# Table of Contents

[ANALYSIS 1](#__RefHeading___Toc532_980705524)

[Introduction to Problem 1](#__RefHeading___Toc534_980705524)

[Research / Consumer 2](#__RefHeading___Toc536_980705524)

[Current Systems 3](#__RefHeading___Toc538_980705524)

[Identification of End Users 4](#__RefHeading___Toc540_980705524)

[Client Interview 4](#__RefHeading___Toc542_980705524)

[User Needs 5](#__RefHeading___Toc544_980705524)

[Acceptable Limitations 6](#__RefHeading___Toc546_980705524)

[The Rules of Go 6](#__RefHeading___Toc548_980705524)

[The Smart Game Format 9](#__RefHeading___Toc550_980705524)

[System Objectives 10](#__RefHeading___Toc552_980705524)

[Optional Objectives 11](#__RefHeading___Toc554_980705524)

[DESIGN 11](#__RefHeading___Toc556_980705524)

[Modelling 11](#__RefHeading___Toc558_980705524)

[Internal Data Storage 14](#__RefHeading___Toc976_2099725429)

# ANALYSIS

## Introduction to Problem

In this project, I intend to implement the board game Go, as a playable program.

Go is akin to chess, draughts, and other strategy board games in that it is played by two players, black vs white, in a turn-by-turn fashion. The board is a grid, where pieces are placed on the intersections of lines – the standard size in professional tournaments is 19x19. Both players aim to surround the other’s pieces with their own and ‘capture’ them from the board. Stones cannot be moved once placed, and the empty points that a player surrounds become part of their ‘territory’. When the game has ended, their score is calculated as their total territory minus the number of their stones that their opponent captured, and the highest score determines the winner. Players who want to improve at the game often do so through through short “Go Problems”, puzzles where the goal is to find the optimal sequence of moves.

It is my intention that my implementation allows local play between two human players, and also allow users to select and play through Go problems.

## Research / Consumer

To gain an understanding of the problem, I first had to understand the game itself – its rules, the interactions they create, and how these might be modelled in a computer program. For this, I turned to an excellent website, “Senseis Library”:

<https://senseis.xmp.net/>

Through it, I was able to get a good understanding of the terminology used by Go players, various common formations and situations, as well as look at some example games playing out for myself. This gave me a good feel for how a game of Go evolves as moves are made, and some idea of how my program might be able to best accommodate this. I also read through the Wikipedia entry on the [rules of Go](https://en.wikipedia.org/wiki/Rules_of_Go" \l "Scoring_systems), particularly the section on scoring systems. This helped me start to think about what rules my implementation would use, and if any could be chosen by the user.

The site also introduced me to the “Smart Game Format”, used by many existing Go solutions; a description of its purpose and function can be found on its [official site](https://www.red-bean.com/sgf/user_guide/index.html). Files using the format (with “.sgf” extensions) can actually describe numerous other games, such as backgammon. The format consists of a tree made up of nodes, allowing it to keep a record of a game (by having each node contain a move) and variations in the lines of play. That latter quality is very useful for describing Go problems, and this turns out to be a very common use-case in practice. I soon found [a large repository](https://github.com/gogameguru/go-problems) of problems in the form of .sgf files, a far simpler method of providing them to a user than generating them. That would require some form of AI and machine learning, but as this is an area of ongoing research from organisations like [DeepMind](https://deepmind.com/), I decided it was beyond the scope of this project.

## Current Systems

Online, you can find many free Go sites, with varying degrees of quality and differing features. Almost all of these choose to use a GUI, with moves made with the mouse rather than the keyboard. They also usually have accounts, lobbies, chat systems – all useful for people interested in joining an online community or getting on a leader-board, but quite the opposite for anyone who wants to simply play the game. Indeed, the focus many of them place on ranked tournaments and chat rooms means they can be incredibly distracting to use. Those that offer Go problems, such as ‘[goproblems.com](http://goproblems.com/)’, have very mixed collections in terms of quality and difficulty – coupled with their distracting interfaces, someone wishing to simply play them would be better buying a published collection and doing them on a physical board.

Outside of the Web, there are stand-alone applications, such as ‘[gnugo](https://www.gnu.org/software/gnugo/)’, a popular open-source version. It has an excellent, simple interface that runs in the terminal, as well as extensive options for different rules and board-sizes. However, while it does support loading from .sgf files, it can only do so to load a game that has been played. When I presented it with Go problem files, it simply chose the first variation and played the game from there; it fundamentally isn’t designed with problems in mind, focusing instead on an AI opponent.

## Identification of End Users

I have chosen my brother, Elijah Mullan, as my end-user. He likes board-games, and him and I have often played Go together on a physical board. I am going away for Uni next year, so he wants to stay practiced, but doesn’t like playing against the computer. Additionally, he’s intent on getting better, and has heard that many masters of the game like to practice using problems.

## Client Interview

In order to understand my brother’s needs more precisely, I conducted an interview with him.

Why do you need my program?

I like playing Go on a physical board with my brother, but a lot of the time that isn’t possible. I want to be able to keep practising playing so that I can keep improving, and problems have lots of the tricky situations that I find really hard when I’m playing a real game. I’ve tried playing them from a book, but it’s hard to not accidentally spoil them while you’re reading!

What device would you be running it on?

Since I don’t own a smartphone, my laptop would probably be the best choice. I’d want it to be easy to install and run, so that if I want to play it on a different computer sometime I can do so without any trouble.

