A Go Program for Solving Life and Death Problems

Thomas Reed

School of Computing Science

Sir Alwyn Williams Building

University of Glasgow

G12 8RZ

A dissertation presented in part fulfilment of the requirements of the Degree of Master of Science at the University of Glasgow

7th September 2015

Abstract

This dissertation describes the design and development of a program to tackle problems, known as life and death problems, that arise in the ancient strategic board game of go. Go has long been of interest to computer scientists working in the field of game AI. Due to an unimaginably vast number of possible game states and intractability in defining a function to assess their utility, success has been limited. The largely brute force strategies used to achieve grand master levels in chess have failed with go. However, tackling life and death problems is a rather different prospect than dealing with a full game of go. By constraining the problem to the small areas in which local battles, the life and death problems, are fought, the search space can be massively reduced. With this approach a program was developed, using the conventional alpha-beta tree search algorithm, that is capable of solving and playing out moves in problems of interest to amateur go players.

Education Use Consent

I hereby give my permission for this project to be shown to other University of Glasgow students and to be distributed in an electronic form.

Name: Thomas Reed Signature: TR

Acknowledgements

I would like to thank my supervisor, John O'Donnell, firstly for his well-judged support and guidance throughout the project, and secondly for introducing me to the game of go.

I would also like to thank members of the Glasgow Go Club for kindly agreeing to help evaluate my program. Their suggestions and ideas were much appreciated.

Contents

Chapter 1 Introduction 1

Chapter 2 The Game of Go and Life and Death 3

2.1 The Rules 3

2.2 Life and Death 5

Chapter 3 Computers and Go 6

Chapter 4 Requirements 8

Chapter 5 High Level Architecture and Design Decisions 9

5.1 Initial Design Decisions 9

5.2 Architecture 11

Chapter 6 Development 13

6.1 The Game State Model and File Format 13

6.2 The User Interface and Controller 14

6.3 The Legal Move Checker 16

6.4 Implementation of MiniMax with Alpha-Beta Pruning 17

6.5 The Static Board Evaluator 18

6.5.1 The need for a static board evaluator 18

6.5.2 Go Heuristics 18

6.5.3 Implementing the board evaluator 20

6.6 Next Moves Selector 23

Chapter 7 Testing and Evaluation 24

7.1 Module Testing 24

7.1.1 Testing the alpha-beta class 24

7.1.2 Testing the legal move checker 26

7.1.3 Testing the Board Evaluator 27

7.2 Performance 28

7.3 Evaluating behaviour in real Life and Death Problems 33

7.4 User Evaluation 37

Chapter 8 Conclusions and Further Work 38

Appendix A Problems Solved by the program in under five minutes. 41

Appendix B Problems that the program did not solve in 15 minutes on first testing 48

Appendix C User Evaluations 50

Chapter 1 Introduction

This dissertation concerns the design and development of a computer program to solve a family of problems arising in the fascinating, easy to learn yet hard to master, and increasingly popular game of go. The family of problems of course being the "life and death" problems of the title. The general characteristics of go particularly pertinent to this challenge are discussed in the paragraphs below. A brief outline of the rules for those unfamiliar with go are provided in Chapter 2.

Go is an ancient board game originating in China which some claim to be the most often played game in the world. (Shotwell, 2010) The rules are simple but the winning strategies complex. It is, like chess, draughts and noughts and crosses, a two player, turn by turn, deterministic, zero-sum game with complete information.

It is zero-sum as, at any point in the game, the magnitude of the advantage held by one player will be equal to the disadvantage of her opponent. It has complete information as the game state is entirely represented by the board position, previous board positions and knowledge of whose turn it is. This information is available to both players at all times.

It is deterministic as for any given board position, a set of all possible positions that could immediately follow can be determined. For each positon in this set, a further set of possibilities can be determined and so on and so forth until the diverging paths eventually terminate in a win, loss or draw. A finite game tree of possibilities can be imagined.

Go is played on a relatively large board, a 19 x 19 grid with 361 points where playing pieces (go stones) can be added. No more than one stone is added at each turn and there are few restrictions on where one can play. Both the set of possible moves at each turn (the branching factor) and the average number of moves before an end game are larger than most comparable board games. There is some discussion on exactly how many distinct games are possible (see Schafers (2014) or Number of Possible Go Games at Sensei's Library (no date)) nevertheless it is clear that this number vastly exceeds both the number of atoms in the universe and the number of games possible in chess. For this reason the game tree remains very much a theoretical construct.

The nature of go, that it is played with many stones, each, in isolation, of equally low value but in numerous combinations and permutations of great strategic utility, only compounds the complexity inherent in its vast game tree. Functions that effectively evaluate the utility of a board position before an end game is reached are difficult to conceive, complex and computationally expensive. This is discussed further in Chapter 3 and also in (Bouzy and Cazenave, 2001).

When playing go, a human might scan the board looking for advantageous shapes and patterns in the stones. He might consider this a straightforward task as he simultaneously takes in the positions of tens of stones from above. During this project, it became apparent that this is a rather less straightforward task for the computer. With the board represented as an array the program inspects one point

at a time, stores some information and perhaps moves on to a neighbouring point. This is somewhat analogous to trying to find out the shape of a large forest from the inside versus flying over it and taking a picture, theoretically possible but complex and very time consuming.

There are some serious challenges then to developing programs to play a full game of go, not least of which is the huge search space on a 19x19 board. (Beginners sometimes play on smaller boards such as 9x9 or 13x13) However, it is often the case that local battles affecting just a portion of the board can develop. Especially towards the end of the game these may be hard fought in order to gain a few points advantage. It is in these situations when the type of problem known as life and death becomes most relevant and a reason for go players to study them. As they are localised it is possible to exclude the majority of the board from the problem and massively reduce the search space. Indeed for the simplest problems an exhaustive search becomes feasible.

Through exploiting a reduced search space around a life and death problem, the aim of the project was to build a program capable of finding and playing optimal moves against a human opponent. The program should either play the generally decisive first move itself or allow the human to play first and play an optimal response. The computer should continue to respond to human moves, playing optimally, until a definite win or lose situation had developed. More detailed requirements are given in Chapter 4.

It was hoped to develop a program capable of playing a selection of problems that would be of genuine interest, and challenging enough for at least a beginner go player. The goal was to produce a useful learning tool where it was possible to create, load, save and play real life and death problems against the computer.

(All images of go positions shown in the project were created by this program.)

Chapter 2 The Game of Go and Life and Death

2.1 The Rules

An outline of the rules needed to understand Life and Death problems is given in the following paragraphs. They are few and simple but there are some regional variations in scoring and different versions of the ko rule that are not discussed here. More detail can be found in Shotwell, Huiren, and Chatterjee (2011).

The game is played on a board comprising 19 vertical and 19 horizontal lines. The 361 intersections of these lines are known as **points**. Players, one with black stones and one with white, take it in turns to place stones on the points with the aim of acquiring **territory**. Once a stone has been placed on the board it cannot be moved unless captured by the opposing player, in which case it is removed from the board. Territory is gained by surrounding an area with your stones. In Figure 2.1, black has surrounded 6 points of territory in the bottom left corner making use of the edges of the board on two sides. To the right of this white has encircled 5 points.

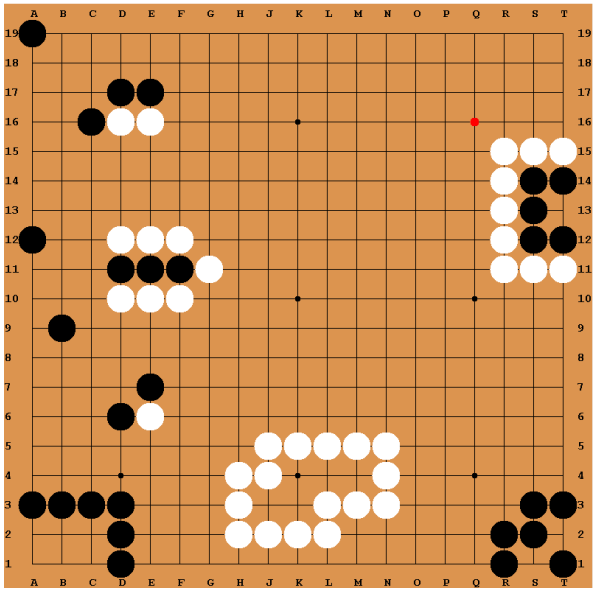


Figure 2.1: The Go Board **(Image produced with the go program.)**

Stones are said to be in a **group** if, imagining the top of the board to be North, they have one or more stones of the same colour to their immediate North, South, West or East. All stones connected like this are in the same group. The black stones at D17 and E17 are in the same group whereas D6 and E7 are not. D10 is part of a group of three stones and D1 a group of 6.

A stone has **liberties** if either it or any stone in its group has an unoccupied point to its North, South, West or East. The single stone played in the top left corner of Figure 2.1 has two liberties, one to the South and one to the East. The white group at E16 has three liberties.

The black group at E11 has one liberty, individual stones or groups with only one liberty are said to be in **atari**. If white removes the single liberty by playing at C11, this group is **captured** and the three black stones are removed from the board.

You cannot play a stone at point where it would have no liberties. White **cannot play** at T2 or S1. However, white can play at T13 as in doing so the surrounding black group is deprived of liberties and thus captured and removed from the board. In assessing the legality of a move in go, you first consider what, if anything, would be captured and then if the new stone has liberties.

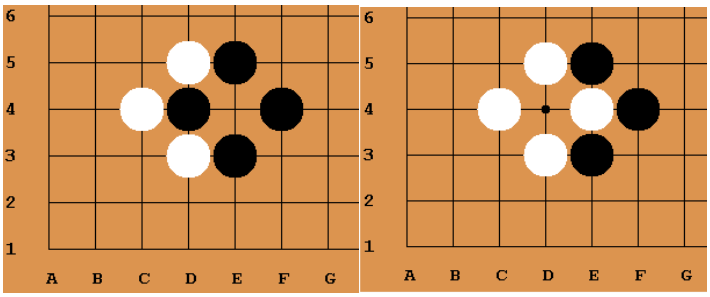


Figure 2.2: An illustration of ko. **(Image produced with the go program)**

Sometimes a pattern develops on the board in which black and white can capture each other repeatedly, this is known as **ko**. In Figure 2.2, if white plays at E4 in the left hand image, black D4 will be captured resulting in the positions in the right hand image. It can be seen that if black were now to play at D4 we would be back where we started. A **ko rule** is used to prevent the immediate recapture when a stone captures another in **ko**. The ko rule chosen for this project is **positional superko** as it lends itself fairly well to computer implementation. Positional superko simply forbids a player from placing a stone if it would result in a board position that has been seen before at any point in the game. There are other version of the ko rule, and according to Shotwell(2010) arguments have existed over it for hundreds of years in Japan.

2.2 Life and Death

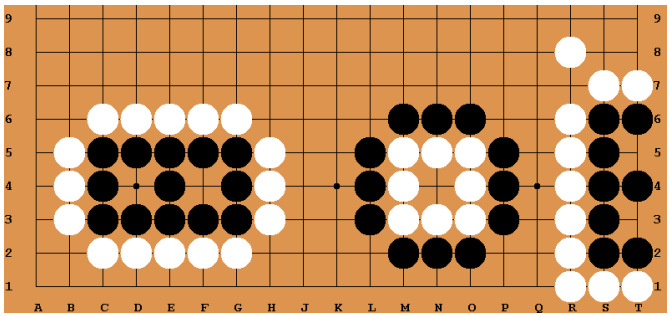


Figure 2.3: Two eyes live, but playing black in the eye at N4 removes white's only liberty and the group dies. **(Image produced with the go program.)**

The white group of stones including M3 above is said to be dead as it is impossible for white to prevent black from capturing these stones by playing at N4. To live a group must form two **eyes**. The black group including E3 has eyes at D4 and F4. White cannot play in either eye as the second eye ensures the group still has liberty and so cannot be captured. This group is alive.

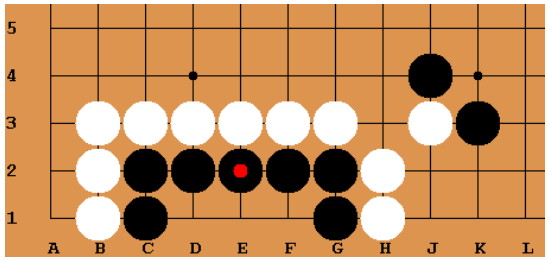


Figure 2.4: A simple Life and Death Problem. The red dot indicates the group to kill or make live.

Figure 2.4 shows a simple life and death problem. If it is white's turn the question is: Where must white play to kill black? If white plays at D1 or F1 it removes two liberties from the black group but black can immediately re-take the stone and form two eyes by playing at E1. The only move that can kill this group is for white to play first at E1, preventing black from forming eyes. Black is now powerless, if it attacks the white stone it loses the space to form two eyes and is dead anyway.

Chapter 3 Computers and Go

In May 1997 the computer program Deep Blue beat world champion Gary Kasparov. Its success was largely the result of brute force. Using the Alpha-Beta tree search algorithm, then the cornerstone of game AI, and massively parallel computing it was able to explore 500,000 positions a second. (Bouzy and Cazenave, 2001) Deep Blue was already playing at the level of a Grand Master.

Four years later, Bouzy and Cazenave (2001) stated in their survey of computer go that the best programs were still much weaker than human players. They went on to give examples of the best programs being beaten by professional go players even with handicaps of over twenty stones. The successful approach used in Deep Blue failed in Go. This failure has been attributed to the far higher number of move choices on the 19x19 board and higher complexity in estimating the utility of board positions. (Schafers, 2014) The branching factor (the number of moves available at each move) is around 40 to 50 moves in chess but can be more than 250 in go. (Schaeffer, Müller, and Kishimoto, 2014) It is both far more expensive to find moves in go and to evaluate the strength of those moves than it is in chess.

Rather than rely on brute force then, efforts were made to better incorporate go knowledge into the programs. In tree search approaches this knowledge can be used to improve predictions and pick winning paths on the game tree. Bouzy and Cazenave (2001) described finding a good board evaluation function as "undoubtedly the biggest obstacle in computer go". However, since breakthroughs in 2006, the most successful approaches now use the Monte-Carlo Tree Search algorithm (MCTS). This algorithm relies on random sampling and far less upon go knowledge. (Schafers 2014) Indeed it is possible to implement a Monte-Carlo go program with no knowledge of go at all beyond the rules. A series of moves are played randomly until a win loss or draw is reached from candidate starting positions. The outcomes of these random moves are evaluated statistically to pick the first move most likely to end in a win.

In 2009 one of the world's best go players, Chou Chun-Hsun, was beaten by a computer running the go program Fuego, an MCTS program. Although only played on a 9x9 board, with its much smaller search space, this was seen as a milestone in go computing. (Schaeffer, Müller, and Kishimoto, 2014) (Exhibition Games: Fuego vs Humans, no date)

This project, of course, is more concerned with building an application suitable for a home pc than running MCTS with the massively parallel high powered computing that may begin to approach the performance of professional players on a full board. There are several commercially available computer programs for amateur go players. The British Go Association maintain a list of commercial go software on their website where the program The Many Faces of Go (MFOG) by David Fotland is given particular praise. (Go-playing Programs | British Go Association, no date). According to the MFOG website the algorithm combines MCTS and significant go knowledge and plays at dan level on smaller boards. (Fotland, no date)

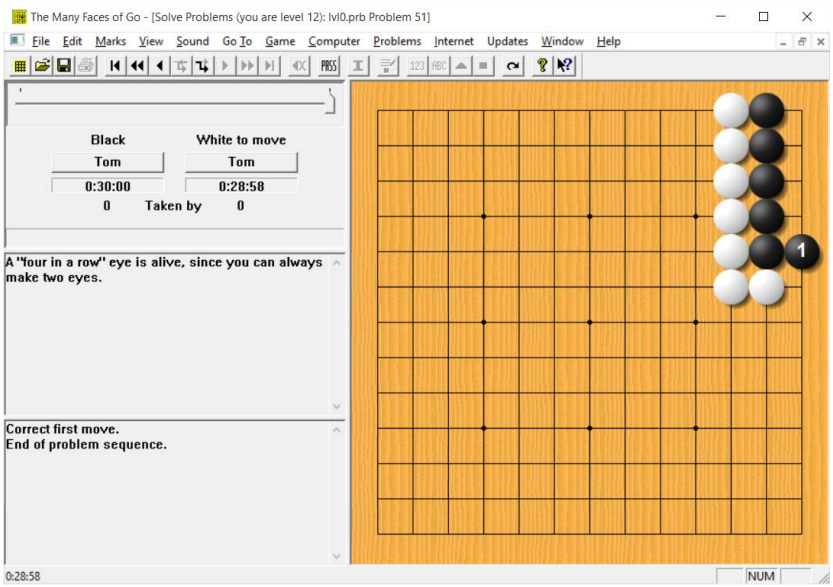


Figure 3.1: A screenshot from David Fotland's The Many Faces of Go showing a life and death problem.

