University of Glasgow | School of Computing Science

Team I: Go Problem Solver With Alpha-Beta Tree Search

Eilidh Anderson
Jamie Dale
Scott Hood
Kiril Hristov
Niklas Zwingenberger

Level 3 Project — 27 March 2015

Abstract

The abstract goes here. e.g.: A project which was designed and implemented with the outcome of a functional Go problem solver.. etc.

Education Use Consent

We hereby give our permission for this project to be shown to other University of Glasgow students and to be distributed in an electronic format. **Please note that you are under no obligation to sign this declaration, but doing so would help future students.**

Name: _____ Signature: _____

Name: _____ Signature: _____

Name: _____ Signature: _____

Name: _____ Signature: _____

Name: _____ Signature: _____

Name: _____ Signature: _____

Contents

Chapter 1

Introduction

This is a group project with the aim to create an interactive program that plays out and attempts to solve life and death problems within the board game, Go. The finished problem solver contains several elements. It features an artificial intelligence which selects future moves and a game engine which stores current game boards. Each of these components communicate with a graphical user interface, allowing an individual to interact with the board, create problems and play against the game's artificial intelligence or another human. The finalised problem solver created allows a user to improve their ability and tactics to enable better play through of similar problems within a real-life game of Go.

When analysed, Go can be shown to be a very interesting game. It is a perfect information game that makes use of pure calculation during the making of moves allowing for perfect play. Go in itself is also a very complex game, with its number of possible games being 10^{761} [9]. In comparison, chess has 10^{120} possible games [8], highlighting the vast search space of Go. As such, whilst a brute force chess program, which exhaustively searches for moves, can play at a championship level a Go program would take far too long due to the greater possibilities. A practised human Go player may be able to look upon a game in progress and predict numerous future moves, allowing them to select moves that are beneficial to their current strategy. To create a program to achieve a similar level of skill such as this whilst playing Go requires a huge amount of time as well as a trade-off between fast running times and a more intelligent AI. This group project had 5 team members and 6 months to create a Go problem solver including an extensive artificial intelligence designed to be similar to a human player's thought process.

There are several phases within the game of Go and like other board games, many types of recurring problems which can be identified through specific patterns. This project's focus is on the life and death problems that occur during a game of Go and the creation a program which can successfully solve these problems whilst playing against a human. The game itself is played between two players using black and white stones respectively. These stones are placed on intersections within a board of grid lines and the main aim of the game is for a player's stones to enclose a larger area of the board than their opponent's. A player can also remove their opponent's stones from the board by surrounding them with their own stones, which is called capturing. Within the many rules of Go, such as capturing, life and death is a prominent concept. A group of a player's stones can either be classified as alive or dead. If this group of stones is mostly surrounded by opponent's stones but cannot be captured and removed from the board even when the opponent moves first, it is alive. For a group to be alive, it must contain at least two "eyes," being a section of board only surrounded by

friendly stones. Whereas, if the group cannot avoid capture even when the player owning the group of stones moves first, it is classed as dead. The dead state normally occurs when a group has none or only one eye within it. Another state of stones that can occur is unsettled, where the outcome and eventual life or death of a group of stones depends on which player moves first. Normally, objectives accompany life and death problems - such as white to kill a group of black stones or black to escape capture from a group of white stones.

Life and death problems were selected as the project's focus for numerous reasons. One such reason is that the search space is smaller for these problems in comparison with a full game which contains many different phases of play. This allows the AI to be created and run more efficiently as less searching is required during move selection. Several methods can be used to provide move selection such as visual intuition or a randomized algorithm, e.g. Monte Carlo [reference here]. In this project the paired methods of board evaluation and move look ahead through Minimax [reference here] and AlphaBeta [reference here] were used, providing the most efficient basis of life and death problem solving. Life and death problems also allow the program to have a specific objective, as previously described, enabling specialised heuristics to be implemented which led directly on from common problem objectives. Following on from this, life and death problems also feature repeated common stone patterns which will normally influence the outcome of the problem. Hence, this allows heuristics to be implemented with these distinct stone positions and their usual outcomes kept in mind. To create a program that is able to mimic the game intelligence used during life and death problems, careful planning and several algorithms were used, including tree-searching techniques and numerous heuristics.

The first step in creating a Go program capable of playing against a human and solving life and death problems, is implementing a brute force strategy during move selection within a game. To enable this in the Go problem solver, a legal move checker utilizing a detailed algorithm must be created. This checker allows the program to filter out all illegal moves, minimizing the search space for the best possible move whilst the checker also removes any captured stones from the board. From there, minimax tree searching is utilized to decide on the game engine's next move through minimizing the possible loss for a worst case scenario - where scenarios are certain selections of future moves. The minimax algorithm plays out all possible legal moves and evaluates them through the user-given objective. If one of the first available moves successfully completes the objective, it is returned. Otherwise, minimax will play out the best moves for the AI and the opponent sequentially. This sequential play through creates branches within the search tree until no legal moves remain. Hence, through exhaustive search of all moves, minimax returns the next possible best move according to the objective supplied - allowing for successful brute force play through of life and death problems. In order to make minimax more efficient, the alpha-beta algorithm can also be implemented in accordance with the minimax search tree. In essence, alpha-beta will return the same next best move for the artificial intelligence as the minimax but it works more effectively by removing branches from the original minimax search-tree that would not influence the final best possible move returned. The algorithm does this by stopping the evaluation of a branch of moves when it is found that these moves are worse than previously considered moves, which therefore creates a more optimal subtree - instead of the exhaustive search tree produced through the sole usage of the minimax algorithm, as described previously.

Whilst minimax successfully allows for brute forcing of problems and alpha-beta decreases the search space yet further, even using these two tree-search algorithms in conjunction with each other can still lead to long running times for more complicated problems. There is also no usage of strategy implemented through the utilisation of minimax and alpha-beta for move selection. The

implementation of heuristics and evaluation functions is fundamental to improving a program's artificial intelligence through adding algorithmic strategy. Heuristics are implemented within the Go problem solver to improve its move selection during play through, instead of solely taking a brute force approach. The use of heuristics is essentially akin to implementing the techniques and rules that humans use during play within the problem solver. By giving a program heuristics, there is an attempt to provide the judgement that humans possess by coding several game-related rules and regularly occurring board situations within it. Through the use of pattern matching and recognition algorithms, the system should be enabled to recognize when these heuristic situations occur and move accordingly - allowing for quicker and more human-like move selection.

The project's final program structure contains several key elements, including the graphical user interface and an artificial intelligence comprising of a legal move checker, tree searching algorithms and extensive heuristics. Each of these components has accompanying tests and documentation to prove its success, as well as user evaluation. The program allows for life and death problem creation and play through, either against the AI or another human. The AI itself makes use of the minimax tree searching algorithm combined with the alpha-beta search algorithm to allow for precise move selection. Implemented heuristics based upon Go strategies improve upon these move selections further during life and death problems within the program. With a fully-developed AI, the project's end program has the heuristic capabilities to play against humans with higher skills levels and also the ability to reach specific objectives of harder to solve life and death problems within efficient running times.

Chapter 2

Background

2.1 Go Background

Go is a two player board game that originated in China more than 2,500 years ago [reference needed] and is still mostly played in its original form. Players have ranks from 30-1 Kyu and 1-9 Dan, where 30 Kyu is the lowest rank and Dan ranks are reserved for the grand masters [reference needed]. Go can be classified as a "*territorial*" game, simply meaning that the main aim of Go is to surround more space on the board than the opponent. The board itself can be thought of as a piece of land to be shared amongst players and is usually marked with a grid of 19 by 19 lines, although it may be smaller. The game is played between two players, one using black stones and the other using white stones. Players take turns to place stones upon intersections of the board during gameplay and the game continues until either both players pass or there are no more legal moves remaining. Some of the basic Go rules are as follows:

- The board is empty at the beginning of the game unless the players agree to a handicap, e.g. If one player has a handicap of 3 then that player will start with 3 stones on the board at the start of the game.
- Once placed on the board, stones may not be moved, but stones may be captured. This is done by surrounding an opposing stone or group of stones.
- A player may pass his turn at any time.
- A stone or solidly connected group of stones of one colour is captured and removed from the board when all the intersections directly adjacent to it are occupied by the enemy, meaning the group has no liberties.
- A player's territory consists of all the points the player has either occupied or surrounded.
- The game is won by gaining the most points, being determined by:
    1. The number of opponent pieces captured.
    2. The amount of territory held at the end by each player.

2.1.1 Why Life And Death?

Life and death problems are fundamental to the game of Go and occur frequently during game play. As described previously, a life and death problem occurs due to a player's group of stones either being "*alive*" or "*dead*." When a player's group of stones cannot be captured in any game state it is alive, whilst if the opponent is able to capture an opposing group of stones, then that group is dead. Normally, a player will have an objective in mind when playing through a life or death problem - whether it is keeping their group of stones alive or killing an opponent's stones group. Go itself is a large and complex game with a vast search space, which is both wider and deeper than that of chess, and it is even possible for the search space for a single life and death problem to be larger than all of chess. With the timescale and number of people working upon this project, creating an AI to play a full sized game of Go would not be feasible, although it would be possible. Therefore, the decision to concentrate efforts into solving life and death problems was a more realistic yet still challenging goal. The project initially focused on a small subset of life and death problems before moving on to more computationally challenging problems once the initial problems were being solved with a reasonable success rate.

2.1.2 Related Work

Reference some other similar Go projects, mention algorithms they used and the ones we used, etc.

2.2 Go Problems

2.2.1 Go Terms Relevant For Life And Death Problems

To enable an understanding of life and death problems, as well as any game of Go, several conventional Go moves and concepts must be recognised.

Liberty: A liberty is an empty intersection that is adjacent to a stone or to a connected chain of stones. As capturing stones occurs when opponent stones have no liberties, the number of liberties is fundamental to the solution of life and death problems.

Eye: Eyes can be described as internal liberties of a group of stones that, like external liberties, prevent the group's capture but are harder for an opponent to fill in. Eyes are very significant for life and death problems as the existence or non-existence of eyes in a group determine whether that group is alive or dead. A group with no eyes or one eye will die unless its holder can develop them into two eyes. A group with two or more eyes will live because it is impossible to remove all liberties of the group in one move by the attacker. In Figure 2.1 all internal liberties with a red circle can be described as eyes.

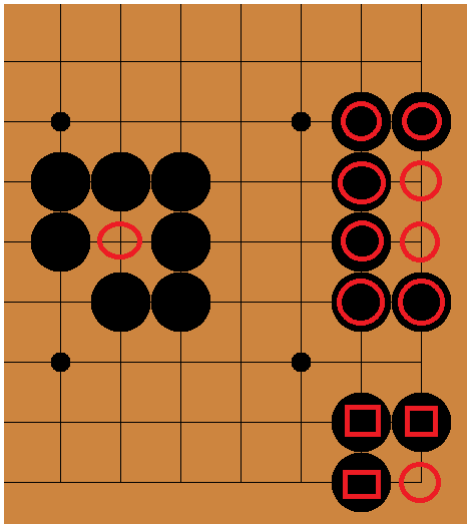


Figure 2.1: Examples of eyes.

Straight Four Eye Space: A straight eye of four liberties makes a group alive since the owner of the stones can always make two eyes. The two points labelled '*m*' in Figure 2.2 are miai, meaning equally valued moves. If white plays one of the marked moves, then black can play the other and create two eyes.

