University of Glasgow | School of Computing Science

Honours Individual Project Dissertation

Go - Project

Jude Haris

May 2019

Abstract

Every abstract follows a similar pattern. Motivate; set aims; describe work; explain results.

“XYZ is bad. This project investigated ABC to determine if it was better. ABC used XXX and YYY to implement ZZZ. This is particularly interesting as XXX and YYY have never been used together. It was found that ABC was 20% better than XYZ, though it caused rabies in half of subjects.”

Education Use Consent

Contents

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 Silver et al. (2016) 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.

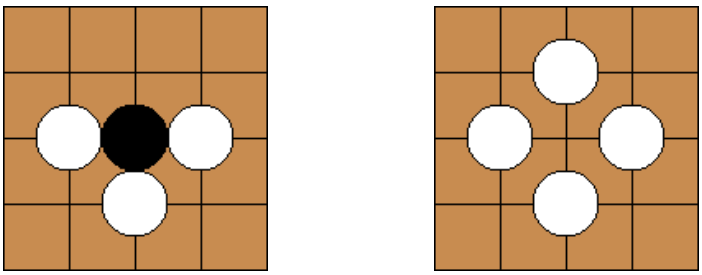
Chapter 7 Conclusion – Concludes the whole paper by looking at the successes of the project, places of improvements and any possible future work.

2 Background & Related Works

In order to achieve the aim of creating a tool program, a 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 was important for the progression of 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 in general like how a beginner 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 and translated by James Davies Kiin were used to define the basic ruleset used within the project. Some alterations were made to these rules to suit the problem-solving nature of the project rather than playing a whole game of Go which will be explained in a later section.



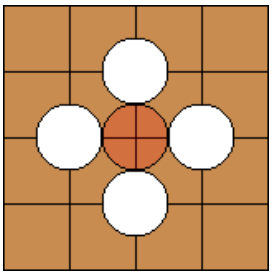
(a) *Before Move* ***(b)*** *After Move*

Figure 2.1: *The black stone is "captured" when the white stone at the top is placed because the black stone loses its last liberty*

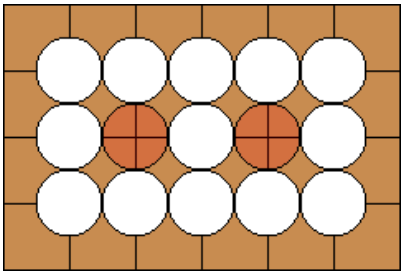
The basic rules of Go are as follows:

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 a 19x19 gridline pattern with 361 intersections. Stones are played on the intersections and not the square in between. Stones can only be played on empty intersections. These intersections are also referred to as points on the board.



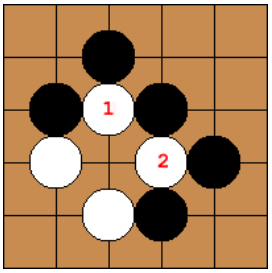
(a) *Middle point is invalid for black to play*



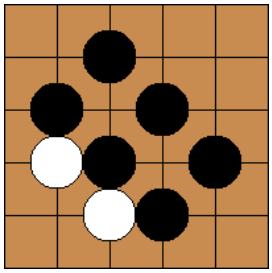
(b) *Both middle points are invalid for black to play*

Figure 2.2: *A black stone cannot be placed in the middle of both of these shapes because it 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 an empty intersection of their choice.
   (b) Then all opponent's stones that do 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 the opponent's stones takes precedence over the self-capture check. This allows for moves which capture opponent's stones but is suicide if the opponent's stones are not removed first. See Figure 2.3.



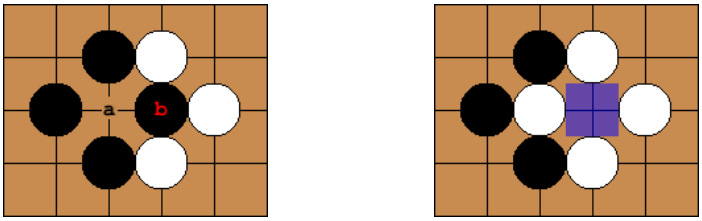
(a) *Before Move*



(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.*

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 an 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 more territory wins. (Scoring can be complicated but is irrelevant to understand the basic rules)



(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, a black stone cannot be played at b next turn by the opponent to capture the white stone placed at a.*

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 to 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 position.
2. Not all 361 intersections can be played on – boundaries are set by the creator of the problem
3. For the purpose of these problems, one player is considered to be the attacker and the other to be the 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 do not end the game/problem.
6. The problem is solved when the attacking player has no valid moves and cannot pass, if the keystones are still on the board then the defending players have 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 :

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

The reason behind the 4th alteration to the basic rules is to restrict infinite passing which would be 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 is invalid to play on due to the Ko rule, this resolves some rare occurrences where boundaries set by the creator of the problem restricts the attacking player from winning.

2.2.1 Boundaries

A normal game of Go is played on the a 19x19 board where every intersection of on the board is playable if the rules of the game allow it. The implications of the size of the board for this project and in computer Go in general is that the search space is too large to obtain good results in a short period of time even with a limited depth search.

To achieve the aim of this project there was 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 computer would be too large to determine the best move in a limited time. While setting boundaries is great to enable the computer solve more problems, the drawback of adding boundaries is that problems themselves become substantially easier compared to the same problem with no boundaries Setting boundaries is limiting the options given to the computer hence increasing the chance of it finding the correct answer. For the computer to solve a difficult problem with no boundaries and therefore have more than 300+ moves to search through is an unachievable task using standard search algorithms. In comparison for the computer to solve the same problem with boundaries which limits the possible moves to less than 20 is a much simpler task which could be done in a reasonable time.

To set boundaries during the creation of problems, the creator can deem which points are relevant for the problem by placing "valid points" on the board. Any points on the board with preplaced stones are also regarded to be within the boundary. For example, a black stone is captured, the empty point remaining will be regarded within the boundary of play. Ideally the creator of the problem should distribute "valid points" in a way to maintain the same level of difficult for human player but also limit the number of points for the computer to search through. It can be very difficult or even impossible for a problem creator to determine all the 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 a feasible time.

2.3 Go Terminology

Throughout this dissertation many terms are used to describe various aspects of Go, some of which are commonly used term within the Go world and others are created for the purpose of this 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.

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

Board position – Refers to the entire state of the board, not just a single point on the board.

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

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 is connected to both stones.

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

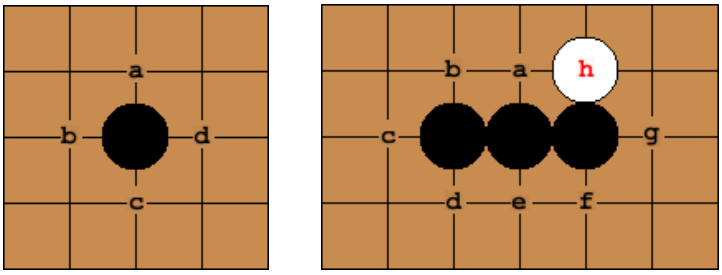
Keystring – Any string which contains one or more keystones.

Liberty - The liberties of a stone are all adjacent empty intersection of the stone and all adjacent empty intersection of any stone connected to the original stone. See Figure 2.5.

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

Atari Point – The last remaining liberty of a string.

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



(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.*

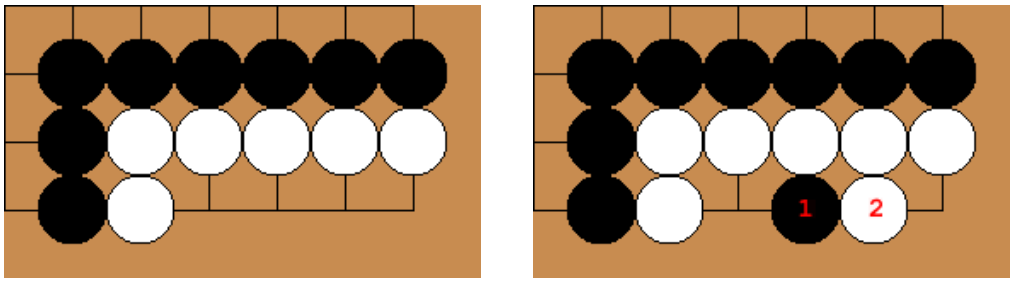
Seki – Two strings of opposing colour that are alive next to each other. Neither player can capture the opponent string without first sacrificing their own string.

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 attack and capture enemy stone strings or to defend and keep alive ally stone strings. Situations on the board where it becomes 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 how to play out these battles as they are clear deciders in the game. Many of the literature on Go is purely based around providing life and death problems and explaining the importance of them and how to tackle such situations appropriately.Chikun (1993)Davies (1975) 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 ways to describe the status of a string 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.

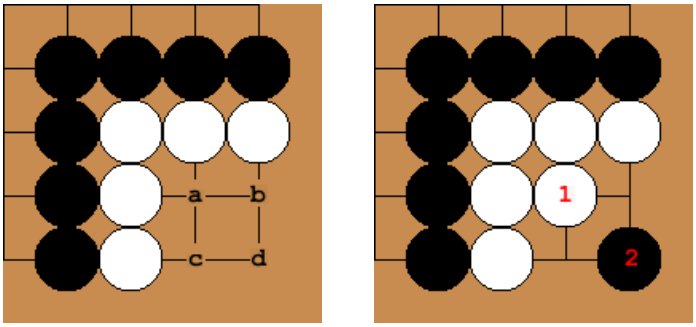


(a) Before Moves *(b) After Moves*

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

play 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.