A trial version of MFOG was downloaded for evaluation (the full version is US$90). It comes with some interesting features, such as banks of historic professional games that could be replayed. It also has a "Solve Go Problems" option. In this mode seemingly randomly selected problems appears on the board with instructions such as "Move to capture two stones". There were some life and death problems amongst these as shown if Figure 3.1. The program would only allow play in the "correct move" and once this position had been clicked a note appears explaining why this was the correct move. This is a great learning tool but doesn't allow the user to investigate subsequent or alternative moves. (The problem solution probably just being loaded from file rather than solved). It does seems it is possible to edit, save and set the algorithm to work on any board position in the full version though, so problems could be investigated this way.

Numerous other programs were rated highly on the BGA website and elsewhere including, HandTalk, GO++ and SmartGo, all of which have been contenders in computer Go Competitions. (British Go Association Home Page | British Go Association, no date)

Chapter 4 Requirements

Some though was given to what a useful program might look like and how it might best function. A list of general requirements was drawn up and due to the limited time available was prioritised using the MoSCoW method. This list is presented below. It was hoped that at least those functions in the Must and Should categories would be incorporated into the product.

Must:

- Be able to load life and death problems from file.
- Be able to save problems.
- Display the problem graphically in a format clearly representing a go board.
- Be able to create new problems.
- Allow only legal moves.
- Be able to compute the optimal moves in some simple life and death problems using brute force and display the resulting board position.
- Be able to select whether the user or the program plays the first move.

Should:

- Be able to play out full sequences of possible moves responding to a user's play automatically.
-
- Be able to edit problems and create and save new ones with a mouse from within the GUI.
- Display problems graphically in a format that looks like a go board.
- Have adjustable depth and breadth searches and make use of statically evaluated heuristic values based on good and bad shape.
- Have a folder of initial problems that can be handled by the program available to the user.

Could:

- Allow the user to save notes along with a problem.

Would like to have:

- A complex board evaluator using higher level heuristics beyond scanning for good and bad shape.
- A check for whether the board positon of a newly created game is legal.
- A large bank of go problems that are solvable with the program.

Chapter 5 High Level Architecture and Design Decisions

5.1 Initial Design Decisions

As has been discussed an exhaustive tree search is not a feasible approach for a full game of go. However, many Life and Death problems play out in only a small section of the board, with the majority of points being irrelevant to the survival of a particular small group of stones. If we put constraints on the problem such that all these points are excluded, the branching factor can be massively reduced and tree-search algorithms such as mini-max become realistic.

It was therefore decided to reduce search space by building in a mechanism for excluding points judged to be irrelevant to a problem and to use the minimax algorithm with alpha-beta pruning at the heart of the program. A detailed descriptions of the minimax algorithm with alpha-beta pruning are given in Norvig and Russell (2010) and Nilson and Nils (1998).

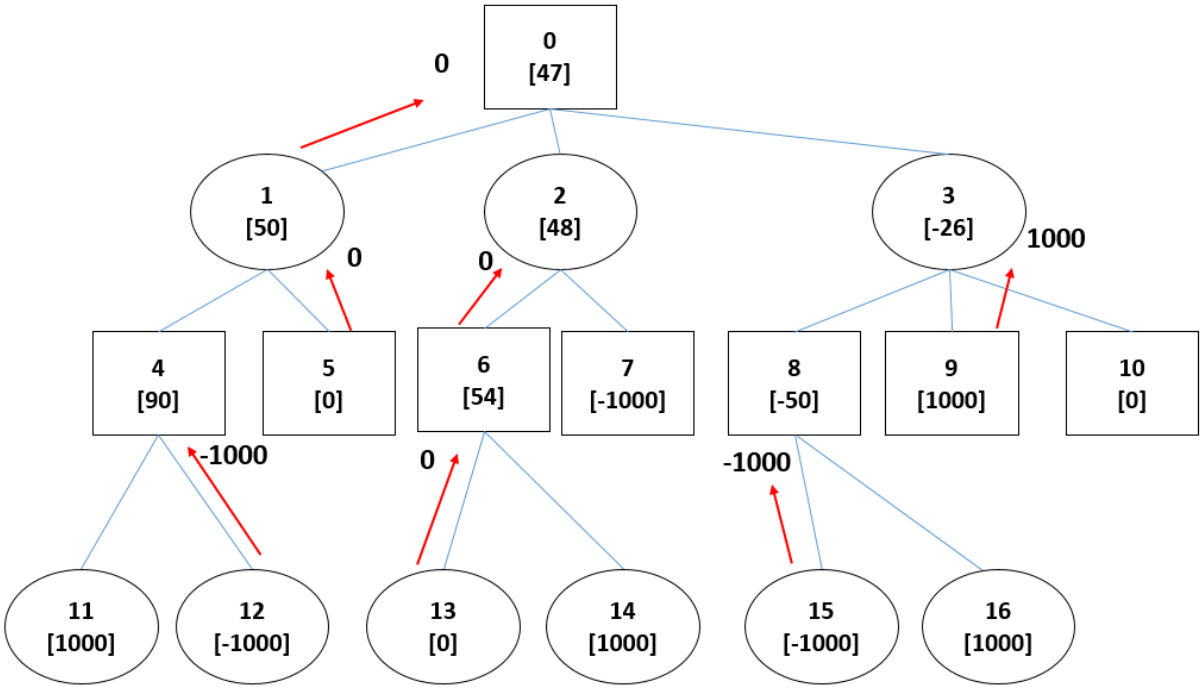


Figure 5.1: A general game tree showing the propagation of leaf values in the minimax algorithm.

The idea of the minimax algorithm is illustrated in Figure 5.1. The complete game tree shows game states 0 to 16. Say a player wants to find out whether a move he is considering that would create game state 0 is going to result in a win, assuming our opponent does her best to stop him. Imagine we have some function that can determine whether a game state is an end of game, (a definite win, loss or draw). Image we have a second function, called **evaluate**, that will return a value of -1000 for games we have lost, +1000 for games we have won and 0 for tied games.

Although not essential for a minimax, there are advantages in having an evaluation function that will also return a value between win and lose designed to give an indication of board strength if the root nodes have not been reached. The detailed design and use of this function is given in Chapter 6.5. The value that would be returned by the evaluation function if called on each node are in the square brackets. Here, they are chosen arbitrarily for illustration and are from the perspective of the player considering move 0.

The rectangles represent the choice of moves available to the player and the ovals the moves that can be chosen by his opponent, thus to evaluate the score of state 0 we take the minimum of the scores of its children (States 1, 2 and 3), as the player's disadvantage is his opponent's advantage and the opponent gets to choose. States 1, 2 and 3 are not end games though, so to determine their score they must be evaluated in a similar way, this time picking the maximally valued child of each as the player gets to choose. This process is continued, recursively, and alternately returning the value of the minimum or maximum value of child nodes to their parents until a terminal node is reached. (A win, loss or draw in an exhaustive search). In this manner values at terminal values are propagated to a root node. The process also gives us the information that if game state 0 does occur, 1 would be an optimal move for the opponent.

The minimax algorithm is guaranteed to correctly score the utility of a move (according to the limitations of the evaluation function) but is wasteful in that it continues to search the tree even when it could be fairly simply ascertained that they will be less fruitful(or no better) than branches already explored. In the simplest case, imagine the first branch of a tree had been evaluated at a minimum node and it returned -1000. There is no point exploring other branches as 1000 is optimal from the opponent's point of view and no lower value could possibly be returned to the parent. Alpha-Beta pruning is a modification to mini-max that prunes unnecessary branches, saving computational expense, whilst still guaranteeing to find at least one optimal path. It works by keeping tabs on the minimum value (alpha) that must be reached at max nodes to improve on the current best score and the maximum value (beta) that must be reached in order to improve at minimum nodes. If a minimum node then has a beta value less than the alpha value of its parent (or a maximum node alpha greater than the beta value of its parent) then the branch can be pruned.

The order in which the branches are explored is obviously important. If there was a single optimal path and this led through to final leaf on all the final branches then there would be no advantage over minimax. In this project the evaluation function mentioned earlier was used to sort the moves, with the aim of processing the strongest contender first.

It was also decided to have an option to reduce the search tree with depth and breadth (the number of branches allowed at each node) limits. This relies on a good evaluation function both to return the values of nodes that have not yet reached end of game and to ensure the strongest contenders are not cut from the breadth. There are several factors than might affect the success of reduced depth and breadth search including the particular game knowledge used in the function and the type and size of problem to be solved. It was therefore decided to have breadth and depth controls within the GUI so they could be easily adjusted to suit a problem. With large problems, reducing the search tree may give the only chance of finding the optimal move within a reasonable amount of time. The design of

methods to statically examine the board that make breadth and depth limited effective was left open during the initial design phase. The process of experimentation and analysis that led to the final design of this module is presented in Chapter 6.5.

5.2 Architecture

It was decided to use an architecture along the lines of Model-View-Controller. For simplicity though, the program control flow and GUI components were incorporated into the same class, GoGUI.java. Initially a text interface was used which displayed a representation of the board in a text area with 'x' for black stones and 'o' for white stones. In later versions the class, BoardImage was developed to draw a more realistic representation of the board. BoardImage took the place of the text area in previous versions with some other GUI components being left in place.

An outline of the responsibilities and relationships of each class is given below:

GoMain: This contains the main method and instantiates GoGUI.

GoBoard: This encapsulates the data needed to define a game state in go. Additionally it holds the location of a stone that must be captured to define either success or failure in a life and death problem. It has getter and setter methods and a copy method to obtain a deep copy of the game state.

GoGUI: This creates and displays the GUI and all of its components. Event and mouse listeners and methods handling related program flow are within this class. The **GoGUI** instantiates a **GoBoard** object modelling the current game state and passes a reference to the **BoardImage** to display the board. An update method is called following any moves.

BoardImage: The board image takes a character array representing the game position from the GoBoard model and draws stones, or out of bounds markers at appropriate locations. A mouse listener is added to the BoardImage object within **GoGUI.**

GoFileHandler: Contains the code for extracting information from a GoBoard object and writing them as character strings to be saved as a text file. It also contains the code for extracting information in the text file format and updating the model. It is instantiated in **GoGUI**.

GameHistory: This holds a record of all previous board positions in order to implement positional super ko (see Chapter 2). A reference to a GameHistory object is held as an instance variable in **GoBoard**. The history is held in an ArrayList<ArrayList<>>. Board positions are added to the ArrayList contained in the index equal to the number of stones on the board. This cuts down search time as it is unlikely there will be many boards with the same number of stones.

MoveChecker: This class checks for legal moves. If instructed to play a move it will, if legal, update the GoBoard model with resulting changes and return boolean true. If the move is illegal it will will return boolean false.

NextMoves: Contains a static method for returning a list of the next legal moves possible. This method is called from **AlphaBetaDB**. The method uses **MoveChecker** to test moves for legality and **BoardEvaluator** to evaluate and sort the legal moves.

AlphaBetaDB: Contains the alpha-beta algorithm. DB stands for depth and breadth to distinguish it from earlier versions that did not incorporate depth and breadth limits. **AlphaBetaDB** relies on the methods of **BoardEvaluator** both directly when analysing a terminal node and through **NextMoves**.

BoardEvaluator: This class has two key methods. The first inspects a GoBoard object (the model) to evaluate whether it is a terminal node in the game tree. The second uses heuristics to provide a statically evaluated value for the utility of the board.

It proved fairly simple to swap out different versions of BoardEvaluator and AlphaBetaXX throughout the project. Building a new GUI would involve more work as it is integrated with the program control flow, although the BoardImage itself would be easy to swap out. (e.g with one with more realistic stones)

More detailed discussion of the development of these classes is presented in Chapter 6.

Chapter 6 Development

6.1 The Game State Model and File Format

The minimum information needed to describe a game state at any time is:

1. The player whose turn it is.
2. The location of all stones on the board.
3. A history of all previous board positions. (But not turn for positional ko)

Additionally, to describe the life and death problem we need to know:

- Which points on the board that are to be considered or excluded from the problem.
- The position of any one stone in the group to live or die. (If one stone is captured in a group, all are captured.)

In order to save and load problems this information also needs to be written to file. To this end a simple text file format was developed where the first 19 lines of the file represented the nineteen rows of a go board. The character 'x' is used to represent black stones, 'o' for white, '-' for an empty space and '*' for a position considered outside of the problem space. This format was chosen so problem could be easily read or edited by a human within a text editor. This was particularly useful for the early development of algorithms such as the legal move checker before a full GUI was implemented. Line 20 in the text file is used to hold the location of the stone defining the group to be captured and to set the player whose turn it is. Line 21 contains a string where notes can be stored about the problem. This line is read in and displayed in the GUI. The game history is not currently saved to file.

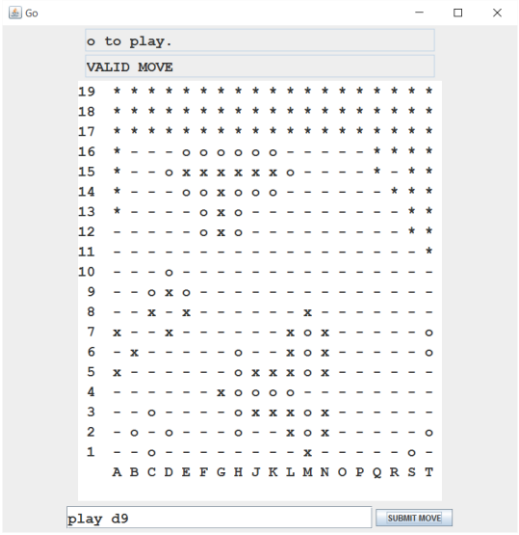


Figure 6.1: Development Interface showing the char array here being used to test the legal move checker. The '*' character shows positions that are out of bounds.

Within the GoBoard class, the information on board position is held in a char array, using the same characters as those used in the text file. Again, this simplified early development as the elements of the array could simply be printed to a text area within the development GUI. The remaining information from the file is held in private instance variables with getter and setter methods. Some other methods operating on the array were included to reduced repeating code in other parts of the program, for example the method .hasAdjacentSpace, returns true is an element in the array is adjacent to an empty space (and thus has a liberty).

To facilitate sorting using java's array sorting methods this class implemented the Comparable interface.

6.2 The User Interface and Controller

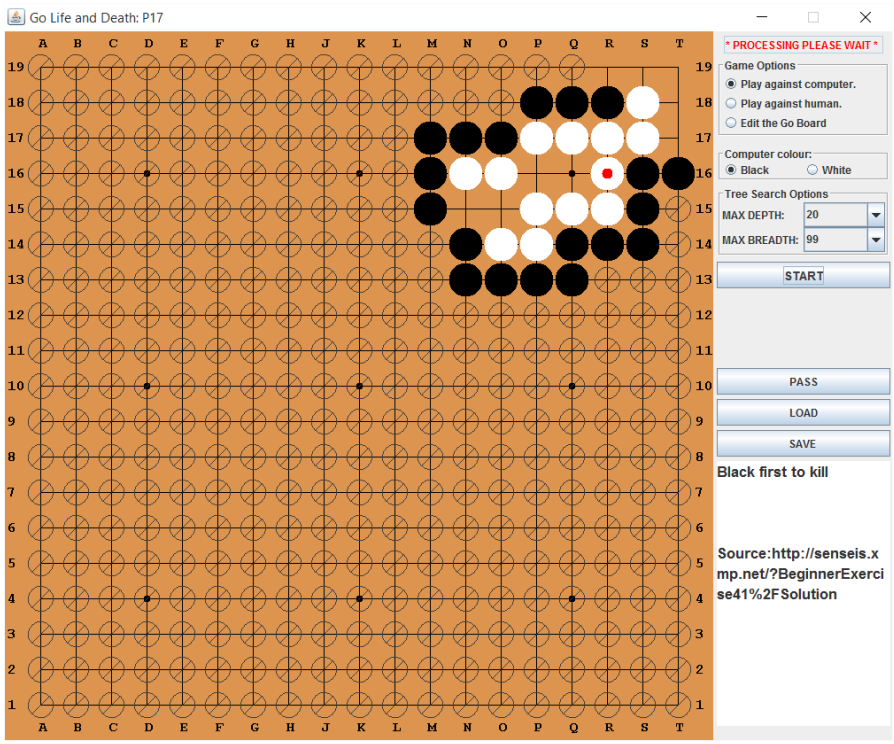


Figure 6.2: The final GUI. Moves are played by clicking on points on the board. Problem notes are displayed at the bottom right.