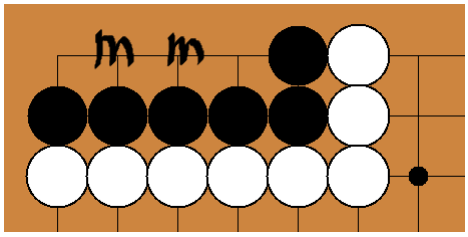


Figure 2.2: Straight four eye space.

Atari: Atari is a term used in Go for a situation where a stone or group of stones only has one liberty. Stones in atari can hence be captured on the next move unless the defending player places a stone on the liberty, thus giving the group more liberties and at least temporarily taking the group out of Atari. In Figure 2.3 if black places a stone at any of the liberties with a red circle then black will capture the white stone. In Figure 2.4 if white places a stone at either the liberty at *a*" or the liberty with a red circle then the black stones will be captured.

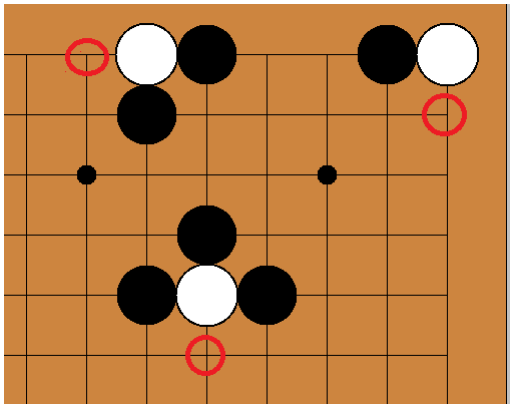


Figure 2.3: Single stones in atari.

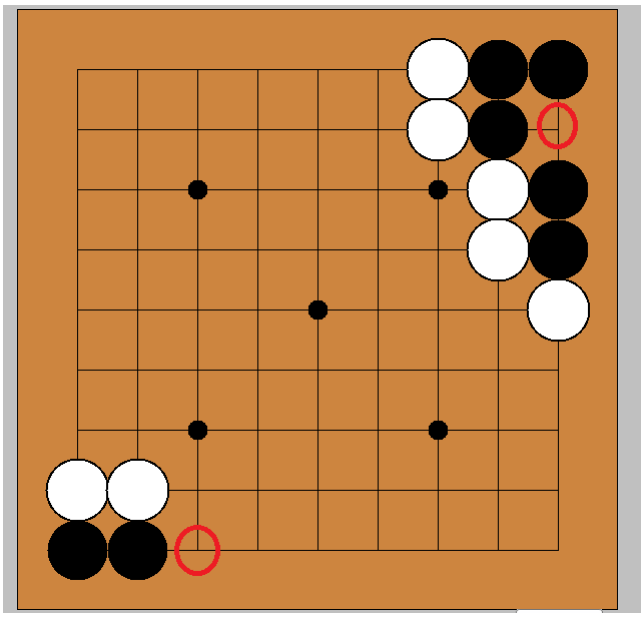


Figure 2.4: Groups of stones in atari.

Hane: A hane is either a move that wraps around an opposing player's stone or group of stones and doesnt touch any of the players stones or a diagonal move played in contact with an enemy stone. There are many positions where a hane is considered a good move for example in the six die eight live group of problems which with be detailed later in the chapter. In Figure 2.5 the stone labelled with a red circle is a hane as it wraps around the black stone and isnt adjacent to another white stone.

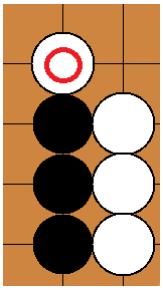


Figure 2.5: A hane move.

Ko: Ko describes a situation where two alternating single stone captures would repeat the original board position. These alternating captures could repeat indefinitely creating an infinite loop in the game. The Ko rule stops this infinite loop from occurring: If one player captures the Ko, the opponent is prohibited from recapturing the Ko immediately. In Figure 2.6 above black can capture the white stone with the red circle by a play at a. The resulting position is shown in Figure 2.7. Without a Ko rule, in this position White could recapture the black stone with the red circle by playing at b, which would return the board to the position in Figure 2.6 and then Black could also recapture creating an infinite loop. So, if in Figure 2.6 Black captures at a, White may not play at b for his first move after the black capture. Instead White has to play elsewhere. After that Black can choose either to win the Ko by playing at b in Figure 2.6, or to play elsewhere as well. Playing elsewhere however would allow White to take the Ko back, since the recapture restriction is only valid for the next turn.

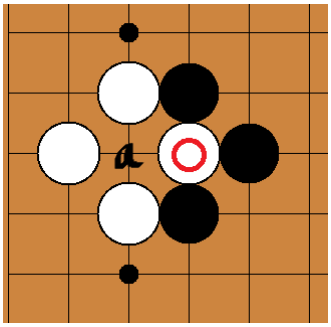


Figure 2.6: Ko rule: part 1.

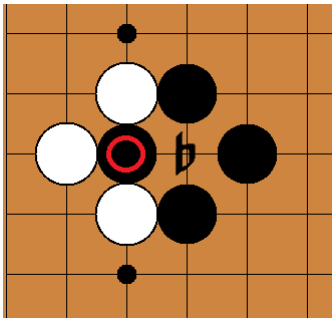


Figure 2.7: Ko rule: part 2.

Miai: Miai can be defined as a pair of empty points that have the same value. For example if black

plays at A, white can play at B and suffer no disadvantage from the exchange. Equivalence is an important aspect of miai, in the sense that the two options allow the same objective to be achieved more or less. For example, miai in a life and death context might mean the existence of two different moves resulting in the same outcome of a group living or being killed. In Figure 2.8 both point a and point b have the same value in terms of the life and death of the group if black plays at a then white plays at b and vice versa. If white places a stone at either a or b he creates a second eye for the group and thus keeps it alive.

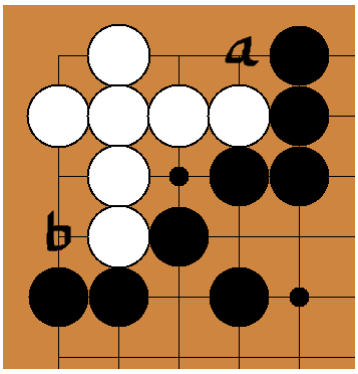


Figure 2.8: Miai.

Seki: The term Seki translated into English means mutual life, this situation occurs when two live groups share liberties which neither of them can fill without dying so neither player will. In Figure 2.9 the white group of stones and black group of stones with red circles share two liberties a and b. If either player plays into one of these points the opposing player with play into the other and capture the opposing players stone so neither player will. [*maybe include another seki example where the group has eyes*] .

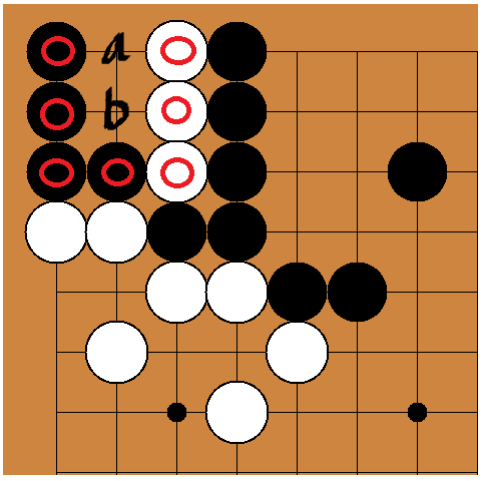


Figure 2.9: Figure 2.1.9 seki

2.2.2 Common Life And Death Problems

Most life and death problems be grouped into certain shapes or techniques used to solve the problem given the specified objective. These life and death shapes occur frequently during games of Go and

in many cases one small positional change of a stone can create a big difference to the outcome of the problem.

2.2.3 Unsettled Three

This shape is concerned with when a group's interior eye shape consists of three spaces in either the form of a bent three space or a straight three. Here the key number is three; if the space is any less than three then the group is dead, four spaces and the group is alive. In Figure 2.10 below, if white plays at either of the red circles the group lives because it creates two eyes for that group. If black plays at either of the red circles it kills white.

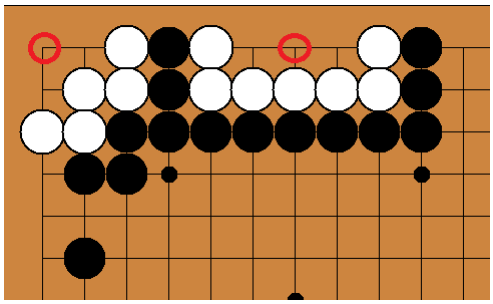


Figure 2.10: Unsettled three examples.

2.2.4 Six Die, Eight Live

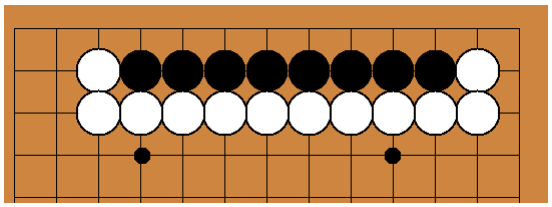


Figure 2.11: Eight stones (alive).

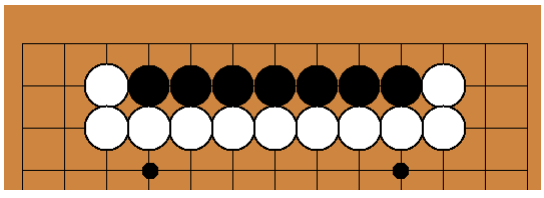


Figure 2.12: Seven stones (unsettled).

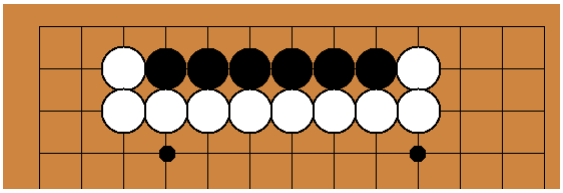


Figure 2.13: Six stones (dead).

The groups of stones in this section consist of rows of stones on the second line in from the edge of the board. Therefore the groups eye space lies on the edge of the board. When the group is away from the corner the rule is six stones die, eight stones live. Figure 2.2.2 above shows the case where the group is alive ie. has eight stones. Figure 2.2.3 shows the case where the group is unsettled ie. has seven stones and finally figure 2.2.4 shows the case where the group has six stones and is dead.

Figure 2.2.5 above shows the sequence of moves made when the defending group has eight stones assuming both players play perfectly. The points labelled m are miai if black plays one white will play the other creating a straight four eye space keeping the group alive.

Figure 2.2.6 above shows sequence of moves in the unsettled case when black has seven stones and white moves first. In this case, again black ends up with a four straight eye space keeping the group alive.

The case when black has seven stones and plays first is shown in Figure 2.2.7 above. After the first four moves black is still alive as it has created an enough space for two eyes but whites move at 5 kills black.

Finally the instance of moves where black has six stones is shown in figure 2.28 above. Even if black goes first, after whites move at 4 he is dead as there is not enough space to create two eyes.

Chapter 3

Design And Implementation

3.1 Requirements Gathering

At the beginning of the project, a formal specification was given and described the creation of:

An interactive program, with GUI, that presents a Go problem on a board, and interacts with the user as they explore the solution space of the problem. The program uses tree search algorithms to find good moves.

