University of Glasgow | School of Computing Science

Honours Individual Project Dissertation

Go - Project

Jude Haris
May 2019

1 Introduction

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

1.1 Motivation

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

1.2 Aim

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

1.3 Project Outline

2 Background & Related Works

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

2.1 Go

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

2.2 Rules

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

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

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

2.3 Go Terminology

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

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

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

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

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

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

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

Keystring – String which contains one or more keystones.

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

2.4 Life and Death Problems

Life and Death is an essential part of Go. Life and Death is the term used to describe the battle that takes place to either capture stone enemy stone strings or to defend and keep alive ally stone strings. Situations on the board where it become crucial to defend one's own stone strings or capture enemy ones arrive very often. During the end game in Go the ability to win these battles with as much territory as possible will decide the victor. Beginners are highly recommended to understand and learn the how to play out these battles as they are clear deciders in the game. Many of the literature on Go are purely based around providing life and death problems and explaining the importance of them and how to tackle such situations appropriately.[3][4] The purpose of GoLD is to create a tool which helps players to understand and solve these problems.

2.4.1 Dead, Alive or Unsettled

The three main way to describe a string of stones are alive, dead or unsettled. To deduct accurately the state of a group of stone can be difficult and requires a lot of experience and knowledge about the possible moves that could be made and the counter plays to them and the result of counter plays. A group of stones are said to be alive if the group can not be captured even if the opponent is to plays first. To expand on this, even if opponent plays a move which threatens the group of stone there is always a responding move which will in turn keep the group of stones alive.

Figure 1 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 group of stones is said to be dead if the group can be captured even if the group's colour can play first. A white group can be deemed dead if white can play the first move and black has a responding move which will keep the white group in a dead state. Figure 2 show group of dead white stones, for this group to live it requires white stones on two points diagonals to each other and no white stones on the other two , for example if white can play at a and d then it becomes alive. This is impossible to achieve as black can respond to white's initial move by playing at diagonally opposite point seen in 2-b. 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 out come of the board, who ever can play first near the group can determine who controls the whole territory the unsettled group surrounds. Figure 3 show an unsettled white group which has vital point at a where if white plays first at a then the group achieves two eyes and is alive. On the other hand, if black plays first at a then white cannot achieve two eyes hence it becomes dead.

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 as in figure 4. In these shapes stones surrounding eyes are important if any of them are missing then the eye will no longer be a real eye. An important deduction about a single point real eye is the fact opponent cannot play on the single point eye unless the eye is only liberty of the surrounding stones. If two real eyes are connected by the same group of stones, then it becomes impossible to capture that group of stone unless the eyes are covered up by the its own colour. These groups of stones are unconditionally alive which means even if the opponent is able to play multiple times in a row the group cannot be captured. Figure 5 shows an example of a group of white stone which contains two eyes at a and b , due the self-capture rule black can never place in a without having a stone at b and same applies of b hence the white group in unconditionally alive.

In cases of larger groups which surround more empty points, these empty points are referred to as the eye space. Eye space are important in determining early on within a game whether a group is dead or alive. Eye space can be reduced to create eyes , the greater the amount of eye space a group surrounds the greater the ability it has to create two real eyes and live unconditionally.

When eyes are not full developed on the board for example at a in fig 6a 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 fig6b which has a real eye at a and 2 half eyes at b and c can be deemed as alive due to the fact that when black plays to deform one half eye at 1 white can play at 2 to fully form the other half eye into a real eye creating an unconditionally alive group.

