University of Glasgow | School of Computing Science

Honours Individual Project Dissertation

Go - Project

Jude Haris
May 2019

1 Introduction

The ancient game of Go is regarded as the oldest board game still being played in the modern day. The origins of Go are not completely known but it is said to be invented in China 3000-4000 years ago. The game is formed by very simple rules that is played out most commonly in a 19x19 board. Each player takes turns placing a black or white stone on the intersections of the grid lines on the board. The board size is very crucial for the project. Due to increased board size compared to most western board games, a computer will take significantly more time to process through each possible move in every turn of the game to determine the final outcome. On the other side the basic principles of the game are much simpler to program and execute in software. Some complicated rules such as self-capture, and ko rule do exist. While implementing these rules can be simple,the effect of them being implemented correctly is great. For instance without implementing the ko rule it would be impossible to make progress in the game search tree in certain situations where moves could be played so that a never ending battle for the same point on the board takes place. Brute force methods to solve board positions would get stuck in these situation and loop infinitely.

1.1 Motivation

The simplicity of the game's rules are deceiving as the game's strategy and tactics used are very complex. The possible amount of different games of Go to occur is inconceivably large. It has been said to learn the game it takes little time but to become an expert or remotely good at it takes years maybe even decades. To allow a computer to calculate the perfect move via brute force would take too long to be considered viable. The current best AI built to play Go is AlphaGo, it has defeated one of the world's best Go player Kie Jie in 2017 at Future of Go Summit. AlphaGo uses Monte Carlo Tree Search used with neural networks to evaluate the board and calculate the best move. With that in mind AlphaGo is yet to be anywhere near perfectly solving Go due to the enormous complexity and possibility of moves.

1.2 Aim

The aim of this project was to create an AI to play against users in Go, which can choose the correct move given a board position. Specifically, the project's aim was to create a program which given a Go life and death problem it would solve it, if solvable or return as unsolvable. First milestone was to achieve perfect AI which would choose the correct move given infinite time or smaller problem with less valid moves. The goal was to use brute force search method to look through every possible board state that could occur given the current state and choose the move which will lead to victory for the computer if there is any. The final goal of the project was to alter and enhance the AI which was achieved through the first milestone to be able to solve larger problems in the same time scale. And to do so, use heuristics to determine how favourable a board position is after searching for certain depth into the possible line of choices and then picking the line which leads to the most favourable board position. This final aim was set so that more realistic problems could be solved using the final product as without larger problems would simply take too long to play through. The motivation behind the project was to create a tool for Go players to use to improve tactical skill in Go via an interactive problem solver which helps the user figure out the solutions to life and death problems by having the computer play against them.

1.3 Project Outline

2 Background & Related Works

To progress through the project certain level of knowledge of the game is required. Here are the key points of research conducted during the project.

2.1 Go

To understand Go as normal player would was crucial in the progress of the project. Many simple questions needed to be answered about the game and how it is played. Also, basic concepts and terminology needed to be understood before tackling complex strategies which are built up from these basic concepts.

2.2 Go Rules

There are many varied rulesets for Go. These rules sets can be broken into rules used during play and rules used at the end of play. The rules during play changes slightly and in most cases, they don't change at all. Any changes of these rules do not affect the strategy of the game at all. Rules used at the end of play refers to scoring and determining the winner of the game, while differences in these rules can be important in normal play, when subjected to the aim of this project they become irrelevant due to dead or alive nature of life and death problems. For these reasons the rules used during play are defined within the project but rules for the end of the game are left out, instead rules to decide if a problem is solved are defined. The basic rules according to Japanese Rules of Go set by Nihon Kiin translated by James Davies [1] were used to define the ruleset used within the project and some variations were made to suit the problem-solving nature of the project rather than playing of Go. They are as follows:

1. The board starting state can be set up as needed by the user to define a life and death problem.
2. Players takes turn alternately placing the colour of their team. One player places black stones only while the other places white stones.
3. The board contains 19x19 gridline pattern with 361 intersections. For purpose of limiting life and death problems boundaries are used to limit area of play.
4. Each turn consists of few parts as follows:
    (a) Current player is to place stone of their colour on any empty intersection within the boundaries set for the problem.
    (b) Then any of the opponent's stones that does not have a liberty is removed from the board. See example 1 at Figure 2.1.
    (c) A move is invalid if it will cause one or more stone of the current player's colour to be captured – this is to prevent self-capturing moves. See example 2 at Figure 2.2.
    (d) Removal of opponent's stones take precedence over self-capture check. This allows for moves which captures opponent's stones but is suicide if the opponent's stones are not removed first. See example 3 at Figure 2.3.

5. Ko rule – Player is not allowed to place a stone on point A if it captures one stone which was placed on point B during the opponent's previous turn which captured one stone on point A. This is to prevent infinite repetition of the same few moves. See example 4 at Figure 2.4.
6. The problem is solved when attacking player has no valid moves to play and the keystones are still on the board. Otherwise the problem is solved if all the keystones are captured.

2.3 Go Terminology

Throughout this dissertation many terms are used to describe many aspects of Go some of which can be said to be common agreed upon terms to desrcibe Go and others are created for the purpose of the project.

Keystones – Set of predetermined stones which are used as the objective of the problems. These stones are to be captured by the attacking player and kept alive by the defending player.

Attacking player – In terms of life and death problems, the attacking player is the one which is trying to capture the keystones on the board.

Boundaries – Set of predetermined intersections that are removed from play for the given the problem, stones cannot be placed in them.

Liberty - Liberty of a stone is any adjacent empty intersection of the stone or any empty intersection of any stone connected to the original stone. See example 5 at Figure 2.5.

Connected - Two stones are connected if they are of the same colour and is adjacent to each other or there is a set of stones of the same colour which are connected to both stones. See example 6 at Figure 2.6.

String – Set of stones that are connected of the same colour.

Keystring – String which contains one or more keystones.

Atari – String is in atari when it only has 1 liberty.

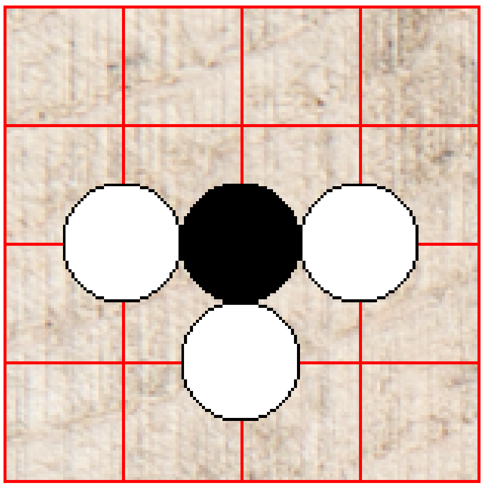
2.4 Life and Death Problems

2.5 Examples

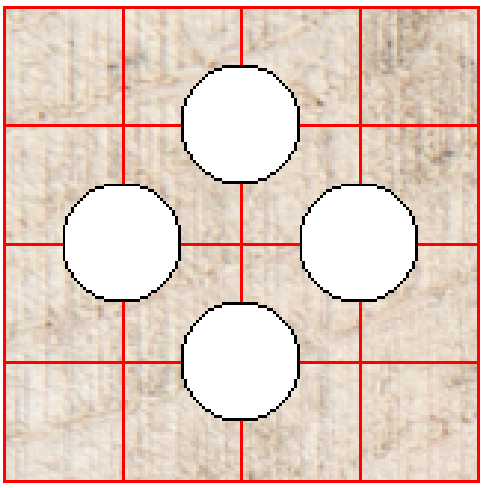
2.6 Related Works

Martin Müller's Explorer [2] was a strong program developed with the intention of playing Go. It captures key aspects of computer Go which can be used along with the minimax algorithm in one program. Explorer can evaluate the board in terms of safe territories for each player and can distinguish between safe stones and safe territories. Safe stones are any stones that are deemed alive but safe territories requires that no opponent stones can live within the safe stones. Explorer also takes into consideration of Semeai positions which are capturing races that occur within a game of Go. The victor of Semeai can be determined early sometimes and hence doing so during the evaluation could save searching further during minimax.