As this original specification itself was quite brief, several ambiguities were identified in the requirements. These ambiguities included what tasks, goals, and objectives would be explored when designing and implementing the program. Hence, a requirements gathering interview was arranged with the project supervisor, Dr. John O'Donnell, to help specify the requirements further and allow for a thorough plan to be created for the future development of the project.
Prior to the interview, it was already decided that the program would solve life and death problems using the tree search of minimax and alphabeta, due to the reasons specified previously. This meant that the interview questions themselves were targeted at understanding the complexity of solvable life and death problems for the project, the artificial intelligence's capabilities and the production of the graphical user interface utilised by the individual playing through created Go problems. These created questions allowed for a complete problem solver specification as well as an organisation of team member's tasks and milestones at the end of the requirements gathering process to be formed.

3.1.1 Interview Questions

Nature Of Problem

What are the types of objectives the program will be asked to solve?

- It is hard for the program to determine the nature and goal of a problem as they are laid out in the problem book. The program would have to choose its own boundary for the problem and determine which pieces are irrelevant to the overall solution of the problem. This is too

ambitious, so a boundary should be defined by the user along with a nominated group of stones that should either be captured or saved.

What difficulty of problems (30 Kyu - 9 Dan) should the program be able to solve?

- It should be able to solve problems correctly at as strong a level as possible. At a minimum, it should be able to solve simple problems using brute force. The program should be tested with problems of an estimated strength against a competent human player to find the kyu ranking of the program. 20 - 25 Kyu is around the right ranking that the program should be able to achieve. If the program ends up better than this, problems of better ranking could be used to improve it further.

How complex, as in the number of moves before the solution, should the solvable problems be?

- The number of moves to solve is a useful metric for calculating complexity when we use a brute force algorithm, however for more complicated heuristics it becomes less useful. A directory of collections of test cases should be created so that many tests cases are able to be ran over the program and calculate the complexity. However calibration is difficult as humans and computers find different things difficult. Humans can perceive good moves and can deliberately structure the tree deeper rather than wider by forcing the other player to make specific moves to save their pieces.

Solution

How long should the AI have to respond?

- The program should have no time limit to solve, but it should be reasonable ie. making a move in a few seconds or minutes compared to days and weeks. The interface should have a way for the user to define how long the AI has to take a turn as thirty minute turns are too long in practice. To control time, a difficulty chooser could be implemented that would set time limits for turns or set depth limit for move searches. This should be a rough guide and not a set limit ie. if the time limit is set to 30 seconds, the AI can take from 10 - 60 seconds. As long as the times are reasonably similar.

Whom should the AI be able to play against?

- It should be able to play against itself or against humans.

What minimum success rate should the AI have?

- The program should be able to solve 28 kyu problems with certainty. It may be unable to solve 15 Kyu problems reliably but may solve them on occasion.

How do we know the AI has succeeded?

- A main group could possibly be elected to either be captured or be saved depending on problem. If the group is captured, or is unable to be captured then the program could be alerted that the problem has been solved. Pitting the AI against itself is not a good way to test the successfulness of our program, so the program should be tested with humans who know how to solve the problems so they can pick out problems in the program. There are two avenues that the testing process should pursue, the first being Small data set; in depth evaluation, where good Go players are asked to make good moves against the AI on a problem and then play again but make bad moves. Then they should be asked if they think the AI is good in the program. Ideally, there should be 5 (minimum) to 10 (ideal) testers for this step. The other testing avenue is Large data set: low depth evaluation, where a person, aided with the answers to the problems, is asked to test the first move the AI makes of many problems (several hundred). If the AI makes the correct first move, it can be assumed that the program is likely able to find the correct solution.

Graphical User Interface

Is a textual user interface required in addition to a graphical user interface?

- The program could start in TextUI, then have a command to start GUI but the ability to be able to switch between TextUI and GUI is not needed. A TextUI is important for the portability of the program, as it can be hard to get GUIs to work on different systems.

What information should be shown from the AI?

- The ability to go back through the moves showing: What moves were considered; What the heuristic evaluated the value of the moves as; Why it did not take moves that would be considered as good; Why it took moves that would be considered as bad. Also, make it possible to query the AI, to ask it what moves it is considering or show the value the heuristic gives for a specific position. This would help to find the bad decisions in AI so that fixes are more easily targeted.

3.2 Requirements Specification And Organization

Following on from the described requirements gathering process involving the conducted interview, a more descriptive requirements specification was created which built upon the originally given project brief:

The resulting program of this project should be able to solve Life-and-Death Go problems. This should be done either independently by an AI and/or with user interaction via a GUI that shows the board and allows legal moves to be made. The AI should be able to be given an objective such as defend group A or kill group B for either colour and be able act accordingly. The problem difficulty for the AI to solve should be as high as possible (on a range from 30 Kyu - 9 Dan) and it should solve the problem correctly with a success rate of at least 80%. This will be assessed by using an unspoiled testing set that meets these requirements.

Using this new specification, the planning of the program began. To enable specific design and implementation tasks to be assigned to members, the major components of the problem solver first had to be identified. These components included three separate entities; the game engine, the graphical user interface and the artificial intelligence. In this case, the game engine acts as a controller between the AI and GUI, by providing board representation, communication abilities and a legal move checker. After this identification of components, the decision to split the project into a series of milestones was made. Therefore allowing for the problem solver to be created in accordance with a working deadline. To determine these milestones, several issues were considered. As the GUI and AI could not work without the underlying game engine, implementing the game engine was to be the first milestone. The AI was identified as being the bulk of the work in this project and hence the GUI was planned to be finished by January during the second milestone. Due to the work needed to implement the artificial intelligence, the basic functionality of the AI was also planned to be completed within this second milestone. Once the GUI and basic AI functionality were implemented, the final section of the program was to implement heuristics for the AI to use to solve many different types of problems. This was planned as the third and final milestone.

The organisational structure of the team evolved and developed throughout the project. In the first milestone, the GameEngine was split up equally between all members of the team, with each person having a section to implement. This milestone was completed in November when a text based UI was also created so that it was possible to interact with the program before the GUI was implemented. At this point some basic tests were also implemented by everyone in the team on the underlying game engine to ensure that the base of the program was robust. After the game engine was implemented, the team was split into two groups for milestone two. One group, consisting of Eilidh, Jamie, and Scott began working on the GUI whilst the other group, being Kiril and Niklas began working on the basic artificial intelligence functionality. A working GUI and the MiniMax and Alpha-Beta algorithms were completed in January after which some tests were again implemented to guarantee the program was working as intended. For the third and final milestone, the GUI team joined Kiril and Niklas in working on the AI until the end of this project in March. For this milestone, everyone began work on implementing heuristics for the Alpha-Beta algorithm to use to solve different types of problems and evaluating the ability of the program. During this stage, whilst the GUI was being used some bugs were discovered and some extra functionality was found to be needed. Therefore part of this milestone was also to iron out these bugs and implement any needed functions.

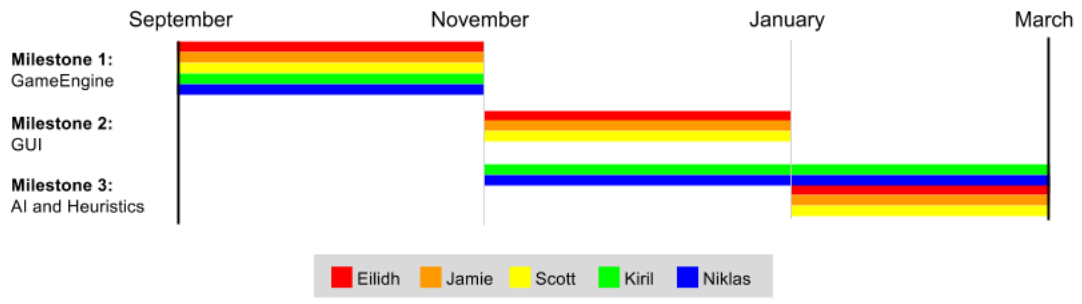


Figure 3.1: Gannt chart showing organization timeline.

The process of the project was a flexible one. Whilst milestones and deadlines were structured out, the revisiting of previously made components and improvements were made frequently. This agile approach to the design and implementation of the problem solver allowed for a final program which

is both user-friendly and includes an artificial intelligence that plays well, meeting the initially specified requirements.

3.3 Design & Implementation of Basic Play

Upon having established the requirements, the first development step towards a problem solver was representing a board and the being able to make exclusively legal moves on it. Further, a board loading and saving mechanism as well as a command-line based UI were also created during the first stage of design and implementation. These components allowed for easier interaction, testing and future development. The basic interaction structure of the program can be seen in *Figure 3.2*. Notably, each of these features were implemented as a Java class, designed to function as fairly independent components.

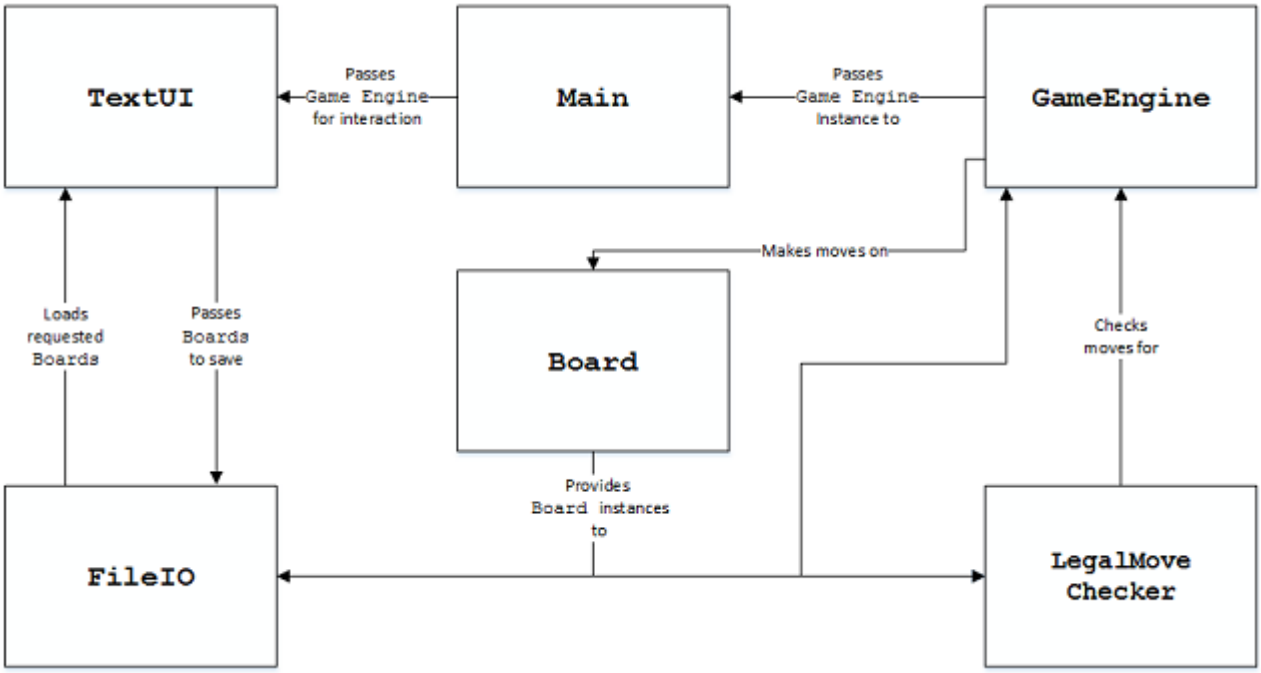
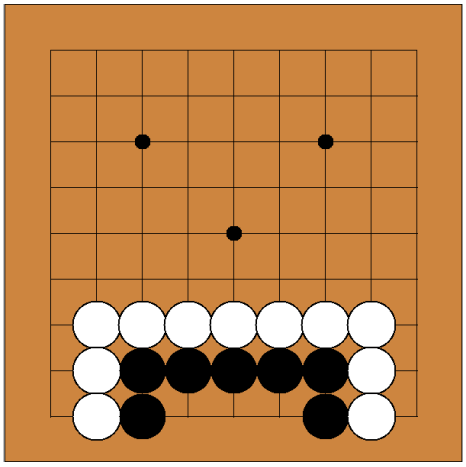


Figure 3.2: Basic program class structure.