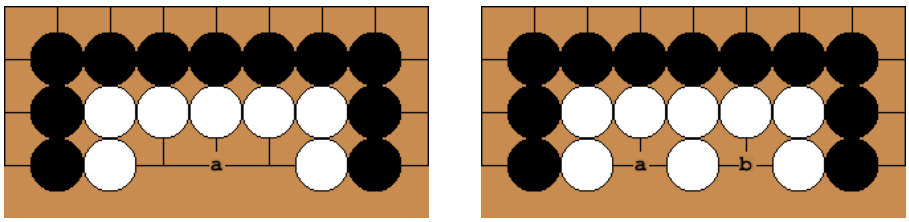


(a) Before Moves *(b) After Moves*

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 is 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 a group of dead white stones, for this group to live it requires white stones on two points diagonally opposite to each other and no white stones on the other two points. 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 the 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 a 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.



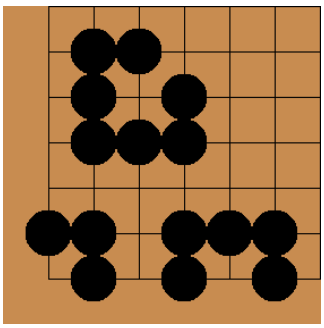
(a) Vital point at a *(b) White lives with 2 eyes*

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, the stones surrounding the 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 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 to 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.

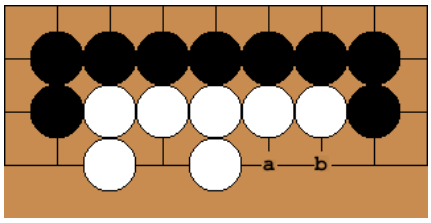


(a) *Different Single Point Eyes*

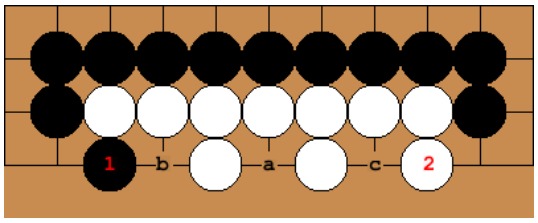
Figure 2.9: *Single Point Eye in the middle, side and corner*

In cases of larger groups which surround more empty points, these empty points are referred to as the eye space. The eye space of a group is important in determining early on within a game whether a group is dead or alive. An eye space can be reduced to create real 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 fully 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 two half eyes at b and c can be deemed as alive due to the fact that when black plays 1 to deform one half eye white can play 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 types 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 the 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 without uncertainty. Another but a 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 an 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 to the end game scoring. In isolated life and death problems and for the purpose of GoLD this difference between unconditional life and Seki is not taken to account, Seki is 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 the 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 Müller (2002) 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 some 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 opponent stones cannot 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 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 the safety of keystrings within a problem is important. Explorer's methods of recognising safe strings such as Benson's algorithm Benson (1976) and also the improved version of the algorithm described in Eth Zurich and Müller (1998) were regarded during the initial design of GoLD.

Work done by Ken Chen and Zhinxing Chen Chen and Chen (1999) 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 the value of a shape of stones 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 Danieli (2010) on his own TsumeGo problem solver. In his paper, he explores in-depth the process in which his program was developed. Explanation of key concepts such as move generation and board evaluation were helpful in understanding how to develop my own work. A particularly relevant piece of information which was used as advice during the design 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 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 positions 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 position for each node.

3 Requirements & Software Development

The project needed a well-defined list of requirements to outline key features and help 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. 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

The program was split into to three core parts from beginning of the project : play mode, editor mode and the "computer". Hence splitting the functional requirements into these three sections of the program was ideal.

3.1.1 General Requirements Across Both Modes

There were many requirements which were required for both play mode and editor mode, to save redundancy its better to express them together.

Go Board The Go board is an integral part of both modes and is required for the user and the computer to interact with the program. A board object which is capable of storing relevant data and contained methods which are able to process the relevant data was required.

The relevant data needed to include the following:

- A 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

Other Features Along with the Go board, play mode and editor modes had multiple features which were both required to allow the users to perform the tasks they needed.

Both modes needed the following features implemented:

- Load problem
- Save problem
- Switch modes
- Reset the problem
- Undo & Redo moves

3.1.2 Play Mode

The play mode required some unique features which allowed the user to play through problems and also to adjust the computer's settings.

Play mode needed the following features most likely in the form of a button:

- Request the computer to start search for the best move
- Request the computer to stop search
- Pass
- Swap turns
- Reset the problem to when it was loaded
- Set whether the computer responds automatically to user's move or not
- Set the breadth limit for the search
- Set the depth limit for the search

3.1.3 Editor Mode

The editor mode required some unique features to allow the user's create Go problems with ease.

Editor mode needed the following features most likely in the form of a button:

- Set who plays first during the problem
- Reset the board into an empty state
- Rotate and flip the board
- Place keystone
- Switch keystone colour
- Choose which colour of stone to place
- Set and remove "valid points"

3.1.4 The Computer

Main aim of project required the computer 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 be able to find the best move possible and play it on the board in a limited time frame. This needs to be done through a tree search where many valid sequences of moves are searched through and the move which results in the best board position for the current colour needs to be played. An additional requirement for the computer was that it should be able to limit the depth of the search according to the user's choice. This consequently implies a requirement for a board evaluation function which is used to score the board positions found at the depth limit. The computer was also required to have the ability to limit the breadth i.e. the number of move per ply according to the user's choice. Hence there was a need for a move generator to produce a subset of moves from all the valid moves.

3.2 Non-Functional Requirements

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

Performance / Time - Important concern was the time a user wants to spend during the process of solving problems using GoLD. Hence the time it took for the computer to make a move.

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 without any failures or crashes

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

3.3 Software Development Process

From the start, the project was divided into two major milestones each containing a set of requirements to achieve the milestone. These milestones were set up to help evaluate the progress throughout the project and also to help streamline the workflow throughout the project so that at all points there was a goal to strive towards.

Milestone 1 The first milestone consisted of creating and implementing most of essential features of the program. The first choice made during the project was to choose Java was the main programming language within the project. This choice came completely out of preference and ease of use. Once this was determined a 2D graphics engine was required to draw the program. Initially to immediately start work on the project , the Swing tool kit along with Java's Canvas was used. After further development and research , it was found that this combination of tools kits was sufficient for the project. From there onwards the Slick2D graphics tool kit was used to render the program.

The first milestone came with the challenge of implementing a tree search. The final tree search used within GoLD was created after two iteration of designing and implementing a tree search. The first iteration was based on the Minimax tree search algorithm which was adapted to play Go. Following on the next iteration of the tree search consisted of Alpha-Beta pruning tree search which greatly improved the performance compared to the simple Minimax search. Finally, to improve quality of the over program in terms of usability and to improve performance, Iterative Deepening was used along with Alpha-Beta. Note that Iterative Deepening was implemented much later on the project and was part of the second milestone.

Milestone 2 The second milestone was concerned with implementing heuristics within GoLD. This is included board evaluation, move generator and move ordering. Board evaluation features was the first of the three to be designed and implemented because it was arguably the most important of the three and required the most time and effort.

To implement board evaluation research was done about Go and how humans perceive the value of different Go board positions. Countless hours were spent understanding some of essential strategy regarding eyes and alive shapes and dead shapes. Lot of the information gathering was done through Sensei's Library Library which is website which has information on almost every

aspect of Go from simple to advanced. The result of researched inspired the pattern matching based board evaluation used within GoLD

Once the board evaluation function became sufficient enough to solve moderately difficult problems, the focus was set on Move ordering. At this is point the Killer Move heuristic was already thoroughly searched when research was done about Alpha-Beta. Hence it was simple matter of apply it to Go and implementing it within the search. When Iterative Deepening was introduced it was simply a matter of some alteration to the code to implement move ordering using the principal variation of each previous iteration. This was also essential to keep the performance of Iterative Deepening comparable to a fixed-Depth search.

The final part of program to be implemented was the move generator. It was essential that this was kept simple it is called during every ply of the search hence a complex move generator would drastically slow down the performance. To keep it simple a distance-based generator was designed , which prioritised moves nearer to keystones. After implementation and testing the simple distance-based generator was found to be in adequate to generate moves for spread-out Go problems. Therefore, a pattern-matching based move generator was designed and implemented combined with distance-based generator to yield better results.

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 board position. The simplest way to achieve this is by searching through the game tree and determining the end result of each valid sequence of moves. The game tree consists of every single board position that can occur from the current board position. The 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 a game-ending.

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. 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.

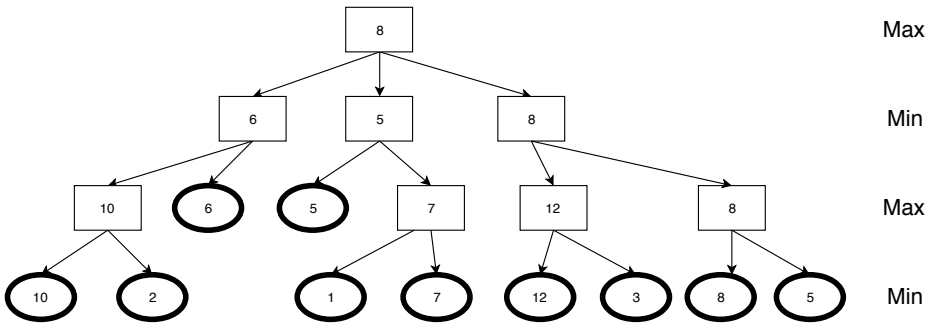


Figure 4.1: *Minimax used for simple number game. Best score the maximizing player can get is 8. Even though scoring 10 and 12 are possible if the minimizing player plays optimally these scores cannot out be achieved.*

Game-Ending State 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 an oval shape.

