Design and Analysis of a Computer Go Program

By Charlie Howlett

Introduction:

For my EPQ I set out to determine if a non-specialist in both Go and artificial intelligence could create a computer program capable of playing the board game Go. I attempted to answer this question by creating my own computer Go program based on popular techniques in the Computer Go community.

In this report I will outline the decisions made whilst writing this program, explain the techniques that I have chosen to implement in my program (and why I have chosen to implement them), comment on problems encountered in producing my program and finally I will evaluate the competence of my program.

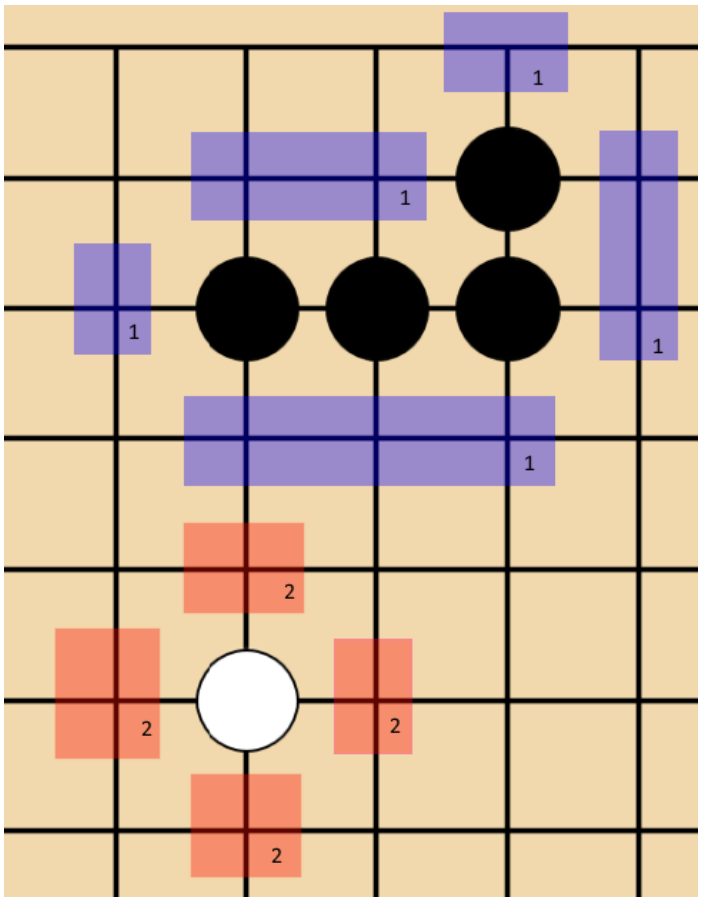
I have chosen computer Go as my EPQ topic because of the infamous challenge that is has become in the field of artificial intelligence. This is because Go is a very intuitive board game, making it quite easy for humans to grasp but difficult for a computer to play properly. Also, Go has a particularly large state space (the number of different states that the game can exist in) (Gelly, S, et al., 2012) meaning that techniques used by computers in the past, to master games such as Chess, that utilize a "brute force" (Gelly, S, et al., 2012) approach (explained later in this essay) cannot be applied to Go. As such, more sophisticated techniques have been developed to address the problem of computer Go, yet the ability of even professional computer Go programs are still limited (to the ability of lower level professional humans at best).

Throughout this essay I have provided diagrams (that I've independently created) to help convey understanding of topics to the reader.

Introduction to Go:

In order for the reader to better understand the algorithms I will outline in this essay, a basic understanding of Go is required. In this section of my essay, I will briefly explain the rules of the game.

