Decision Trees for Local Search in Monte Carlo Go

Bennett Lohre '12, Samuel Dodson '13, Nick Sylvester '13 and Professor Peter Drake
Department of Mathematical Sciences, Lewis and Clark College
{blohre, sdodson, nsylvester, drake}@lclark.edu

Abstract

Writing programs to play the classical Asian game of Go is an open problem in artificial intelligence. Current Go-playing programs use Monte-Carlo sampling to choose moves. The most common variation, global Monte-Carlo tree search, must repeatedly discover solutions to local problems. We introduce a new technique that alleviates this redundancy by storing Monte-Carlo samples in a group of decision trees. These decision trees are able to outperform existing techniques when simultaneously confronted with multiple local search problems.

Computer Go

Go is a board game that originated in China several thousand years ago [2]. Writing a program that plays Go well is difficult because of the large search space and the absence of powerful evaluation functions. Today the best human players are still able to beat the best computer players [4]. Techniques used to play Go can be applied to other problems involving very large search spaces.

Monte-Carlo Techniques

Monte-Carlo techniques use random sampling to solve complex problems [5]. In Monte-Carlo Go, this random sampling consists of playing a series of simulated games (playouts). The moves within each playout are chosen according to some policy. After each playout, the winner is determined; the policy is adjusted to encourage the winner's moves and discourage the loser's moves. Each playout is thus affected by the results of previous playouts. When time runs out, the program plays the move that fared best, e.g., the move that started the most playouts.

Decision Trees

Decision trees are a popular technique for supervised learning. A set of training data points are used to create the tree. Each data point has values for a number of attributes and a classification. The goal of a decision tree is to predict the classification of a point given its attribute values. Each internal node in the tree separates the data according to one attribute. A recursive algorithm utilizes information theory to determine on which attribute to split the data [5].

Local Search

Given a board position in which there are several local problems, global search must do considerable redundant work (Figure 1); good responses have to be rediscovered in several different contexts. Our forest of decision trees keeps track of how good each move is in response to immediately preceding moves, regardless of the global context. This makes better use of the information gained from each playout.

For each point on the board we maintain a decision tree — originally a single node. The trees grow as more playouts are completed. A data point is added to the appropriate tree for each move made during a playout. A data point remembers its preceding move and whether the playout was a win or loss. When a tree node contains enough data it is split based on the most useful preceding move. The node is given two children: one corresponding to the data points where the specified preceding move was made and the other containing the remaining data.

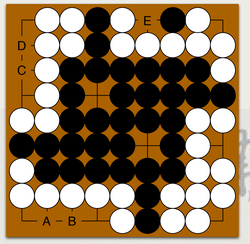
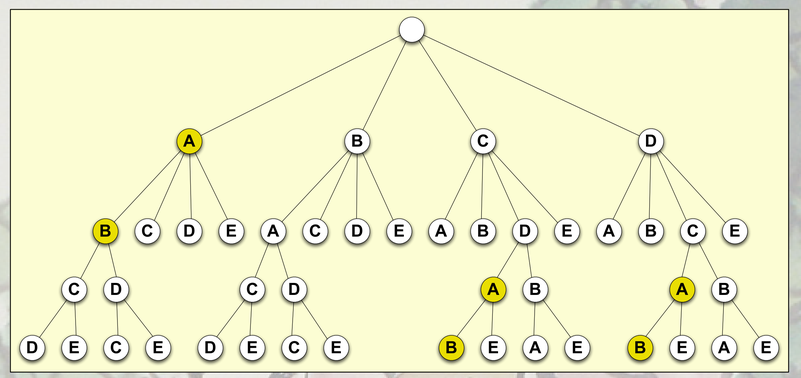
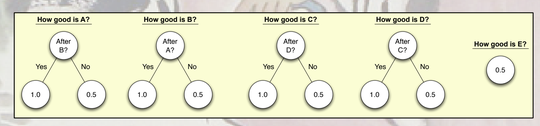


Figure 1: On the board at left, the proper response to black A is white B, and vice versa. The same can be said for moves C and D. White E is unimportant. In a global search (middle), the correct response to A must be rediscovered three times. The decision trees (bottom) only have to discover this information once.