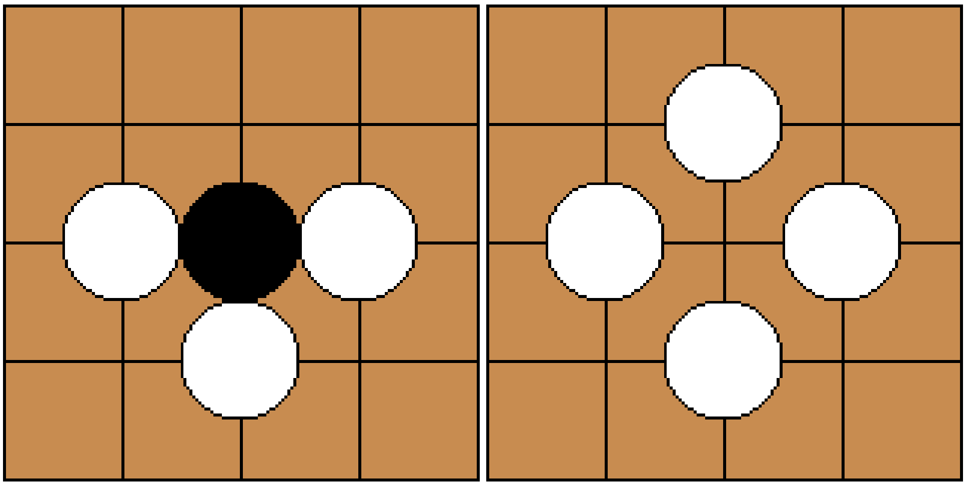
2.4.3 Outcomes of Life and Death

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

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

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

2.5 Examples



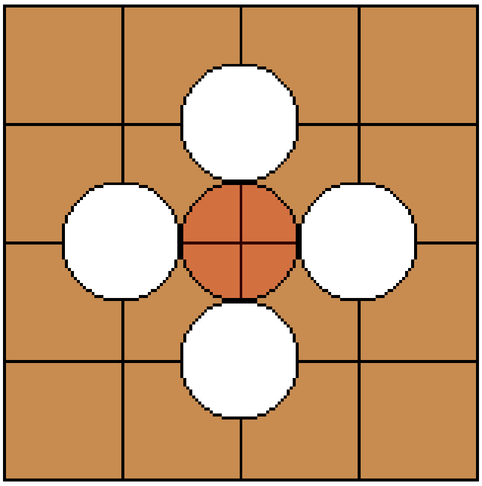
(a) Before Move *(b) After Move*

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

2.6 Related Works

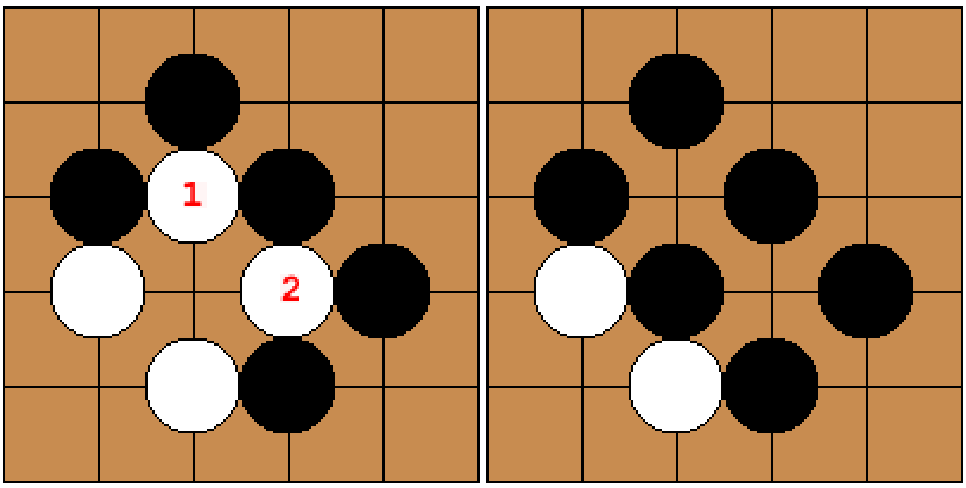
Martin Müller's Explorer [2] was a strong program developed with the intention of playing Go. It captures key aspects of computer Go which can be used along with the minimax algorithm in one program. Explorer can evaluate the board in terms of safe territories for each player and can distinguish between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