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 "alpha-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.

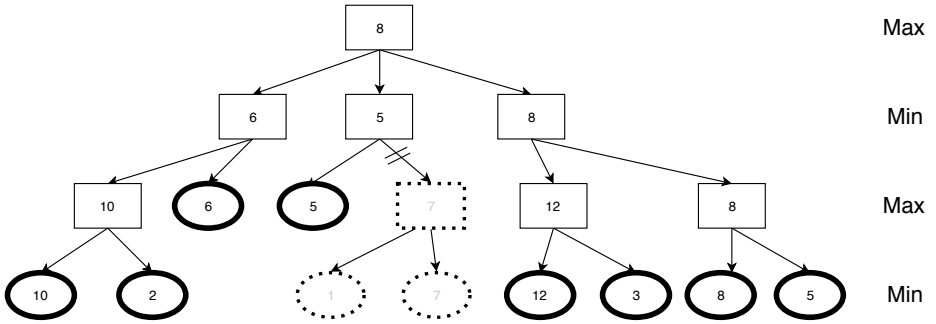


Figure 4.2: *The same simple number game as Figure 4.1 but using Alpha-Beta pruning. In the second subtree after searching the first move which returns 5, the algorithm cuts off from any further search in that subtree as it realises the first subtree will always be better than 5.*

Limiting Depth 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. Therefore, the Alpha-Beta search needs to do some type of board evaluation to determine the value of the current board position , board evaluation used within GoLD is detailed further on the chapter.

Go Adapted Alpha-Beta Pseudocode 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 − ∞
    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

4.1.3 Alpha-Beta with Iterative Deepening

Due to the depth-first nature of a fixed-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. 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.

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. This effectively alters AB search to only increment depth by 1 at a time. Figure 4.3 illustrates the process of Iterative Deepening , it is essential to understand that Iterative Deepening is not a new search Algorithm itself but rather a framework in which the Alpha-Beta search is fitted into.

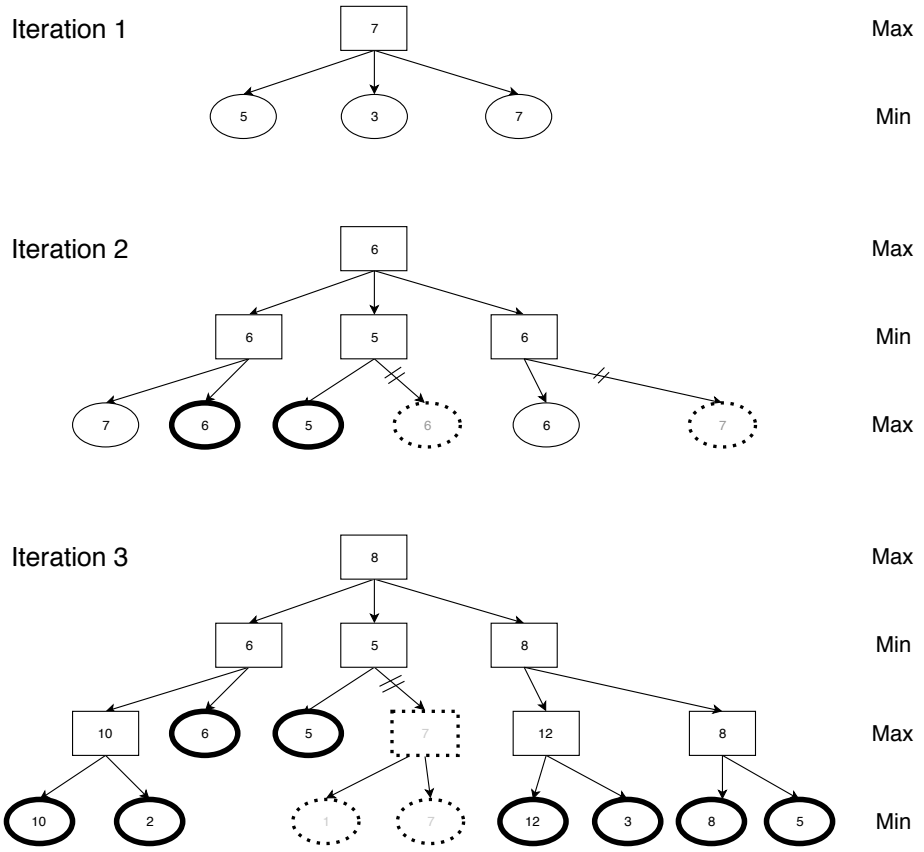


Figure 4.3: *The same simple number game as Figure 4.2 but using Iterative Deepening. Note here the thinner oval shapes represents an evaluated state due to depth limitation. Whereas the emboldened oval shapes represent a game-ending state. You can see at the end of each iteration there is resulting answer which the search can return if the user decides to stop in the middle of the search.*

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. Korf (1985) 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 the overall run time of the search. The Principal Variation move ordering heuristic is based on this idea , explained later on in this chapter.

4.2 Heuristics

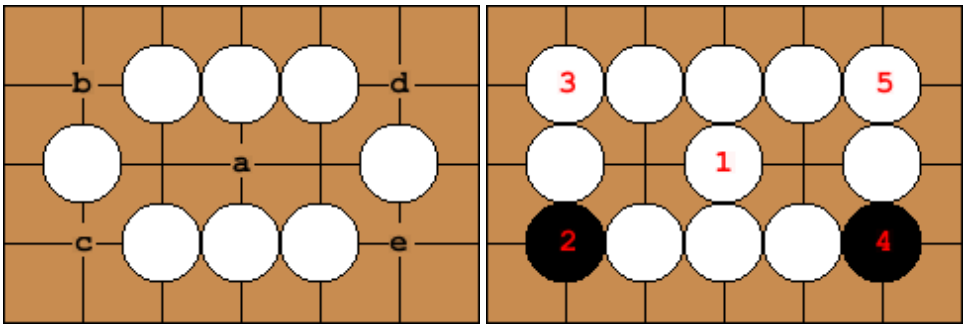
The Alpha-Beta pruning search algorithm by itself is a great improvement over the standard Minimax algorithm but even with this improvement, the sheer size of search space for a game tree can become quickly overwhelming for the search to handle in a reasonable time. Alpha-Beta even at best case has a 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 Board Evaluation

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.

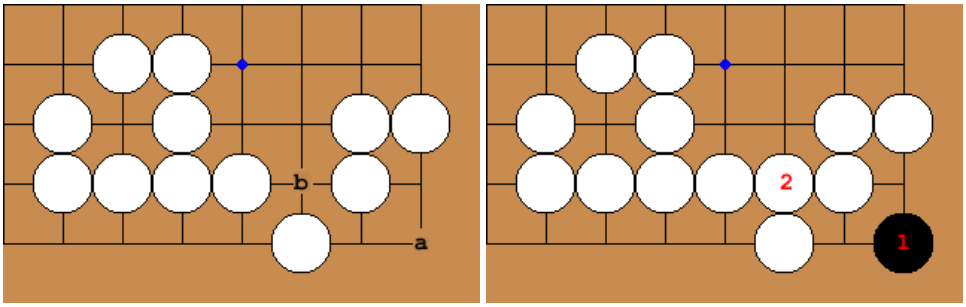


(a) Before *(b) After*

Figure 4.4: *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 4.4a 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 4.4b 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

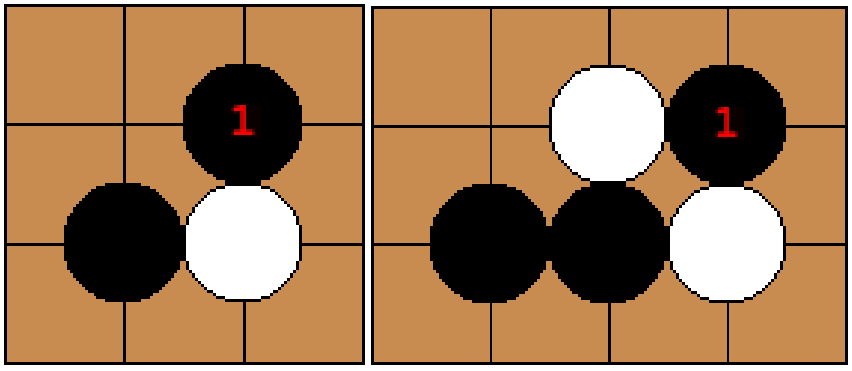


(a) Before *(b) After*

Figure 4.5: *Bent Three in the corner with a Single Point eye nearby to connect.*

will still be able to create 1 real eye which should be valued quite highly. By connecting two 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 4.5 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 4.6: *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 4.6a 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 4.6b 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 Kranenburg and Kusters (2001). 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 Akl and Newborn (1977) method, Killer Heuristic, introduced by Huberman, B.J. Jane. Huberman (1968) 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 alpha-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.