Figure 6.2 shows the final implementation of the GUI. (Note: the omission of I in the grid reference labels is deliberate and a convention in go to avoid confusion with J.) The text area to the bottom left displays the information in the last line of the text file. It is useful to store instructions and notes on a problem and can be edited by selecting the "Edit the go board" radio button at the top right.

If it is a player's turn, a mouse click anywhere on the image is interpreted as an attempt to place a stone at the nearest point to that click. The intersections are forty pixels apart. It is tricky to click exactly on a point so the coordinates returned from the mouse listener are rounded to the nearest 40. This seemed to give satisfactory behaviour with users during evaluation.

```
int x = (event.getX()+20) - (event.getX()+20)%40;
```

If a user attempts to play in an illegal position an option pane will pop up informing him of this.

When a stone is played a call is made to a compute method to process the computer's response. So that the GUI can be updated with the stone just played and a "processing please wait" message displayed the processing is placed on a SwingWorker thread. The call to AlphaBetaDB is placed on the SwingWorker thread. This method is also invoked when the computer has first move and the start button is pressed. The used can select which colour stone the computer will play.

When the "Play against human" radio button is selected, calls to compute a move are not made following a move and a game can be played out manually with the MoveChecker checking moves are legal and making captures as appropriate. Once the stone defining the life/death group has been captured a message pane will appear saying so.

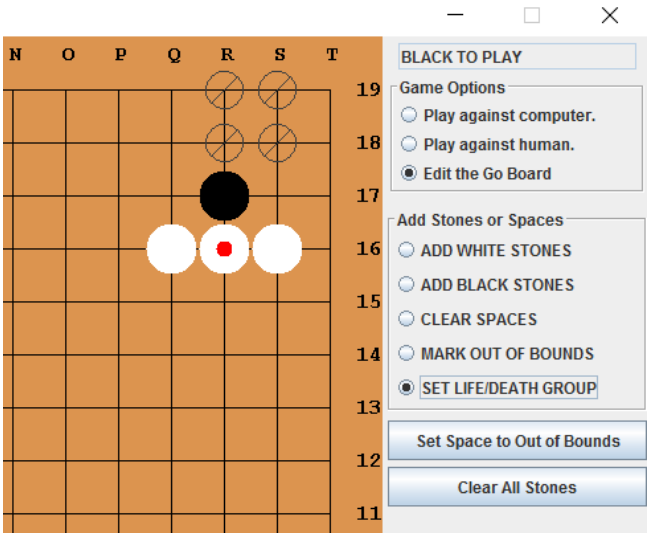


Figure 6.3: The Edit Options

With the "Edit the Go Board" radio button selected a problem can be modified or a new one created. To create a new problem pressing "Clear All Stones" would create an empty board. To add stones, the "add white stones" or "add black stones" button is selected once, followed by clicks wherever a stone should be. To set one point to out of bounds (shown with the stop sign like marker at S19 above) "mark out of bounds is selected". Typically though, most points on the board are out of bounds so it is better to use the "set space to out of bounds" button which will set all free space to out of bounds and then use "clear spaces" to add in the places you want to be considered as part of the problem space.

Finally, the stone defining the group pertinent to the life and death problem is marked with a red dot using the "set life/death" option.

6.3 The Legal Move Checker

The legal move checker takes coordinates of a potential move and a reference to a GoBoard as arguments. If the move is legal it modifies the GoBoard appropriately and returns the boolean, true. If the move is illegal it returns false.

An outline of the algorithm used is set out below (Steps 1 to 5 are numbered similarly in the code should the reader wish to cross reference):

1. If the point is occupied or marked out of bounds:
   1.1. Return false and terminate.
2. Add a new stone at this point.
3. For each direction, North, South, East, and West:
   3.1. If there is an adjacent enemy stone here:
      3.1.1. If the group containing this stone has no liberties:
         3.1.1.1. Remove this group from the board.
4. If the group containing the new stone has liberties:
   4.1. If the board position is in the history of board positions: //(ko!)
      4.1.1. Set the point at which the new stone was placed back to empty space.
      4.1.2. Return any captured stones to the board.
      4.1.3. Return false and terminate.
   4.2. Else:
      4.2.1. Switch the player's turn variable in the model
      4.2.2. Set the number of passes variable to zero in the model
      4.2.3. Add a copy of the board positions to the history
      4.2.4. Return true and terminate.
5. Else:
   5.1. Set the point at which the new stone was placed back to empty space.
   5.2. Return false and terminate.

Note:

It is necessary to reset the variable at 4.2.2 as two passes are used as a condition to stop in the alpha-beta. A pass is always a valid move and in this case, the player's turn is changed and the pass variable incremented by one. There is no need to add the board positon to history in this case.

A helper method, .capture(int, int), that take the coordinates of an enemy stone to the immediate North, South, East, and West and returns an ArrayList of coordinates of captured stones in this direction was used for step 3. If there are no stones to be captured in this direction it sets a boolean flag to false. It works by changing the char at its current location to an 'e' to indicate it is an enemy stone that has been examined. It then checks whether it has adjacent free spaces, if it does it terminates as there is no group to capture here and it set the capture flag to false. If it has no adjacent free space but has more stones of its colour as neighbours it calls itself on each of these recursively. Eventually a liberty is found or the group is found to have no liberties in which case the stones at the coordinates

stored in the ArrayList are set to empty space. If there is no capture the stones marked 'e' are returned to enemy stones. A similar method is used at step 4 to explore the group containing the new stone.

The move checker appears to work perfectly with no bugs being found during testing. However, as it is called upon at every free board point and in every game state in the search tree it might be an area worth investigating for efficiency improvements in future versions.

6.4 Implementation of MiniMax with Alpha-Beta Pruning

```
public int alphaBeta(GoBoard node, int alpha,int beta,boolean maximisingPlayer, int depth,int breadth){
        int bestScore;
        if(BoardEvaluator.stop(node)||(depth<=0)){
            bestScore = BoardEvaluator.evaluate(node,player);
        }
        else if(maximisingPlayer){
            ArrayList<GoBoard> children = NextMoves.getBestMoves(node,breadth);
            bestScore = alpha;
            for (GoBoard child : children){
                int childScore = alphaBeta(child,bestScore,beta,false,depth-1,breadth);
                bestScore = Math.max(bestScore, childScore);
                if (beta <= bestScore){
                    break;
                }
            }
        }
        else{
            ArrayList<GoBoard> children = NextMoves.getBestMoves(node,breadth);
            bestScore = beta;
            for (GoBoard child : children){
                int childScore = alphaBeta(child,alpha,bestScore,true,depth-1,breadth);
                bestScore = Math.min(bestScore, childScore);
                if (bestScore <= alpha) {
                    break;
                }
            }
        }
        return bestScore;
    }
```

Figure 6.4: Java Code from the AlphaBetaDB class

The version of the alpha-beta algorithm implemented in this project was adapted from one found at the Algorithm Wiki page at Will Thimbly.net (Minimax Search with Alpha-Beta Pruning [Algorithm Wiki], no date). The algorithm was adapted to include depth and breadth arguments for a limited search tree and to make use of the methods in the BoardEvaluator and NextMoves classes.

6.5 The Static Board Evaluator

6.5.1 The need for a static board evaluator

The function of the board evaluator is to inspect board positions and return values representative of their likely strength statically, i.e. without exploring possible further moves. This is achieved through scanning the board for patterns of stones known to be generally advantageous or disadvantageous.

This function is implemented in the method, *.evaluate(..)* in the class, BoardEvaluator. It is useful in two situations.

Firstly, if a game-tree is prohibitively large it can be truncated through the use of a depth limit variable. Should the depth limit be reached before a terminal node the static value of the node is returned instead of a value for a definite win, lose or draw. The success of the algorithm is of course no longer guaranteed and depends very much on the quality of the board evaluator.

Secondly, it allows alpha-beta pruning to be optimised by providing one means to rank different positions. As discussed, at full depth the mini-max algorithm will always yield a winning move if one exists. The addition of alpha-beta pruning makes this process less expensive by ceasing to explore branches as soon as it can be ascertained that they can only return values worse(lower for a max node or greater for a min node) than the best value so far discovered. If the winning position happens to be on the last path explored, there is no pruning and so no speed advantage over mini-max. Whereas, in the best case, the strongest moves are explored first and there is a big speed advantage. It is therefore useful to sort potential board positions so that those more likely to be strong moves are explored first. Furthermore a breadth limit can be imposed such that the lowest ranked moves are not included at all, but again in doing so we can no longer guarantee an optimal result.

In this program the .evaluate(..) method is called from the following two locations:

1. A depth limit can be set from within the GUI and, if reached, a call is made to the board evaluator from the *AlphaBetaDB.alphaBeta(..)* method.
2. Given a game state, the method *NextMoves.getBestMoves(GoBoard b,int maxNumberOfMoves)* returns a list of game positions resulting from each legal move that could be played. Each is assigned a value by calling board evaluator and the list sorted by these values. The parameter *maxNumberofMoves* is used to set the size of a subset of best moves. The *getBestMoves(..)* method itself is called from within the alpha-beta algorithm at each node except terminal nodes.

6.5.2 Go Heuristics

A static evaluation of the board relies on capturing and computerising the use of heuristics that Go players have learnt from experience and goes beyond a simple application of the rules. The simplest of these are recurring patterns of stones that are remembered and recognised by players as likely to lead to either favourable or unfavourable outcomes.

Many of these heuristics are well known and documented. Patterns of stones favourable to a player are often referred to as "Good Shape" and those that are unfavourable as "Bad Shape". Discussions of these shapes and examples can be found at several websites run by enthusiast Go Players. (Shape Collection at Sensei's Library, no date) (What is good shape? | British Go Association, no date)

One basic move known to be favourable for an attacking player is known in the West by the Japanese term **hane**, meaning "quick turn".(Kim, Soo-hyn, and Lee, 2010)

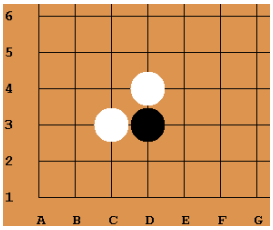


Figure 6.1: Hane

Figure 5.1 illustrates the hane. White plays at C3, Black plays at D3 making contact with White and White responds by playing the hane at D4.

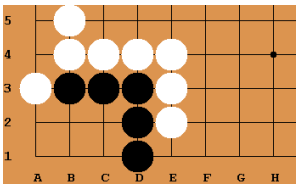


Figure 6.2 Hane to attack a white group

Figure 5.2 illustrates the relevance of hane to life and death problems. Here, White has played the hane at A3 preventing Black from forming two eyes in corner and thus life. There is a proverb, "There is death in the hane". (There is Death in the Hane at Sensei's Library, no date)

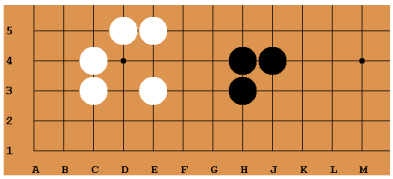


Figure 6.3 The Mouth(White) and The Empty Triangle(Black)

Figure 5.3 illustrates examples of good and bad shape. The Mouth represents good shape, as an eye can be formed here. The Empty Triangle is considered bad shape as it represents a wasted move, here played at H4. H4 connects the other two black stones solidly, however, this could also be achieved by playing at J3 so the connection is not yet under threat, and perhaps the turn could have been spent

more fruitfully elsewhere. (Further discussion is available at Sensei's Library (no date) or Shotwell et al (2011).)

6.5.3 Implementing the board evaluator

In the first iteration of the program a basic board evaluator was created that rated boards, from the perspective of a player who was either black or white according to this algorithm:

Variables: SCORE and PLAYER (either black or white).

"Life Group" is the set of stones under attack.

1. If there is an empty space at the location set for the life group(a capture):
   1.1. If life group is same colour as PLAYER:
      1.1.1. Return -1000 and terminate.
   1.2. Else:
      1.2.1. Return 1000 and terminate.
2. If the last move was pass:
   2.1. Return 0 and terminate.
3. If the last move was a **hane:**
   3.1. Set SCORE = 50.
4. Else if the last stone **removes a liberty** from an enemy:
   4.1. Set SCORE = 30.
5. If last stone played was by PLAYER:
   5.1. Return SCORE and terminate.
6. Else:
   6.1. Return SCORE*-1

Note that the same method was used to score both terminal nodes and to score games for the selection of moves to be considered. In both cases only the last move made is considered.

The behaviour of the board evaluator can be investigated within the GUI by setting the depth to zero causing the program to simply play the move (or one of the moves) with the highest statically evaluated value. In doing so some undesirable behaviour became apparent.

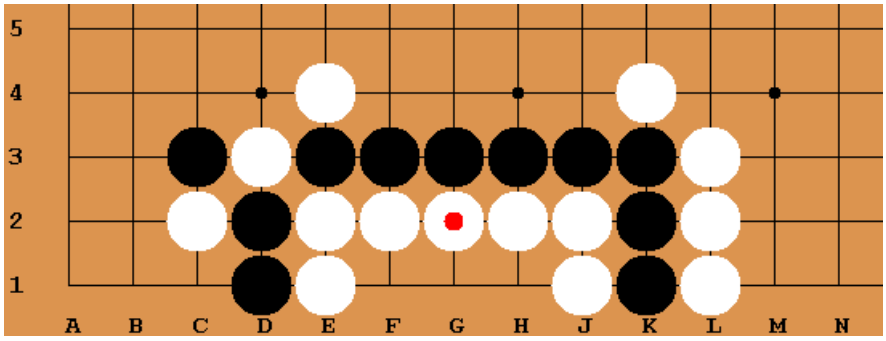


Figure 6.4

In the situation shown if Figure 6.4, white needs to defend the group by playing at G1 and forming two eyes within its territory. Of course this algorithm does not know this and the white stones outside the black lines are the result of successive white moves with the depth limit set to one. Playing adjacent to black stones is favoured as it scores 30 points for removing a liberty and 50 at the corners of the group for the hane. If we limited breadth in this situation, the best moves on white's turns are not going to be selected as playing within its own group it can only score zero. The situation for the attacking player, however, is different. Playing within white's territory and reducing its eye space is rewarded. It can be seen that, in life and death problems, a heuristic generally helpful in considering the placement of an attacking player's stones might be unhelpful when considering the defending player's stones.

There is another major flaw in using this approach. In selecting a subset of legal moves to score using alpha-beta only nodes sharing a single parent are considered. Whereas, when scoring terminal nodes in a depth-limited game tree many different paths are involved. In this case particularly, it doesn't make sense to rank the strength of positions based solely on the final move as previous moves were also different.

For these reasons the following points were kept in mind when building the next iteration of the program:

- The optimum evaluation for selecting a set of moves might differ from that for scoring terminal nodes.
- The static board evaluator should take into account all stones on the board, not just those in the vicinity of the last stone played.
- Heuristics indicative of strong moves for an attacking player may not hold for defending players and vice versa.

A new board evaluator was implemented which scans the whole board for contact on enemy stones, hane, empty triangles and mouths. It was reasoned that attacking stones making contact or hane on defending stones should score positively for the attacking player and negatively for the defending player. The defending player's hanes are not counted. Mouths and empty triangles are counted positively and negatively respectively irrespective of whether the player is attacking or defending.

The points awarded in this evaluation are as follows:

From perspective of Player.

- Attacker contact on defending stone: +30 for each if Player is attacker (-30 if defender)
- Hane around defending stone: + 50 for each if Player is attacker (-50 for each if defender)
- Empty Triangle: -75 for each of Player's and +75 for each of opponents.
- Mouth: +100 for each of Player's
- Life group is captured: – +1000000 if Player is attacker or -1000000 if defender.