3.3.1 Board Representation

The Go game board was represented by the `Board` class, holding the main representation of the board, which is essentially a `byte[][]` of the board intersections. As each intersection of the board can either be empty or hold a black or white stone, these states could be represented numerically as 0, 1 and 2 effectively. An example of this can be seen in *Figure 3.3*.



```
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 2 2 2 2 2 2 2 0
0 2 1 1 1 1 1 2 0
0 2 1 0 0 0 1 2 0
```

Figure 3.3: Board-Array Representation

Notably, a special empty space type (set to 3) was added later on to allow the artificial intelligence to better determine relevant search areas (See *Section 3.5.1*). Other functions of the `Board` class itself are the getting or setting of intersection values, testing for board equality and producing deep copies of the board upon request.

3.3.2 FileIO

The `FileIO` was responsible for reading and writing board states in order to create, store and load libraries of life & death problems. A board could be loaded from a file as a String, parsed into a usable board, modified via a UI and stored again. As the boards were initially shown via a `TextUI` and edited manually, the `FileIO` represented black, white and empty as 'b', 'w' and '.'. The *Figure 3.4* shows this format with the previous example.

```
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 2 2 2 2 2 2 2 0
0 2 1 1 1 1 1 2 0
0 2 1 0 0 0 1 2 0
```

```
9 9
.........
.........
.........
.........
.........
.........
.wwwwwww.
.wbbbbbw.
.wb...bw.
```

Figure 3.4: Array-File Representation

The two initial integers at the start of the board file denote the dimensions of the board. This mechanism was further enhanced as the program grew, having files hold further details such as AI

search spaces, being denoted by '-', as well as game objectives, e.g "White to kill 2 7". These user chosen objectives instruct the AI as to what group of stones is relevant during gameplay.

3.3.3 LegalMoveChecker

The `LegalMoveChecker` was designed to

3.3.4 GameEngine

The `GameEngine` was designed to be the heart of the program, holding the current `Board` representation and allowing players to make legal moves on it. In order to check the legality it holds an instance of the `LegalMoveChecker` that it asks to analyse each move attempted and will only modify the board when said test returns true. Other functions entail undoing moves, restarting the board and other user-oriented features. The respective calls to the AI were also later done through this entity.

3.4 Graphical User Interface Design And Implementation

To create a fully-fledged Go Problem solver allowing the user to interact with life and death problems, a graphical user interface had to be designed and implemented to allow for communication between the user, game engine and artificial intelligence. The finished GUI was created using Java Swing [reference here] and the Graphics package [reference here]. The user interface provides a wide range of functionalities through graphical board representation and the implementation of interactions such as placing stones and saving board states. Hence the final design of the GUI successfully allows the user to create and play life and death problems as specified within the requirements.

3.4.1 Initial Design

During the preliminary design process of the GUI, a paper prototype was created as a basis for implementation. As can be seen in Figure 3.1, the paper prototype showed the original layout of the user view of the GUI which consisted of several elements. These elements included a representation of the board, featuring white and black stones where the human and AI would both play moves. Another component that can be seen are the radio buttons that were drawn out, which would allow the human and computer's stone colours to be chosen. Lastly, an undo button and a reset button were also included in the paper prototype. The undo button would remove the latest move on the board, whilst the reset button would adjust the board back to its original loaded state. The original design also included a toolbar featuring a single button, labelled "Menu". Pressing the menu button reveals the options seen in Figure 3.2 for loading and saving the board, as well as saving a log which would be a series of played moves. Whilst the paper prototype was a strong basis for the beginning of implementation, it was missing a wide range of functionalities, such as bounds and player modes which were included within the final implementation. Some redundant options within the original paper prototype's file menu were also removed during implementation, such as the save log.

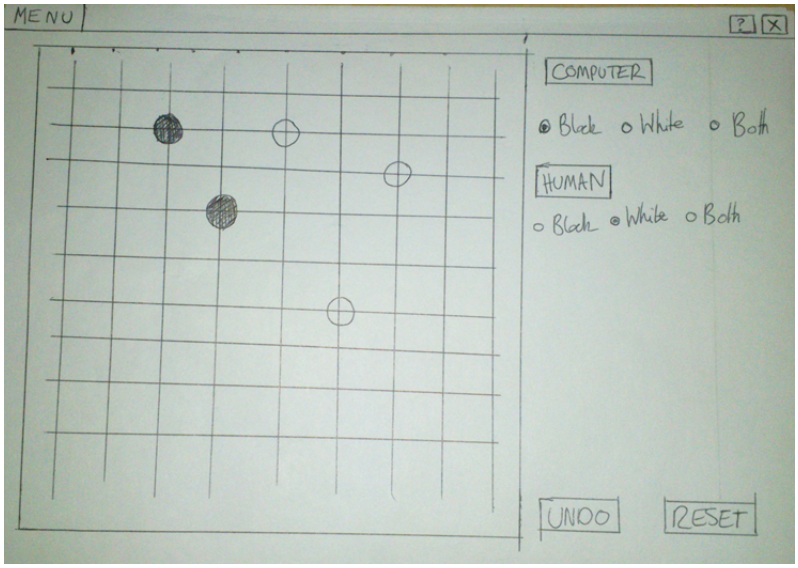


Figure 3.5: Original paper prototype GUI view.

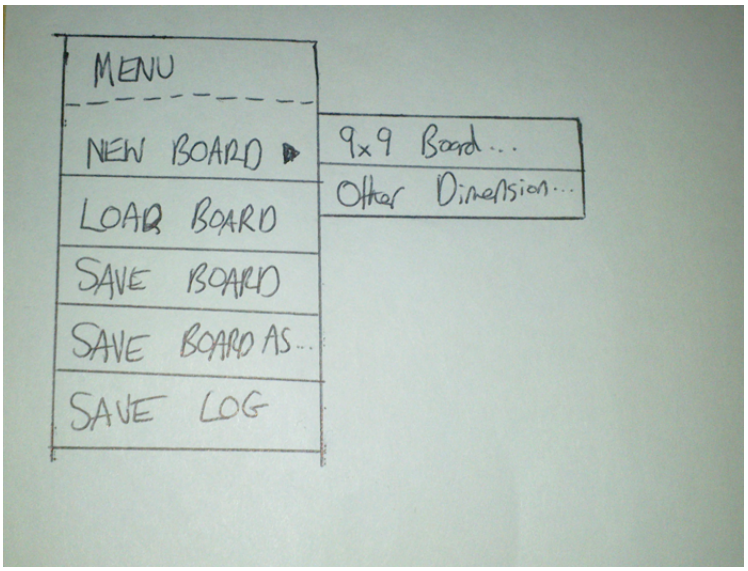


Figure 3.6: Original paper prototype file menu.

3.4.2 Visual Implementation

The first step of implementing the GUI was the graphical board representation, which can be seen as the main visual feature of the problem solver. As described previously (see *Section 3.3*), the GameEngine class uses a 2D array to represent the board within a Board object. This 2D board array had to be translated to a graphical representation to enable the GUI to communicate directly with the game engine and AI as well as the user. As the GUI was created using Java Swing [reference here], the Graphics package [reference here] was utilized to provide a visual of the board. To depict the board on screen, the current number of board lines (height/width) are taken from the Board object specified within the GameEngine. A series of brown rectangles are then drawn and filled in

to represent the board using the number of specified board lines. These rectangles overlap to form the appearance of a grid and, again using the line number provided, coordinates are drawn aligned to each of the rectangles. The user can control whether these coordinates are shown or not and they can be a useful aid during play. This described method of drawing out the board means that the GUI is flexible to representing any size of board specified by the user, as long as it is a square. Following on from the successful drawing of the board layout itself, the GameEngine's 2D array board representation is scanned and when a stone was found - being '1' for black and '2' for white - the appropriate counter is drawn on the board. The counters are painted in a top-left across and down fashion, comprising of a fully coloured circle with a given edge around them. Hence, a full board containing a grid, counters and available coordinates is drawn as seen in Figure 3 using the 2D Board object array provided to the GUI.

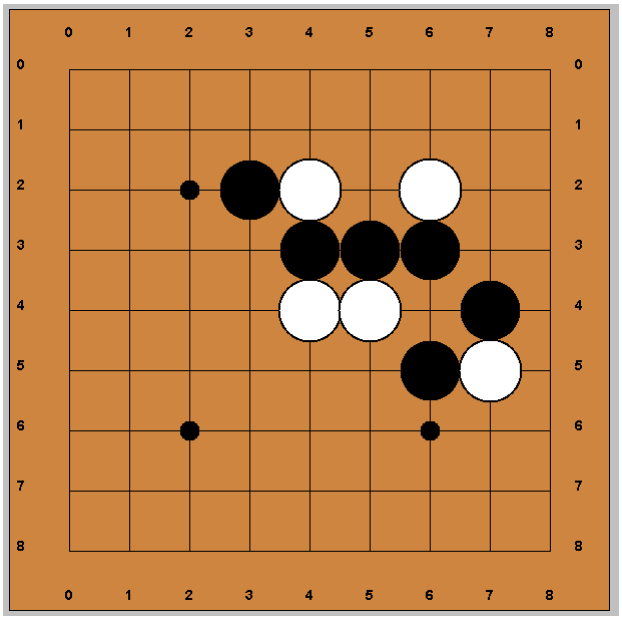


Figure 3.7: Full board (grid, counters and coordinates).

Several other visual enhancements were made following the original drawing of the board, allowing for a higher level of feedback and a more user-friendly system. These included the use of transparent stones upon the board when the user's mouse hovers over an intersection showing where a stone can be placed upon the board. These transparent stones are colour-coded and are either black, white or red - which signifies an illegal move through the utilisation of the legal move checker, an important part of the GameEngine as described previously (see Section 3.3). The see-through stones are depicted in Figure 4 and were also implemented using a 2D array, likewise to the current board representation. This 2D array refreshes itself each time a mouse move is detected. After this refresh, a stone is placed within the transparent stone's 2D array at the current location of the mouse. Checking whether the current stone location is legal or not occurs after the updating of the 2D array and once the check occurs, the appropriately coloured see-through stone is displayed.