While explorer aims to play the Go, this project aims to create a program to solve life and death situations. To do so, determining safety of keystones within a problem is important. Safety of strings during the board evaluation can determine whether a correct move is made or not.

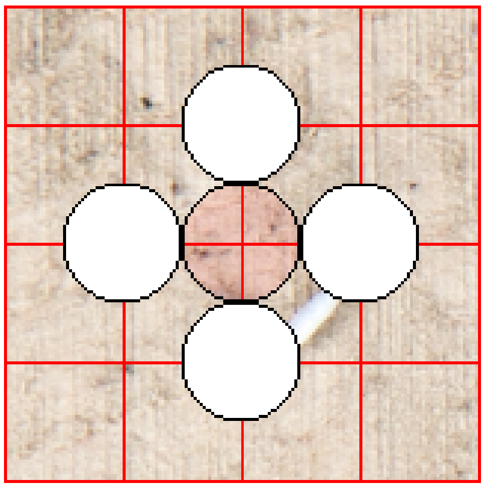


(a) *Before Move*



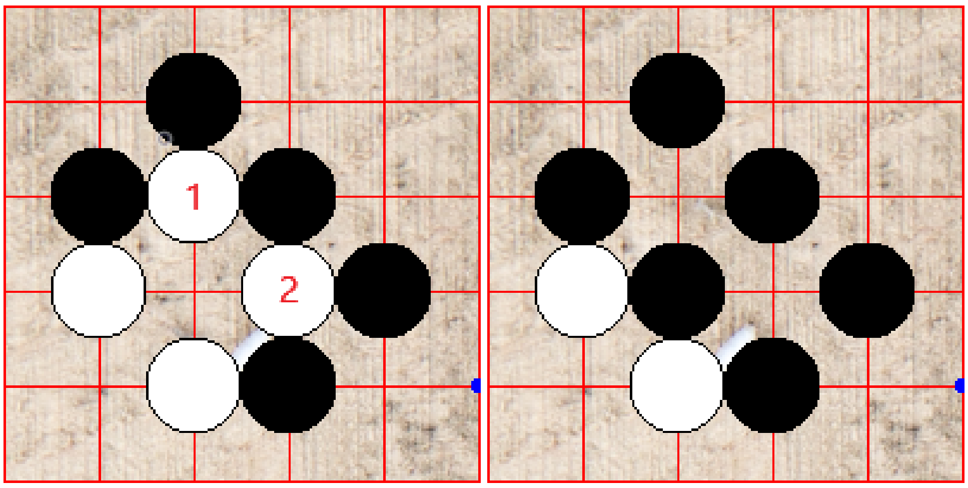
(b) *After Move*

Figure 2.1: *Example 1: The black stone is "captured" as the top white stone is placed because the black stone has no liberty left.*



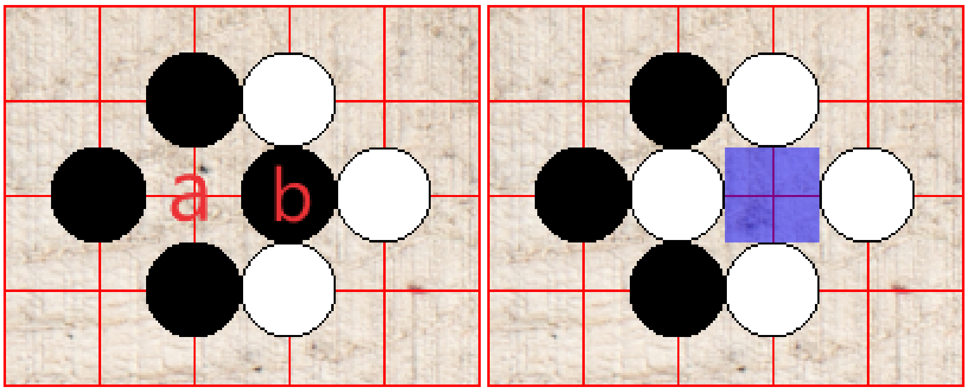
(a) *Middle point is invalid*

Figure 2.2: *Example 2: A black stone cannot be placed in the middle of the four white stones because this would lead to self-capture.*