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



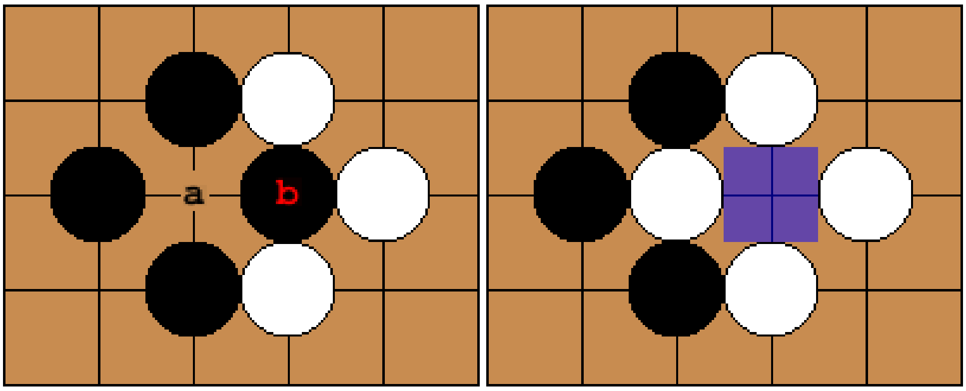
(a) Middle point is invalid

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



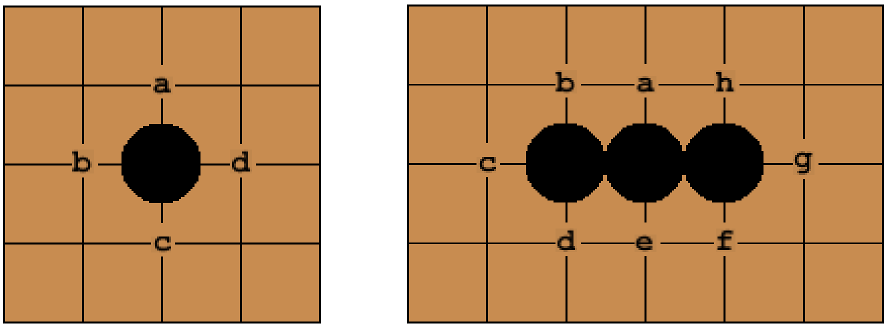
(a) Before Move *(b) After Move*

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



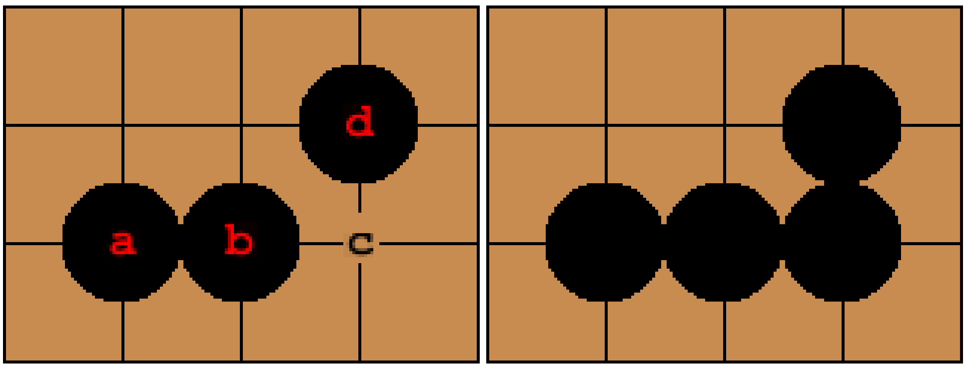
(a) *Before Move* *(b)* *After Move - Blue Square to represent KO*