In this program the capture condition is assessed first followed by a method, *heuristicValue(GoBoard, char)*, which is called to assess the static value of the board if needed. The algorithm used in *heuristicValue(..)* is outlined below:

From the perspective of Player:

SCORE=0

1. For each point on the board:
   1.1. If this point is occupied by a stone the same colour as the life group:
      1.1.1. Increment SCORE by 30 for each (0-3) attacking stone to N, E, S or W.
      1.1.2. Increment SCORE by 50 for each (0-2) hane pattern around stone.
2. If Player is same colour as life group:
   2.1. SCORE = SCORE * -1
3. SCORE = SCORE + (Number of Player's empty triangles on the board * -75)
4. SCORE = SCORE + (Number of opponent's empty triangles on the board * 75)
5. SCORE = SCORE + (Number of Player's mouths on the board * 100)
6. SCORE = SCORE + (Number of opponent's mouths on the board * -100)
7. Return SCORE

It might be worth noting that, although some asymmetry appears to have been introduced through distinguishing between attackers and defenders, the calculated utility for black will still always be the exact negative of that for white. A quick check of this was made in testing, as if it turned out not to be the case, it would be obvious there was a problem here.

Separate methods *countMouths(..)* and *countEmptyTriangles(..)* were written and tested independently. Both methods take an array representing the board and a char representing the stone colour as arguments and return the number of stones of that colour.

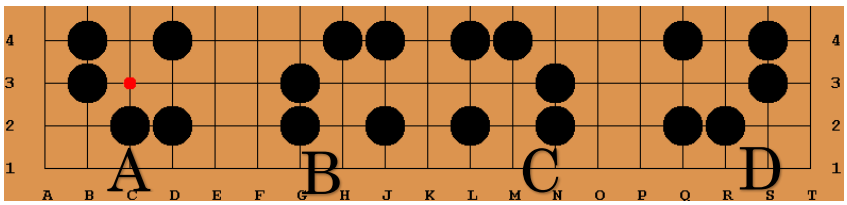


Figure 6.5: Rotations of the mouth about centre (marked with red dot).

Figure 6.5 shows the four rotations of the empty mouth shape that are searched for in the *countMouths(..)* method. The search takes the centre of the mouth as a starting positon, therefore iterating through all points on the board except for those along the edges. If searching for Player's mouths and the point is not occupied by Player then a search is made of the surrounding stones. Nested if statements are used with the goal of ruling out the possibility of a mouth as soon as possible without examining all eight surrounding points. For example if North and South are both occupied by Player's stones, or if neither are, there cannot be a mouth here and the other six locations are not examined. Nevertheless, repeatedly scanning the board for different patterns is clearly an expensive process. The gains to be made in a reduced branching factor if we can successfully cut down the breadth far outweigh this cost though.

After some evaluation of this version of the board evaluator it was found that some obvious optimal moves were being excluded from the bestMoves list. This occurred when a strong move for the defending player involved forming the empty triangle shape. For this reason a final version of the board evaluator was written that only considers empty triangles for the attacking side. Mouth shapes from both sides were still considered. This is discussed further in Chapter 7.

6.6 Next Moves Selector

A short class, *NextMoves.java*, was written to inspect a game state (GoBoard object) and return a list of possible moves ordered on their statically evaluated values. To obtain the initial list of moves it calls the legal move checker with an attempted play at every point on the board. If the legal move checker returns **true**, a copy of this game state is added to an ArrayList called allMoves. A pass is also added as a possible move. The program then iterates through allMoves calling the *BoardEvaluator.evaluate(..)* method on each GoBoard object to obtain a value by which to sort them. This value is then written to the variable, *currentScore*, within the GoBoard class.

The variable, currentScore, was added to the GoBoard class along with an implementation of the Comparable interface in order to establish a natural ordering and utilise the native ArrayList.sort(Comparable) method in Java. Once sorted by this method allMoves is truncated to an ArrayList, bestMoves, with a size limit passed as a parameter. In this program, the size limit is set to the breadth limit set in the GUI.

The NextMoves.getBestMoves(GoBoard,char) is called from each non-terminal node in the alpha-beta algorithm.

Chapter 7 Testing and Evaluation

7.1 Module Testing

7.1.1 Testing the alpha-beta class

Even for simple problems game trees in go contain many thousands of nodes and are impractical to analyse manually. For this reason a simple generic game tree was constructed to test the alphaBeta.java class. This is shown in Figure 7.1 below.

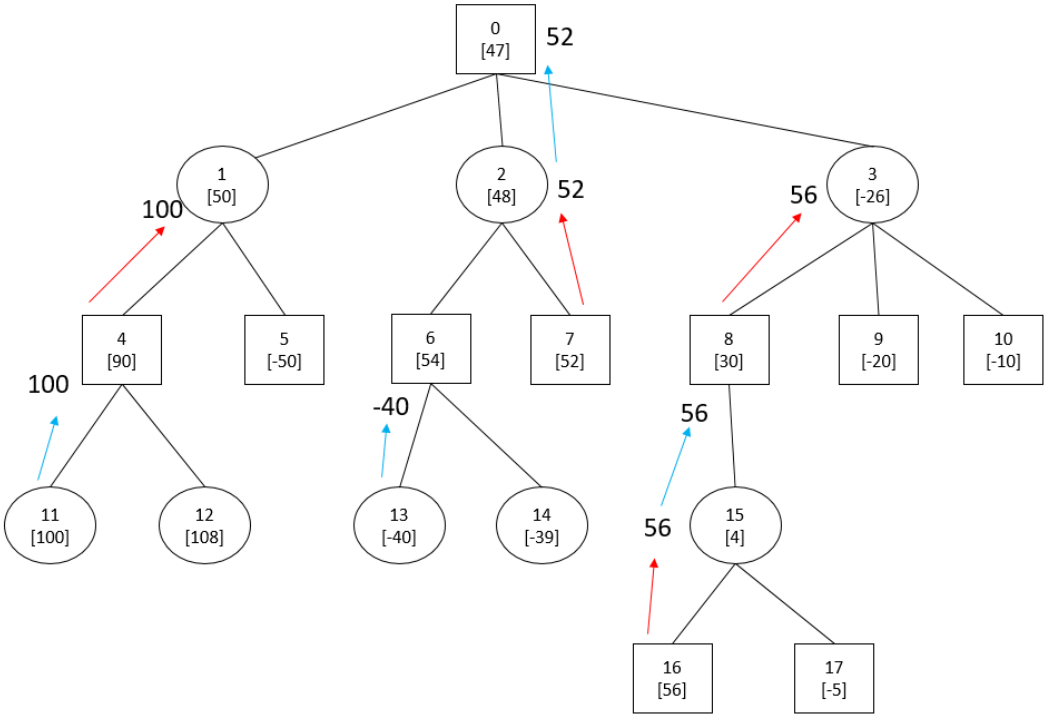


Figure 7.1: A game tree constructed for testing the alpha-beta algorithm class.

The numbers in square brackets represent arbitrarily chosen static values for board positions one to seventeen from the perspective of the player at the root node (rectangles). The numbers in bold represent the minimax values that should propagate up the tree. With no depth limit alpha-beta should return a value of 52 for the root node.

Several test classes were written in order to test the alpha-beta class. The class GameTree.java reads in a text file and constructs an ArrayList of dummy GoBoard objects with instance variables containing their static value, turn and a list of their children. A test NextMoves class was constructed that reads the list of children,

extracts these GoBoard objects from the GameTree ArrayList and adds them to a new ArrayList which it returns to alphaBeta.

The format used to represent information in the text file was:

<node reference>,< static score>,< player>,<children>.

This allowed for various game trees to be quickly created in a text editor for testing. The first three lines representing the tree in Figure 7.1 are shown below:

```
0,47,o,1,2,3
1,50,x,4,5
2,48,x,6,7
```

Print statements were used to output the alphaBeta value with different depth limits. The results for this tree were:

```
Depth= 0 Root Value: 47
Depth= 1 Root Value: -26
Depth= 2 Root Value: 30
Depth= 3 Root Value: 4
Depth= 4 Root Value: 52
Depth= 5 Root Value: 52
Depth= 6 Root Value: 52
```

It can be seen by inspecting the game tree that these results are correct. Figure 7.2 shows the effect of limiting depth.

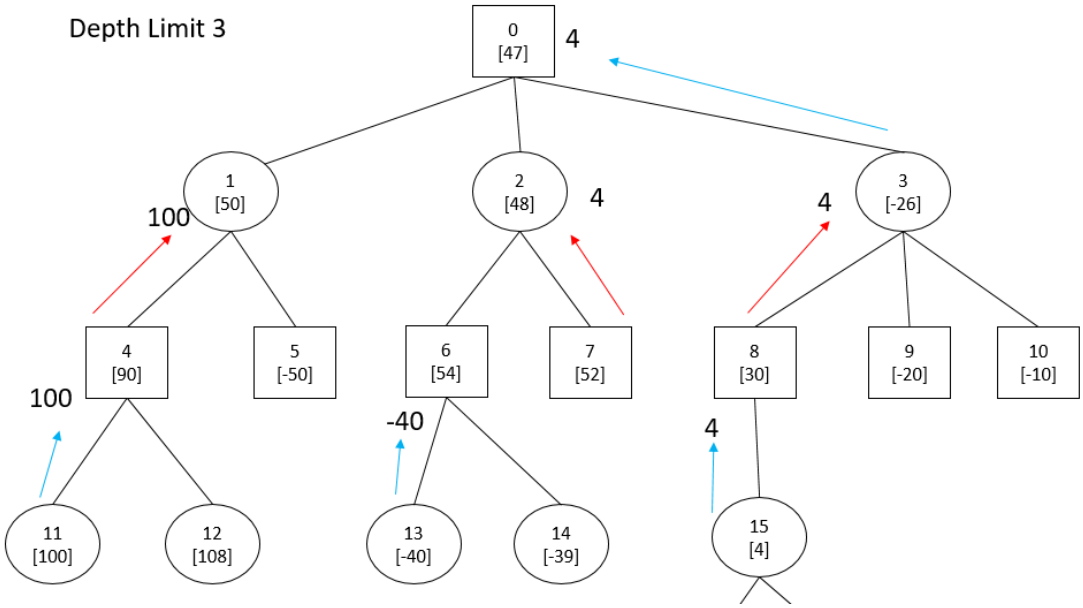


Figure 7.4 Alpha-Beta scores with depth limited to three.

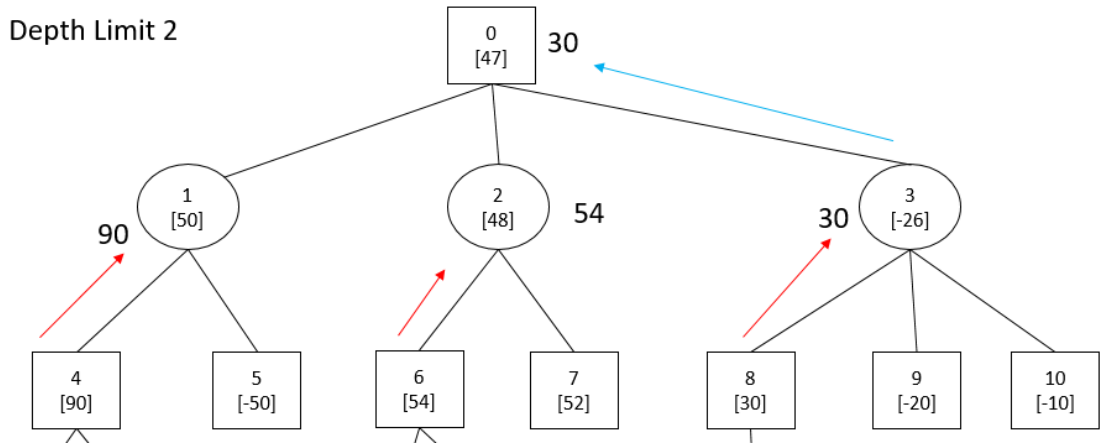


Figure 7.3: Alpha-Beta scores with depth limited to two.

7.1.2 Testing the legal move checker.

The program was tested for the following expected behaviours:

1. Does not allow play and displays an invalid move warning message when an attempt is made to play at:
    a. An occupied position on the board.
    b. A position with no liberties (after any captures).
    c. A position set as out of bounds.
    d. A position that would otherwise result, through capturing opponent stone(s), in a board that had previously occurred in the game. (Positional superko rule)
2. Capture groups of stones to the North, South, East and West according to the rules of go.
3. Captures multiples groups with a single stone according to the rules of go.
4. Captures groups along the edges of the board according to the rules of go.
5. Behaviour with groups adjacent to positions marked as out of bounds is the same as with empty spaces.
6. Capture groups in corners of board according to the rules of go.

These behaviours were all tested through playing at various points on the board shown in Figure 7.3. No undesirable behaviour was discovered.

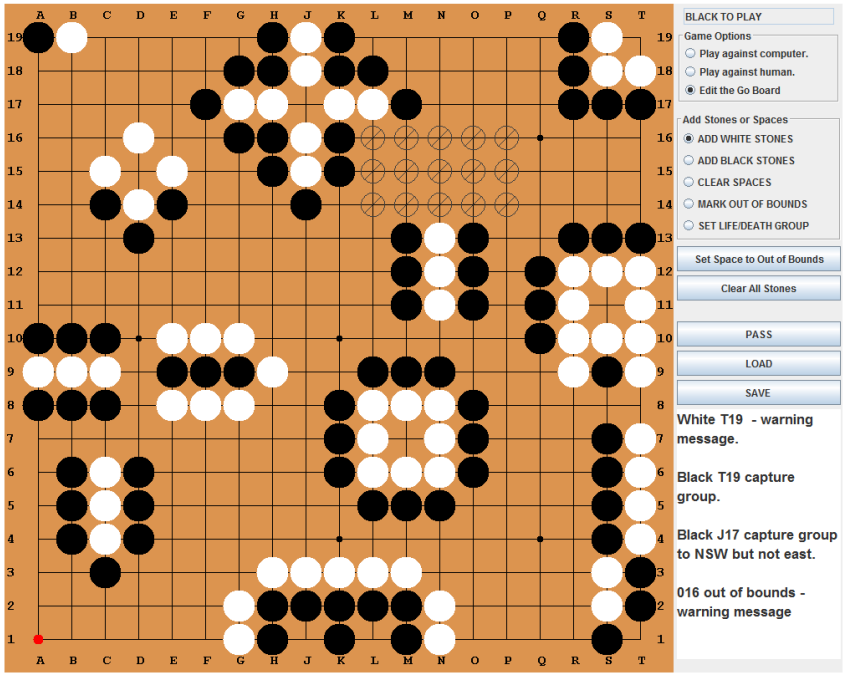


Figure 7.3: A board position created for testing the legal move player with examples of expected behaviour in the problem comments section.

7.1.3 Testing the Board Evaluator

The methods in the heuristics code could often be tested individually as they were developed with test code such as the lines below inserted into the GUI class:

mouseClicker(MouseEvent event) {

System.out.println("Number of Mouths: "+ BoardEvaluator.countMouths(.

These lines call the method countMouths whenever the mouse is clicked so, with a program running in edit mode the method can be tested as mouths were added to the board. This was repeated for the method countEmpty triangles and no bugs were found.

A line was also placed in the code to output the statically evaluated value of different boards:

System.out.println(BoardEvaluator.evaluate(theBoard,'x') +" ");

This was compared with manually calculated values using the heuristic and again, no bugs were discovered.

7.2 Performance

The two sections of code involving significant computation are the legal move checker and the static board evaluator. The legal move checker is called on all 361 points on every node in the search tree, in order to find all child nodes. The board evaluation function, involving repeated heuristic scans, is then run on each child. The total time to process will be proportional to the number of nodes examined.

Improvements to these two processes can only have a linear effect on processing time per move. The variables really limiting the size of problem that can be tackled are the number of branches at each move (the breadth) and the depth it is necessary to search to. If it were possible to determine an average depth, d and breadth b, the total number of nodes in the game tree could be found from:

$$\sum b^0 + b^1 + b^2 + b^3 + \cdots + b^d$$

The time complexity then is: $O(b^d)$

There is no obvious way to determine the average breadth or depth of a game tree for any particular go problem. It seems a reasonable assumption, though, that both breadth and depth will tend to rise with the number of legal moves available at the start of the problem (though not necessarily linearly). Growth in processing time for an exhaustive search then is likely to increase at least exponentially as the number of initial moves is increased.

Three methods were introduced to reduce the size of the search tree. The first was branch pruning using the alpha-beta algorithm, the second the use of a breadth limit at each node and lastly depth limiting. Alpha-beta will only prune unnecessary branches but in limiting breadth and depth, of course, we run the risk of excluding the most fruitful branches too.

