**Go Representation and AI**

Jingpeng Wu

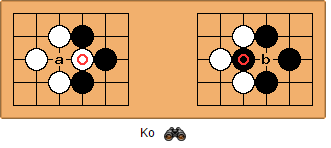
**Abstract**

The motivation is to find efficient ways to represent a game of Go and to create an AI in order to become better at problem solving and to learn about the different approaches in AI. The approach is to focus on the group structure (collective adjacent pieces) as an object since it has many useful properties and is more efficient than keeping track of individual pieces. The approach for an AI was initially planned to be minimax but even though old and new approaches were combined in an attempt to make a minimax implementation feasible, I eventually went with the Monte Carlo search tree method instead since it is a much better approach (see analysis).

**Introduction to the rules of Go**

Two players take alternating turns placing 1 piece of their own color on an intersection of a 19x19 board per turn. The ultimate goal of the game is to gain more *territory* than the other player. *Territory* is an intersection that is completely surrounded by the edge of the board and / or solid walls of your pieces (diagonals never matter).

A *liberty* of a piece is an empty intersection that is adjacent to the piece. Connecting two pieces of the same color forms them into a “group” that shares liberties and *life and death.* For example, if one lone black piece has four liberties, two touching black pieces will have six liberties total. Enemy pieces do not share liberties and reduce them instead. When a piece or a group is completely surrounded by enemy pieces, it will have no liberties and will be *captured* and removed from the board. Players are not allowed to play a move that will lead to self-capture. If a group surrounds two empty nonadjacent liberties (called *eyes*), then it can never be captured because an opposing player can never simultaneously fill both liberties since suicidal moves are not allowed. Scoring begins when both players pass their turn which usually happens when the board is completely filled and no groups can be captured.

A last interesting mechanism in Go is called *ko*. A *ko* occurs when one player captures a piece from another and in the next turn, the other player can capture the newly placed piece back which would lead to an infinitely repeating state. To prevent this, the rules state that you are not allowed to place a piece where your piece was just captured for 1 turn.

**Introduction to Go strategy**

I will briefly talk about go strategy in order to demonstrate the difficulties in creating a strong AI. An average 200 move Go game on a 19x19 board has ~3^500 unique games, making brute force searches and even representing the game tree very difficult. In addition, Go requires a high degree of abstract pattern matching on both the micro and macro level which makes humans much better players than machines. When humans play Go, we use a wide variety of heuristics that are not easily and efficiently quantifiable in code. Strong Go AIs that are able to play on par with strong human players usually use a combination of methods and algorithms and an extensive knowledge base.

**Implementation of the Go Engine**

I will go over the major parts of the engine implementation; the details are clear enough to use as pseudocode without ambiguity.

I created an object called a **Group** that efficiently keeps track of the pieces and liberties of the group (java hash sets of points) and the x, y, and display picture of the group’s center. Intuitively, the board is represented as a two dimensional array but under my approach the array holds groups instead of intersections/points.

Advantages of new implementation vs. traditional:

* When allied pieces are adjacent, they naturally form groups and never separate. Thus it is more computationally efficient to treat the pieces as groups from the start.
* Less redundancy since *n* adjacent pieces forming a group can be represented in the 2d array as a single group centered at some location (only matters for displaying) and *n – 1* references to the group.

Using the Group object via the BoardManager class, we can efficiently perform all game operations with some logical reasoning.

Steps of piece placement:

1. Make sure piece is legal.

* Unoccupied space that’s within the bounds and not labeled with a “K” (*ko*).
* The move must not be a suicidal move so all adjacent groups must have two or more liberties. (If an allied group only has one liberty and this piece takes it, you have a dead group)
* If this move captures an enemy piece, then this takes precedence over suicides. (Check if any adjacent enemies only have 1 liberty; logically, that liberty can only be where you plan on moving)

1. Find the liberties of this one-piece group treated individually and remove this point from the list of liberties of all adjacent groups.
2. Union and capture check depending on the color of adjacent groups. Upon capture, the enemy group of N pieces is removed from the board and replaced with N empty groups. In addition, the liberties are also refunded and added to any group touching the edge of the captured group.

We save time and coding by treating the piece as an individual group in step 2) before we take any action since this will cover the common liberty calculations for both allied and enemy groups.

*Ko* is calculated by checking if any adjacent enemy groups only have one piece and one liberty. Also, after the capture the piece you just placed must only have one liberty. Then the contested square is marked as *ko* and will be locked for one turn.