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



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

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



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

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

3 Requirements

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

3.1 Functional Requirements

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

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

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

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

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

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

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

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

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

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

3.2 Non-Functional Requirements

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

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

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

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

4 Design

4.1 Tree Search

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

4.1.1 Minimax

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

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

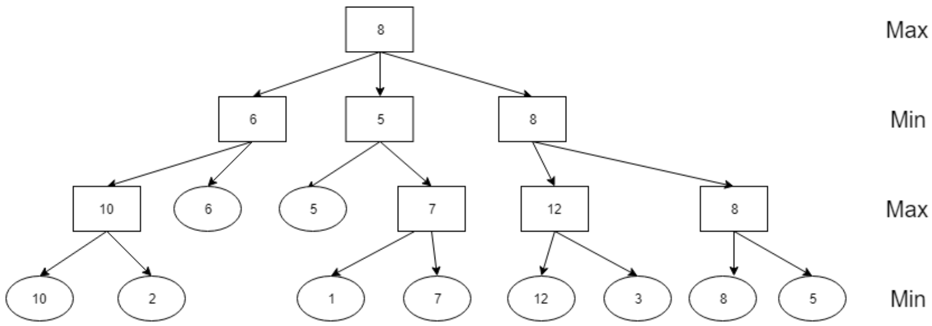


Figure 4.1: *Minimax used for simple number game. Best score the maximizing player can get is 8. Even though scoring 10 and 12 are possible if the minimizing player plays optimally these scores cannot out be achieved.*

4.1.2 Alpha-Beta Pruning Search

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

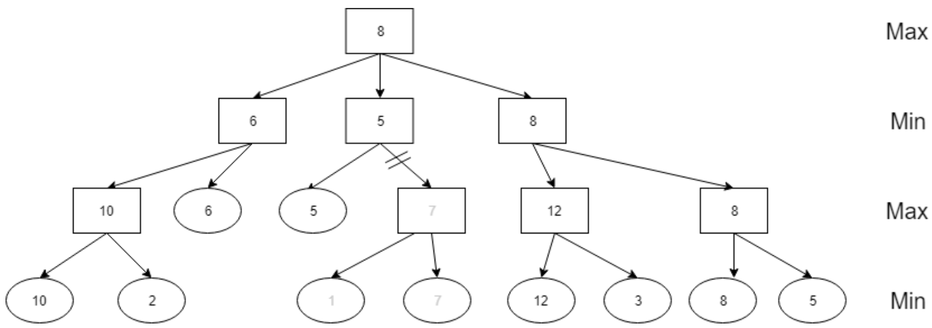


Figure 4.2: *The same simple number game as Figure 4.1 but using Alpha-Beta pruning. In the second subtree after searching the first move which returns 5, the algorithm cuts off from any further search in that subtree as it realises the first subtree will always be better than 5.*

The following pseudocode addresses the Alpha-Beta pruning search algorithm with Go’s Life and Death problems in mind. It was used for the initial design of GoLD’s move finder function and later adapted to implement heuristics described in the following sections of this chapter.