Go is an ancient Chinese board game. In Go, two players sequentially place black (for the black team) or white (for the white team) stones on the intersections of a uniform, square grid (usually 19 by 19 intersections in size, although beginners typically play on 9 by 9 boards). The number of stones that can be placed by each player is theoretically infinite (as stones can be repeatedly "captured"/taken from the board and placed on the board). And, like Chess, Go is a perfect information game meaning that both players are privy to all of the information of the game (i.e: each player can see all of the positions of all of their and their opponent's stones). In a players own turn they may pass, subsequently forfeiting their turn, or place one stone on any legal intersection. A game ends when both players have consecutively passed. The only way to remove pieces from the board is to capture them, this is done when a group of stones' "liberties" are filled completely by stones of the opposite team. The liberties of a stone, or a group of stones, are the intersections that surround the stone, or the edge of the group of stones.



his visual aid shows:

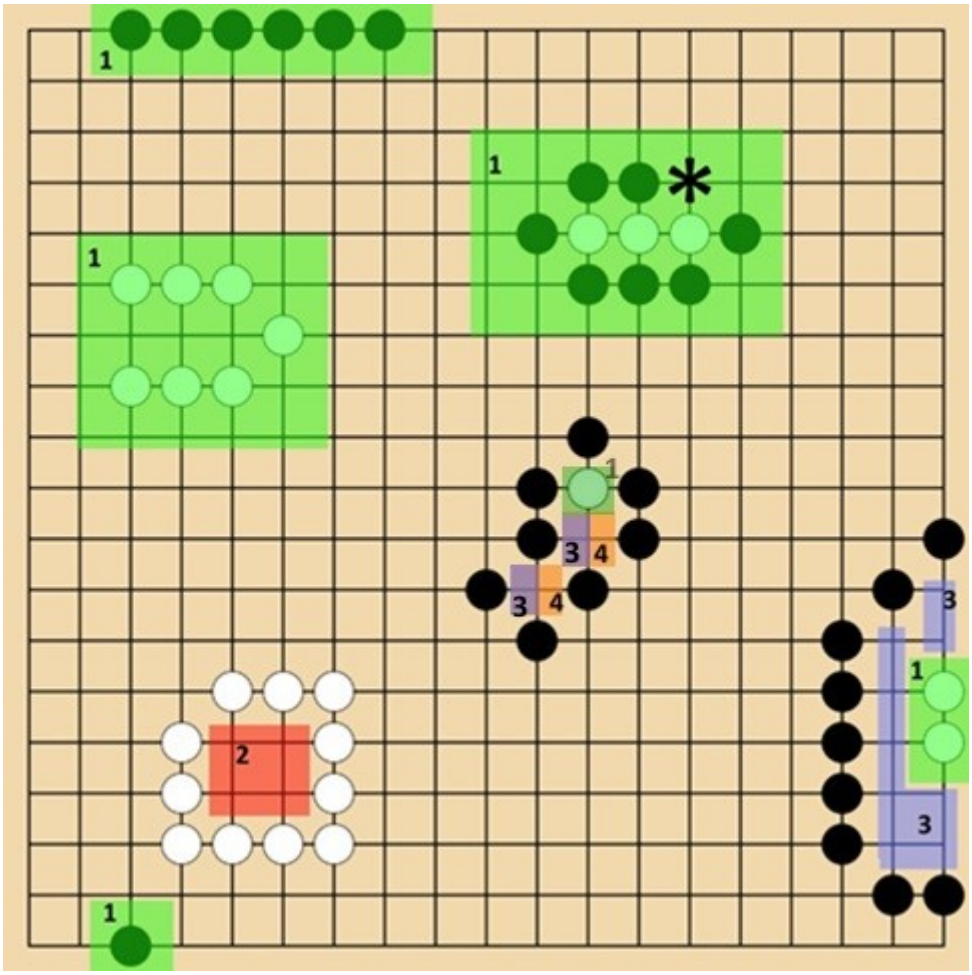
***1 (Highlighted in blue)**, liberties of the black group of stones.*

***2 (Highlighted in red)**, liberties of the white stone.*

**In layman's terms, to capture a teams' stones/stone they/it must be completely surrounded by the opposite team's stones. This is why Go is often referred to as the "Surrounding game".*

One of the few rules of Go is that a stone cannot be placed on an intersection that commits that stone to capture (i.e: be placed in a position where it's liberties are filled by the opposing team).

At the end of the game (using Japanese scoring), each player receives a score (the number of board intersections surrounded by their own pieces with the number of pieces of theirs that have been captured subtracted) and the player with the highest score wins. In an even game, black places the first stone and it is traditional for white to pass last for the game to end (to make up for the advantage that black is given by moving first).