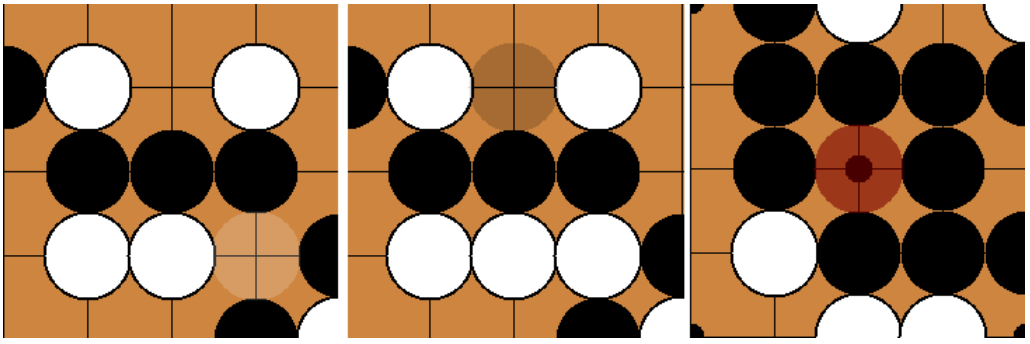


Figure 3.8: Transparent stone: White, black, illegal move.

Yet another visual enhancement within the GUI can be seen in Figure 5, being the depiction of the artificial intelligence's search space as bounds. These bounds are depicted graphically and represent the board positions that the AI will consider moving to. The intersections that are available to the AI are coloured brown, whilst the rest of the board is greyed out. This graphical feedback allows the user to easily see the multiple positions which the computer will possibly move to and this feedback can be toggled on or off according to the user's preference. Originally the bounds took only the shape of a rectangle or square between two specified user points, however through a process of improvement the finished GUI allows the user to select a flexible set of bounds simply by clicking on the intersections to be included within the AI's search space. This change not only provided better visual feedback but was also more efficient for running the artificial intelligence whilst in-game.

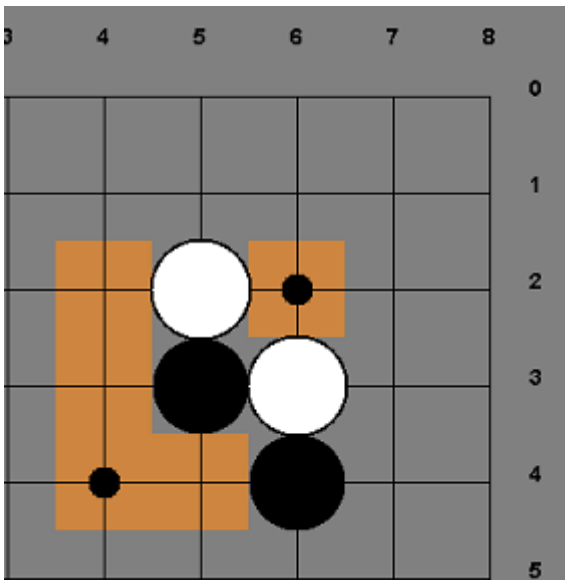
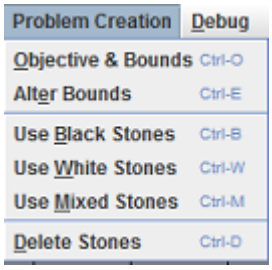


Figure 3.9: Artificial intelligence's graphically represented bounds.

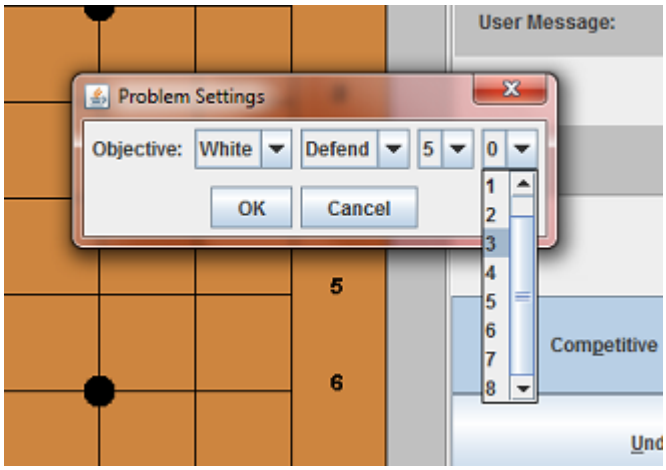
3.4.3 Player Modes

Following the implementation of the board representation itself, as well as several other visual features, a major design change from the paper prototype took place. The design decision was made

to implement two separate user playing modes, being "*Creation Mode*" and "*Competitive Mode*." The availability of user modes allows for easy distinguishing between the two main features of the Go problem solver - being allowing a player to create a problem or playing out a problem. The creation of a problem provides the user with various options whilst playing out a problem allows the user to choose between playing against the AI or another human. The separation of the two modes also rids the GUI of the possibility of having overlap of certain features between modes, which could create numerous errors. For example, deletion of stones is only available in creation mode. Each mode also has their own dedicated menu upon the toolbar, named "*Problem Creation*" and "*Competitive Play*." When the GUI is initially started, it is opened in problem creation mode and displays a blank 9x9 Go board ready to begin letting the user design their own life and death problems. The problem creation menu in the toolbar, as seen in Figure 6, allows access to various features during the creation of problems within the GUI. These include options for the placement of stones, such as using only a specific colour of stone (white/black) or being able to delete stones by pressing on them. Problem creation mode also allows the user to alter the AI's objective and bounds. The objective is specified in a pop up box, as displayed in Figure 7, whilst the bounds can be altered by the user selecting bound selection and then specifying their bounds by selecting specific intersections. This collection of features easily allows for problems to be made quickly and can then be saved by the "*Save Problem*" option in the file menu, ready to be played in competitive mode. The user may also select heuristics during problem creation mode which Alpha-Beta will make use of during play by selecting the "*Heuristic Chooser*" from the problem creation menu. A list of all available heuristics will be displayed as seen in Figure ?, and allows the user to choose any which are relevant to the life and death problems they wish to solve if they choose to play against the Alpha-Beta AI.



(a) Problem creation menu.



(b) Objective dialog choice box.

Figure 3.10: Creation mode user options.

/*INSERT HEURISTIC CHOOSE MENU FIGURE*

Competitive mode meanwhile is intentionally selected by the user after the initial opening of the GUI. This mode allows for the play through of a problem in several approaches, including human versus human, human versus AI (either Minimax or AlphaBeta) and AI versus AI. Upon selection of the "*Competitive Mode*" button a pop up box appears allowing for player choice as seen in Figure 8. If the bounds and objective are not specified before this selection, the GUI will prompt the user to choose them before allowing competitive mode to be entered. Once a game is in play,

the problem creation menu is greyed out and the competitive play menu is available to use. This menu includes such features as swapping player colours, forcing the AI to move and also allowing the user to change the AI type currently being used within play. An example of an AI move can be seen in Figure 9 and Figure 10, where labels on the side of the board specify the AI type and its move selection which allow the user to easily learn from its position selection, as seen in Figure 11. Hence the options available in competitive mode allow for a flexible and user friendly environment whilst playing through a problem with the user interface.

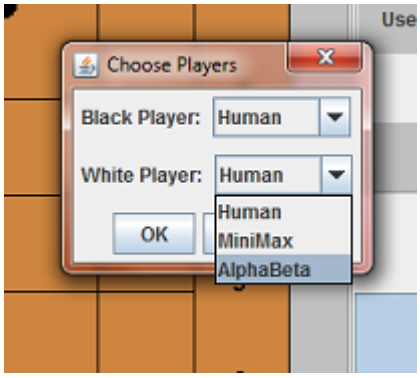


Figure 3.11: Player dialog choice box.

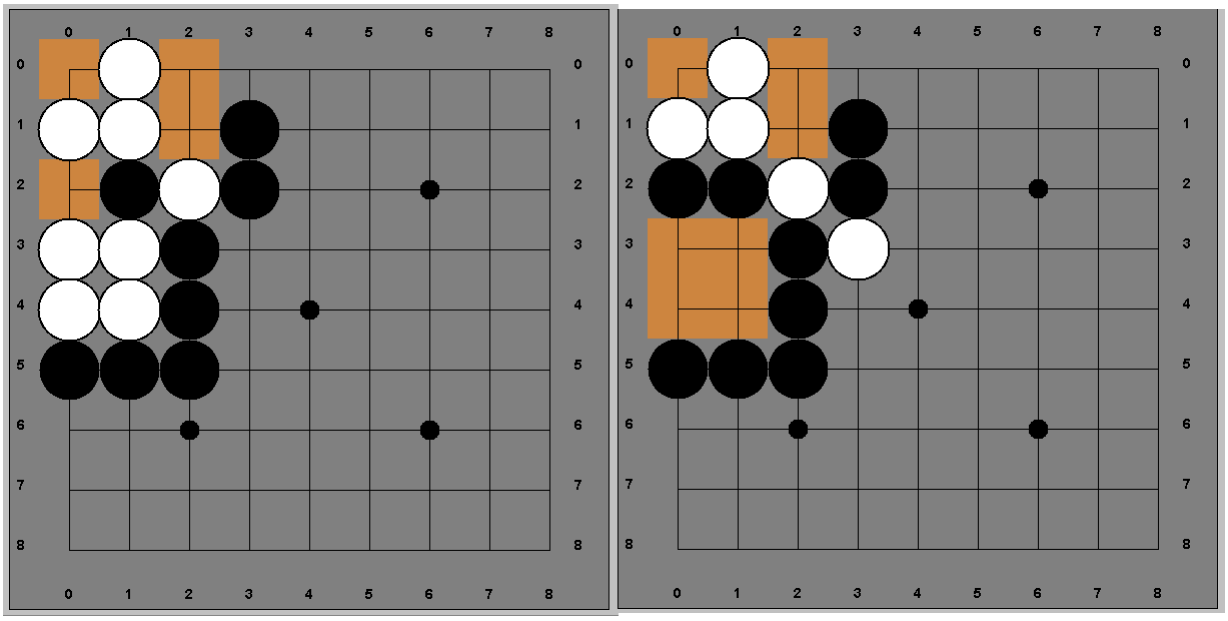


Figure 3.12: Example of AI moving (successfully capturing black stones).

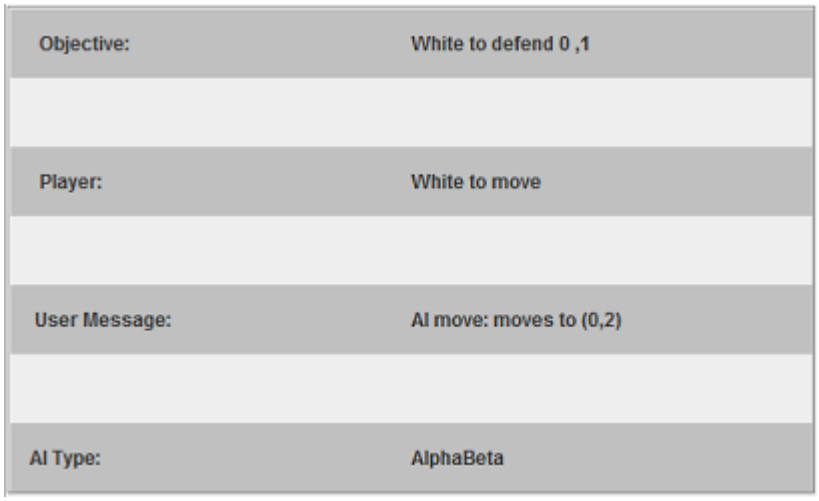


Figure 3.13: AI feedback following its move.