Principal Variation Heuristic The principal variation (PV) is regarded as the best sequence of moves found by the search, in a fully explored game tree the PV will be regarded as the best set of moves are optimal for both players to play. Hence an optimal search should look at the PV sequence of moves first to produce Alpha-Beta cut-offs throughout the rest of the search and drastically reducing the search space. The Iterative Deepening framework lends itself to this idea, after each iteration of the search, the PV found during the search can be used as the first sequence of moves to be searched within the next iteration. This type of move ordering at the start of each iteration of the search will take advantage of the knowledge gained by the previous iteration. An issue with PV is that increasing depth limit might cause the PV found by the search to completely reset to another sequence of moves. This is due to the better understanding gained by searching further ahead. Even so PV ordering will help produce more Alpha-Beta cut-offs.

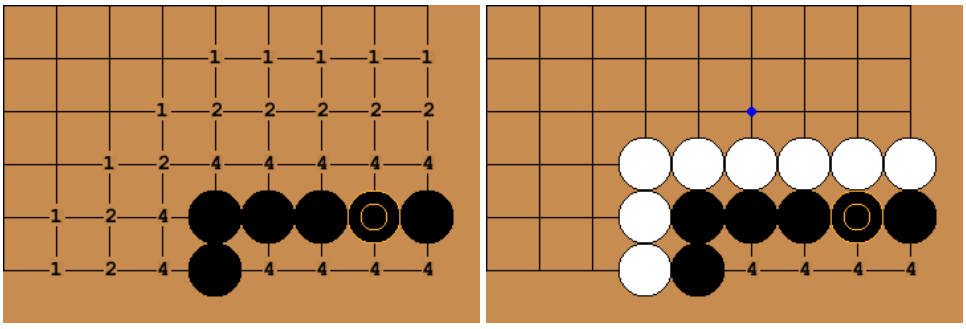
4.2.3 Move Generator

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.

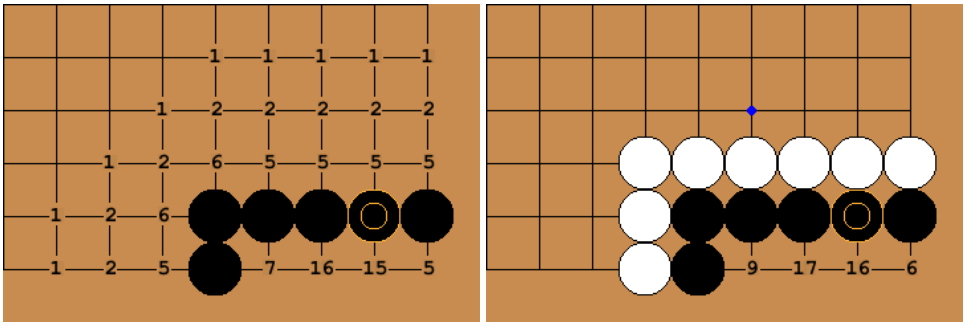
One heuristic used to process the current board is a simple distance-based method. It relies on the school of thought that the closer a move is to the key group of stones the higher relevancy they have. Within GoLD, the search is aware of the key group of stones (marked by keystones) hence the distance-based 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 4.7b 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 4.7: *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 4.8:** 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 4.8 shows the same board positions shown in Figure 4.7 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.

4.2.4 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 SLI which is based on LWJGL LWJ. All of the major constructs of Go are implemented within GoLD to create a stable environment for the user to solve life and death problems. Some of the key aspects of the program are the Go board, the Pattern Matcher and the two modes of play. These features were implemented in order to facilitate GoLD's requirements such as the "computer" which plays against the player. Which is essentially a search function greatly dependant on how the Go Board is structured. Many smaller but crucial features had to be implemented 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 during board evaluation or move generation. The Board object also implements public methods which allow the computer and also the users to interact with the board. The rules of Go are implemented via the Board object which automatically checks if a move played by the user or the computer is valid. If it is valid then the board position is updated to represent the result of the move played otherwise if the move is invalid the Board object denies the move and creates an in-game message to let the user know.

5.1.1 Stones

Representation of stones on the board is simple but effective to keep the operation of gathering information about strings and liberties fast. 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 avoid 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 slow down the entire program.

5.1.2 Strings

A string of stones is 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 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 collections 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 collections of strings are used to iterate through each string inside the collection of opponent strings and determine whether any opponent strings lack liberties and if there is a string which do not have any liberties then it is captured. After which the same is done for one's own strings to keep the integrity of the board. If an opponent string has zero liberties, then every stone in the string is 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 player's strings. The 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 checking for capture, a simple test for Ko is performed. 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 and in the next turn the opponent cannot play there.

5.1.4 Valid Move Checker

Valid move checker is a simple function which returns true or false depending on whether placing on the 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 intersection in the board's 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 make sure that if the current player plays on the point it will not lead to self-capture. Self-capture check is 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 ← getAdjancentPoints(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 on the 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.

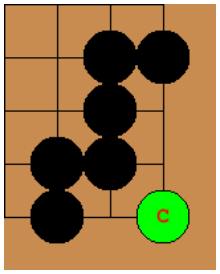
5.2 Pattern Matcher

The pattern matcher within GoLD is able to identify a pattern at the 16 possible variations 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 on the side, corner or the middle of the board. Note that pattern for the middle of the board can be regarded as a pattern that is allowed to occur anywhere on the board as long as the pattern shape 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 simple text notations 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 are able to relate each Point inside the pattern to the starting point of the search in terms of distance. The other fields include boolean variables to check whether a Point is on the side, 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 5.1 shows a Bent Four Corner pattern represented by text notation and the visual representation of the pattern within GoLD. Note here the green stone marked with C shows that the pattern requires that point to be a corner point.



(a) "xrxldxdxlxdxrr#"