This visual aid shows:

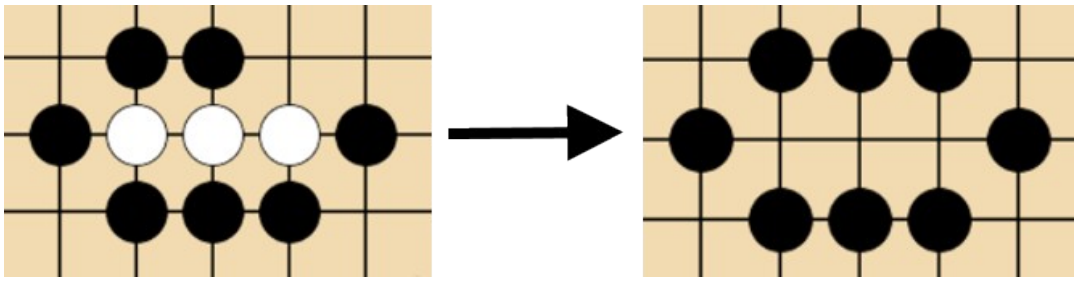
1 (Highlighted in green), *stones that contribute no score.*

2 (Highlighted in red), *intersections that contribute score to the white team (in this case 4 intersections are completely surrounded so the white team has a score of 4).*

3 (Highlighted in blue), *intersections that contribute score to the black team (in this case 9 intersections are completely surrounded so the black team has a score of 9).*

4 (Highlighted in orange), *intersections on which that white cannot place a piece as it would result in the immediate capture of that piece.*

An asterisk, *this asterisk marks the point at which black would have to place a stone to capture the enclosed white stones, as shown below.*



This move would result in 3 captured white pieces (reducing the white team's score by 3) and territory for black that would increase the black team's score by 3.

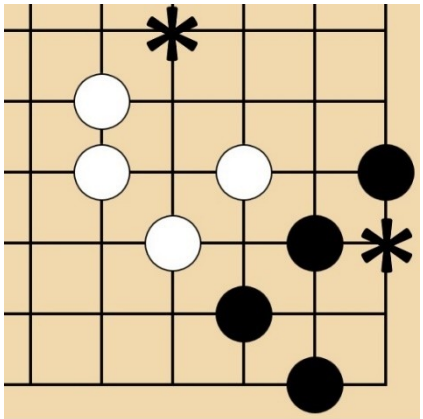
(More information on the rules of Go can be found at: "Sensei's Library: Rules of Go", 2015).

Problems for Go AI:

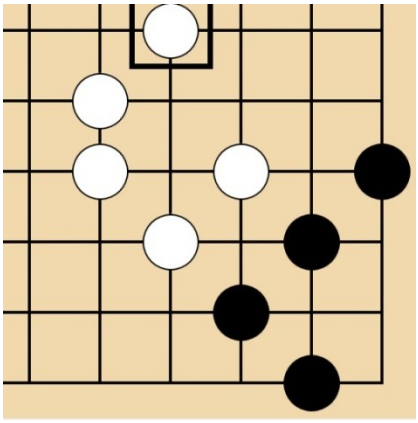
In this section I will discuss some of the problems facing current Go AI algorithms.

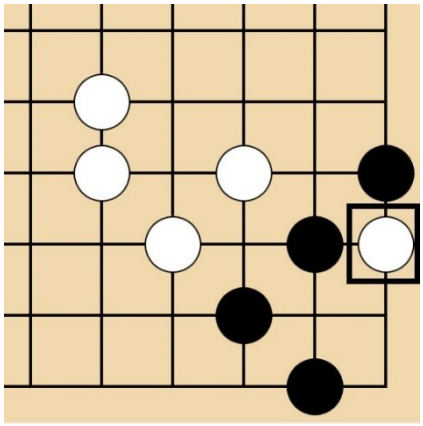
One of the most limiting factors to computer Go is that techniques used by previously successful board game AI are typically not applicable to Go. For example, a "brute force" search is a technique that is very effective when applied to games with smaller state spaces (Gelly, et al., 2012) (a state space is the number of possible states of a game), such as tic tac toe, in which the computer simulates thousands of games from the games current state and then selects the move with the most promising simulated results. However, this widely used technique is not applicable to Go. One of the reasons behind this is that the state space of Go is so large that it would not be possible to simulate a reasonable amount of the possible moves in a sensible amount of time. There are 361 possible moves in only the first move of Go (compared to the 20 possible first moves in Chess) meaning that in simulating all moves in just a 2 ply brute search of Go would mean simulating 129,960 moves. And even if this search were performed it is impossible for any territory to be claimed for either team in the first 2 moves so it would be difficult for the computer to evaluate either state as better or worse than another.