Experiments

For each experiment, we created a board state with several independent life and death problems [3]. In these problems there are several black moves to which white has only one winning response. In order to win, a program must respond correctly to each forcing move, regardless of the order in which they are played. We ran a fixed number of playouts using version 7.10 of Orego, our existing Go program. These playouts were used to build a group of decision trees. Wins and losses were recorded for both the player and our decision trees.

In each condition we ran the problem 200 times with 10000 playouts. Experiment 1 used a problem with black playing a forcing move in each of four different areas (Figure 2). Playouts were completed either randomly or using a biased policy favoring more realistic moves. The three life and death problems in Experiment 2 are more difficult.

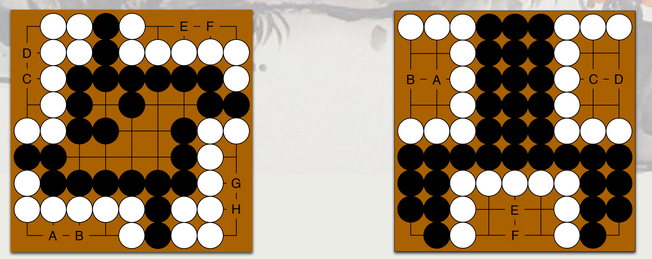


Figure 2: Board setups for Experiments 1 (left) and 2 (right). White must respond to black A with B, C with D and so on.

Results

Our decision trees outperformed Orego with the random policy condition in Experiment 1 with a win rate of 100% versus 59.5% (Figure 3). This is statistically significant ($p < 0.05$, two-tailed z-test). In the second part of Experiment 1 we won 95.5% of the time while Orego won 96% of the time (not significant). The second experiment had much lower win rates. Orego did not win any its games in Experiment 2 with the random playouts or the more realistic playouts. The decision trees performed slightly better with the random playouts winning 3 of the 200 games (not significant). With the more realistic playouts however, the decision tree won 11.5% of its games; this was significant.

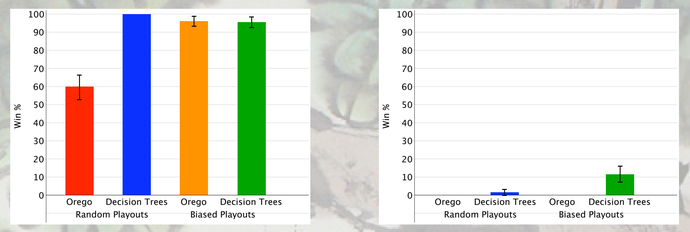


Figure 3: Results from Experiments 1 (left) and 2 (right). Error bars indicate a 95% confidence interval.

Conclusions and Future Work

We have introduced a new technique for computer Go using decision trees. When multiple local problems must be solved, our technique outperforms existing methods.

We include two caveats. First, pilot experiments with more difficult problems produced inconclusive results. Second, computation time was not taken into account, as we have not yet optimized our implementation; it is conceivable that our technique make more of each playout, but existing techniques make better use of time.

Because this new technique works well in these experiments we would like to make it a fully functioning player instead of relying on playouts from the existing Orego player. Possible enhancements include optimization, examining more preceding moves [1] and board information, and adding domain-specific prior knowledge.

References

[1] H. Baier and P. Drake. "The Power of Forgetting: Improving the Last-Good-Reply Policy in Monte-Carlo Go," in *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 2, no. 4, pp. 303-309, 2010.

[2] K. Baker. "The Way to Go." Internet: http://www.usgo.org/usa/waytogo/index.asp, accessed June 5, 2011.

[3] J. Davies. *Life and Death*. Tokyo, Japan: Kiseido Publishing Company, 1996.

[4] M. Müller. "Computer Go," in *Artificial Intelligence*, vol. 134, no. 1-2, pp. 145-179, 2002.

[5] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Saddle River, NJ: Pearson, 2003.

Acknowledgements

This research was funded by the John S. Rogers Science Research Program at Lewis & Clark College and the James F. and Marion L. Miller Foundation.

Background image: Liang Dao, 2008, "The Council of Chess" [sic]. http://www.orientaloutpost.com/proddetail.php?prod=ld-chess