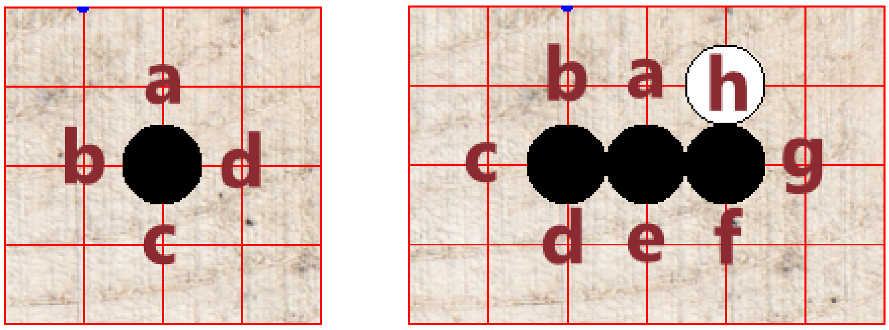
(a) Before Move *(b) After Move*

Figure 2.3: *Example 3: A black stone can be placed in the middle of the four white stones because 1 and 2 will be immediately captured before the self-capture rule is applied.*



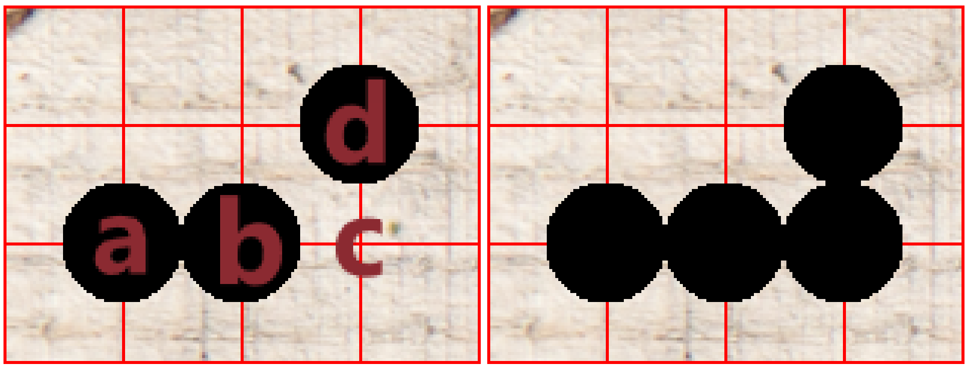
(a) Before Move *(b) After Move - Blue Square to represent KO*

Figure 2.4: *Example 4: Once a white stone is placed at a to capture the black stone at b , black stone cannot be played at b next turn by the opponent to capture the white stone placed at a.*



(a) A black stone and its liberties *(b) A black stone string and its liberties*

Figure 2.5: *Example 5: The liberties of a stone or a string of stone are the empty adjacent intersections. In Figure 2.5a intersections a,b,c and d are the liberties of the black stone. In Figure 2.5b intersections a to g are the liberties of the black stone string but h is not a liberty due to the white stone that occupies it.*



(a) Before Move *(b) After Move*

Figure 2.6: *Example 6: The black stones a and b are connected but d is not connected to either a or b. If a black stone is placed at c then a,b,c and d are all said to be connected to each other.*

3 Requirements

The project requires a well-defined and clear list of requirements due to the possible scale of it. Time constraints and resources available had to be taken to consideration before further work on the project was done. More precisely there was a need for identifying key aspects of the goals of this project and put them into more concrete requirements, so they can be implemented accordingly to achieve the goals of the project. Also, to keep track and have a streamlined work flow throughout the project the following requirements and scope were set.