As improvements were introduced to the program its performance was assessed by running very simple problems and incrementally increasing the number of points at which a stone can be placed on the first move. One problem used for this purpose is shown in Figure 7.XX below.

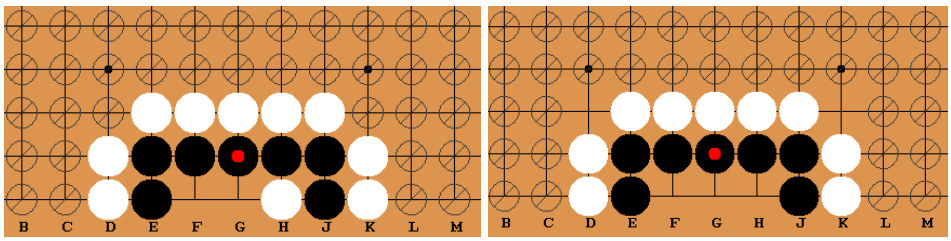


Figure 7.4: Varying free space within a problem. White can play at two points in the board on the left or five in the right. (G1 to kill black)

Code was added to the SwingWorker thread to obtain a processing time from the system.

```
long endTime = System.currentTimeMillis();
        System.out.print("Compute     time:     "+     (endTime     -
startTime)/1000.000+ " seconds");
```

The data obtained for the problem above are shown in Table 7.1. The alpha-beta algorithm was used without depth or breadth limits set. The moves were sorted using the latest version of board evaluator.

Number of legal moves (excluding pass)	processing time/s	
	minimax	alpha-beta
1	0.006	0.016
2	0.027	0.031
3	0.035	0.063
4	1.167	0.219
5	11.101	0.656
6	149.044	2.172
7	2454.678	9.157
8		36.358
9		167.55
10		1079.077
11		6069.108

Table 7.1: Processing times for an exhaustive search with minimax and the alpha-beta algorithm.

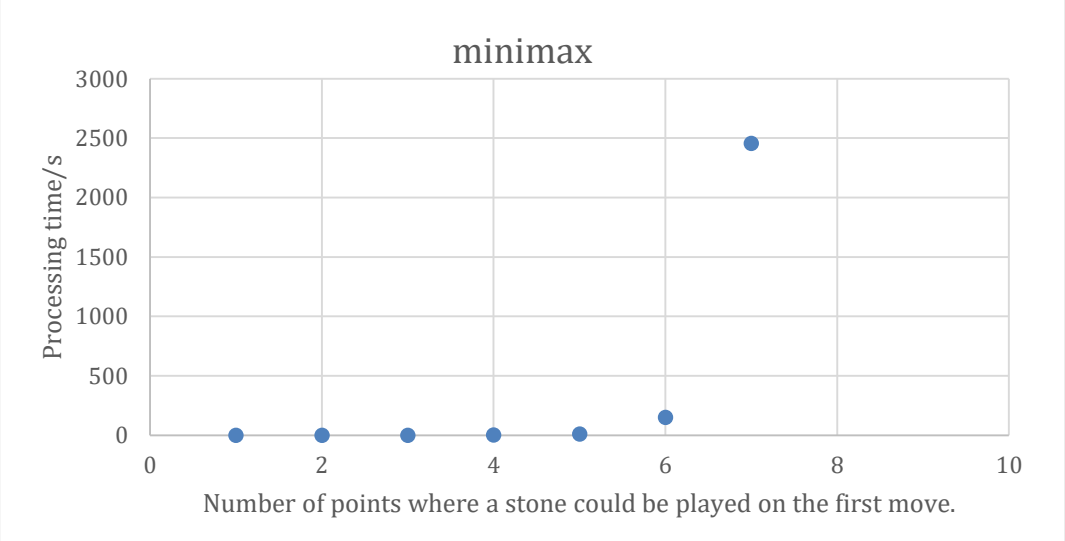


Figure 7.5: Graph showing the performance of the alpha-beta algorithm as the number of options for first move were increased.

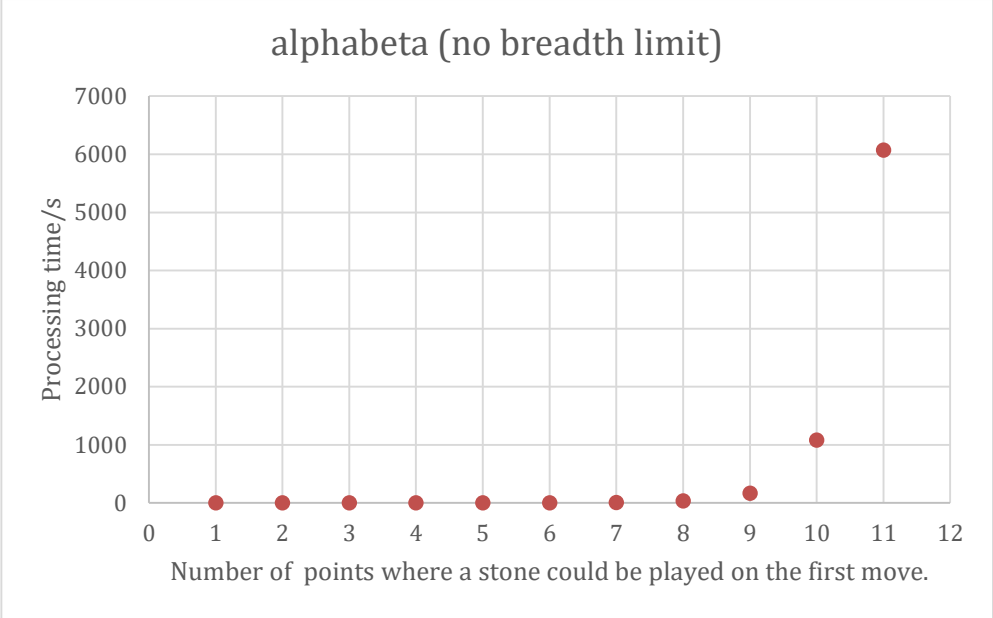


Figure 7.6: Graph showing the performance of the alpha-beta algorithm as the number of options for first move were increased.

The effect of additional space on processing time is most obvious when plotted on a graph. Moves were processed fairly rapidly with either algorithms up to a limit after which processing time grew very rapidly with each additional move and soon becoming impractical. An exhaustive brute force search with minimax completed in a reasonable time up to about six moves. Seven moves took around forty minutes. Eight possible moves was not investigated but from the trend it can be estimated to be at least several hours with further moves almost certainly being measured in days. The pattern with alpha-beta was similar except the sudden growth occurs later at around 10 moves. This suggests alpha-beta may be able to tackle problems involving around two more free points than would be practical without pruning whilst still being guaranteed to find the optimal move. This is a significant advantage allowing many more interesting problems to be attempted.

	processing time to play (white at g1)/s					
Number of possible moves	minimax exhaustive search	Alpha-beta b = 99 d=99	Alpha-beta b = 8 d=99	Alpha-beta d=5 b =8	Alpha-beta d=6 b=8	Alpha-beta d=7 b=8
1	0.006	0.016	0			
2	0.027	0.031	0.031			
3	0.035	0.063	0.076			
4	1.167	0.219	0.25			
5	11.101	0.656	0.562			
6	149.044	2.172	2.14			
7	2454.678	9.157	8.658			
8		36.358	26.028	6.89		
9		167.55	94.599	9.743		
10		1079.077	480.215	14.068	20.778	
11		6069.108	3844.457	FAIL	42.738	
12				FAIL	52.834	
13				FAIL	56.934	
14					59.021	
15					61.046	
16					62.215	178.895
17					68.825	
18					78.236	
19					83.906	
20					93.491	
25					125.605	
27					131.994	
28					FAIL	
30					FAIL	

Table 7.2: Results from tests of processing times with limited breadth and depth. "FAIL" indicates the move chosen by the program was not optimal. "d" denotes breadth and "b" breadth.

It became clear that to solve problems with more than around nine possible first moves in a reasonable time it would be necessary to limit breadth and/or depth. Table 7.2 shows the results of experimenting again with the problem illustrated in 7.xx. Firstly the breadth only was limited hoping that moves in the path for optimal play were included in the ranked list scored with the board evaluator.

A breadth of eight was chosen as it was found that at breadths lower than this the program failed to play at the winning position. A pass is included in the list of all possible moves generated in the program so setting the breadth to eight starts having an effect, cutting off the lowest scoring move, once there are eight free points to play at on the board. The results are shown in the fourth column of the table. As expected there is no effect until row eight after which there is an increasing, but not hugely significant, reduction in processing time as the moves list is truncated. Again, the problem of massively increasing processing times after around nine moves is present. This prevents attempting problems beyond eleven free points.

To move beyond this limit of eleven possible first moves the search tree was further reduced by limiting the depth. Reducing the depth to five gave a significant speed reduction for smaller problems but failed to yield the correct move at eleven free points. With a depth of six the program processed the correct move up to 27 free moves in reasonable time, after which again an incorrect move was chosen. As the

size of the search tree rises exponentially with depth very significant reductions in processing time can be made by limiting it.

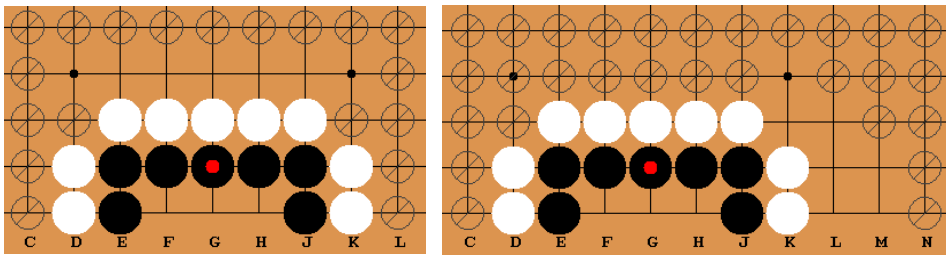


Figure 7.7: Choice of boundary can significantly affect the complexity of the game tree.

In the tests described some effort was taken to keep the problem as simple as possible. For example, C1 and L1 were kept out of bounds in all cases. Fig 7.7 shows the same groups of stones with different sets of ten free points. The group on the right will have a larger game tree than the one on the left as K1 and K2 are at risk of being captured opening up the right hand side for further combinations. Figure 7.xx shows one of numerous messy combinations on the full game tree. In the game on the left the boundaries ensure that only the black group or stones played in its territory may be captured. With a breadth limit of eight the problem on the left was solved in eight minutes. The problem on the right remained unsolved after an hour.

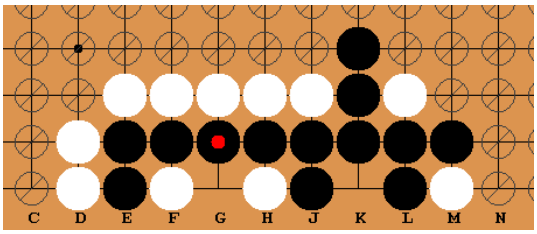


Figure 7.8

The number of initial legal moves is clearly not a good indicator of the expense of solving a particular problem. It is not difficult to imagine problems of various complexities even if they involve the same number of stones and options for first move. However, as they derive from a deliberately simple problem, the figures in the table might be useful as an indicator of the very best performance likely to be achieved. For example, if we have a problem with more than ten free points it will almost certainly be necessary to limit the depth and breadth. With depth and breadth limits greater than six and eight any problem with more than twelve options will probably take significantly longer than a minute. Of course, with the use of size limits and better heuristics, or heuristics particularly attuned to a specific type of problem we might do better than this.

7.3 Evaluating behaviour in real Life and Death Problems

It seemed plausible that the static board evaluator would be effective in selecting a few moves likely to containing an optimal move. It seemed less likely that it could accurately predict the utility of a node terminated at a depth limit. For this reason, and also to avoid changing two variables at a time, the problems were initially investigated by adjusting the breadth only to find the minimum required to yield the optimal move.

Boundaries were set to restrict play as much as possible to moves relevant to the problem. This reduces the search space and enables problems to be solved in the minimum time possible. However, care had to be taken as any group in contact with a boundary is in effect un-killable as it had a liberty that cannot be played on. Survival by bridging the gap to the boundary was avoided by blocking with an enemy stone or pushing the boundary back.

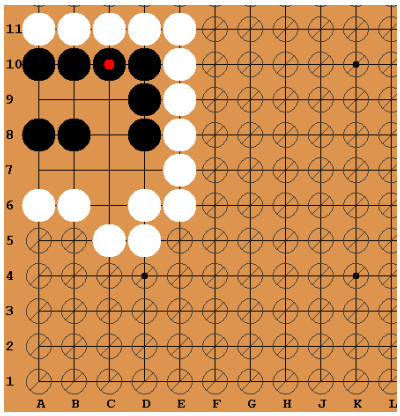


Figure 7.9 Where must black play to live (and gain the most points)?

During the user evaluation an experienced player from Glasgow Go Club considered the problem above and paused for thought before suggesting C7 as the optimal move. He was correct. A play here leads to a shape known as a bent four and three points of territory can be gained. The program however played at B9, forming an eye. This move is less good as, although white cannot prevent the formation of a second eye, it can reduce black's territory by playing at C8. Of course, from the program's point of view the moves are equally good as it does not consider territory. (A program that considered territory gain would be nice to have.)

Experimenting with this problem led to further insight. The user evaluation was conducted before the addition of the empty triangle heuristic. Once added, it was noticed that the breadth required for the solution increased from 5 to 7. This led to an increase in processing time from 30 seconds to nearly two minutes. The behaviour can be explained as the correct first move at B9 will create two empty triangles (-150) so will no longer appear in the bestMoves list. In fact with a

breadth of 7, the move is still not included but the even better move at C7 is and the computer plays here.

With this in mind tests were carried out on different problems both with and without the empty triangle heuristic included. (A copy of the program was used with the calls to the count triangles methods commented out). It was found that optimal moves for the defending side were excluded in several cases. To solve these problems it was necessary to use a larger breadth when including the heuristic that when not using it.

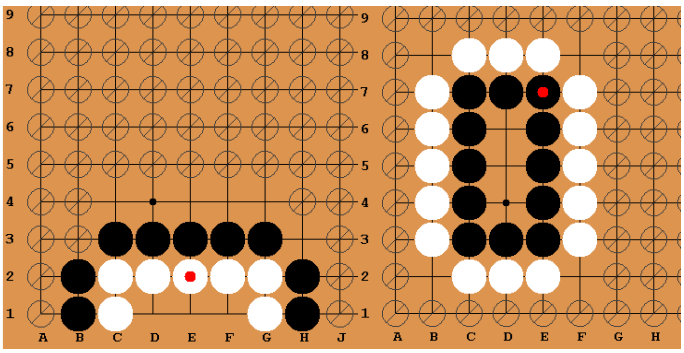


Figure 7.10: Two situations where playing to live creates the empty triangle shape.

Figure 7.10 shows two more situations where, if we are playing to live, we must not exclude positions where empty triangles are formed. These include the optimal moves of white at E1 on the left or black at D5 on the right. The game on the right is solved with a breadth of three using just hane and contact heuristics. When the empty triangle heuristic was included a breadth of eight was necessary to ensure this move was considered. (There are only eight moves including pass here so the empty triangle scoring caused the optimal move to be the lowest ranked.) Although there may be other board positions where the heuristic performs well, it was decided situations like those above are common enough to warrant changing the algorithm.

It might be possible to modify the heuristic to explore surrounding stones and decide when and when not to apply it. However, for simplicity and due to the time constraints of the project (and rather than abandoning the empty triangle shape altogether) it was decided to try searching only for empty triangles belonging to the attacking side (the side that is not in the “life group”). This was the final build of the program.

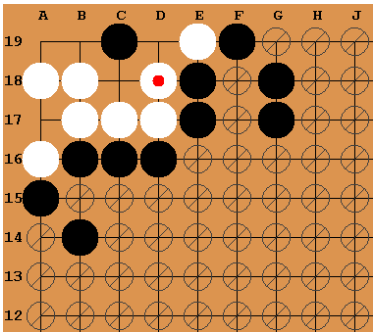


Figure 7.11: An example of a problem where the empty triangle heuristic gave an advantage. Black has played at C19 to kill white.