Several features of the GUI are included in both creation and competitive modes. These features include the "*Show Coordinates*" button which displays the coordinate numbers along all sides of the board, allowing easy stone placement. The user can also load or save problems at any point as well as using debug features. Debug features include displaying the log, being the history of all actions within the GUI. Also, the usage of the help menu is available in both modes which has options including the display of keyboard shortcuts. For example making the AI move against itself in AI versus AI mode can be quickly done using $ALT + I$ and there are several other keyboard shortcuts which enable even quicker use of the GUI. Other useful functionalities provided in both modes include the undo and reset button, allowing for the quick backing up of mistakes or reset to the original board, which is practical at all times for the user.

3.4.4 Finalised Graphical User Interface

Through the evidence given prior, it is clear that a user interface was completed successfully allowing for easy interaction during problem creation and competitive play. The entire finished view of the GUI can be seen in Figure 12. The final implementation of the interface allows for the quick creation and saving of problems before the user is able to play them in a variety of ways and against several different entities, including the AI and other human players. The GUI also has a wide range of useful features, such as making the AI's functions, including search space and heuristics, available to the user, as well as having a clean look and layout.

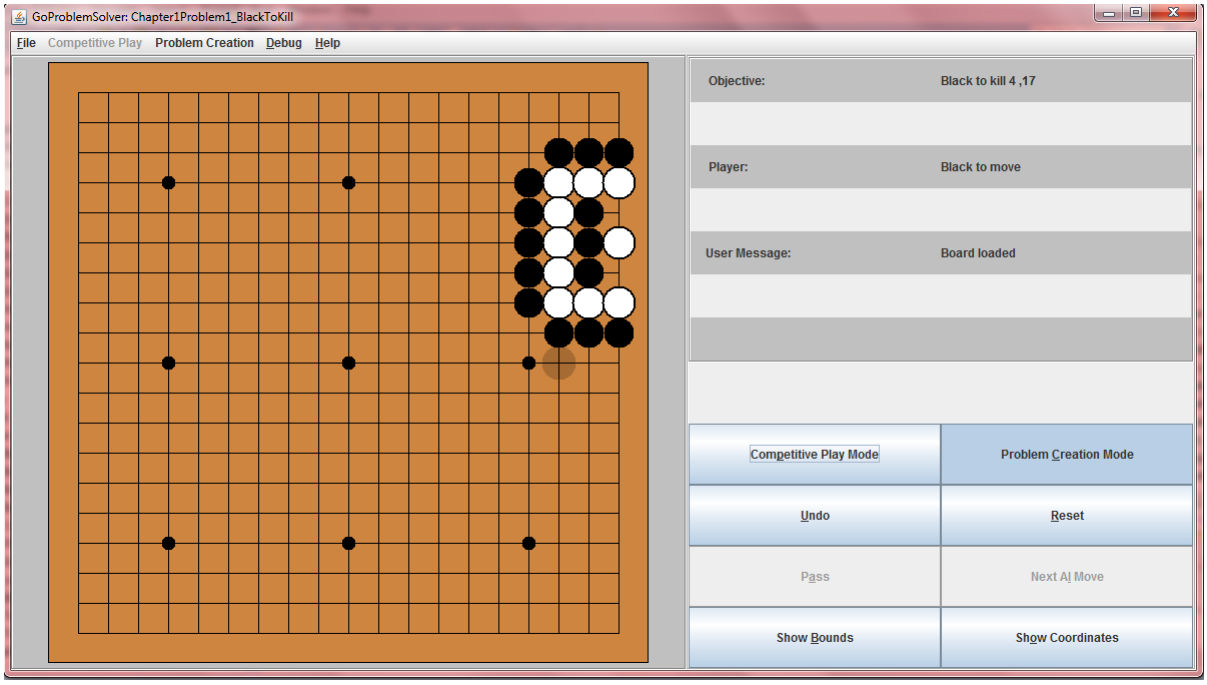


Figure 3.14: Full view of the final GUI.

3.5 Artificial Intelligence Design And Implementation

The aim of our program is to play interactively with the user through a solution of a given life and death problem. In order to do that, the program needed to be capable of solving the problem by itself by making use of move look ahead. Accomplishing this required an extensive artificial intelligence to be created, which can make human-like moves. Typical life and death problem involve a board state comprising of various stones and an objective to be achieved relating to these stones. The goal of the artificial intelligence is to find a sequence of moves in the given situation that satisfies this objective. The use of an objective gives the program the ability to respond properly to any move made by the user, or by the program itself in AI vs. AI game mode. As the artificial intelligence is of major importance within the problem solver, the design of it had to be thoroughly considered. There are several ways it could have been implemented, including using a tree search algorithm like MiniMax or Alpha-Beta, which is the route that was chosen within this project.

A tree search algorithm considers each possible action at a given moment alongside each of the subsequent possible actions. It continues this action search until a state is found that satisfies a given goal. This allows for brute-forcing of life and death problems in an attempt to find a winning sequence of moves. When the search reaches a state where the objective is completed or where there is no chance to complete it in future, the corresponding result is returned and the move with highest score is picked. By integrating this brute forcing functionality into the problem solver, it can successfully brute force and solve user defined problems.

3.5.1 Search Scope And Objective

As described previously, as all possible moves are considered during brute forcing of problems, the larger the number of possible moves, the slower the response time of the artificial intelligence.Therefore, to get a decent result in reasonable time, the program should not deal with problems that involve considering a large number of legal moves. To solve that problem and to avoid considering irrelevant empty intersections the implemented search algorithms take into account for possible response only those legal moves that are marked as bounds during the GUI‘s problem creation mode. If capturing occurs during the solution the corresponding empty intersections are as well later considered in the computation of the response, as they are relevant to the problem.

To actually know when to stop searching the algorithm needs something to tell it that it has completed the goal or that it failed to do it. This is achieved with the help of the Objective class. It stores the coordinate and the color of a stone, which is part of the group to be captured or defended, and the action that the AI has to perform either kill or defend. Every state founded by the tree search algorithms is then evaluated using the Objective to determine if it is terminal or not. If the action is kill, the Objective checks if the stone is on the board. If it is not it has been captured, the goal is met and a winning sequence is found. This is clearly a terminal state, which is assigned with the highest score. Another terminal state for action kill is the case when the stone is on the board and there are no more legal moves. This means that the goal is no longer achievable and the state is assigned with the lowest score. Similar checks are performed for action defend.

3.5.2 Minimax: Design And Implementation

In games like Go where two players are involved the search tree algorithms become more complicated. They have to take into account the assumption that the opponent also chooses the move that gets the best result. That is the case with the Minimax search, where there are two types of nodes a max and a min node. The max node is where the player tries to maximize his score and the min one is where the opponent attempts to minimize his. The leaf nodes are the terminal positions of each sequence and contain a value illustrating the result of the particular sequence. Starting from there and going up the tree the max nodes get the maximum value of its children and the min nodes get the minimum of its children nodes. At the end, the root stores the value of the best move at that moment.

The implementation of the Minimax within the problem solver follows the stated description, but the actual tree is not constructed. Instead, each move is created recursively. The type of the AI, either Minimax or Alpha-Beta, is chosen from the GUI by the user. If the Minimax is selected, it is instantiated and receives a stone colour as well as the objective for the current problem. On every move that the computer has to make it gets the current board and runs a recursive function for every legal move it finds provided by the legal move checker. The next step of the recursion is to consider every legal response that the opponent could perform. This recursion and comparison continues until a terminal state is found where either the computer wins, loses or simply cannot improve the situation by placing stones and has to pass. The values of the outcomes are then passed back and the results of all the initial moves are compared. The move with the highest result is chosen as a reply and its board coordinates are given to the game engine to actually perform the move.

A significant improvement was made during the implementation of the algorithm. Initially in the design on every recursive move generated a separate function had to create all the possible boards,

which was found inefficient. It was replaced by a loop which iterates over the board and when a legal move is found the recursive function is called for that move.

3.5.3 AlphaBeta: Design And Implementation

Alpha-Beta search is similar to the MiniMax tree search. In fact they are guaranteed to always give the same result, but Alpha-Beta has better performance, since some of the branches of possible future moves are not investigated. The algorithm maintains two values, alpha and beta, which represent the maximum score that the maximizing player is assured of and the minimum score that the minimizing player is assured of, respectively. Initially both players start with their worst possible score - alpha is negative infinity and beta is positive infinity. It can happen that when choosing a certain branch of a certain node the minimum score that the minimizing player is assured of becomes less than the maximum score that the maximizing player is assured of (beta¡=alpha). In that case, the parent node should not choose this node, because it will make the score for the parent node worse. Therefore, the other branches of the node do not have to be explored. This property of the algorithm decreases significantly the time needed to find a good move when brute-forcing life and death problems.

Within the program, the implementation of Alpha-Beta does not differ much to the one of the MiniMax and again the actual tree is not constructed. The values of alpha and beta are maintained in order to break a particular recursion if the situation described previously occurs where beta becomes less than or equal to alpha. The connection of the Alpha-Beta with the other components in the program is the same as it was for the Minimax, allowing easy selection of either artificial intelligence by the user.

3.5.4 Performance Comparison

Both MiniMax and Alpha-Beta are implemented to count the number of moves considered during the computation of each move the AI has to make. This feature allows for performance comparison between the two algorithms. The major performance advantage of the Alpha-Beta is illustrated when the program solves Seeing under the stones life and death problem.

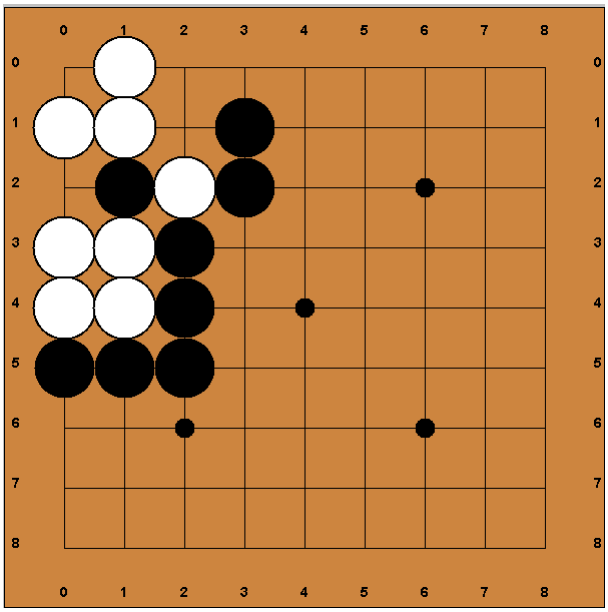


Figure 3.15: Seeing under the stones Go problem.

As can be seen from the figures below, to find the correct initial move here (2,1) in order to defend the white group, Alpha-Beta considers 43803 and MiniMax 60467605 moves.