3.1 Functional Requirements

To begin with the project's first milestone's aim which is to create an AI to play against user in Go life and death problems using brute force search. To achieve this, certain key requirements needed to be met:

The basic Go board should be displayed on the screen in which users can place stones in turn through mouse clicks. It needs comply to all the rules of Go and should capture and remove stones from the Board automatically after each turn. The program must be able to generate all the valid moves that can be made each turn and highlight them on the screen. Furthermore, the program itself must let the user create Go problems within it. To do so, split the program into two modes of use, an editor mode and player mode.

Editor mode needs to permit the user the ability to place black and white stones where needed without needing to take turns to create the initial board state they want their problems to be in. The state of the board needs to be valid always such that rules of Go would allow the board position to occur in normal play for it be able to be created in editor mode. Captured stones must be removed from the board and also only one ko point exists on the board. Editor should be able to define the boundary of play area, specifically to be able to choose which points on the board are in play for the problem. Keystones should be place able via editor mode, the keystone/s's purpose is to identify the stone in which the problem revolves around. Users should also be able to select who to play first, and whether to capture or keep alive the keystone/s. Finally, the editor mode is required to let the user to create a text description of the problem and save the entire problem into a text file. The save files needs be human readable and the program should be able to load them back to either edit the problem or to play them. The player mode is required allow the user to play through problems. This means it must allow users to load problems in from save files and play them out against the computer. To achieve the goal of creating an AI to play the game, there needs to be some features within the program to help with debugging. The ability to reset the problem back to original state, undo or redo a move and to also disable or enable the AI when required must be implemented into player mode.

The final requirement to achieve the first milestone needs to be addressed. There is a need for the program to contain a simple but effective Alpha-Beta pruning search algorithm to allow the correct move to be determined by the computer given board of stones and which colour's turn it is, so that the computer can make that move to allow that colour to win. To make things simpler when problems are created, keystone/s needs to be placed on the board to allow the algorithm to surround its evaluations according to the status of the keystone/s on the board. For example, if the current turn is white and there is/are black keystone/s on board then the AI run the search algorithm and would evaluate to choose the move which would lead for the black keystone/s to be captured.

To achieve final aim of the project which is to adapt the AI created in the first part of the project to solve larger and more complex problems the following critical requirements are needed:

To introduce heuristics to the brute force Alpha-Beta search algorithm the program should contain a specialised module which evaluate the state of the board of each colour and returns how good or bad it is. This is so that given depth cut off to the search tree is implemented, the algorithm can return a value more accurately predict what would have been the result if the search has continued deeper. The AI must also use a complex move generator which given set of valid moves it picks certain number of moves and orders them according to the likelihood of being a good move using heuristics. Then the AI searches through this list of moves before going through and checking all the other possible valid moves.

In addition, with these critical requirements some sub-requirements are needed to help with testing and to finalise the product. A set of Go problems are needed to be created and built into the product. This is to allow users to immediately try out the player mode without needing to create problems by themselves, will be especially be useful for players new to Go. Another feature needed to be implemented was a valid move checker, to return a list of all possible moves either to enforce the rules of the game to the users to play properly or to allow the AI playing against the human to have a way to generate a list of possible moves to search through.

With all the requirements of the project it had become evident the scope of the project could vary drastically. To create a perfect Go playing AI on full 19x19 board would be far from achievable given the time and resources put into the project. The main constraints to decide the scope were:

- Time given for AI to search and find the correct move?
- The number of given valid moves a problem contains which the AI should solve?

These two constraints are closely related to each other, more valid moves the AI is expected to handle the more time it should be allowed but with an upper bound set to allow users to interact and solve the problems without waiting too long. These were needed to be adjustable according to the feedback given by user testing. Hence the AI had to be built in way to allow for flexibility regarding these two parameters. To do so, the program gives the ability to choose what depth the AB pruning algorithm searches to before it cuts off to the user. This directly affect both run time of the algorithm and scale of the problem as bigger problems can be allowed with smaller cut off depth also would lead to less time required.