The problem shown above was solved with a breadth limit of four when empty triangles for both players are scored. This is increased to five when triangles were not included at all. Including only the attacking player's triangle also offered no advantage over not including the heuristic at all. The advantage to scoring defending (white) triangles in this problem can be explained as follows: Plays at C18 or C17 would fill empty white triangles, increase white's score and decrease blacks. C17 and C18 are then pushed down the ranks and are cut off by the breadth limit rather than the winning move at C19. C19 itself is not in contact with enemy stones nor is it part of any shape, so is in a neutral positon with regards to the heuristics.

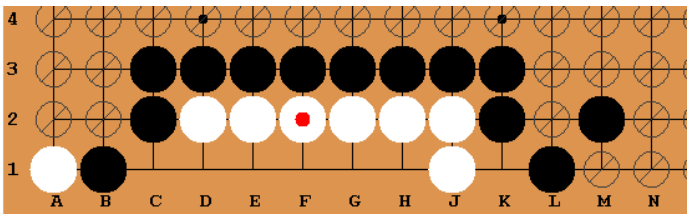


Figure 7.12: Six stones will die on the second line. White has played at J1. Where should black play to kill the group?

There is a Go proverb: six stones on the second row will live, but eight will die. (Six Die But Eight Live at Sensei's Library, no date) In the critical case of seven stones, the outcome depends on who plays first.

The problem illustrated above contains such a row. There are six stones here, so according to the proverb with correct play they will be killed despite white taking the first move at J1. With a breadth of 4 the computer played at first at F1 in 17 seconds and answered any subsequent plays by reducing eye space and eventually capturing the group.

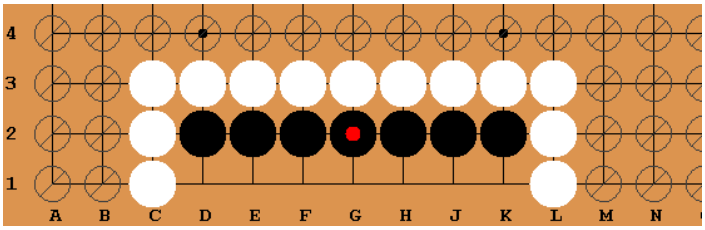


Figure 7.13: 7 stones will live if black plays first or die if white plays first.

In the problem above the computer behaved as expected, placing stones to reduce eye space when killing or forming eyes when playing to live. There are eight possible first moves in this problem including pass. It was necessary to use nearly the whole breadth to solve the problem. A breadth of six always resulted in life for black when playing first and a breadth of seven for white to kill with first move. The hane heuristic will favour white at D1 and K1 from white's perspective. The implemented heuristics are not particularly useful for this problem but it was processed quickly due to a reasonably small search tree.

The program was tested with many further problems, a selection of some of those that could be solved by the program in under five minutes is presented in Appendix A. Appendix B lists a few problems that have been tested with the program were not solved despite processing times of at least 15 minutes. It is possible they could be solved in this time with more experimentation with depth and breadth settings but this is yet to be done.

The main sources for these problems were the introductory go book by Shotwell (2010), two books by Janice Kim (2010) and the Sensei's Library website. (Kyu Exercises at Sensei's Library, no date). It was found that a majority of these problems could be solved fairly rapidly by the program although to achieve this, it was often necessary to limit the boundaries as far as possible without interfering with the nature of the problem (which is not always obvious). As well as playing the optimal first moves as described in the textbooks it was pleasing to see the program went play subsequent moves exactly as described in the text books. When the user played a suboptimal first move, the program again responded as expected and it was usefully demonstrated why this move was a fail.

7.4 User Evaluation

The program was trialled on five separate users, one of whom was an experienced go player at the Glasgow Go Club, two who had been playing go for only a few months and two who had no experience of the game at all. An idea of the general rules of go and the objective of capturing or saving a group of stones was explained to those with no go experience. The users were then asked to perform a series of tasks such as: load a problem and make the computer play the first move. These were printed on an evaluation sheet and the completed sheets are in Appendix C. They were then asked to score the usability of the interface on a five point scale after each task. The richness really though was in watching them interact with the program and the questions they asked verbally when doing so.

On the whole users rated the playing and editing functions either, "intuitive and easy to use" or "mostly easy to use". All managed to complete the tasks without help. Although not asked to use them though, most asked questions about the depth and depth controls. A summary of ideas and feedback that came out of the user evaluation is shown below:

1. Include a stop button as well as start button so that if a problem is taking too long to solve it can be aborted and the settings adjusted. (Currently the only method of stopping the processing is to quit the program)
2. Include an indication of progress towards solving a problem within the GUI. This is an interesting idea, although there is no easy way of assessing the size of the search that will be needed.
3. Package the program with a large bank of problems.
4. Make the GUI resizable so it can be adjusted for eyesight and monitor.
5. Make the "black to play/white to play" window larger so it is more obvious whose turn it is.
6. As discovered in the bent four problem it would be nice to count territory gained.
7. More than one user suggested a pop up window giving instructions on first loading the program.

The experienced player found the three or four problems he tried pretty easy and so not particularly instructive. They proved to be more challenging for a less experienced member of the go club. She had some level of proficiency in a full game (unfortunately beating the author by four points in a 13x13 game) but failed to spot the key move in several of the life and death problems. This player rated the program as "very likely to be useful to a beginner player whereas the more experienced player rated it as "might be useful".

Players with no experience seemed to pick up the idea of liberties and captures very rapidly through interacting with the interface and playing problems.

Chapter 8 Conclusions and Further Work

The approach of constraining the search space around a problem and applying traditional AI algorithms was successful. The project has met the general goals set out in the introduction to produce a program able to solve a selection of life and death problems that would be of interest to a beginning go player. It also meets the requirements set out in the "must" and "should" sections in Chapter 4 although a larger set of initial problems than those listed in Appendix A would undoubtedly make it a more attractive proposition to potential users.

Simple changes that would be well worth implementing came to light during the user evaluation. The most useful of which would be the inclusion of a stop button. It was often the case when experimenting with problems, that after a few minutes of processing it was decided to abandon it and try again with a more restrictive search. Another change would be to increase the size and/or location of the "Black to play" text box to make it more obvious whose turn it is. The notes section could be improved with a scroll bar so that more extensive problem notes could be made by a user. Also this text area is currently one long line with text wrapping, it could be improved with proper handling of carriage returns and multiple lines.

The use of heuristics to extend the capability of the program beyond what is possible with a brute force search were only beginning to be explored. The analysis so far has been limited to reflecting on observed behaviour in a few limited situations. Rather than answering questions this has only given clues as to what kind of questions should be asked. It is not yet obvious how useful the heuristics already implemented are and to what kind of problem. Is there a category of problem for which the mouth heuristic is particularly useful? A deeper understanding of go would be useful for answering these questions as would systematic methods to evaluate it.

During the project there was experimentation with selecting some heuristics and not others and adjusting for whether a player was defending a set of stones or trying to capture it. One approach to investigating the efficacy of different heuristics to particular families of problems might be to build a GUI with selectable heuristics that could be turned on and off. The scope for further work in this area seems almost limitless. There are other good and bad shapes that could be explored. There are also heuristics that do not involve shape. Some go programs make use of an influence function (Bouzy and Cazenave,2001) to give weight to the effect a stone has on neighbouring stones due to its proximity.

Bibliography

Bouzy, B. and Cazenave, T. (2001) 'Computer Go: An AI oriented survey', *Artificial Intelligence*, 132(1), pp. 39–103. doi: 10.1016/s0004-3702(01)00127-8.

British Go Association Home Page | British Go Association (no date) Available at: http://www.britgo.org/ (Accessed: 5 September 2015).

Exhibition Games: Fuego vs Humans (no date) Available at: http://fuego.sourceforge.net/exhibition-games.html (Accessed: 4 September 2015).

Fotland, D. (no date) *The Many Faces of Go, Version 12*. Available at: http://www.smart-games.com/manyfaces.html (Accessed: 5 September 2015).

Go-playing Programs | British Go Association (no date) Available at: http://www.britgo.org/gopcres/playit.html (Accessed: 4 September 2015).

Kim, J., Soo-hyn, J. and Lee, A. (2010) *Learn to Play Go: A Master's Guide to the Ultimate Game (Volume I)*. 3rd edn. United States: Good Move Press.

Kyu Exercises at Sensei's Library (no date) Available at: http://senseis.xmp.net/?KyuExercises (Accessed: 6 September 2015).

Minimax Search with Alpha-Beta Pruning [Algorithm Wiki] (no date) Available at: http://will.thimbleby.net/algorithms/doku.php?id=minimax_search_with_alpha-beta_pruning (Accessed: 31 August 2015).

Nilsson, N. J. and J, N., Nils (1998) *Artificial Intelligence: A New Synthesis*. 4th edn. San Francisco, CA: Morgan Kaufmann Publishers In.

Norvig, P. and Russell, S. J. (2010) *Artificial Intelligence: A Modern Approach: International Version*. United States: Prentice Hall.

Number of Possible Go Games at Sensei's Library (no date) Available at: http://senseis.xmp.net/?NumberOfPossibleGoGames (Accessed: 5 September 2015).

Schaeffer, J., Müller, M. and Kishimoto, A. (2014) *AIs Have Mastered Chess. Will Go Be Next?*. Available at: http://spectrum.ieee.org/robotics/artificial-intelligence/ais-have-mastered-chess-will-go-be-next (Accessed: 4 September 2015).

Schafers, L. (2014) *Parallel Monte-Carlo Tree Search for HPC Systems and its Application to Computer Go*. Germany: Logos Verlag Berlin GmbH.

Shape Collection at Sensei's Library (no date) Available at: http://senseis.xmp.net/?ShapeCollection (Accessed: 26 August 2015).

Shotwell, P., Huiren, Y. and Chatterjee, S. (2011) *Go! More Than a Game More Than Just a Game*. 1st edn. New York: Tuttle Publishing.

Six Die But Eight Live at Sensei's Library (no date) Available at: http://senseis.xmp.net/?SixDieButEightLive (Accessed: 1 September 2015).

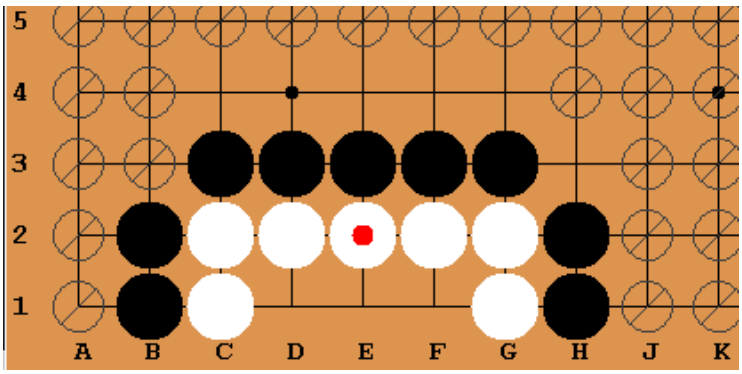
Soo-hyun, J., Kim, J. and Lee, A. (2010) *The Way of the Moving Horse (Learn to Play Go, Volume II) (Learn to Play Go Ser)*. 2nd edn. New York: Good Move Press.

There is Death in the Hane at Sensei's Library (no date) Available at: http://senseis.xmp.net/?ThereIsDeathInTheHane (Accessed: 26 August 2015).

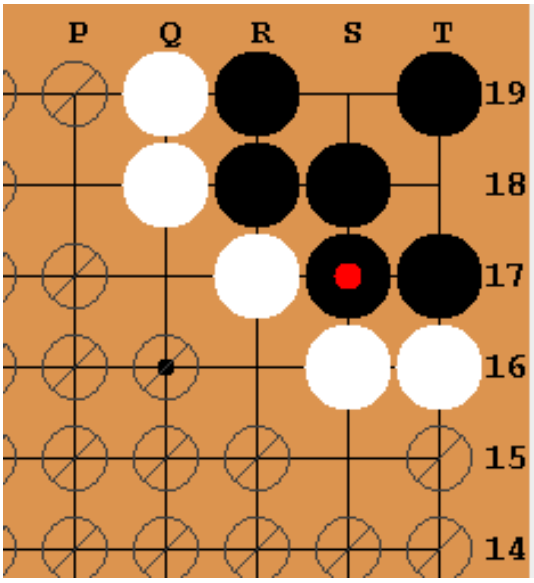
What is good shape? | British Go Association (no date) Available at: http://www.britgo.org/bgj/06224.html (Accessed: 26 August 2015).

Appendix A Problems Solved by the program in under five minutes.

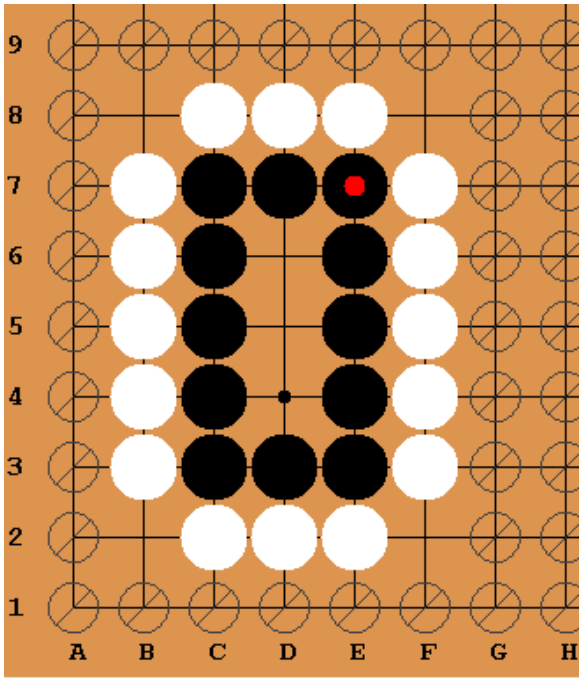
The following problems were all successfully tested in the program with the tree setting indicated. Where used, D stands for Depth Limit, B for Breadth Limit and A for Answer.



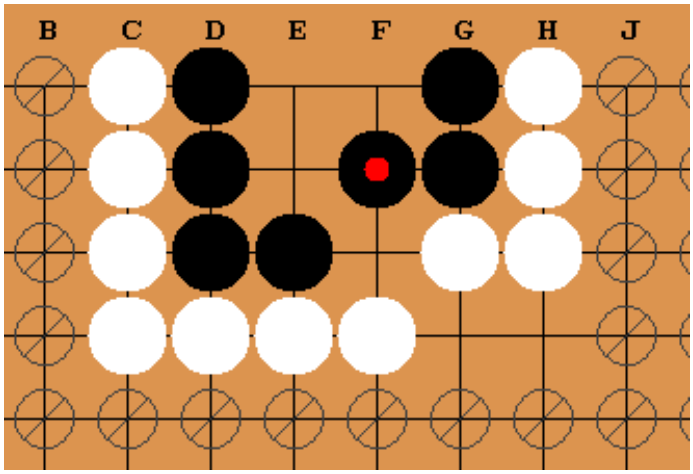
P1: Black to kill. Optimal move E1. D=6 B=8.



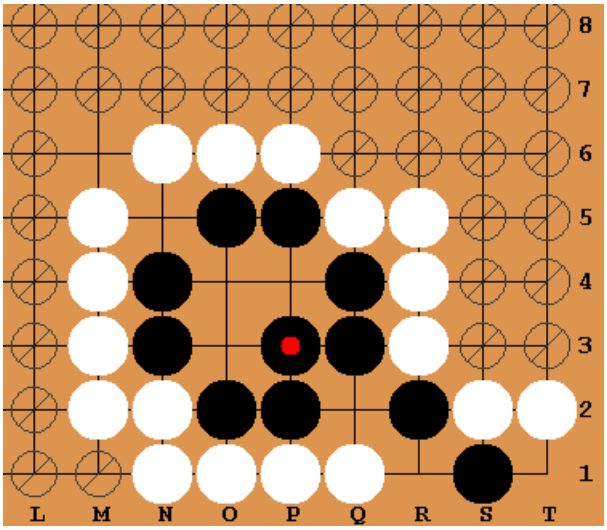
P2: Black to live. Optimal move T19 (played). D=5 B=8.



