University of Glasgow | School of Computing Science

Honours Individual Project Dissertation

Go – Project

Jude Haris
May 2019

1 | Introduction

Go Life & Death (GoLD) is a program developed in Java which can solve Go problems. 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 has very simple rules and by most official rulesets played on a 19x19 board. Each player takes turns placing a black or white stone on the intersections of the grid lines on the board. There are 361 intersections on the Go board whereas Chess's only has 64 squares to play on. The size of the Go board is a key factor in this project. Due to increased board size compared to most western board games, a computer will take significantly more time to search through each valid move in every turn of the game to determine the final outcome. On the other hand, the basic principles of the game are much simpler to program and execute in software. Some complicated rules such as self-capture and Ko rules 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 while searching Go's game tree. Some situations occur where it becomes optimal for both players to repeat the same moves over and over in order to capture a single point on the board. This would lead to a never-ending battle for the same point on the board without the existence of the Ko rule which forbids this to occur. Therefore without a correct implementation, any type of game tree search algorithm used to solve the board position would get stuck in these situations and loop infinitely. This project implements Go to allow the user to not fully play the game of Go but rather solve Go problems.

1.1 Motivation

The simplicity of the game's rules is deceiving, anyone can learn the rules of the game within 5-10 mins and will be able to play a full game of Go. The game's strategy and tactics are incredibly complex compared to the rules. Go players agrees with the sentiment that, to learn the game it takes little time but to become an expert or remotely good at it takes years maybe even decades. Due to the nature of the size of the board, the possible number of different games that could be played out in Go is inconceivably large. For a computer to calculate the perfect move via searching every valid move that could occur would take too long to be considered viable, not months or years but rather millenniums.

The current best AI built to play Go is AlphaGo [9] and its successors built by Google's DeepMind team. It has defeated some of the world's best Go players such as Lee Sedol during a challenge match in South Korea dropping only 1 game out the 5 played. AlphaGo has also defeated Ke Jie another highly ranked player in 2017 at Future of Go Summit winning all three games played. AlphaGo uses Monte Carlo Tree Search along with neural networks to evaluate different board positions and calculate the best move according to what it has learned through machine learning. With that in mind, AlphaGo is yet to be anywhere near perfectly solving Go due to the enormous complexity and possibility of moves, the team working on it is still finding ways to improve it.

The main motivation behind this project is to create a tool for Go players to enhance their skill at a certain aspect of Go. It is not to create an AI which plays whole games of Go but to create an AI which can play Life and Death problems and solve them.

1.2 Aim

The aim of this project was to create a tool program for Go players, from beginners to experienced which allows them to practise life and death problems and in doing improve their overall skill level at the game. The aim also includes that the tool should contain an AI to play against its users in Go, which can identify all the valid moves and play the correct move given a board position.

Specifically, the project's aim was to create a program which given a life and death problem for Go it would solve it. If the problem is solvable, the AI should play the move which solves the problem otherwise it should play the move that it thinks is the best possible. The project was divided into two big milestones to keep track of the progress.

The first milestone within the project was to implement a substantial number of features, enough to let the users create problems and view them. This included implementing the rules of Go, having the ability to place stones and also to implement the bare bones of the AI which would be altered in the second part of the project to implement heuristics. This meant to implement a simple tree search which would be able to choose and play the correct move given infinite time or a problem small enough so that every valid sequence of moves could be searched. Other features such as loading and saving problem files were also part of the aim for the first milestone.

The second milestone was to alter and enhance the tree search implemented in the first milestone to be able to solve larger and complex problems. To achieve this GoLD needed a way to limit the search space hence decreasing the time taken to search but also creating uncertainty of choosing the best move. The aim for the second milestone was to introduce heuristics to determine how favourable a board position is after searching for certain depth into the possible sequence of moves and then picking move which leads to the most favourable board position. The overall goal of the second milestone was to enable GoLD to solve more realistic problems in a viable time frame for users to use the program without waiting too long.

1.3 Project Outline

Here is a short summary of each following chapters:

Chapter 2 Background & Related Works – Explains key concepts which are used within the project, introduces Go and some relevant details of Go in more detail. Also looks at some similar work done.

Chapter 3 Requirements – Deals with expressing the functional requirements of GoLD and also some non-functional requirements.

Chapter 4 Design – Goes in depth about the AI and the tree search used with GoLD. Explores the heuristics used to enable the GoLD to limit time spent searching the game tree. This includes board evaluation, move ordering and move generation.

Chapter 5 Implementation – Talks about the key features of GoLD and how they are implemented. Goes into some detail on how the Go board is represented within the program. Explains the pattern matching functionality used during board evaluation and move generation.

Chapter 6 Evaluation – Contains result of Beta testing and analyses of it. Also evaluates the performance of GoLD for every problem within the problems folder provided to beta testers. Goes into detail about the unit tests designed and tested to prove the correctness of the program.

Chapter 7 Conclusion – Concludes the whole paper by looking at the successes of the project, places of improvements and any possible future work.

2 | Background & Related Works

In order to achieve the aim of creating program certain level of knowledge of the game of Go was required. Knowledge such as the rules of Go and details regarding Life and death problems were important factors while designing and implementing GoLD. Along with understanding Go, researching papers and finding inspiration from similar work done were key to this project.

Go is a complex board game where the rules are far simpler than the strategies and tactics used to play it. To understand Go as a human player would was crucial in the progression of the project. To begin with, many simple questions needed to be answered about the game and how it is played in order to design GoLD. Basic concepts and terminology needed to be understood before tackling complex strategies which are built up from these basic concepts.

2.1 Rules

While there are no official rules for Go and many varied rulesets being used around the world , they are all quite similar. None of them really affect the strategy or tactics used within the game hence comprehending one ruleset was enough to understand the game in order to implement the rules within GoLD. This is especially true because the GoLD does not require counting scores which is the part of the rulesets that vary the most. The basic rules according to Japanese Rules of Go set by Nihon Kiin translated by James Davies [1] were used to define the basic ruleset used within the project. Some variations were made to these rules to suit the problem-solving nature of the project rather than playing whole game of Go which will be explained in a later section.


(a) Before Move   *(b) After Move*
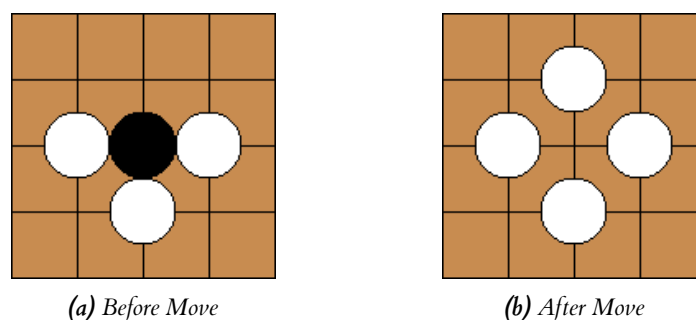
Figure 2.1: *The black stone is "captured" as the top white stone is placed because the black stone has no liberty left.*

The basic rules are as follows:

1. Initial Go board is empty containing no stones unless handicaps are applied.
2. Black plays first and then white. Players take turns alternately placing stones of their own colour.
3. The board contains 19x19 gridline pattern with 361 intersections. Stones are played on the intersections not the square in between. Stones can only be played on empty intersections. These intersections are also referred to as points on the board.
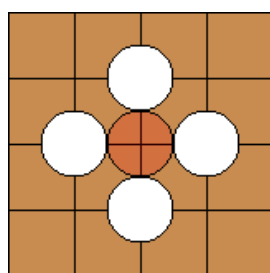
(a) Middle point is invalid for black to play
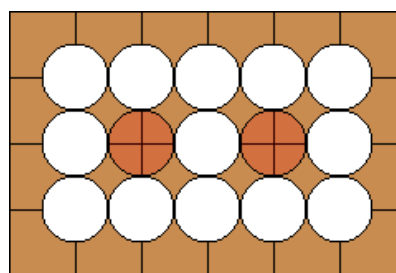
(b) Both middle points are invalid for black to play

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

4. Each turn consists of few parts as follows:
   (a) Current player is to place a stone of their colour on any empty intersection they desire.
   (b) Then any opponent's stones that does not have a liberty is removed from the board. See 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 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 Figure 2.3.
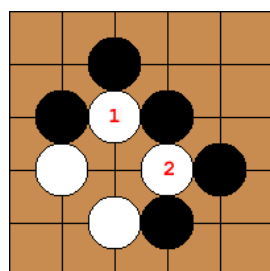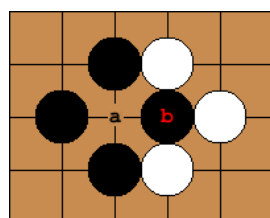

(a) Before Move
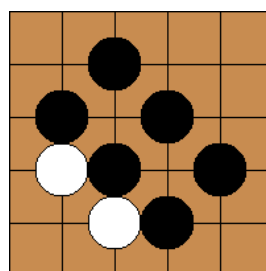
(b) After Move

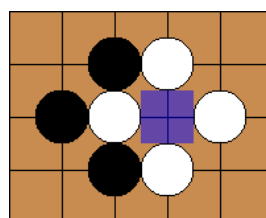**Figure 2.3:** *A black stone can be placed in the middle of the four white stones because 1 and 2 will be immediately captured before the self-capture rule is applied.*

(a) Before Move

(b) After - Blue Square is Ko

Figure 2.4: *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.*

5. Ko rule – Player is not allowed to place a stone on point A if it will capture exactly one opponent stone which was placed on point B during the opponent's previous turn. This rule only applies if the opponent in the previous turn captured exactly one stone on point A. This is to prevent infinite repetition of the same few moves. See Figure 2.4.