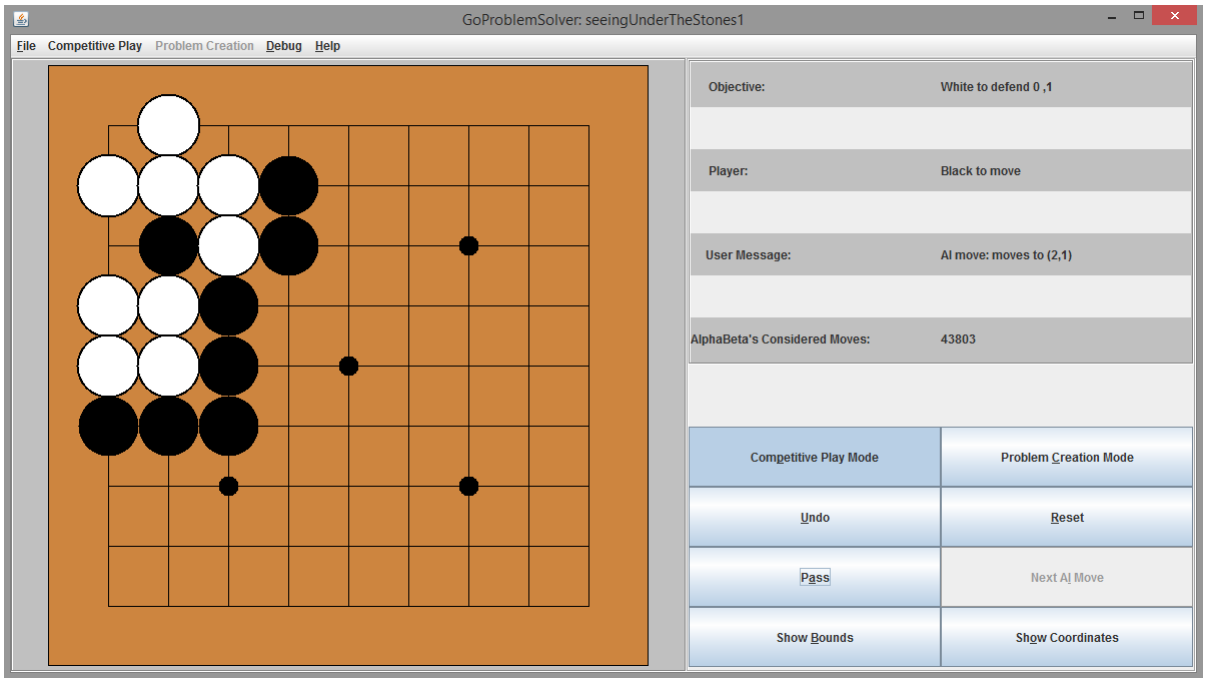


Figure 3.16: Alpha-Beta initial move to Seeing under the stones Go problem.

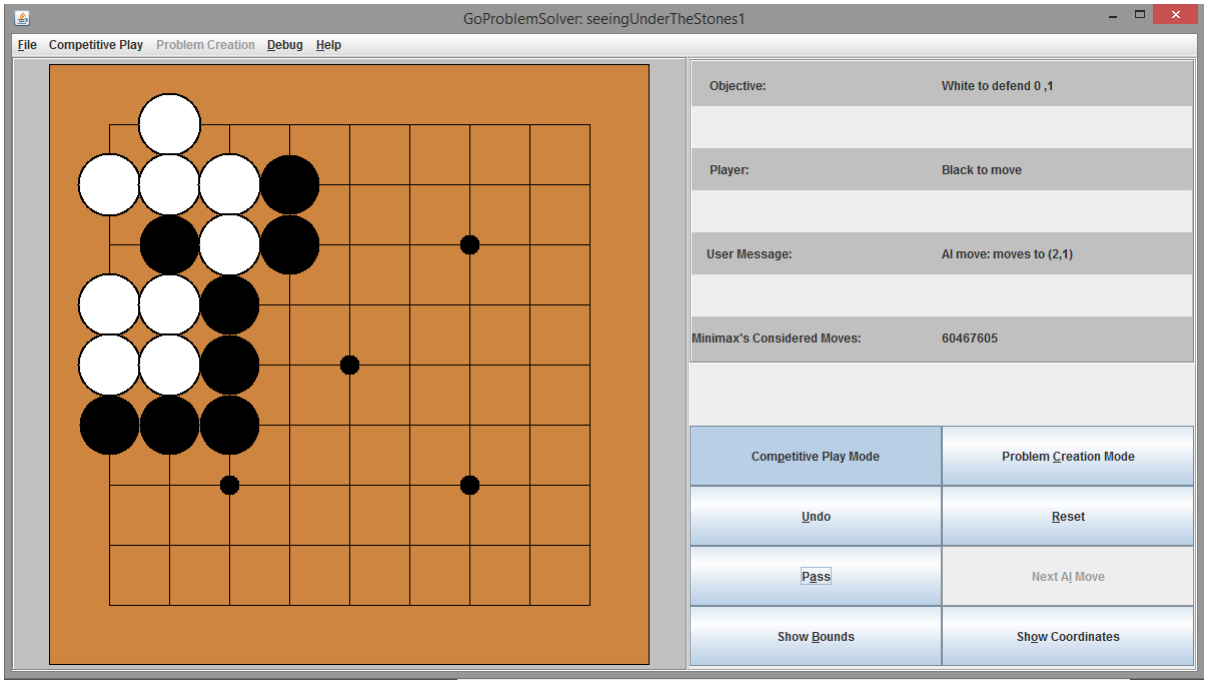


Figure 3.17: MiniMax initial move to Seeing under the stones Go problem.

Being able to solve this problem and many others during the implementation and evaluation states of the project, suggest that we successfully managed to implement two functioning tree-searching algorithms for move selection.

3.6 Heuristics Design And Implementation

Move selection through minimax and alpha-beta allows for a variety of problems to be solved when making use of brute force alone. However, the more complicated life and death problems take a huge amount of time due to the number of exhaustive moves considered and there is no real strategy used by the program when choosing moves in the brute force case. Hence, to improve the performance of the problem solver, a wide range of heuristics were also implemented to be used in conjunction with alpha-beta. Heuristics are used to provide artificial intelligence with game strategies and are fundamental to the creation of a game playing program which is more able to mimic a human player's thought process during play. They allow the AI to quickly pick out moves through the implementing of evaluation functions which recognize specific patterns and highlight strong moves algorithmically. In effect, implementation of heuristics within this problem solver enables the artificial intelligence to play more effectively against a human with previous knowledge of life and death problems tactics.

The heuristics added to the problem solver function by implementing a heuristic interface which provides one method to be overridden which generate an evaluation score. As parameters, the method takes both an initial board and a current board. The initial board is the board state before alpha-beta begins considering moves and the current board is a board state with possible future moves placed on it. From there, the heuristics also make use of the AI's colour, the problem's objective and an instance of the legal move checker. This allows each heuristic to generate a current

score by comparing the initial board and alpha-beta's current board, returning a rating dependent on whether they evaluate alpha-beta's chosen moves to be productive towards the problem's objective in favour of the artificial intelligence.

3.6.1 General Heuristics

Several general heuristics were created, enabling the AI to choose advantageous moves across various problems. This means that even if alpha-beta were faced with a more free-style Go problem, as opposed to a life and death problem, it would still be able to play to a relatively strong level. Many of the heuristics specified below relate to moves and problems discussed in *Chapter 2 - Go Background*.

Eye Creator: A heuristic allowing for the creation of eyes with at least four stones surrounding one empty point. Hence, this also enables the problem solver to create multiple eyes next to each other and completely full eyes (e.g. one empty point with eight stones surrounding). As a group is alive when it has two eyes, this heuristic is key to many life and death problems.

Two Point Eye: As well as eyes with one empty intersection in the centre, the program also recognises eyes with two empty intersections and six surrounding stones. The heuristic allows for both vertical and horizontal two point eyes to be created by the problem solver.

Liberty Counter: By comparing the initial board with alpha-beta's current board containing several future moves, the liberty counter heuristic counts the number of liberties of the objective specified group on each board. Dependant on the objective, the difference between the number of liberties can be beneficial. For example, if the problem objective is for alpha-beta to defend one of its own group, if the number of liberties for this group has risen then the moves on the current board could be evaluated as positive. Likewise, if the problem objective is to kill the opponent group, then a reduction of the number of liberties of that opponent group would again be beneficial to alpha-beta.

Living Space: For this heuristic, the living space defines the space which the player currently defending a group of stones has to create eyes allowing their group to stay alive. The number of intersections within the living space is counted on both the initial and current board. If the current board has more living space for the group, and hence more room to create eyes, then this heuristic returns that branch of alpha-beta look ahead as a potential good move set. As many of the implemented heuristics target killing objectives, this is a highly useful one for problems concerned with defence.

Three Liberties: The tactic of placing a stone in a relatively empty space upon the board is a strong move during play through, as it allows for the future creation of eyes and groups with many liberties. This heuristic checks for intersections on the board with at least three liberties and returns a positive score when alpha-beta's future moves include stones placed upon these intersections.

Several of the general heuristics, also allow for life and death problems to be successfully solved. For example, the three liberties heuristic allows for many Rabbity Six instances to be solved (see *Chapter 4 - Evaluation*).

3.6.2 Specific Problem Heuristics

To allow the AI to successfully solve specific problems, a range of heuristics targeted at solving common life and death problems were also implemented.

Unsettled Three: As described previously (see *Chapter 2 - Go Background*), a set of three adjacent empty intersections surrounded by a group is the unsettled three. The key move is for either player to move into the middle of these intersections, either creating two eyes or disallowing their opponent to create two eyes. Both minimax and alpha-beta brute force successfully complete these problems except for when the pattern occurs corner position. This heuristic evaluates and returns a positive score to alpha-beta when it's future moves include a placement on the key unsettled three intersection, including in corner positions.

Hane: The hane move could be classed as both a general and specific heuristic, as it is a move which could be beneficial in many situations. However, this heuristic was specifically implemented with six die, eight live in mind. The heuristic checks the initial board for an opportunity of a hane, being a move which wraps diagonally around an opponent's group of stones, and will return a positive score if this move was completed on the current board.

Eight Stones In A Row: Another heuristic implemented with six die, eight live in mind, was the eight stones in a row heuristic. Searching for seven stones on the initial board, it allows the alpha-beta to identify moves which will allow for a string of eight stones a long the second lines in from the edges of the board. As any group of stones which is eight stones long is alive, this heuristic allows for the AI to easily stay alive and also have the opportunity to create two eyes.

Six Stones In A Row: This heuristic completes an initial board which already has five stones in a row and completes this to six stones. In the six die, eight live problems, a row of six stones is dead. However, if this row of six stones is on one of the second lines in from the edge of the board and at a corner, then the row of stones is alive - referred to as the four die, six live set of problems. This corner situation is the one which the heuristic specifically looks for and allows for alpha-beta to successfully make beneficial moves in yet another set of life and death problems.

Hence, through the addition of heuristics, the skill level of the program was boosted immensely, allowing for alpha-beta to choose more resourceful moves. Many of the heuristics are demonstrated in *Chapter 4 - Evaluation* and prove their successful design. This implementation of strategy through evaluation functions allows the problem solver to play through problems more akin to a human player, as heuristics and brute force working together allow for suitable move look ahead as well as more comprehensive move evaluation.

3.7 Integration And Implementation Reflection

As described previously, the finished Go problem solver is made up of a series of components - namely the GameEngine, GUI, AI and heuristics. Each of these components had to be integrated together and given communication abilities to create the finalised program. Once integration had occurred, a simple reflection was carried out, allowing insight into any design or implementation changes that would have been made given more preparation time.