3.2 Non-Functional Requirements

3.3 Must/Should/Could/Won't have

4 Design

4.1 Design Principles

4.2 Brute Force Search

For the computer to play against a human it needs to be able to determine the best move it can make. Simplest way to do this is by searching through the game tree and determining the end results of each valid move. The game tree consists of every single board position that is possible to occur from the current board position. Every valid move's board position is explored deeper and deeper in the game tree until the game is over or a condition is met. Within the program the ability to do this "brute force search" was essential as it is the foundation to which rest of the program's AI is built upon.

4.2.1 Alpha-Beta Pruning Search Algorithm

The basis of finding the correct move within the project is the Alpha-Beta pruning search algorithm. Alpha-Beta is an optimisation of the simple minimax algorithm. Minimax is perfect for searching Go game tree where two players are playing and where one player's gains are directly linked to the other player's losses. A simple implementation of minimax will search each valid move of the initial board position and choose a move which results in victory even if the opponent plays perfectly, if there is such a move. In terms of Go given the computer is black, during the search minimax alternates between playing as black and white until an end goal is reached and then chooses an initial move which leads black to win.

Within life and death situation of this project the end goal is easy to determine if the attacking player as no valid moves and the keystones remain on the board then the defending player has won. Alternatively, as soon as all the keystones on the board is captured then the attacking player has won. In a real game of Go, results of end games are not so simple. There might be multiple solutions which results in capture of a string but there might only be one solution which won't allow for losses in other parts of the board for the attacking player. This is ignored in this program as we are only interested in the result of life and death problems not the outcome of the board. Hence any solution which captures the given keystones are sufficient for the project. An example of minimax algorithm used in a simple number game is show in Figure 4.1

The alpha-beta pruning algorithm is smart improvement to reduce the search time of the minimax algorithm. It allows for the correct result to be determined without having to search through every possible branch in the tree of possible moves. It uses alpha and beta values determine whether further search is required or not. To begin with alpha is set -inf and beta is set to +inf, these values are passed down through the layers of the search algorithm and changes to value only affect the deeper layers of the search. If a maximizing layer finds a move which gives a score greater than the previous best move of that layer, then the alpha value of that layer is increased to the value of new best move. If a minimizing layer finds a move which gives a score less than the previous best move of that layer, then the beta value of that layer is decreased to the value of new best move. After each move is searched in a layer, the algorithm checks if the alpha value is greater than or equal to the beta value and if it is the algorithm decides that checking more

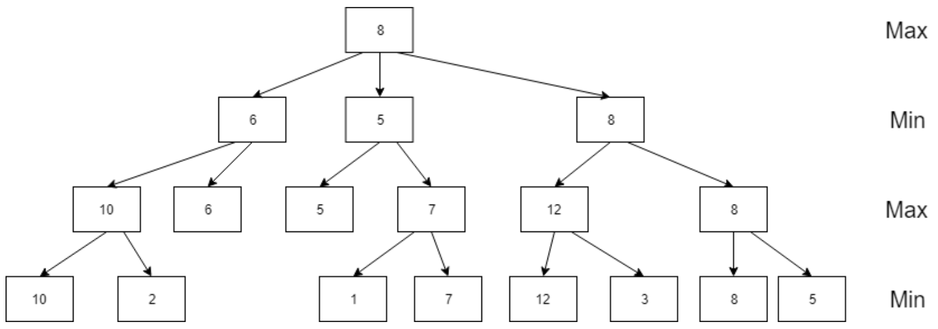


Figure 4.1: Minimax being used for simple number game. The best score the maximizing player can get is 8. Even though scoring 10 and 12 are possible when minimizing player plays optimally the score can't out be achieved.

moves in that layer is pointless and returns the best move back to previous layer. This results drastic improvements to the minimax algorithm without changing the result as it cuts out the search of many branches of the game tree which doesn't affect the result of the algorithm. An example of Alpha-Beta pruning algorithm used for the same simple number game is show in Figure 4.2