6. The game is over when both players pass consecutively and the player with the more territory wins. (Scoring can be complicated but is irrelevant to understand the basic rules)

2.2 Rules for GoLD

The rules stated in the previous section outline all the important rules of Go. Some of these rules are slightly altered in order suit the problem–solving environment set in GoLD. The alterations are as follows:

1. The initial board is set up to represent a life and death problem - should be a valid initial board state.

2. Not all 361 intersections are can be played on – boundaries are set by the creator of the problem

3. For the purpose of there problems, one player is considered to be the attacker and the other defender.

4. Attacking player is not able to pass unless the attacking player has no valid moves and the Ko rule is restricting play.

5. Two consecutive passes does not end the game/problem.

6. The problem is solved when attacking player has no valid moves and cannot pass, if the keystones are still on the board then the defending players has won. Otherwise the problem is solved if all the keystones are captured hence the attacking player has won.

A valid initial board state is defined by the following :

- Contain at least one keystone
- All keystones are the same colour
- Must have a valid empty intersection to play on within the boundaries set
- Can only contain up to one point of Ko
- Stones that have no liberties cannot exist in the board.

In a normal game of Go there are no boundaries set on the board, every intersection on the board is playable if the other rules allow it. To achieve the aim of this project there is a definite need for boundaries to be set on the board for each problem. Without doing so, the number of valid moves required to be searched by the AI would be too large to determine the best move in a limited time. To set boundaries during the creation of a problem, the creator can deem relevant points on the board as valid points to play on. Any points on the board with preplaced stones are also regarded as within the boundary. For example, a black stone is captured, the empty point remaining will be regarded within the boundary of play. It can be very difficult or even impossible for a problem creator to determine all relevant points for the problem. GoLD leaves it in the hands of the problem creators to define life and death problems with relevant boundaries. This is a crucial trade-off which is required to allow the program to be able to find the correct answer in feasible time.

The reason behind the 4th alteration to the basic rules is to restrict infinite passing which would allowed due to the 5th alteration. Attacking player is allowed to pass under the circumstance that they have no valid moves and an empty intersection invalid due to the Ko rules, this resolves some rare occurrences where boundaries set by the creator of the problem restricts the attacking player from winning.

2.3   Go Terminology

Throughout this paper many terms are used to describe many aspects of Go some of which can be said to be common agreed upon terms to describe 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.

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

Valid points – Set of predetermined intersections that are allowed to be played on for the given the problem. Valid point can also refer to a point which the current player is able to play on.

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

Keystring – String which contains one or more keystones.

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 Figure 2.5.

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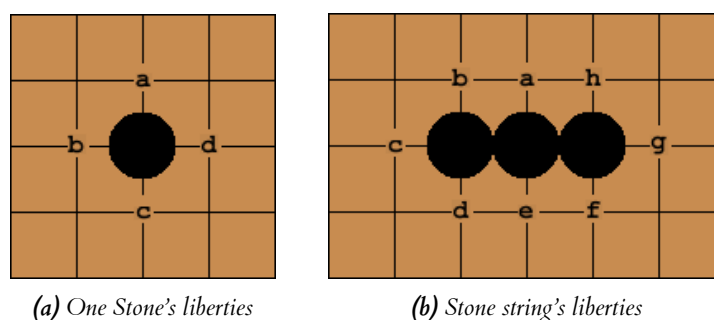
Figure 2.5: *The liberties of a stone or a string of stone are the empty adjacent intersections. In Figure 2.5a intersections a,b,c and d are the liberties of the black stone. In Figure 2.5b intersections a to g are the liberties of the black stone string but h is not a liberty due to the white stone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

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.

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

Atari Point – The last remaining liberty of a string.

Ko Point - An empty intersection which restricted from play due to the Ko rule

Seki – Two strings of opposing colour that are alive next to each other. Neither player can capture the opponent string without first sacrificing their own.

2.4   Life and Death Problems

Life and Death is an essential part of Go. Life and Death is the term used to describe the battle that takes place to either capture stone enemy stone strings or to defend and keep alive ally stone strings. Situations on the board where it become crucial to defend one's own stone strings or capture enemy ones arrive very often. During the end game in Go the ability to win these battles

with as much territory as possible will decide the victor. Beginners are highly recommended to understand and learn the how to play out these battles as they are clear deciders in the game. Many of the literature on Go are purely based around providing life and death problems and explaining the importance of them and how to tackle such situations appropriately.[3][4] The purpose of GoLD is to create a tool which helps players to understand and solve these problems.

2.4.1 Dead, Alive or Unsettled

The three main way to describe a string of stones are alive, dead or unsettled. To deduct accurately the state of a group of stone can be difficult and requires a lot of experience and knowledge about the possible moves that could be made and the counter plays to them and the result of counter plays.

(a) Before Moves   *(b) After Moves*
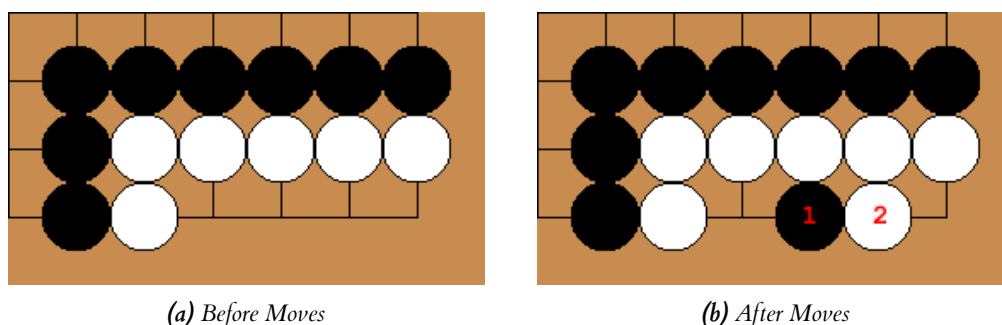
Figure 2.6: *Straight Four in the corner is alive*

A group of stones are said to be alive if the group cannot be captured even if the opponent is to plays first. To expand on this, even if opponent plays a move which threatens the group of stone there is always a responding move which will in turn keep the group of stones alive. Figure 2.6 shows an example of white group which can be deemed to be alive, if black first plays at 1 then white will play at 2 , if black first plays at 2 then white will play at 1 both outcomes will lead white group having two eyes.

(a) Before Moves   *(b) After Moves*
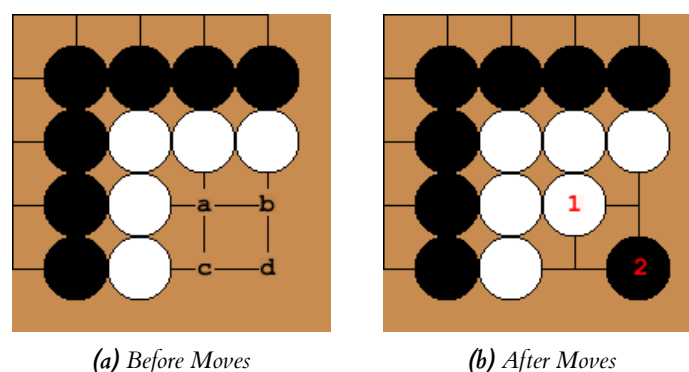
Figure 2.7: *Square Four in the corner is dead*

A group of stones is said to be dead if the group can be captured even if the group's colour can play first. A white group can be deemed dead if white can play the first move and black has a responding move which will keep the white group in a dead state. Figure 2.7 shows group of dead white stones, for this group to live it requires white stones on two points diagonals to each other and no white stones on the other two , for example if white can play at a and d then

it becomes alive. This is impossible to achieve as black can respond to white's initial move by playing at diagonally opposite point seen in Figure 2.7b.