Figure 5.1: *Visual representation of a pattern*

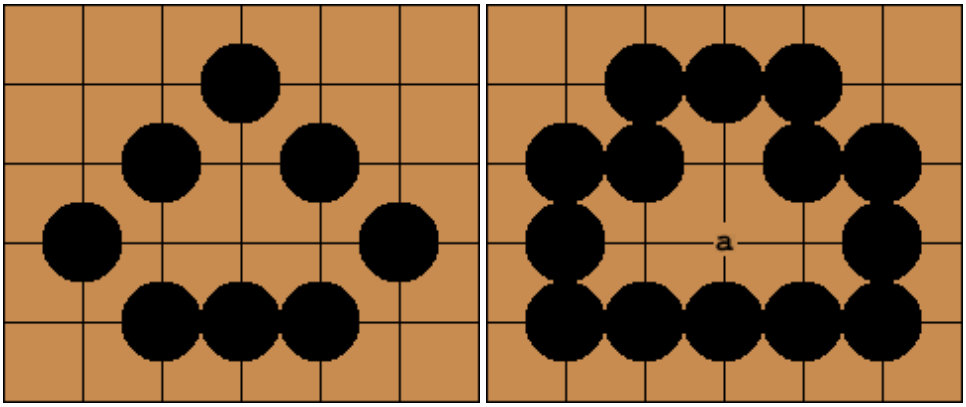
5.2.2 Matching Patterns

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 pattern matcher always deems the starting point as x and y = 0 for the purpose of the pattern matching and then it will search for the rest of the pattern relative to this starting point, starting point is also required to contain a stone. 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 additional enhancements is shown below.

```
Procedure matchPattern(startingPoints, pattern, defColour):
    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,defColour) 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 the matched patterns but also the rotation in which the pattern was found in. GoLD uses the matched patterns along with its rotations to find vital points within each pattern found. Then checks if these vital points are empty, contains defending colour stones or attacking 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 5.2 shows Pyramid Four pattern incomplete where the vital point is not as important, next to a fully complete variation of the same shape where the vital point at a is the only point that matters.



(a) *Incomplete & unsafe walls* ***(b)*** *Complete & safe walls*

Figure 5.2: *Pyramid Four pattern with and without corner stones*

6 Evaluation

This chapter covers the evaluation process of the project. Which includes evaluation of the "computer" player within GoLD and how well it solves problems and also the evaluation of the overall program. First, we look at how the computer handles some specific Life and Death problems which are explained in detail for the reader to understand. A set of problems solvable by the computer were used to evaluate the computer's performance quantitatively. We also go into detail about beta-testing performed at the end of the project and review the feedback/reports received.

6.1 Evaluation of Specific Life and Death Problems

6.1.1 Problem 1

Figure 6.1 shows a basic problem the computer can solve at depth 1. To begin with, let us look at what happens if black is to play the initial move. Black can only live here by playing at **a** or **b** otherwise optimal white play will capture the black group. If black plays at **a** then white needs to play at both **b** and **c**.Otherwise if black play at **b** the white needs to play at **d**and **a**. Since white cannot make two moves at once, the black group lives if played correctly. Let us assume black plays at **a** then the best move possible is for white to challenge black is either **b** or **c**.Keep in mind in a real game of Go white would avoid playing in this area once black plays at **a** or **b** as it becomes impossible for white to win the situation.

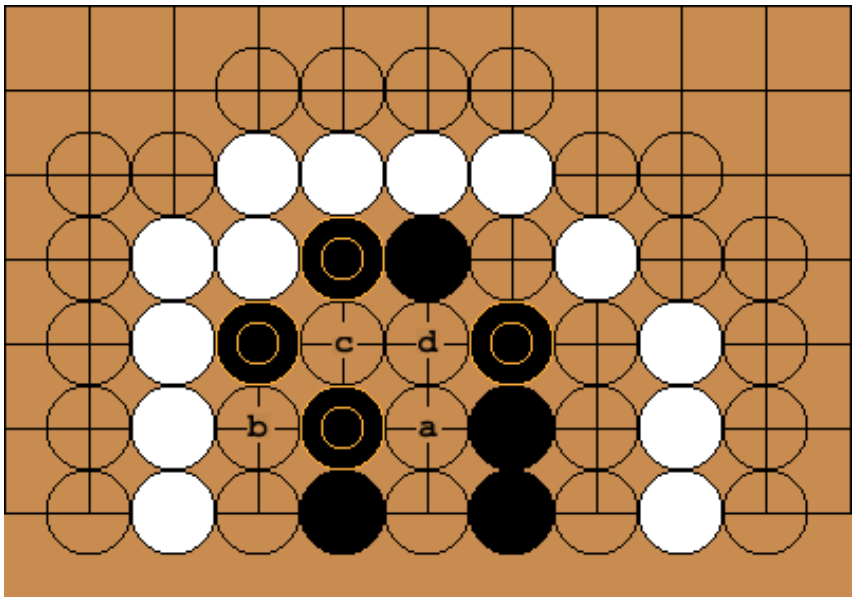


Figure 6.1: *Basic Life and Death Problem*

Figure 6.2 shows the sequence of moves which occurs when the computer plays black. We can see the computer(black) can play out the problem as black to produce two eyes in order to keep the black keystones alive. The computer can solve the problem with both variations of white's response.

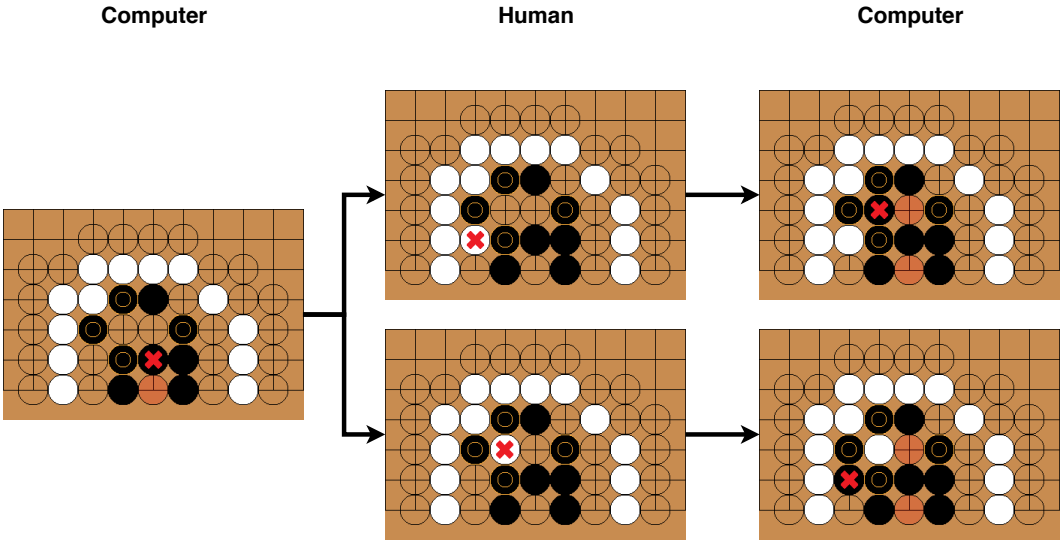


Figure 6.2: *GoLD's "computer" solving the problem as Black first. Stone marked with x shows the move played.*

Now let us look at what happens if white is to play first. White can kill the black group in two ways. White can play at **a** or **b** to capture the black group. If white plays at **a** then black needs to play at **b** and **d** to live. If white plays at **b** then black needs to play at **a** and **c** to live. In both scenarios black needs to play at two points at the same turn which is impossible.

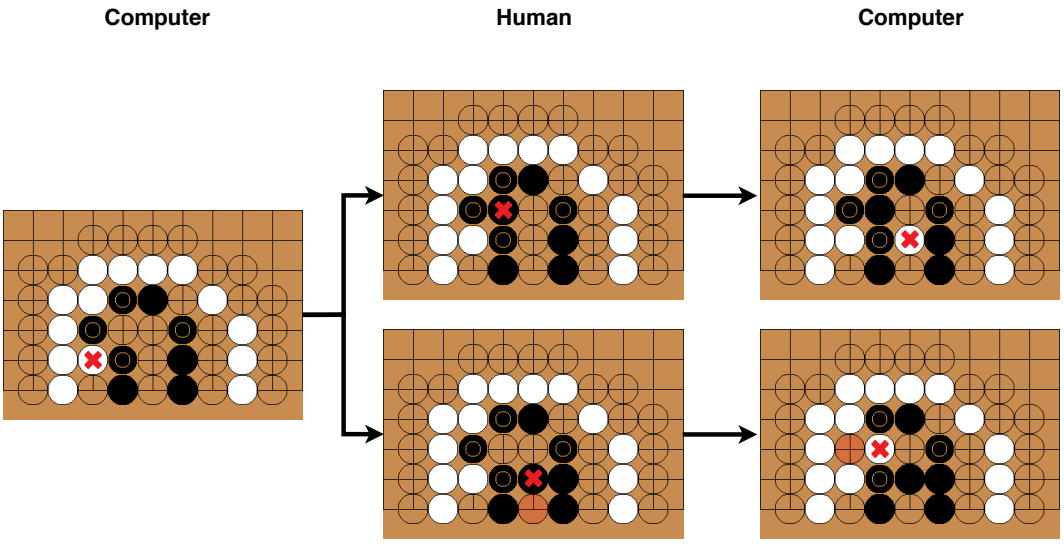


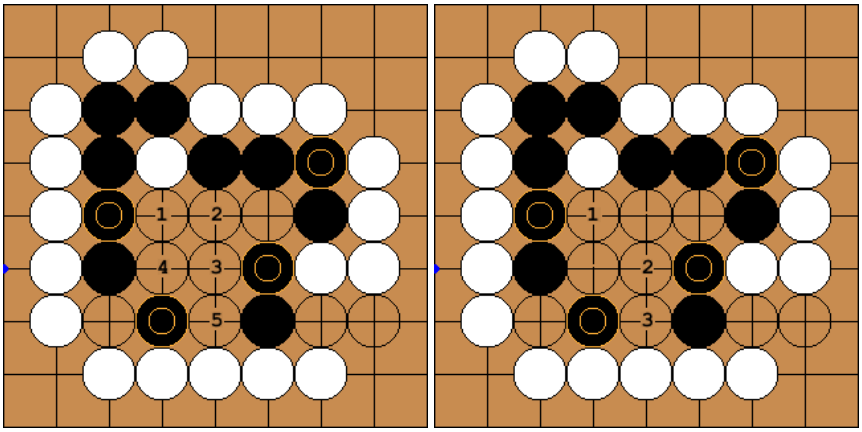
Figure 6.3: *GoLD's "computer" solving the problem as White first. Stone marked with x shows the move played.*

Figure 6.3 shows the sequence of moves which occurs when white is to play first and controlled by the computer. The computer (white) decides to play at **b** to begin. After this black has two

choices, both will lead to death but will create an opportunity to live if white makes a mistake. We can see that the computer will respond correctly to both of black's moves correctly.

6.1.2 Problem 2

Figure 6.4 shows a more complex problem the computer can solve at depth 6. Figure 6.4b shows the sequence of moves which white should play to kill black and Figure 6.4a shows the sequence of moves black should play to live.



(a) White to kill *(b) Black to Live*

Figure 6.4: *Different variations depending on who plays first*

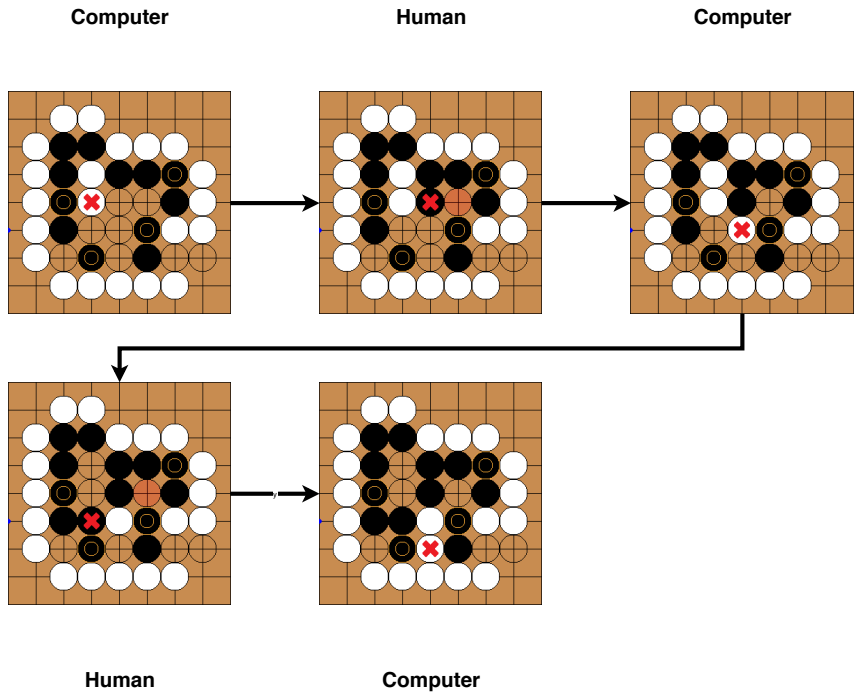


Figure 6.5: *GoLD's "computer" solving the problem as White first. Stone marked with x shows the move played.*

In Figure 6.5 we see the moves the computer plays as white if white were to play first and black

responding in a way to maintain a chance at life. The computer figures out that playing **1** is the correct move and responding to black at **3** will disallow black to live. The computer kills black even if black plays its best moves.

In Figure 6.6 we can see the sequence of move the computer plays as black if black were to play first and white (human player) responding accordingly. We can see the computer provide the correct moves for black to live.

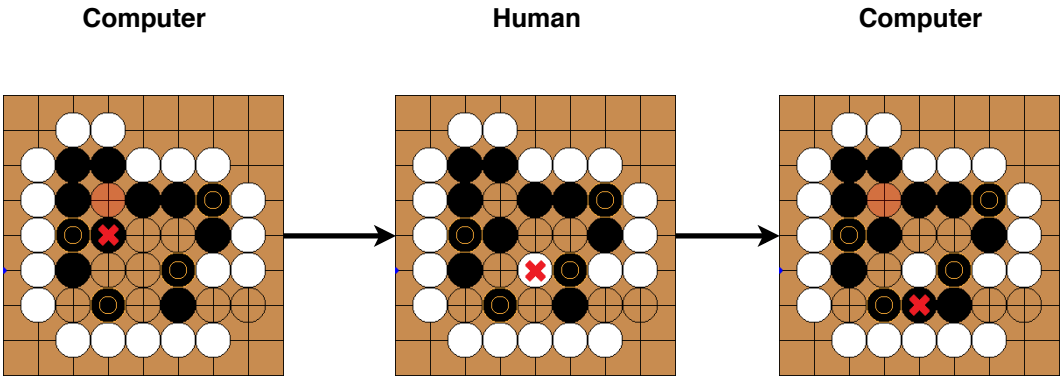


Figure 6.6: *GoLD's "computer" solving the problem as Black first. Stone marked with x shows the move played.*

6.1.3 Problem 3

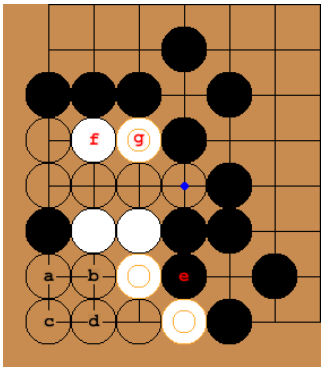


Figure 6.7: *2 Dan difficulty problem*

Figure 6.7 shows a more complex problem than the previous examples taken from GoProblems and is of 2 dan difficulty according to the website. Looking from white's perspective, we can see the white group in a dangerous position. The black stone at **e** reduces the eye space in the corner to 4 square points (a,b,c,d). White can only create one real eye from this eye space because black can place a stone diagonally opposite to the point white places to disable a second eye from forming. In order for white to create two real eyes it needs to make use of the two whites stones at the **f** and **g** to create another eye above the white group.

Figure 6.8 shows the sequence of moves which allows white to live against the optimal play from black, this is the main line of play given in GoProblems. The computer can solve this problem to win as white playing first even if black plays the optimal moves shown in Figure 6.8. The computer is also able to solve this problem from black's perspective by playing at **1** if black is supposed to play first.

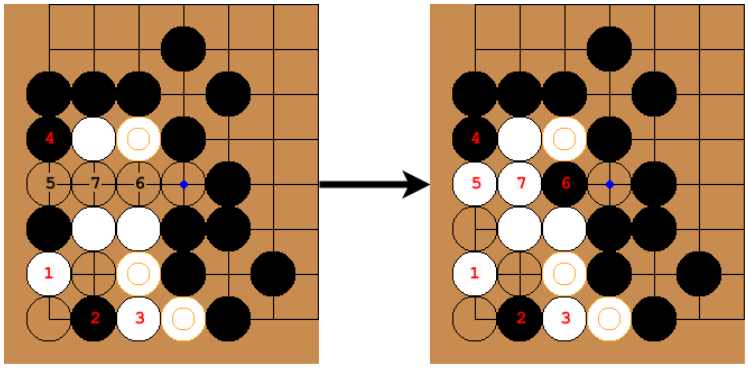


Figure 6.8: *The sequence of moves required for white to live. GoLD's "computer" does replicate this exact sequence as white if the opponent black plays the same points.*

6.2 Performance Evaluation

The performance evaluation process for this project was executed to evaluate the speed at which the computer can find moves and the level of difficulty the computer can solve within a reasonable amount of time. Iterative Deepening was also evaluated during this process to determine if the theoretical improvement to the speed of the tree search is present within GoLD or if Iterative Deepening decreased performance as a trade-off to give the user the ability to stop the search early.

The performance evaluation process contained the following steps:

1. Create a folder of Go problems from a library of problems which indicates the difficulty of each problem.
2. From this folder of Go problems, test each problem individually to determine if the problem is solvable within a reasonable time frame.
3. For every solvable problem collect relevant data to analyse.