Are there no existing programs that let you do this?

There are lots that I’ve look at on the Internet, but they always want you to sign up and make an account, and a lot of them try to get you to pay for premium stuff that I don’t really need. I don’t want to play with strangers, so the tournaments they put on aren’t of any interest to me, and I’ve never enjoyed playing the computer. Plus, my laptop’s a hand-me-down and starts to struggle when a lot’s happening on the screen at once, and most of these websites do that.

What features would you want this program to have?

To let me through Go problems. I want to be able to select them from a list, and for it to tell me when I’ve made a wrong move. I’d also like the option to play against another person on the same computer, in case I want to play a friend and a physical board isn’t around.

How would you like the program to look and behave?

Nothing fancy. I’m pretty good with computers, so I’d be comfortable with it running in the terminal, but in case my friends who aren’t as good with computers want to play, I want it to be clearly laid out. I’d like moves to be entered with board coordinates so that it’s easy to use. Also, when a human game is over, I want the winner to be displayed, as well as the actual score, so I can see how close it was.

## User Needs

* A program that allows the user to play Go, against a human or within a pre-made “Go problem”.
* It must provide a selection of such problems, and have the capability for more to be added.
* A working handicap/”komi” system to ensure games stay competitive.
* Support for 13x13 and 19x19 board sizes, to accommodate how much time the users have.

## Acceptable Limitations

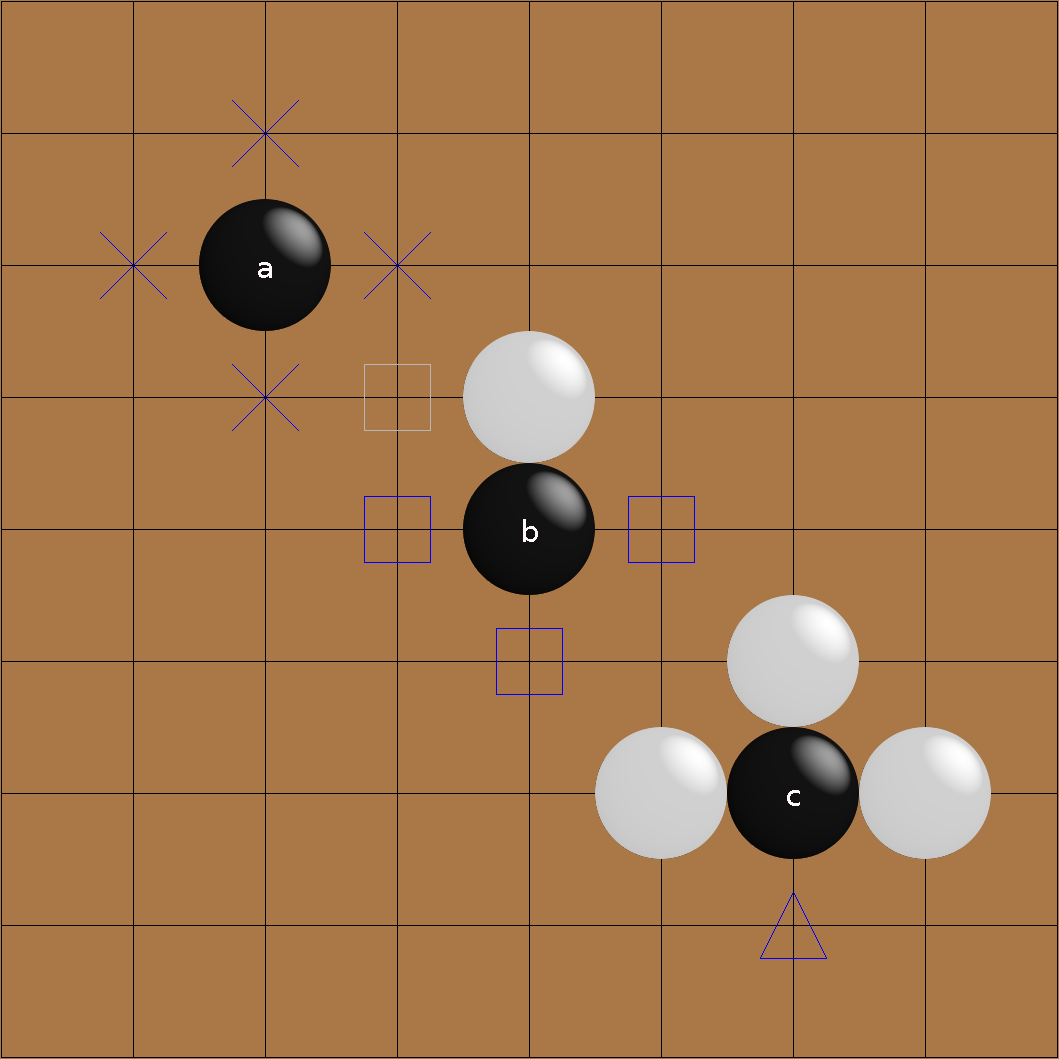
* It does not need an extensive range of optional rules for games– it can choose to implement the most common ones.
* The number of problems included does not need to be huge, and they don’t have to be of a especially high difficulty.
* It can use a simple console-based display, without GUI elements like scroll bars or button prompts.