A group of stone is said to be unsettled when the group is alive if the group's colour can play first but dead if the opponent plays first. Groups which are unsettled are crucial to outcome of the board, whoever can play first near the group can determine who controls the whole territory the unsettled group surrounds. Figure 2.8 shows an unsettled white group which has vital point at a where if white plays first at a then the group achieves two eyes and is alive. On the other hand, if black plays first at a then white cannot achieve two eyes hence it becomes dead.

(a) Vital point at a   *(b) White lives with 2 eyes*
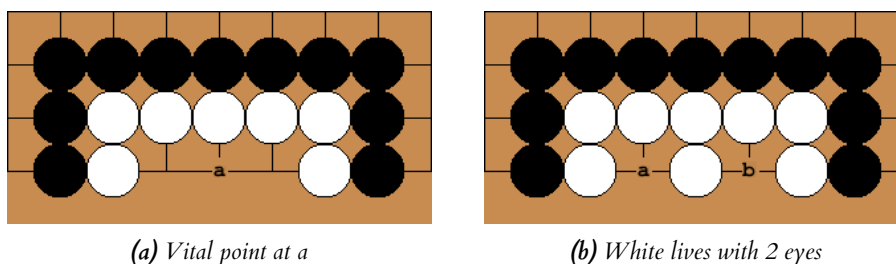
***Figure 2.8:** Straight Three on the side is unsettled*

2.4.2   Eyes

The concept of an eye is key in understanding life and death problems and Go in general. There are no perfectly defined statements on what an eye is but in general an eye is a space surround by a group of the same colour. Single eye point shapes are shown in Figure 2.9. In these shapes stones surrounding eyes are important if any of them are missing then the eye will no longer be a real eye. An important deduction about a single point real eye is the fact opponent cannot play on the single point eye unless the eye is only liberty of the surrounding stones. If two real eyes are connected by the same group of stones, then it becomes impossible to capture that group of stone unless the eyes are covered up by the its own colour. These groups of stones are unconditionally alive which means even if the opponent is able to play multiple times in a row the group cannot be captured. Figure 2.8b shows an example of a group of white stone which contains two eyes at a and b , due the self-capture rule black can never place in a without having a stone at b and same applies of b hence the white group in unconditionally alive.
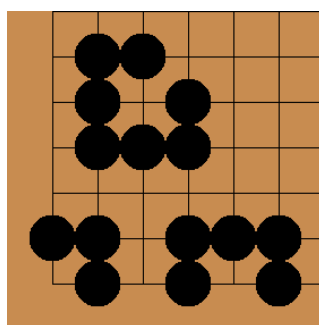
(a) Different Single Point Eyes

***Figure 2.9:** Single Point Eye in the middle, side and corner*

In cases of larger groups which surround more empty points, these empty points are referred to as the eye space. Eye space are important in determining early on within a game whether a group

is dead or alive. Eye space can be reduced to create eyes , the greater the amount of eye space a group surrounds the greater the ability it has to create two real eyes and live unconditionally.

When eyes are not full developed on the board for example at a in Figure 2.10a they are deemed to be half eyes, depending on who plays first at b decides whether a real eye is formed or is stopped from being formed. Half eyes are important because two half eyes add up to one eye. A group such as the one shown in Figure 2.10b which has a real eye at a and 2 half eyes at b and c can be deemed as alive due to the fact that when black plays to deform one half eye at 1 white can play at 2 to fully form the other half eye into a real eye creating an unconditionally alive group.
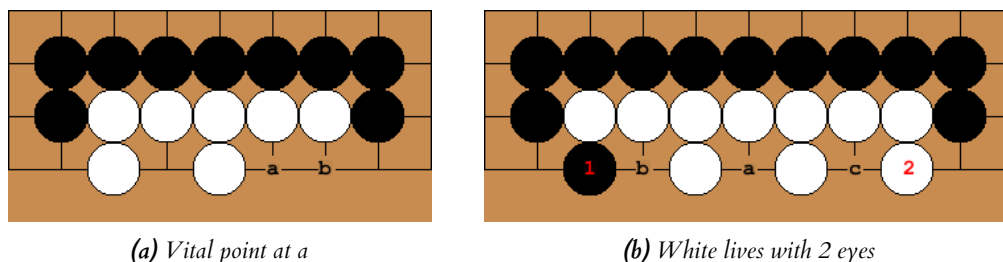

(a) Vital point at a   (b) White lives with 2 eyes

Figure 2.10: Straight Three on the side is unsettled

2.4.3   Outcomes of Life and Death

The possible outcomes of some life and death situations are not as obvious it would seem. There are different type of life and death results. The solution found for a problem can vary the result of the whole board depending on which the following categories it fits into.

Unconditional life or escape into centre of the board is the best outcome a defending player can expect , this means their group of stone will be scoring them the points with uncertainty. Another but lesser type of victory is life through Seki, which means mutual life, this is where two groups opposing colour of stones live sharing liberties and neither player can try to capture the opponent's group because it will lead to their group's death first. When attacking group of stones is able to live in mutual life along with the defending player's group then the outcome is slightly less favoured than unconditional life due the end game scoring. In isolated life and death problems and for the purpose of GoLD this difference between unconditional life and Seki in not taken to account , Seki is to be deemed victory for the defending team.

Life and death problems can also conclude where Ko occurs within the area, in the full game of Go this is very important in deciding ko fights in other parts of the board and can be seen as leverage to be used. For the purpose of GoLD and in isolated problems these situations are further played on until a defining end is met where group is dead or alive. The final possible result is death where the defending player's group is captured without any ko situation which the best-case scenario for the attacking player.

2.5   Related Works

Martin Müller's Explorer [2] was a strong program developed with the intention of playing Go. In his paper he delves into some key aspects of Explorer which have been useful to comprehend ideas behind creating a Go playing software. He describes the importance of board evaluation within a tree search algorithm such as minimax.

Explorer can evaluate certain areas of 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 Go. The victor of Semeai can be determined early sometimes and hence doing so during the evaluation could save searching further during a tree search algorithm. Müller also goes into detail about the his zone based evaluation where Explorer is able to distinguish areas which are in conflict and areas which are clearly controlled by a certain colour.

While Explorer aims to play 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. Explorer's methods of recognising safe strings such as Benson's alogrithm [10] and also the improved version of the algorithm described in [11] were regarded during the intial design of GoLD.

Work done By Ken Chen and Zhinxing Chen [11] in creating Go Intellect and HandTalk were also quite interesting and relevant while working on this project. The key aspects of their work regarding eyes and determining value of a shape depending on the number of eyes the shape can produce was great a motivator during the implementation of GoLD's pattern based heuristics.

Another relevant work researched during this project was Adrian B. Danieli's paper [12] on his own TsumeGo problem solver. In his paper he explorers indepth the process in which his program was delevoped. Explanation of key concepts such as move generation and board evalution were helpful in understanding how to delevop my own work. A particularly relevant piece of information which was used as advice during the desgin of GoLD was that there was a problem in his representation of each node during the tree search. He believed his representation of each node caused extra workload without during tree search without a good reason to do so. He goes on to explain due to the depth–first nature of his program "compressing" and "decompressing" the board posistions for every node was waste of time. Rather than saving some space which is not as relevant when using a depth first search, it would have been better to save time by storing the entire board posistion for each node.

3 | Requirements

The project needed a well-defined list of requirements to outline key features and help to with the implementation order of those key features. Time constraints and the resources available had to be taken to consideration before further work on the project was done. From the start the project was divided into to two major milestones each containing a set of requirements to achieve the milestone. These milestones were set to help evaluate the progress through out the project and also to help streamline the work flow throughout the project so that at all points there was goal to strive towards. The functional requirements identified earlier on the project captured most of the work required but throughout the project these requirements were refined and added to in order to solve unseen problems that arose during the project duration. Some non-functional requirements were also discovered earlier on which were used to refine the goals of the project.

3.1 Functional Requirements

3.1.1 Milestone 1

The project's first milestone's aim was to implement most of the key features required for users to create and view problems. In addition to this the aim consisted of enabling the computer to play moves according to a simple tree search. This milestone consisted of the following requirements.

Go Board The first milestone required a Go board which the user and the computer will be able to interact with. This meant to implement a Board object was capable of storing relevant data and contained methods which is able to process the relevant data.