The problems used during this process were attained from GoProblems , this website was especially helpful because it contained difficulty rating for each problem depending on how the users of the website found the problem. The reasonable time frame was two mins, but some problems were allowed slightly more time. To determine if the computer can successfully solve a problem is a difficult task. For larger problems, it becomes impossible to test every variation of move sequences to determine if the computer can solve the problem. For this reason, the main line of play shown on GoProblems along with some other lines of play which seemed relevant were the only ones tested. Some solution found by the computer were regarded as incorrect by GoProblems due to the difference in Ko is treated. Within GoLD, if the attacking player can capture then they win, the occurrence of Ko does not matter whereas it does on the website. Every problem was also tested with the objective and starting condition inverted to prove the computer is able to solve the problem from both perspectives. For example, what this means is a problem where the objective is for white to kill black and white play first then the problem is also tested with black to live and black plays first. Each problem was tested for up to 30 minutes (multiple run-throughs) to determine if it was solvable in a reasonable time frame. The relevant data collected for each solvable problem included: The depth in which the problem is solved; The difficulty indicated on GoProblems; The identification number on GoProblems; Number of valid moves S1, S2 ; Four different times T1, T2, T3 and T4.

S1 – Number of valid moves with the original objective (excluding passing).

S2 – Number of valid moves with the original objective inverted (excluding passing).

T1 – Time taken for the computer to play the first move with Iterative Deepening and the original objective.

T2 – Time taken for the computer to play the first move with Iterative Deepening and the inverted objective.

T3 – Time taken for the computer to play the first move with fixed depth search and with original objective.

T4 – Time taken for the computer to play the first move with fixed depth search and with inverted objective.

In addition to these data point, each solvable problem was categorised by where they appear on the board, i.e. the middle, the sides or the corners.

6.2.1 Results

The data collected was aggregated and processed to find relevant averages and trends within it to evaluate the performance of the computer. The primary goal here was to evaluate the solving capability of the "computer" and also to find the average duration of tree search and the optimal depth parameter.

Table 6.1 shows an extract of data gathered from the subset of middle-based problems. Overall, data from 72 different solved problems were used, 30 for middle-based, 21 for side-based and another 21 corner-based.

ID#	Difficulty	Difficulty Rating	Depth	S1	S2	T1 (s)	T2 (s)	T3 (s)	T4 (s)
11650	12 - kyu	1224	5	17	19	13.62	17.96	28.30	24.25
11966	18 - kyu	1047	5	11	12	3.10	4.10	3.10	4.52
615	8 - kyu	1320	6	9	9	6.89	7.28	6.11	4.71
3206	7 - kyu	1365	6	13	14	35.29	19.74	41.90	39.96
3255	17 - kyu	1064	6	14	14	15.45	23.59	34.45	40.72
3350	17 - kyu	1085	6	12	12	11.38	13.41	20.80	16.82
3630	21 - kyu	964	6	14	13	19.42	23.82	17.84	31.59
3984	5 - kyu	1412	6	11	11	25.51	15.40	26.39	15.94
4519	11 - kyu	1259	6	15	14	34.16	32.90	58.80	26.39
4833	10 - kyu	1282	6	10	10	9.25	6.34	7.91	4.90
6219	8 - kyu	1339	6	9	9	6.33	3.92	5.80	4.81
15254	2 - kyu	1508	6	8	8	2.93	2.20	1.72	2.50
15410	30 - kyu	702	6	11	11	7.81	9.10	9.97	12.90
611	2 - kyu	1495	7	9	9	12.22	13.23	11.30	10.55
3267	9 - kyu	1313	7	9	9	12.10	14.55	7.44	14.77
6218	11 - kyu	1240	7	10	10	30.93	25.69	29.51	22.60
8626	12 - kyu	1209	8	10	10	53.37	69.68	94.79	55.61

Table 6.1: *Extract from the collected data for the middle-based problems subset*

Difficulty vs. Depth To find the relationship between the difficulty of a problem and the depth in which it is solved the scatter plots shown in Figure 6.9 were produced which includes the line of best fit. From the plots, we can see that increased difficulty does correlate to a linear increase in depth required. Look the plots we can is exceptions to this rule especially in middle-based problems we can see instances of problems which are low in difficulty but require higher depths of search to solve. For side and corner-based problems, we find that increase in difficulty does in general results in a linear increase in depth required. We can also gather from the plots that the

computer can solve difficult side-based problems at lower depths than middle or corner based ones.

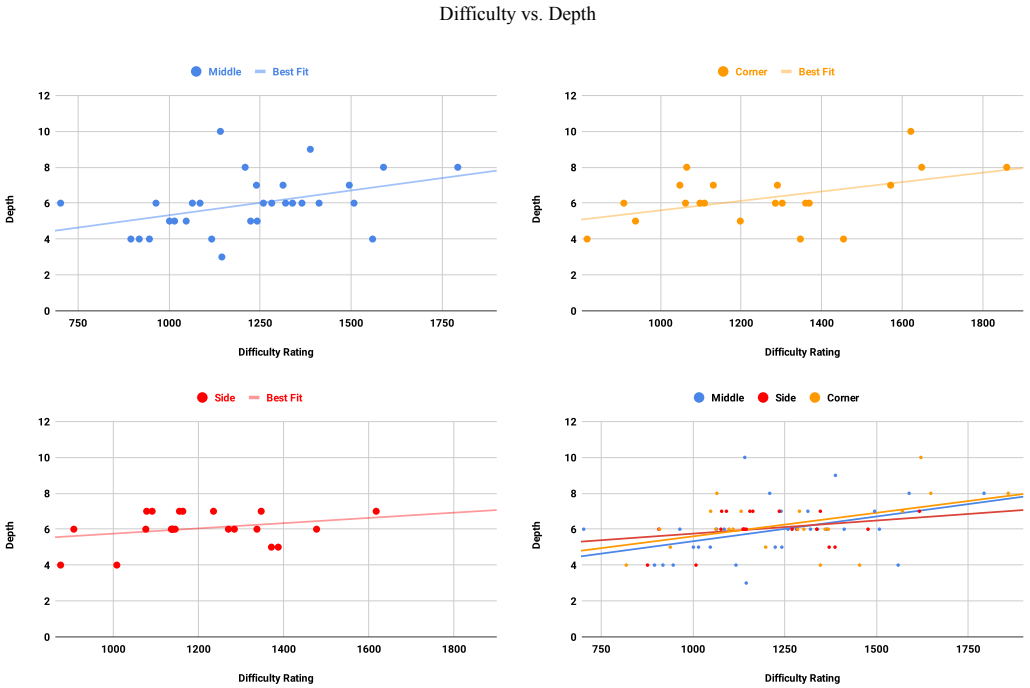


Figure 6.9: *Scatter Plots of Difficult v Depth*

Difficulty Range	Difficulty Rating Range	Depth
~ 30 - 24 kyu	700 - 900	4.5
~ 24 - 16 kyu	900 - 1100	5.7
~ 16- 9 kyu	1100- 1300	6.2
~ 9 - 2 kyu	1300 - 1500	6
~ 2 kyu - 4 dan	1500 - 1700	6.1
~ 4 - 7 dan	1700 - 1900	8

Table 6.2: *Average depth required for varying ranges of difficulties*

Table 6.2 shows the average depth required for different ranges of difficulty ratings. The naming of the difficulty is similar to the player ranking system used in Go Mechner, but the actual difficulty does not precisely correlate to the skill level required to solve the problem. Using this table, we can estimate the depth parameter that GoLD should use for a problem of difficulty X. The table also shows that the minimum depth required to solve basic problems is around 4.5. Which is a good indication of the ability of the board evaluation function, we can see that even at lower depths the board evaluation function can accurately score the board to find the correct move for easier problems. During the testing the average problem could be solved at depth 6 therefore 6 is set to be the default depth limit for the computer.

Difficulty vs. Time & Iterative Deepening To evaluate GoLD performance in terms of difficulty versus time, we need to set a time frame which can be considered as an interactive period. Using a 45 second as the time threshold, we can estimate the average difficulty of a problem which GoLD can solve within this threshold. The average difficulty for threshold was

calculated for T1 and T3 because T2, T4 used the inverted objective; hence the difficulty rating shown online cannot be applied to the problem. Performing the calculation on T1 which is the time taken for the search to find the first move with Iterative Deepening resulted in an average difficulty of rating 1179 which around 13 kyu according to GoProblems. The same calculation for T3 which is with fixed-depth search results in an average difficulty rating of 1181 also around 13 kyu.

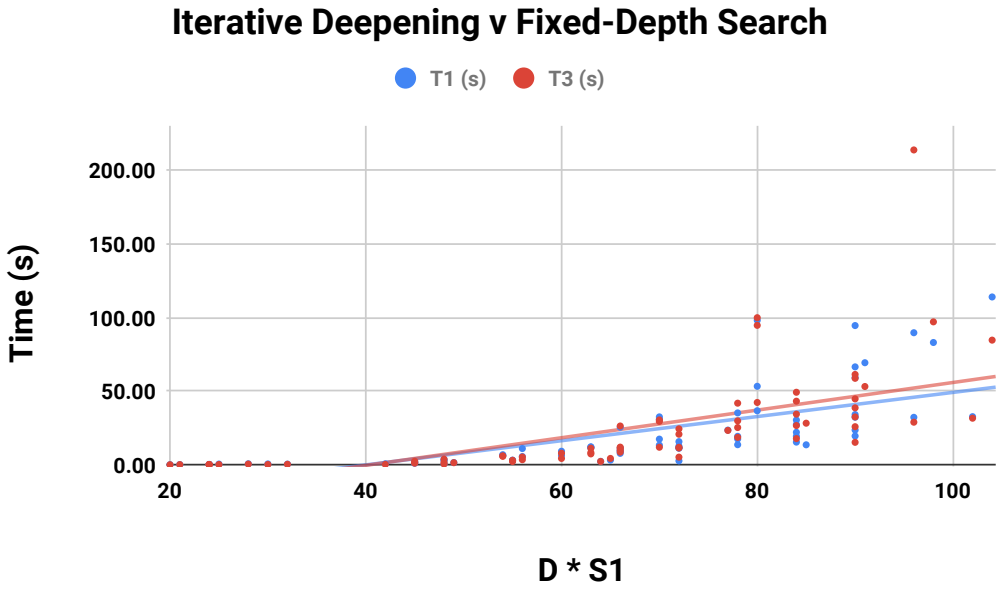


Figure 6.10: *Scatter Plots of Difficult v Depth * S1*

We can look T1, T3 without setting a threshold to evaluate the addition of Iterative Deepening more clearly. Figure 6.10 show a plot of T1, T3 against the depth * S1 of each problem where S1 is the number of initial moves without inverting the objective. Looking at the line of best fit we can see that for shorter searches (around < 10s) the fixed depth search works better but as depth * S1 increases the more effective Iterative Deepening becomes. Note: T1 is with Iterative Deepening and T3 is fixed-depth.

6.2.2 Performance Evaluation Summary

Through the performance evaluation process, the extent of GoLD's "computer" player's ability was found. Out of the 72 solvable problems gathered 18 took more than 30 seconds by the Iterative Deepening search. The remaining 54 had an average difficulty rating of 1179 (~13 kyu). Iterative Deepening had a minimal negative effect for shorter searches and had a positive effect for longer searches compared the fixed-depth search. Therefore, the choice of implementing Iterative Deepening is reasonably validated as it will give users more control over the search result without hindering the performance, sometimes even improving it. This process also made the limitation of search evident; there were around 30 other problems initially re-created with GoLD which were unable to be solved in a reasonable time frame. Most of these were of higher difficulty and larger in size than the solvable problems. This limitation was for seen as it is the nature of heuristics, it can never be perfect and only through increased knowledge can it become more accurate. Increased knowledge requires a deeper and broader search which leads to exponential time increase.