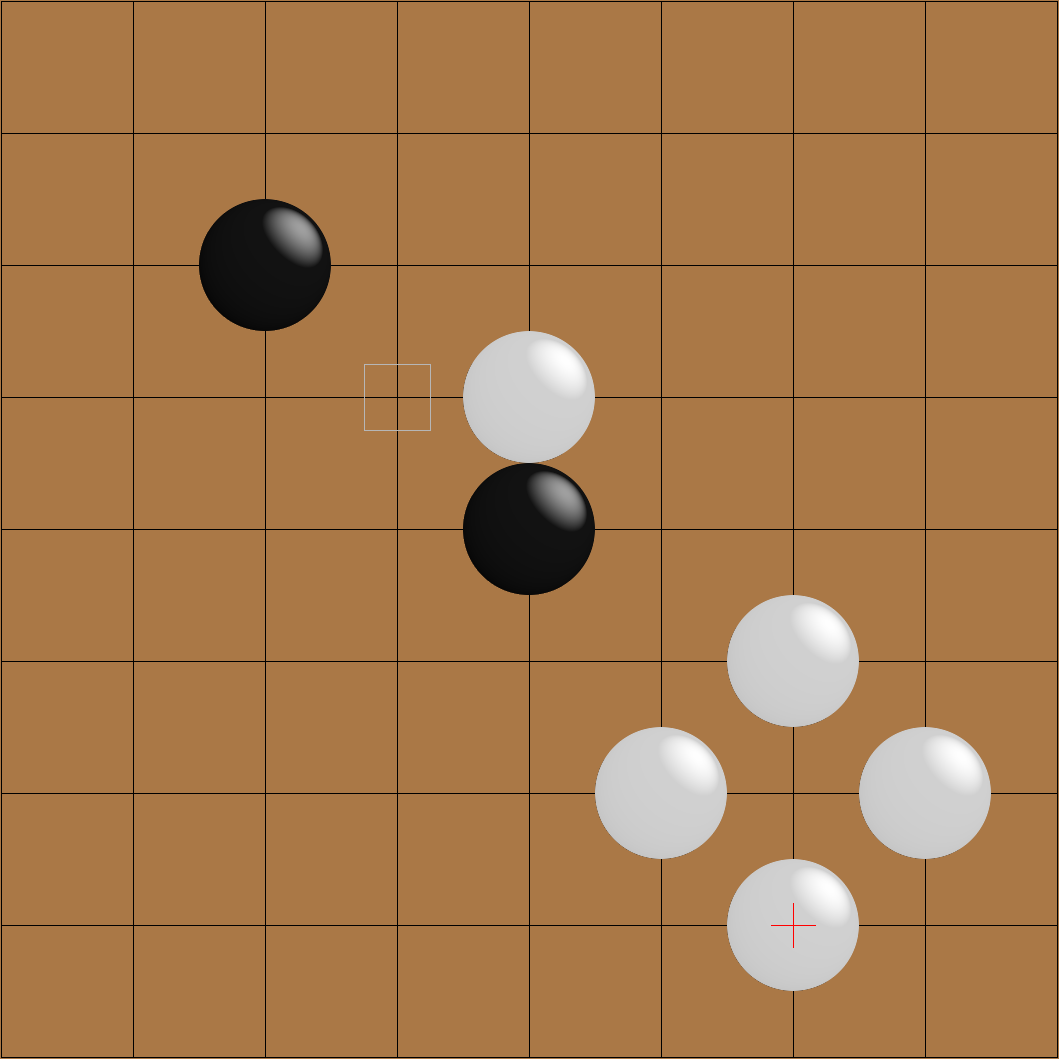
## The Rules of Go

As I’ve found when researching the game, its rules are quite simple for a human to understand, but may be quite complex for a computer to accurately simulate.

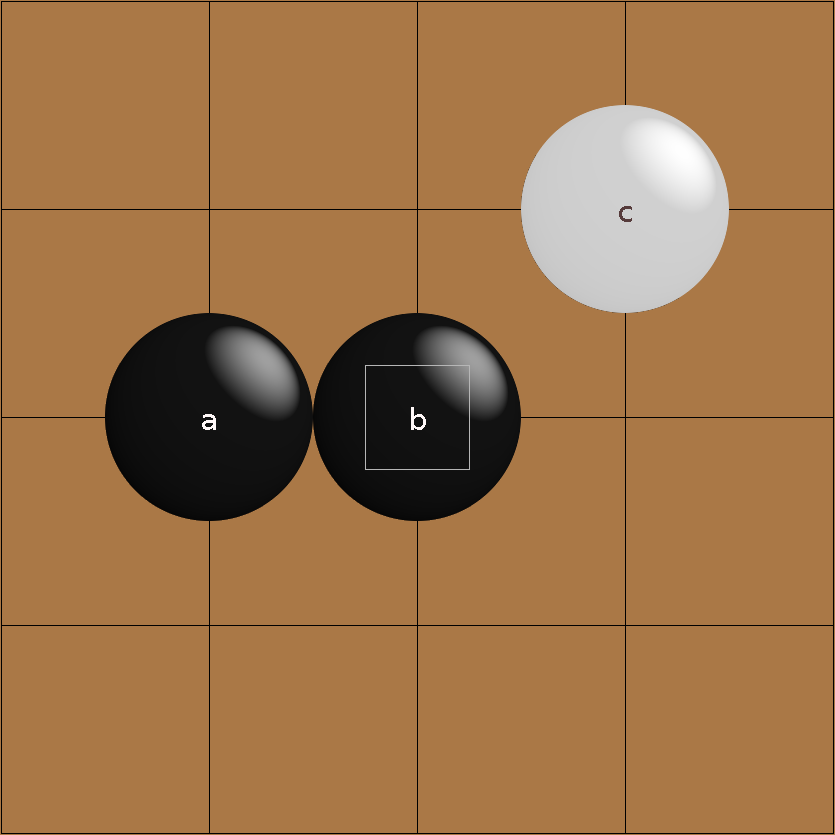
The game is played on a 9x9, 13x13 or 19x19 grid, with the last being the most popular. It is played by two people, one with black stones and the other with white, with the only action being to place one of their own stones onto an intersection (point) of the grid, or to pass their turn. Indeed, the game ends when both players pass consecutively. The goal is to have the most territory on the board at the end – I’ll explain that later.