The relevant data needed to include the following:

- Representation of all the points on the board
- Current turn
- All of the stone strings for both colours
- Colour of keystone
- Point of Ko
- Lists of Atari points for both colours
- List of valid moves for the current turn

The relevant methods which are able to do the following were required:

- Make a move on the board and update the entire board accordingly
- Generate all the valid moves for the current turn
- Check if a move is valid – this should include self-capture check and Ko check
- Check for capture and remove stones accordingly
- Obtain all the liberties of a given stone string

Play Mode Play mode is the screen in which the user can load a problem up and play through it. Play mode needed to contain a Go board in order to display problems. The board itself needs to be interactable, the user should be able to click on valid points on the board to make a move.

Play mode also needed to contain buttons which do the following:

- Load a problem
- Save a problem
- Switch to editor mode
- Tell the computer to start search for the best move of the current board position
- Tell the computer to stop search
- Reset the problem to how it was loaded
- Undo Redo moves
- Switch Turns
- Pass the turn

Editor Mode Editor mode is the screen in which the user can create problems from scratch or edit problems loaded in and save them for future use. Similar to play mode, editor mode needs to contain a Go board in order to display the problem which is being created/edited. The board itself needs to be interactable, the user should be able to place stones by simply clicking on the points on the board. Editor mode also needed a way for the user to decide which stone is currently being placed or whether if stones are being removed when the user clicks on the board.

Editor mode also needed buttons to do the following functions:

- Load a problem
- Save a problem
- Switch to play mode
- Set who plays first during the problem
- To reset the board into an empty state
- Rotate and flip the board
- To place a keystone
- Switch keystone colour

Computer To achieve the first milestone the computer was required to perform a tree search. The tree search chosen for this project was the Alpha-Beta pruning algorithm. The requirement for the computer was that it needed to produce the best move possible given infinite time and or small enough problem via searching every valid sequence of moves then interact with the board in play mode by making the best move found.

3.1.2 Milestone 2

The second milestone's aim was to enhance the computer AI by introducing heuristics to allow the user to restrict the time the computer is able to search for the best move. To do this following the requirements were added on top of the requirements for the first milestone.

Play mode required an improved interface containing additional buttons to configure the parameter of the Alpha-Beta search used by the computer. Within the play mode the user needed to able to toggle on and off depth limiting and breadth limiting. Play mode should also allow the user to increase or decrease the limit and the breath of the search as they desire.

The computer's requirement was that it should be able to limit depth of the search according to the user's choice. To do this it was required to introduce a board evaluation function. In order to allow the computer to limit the breath of the search it was required to introduce a move generator function which was able to generate a limited number of moves depending on the user's breadth choice.

3.2 Non–Functional Requirements

Throughout the project some non-functional requirements where discovered and needed to be addressed , following categories were made to specify the different criteria GoLD need to achieve.

Performance / Time - Important concern was the time factor a user wants to spend during the process of solving problems using GoLD.

Usability - Another concern was the ease of use , whether the program is simple and understandable for untrained users.

Reliability – The program should be able to cope with any kind of user inputs and should maintain running with any failures or crashes.

In order to test all three categories of these non-functional requirement the project also had an additional requirement of producing a folder of problems which can be used during the evaluation phase. This folder of problem was required so that beta tester could have pre-defined problem to test the program on. The folder was also required in order to perform quantitate evaluation of GoLD during the evaluation process, to judge what level of difficult the program can solve under what length of time period it takes.

4 | Design

4.1   Tree Search

GoLD allows the user to play against the computer which is one of the main requirements of the project. To create a computer which is able to play against a human, GoLD must be able to determine the best move out of all the valid moves in the given board position for the current turn. The simplest way to achieve this is by searching through the game tree and determining the end result of each valid move sequence of moves. The game tree consists of every single board position that is possible to occur from the current board position. Board position refers to the state of the entire board, where each stone is on the board , whose turn it is and also if there is an intersection of the board where the Ko rule is applied. Each subtree of the game tree is the result of making a valid move in the current board position. To determine the best move, every sub tree of the current board position needs to be explored until the resulting board position is a game ending state. This type of search where every sequence of moves is processed is referred to as a brute force search. The ability to "brute force search" Go's game tree is essential to have in GoLD as it is the foundation to which rest of the program's AI is built upon.

4.1.1   Minimax

The basis of finding the correct move within GoLD is the Alpha-Beta pruning search algorithm which is based on the Minimax search algorithm. In order to understand Alpha-Beta, it is essential to understand Minimax. Minimax is based on the idea of taking turns, and fundamentally how one player is trying to maximise their own score and the other player is trying to minimise the opponent's score by increasing their own. Minimax is perfect for searching Go's game tree where two players are playing, and one player's gains are directly linked to the other player's losses. A simple implementation of minimax with no depth limit will search each valid move of the initial board position until a gaming-ending state occurs. Minimax will choose a move which results in victory even if the opponent plays perfectly if there is such a move. In terms of Go, if it is currently black's turn then minimax will start the search as black and during the search will alternate between playing as black and playing as white until a game-ending state is reached. Each subtree of the initial game tree is searched until either a subtree (move) is found where white cannot avoid defeat or all the subtrees are searched.

GoLD is only concerned with life and death situation in Go and hence it is simple to determine if the current board position is in a game-ending state. If the attacking player has no valid moves and the keystones, marked by the problem creator, remains on the board then the defending player has won. Alternatively, as soon as all the keystones on the board are captured then the attacking player has won. In a real game of Go, results of life and death situations are not so simple. There might be multiple solutions which result in the capture of all the keystones but there might only be one solution which will do so without allowing losses on other parts of the board for the attacking player. This is ignored within GoLD as we are only interested in the result of a life and death problem, and not the outcome of the board. Hence any solution which captures all the keystones are sufficient for the GoLD. Figure 4.1 show an example of minimax for a simple game where the max player tries to obtain the highest value and the min player tries to obtain the lowest value. Note the game-ending states are shown by a circular shape.
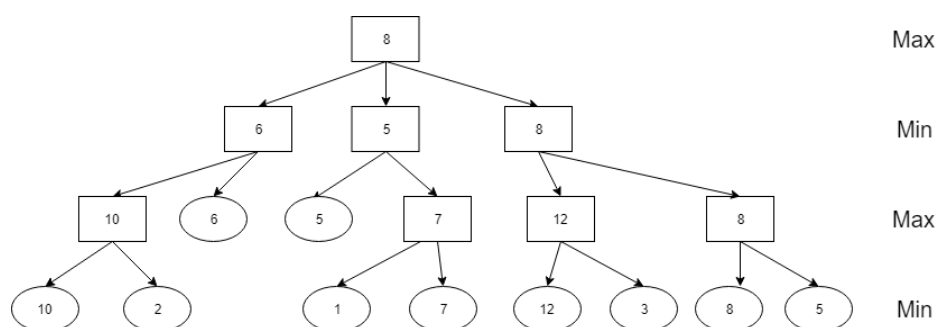
Figure 4.1: Minimax used for simple number game. Best score the maximizing player can get is 8. Even though scoring 10 and 12 are possible if the minimizing player plays optimally these scores cannot out be achieved.

4.1.2   Alpha–Beta Pruning Search

The Alpha–Beta pruning algorithm is a smart improvement on the Minimax algorithm in order to reduce the search time. It allows for the same result to be determined without having to search through as many subtrees as Minimax does. It uses alpha and beta values to determine whether searching a subtree further ahead is a waste of effort or not. The search begins with the alpha value set to –inf and the beta value set to +inf. These values are passed down from the parent tree to each of its subtrees during the search. If a maximizing subtree finds a move which results in a score greater than the previous best score for that board position, then the alpha value of that subtree is increased to the value of the new best score. If a minimizing subtree finds a move which results in a score less than the previous best score for that board position, then the beta value of that subtree is decreased to the value of the new best score. After searching each move within a subtree, the algorithm checks if the alpha value is greater than or equal to the beta value and if it is then the algorithm decides that searching more moves within that subtree is pointless and returns the best score found so far back to the parent tree. These "beta cut-offs" reduces the search space for the algorithm by cutting off all the other moves from being searched. This amounts to a great improvement in search time compared to the minimax algorithm without changing the result. This is because the move that caused the beta-cut off to occur is worse than the parent tree's best move found so far.
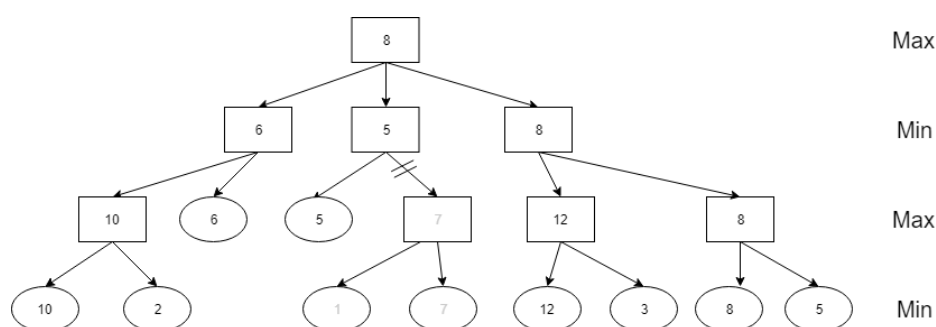
Figure 4.2: The same simple number game as Figure 4.1 but using Alpha–Beta pruning. In the second subtree after searching the first move which returns 5, the algorithm cuts off from any further search in that subtree as it realises the first subtree will always be better than 5.