Before the implementation of heuristics; the GUI, AI and GameEngine were created separately and then integrated together. The GameEngine easily interacts with the AI through the use of a private AI variable, allowing the GameEngine to access AI search space, the AI type and AI move selection. The more difficult integration was matching the GUI against the GameEngine and hence allowing it to communicate with the artificial intelligence. Despite the difficulties, this was achieved effectively before heuristics were implemented. The integration was carried out by matching GameEngine methods to GUI buttons and several methods were also added to the GameEngine to allow the GUI to directly interact with the AI instance. After integration, a range of bug fixing occurred and the visual representation of the artificial intelligence within the GUI made it easier to identify problems. Heuristics were then added to the AI later and integration was relatively simple in this respect. Figure 1 shows the interaction between the finalised components and the success of integration can be seen clearly.

TBA: ADD FIGURE 1 (UML Diagram of finished implementation/integration)

Looking back upon design, implementation and integration there are several design choices and processes that may have been different with more time availability. For example, if this were a long-running Go project several more heuristics would have been design and coded to allow for a more strategic artificial intelligence. Such heuristics include *ADD HEURISTICS/EXPLANATION* Also, in terms of board representation the choice of a 2D array was probably not the wisest design choice and a data structure such as a hash map may have allowed for a faster running program and less confusion about the laying out of coordinates within the array. Other design choices that can be reflected upon also include more features within the GUI, such as a more realistic board which could have been created using web-coding, for example HTML [3], or a different Java package that was not Java 2D [reference here]. Another GUI feature that could have been added include allowing the AI to run against itself automatically without being prompted against the user. *ADD AI EVALUATION HERE* Hence, there were several design and implementation choices identified after integration that could have been benefited from a different approach and these were expanded on during the evaluation process (see *Chapter 4*).

Chapter 4

Evaluation

4.1 Solvable Problems

To determine the level of quality of the final Go problem solver created, a series of evaluation techniques were carried out. As the original requirements required a problem solver that can solve life and death problems, the most deterministic approach of measuring the success of implementation was through finding life and death problems that the finalized program could successfully complete. Also found were several unsolvable problems that the program could be extended to solve in future through added heuristics. Both types of problems identified during evaluation have been included in "*Solvable Problems*" and "*Unsolvable Problems*" directories that come with the problem solver. These directories are automatically displayed when the "*Load Problem*" option is chosen under the file menu, hence allowing user access to the problems found during evaluation.

4.1.1 Solvable Problems With Brute Force

Using brute force alone, several life and death problems were able to be solved by the problem solver.

Add examples of solvable brute force problems (Include difficulty estimates/time to solve)

4.1.2 Solvable Problems With Heuristics

Once brute force was extended through the use of heuristics, as described in *Section 3.6*, the program was able to solve several new, more specialised problems.

Add examples of solvable heuristic problems (Include difficulty estimates/time to solve)

4.1.3 Not Yet Solvable Problems

Aside from solvable problems, there are also an array of unsolvable problems that the problem solver could be extended to be able to solve in future. *Section 3.7* describes several heuristics that could be added with further development, and the following unsolvable problems relate to these specific heuristics.

Add examples of not yet solvable problems (Include difficulty estimates)

The original requirements (detailed in *Section 3.2*) desired a problem solver which could solve 15 Kyu problems, being of around a median difficulty, and a success rate of 80

4.2 Subject Testing

Whilst evaluation through the finding of solvable problems was carried out by the team itself, testing using subjects outwith the team was also performed and provided valuable feedback. A testing document was prepared (see Appendix [insert number here]) and was given to participating test subjects. The document asked the tester to input a set problem given, or their own problem, into the program and attempt to run it in a mode of their choice: AI vs. human, AI vs. AI or human vs. human. Following this, the testers were asked to write down their feedback on the general problem solver, as well at its two modes: problem creation and competitive play. Before beginning testing, participant's consent was acquired, they were given the team's contact details and they were also informed them they could withdraw at any point during testing. Once testing was complete, the testers participated in a discussion focused on the evaluation. Hence, subject testing was carried in accordance with ethics procedures.

4.2.1 General Interface Testing

Through the evaluation conducted by real life subjects, a wide range of feedback was supplied concerning the general interface of the problem solver itself, including graphical representation and layout.

Insert interface feedback/evaluation

4.2.2 Problem Creation Mode Testing

As described above, participants were asked to create a problem within the problem solver during problem creation mode. The problem itself was simple and required a range of white and black stones, as well as user set objective and bounds. User attempts at problem creation provided a wide range of feedback relating to the creation of problems within the problem solver.

Insert problem creation mode feedback/evaluation

4.2.3 Competitive Play Mode Testing

The testing subjects were also asked to play through the problem they had previously created using competitive play mode. They were asked to use any specific mode they wished (human vs. human, AI vs. human, etc.) and provide written feedback on their selection and thoughts on play through following their use of competitive mode.

Insert competitive mode feedback/evaluation

4.3 Other Testing

The most valuable evaluation feedback for the Go problem solver was provided through the finding of solvable problems and user testing, as described above, but other miscellaneous testing also helped to prove the program's degree of success as well as improve it during actual implementation.

Insert description of JUnit tests/attempted bug finding, etc.

4.4 Testing Overview

Overall, a wide variation of testing procedures were carried out using the program, both during development and following its completed implementation.

Describe outcome of solvable problems (number of found problems)/highlight recurring user feedback/reiteration of other testing

Chapter 5

Conclusion

ADDING CITES HERE JUST NOW SO BIBILOGRAPHY DISPLAYS [4, 6, 5, 10, 2, 7, 1, 9]

Bibliography

[1] Bernd Brugmann. *Monte Carlo Go.* Max-Planck-Institute of Physics, 1993.

[2] James Davies. *Life And Death. Volume Four: Elementary Go Series.* Kiseido Publishing Company, 2nd edition, 2014.

[3] Mozilla. *HTML (HyperText Markup Language).* https://developer.mozilla.org/en-US/docs/Web/HTML, 2015.

[4] Oracle. *Java 2D Graphics and Imaging*. http://docs.oracle.com/javase/6/docs/technotes/guides/2d/, 2011.

[5] Oracle. *Graphics (Java Platform SE 7)*. http://docs.oracle.com/javase/7/docs/api/java/awt/Graphics.html, 2014.

[6] Oracle. *Swing APIs and Developer Guides*. http://docs.oracle.com/javase/8/docs/technotes/guides/swing/l, 2015.

[7] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Pearson Education, 3rd edition, 2010.

[8] C. E. Shannon. Computer chess compendium. chapter Programming a Computer for Playing Chess, pages 2–13. Springer-Verlag New York, Inc., New York, NY, USA, 1988.

[9] John Tromp and Gunnar Farnebäck. Combinatorics of go. In *Proceedings of the 5th International Conference on Computers and Games*, CG'06, pages 84–99, Berlin, Heidelberg, 2007. Springer-Verlag.

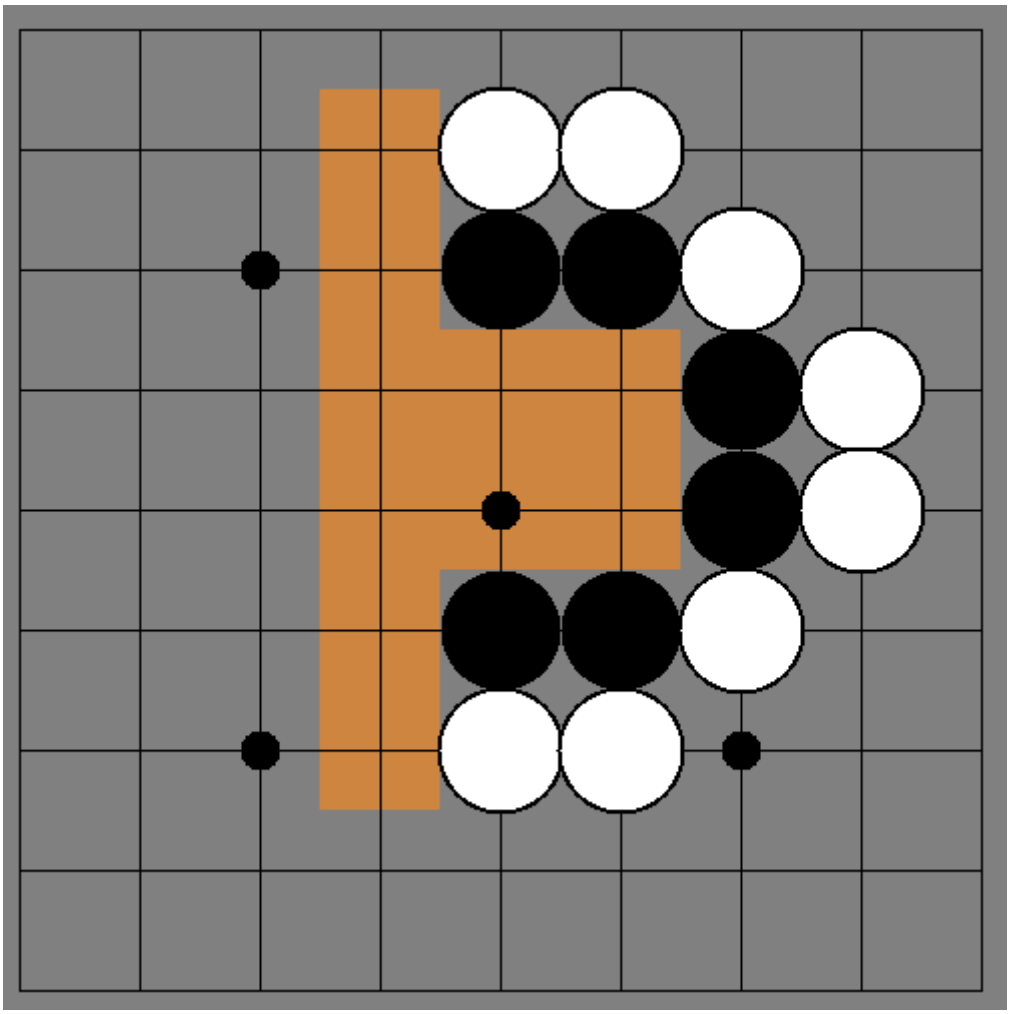
[10] Kano Yoshinori. *Graded Go Problems For Beginners. Volume Two: 25 Kyu To 20 Kyu Elementary Problems.* Nihon Ki-in, 1985.

Chapter 6

Appendix

Team I: Go Problem Solver Testing Document

1. Using problem creation mode attempt to recreate the free-play style problem displayed below upon the problem solver's board or, if preferred, create your own more specialised life and death problem. We recommend using the bounds displayed and the Objective: *White to kill 5,2*.



2. Please answer the following questions about your experience creating the problem.

Did you create the problem above or a new one? (If new: please describe it.)

How difficult/easy was it to create the problem?

What features did you like whilst creating a problem?

What features did you dislike or think were missing?

Any other comments?

3. Using competitive play mode attempt to play the problem in the mode of your choice: human vs. human, AI vs. human or AI vs. AI. We recommend using AlphaBeta for the AI.

Which mode did you choose and why?

How difficult/easy was it to play through the problem?

What features did you like whilst playing through the problem?

What features did you dislike or think were missing?

Any other comments?

4. After your usage of the problem solver, consider the questions below.

Do you find the user interface graphically appealing? Please gives reasons for or against.

Do you find the layout of the user interface easy to use? Please give reasons for or against.

How well do you think the AI played if you used it? Please give reasons for or against.

Any other comments on the general program?