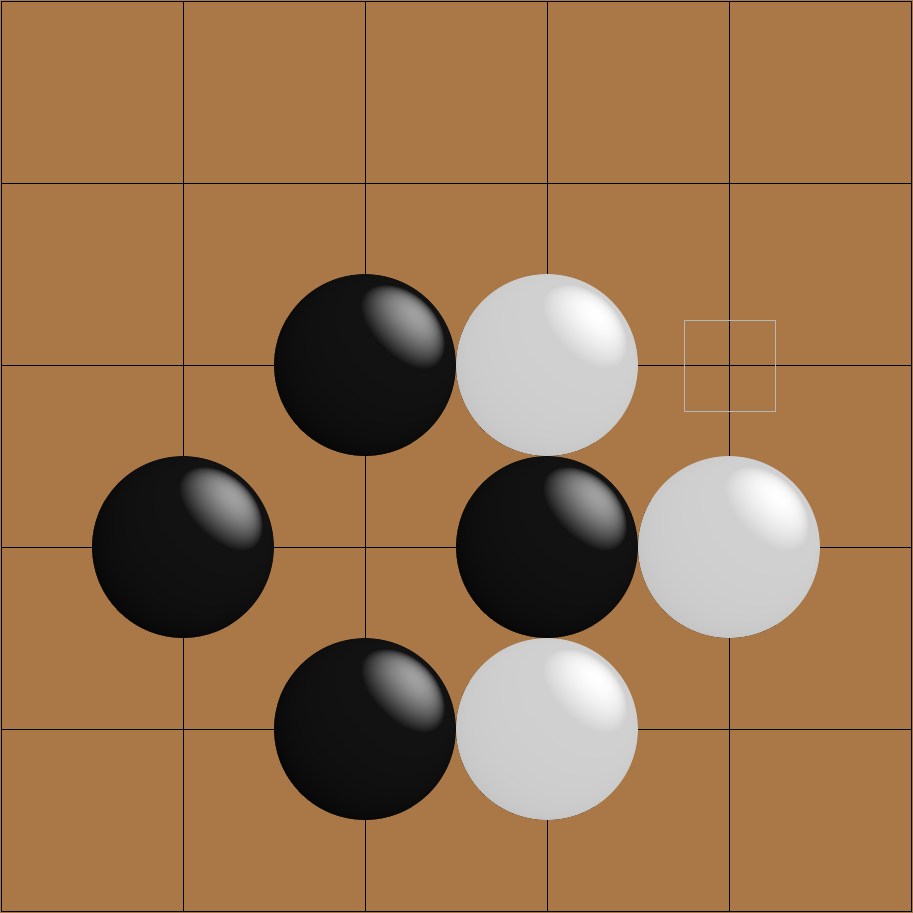
At any given time, every stone on the board has a certain amount of liberties, equal to the number of empty points neighbouring it in the cardinal directions. For example, the black stone A has four liberties; B has a white stone beside it so it has three. C has just one liberty, and so if white chooses to play in the last empty point on her turn…



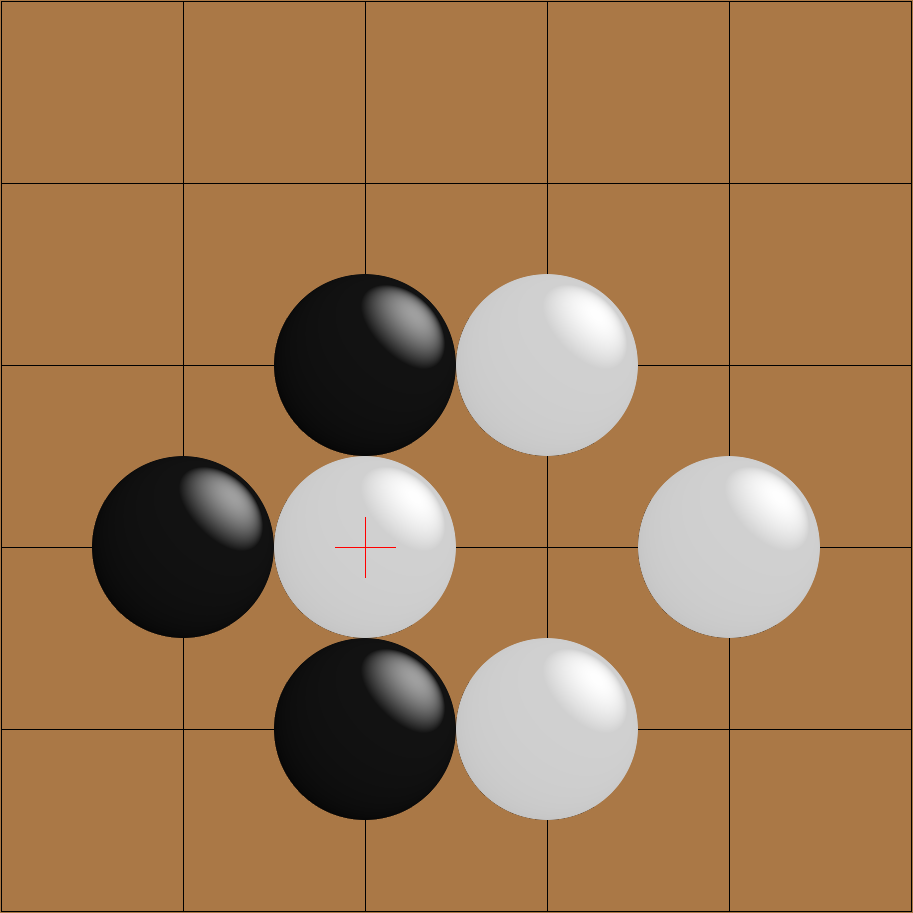
C gets captured by white, and is removed from the board.