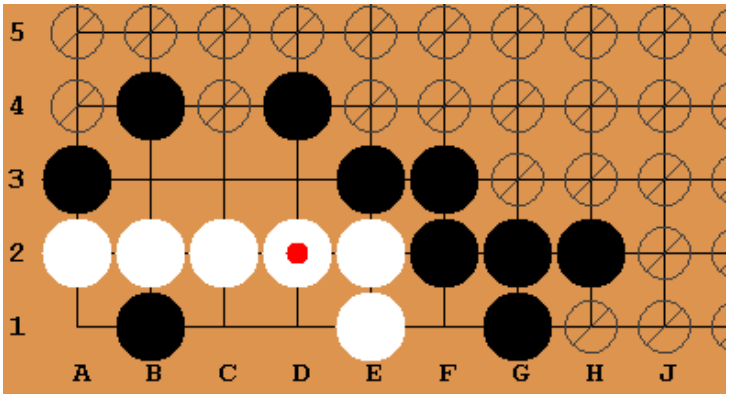
P3: White to kill. D=3, B=4.



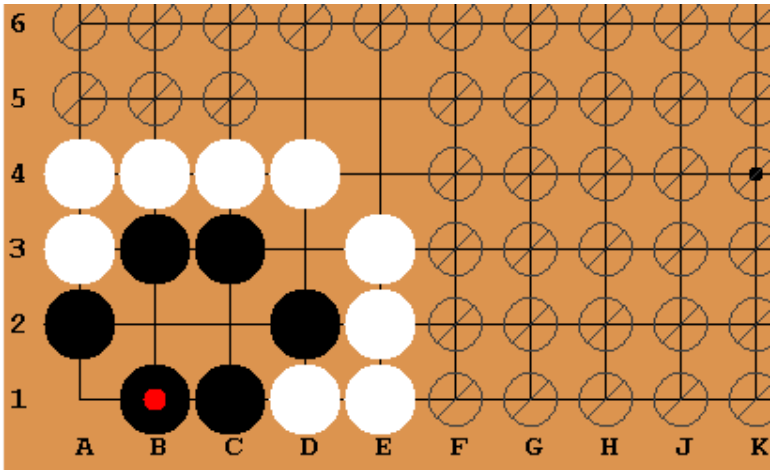
P4: White to kill. D=6, B=8 (A=E19)



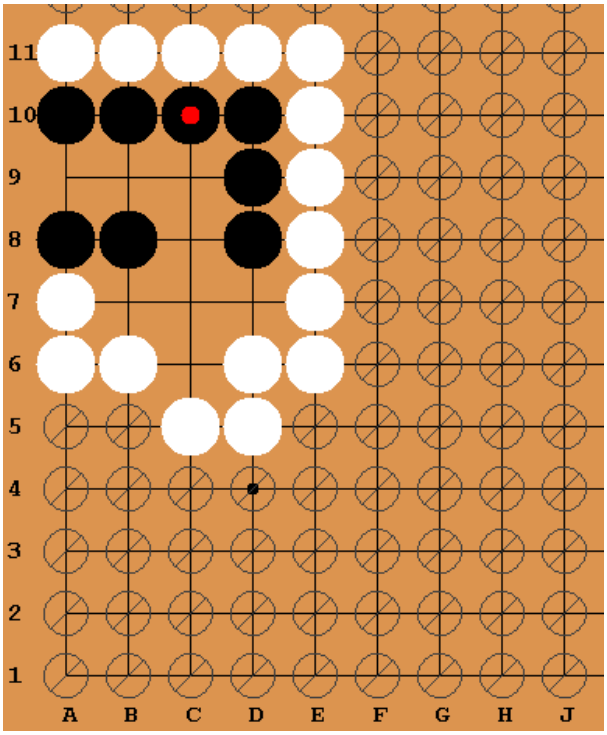
P5: White to kill. D=6, B=8, A=O4 (Black and white form mouths here)



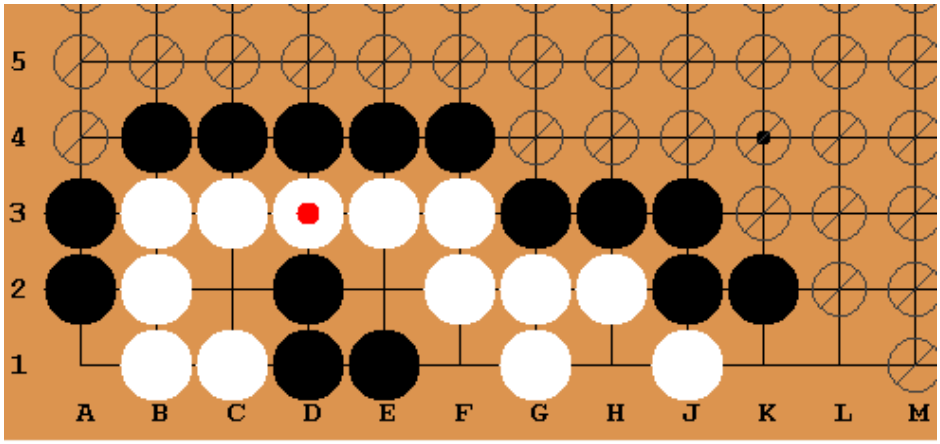
P6: Black to kill. D=10, B=5, A=C1 (Depth 10+ needed)



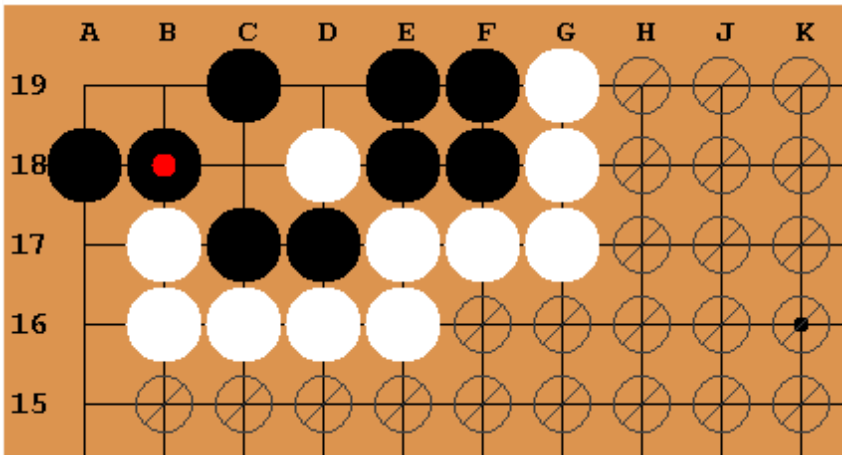
P7: White to kill. D=12, B=7, A=C2



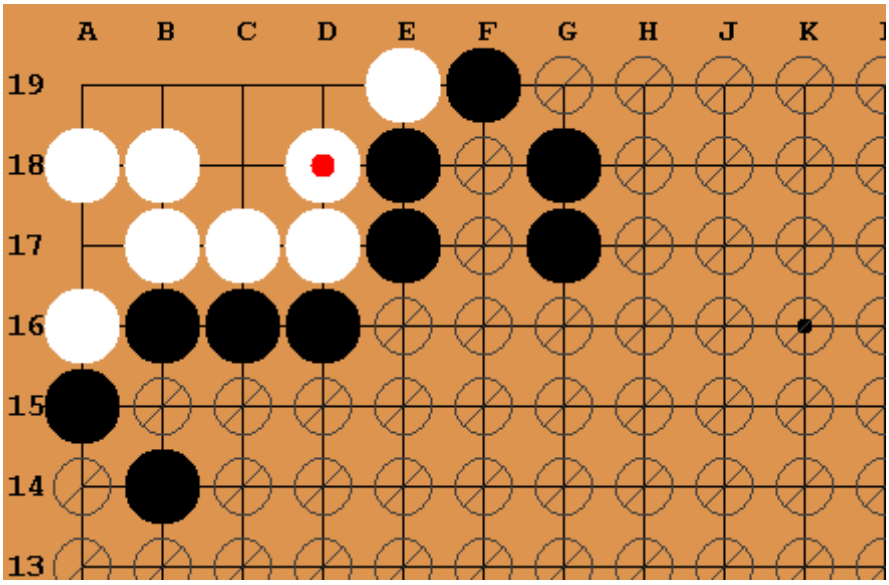
P8: Black to live: A Bent Four Problem. The optimal move is C7 gaining 3 points. B9 for life with two points. D=15, B=7.



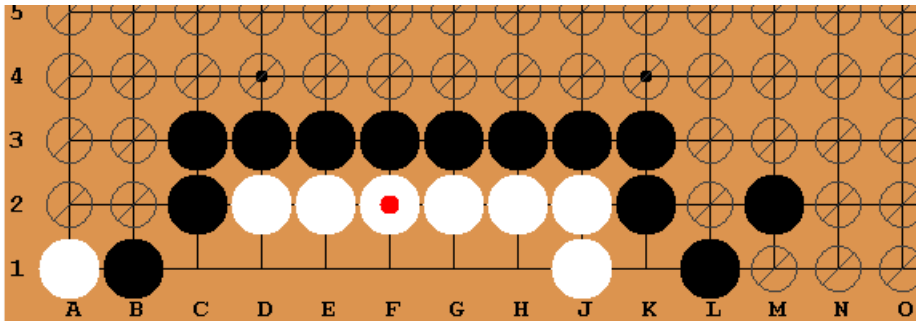
P9: Black to kill white. D=8, B=7, A=E2



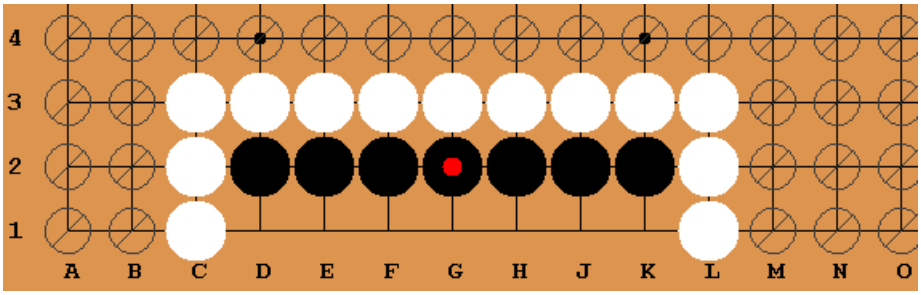
P10: Black to live. D=7, B=8, A=C18 (Allowing white to capture four stones)



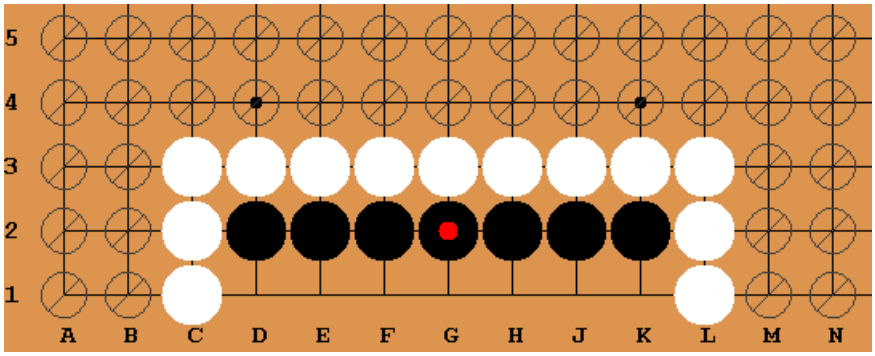
P11: Black to kill white. D=6, B=6, A=C19



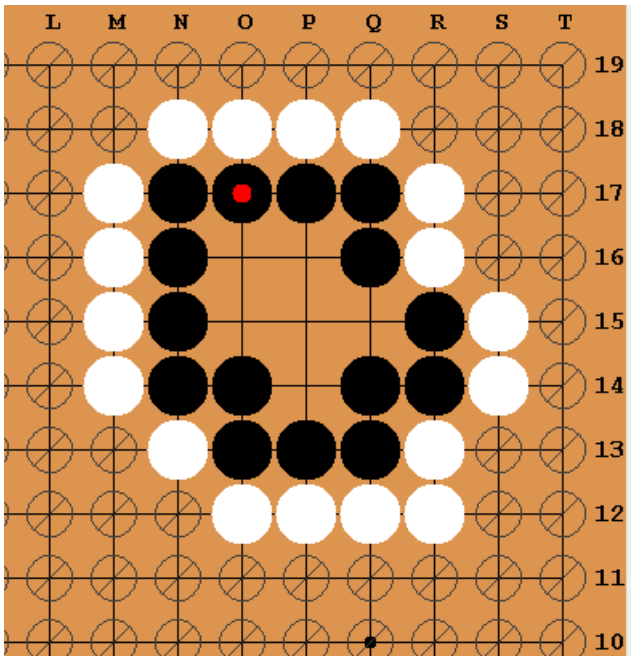
P12: Black to kill white. D=6, B=6



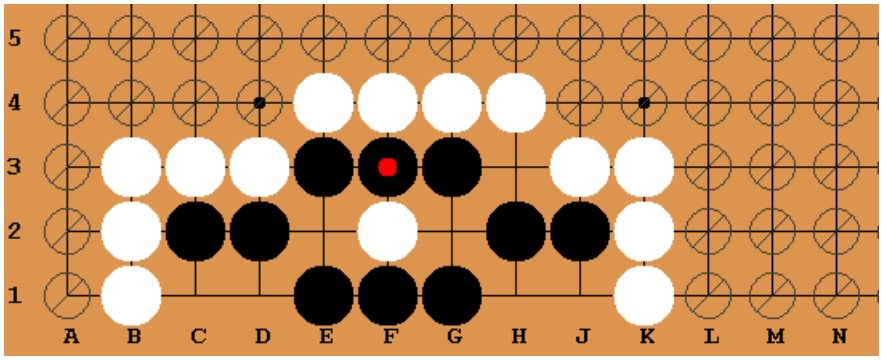
P13: White first to kill. D=11, B=8



P14: Black first to live. D=11, B=8

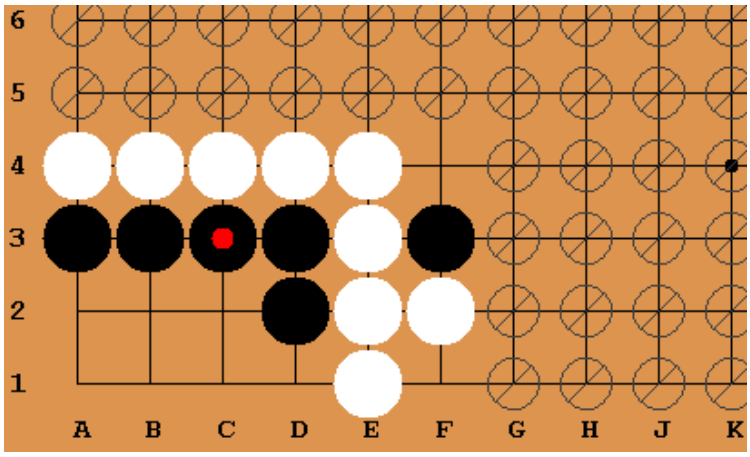


P15: Flower Six(rabitty six). White to kill. D=5, B=6

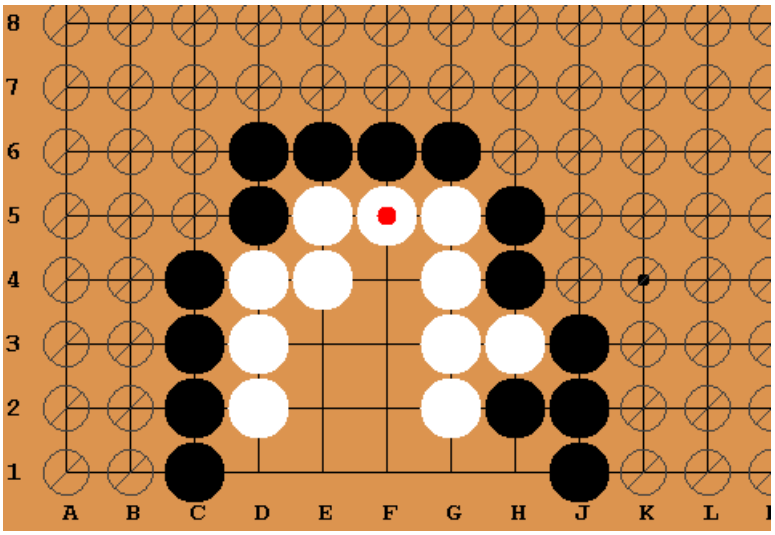


P16: White to capture all black. D=10, B=8, A= E2 (Blacks are captured in several moves, the game played out exactly as described at http://senseis.xmp.net/?KyuExercise17%2FSolution)