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

Algorithm 1: Alpha Beta Pruning Search

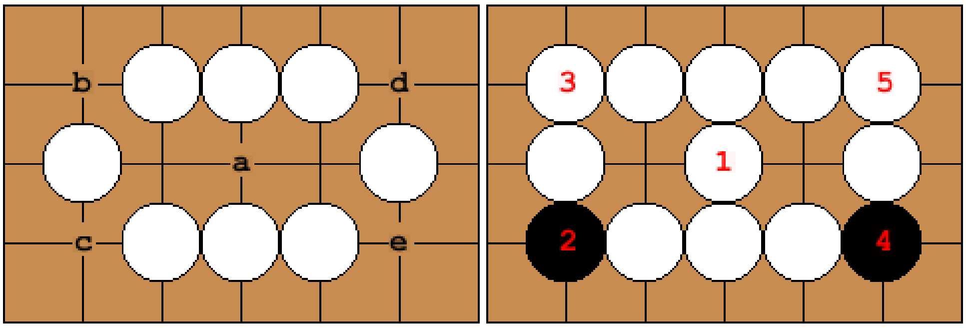
4.2 Heuristics

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

4.2.1 Depth Limited Search with Board Evaluation

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

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

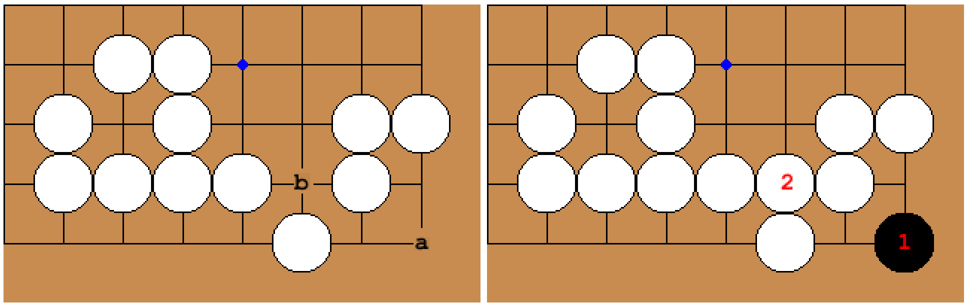


(a) Before *(b) After*

Figure 4.3: *Straight Three in the middle of the board with missing corners.*

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

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

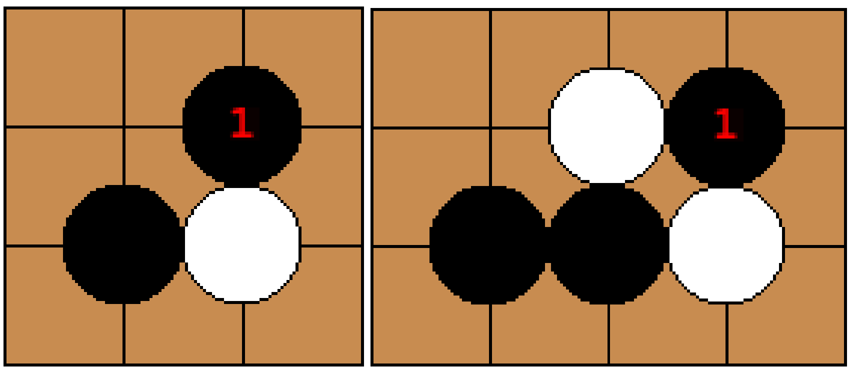


(a) Before *(b) After*

Figure 4.4: Bent Three in the corner with a Single Point eye nearby to connect.

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

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



(a) Black Hanes at 1 *(b) Black Cuts at 1*

Figure 4.5: Important type of moves

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

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

4.2.2 Move Ordering

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

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

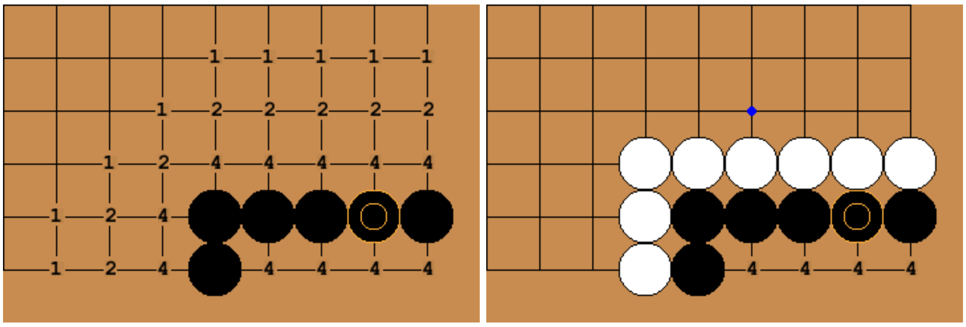
Move Ordering for a board position A which occurs at depth d consists of searching the list of valid moves for k2 of depth d and if k2 is a valid move in this board position then k2 is ordered to the top of the valid moves list. After which, the same is done for k1 of this depth(k1 is searched second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