This rule is by itself quite simple. Where it becomes more complicated is that neighbouring stones of the same colour share liberties. Here, A and B have a total of six liberties. Note that C does not share liberties with them (it has four), as it is diagonally adjacent – however there is no problem with two of its liberties being on the same points as two of A/B’s.

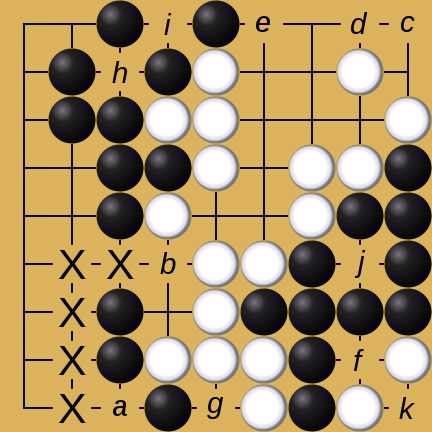


Then there is the rule of ko, in which infinite loops of moves by preventing a player from playing a move that would repeat a previous board state. It typically arises in a formation like this one:

If white plays to capture black’s stone:



Then black cannot capture white’s, as this would repeat the position from two turns previously. Therefore, he must play elsewhere on the board first.

At the end of the game, territory is counted. An empty intersection is said to belong to a player’s territory if all stones adjacent to it or to an empty intersection connected to it are of that player’s colour. Take this example:

a is adjacent to black stones, so it cannot belong to white’s territory. However, it is connected to b, through the path in the diagram, and b is adjacent to a white stone. So, a is neutral territory. The same logic applies to c: it’s connected to d, which is adjacent to a white stone, but also e, adjacent to a black. Ergo, c is neutral, as are f and g. h is only adjacent to black stones, so it is black territory. So are i and j, and k is white territory for the same reason.

Both players final score is equal to their territory minus the number of stones their opponent captured, and the highest score wins.sssss

## The Smart Game Format

A .sgf file contains text that describes a game tree. That tree consists of nodes, and those nodes can have certain properties.

The root node contains info pertaining to the entire file, such as: the version of the format being used, in the FF property; the game being played, in GM, which is set to 1 for Go; the size of the board, in SZ; and the initial board state for a problem can be expressed through lists of coordinates in the AW and AB properties, which place down white and black stones respectively (critically, this is different from making a move).

Subsequent nodes can have annotations (using various properties), comments (using C), and black and white moves (with B and C).

The general syntax is very helpfully summarized on the official website with an EBNF definition:

Collection = GameTree { GameTree }

GameTree = "(" Sequence { GameTree } ")"

Sequence = Node { Node }

Node = ";" { Property }

Property = PropIdent PropValue { PropValue }

PropIdent = UcLetter { UcLetter }

PropValue = "[" CValueType "]"

CValueType = (ValueType | Compose)

ValueType = (None | Number | Real | Double | Color | SimpleText | Text | Point | Move

| Stone)

## System Objectives

1. On opening the program, the user should be greeted with a menu allowing them to choose between playing a game against a human, or a Go problem.
2. In a game, the user should be able to enter coordinates through the terminal to specify where they want to place a stone, or to pass their turn.
3. Stones should have their liberties kept track of, and those with no liberties should be captured and removed from the board.
4. Adjacent stones of the same colour should share liberties, and be captured at the same time.
5. Players should be unable to make a move that:

* Attempts to place a stone on an occupied point.
* Is suicidal (results in the players own stones being captured).
* Violates ko (repeats a previous board position).

1. The game should end when both players pass.
2. The score should be calculated based on territory and captured stones.
3. The user should be available to give the black player an arbitrary handicap before a game has been started.
4. The user should also be able to select between (at least) 13x13 and 19x19 board sizes in games.
5. The user should be able to choose from a range of included problems.
6. While playing a Go problem, the same limitations on moves should apply, with the addition that a move not described in the problem should give a “Out-of-tree” error.
7. The user should be able to toggle “hints” to assist in solving problems.
8. Comments from the sgf file should be displayed at the appropriate game states.
9. When the problem is complete, the user should be returned to the menu.