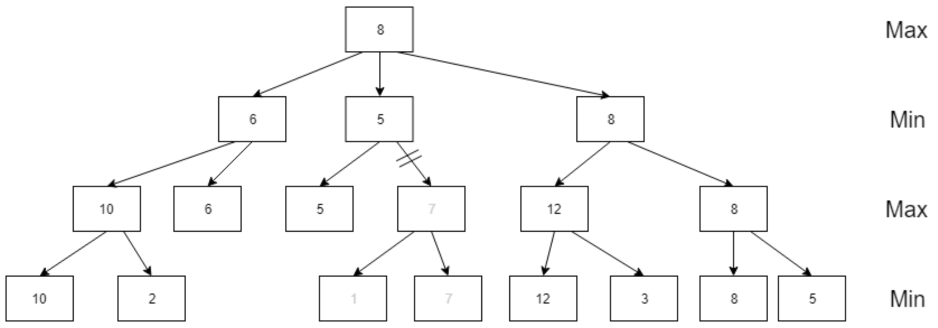


Figure 4.2: Same simple number game using alpha-beta pruning. In the second layer of the second branch after searching the first child node which returns 5 the algorithm cuts off from any further search in that branch as it realises the first branch will also be better than 5.

Pseudocode of the implementation of alpha-beta algorithm for the program is as follows:

4.2.2 Bad Move Removal

Keystones are an absolute objective in the program, they are one of the refence points which the program checks to see if the problem is solved or not. From the defender's perspective any moves that are detrimental to the immediate safety of keystones need to be removed from the list of valid moves to be searched by alpha-beta algorithm. This entails removing any moves that decreases the number of liberties of keystone to 1.

A solution can be derived from the fact that placing 1 stone can at maximum only remove 1 liberty of a keystring. Hence, there is only a need to check the liberties of keystring with 2 liberties as anymore liberties would ensure at least 2 liberties would remain intact. For keystrings with 2 liberties, placing in one of those 2 liberties does not automatically mean it is a suicidal move for 2 reasons: Move would capture opponent string; Move would increase or maintain the liberty count. If any of these two reasons are met, then the move is not suicidal. Within the program, all keystrings with only 2 liberties are checked. During the check the following occurs:

- Both liberties of the keystring is added to the suicidal move list.
- Moves that place in the liberty of any opponent string which is in Atari is removed from the suicidal move list.
- Moves which connects keystring to another string which fits the following criteria are removed from the suicidal move list: Must be same colour as the keystring; Has more than 2 liberties or has 2 liberties which are not the same 2 liberties of the keystring.
- Moves which places a stone adjacent to any empty point which is not a liberty of the keystring.

After these checks are done for all keystrings, any moves remaining in the suicidal moves list are removed from the valid moves list and returned as the good moves list. The good moves list is used during AB search instead of valid moves list to save time by not searching valid moves which would lead to immediate capture of keystones. If no moves are considered "good" then the defending player is to pass instead of making a bad move during AB search.

4.3 Heuristic Search

4.3.1 Board Evaluation

4.3.2 Pattern Searching

4.3.3 Move Generator

4.4 Problem Files

4.5 Modes

4.6 Overall Architecture

5 Implementation

5.1 Play Mode

5.2 Edit Mode

5.3 Game Board

5.4 Heuristic Evaluators

6 Evaluation

6.1 Unit Testing

6.2 Evaluation of Heuristics

6.2.1 Types of Problems

6.2.2 Success Rate

6.3 Beta Testing with Go Players

6.3.1 Questionnaires

6.3.2 Interviews

7 Conclusion

7.1 Summary

7.2 Future Works

7.3 Reflection

8 References

1.http://www.cs.cmu.edu/ wjh/go/rules/Japanese.html 2.Müller, M. (2002). POSITION EVALUATION IN COMPUTER GO. ICGA Journal, 25(4), pp.219-228.