6.3 Beta Testing

At the end of the project, the program was put forward for beta testing. The primary goal of the beta testing was to evaluate usability and functionality. A small number of beta tester were given the program for a period two weeks to test and provide feedback on different aspects of the program. They were also to report any bugs or issues found during testing. Testers were a given brief which contained an introduction to the project along with simple step by step tutorial to play through the tutorial problem provided to them. The testers also received 30 self-designed problems and the collection of 72 problems used during performance evaluation. All problems delivered to the testers contained predefined search parameters at which the computer should be able to solve the problem, and these parameters were set to load up along with the problem.

6.3.1 Pre-Testing Questions

The testers were sent a google form which they had to fill in throughout testing. To gauge the experience of the testers with Go few simple questions were asked. The results showed 4 out of the 6 testers were aware of the rules of Go, if they were not aware they were provided with a link to a simple Go rule guide to help them with the rest of the testing. Following this they were asked if they had played Go before, the result was that only 2 in 6 had played Go before and more importantly only one tester have tried solving life and death problems. A crucial take away from this is that the lack of experience playing Go within the tester will limit the extent to which they can test the GoLD's computer. An experienced Go player might not be able to tell if the computer is playing a poor move hence will not be able to report on it.

6.3.2 Interface & Usability

The testers were asked a series of questions regarding the interface and usability of the two modes of GoLD (Play, Editor). Most testers agreed that the representation of the board was simple and clear, even though the testers answered this question in regard to the board in play mode it would apply for the editor mode as well because the board is represented the same for both modes.

Testers found the buttons on the play mode interface quite simple to identify. Most testers were able to distinguish between the toggleable buttons and the standard buttons. They also rated the labelling on buttons on average 4.5/5 in regard to how useful they were to identify the functionality of the button. One tester reported a lack of visual feedback on buttons and explained: "Sometimes I'm not sure if I've pressed a button or not."

Some testers found parts of the text displayed on the screen difficult to read. One tester found the difference in font size for some buttons difficult to read and would have preferred the larger font which was already in use for bigger buttons. Another tester suggested outlining the text used on the screen would help. In terms of interactively playing through problems, all tester found that it was quite simple to place stones on the board. The testers also agreed that playing through problems were quite simple in terms of usability and the in-game messages provided were helpful.

For the editor mode interface, testers found the radio buttons to choose between which stones to place were appropriately labelled and simple to use. One tester was not able to distinguish the difference between placing a keystone and a normal stone. Which highlights a lack of visual indication for keystones but also a lack of explanation on what a keystone's purpose is within the GoLD and how they differ from normal stones. Testers on average found the creating problems using the editor mode and placing stones of their choice on the board relatively simple. One tester did find that placing valid points one by one "very tedious" and suggested addition of functionality to place an area of valid points along with the ability to place one by itself.

Testers found the background image used for GoLD during beta testing too bright and one tester, in particular, found it “really hard on the eyes”. Since this issue could be easily fixed the background was changed to something more suitable in opacity.

6.3.3 Functionality & Issues

In terms of functionality, the testers provided quite insightful feedback to evaluate the program. When asked about the speed in which the computer responded with move testers had a variety of responses. Some deemed it to be very quick; one even said that they found it difficult to see where the computer had placed the stone. Other testers found that speed decreased with increased complexity of the problem. One tester in particular identified depth up to 6 was “fast enough to remain interactive”. The varied tester responses show that the computer dealing with simpler problems can be too quick for the user to see the stone placed on the board and when dealing with difficult problems requiring higher depths the user has to wait longer on the computer to make a move. These responses confirm the data showed during the performance evaluation process about the speed in which the computer solves the problem is relative to the difficulty of the problem which is only natural. All testers combined found one instance of a Go problem provided to them not being solved at the set default depth for that problem. Apart from this instance, the computer was able to solve every problem the testers played if they let the computer play first.

The testers were asked if there were any difficulty during testing, one tester pointed out that the editor mode’s board taking priority over play mode’s board when switching between the two modes was an issue. While this is a functionality to allow a problem created on editor mode to be transferred over to play mode in retrospect having the ability to choose whether or not to transfer over the problem from one mode to another would have proved to be a better implementation of the functionality. Another tester found that it was quite difficult to judge the increase in time when adjusting the depth and breadth limits hence would have liked to have a warning of sorts to let them know the effect of increasing the limits too high.

A critical issue that occurred during the early phase of beta testing was that the testers found it difficult to run the program due to the difference in java version required to run GoLD to the version they had installed. Since this was a significant hindrance, the issue was fixed during the beta testing phase by bundling the Java Runtime Environment required along with the entire GoLD beta testing package. Furthermore, the portability has since been tested for on Windows 10, Ubuntu and Fedora systems to ensure the program can simply run from the files inside the zipped folder containing the final version of GoLD. Due to a lack of access to a Mac system, Go-LD was not tested on the Mac OS.

6.3.4 Beta Testing Summary

Beta testing was an excellent opportunity to understand how the software tool would perform in the hands of inexperienced users. To begin with the positives, users found the program itself pretty simple granted a tutorial was provided to them which mostly focused on introducing the testers to the basic concept of a Life and Death problems more than the whole program itself. In terms of the reliability of the program, there were no issues reported by the testers, no major bugs or crashes were reported. GoLD could be said to be adequate in terms of performance (of the "computer") from the tester’s responses. Tester found that simpler problems could be solved quickly but increasing the depth higher could result in longer waiting times which is to be expected. The visual aspect of the program was overall the weakest point; testers reported two significant concerns, text readability and the background image. Due to the simplicity of these issues, they were immediately improved upon. The tester’s issues highlighted the visual aspects of the program which requires much improvement for future versions.