4.2.3 Alpha-Beta with Iterative Deepening

An issue with AB search is that if the best move is found at depth d then searching past depth d for the other branches is always a waste of time. For example, black is to play and kill white. Moves A to E are the valid moves of the current board position and the moves will be searched in alphabetical order. If D is the best move and will result in a victory for Black at depth 3 then searching past depth 3 for move A, B and C is a waste of time. Without searching D first there is no way to obtain this information hence A, B and C will be searched to deeper depths. Iterative deepening is an effective method to work around this problem. Alpha-Beta with Iterative Deepening is a process in which AB search is performed multiple times and at each iteration, the depth limit is increased by 1. This cycle takes place until max depth limit is reached, or a favourable game-ending state is found (in the example it would be white dead). This effectively alters AB search to only increment depth by 1 at a time and hence solving the problem of searching moves to deeper levels than not required. The drawback of this process is that it requires multiple iterations of AB search and will have to perform redundant computation for the lower depths of the game tree. Work done by Korf R. [8] shows that redundant computation does not affect the overall time of the tree search as much as it would seem. In fact, using the result of each iteration to perform the next iteration can reduce the time taken to perform the next iteration and possibly the overall run time of the search. Each iteration will return the principal variation(PV) which is the sequence of moves which is considered to be the best determined by the search. The PV of the previous iteration will be set as the first sequence of moves the next iteration searches. This is another form of move ordering which effectively improves overall performance compared to a normal fixed depth limit search. Due to the depth-first nature of the depth limited AB search, the search will not provide a result until every move is searched to a game-ending state or to the limited depth. This means the search will not be able to produce the best move searched so for if it is stopped in the middle of searching. Without the introduction of Iterative Deepening in GoLD's search the user is not able to search for a limited amount of time, rather the user would have to predict which depth limit to set in order for the search to complete in the time they desire. Predicting the depth limit could lead the user to set the depth limit too high which would lead the search to last longer than what the user would like. Iterative deepening fixes this issue by allowing the user to stop the search whenever they desire and to use the result of the last fully performed iteration to make the best move found so far hence improving the usability of the GoLD.

4.2.4 Move Generator

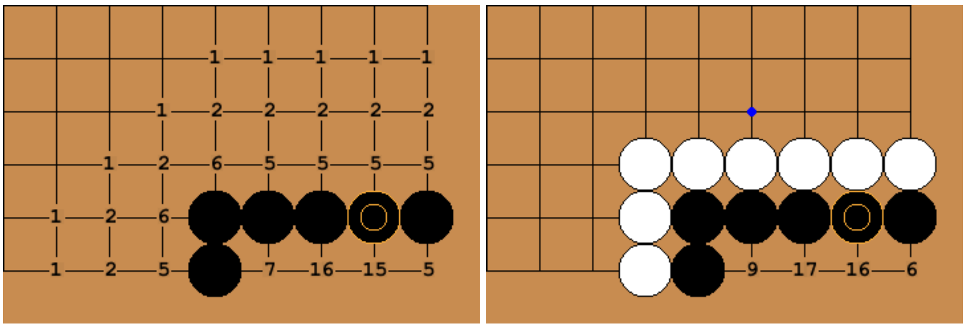
GoLD allows the creator of the problem to set the valid points of play on the board and any points on the board which contains a stone when captured become a valid point of play within the problem. A high number of valid moves to begin the problem with creates extremely large game trees which would take too long to search even with the best heuristic move ordering. The number of valid moves can be regarded as the breadth of the game tree. In order to limit the breadth of the game tree GoLD introduces a move generator. The goal of a move generator is to pick out x number of moves from all the valid moves of the current board position in order to cut down the number of moves to be searched. A perfect move generator will always pick the best move if it could only pick one move to generate but this is impossible to achieve without searching ahead. The next best result is that the generated moves list has a high chance of containing the best move and this chance obviously increases with the number of moves generated. The generated moves list should contain the best move possible otherwise the best move will not be searched hence the search will never return the correct result. If more moves are generated for use, then there is a high chance of one of them being the correct move but then more moves have to be searched. In order to maintain a low number of generated moves and still have a high chance of containing the correct move, the move generator within GoLD uses different types of heuristics to determine which moves have a high chance of being the correct move and which moves are irrelevant. One approach to determine which moves are good moves is to simulate each move being played and then evaluate the board to see which move will lead to an immediately better board position. While this could be an effective approach, it will also increase the worked load to process each board position. This might, in turn, slow the search down rather than speeding it up. The approach which GoLD uses is to simply process the board position as it is and then pick out moves which are relevant and give them each a score indicating how relevant they are. Then the top x number of relevant moves are generated for search. One heuristic used to process the current board is a simple distance-based method. It relies on the school of thought that the closer a move is to the key group of stones the higher relevancy they have. Within GoLD, the search is aware of the key group of stones (marked by keystones) hence the distance-based heuristic will give a high relevancy score to any moves 1 step away from the key group then a lesser score to moves 2 steps away and so forth. This process is used for moves up to 3 steps away from the key group as the relevancy of moves further away is minuscule and hence a waste of time to process. This heuristic also accounts for enemy stones which are blocking the key group's access to other points hence in situations like the one in Figure 4.6b where points that are blocked by the surrounding white stones are considered irrelevant.