**Extra Engine Features**

You can pass in a queue of legal points and have the BoardManager play all of those automatically. It is useful for easily reproducing positions for testing and debugging.

**Implementation of the AI**

This AI is more of an exercise in implementation rather than a Strong AI, due both to time and resource limitations. A large amount of collaborate work would be needed to make a go AI that is competitive with the average player.

Initial Approach:

The initial plan as stated in the proposal was to implement the minimax algorithm since it was covered in class but producing the game tree quickly became infeasible. A game tree with a depth of only three has roughly 361 \* 360 \* 359 = 50 million states.

Work done towards a minimax solution:

* The use of alpha-beta pruning will reduce the number of board states that need to be evaluated.
* Local minimaxing

The creation of “**local minimaxing**”, where a square of variable length centered on the last human move is saved as the state and the tree is built for this. The idea for this was that the game tree would be drastically reduced while not severely hurting performance since the moves of two players are relatively close to each other for the majority of the game. A new reduced game tree will have to be created for every turn but the combined size is still magnitudes smaller than the full game tree.

The AI includes a method to create a partition of variable size, manually clone the reduced board, and to display it. However, these are not used in the final project due to the abandoning of the minimax algorithm.

Obstacles towards the work done:

* Alpha-beta pruning only helps with heuristic evaluation and not with building the game tree. Also, knowing the state easily gives us a heuristic value from the group properties, ex. the number of remaining pieces and liberties, etc.
* Local minimaxing is still inefficient and has tricky edge cases. (Even on a reduced 5x5 area, a depth 3 tree will roughly have 24 \* 23 \* 22 = 12 thousand states)

**The new solution, Monte Carlo**

Monte Carlo is a vastly superior algorithm than minimax for Go for the following reasons:

* We do not need a game tree which saves a tremendous amount of time and memory.
* With an efficient game engine, simulations can be run extremely quickly. 5000 x 100 move games can be played in roughly one second on my laptop.
* Only the Monte Carlo search tree needs to be implemented which is just the moves you made during a particular run.

Monte Carlo Implementation:

Keep one move tree and play an arbitrary amount of randomized games and record all of your moves with root. Nodes are created with a move and the x and y coordinates. Each node has a depth, a hash set of child nodes and the value. During back/forward propagation, relay the value back to the root/child and increment/decrement the depth as needed. If a node with the same x, y coordinates and the same depth exists, discard your new node and add its value to the existing node. Otherwise, link your new node to the tree.

The value is calculated from a basic heuristic involving the summation of the number of pieces and liberties a certain color has (times negative one for white, the AI is always black and tries to maximize this value). A simple heuristic was chosen for demonstration purposes since a good heuristic is hard to quantify and does not contribute to the overall Monte Carlo implementation. See experiments for more information.

**Experiments**

Testing was done frequently at each step of implementation and was expedited by using the extra engine features to automatically recreate the relevant positions that I was testing. In addition, I will prepare a list of states to show off in the demo (only one prepared state is included as an example). There is extensive logging that can be used to try and find a problem; anything that needs to be printed in any format can be done through existing methods. In addition, 5000 x 100 Monte Carlo games were simulated in 1 second which shows that the Go engine implementation was a success.

Through visual inspection, the quality of the Go AI appears to be making smarter moves as the number of simulations increase. Characteristics include closer clustering and moves that are more closely linked together, but once again this depends greatly on the heuristic which is hard to choose and not a reflection of the implementation. In addition, it is difficult to quantify the quality of the AI since we are only using one algorithm and comparing them against existing long-standing Go AI’s that implement many different algorithms would be unfair.

In addition, even in professional Go engines, the scoring at the end is done by humans (they select the borders and the engine just adds). This is due to the extreme complexity of determining life and death automatically especially in nested cases. Thus, for my implementation, the player will also have to manually do scoring.

**Conclusion**

From the Go engine, I believe I have increased my programming skills especially for game engines in general since I am now better at efficient state representation and modularizing code. From the AI, I have learned that it is very difficult to create a good AI for go but I feel more confident with creating AIs in general now that I have created a working one for a hard game. Although this project was fun and successful overall, I wasted too much time on the attempted minimax / local minimax implementations and might have better used that time to further enhance the Monte Carlo algorithm.

**Sources**

http://en.wikipedia.org/wiki/Negamax

http://en.wikipedia.org/wiki/Computer\_Go

http://en.wikipedia.org/wiki/Rules\_of\_Go

<http://en.wikipedia.org/wiki/Go_and_mathematics#Game_tree_complexity>