## Optional Objectives

1. Enable “Go-by-Email”, whereby the user can have long-term “correspondence” games with multiple people, where moves are sent via email and parsed by the program.
2. Allow the user to enter variable board sizes before a game, between 5x5 and 30x30s.
3. Add timers for when the players are selecting a move.
4. Allow played games to be exported to .sgf files.

# DESIGN

## Modelling

My first major design decision was to decouple the three major parts of my project: the logic of the game itself, the implementation of problems, and the interface the user interacts with. This way, each part could be developed independently, and issues could be more easily isolated to being caused by a particular part.

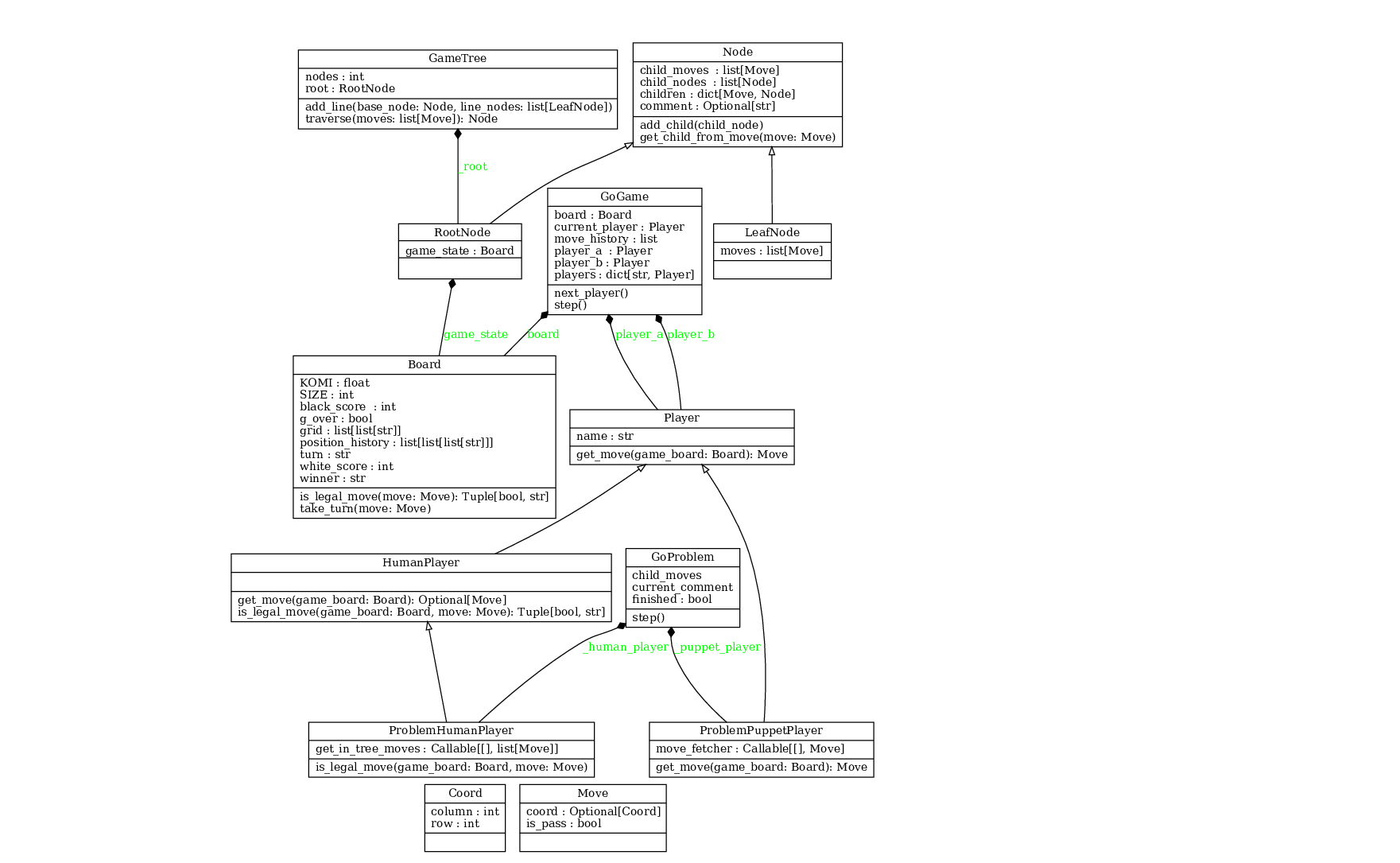
As such, I chose to begin with the game logic itself, the core of which I put in a class called Board. It was clear from my research that this class would need the following to properly enforce Go’s mechanics:

|  |  |  |
| --- | --- | --- |
| Variable/Method Name | Variable/Return Value type | Purpose |
| grid | 2D array (list of lists) | The intersections of the game board need to have their contents kept track of – with a 2D array, this can be represented in a format that suits the shape of the board. |
| turn | String | This is needed so that the current player can be kept track of and displayed on the screen. |
| white\_score, black\_score | Integer, integer | This determines the winner, and can be displayed at the end of a game. |
| g\_over, winner | Boolean, string | The former will allow a program using the game logic to know when the game is over; the latter allows it to display a winner. |
| is\_legal\_move | Boolean | This is needed for a program dealing with human players to see if their inputted moves are actually legal. |
| make\_move | N/A | Once a move is confirmed as legal, this allows it to be performed on the game board. |
| SIZE | Integer | The size of the length of the game board (such that its dimensions are SIZE x SIZE) |
| KOMI | Float | The handicap value to be added to black’s score when the game ends. |

To simplify describing positions on this board, I created Coord, which has the row and column of a particular element in Board’s internal 2D array. Move represents the actions players can take, containing either a Coord or a boolean value to reflect a passed turn.