(a) Value is halfed each step away from the key group *(b) White stones devalues moves outside the wall*

Figure 4.6: *Distance Based Relevancy*

The other heuristic method used for move generation is based on pattern matching. Similar to pattern matching for board evaluation, here we use pattern matching to identify relevant moves. The same patterns used for board evaluations are also used to identify relevant moves on the board. The move generator searches for patterns on the board and increments the relevancy score to any moves which are vital points within a pattern found. This will allow the move generator to consider more complex shapes on the board and generate moves according to these shapes. In general, moves found through pattern matching are given higher relevancy score than ones found through the distance-based heuristic. The flaw in this heuristic is of course only a limited number of patterns can be searched because increasing the number of patterns will increase time taken to generate moves hence the same crucial patterns used for board evaluation are used for the move generator.



(a) Not surrounded by white stones *(b) Surrounded by white stones*

Figure 4.7: *Distance & Pattern Based Relevancy Combined*

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

4.2.5 Suicidal Move Removal

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

A solution can be derived from the fact that placing 1 stone can at maximum only remove 1 liberty of a keystring. Hence, GoLD only needs to look at the moves that place a stone in liberties of keystrings with 2 liberties, any more liberties would ensure at least 2 liberties would remain intact if a stone was to be placed on a liberty. For a keystring with 2 liberties, placing in one of those 2 liberties does not automatically mean it is a suicidal move for two reasons: Move captures an opponent string; Move would increase or maintain the liberty count. If any of these two reasons are met, then the move is not suicidal.

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

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

5 Implementation

5.1 Play Mode

5.2 Edit Mode

5.3 Game Board

5.4 Problem File

5.5 Pattern Searcher

6 Evaluation

6.1 Unit Testing

6.2 Evaluation of Heuristics

6.2.1 Types of Problems

6.2.2 Success Rate

6.3 Beta Testing with Go Players

6.3.1 Questionnaires

6.3.2 Interviews

7 Conclusion

7.1 Summary

7.2 Future Works

7.3 Reflection

8 References

1.http://www.cs.cmu.edu/ wjh/go/rules/Japanese.html 2.Müller, M. (2002). POSITION EVALUATION IN COMPUTER GO. ICGA Journal, 25(4), pp.219-228. 3.Chikun, C. (1993). All About Life & Death. Ishi Pr. 4.Davies, J. (1996). Life and Death. Ishi Pr. 5.Cornelis Diederik van der Werf, E. (2004). AI techniques for the game of Go. [Maastricht]: UPM, Universitaire Pers Maastricht. 6.Akl, S.G., & Newborn, M. (1977). The principal continuation and the killer heuristic. ACM Annual Conference. 7.Huberman, B.J. (1968). A PROGRAM TO PLAY CHESS END GAMES. 8.Korf R. Depth-first iterative-deepening. Artif Intell. 1985;27(1):97-109. doi:10.1016/0004-3702(85)90084-0