The following pseudocode addresses the Alpha–Beta pruning search algorithm with Go's Life and Death problems in mind. It was used for the initial design of GoLD's move finder function

and later adapted to implement heuristics described in the following sections of this chapter.

Procedure `alphaBeta`(*board, isDefending, depth, alpha, beta*)**:**
    depth ← (depth + 1)
    validMoves ← (board.validMoves)
    if *keyStones.size = 0* **then**
        return $-\infty$
    if *(board.turn = attacking* **and** *validMoves.size = 0)* **then**
        return ∞
    if *isDefending* **then**
        best ← −∞
        foreach *move in validMoves* **do**
            board.makeMove(move)
            score ← alphaBeta(board, !isDefending, depth, alpha, beta)
            board.undoMove()
            if *(depth = 1* **and** *score < best)* **then** *bestMove ← move*
            best ← max(best, score)
            alpha ← max(alpha, score)
            if *beta ≤ alpha* **then** *break*
        end
        return *best*
    else
        best ← ∞
        foreach *move in validMoves* **do**
            board.makeMove(move)
            score ← alphaBeta(board, !isDefending, depth, alpha, beta)
            board.undoMove()
            if *(depth = 1* **and** *score > best)* **then** *bestMove ← move*
            best ← min(best, score)
            alpha ← min(alpha, score)
            if *beta ≤ alpha* **then** *break*
        end
        return *best*
    end

**Algorithm 1:** Alpha Beta Pruning Search

4.2 Heuristics

The Alpha–Beta pruning search algorithm by itself is a great improvement over the standard Minimax algorithm but even with this improvement, the sheer size of search space for a game tree can become quickly overwhelming for the search to handle in a reasonable time. Alpha–Beta even at best case has a complexity of $O(\sqrt{b^d})$ and at worst, $O(b^d)$ which means for problems with a higher number of initial valid moves or problems that require to search further ahead to come to a game-ending state could take an endless amount of time to search. For this reason, we have to introduce heuristics to allow the search to come to a quicker solution which could be incorrect rather than taking a long time to find a guaranteed correct solution. There are quite a few methods of implementing heuristics within the Alpha–Beta search, GoLD will be able to use few of these in combination or by themselves to restrict the time taken for the computer to respond to the user.

4.2.1 Depth Limited Search with Board Evaluation

A non-depth limited Alpha–Beta search will search deeper and deeper until a terminal state is found, this is very impractical for larger problems. Note that "depth" and "deeper" refers to how many moves ahead computer searches, the higher the depth the further ahead the computer

has searched. Best way to deal with this is to introduce a depth limit where once depth X is reached the search is not continued. Of course, the problem with this is that when these depth cut-offs occur the board position will not be in a game-ending state and hence the Alpha-Beta search needs to do some type of evaluation in order to return a score which is between the two-opposing game-ending state of being dead or alive. The Alpha-Beta search will have to evaluate the board position when the cut-off depth is reached and return a value which represents how good the board is for the players. For example, if the board position will lead the attacking player to capture all the keystones then the board evaluation should return –5000, on the other hand if the board position is in favour of the defending team then it should return 5000. Creating a function which evaluates the board accurately is a difficult task and doing so which can perform at the level of humans is even more difficult. An important factor to consider when designing an evaluation function is the time it takes to evaluate a board position. More complex the evaluation function becomes the longer it takes to evaluate single a board position hence the more time it takes to evaluate the board position every time the depth limit is reached.

Pattern Matching Evaluation When evaluating a board position GoLD looks at the board similar to how humans would look at a board. It tries to identify general patterns on the board to give values depending on the pattern it finds. These patterns are based on the idea of eyes and eyes space. Identifying patterns which will lead to multiple eyes will be evaluated to give a very high score and hence the search will be able to recognise a group of stones which will be able to live without further searching. The general principle behind the patterns used in GoLD is to identify the number of eyes the pattern can yield when fully played out and values the pattern according to this factor. The values for each pattern found is added to the overall board score.

(a) Before *(b) After*
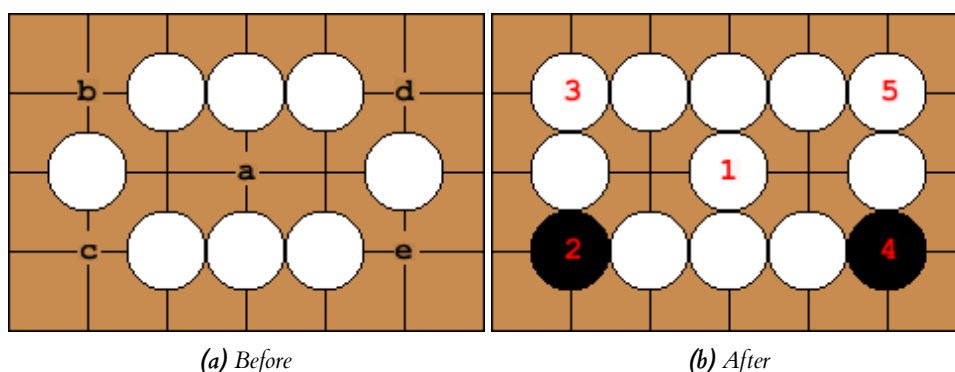
Figure 4.3: *Straight Three in the middle of the board with missing corners.*

Most of the patterns used within GoLD have a distinct feature which is common in life and death problems which is that depending on who plays first the pattern/shape becomes dead or alive. In other words, these shapes are unsettled and hence contains a vital point which will determine the number of eyes that can be produced by the shape. Figure 4.3a shows a pattern used in GoLD, the shape referred to as Straight Three but with its corner stones missing. The vital point of this pattern is a, if white is able to play at a then the pattern is almost ensured to create two eyes unless white makes a mistake filling in the corners. The corners b, c, d and e are also important for this pattern if black is able to have stones in both b and c or in d and e then the wall of shape becomes weak and hence the shape can become captured. Due to alternating play if white plays first in this situation then it will be played out as shown in Figure 4.3b where white is able to produce two real eyes.

An important insight into creating a better evaluation function is to recognise that patterns like the one mentioned above will be able to create a single point eye at the worst-case scenario. If the opponent is able to play at a pattern's vital point and restrict it from creating 2 eyes, the pattern will still be able to create 1 real eye which should be valued quite highly. By connecting two

(a) Before   *(b) After*
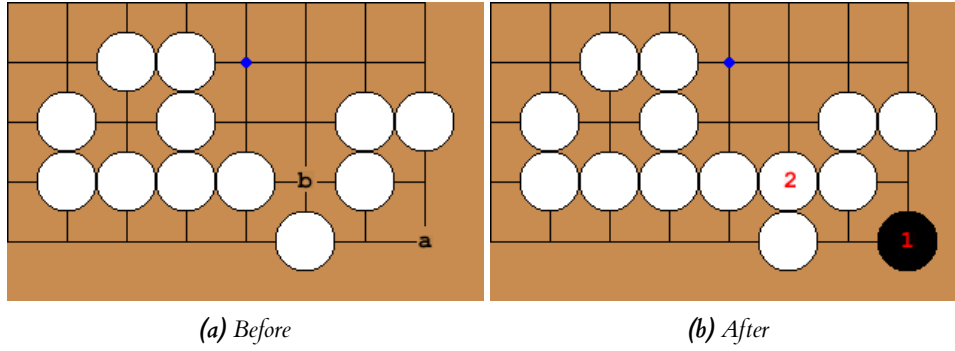
Figure 4.4: *Bent Three in the corner with a Single Point eye nearby to connect.*

groups each containing a 1 real eye the defending player can achieve a live shape. When searched patterns fail to meet 2 eyes, GoLD stores the eye space of each failed pattern in a collection of eye spaces. This is because each of these eye spaces is highly probable to contain one real eye. Once the whole board is searched for all the different patterns used within GoLD and each identified pattern's values totalled, the evaluating function is left with the collection of eye spaces which all contain one real eye. Using this collection of eye spaces GoLD is able to group together multiple eye spaces which are connected through stones of the same colour. Any group of eye spaces that contains two or more unique eye spaces are identified as highly valuable due to fact that they will most likely produce two real eyes.