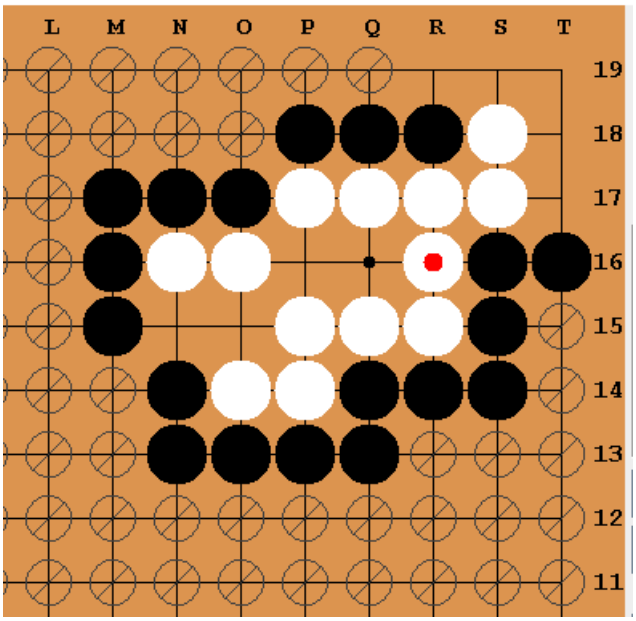
Appendix B Problems that the program did not solve in 15 minutes on first testing



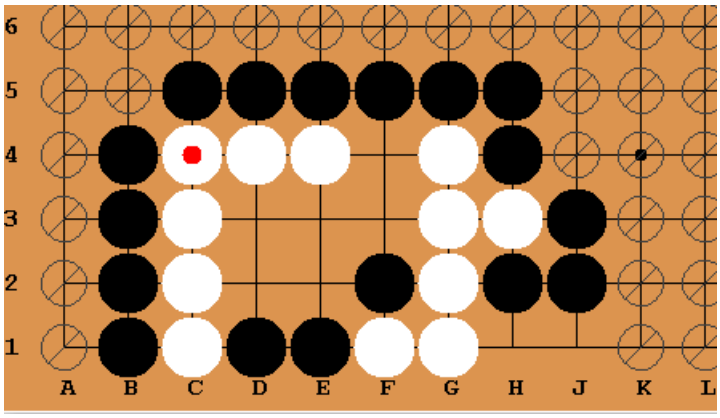
H1: Black to play first and live. A=B1



H2: Black first to kill white. A=G1



H3: Black first to kill



H4: Black first to kill, A=D2

Appendix C User Evaluations

Go Life and Death Solver: User Evaluations

Thank you for taking part in this user evaluation! Your feedback is much appreciated and will be used to inform further development of the program.

The program is designed as a tool to help players learn about or improve their play in life and death situations. If the computer is selected as an opponent it should play the best move to either defend or kill a specified group of stones.

Please have a go at the tasks to below and rate their ease of use:

1) **Try and load a problem from the folder called Simple Problems.**

[1] Intuitive and easy to use.
[2] mostly easy to use
[3] ok
[4]somewhat confusing
[5] I couldn't work it out

2) **Try the problem, making the computer play the first move.**

[1] Intuitive and easy to use.
[2] mostly easy to use
[3] ok
[4]somewhat confusing
[5] I couldn't work it out

3) **Load another problem, this time try playing the first move yourself.**

[1] Intuitive and easy to use.
[2] mostly easy to use
[3] ok
[4]somewhat confusing
[5] I couldn't work it out

4) **It is possible to edit a problem or create your own, try changing a problem or add your own that you'd like it to solve, can you:**
    a. Add stones
    b. Remove stones
    c. Select the group of stones to be killed or protected in this problem.
    d. Set any spaces to out of bounds that are not relevant to the problem
    e. Edit the notes to the problem

[1] Intuitive and easy to use.

[2] Mostly easy to use

[3] Ok

[4] Somewhat confusing

[5] I couldn't work it out

5) **Do you feel this product could be useful to a beginner go player?**

[1] Likely to be very useful.

[2] Likely to be useful

[3] It might be useful

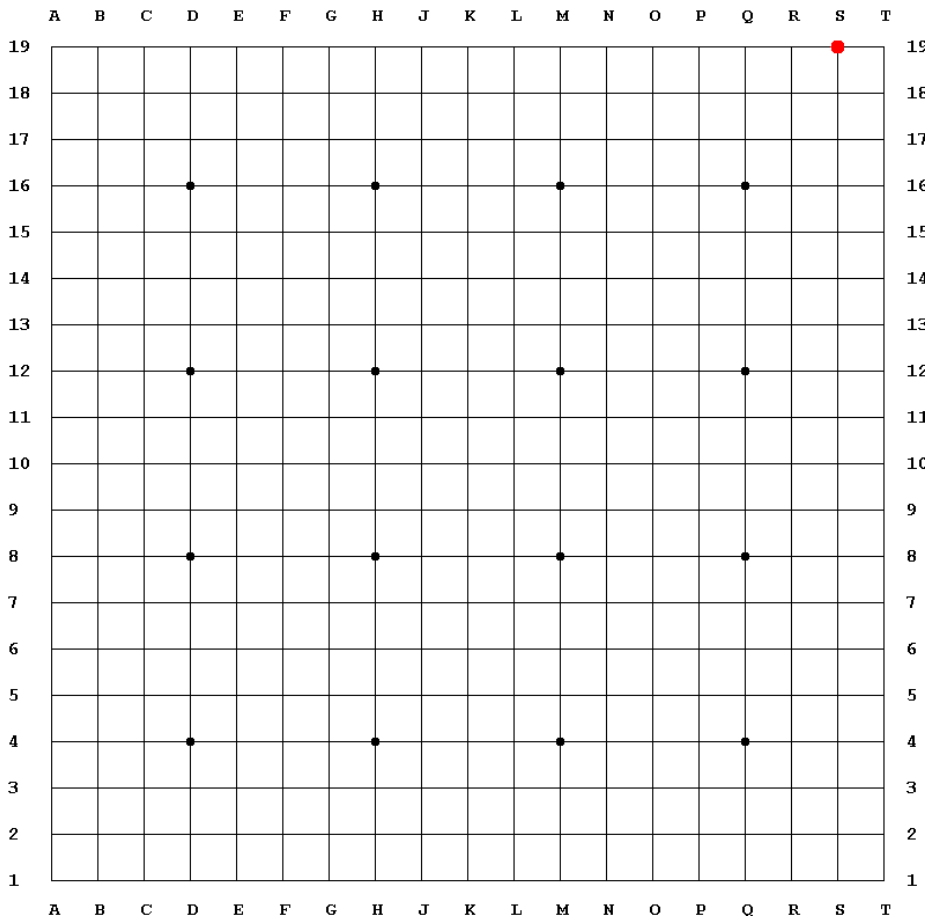
[4] Probably of little use.

[5] Definitely not

6) **Please write down any additional features or changes that you feel could improve a product like this:**

7) **Finally, if you have any favourite Life and Death problems that you think it would be nice to include with the program, feel free to sketch them on the board below.**

Many thanks for your help!!



Experienced Player. - Go Club.

Go Life and Death Solver: User Evaluation

Thank you for taking part in this user evaluation! Your feedback is much appreciated and will be used to inform further development of the program.

The program is designed as a tool to help players learn about or improve their play in life and death situations. If the computer is selected as an opponent it *should* play the best move to either defend or kill a specified group of stones.

Please have a go at the tasks to below and rate their ease of use:

1) **Try and load a problem from the folder called Simple Problems.**
   a. (1) Intuitive and easy to use.
   b. [2] mostly easy to use
   c. [3] ok
   d. [4]somewhat confusing
   e. [5] I couldn't work it out

2) **Try the problem, making the computer play the first move.**
   a. (1) Intuitive and easy to use.
   b. [2] mostly easy to use
   c. [3] ok
   d. [4]somewhat confusing
   e. [5] I couldn't work it out

3) **Load another problem, this time try playing the first move yourself.**
   a. (1) Intuitive and easy to use.
   b. [2] mostly easy to use
   c. [3] ok
   d. [4]somewhat confusing
   e. [5] I couldn't work it out

4) **It is possible to edit a problem or create your own, try changing a problem or add your own that you'd like it to solve:**
   a. Add stones
   b. Remove stones
   c. Select the group of stones to be killed or protected in this problem.
   d. Set any spaces to out of bounds that are not relevant to the problem
   e. Edit the notes to the problem

[1] intuitive and easy to use.
(2) mostly easy to use
[3] ok
[4]somewhat confusing
[5] I couldn't work it out

5) **Do you feel this product could be useful to a beginner go player?**

[1] Likely to be very useful.
[2] Likely to be useful
(3) It might be useful
[4] Probably of little use.
[5] Definitely not

6) **Please write down any additional features or changes that you feel could improve a product like this:**

- More problems
- Stop button

7) **Finally, if you have any favourite Life and Death problems that you think it would be nice to include with the program, feel free to sketch them on the board below.**

Many thanks for your help!!

Go Life and Death Solver: User Evaluation

Thank you for taking part in this user evaluation! Your feedback is much appreciated and will be used to inform further development of the program.

The program is designed as a tool to help players learn about or improve their play in life and death situations. If the computer is selected as an opponent it *should* play the best move to either defend or kill a specified group of stones.

Please have a go at the tasks to below and rate their ease of use:

1) **Try and load a problem from the folder called Simple Problems.**
   a. [1] intuitive and easy to use.
   b. [2] mostly easy to use
   c. [3] ok
   d. [4]somewhat confusing
   e. [5] I couldn't work it out

2) **Try the problem, making the computer play the first move.**
   a. [1] intuitive and easy to use.
   b. [2] mostly easy to use
   c. [3] ok
   d. [4]somewhat confusing
   e. [5] I couldn't work it out

3) **Load another problem, this time try playing the first move yourself.**
   a. [1] intuitive and easy to use.
   b. [2] mostly easy to use
   c. [3] ok
   d. [4]somewhat confusing
   e. [5] I couldn't work it out

4) **It is possible to edit a problem or create your own, try changing a problem or add your own that you'd like it to solve:**
   a. Add stones
   b. Remove stones
   c. Select the group of stones to be killed or protected in this problem.
   d. Set any spaces to out of bounds that are not relevant to the problem
   e. Edit the notes to the problem

   [1] Intuitive and easy to use.
   [2] mostly easy to use
   [3] ok
   [4]somewhat confusing
   [5] I couldn't work it out

5) **Do you feel this product could be useful to a beginner go player?**
   [1] Likely to be very useful.
   [2] Likely to be useful
   [3] It might be useful
   [4] Probably of little use.
   [5] Definitely not

6) **Please write down any additional features or changes that you feel could improve a product like this:**

   Resizable.

7) **Finally, if you have any favourite Life and Death problems that you think it would be nice to include with the program, feel free to sketch them on the board below.**

Many thanks for your help!!

[redacted] - no experience.

Go Life and Death Solver: User Evaluation

Thank you for taking part in this user evaluation! Your feedback is much appreciated and will be used to inform further development of the program.

The program is designed as a tool to help players learn about or improve their play in life and death situations. If the computer is selected as an opponent it *should* play the best move to either defend or kill a specified group of stones.

Please have a go at the tasks to below and rate their ease of use:

1) **Try and load a problem from the folder called Simple Problems.**
   a. (1) intuitive and easy to use.
   b. [2] mostly easy to use
   c. [3] ok
   d. [4]somewhat confusing
   e. [5] I couldn't work it out

2) **Try the problem, making the computer play the first move.**
   a. [1] intuitive and easy to use.
   b. (2) mostly easy to use
   c. [3] ok
   d. [4]somewhat confusing
   e. [5] I couldn't work it out

3) **Load another problem, this time try playing the first move yourself.**
   a. [1] intuitive and easy to use.
   b. (2) mostly easy to use
   c. [3] ok
   d. [4]somewhat confusing
   e. [5] I couldn't work it out

4) **It is possible to edit a problem or create your own, try changing a problem or add your own that you'd like it to solve:**
   a. Add stones
   b. Remove stones
   c. Select the group of stones to be killed or protected in this problem.
   d. Set any spaces to out of bounds that are not relevant to the problem
   e. Edit the notes to the problem

   (1) intuitive and easy to use.
   [2] mostly easy to use
   [3] ok
   [4]somewhat confusing
   [5] I couldn't work it out

5) **Do you feel this product could be useful to a beginner go player?**

   [1] Likely to be very useful.
   (2) Likely to be useful
   [3] It might be useful
   [4] Probably of little use.
   [5] Definitely not

6) **Please write down any additional features or changes that you feel could improve a product like this:**

   Show what colour player is.
   Link to online go instructions/tips.

7) **Finally, if you have any favourite Life and Death problems that you think it would be nice to include with the program, feel free to sketch them on the board below.**

Go Life and Death Solver: User Evaluation

Thank you for taking part in this user evaluation! Your feedback is much appreciated and will be used to inform further development of the program.

The program is designed as a tool to help players learn about or improve their play in life and death situations. If the computer is selected as an opponent it *should* play the best move to either defend or kill a specified group of stones.

Please have a go at the tasks to below and rate their ease of use:

1) **Try and load a problem from the folder called Simple Problems.**
   a. (1) intuitive and easy to use.
   b. [2] mostly easy to use
   c. [3] ok
   d. [4]somewhat confusing
   e. [5] I couldn't work it out

2) **Try the problem, making the computer play the first move.**
   a. (1) intuitive and easy to use.
   b. [2] mostly easy to use
   c. [3] ok
   d. [4]somewhat confusing
   e. [5] I couldn't work it out

3) **Load another problem, this time try playing the first move yourself.**
   a. [1] intuitive and easy to use.
   b. (2) mostly easy to use
   c. [3] ok
   d. [4]somewhat confusing
   e. [5] I couldn't work it out

4) **It is possible to edit a problem or create your own, try changing a problem or add your own that you'd like it to solve:**
   a. Add stones
   b. Remove stones
   c. Select the group of stones to be killed or protected in this problem.
   d. Set any spaces to out of bounds that are not relevant to the problem
   e. Edit the notes to the problem

   [1] intuitive and easy to use.
   (2) mostly easy to use
   [3] ok
   [4]somewhat confusing
   [5] I couldn't work it out

5) **Do you feel this product could be useful to a beginner go player?**

   [1] Likely to be very useful.
   (2) Likely to be useful
   [3] It might be useful
   [4] Probably of little use.
   [5] Definitely not

6) **Please write down any additional features or changes that you feel could improve a product like this:**

   ShowInstruction popup before starting the game

7) **Finally, if you have any favourite Life and Death problems that you think it would be nice to include with the program, feel free to sketch them on the board below.**

Go Life and Death Solver: User Evaluation

Thank you for taking part in this user evaluation! Your feedback is much appreciated and will be used to inform further development of the program.

The program is designed as a tool to help players learn about or improve their play in life and death situations. If the computer is selected as an opponent it *should* play the best move to either defend or kill a specified group of stones.

Please have a go at the tasks to below and rate their ease of use:

1) **Try and load a problem from the folder called Simple Problems.**
   a. (1) intuitive and easy to use.
   b. [2] mostly easy to use
   c. [3] ok
   d. [4]somewhat confusing
   e. [5] I couldn't work it out

2) **Try the problem, making the computer play the first move.**
   a. [1] intuitive and easy to use.
   b. [2] mostly easy to use
   c. (3) ok
   d. [4]somewhat confusing
   e. [5] I couldn't work it out

Move "Black to Play" lower down the screen. + larger font size

3) **Load another problem, this time try playing the first move yourself.**
   a. [1] intuitive and easy to use.
   b. (2) mostly easy to use
   c. [3] ok
   d. [4]somewhat confusing
   e. [5] I couldn't work it out

4) **It is possible to edit a problem or create your own, try changing a problem or add your own that you'd like it to solve:**
   a. Add stones
   b. Remove stones
   c. Select the group of stones to be killed or protected in this problem.
   d. Set any spaces to out of bounds that are not relevant to the problem
   e. Edit the notes to the problem

   [1] intuitive and easy to use.
   (2) mostly easy to use
   [3] ok
   [4]somewhat confusing
   [5] I couldn't work it out

5) **Do you feel this product could be useful to a beginner go player?**
   (1) Likely to be very useful.
   [2] Likely to be useful
   [3] It might be useful
   [4] Probably of little use.
   [5] Definitely not

6) **Please write down any additional features or changes that you feel could improve a product like this:**

   - More guidance in pop-up window

7) **Finally, if you have any favourite Life and Death problems that you think it would be nice to include with the program, feel free to sketch them on the board below.**