A ‘layer’ of logic above this is the GoGame class. It has a Board object, and two Player objects. Every time the step method is called, it 1) passes the current board to the current player, 2) receives a move back, 3) gets the board to execute this move, and 4) changes the current player.

Player itself is only an abstract class. This means adding new kinds of players, such as AI opponents or users connected over the network, should be relatively painless. All you’d need to do is create a new class that inherits from Player with the desired behaviour.

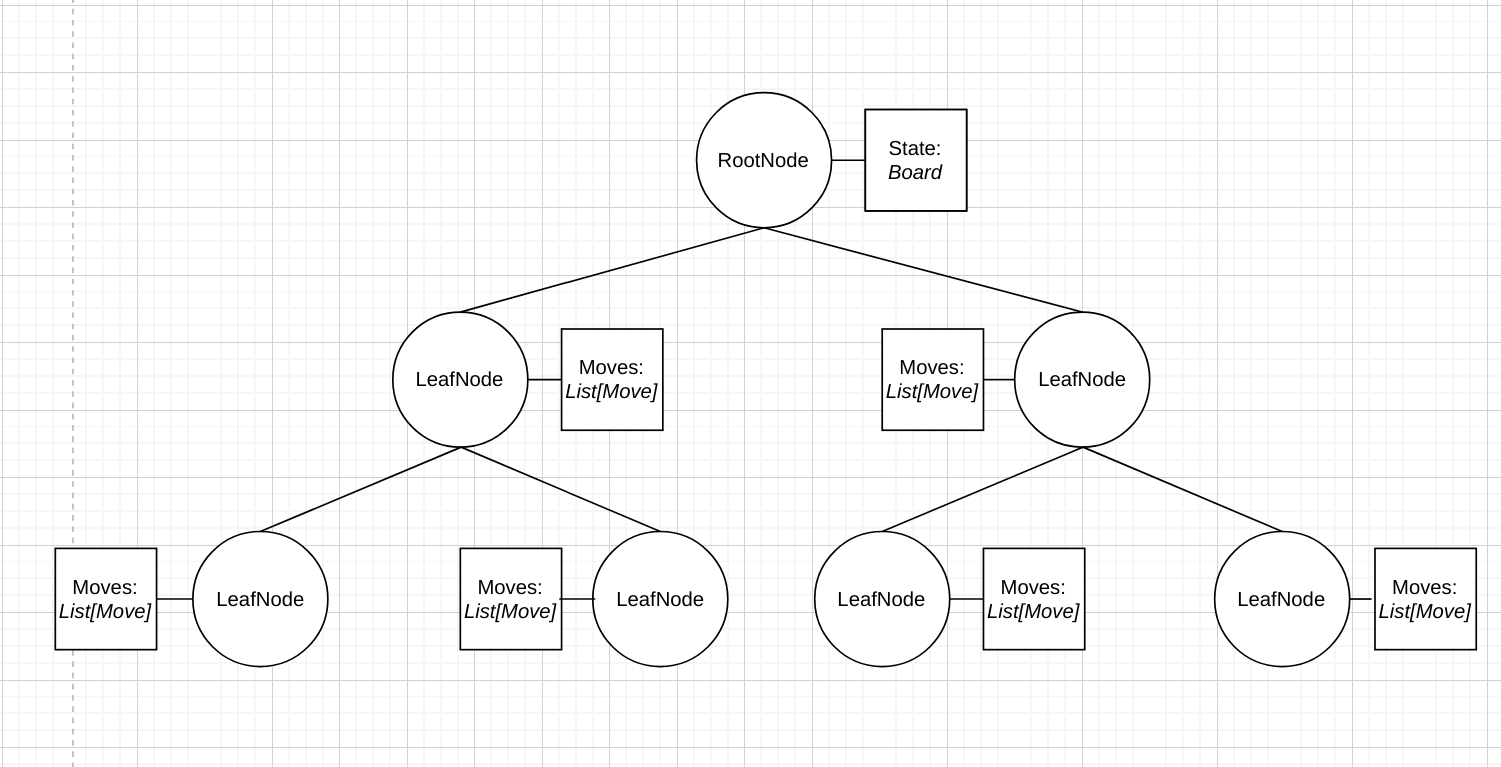
The following UML diagram illustrates how these classes relate to each-other and the rest of the system:

## Internal Data Storage

The GoProblem object, which inherits from GoGame, needs to contain data from the relevant .sgf file. It does so through the composition of a GameTree object – this is because .sgf files use a tree structure

The actual relation between nodes isn’t handled in the GoTree class, but instead the Node class itself. Each Node has a dictionary called children, where each key is the Move object that leads to the child node, and the corresponding value the node itself. So, if you had a node called my\_node and wanted to access its child node through a Move object called my\_move, you’d do so through my\_node.children[my\_move]. Each node can also have a text notation in its comment property.

Node also has two subclasses: RootNode and LeafNode. The former contains a boardstate (of type Board, naturally); the latter a list of moves (a black move and its subsequent white move – although technically both the object and its .sgf counterpart could have an arbitrary number of moves for each node). RootNode objects are the root for each GameTree, and each subsequent child node is a LeafNode.

