Math 480 : JinGo Engine

Jingpeng Wu, Daae Oh, Sean Kim

June 9th, 2013

Contents

1 Introduction

The original project idea was to create a Go engine and a Go AI but since each of those parts require signifigant work, we have decided to just make the engine. The biggest challenges in writing the engine was deciding how to impllement the rules in code. Often, there would be a simple and cost - efficient solution to implementing invidivual parts of the game, ie liberties but in terms of the big picture it would often be a bad choice due to compatibility issues.

Thus, instead of looking for locally optimal implementation solutions, we had to look globablly and eventually came up with the "power group" structure to handle pieces and groups and a "board" structure to handle everything else. Throughout this document, it will be assumed that the reader has a basic familiarity with the game of Go and Go terminology.

2 Rules of Go and how they are Implemented

2.1 Board State

The 361 points of a standard 19x19 board is represented as a single dimensional array instead of a more intuitive two-dimensional array because it allows us to access each piece through its unique id (0 - 360) which is easier than passing two numbers. For any point, with id = id, we can find the piece to the left and right by adding and subtracting 1. To find up and down, we can add subtract by the board size. To convert from x and y to the unique id or vice versa:

```
id = 19 * x + y

y = id % 19
x = (id - y ) / 19
```

The board stores numbers at each user-inputted index as the turn. For example, if at turn 1 the input is 0 0, then there will be a 1 stored at board[0]. This way we can use the parity at each location to group pieces into two "colors"

2.2 Group Properties

The piecies in this engine are stored as a power group object. Each power group object has the following self explanatory fields:

```
self.id = id
self.turn = turn
self.board = board
self.size = 19

#1d array that stores a power group (or power group id) at id.
self.board_groups = board_groups
#set of all power groups
```

```
        self.master_set = master_set
        #A max of one piece is under ko
        self.ko_piece = ko_piece

        self.members = set([id])

        self.liberties = set([])

        #set of enemy pieces that are directly adjacent to this
        self.immediate_enemies = set([])

        #ints of captures
        self.black_caps = black_caps
        self.white_caps = white_caps

        #locates the main group, only one per group of id's
        def locate(self, id):
            #will make faster, check for local group

            for member in self.master_set:
                for element in member.members:
                    if id == element:
                        return member
```

2.3 union

Go pieces have the useful property that combining two same color groups will also combine most of their properties. This is handled by the union method:

```
    #combines other group with this group
    def union(self, other):
        self.members.update(other.members)
        self.liberties.update(other.liberties)
        #maintains state (1)
        self.liberties.difference_update(self.members)
        self.immediate_enemies.update(other.immediate_enemies)

        self.board_groups[other.id] = other.id
        #delete other group
        self.master_set.remove(other)
        del other
```

Self explanatory for the most part. The sets for each are merged whenever two groups are combined. This method of combination assumes that each group is always individually in a valid state so it doesn't have to check for inconsistencies. The only potential merge issue is handled by (1). When a piece is played, it calculates the empty squares and saves that as liberties. If a new piece

is played it doesn't check if its liberties change so that is why (1) is needed. When two groups merge, an arbitrary group is considered the master group which contains all the information for the entire group. The other group is deleted and its index in board groups is changed to the id of the master group. In this approach, there is no redundancy and suprisingly is compatible with all the game rules.
The master set is the set of all groups which means that searching requires O(n) time which is the same as the disjoint set algorithm (list implementation, tree has no practical remove). This approach was chosen over disjoint set because it is intuitive to understand and work with.

endunion

3 Captures, Removing and Ko Check

Each group object has a self capture method which is called when a group is considered dead. There is also a method to check if a group is dead. The dead method is large because of ko. I have a generaled method in directions.py that calls a method in all four directions but the ko case is strange since I require return values. It seems easier to have this four direction check redundancy than to try and generalize it:

```
def verify_enemy_dead(self, place):
    if self.board_groups[place] != -1:
        place_group = self.locate(place)
        if len(place_group.liberties) == 0:

                place_group.self_capture()
#At this point, capturing is complete

#Ko pseudocode, actual block is large
    #Gives counters of enemies
    counter = 0
    For N S E W of the capturing piee played into ko
        if is enemy piece:
            counter ++

    #corner edge and other ko conditions
    if self.id == 0 or self.id == 18 or self.id == 342 or self.id == 360:
        if adjacents == 1:
            self.help_ved(place)
    elif self.id % 19 > 342 or self.id <19 or self.id % 19 == 18 or self.id % 19 == 0:
        if adjacents == 2:
            self.help_ved(place)
    elif adjacents ==3:
            self.help_ved(place)
```

Essentially, it calculates the required number of enemy pieces adjacent to the capturing move required for a ko. Then it compares it with the actual number of enemy pieces and if it matches, help ved is called:

```
def help_ved(self, place):
    self.board[place] = -2
    self.ko_piece.add(place)
```

It marks the ko spot as a -2, as, pieces can only be placed on -1's which is how the board is initially initialized. This is done by putting it in the ko piece set which is cleared after every turn in the board class.

For the meaning of other values, recall that positive integers on the board are used to indicate black or white pieces respectively in the board list. In the board group list, it stores at an arbitrary group member's index, the group itself and stores the group id at all other member indexes.

After each piece the following actions are done:

```
#creates a new group for each new move
self.board_groups[id] = power_group(id, turn, self.board, self.board_groups, \
    self.master_set, self.ko_piece, self.black_caps, self.white_caps)

self.board[id] = turn
self.master_set.add(self.board_groups[id])
directions.directions(id, self.board_groups[id].verify_enemy_dead)


self.value_update(x,y,turn, moves, plot_every)
```

The piece is initialized, the board and group groups and master set is updated and it checks for captures and updates the value graph.

3.1 Value Graph, Influence Map and Liberty Map

The board keeps an internal influence map that is used to estimate the score or influence. We use matplotlib (or alternatively sage) to show a color graph that makes it easier to see which player has more influence in a specific area. The "ai" can be run with the png file open to see the updates in realtime. The winner approximated by adding up the influences at each board index.

```
#uses inverse square law centered on x and y
local_value = [[round( sign/(((x - i) ** 2 + (y - j) ** 2)**.5), 1) \
                if x !=i or y !=j else sign for i in xrange(self.size)] for j in xra

def heat_graph(self, c, name, v, symbol, s):
    clf()
    x = [i % 19 for i in xrange(self.size * self.size)]
```

```
y = []
for i in xrange(self.size -1 , -1 , -1):
    for j in xrange(self.size):
        y.append(i)

scatter(x,y,s=s,c = c, vmin=-1 * v , vmax= v  , marker = symbol)
savefig(name)
```