as they become candidates in step 2 above), so we can effectively search the entire board for matches.

We extracted patterns from 86,343 expert games played on the KGS go server [4]. Patterns which are reflections, rotations, or color inversions of each other were conflated. We then analyzed this data looking for good and bad patterns. We stored, for each pattern, the ratio between the number of games in which it *was played* to the number in which it *appeared*. A pattern with a ratio close to 1.0 is played in almost every game in which it appears, so it is presumably good; a pattern with a ratio close to 0.0 is presumably bad.

For consistency with the way the patterns are used, we made two lists of patterns. The good patterns are those with the highest ratios considering only plays next to the last move. The bad patterns are those with the lowest ratios considering the entire board.



Top: A board state where 'A' matches one of our good patterns and 'B' matches a bad pattern.
Bottom: Some of our best good patterns and worst bad patterns with played/seen ratio.
Black to play in the center for all patterns.

Results

To evaluate the advantage of using these learned patterns we ran an experiment. Orego 7.12 was run with a single thread and four seconds per move. Orego played against GNU Go [5] using various numbers of the automatically accumulated good and bad patterns. 480 games were played in each condition. The win rates are shown in the table below.

# of Worst Bad Patterns Used \ # of Best Good Patterns Used	0	50	100	150	200	250	300	350	400	450	500
0	4.4%	14.2%	14.2%	19.2%	25.0%	25.2%	31.5%	34.2%	**35.2%**	32.7%	31.5%
50	4.6%	13.1%	14.6%	20.6%	23.8%	23.8%	34.0%	28.3%	30.2%	30.0%	27.5%
100	.31%	11.5%	17.1%	21.0%	22.3%	25.6%	28.1%	30.2%	29.4%	30.4%	29.6%
150	.42%	11.5%	18.5%	15.2%	21.0%	26.5%	24.2%	24.2%	34.0%	23.8%	27.5%
200	.52%	10.4%	14.8%	20.8%	19.4%	25.0%	26.7%	28.3%	32.3%	22.5%	30.8%

Conclusions and Future Work

Incorporating learned good patterns into Monte-Carlo sampling proved to be beneficial. Avoiding bad patterns did not provide a similar benefit and seemed to hurt performance as more patterns were added. We suspect that this is because making a bad move is not that much worse (in terms of winning the game) than an irrelevant move; in contrast, playing an irrelevant move instead of a good, urgent move can be extremely costly.

Our learned patterns are roughly comparable to MoGo's hand-coded patterns [3]. Orego's best win rate with automatically-learned patterns (35.2%) is the same as when using the MoGo patterns. This automation may be beneficially applied to other problems where experts are rare or expensive (e.g., medical diagnosis).

We would like to continue tuning the number of patterns we use. In extracting patterns from recorded games, we may also count the number of *times* a pattern was played or seen within a game, rather than the number of *games*. A pattern that was not immediately played, but was played eventually, would therefore receive a lower ratio.

Patterns larger than 3×3 would incorporate more surrounding context, giving a more accurate assessment of the value of each move. Storing and evaluating such patterns would, however, require more memory and computation time.

As an alternative to increasing pattern size, the patterns could be enriched with higher-order Go features such as chain length, eyes, or liberty count. Again, this provides more context at the expense of space and time.

More ambitiously, machine learning may be able to induce a move quality *function* rather than storing a set of patterns.

References

[1] Baker, Karl. (2008, February). “The Way to Go.” Retrieved August 16, 2012 from http://www.usgo.org/way-go
[2] Browne, Cameron, et al. (2012). A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games* 4(1), 1-49.
[3] Gelly, Sylvain, Yizao Wang, Rémi Munos, and Olivier Teytaud. (2006). Modification of UCT with Patterns in Monte-Carlo Go. *INRIA* 1-21.
[4] KGS - http://www.u-go.net/gamerecords/
[5] Gnu Go, http://www.gnu.org/software/gnugo/

Acknowledgements

This research was funded by the John S. Rogers Science Research Program at Lewis & Clark College and the Willamette Valley REU-RET (NSF Grant #1157105).