7 Conclusion

7.1 Summary

During the course of this project the aim of creating a software tool for Go players to create and solve interactively has been achieved. The final tool written in Java has met most if not all the functional requirements put forth at the begin of the project and the ones add during the period of the project. In terms of the software engineering it was a good decision to split the entire project into two achievable milestones to ensure there was a clear goal at each stage of the development process.

The first milestone was to achieve the main structure of the software tool implemented. This included creating all the functionalities required to allow users to play through a Go problem against a computer that did not use any heuristics to find the best move possible. Once the first milestone was reached the tool contained a simple Alpha-Beta search without any depth or breadth cut offs.

The second milestone addressed the issues with the computer's lack of heuristics since it without heuristics larger Go problems becomes impossible to solve. First major step in achieving the milestone was to introduce user defined depth cut-offs to the search which relied on a board evaluation function. The board evaluation was created to mostly using pattern matching heuristics along with simple liberty counting. Next step was to implement move ordering techniques. The reasoning behind doing so was to drastically decrease the number of moves the Alpha-Beta search had to look through by processing moves which could potential produce AB cut-offs first. The Killer Move heuristic was implemented to perform move ordering along. At this point of the project Iterative Deepening was also implemented to use Alpha-Beta search to give users the ability to stop the computer in the middle of search and force it to play the best move found so far. To go along with Iterative Deepening the principal variation found during each iteration was used as the first sequence of moves to be searched during the next iteration. Finally, a move generator based on pattern matching and distance from keystones where created to enable users to restrict the number of moves the computer searches each ply of the game tree.

Once implementation reached this point in production Go problems were created from scratch or re-created from external sources such as Chikun (1993)Davies (1975)GoProblems. This was done to first test and evaluate GoLD and then to produce a folder of problems which could be solved by the computer to be packaged along with the final version of GoLD.

The evaluation process consisted of two parts, the performance evaluation process and beta testing. The relationship between time for the computer to solve a problem and the difficulty of the problem were discovered during the performance evaluation process. In addition to this the average difficulty the computer could solve in a reasonable length of time was also found. Iterative Deepening Alpha-Beta search was compared to fixed-depth Alpha-Beta search. Results showed that Iterative Deepening is slightly slower for problems which took less than 10 seconds to solve but as the time increased Iterative Deepening became more effective and over took the fixed-depth search in performance. Beta testing performed over a two-week period was able to provide results to evaluate the usability and interface of the program. The results convey that the

overall usability of the program was adequate but certain aspects of visual design required great improvements.

7.2 Future Development

In terms of possible future work on there are many things which I would like to look at to improve GoLD. Some of the suggestions made by the beta tester seemed interesting . A popular suggestion was to add a hint button which could shows the user some good moves found determined by the computer searching in the background. Another great idea was to visualise which moves the computer is considering during the search , this would mean to show the current best move according the latest iteration of Alpha-Beta completed.

The nature of heuristics means that they are not perfect, hence looking at new methodologies to perform board evaluation, move generation and move ordering would be an ideal way to improve on what is currently in use. Specifically, some work done by B.Bouzy in his paper Bouzy (2003) where he looks at improving up on the Zobrist model Zobrist (1969) which based on the idea of creating a map of influence based positions of stones on the board. Applying this work to Life and Death problems could help in determining which moves to be generated via the move generator looking at points of high influence on an influence map. A way to improve on the model B.Bouzy provides is by taking into consideration the difference in strength of stone patterns during the creation of an influence map. For example, four stones arranged in a square block would have a lot less influence than four stones arranged to create an eye in the middle.

Another realm of possibilities can be opened up by introducing Machine Learning to identify and group similar stone patterns together and determine the value of each pattern to be used during a board evaluation function. Machine Learning could even be used to create a completely new board evaluation function from scratch. This would provide major improvements over any human designed evaluation function granted a large enough training set is used.

A Appendices

Typical inclusions in the appendices are:

- Copies of ethics approvals (required if obtained)
- Copies of questionnaires etc. used to gather data from subjects.
- Extensive tables or figures that are too bulky to fit in the main body of the report, particularly ones that are repetitive and summarised in the body.
- Outline of the source code (e.g. directory structure), or other architecture documentation like class diagrams.
- User manuals, and any guides to starting/running the software.

Don't include your source code in the appendices. It will be submitted separately.

7 Bibliography

Lightweight java game library. URL https://www.lwjgl.org/.

Slick2d. URL http://slick.ninjacave.com/.

S. G. Akl and M. Newborn. The principal continuation and the killer heuristic. In *ACM Annual Conference*, 1977.

D. B. Benson. Life in the game of go. *Inf. Sci.*, 10:17–29, 1976.

B. Bouzy. Mathematical morphology applied to computer go. *IJPRAI*, 17:257–268, 2003.

K. Chen and Z. Chen. Static analysis of life and death in the game of go. *Inf. Sci.*, 121:113–134, 1999.

C. Chikun. *All About Life & Death*. Ishi Press International, 1993.

A. B. Danieli. A tsume-go life & death problem solver. 01 2010.

J. Davies. *Life and Death*, volume 4 of *Elementary Go Series*. Ishi Press International, 1975.

I. Eth Zurich and M. Müller. Playing it safe: Recognizing secure territories in. . . 04 1998.

GoProblems. Go problems. URL http://www.goproblems.com/.

B. Jane. Huberman. A program to play chess end games. 08 1968.

N. Kiin. The japanese rules of go. URL http://www.cs.cmu.edu/~wjh/go/rules/Japanese.html.

R. E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artif. Intell.*, 27: 97–109, 1985.

H. V. Kranenburg and W. Kusters. Ai techniques for the game of go. 2001.

S. Library. Sensei's library. URL https://senseis.xmp.net/.

D. Mechner. Go ranks. URL http://www.mechner.com/david/go/kyu.html.

M. Müller. Counting the score : Position evaluation in computer go. 2002.

D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. P. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–489, 2016.

A. L. Zobrist. A model of visual organization for the game of go. In *AFIPS Spring Joint Computing Conference*, 1969.