As such, a GameTree object could be visualized like this:

## Algorithms

Several of the key algorithms used in my project are utilized in the capturing of stones. Every time a stone is placed, a procedure called capture\_if\_without\_liberties is run on it and each of its neighbouring coordinates:

PROC capture\_if\_without\_liberties(crd: Coord):

if contents(crd) = None:

chain ← get\_chain(crd)

if NOT(has\_liberties(chain)):

capture(chain)

A chain is a group of stones of the same colour that share liberties – in the program itself, a list of Coord objects.

The get\_chain function has to return every coordinate of the chain that a given coordinate is part of:

PROC get\_chain(crd: Coord):

chain ← [coord]

current ← coord

to\_check ← QUEUE()

to\_check.enqueue(coord)

colour ← contents(coord)

WHILE NOT(to\_check.is\_empty()):

neighbours ← neighbours(to\_check.dequeue())

FOR i IN neighbours:

IF contents(i) = colour:

IF NOT(i IN chain):

to\_check.append(i)

chain.append(i)

RETURN chain

In short, it performs a *breadth-first search* for all coordinates of the same stone colour (as if the board were a graph, with each coordinate as a node connected to its immediate neighbours). A simple implementation of a queue was used for this, and can be seen in the program source code.

Finally, the chain is passed into the has\_liberties function to determine whether it should be captured or not:

PROC has\_liberties(chain: list[Coord]):

FOR stone in chain:

neighbours ← get\_neighbours(stone)

FOR n in neighbours:

IF get\_contents(n) IS None:

RETURN True

RETURN False

(None of course being the value for an empty point on the grid.)

# TESTING

*https://youtu.be/7wxen6BpTIc*

|  |  |
| --- | --- |
| User menu inputs | 0:00 |
| User inputs in a game | 0:13 |
| Capturing stones | 1:00 |
| The ‘suicide’ rule | 1:32 |
| The ‘superko’ rule | 1:46 |
| Ending the game and scoring | 2:13 |
| Komi | 2:34 |
| Territory in scoring | 2:51 |
| Maximum board size | 3:28 |
| Playing a Go problem | 3:54 |
| Enabling problem hints | 4:26 |
| Exitting the program | 4:48 |

# EVALUATION

## Second Interview

To see how well my finished program lived up to my end-user’s needs, I conducted a follow-up interview with him:

Does the completed program perform the tasks you wanted it to?

It does. I can play games against other people, and can try out lots of different problems.

Is it easy to use/interact with?

Most of the program is very easy to use, all you have to do is type. Some of my friends found this a little difficult compared to a mouse or a touch-screen though, so buttons might have been nice, since on big boards it can be a little tricky to figure out exactly what letter and number a point on the board has.

Are there any improvements/additional features your experience/usage would benefit from?

Being able to save games and continue them later would be nice, since some games can drag out over a few days and you have to keep the computer on. Also, while there a lot of problems, most of them aren’t very difficult. A way to add more, or a bigger selection to start with, could be an improvement.

## Errors/Bugs

From my testing, I did not find any errors in either the menu or the player vs player function – game rules behave as expected and scores are calculated correctly. However, there were occasional bugs in the Go problems, where a particular move on a problem would cause the program to crash, with an error message pointing to a failure in retrieving the response to the players action. One such example is playing [s1] on the second problem.

After investigating I found that the problem was that the .sgf file itself had no second move for that node. In my research I never came across this being a valid use of the format, so it seems that the issue is with the creators of the problems. However, I cannot be sure, and my program doesn’t have the capability to detect and avoid these issues.

## Implemented Requirements

All the core system objectives of the system were met – the rules of the game were correctly implemented, with relevant messages to the user when these are infringed; handicaps can be set; the user can choose from a range of problems; they can be given hints if they wish; comments are displayed on appropriate board states; and the user is always returned to the menu when they complete an activity.

Of the optional objectives, only variable board sizes was accomplished, with the user indeed able to choose any size between 5x5 and 30x30. While I could theoretically have allowed even larger boards - using various other symbols once I ran out of capital letters - board length increases game time exponentially, so anything bigger wouldn’t be in the interest of virtually *any* user.

In terms of implementation, I’m happy with how the game logic is handled in Board and GoGame. The problems, less so. Firstly, my parser makes use of a lot of nested selection to work, and could have been made much more elegant through the use of regular expressions, or even parser libraries. Secondly, the GoProblem classes inheritance of GoGame, although architecturally simple, is convoluted in practice, passing functions as arguments to dummy objects in order to ‘trick’ its parent class’s methods. And thirdly, passing while playing a problem used to cause an error. While I managed to *technically* fix this through an if statement that detects passing, I did not actually find the root of the problem.

From my end-users feedback, I can see that the ability to export games as .sgf files (one of the optional objectives) could have been very useful, and I do not think it would have been especially difficult to implement, provided I tidy up my parser as I’ve said above. This could also go hand-in-hand with a “Go-by-Email” system, another optional objective, although I am less sure about how easy that would have been to achieve. As for their comments around the interface, I feel that this would significantly increase the difficulty of the project, as I would have to research Python GUI libraries on top of my existing research. The cross-compatibility of the program could suffer as a result as well – the present system requirements, a command prompt and working version of Python, allow my program to run on practically any OS in active development.