Figure 4.4 shows an example of where GoLD is able to recognise a new shape with 2 real eyes. Bent Three in the corner is about to lose its vital point once black plays at a but white can connect to the Single Point Eye nearby by playing at b. This creates a new shape which contains 2 real eyes. GoLD recognises that the Bent Three in the corner will create one real eye even though it has lost its vital point. When white connects at b, GoLD's eye space grouping algorithm identifies that the new shape contains at least 2 real eyes and hence will score the shape highly.

(a) Black Hanes at 1   *(b) Black Cuts at 1*
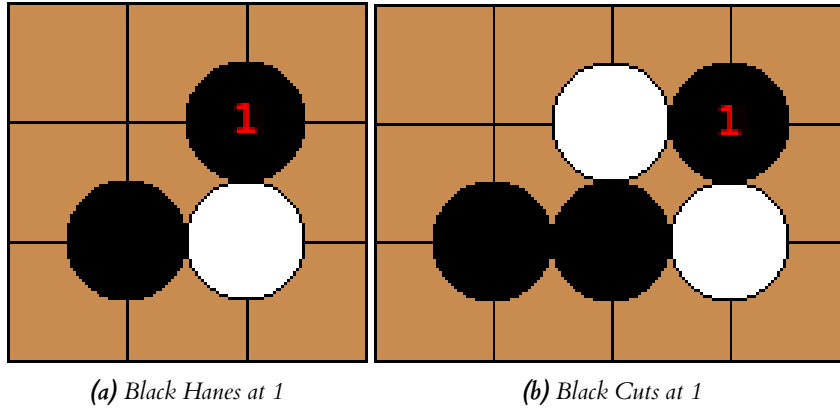
Figure 4.5: *Important type of moves*

To limit time spent on board evaluation only a small number of unique patterns can be searched. Due to this constraint, it was important to identify the essential shapes/patterns which have the ability to create eyes. Fifteen different patterns ranging from a Single Point Eye to a Rabbity Six along with all their variations on the edges and corners of the board are used within GoLD. Along with these patterns, the evaluating function also tries to identify stones which are placed in good positions with respect to the opponent stones. The evaluating function looks for areas of

the board that are the result of a cut, hane or even simply connecting two groups of stones and awards values accordingly.

 Figure 4.5a shows a hane which is a move where a stone is placed next to an opponent stone which is already in contact with one's own stone. Hane is a move which plays around one or more opponent stones. Note: playing at the opposite side of white here is not a hane. Figure 4.5b shows a cut where black is able to stop white from connecting the two white stones by playing at 1.

4.2.2   Move Ordering

The benefit of Alpha-Beta pruning search over simple Minimax search is that when cut-offs occur a great number of subtrees of possible board positions do not need to be searched. Erik van der Werf talks about the importance of move ordering during Alpha-Beta search and the effects of good move ordering heuristics [5]. If the best move is always selected to be searched first, then the algorithm will produce cut-offs when all the other moves are searched afterwards. The idea behind this is if the best move is found first then all other moves will never return a score which is better than the best move. With this in mind, we can see that ordering the moves list to consider the best move first or even a move which will produce more cut-offs first, is a great way to reduce the number of board positions to search without altering the result.

Killer Move Heuristic GoLD uses a deeply researched [6] method, Killer Heuristic, introduced by Huberman, B.J. [7] to order moves when searching through endgames of Chess using AB. Huberman's theory was that a move (killer move) which led to a better board position from initial board position A will also lead to a better position from a similar initial board position B if the move is a valid move for B. Using this theory, these killer moves should be searched first when a similar board position occurs during AB search. The moves which are considered killer moves are defined easily by using the cut-off characteristics of AB search, during the search if a move causes a beta cut-off to occur then this move can be considered a killer move and hence will be stored to be used for move ordering. We can define a similar board position simply by looking at the depth in which the board position occurs in. Hence an ideal way to implement the killer heuristic is to store a number of moves for each depth which caused beta cut-offs to occur. Storing a great number of moves for each depth can lead to increased work during the move ordering process hence it is most effective only to store a few moves. GoLD only stores two killer moves k1 and k2 for each depth.

Move Ordering for a board position A which occurs at depth d consists of searching the list of valid moves for k2 of depth d and if k2 is a valid move in this board position then k2 is ordered to the top of the valid moves list. After which, the same is done for k1 of this depth(k1 is searched second in order to give k1 priority over k2). When a cut-off occurs at depth d then the k1 of that depth becomes k2 and the move that caused the cut-off to occur becomes the new k1 of depth d.

4.2.3   Alpha–Beta with Iterative Deepening

An issue with AB search is that if the best move is found at depth d then searching past depth d for the other branches is always a waste of time. For example, black is to play and kill white. Moves A to E are the valid moves of the current board position and the moves will be searched in alphabetical order. If D is the best move and will result in a victory for Black at depth 3 then searching past depth 3 for move A, B and C is a waste of time. Without searching D first there is no way to obtain this information hence A, B and C will be searched to deeper depths. Iterative deepening is an effective method to work around this problem. Alpha-Beta with Iterative Deepening is a process in which AB search is performed multiple times and at each iteration, the depth limit is increased by 1. This cycle takes place until max depth limit is reached, or a favourable game-ending state is found (in the example it would be white dead). This

effectively alters AB search to only increment depth by 1 at a time and hence solving the problem of searching moves to deeper levels than not required. The drawback of this process is that it requires multiple iterations of AB search and will have to perform redundant computation for the lower depths of the game tree. Work done by Korf R. [8] shows that redundant computation does not affect the overall time of the tree search as much as it would seem. In fact, using the result of each iteration to perform the next iteration can reduce the time taken to perform the next iteration and possibly the overall run time of the search. Each iteration will return the principal variation(PV) which is the sequence of moves which is considered to be the best determined by the search. The PV of the previous iteration will be set as the first sequence of moves the next iteration searches. This is another form of move ordering which effectively improves overall performance compared to a normal fixed depth limit search. Due to the depth-first nature of the depth limited AB search, the search will not provide a result until every move is searched to a game-ending state or to the limited depth. This means the search will not be able to produce the best move searched so for if it is stopped in the middle of searching. Without the introduction of Iterative Deepening in GoLD's search the user is not able to search for a limited amount of time, rather the user would have to predict which depth limit to set in order for the search to complete in the time they desire. Predicting the depth limit could lead the user to set the depth limit too high which would lead the search to last longer than what the user would like. Iterative deepening fixes this issue by allowing the user to stop the search whenever they desire and to use the result of the last fully performed iteration to make the best move found so far hence improving the usability of the GoLD.

4.2.4   Move Generator

GoLD allows the creator of the problem to set the valid points of play on the board and any points on the board which contains a stone when captured become a valid point of play within the problem. A high number of valid moves to begin the problem with creates extremely large game trees which would take too long to search even with the best heuristic move ordering. The number of valid moves can be regarded as the breadth of the game tree. In order to limit the breadth of the game tree GoLD introduces a move generator. The goal of a move generator is to pick out x number of moves from all the valid moves of the current board position in order to cut down the number of moves to be searched. A perfect move generator will always pick the best move if it could only pick one move to generate but this is impossible to achieve without searching ahead. The next best result is that the generated moves list has a high chance of containing the best move and this chance obviously increases with the number of moves generated. The generated moves list should contain the best move possible otherwise the best move will not be searched hence the search will never return the correct result. If more moves are generated for use, then there is a high chance of one of them being the correct move but then more moves have to be searched. In order to maintain a low number of generated moves and still have a high chance of containing the correct move, the move generator within GoLD uses different types of heuristics to determine which moves have a high chance of being the correct move and which moves are irrelevant. One approach to determine which moves are good moves is to simulate each move being played and then evaluate the board to see which move will lead to an immediately better board position. While this could be an effective approach, it will also increase the worked load to process each board position. This might, in turn, slow the search down rather than speeding it up. The approach which GoLD uses is to simply process the board position as it is and then pick out moves which are relevant and give them each a score indicating how relevant they are. Then the top x number of relevant moves are generated for search. One heuristic used to process the current board is a simple distance-based method. It relies on the school of thought that the closer a move is to the key group of stones the higher relevancy they have. Within GoLD, the search is aware of the key group of stones (marked by keystones) hence the distance-based heuristic will give a high relevancy score to any moves 1 step away from

