Go

by Joey K Black

1.0 Introduction

Go is an ancient Chinese board game. The skill required to play is comparable to the skill required to play Chess. For computers however, the challenge of Go is much greater than that of Chess. The reason being that the Go board is 19x19 while the Chess board is only 8x8. These extra spaces means more moves. When using the Minmax algorithm (the standard AI algorithm for Chess) the number of moves that must be examined increases exponentially the further ahead you look. This exponential growth is a big challenge for Chess, but with the added number of moves per turn for Go, the Minimax algorithm becomes infeasible.

Without the Minimax algorithm, the field of Go AI is back to square one. In some ways this is a good thing as the Minimax algorithm is sometimes considered a crutch in computer Chess. Without our crutch we must persevere into uncharted waters of AI.

2.0 Monte Carlo

No, we are not going on vacation. Monte Carlo (or MC) is one of the few effective algorithms for computer Go that does not require the programmer to implement expert knowledge in their agent. You can add all the tricks you know into a program, but without a general purpose decision process, your AI will be handicapped when it finds itself in new situations.

Monte Carlo is a general purpose algorithm that approximates an optimal solution to a problem that would be otherwise impossible to find through brute force search.

> Example 1:
>
> Suppose you want to find an approximation for PI and all you have is some dirt and rice. (Think of this like some sort of Math Survivor reality show.) First draw a square in the dirt and inscribe a circle in it. Next scatter the rice evenly over the square. Now count the grains in the circle and outside the circle and the ratio will be an approximation of pi/4.

Applying this algorithm to Go is fairly simple. For every move you are considering, play out a number of random games starting with that move. The ratio of wins to losses is the relative MC value of that move in relation to the other potential moves. The move with the highest value is the move that has the best chance of resulting in a win regardless of the moves that follow.

This algorithm has some limitations especially when it comes to finding sequences of moves that will result in a win only if they are done in the correct order. Despite its limitations, the MC algorithm is a impressively strong starting point for a computer Go program.

3.0 Monte Carlo Implementation and Results

The following is an outline of my implementation of a MC Go agent. I'll briefly discuss the board structure and MC algorithm I used.

Board Structure:

Your board structure is a critical design element as it will have a major impact on speed. My design is fairly simple. The board state is captured by two 361 bit bitboards. One for white and one for black. For my bitboard I used 6 longs adding in functions to perform and, or and bitshift operations as if it were a single 361 bit board. This gave me a very compact and efficient board representation.

This design is a very straightforward implementation to understand, but the challenge comes when you try to update the board after playing a piece. Most implementations keep track of groups of stones and their liberties to make the capture and removal process simpler. However with just the raw bit map positions, the capture and removal process had to be done purely with bitwise operations.

```
RemoveDeadStonesFromDefender(BitBoard attacker, BitBoard defender):

     BitBoard allDeffendersLiberties = shift defender up, down, left
and right and remove all occupied bits;

     BitBoard defenderStonesAdjacentLiberties = shift
allDeffendersLiberties up, down, left and right and AND with
defender;

     BitBoard liveStones = stonesAdjacentLiberties;

     BitBoard oldLiveStones;

     while ( oldLiveStones !=  liveStones):

           oldLiveStones = liveStones;

           liveStones = (shift liveStones up, down, left and right)
AND deffender OR  liveStones;

     return liveStones;
```

This method uses efficient bitwise operations to remove all the dead stones on the board in a single series of operations. No need for identifying groups and counting liberties.

This also allowed me to do another little trick to speed up random game play out. Instead of doing individual random moves, the program scatters multiple stones on the board for each player and then removes captured stones. This is not quite as accurate as playing out individual moves, but it seems to be a good approximation (and an approximation is all we really want with Monte Carlo).

Monte Carlo Implementation

My implementation of the Monte Carlo algorithm does not use any knowledge of Go technique. Every valid move is analyzed and the move with the highest value is played. Each move is analyzed on its own thread to improve efficiency. The pseudo code to analyze a move is as follows:

```
EvaluateMove (BoardState boardState, Move move):
     do move on board;
     int totalWins = 0;
     for each game to be played:
          copy board;
```

```
        approximate outcome of random game (see above);
        if player won:
            totalWins++;
    return totalWins;
```

And that is it! This implementation can approximate about 1000 games per move tested in about 1 second. After 50 moves against a random agent, it is consistently ahead by about 20 points. When set to analyze 10,000 games per move tested, it can select a move in about 12 seconds. It also makes a challenging opponent for me at this level (however I am not very skilled at Go).

4.0 UCT and Other Optimizations

Upper Confidence bounds applied to Trees (UCT) is a slightly more advanced variation of MC that produces even better results. MC will find good moves, but it does not always converge on the best move. UCT does a deeper search by examining promising moves more closely than bad moves. The first time a new node is visited, its value is determined by playing a random game on it. After that, its value will be determined by expanding its child nodes and selecting the best child. UCT also uses some clever formulas for selecting nodes to visit. For more information on UCT, see the links section at the end of this article.

MC and UCT are an important part of a computer Go agent, but to really take it to the next level, expert knowledge is needed. Competitive programs use all kinds of tricks and preset strategies to select potential moves.

5.0 Conclusion

Computer Go has come a long way, but the top programs are still not quite at the level of the top human players, so there is still plenty of room for improvement. The large number of move options makes computer Go a significant challenge, but as programs and hardware advance, the gap between human and machine will continue to close. All your stones are belong to Skynet.

6.0 Links

Monte Carlo - http://senseis.xmp.net/?MonteCarlo
http://en.wikipedia.org/wiki/Monte_Carlo_method
UCT - http://senseis.xmp.net/?UCT