the key group then a lesser score to moves 2 steps away and so forth. This process is used for moves up to 3 steps away from the key group as the relevancy of moves further away is minuscule and hence a waste of time to process. This heuristic also accounts for enemy stones which are blocking the key group's access to other points hence in situations like the one in Figure 4.6b where points that are blocked by the surrounding white stones are considered irrelevant.
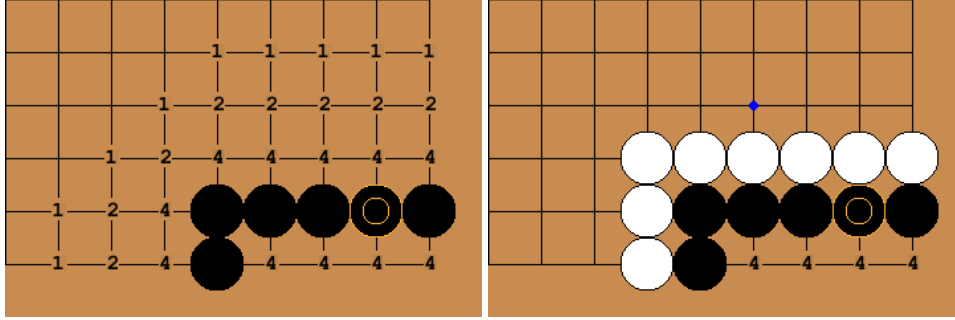
(a) Value is halfed each step away from the key group   *(b) White stones devalues moves outside the wall*

Figure 4.6: *Distance Based Relevancy*

The other heuristic method used for move generation is based on pattern matching. Similar to pattern matching for board evaluation, here we use pattern matching to identify relevant moves. The same patterns used for board evaluations are also used to identify relevant moves on the board. The move generator searches for patterns on the board and increments the relevancy score to any moves which are vital points within a pattern found. This will allow the move generator to consider more complex shapes on the board and generate moves according to these shapes. In general, moves found through pattern matching are given higher relevancy score than ones found through the distance-based heuristic. The flaw in this heuristic is of course only a limited number of patterns can be searched because increasing the number of patterns will increase time taken to generate moves hence the same crucial patterns used for board evaluation are used for the move generator.
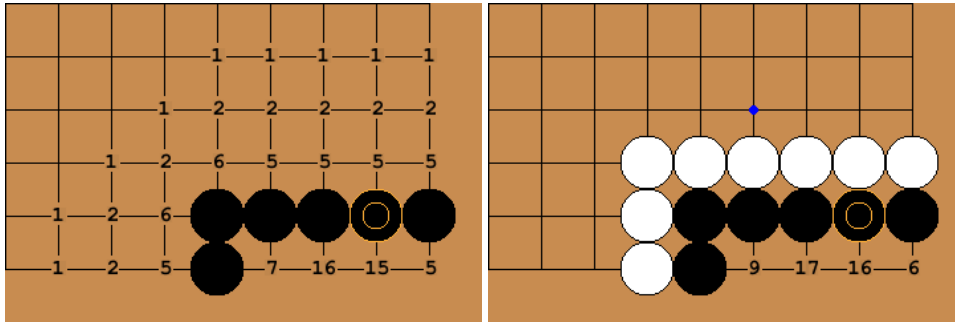
(a) Not surrounded by white stones   *(b) Surrounded by white stones*

Figure 4.7: *Distance & Pattern Based Relevancy Combined*

Relevancy score from the distance-based heuristic and pattern-based heuristic are combined together for each valid move and is ordered into a list from the highest to the lowest relevancy. Depending on the user's choice of breadth limit x, the top x number of moves from this list will be generated and used within the Alpha-Beta search. Figure 4.7 shows the same board positions shown in Figure 4.6 but both the distance-based and pattern-based heuristic applied. We can see that relevancy value for the vital points of the key group is significantly higher than all the

other points nearby. The move generator would pick the two vital points as the first two moves to generate which is the best case scenario for the move generator.

4.2.5   Suicidal Move Removal

Keystones are an absolute objective in the program, they are one of the reference 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 the Alpha–Beta algorithm. This entails removing any suicidal moves that decrease the number of liberties of a keystring 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, GoLD only needs to look at the moves that place a stone in liberties of keystrings with 2 liberties, any more liberties would ensure at least 2 liberties would remain intact if a stone was to be placed on a liberty. For a keystring with 2 liberties, placing in one of those 2 liberties does not automatically mean it is a suicidal move for two reasons: Move captures an opponent string; Move would increase or maintain the liberty count. If any of these two reasons are met, then the move is not suicidal.

- Both liberties of the keystring is added to the suicidal move list.
- Moves that places a stone in the liberty of any opponent string which is in Atari is removed from the suicidal move list.
- Moves that connects the keystring to another string which fits the following criteria are removed from the suicidal move list: Must be the 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 that places a stone adjacent to any empty point which is not a liberty of the keystring are removed from the suicidal move list.

After these checks are done for all the keystrings, any moves remaining in the suicidal moves list are removed from the valid moves list – this new list of moves is referred to 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.

5 | Implementation

This chapter will go into detail about how GoLD is implemented. GoLD is entirely written in Java and uses a simple 2D game engine called Slick2D which is based on LWJGL. All of the major constructs of Go are implemented within GoLD to create a stable environment for user to solve Life and Death problems. Some of the key aspects of the program includes the Go board, the Pattern Matcher and the two modes of play. These features were implemented to be able to facilitate GoLD's requirements such as the "computer" which plays against the player. Which is essential a search function greatly dependant on how the Go Board is structured. Many smaller but crucial features had to be implemented in order to create a completely standalone program which can create and solve problems. To enable complex heuristics to be performed in order to evaluate board positions or generate moves which are relevant, a method of creating and identify patterns of stones is required. A pattern matcher which is able to distinguish between middle , side and corner of the board is built into GoLD.

5.1 Board

The Go board is represented as an object class within GoLD which contains all the vital data to represent the current board position. GoLD's board is designed for the purpose of making tree search and board evaluations as quick as possible. This meant storing more information within the board object was ideal instead of recalculating information every time it was needed. The board object also implements public methods which allows the computer and also the users to interact with the board. The board in essence implements the rules of Go and automatically updates the board position once a move is played either by the user or the computer.

5.1.1 Stones

Representation of stones on the board is simple but effective to keep operation of gathering information about strings and liberties quick. GoLD takes advantage of Java's enum type to define the different possible states each intersection on the board could be. While it is possible to represent all the points on the board with simply black , white or empty constants , this would lead to extra operations to distinguish between different types of points such as an empty intersection which is within a problem's boundaries and an empty intersection which is removed from play. Similar operations are required to distinguish between Ko points and Valid points, black stones and key black stones, white stones and key white stones. To bypass this problem GoLD uses 7 different constants to represent all the intersections on the board. The constants used are BLACK, WHITE, VALID, INVALID, KO, BLACKKEYSTONE, WHITEKEYSTONE . The entire board is represented by a 2D array of enum Stone type. While the use of a 1D array is possible to represent a 2D Go board and is more space efficient than a 2D array, it would require extra operations to define the board within a 2D coordinate system. This would lead to an extra operation each time the board is accessed which would in essence slow down the entire program.

5.1.2   Strings

We can define a string of stone as any number of stones of the same colour which are connected through adjacent stones of the same colour. A key observation of strings is that once a string of stones is formed , the only way to deform it is through capture by placing on all the liberties of the string. Hence the board object maintains two collection of strings , one for black and one for white, for the use of the capture functionality. Both collections of strings are updated after each move to represent any changes on the board. Each string is represented by a list of 2D coordinates, due to the lack of an inbuilt tuple type with Java, GoLD uses a custom-built Tuple object to represent 2D coordinates.

5.1.3   Capture

To process capturing of opponent stones, the two collection of strings are used to iterate through each string inside the collection of opponent strings and determine whether any opponent strings lack liberties and then same is done for one's own strings. If an opponent string has zero liberties, then each of these stones are replaced by a VALID in the 2D array and the string itself is removed from the collection of opponent strings. Same is applied for one's own strings but if a capture occurs then the move is invalidated under the self-capture rule. During this process two additional information is gathered about the board. For every opponent string if the liberty count is one then the string is considered to be in Atari and hence the intersection of liberty is added on to the opponent's Atari list which consists all of the opponent's single liberties and the same applies for the current players strings. Two Atari list ,one for each colour, is maintained within the board and is updated during the process of checking for capture. The other additional information gathered is whether the Ko rule should be in play the next turn. While checking the opponent's strings for capture, if a string containing one stone is captured then the intersection of that one stone is in consideration for Ko. After the checking for capture, a simple test for Ko is occurs. If there is an intersection in considerable for Ko, then the stone placed this turn is checked for the following:

- Is the stone in Atari?
- Is the stone not connected to any stones of the same colour?
- Is the liberty of the stone the same intersection as the one in consideration for Ko?

If true for all three questions, then a KO is placed at the intersection which was in consideration for Ko.

5.1.4   Valid Move Checker

Valid move check is a simple function which returns true or false depending on whether placing on intersection being checked is valid for the current player. This function consists of three core checks. Is the point being checked on the board ? This is in the case of a human player trying to place a stone outside the board grid area. Is the value of that position in the board' 2D array VALID? This is multifunctional as it restricts the user and the computer from playing on points of the board which are removed from play during a problem represented by INVALID and also restricts play on the point of KO. The third check is to makes sure that if the current player plays on the point it will not lead to self-capture. To check for self-capture can be intricate , the following algorithm is used to determine whether a move is considered self-capture hence

invalid.

Procedure selfCapture(*point*)**:**
    if *enemyAtariList.contains(point)* **then**
        return *true*
    adjacents ← *getAdjacentPoints(point)*
    if *(selfAtariList.contains(point))* **then**
        foreach *adjacent in adjacents* **do**
            if *(adjacent is currentColour)* **then**
                adjancentString ← *getString(adjacentPoint)*
                if *(getLiberties(adjacentString).size > 1)* **then return** *false*
            else if *(adjacent point is empty)* **then**
                return *false*
            end
        end
        return *true*
    else
        foreach *(adjacent in adjacents)* **do**
            if *(adjacent not enemyColour)* **then**
                return *false*
        end
        return *true*
    end

Algorithm 2: Self-Capture Check

The valid move checker plays an integral part during the tree search which the computer uses to determines the move to play. Every time a new board position is visited by the search , the Alpha-Beta algorithm requires the list of all valid moves in order to generate moves using the move generator and then continue the search by trying out moves generated. To create the list of all valid moves, every point board is checked by the valid move checker and if a point is valid then it is added on to the list of valid moves. To reduce redundant processing , the valid moves list is maintained within the Board object and is updated after every move.
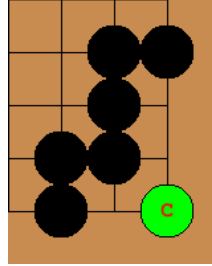
5.2 Pattern Matcher

The pattern matcher within GoLD is able to identify a pattern at the 16 possible variation which can occur. A pattern can be rotated at 45° on the board to create 8 different rotations of the same pattern and each pattern can have the colours as normal or inverted hence 16 variations. Specific to the Go board some points are regarded differently to other points and hence the pattern matcher should be able to distinguish between these. The pattern matcher within GoLD is able to define and identify patterns which are is on the edge, corner or the middle of the board. Note that pattern for the middle of the board can be regard as pattern that is allowed to occur anywhere on the board as along as the pattern shape is occurs as defined by the pattern.

5.2.1 Defining Patterns

GoLD uses a limited number of predefined patterns which are used within its heuristic pattern matching methods. These predefined patterns are written in a simple text notation which are all translated into lists of "Point" objects during the initialisation of static variables . A pattern is just a list of Point objects and these Points are the key components the pattern matcher uses to identify patterns on the board. A Point consists of few fields which allows the pattern matcher to relate each Point to another within a pattern. These fields include two integers which is able to relate each Point inside the pattern to the starting point of the search. The other fields include boolean variables to check whether a Point is on the edge, is on the corner, is defending colour or whether it is a wildcard (always matches). The text notation which is used to translate string

to a list of Points representing a pattern uses each of characters in the string to create a Point object. Figure 1 shows a Bent Four Corner pattern represented by text notation and a visual representation of the pattern within GoLD. Note here the green stone marked with C shows that in the pattern requires that point be a corner and any colour of stone.

(a) *"xrxldxdxlxdxrr#"*

Figure 5.1: *Visual representation of a pattern*

5.2.2   Matching Patterns

The patterns matcher itself is based on the idea that every Point object within the pattern is relative x ,y distance from the starting point of search. Hence the first point of the pattern is always a stone at x and y = 0. The pattern matcher depends on three parameters, the list of Point object which represents the pattern to look for, a list of intersections all containing stones as starting points for the pattern matcher and the colour of the defending team. Simple pseudocode for pattern matching algorithm before enchancements is shown below.

Procedure `matchPattern`(*startingPoints*, *pattern*)**:**
    matches ← init list of Tuple lists
    rotations ← init list of 8 rotations
    foreach *stuple in startingPoints* **do**
        matchTries ← init list of 8 Tuple lists
        skipList ← init list of 8 booleans initialised false
        foreach *Point in pattern* **do**
            if *(skipList.allEquals(true))* **then** *break*
            checkingTuple ← *stuple* + (*Point.x*, *Point.y*)
            foreach *rot in rotations* **do**
                if *(skipList[rot]=true)* **then** break
                t ← *checkingTuple.apply(rot)*
                if *Point.matches(t)* **then**
                    matchTries.get(rot).add(t)
                else
                    skipList[rot] ← *true*
                end
            end
        end
        foreach *tupleList in matchTries* **do**
            if *(TupleList.size() = pattern.size()* **and** *!matches.contain(TupleList))* **then**
                matches.add(TupleList)
        end
        return *matches*
    end

Algorithm 3: Pattern Matcher

The pattern matcher not only returns matched patterns but also the rotation in which pattern

was found in. GoLD uses the matched tuples and rotations to find vital points within patterns found and check whether these points are empty, contains enemy stones or ally stones to evaluate the value of the overall pattern found. After which more complex checks are performed to see if the pattern identified is fully formed or incomplete , whether the walls of patterns are safe. Figure 2 shows Pyramid Four pattern incomplete next to a fully complete variation of the same shape and hence the vital point at a is the only point that matters within this pattern.

(a) Incomplete & unsafe walls   *(b) Complete & safe walls*
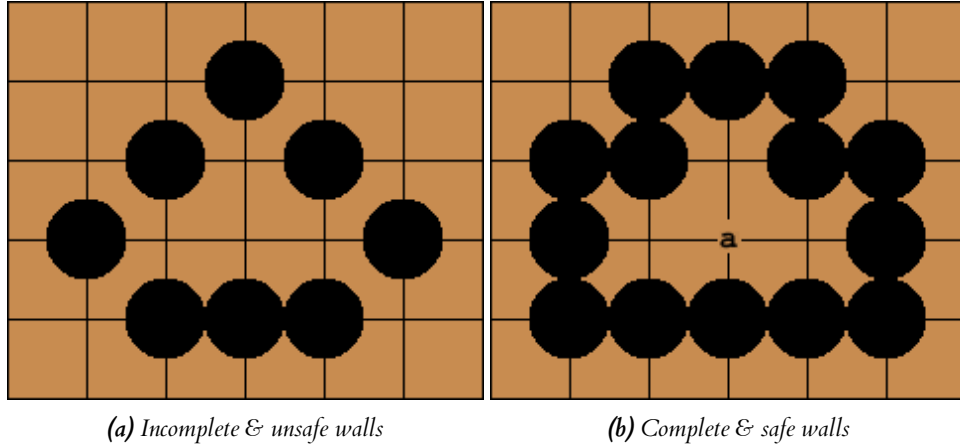
Figure 5.2: *Pyramid Four pattern with and without corner stones*

5.3   Modes

5.3.1   Key Features

5.3.2   Differences

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

6.3.1   Questionnaires

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 EVAL-UATION IN COMPUTER GO. ICGA Journal, 25(4), pp.219-228. 3.Chikun, C. (1993). All About Life & Death. Ishi Pr. 4.Davies, J. (1996). Life and Death. Ishi Pr. 5.Cornelis Diederik van der Werf, E. (2004). AI techniques for the game of Go. [Maastricht]: UPM, Universitaire Pers Maastricht. 6.Akl, S.G., & Newborn, M. (1977). The principal continuation and the killer heuristic. ACM Annual Conference. 7.Huberman, B.J. (1968). A PROGRAM TO PLAY CHESS END GAMES. 8.Korf R. Depth-first iterative-deepening. Artif Intell. 1985;27(1):97-109. doi:10.1016/0004-3702(85)90084-0 9.https://deepmind.com/research/alphago/ 10. Life in the Game of Go - DAVID B. BENSON* 11.https://webdocs.cs.ualberta.ca/ mmueller/p-s/gpw97.pdf - recognise secure territories 12.https://webdocs.cs.ualberta.ca/ games/go/semi-nar/2002/020703/ld.pdf - Static analysis of life and death in the game of Go Ken Chen 13.https://dspace.mit.edu/bitstream/h MIT.pdf?sequence=2 - A Tsume-Go Life & Death Problem Solver Adrian B. Danieli