Another key problem facing Go AI is board evaluation. There is not yet a very accurate method of evaluation of board states. For example, if the computer (white team) is comparing the two moves (denoted by asterisks in the diagram below).



The score of making the move represented by the diagram to the right is no different than the original score (even though now the white team only needs to place one more piece to capture territory). Because of this, a computer judging moves purely on the score they result in may consider this as a worthless move, even though it is clearly not





The score of making the move represented by the diagram to the left reduces the score of the black team by 1 and thus, a computer that judges moves purely on the score they result in may regard this as a good move. Though this is clearly not true as the recently placed white piece can be taken by black in just 1 move (and in some scoring systems of Go this would be counted as a "dead stone" and it would be removed from black's territory before adding the score the black group has accumulated).

This example shows that to evaluate a move in Go not only must a resultant board state be analyzed but many, many more further resulting board states must be analyzed too (as in Go it usually takes multiple, strategic moves before score is added/subtracted). And as the board becomes more and more complex this becomes much harder to do.

I chose to create a Go playing program because of the aforementioned problems. Proficient AI for board games such as Chess have become trivial to create as they are somewhat simpler for computers to evaluate and simulate than Go. In comparison Go offers a much more interesting and difficult challenge.

AI techniques used in Go:

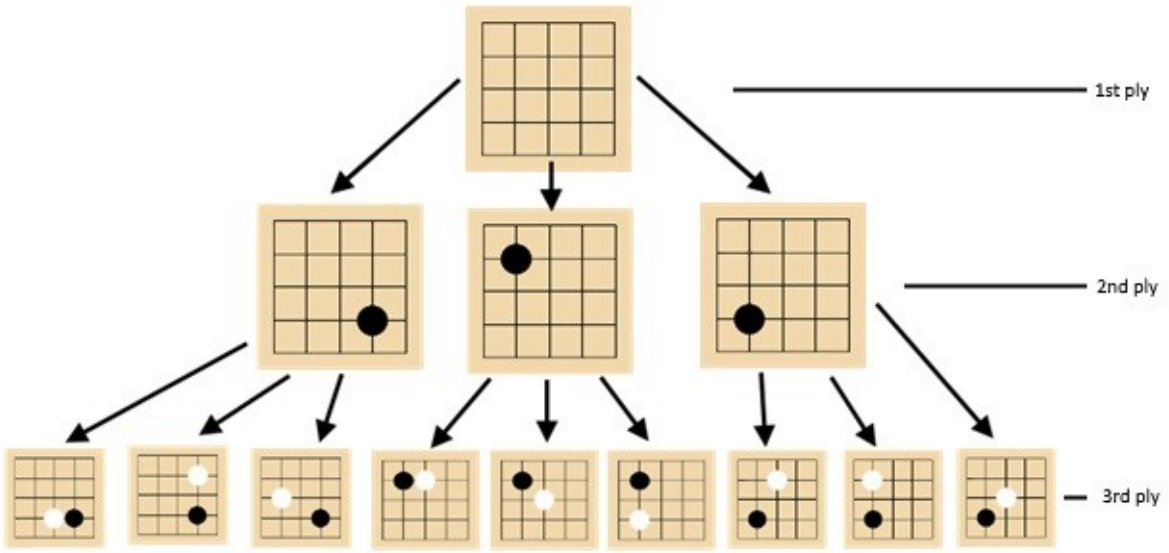
In this section, I will discuss three of the algorithms that I have implemented in my program.

Monte Carlo tree search (MCTS):

MCTS is a heuristic search algorithm that is used to find a move for the computer player to play. A heuristic algorithm is one that returns a solution to an intractable problem (a problem that would take an infeasible amount of time to find an optimal solution). Whilst it is not guaranteed that the selected move is optimal, it will most likely be a good move (Chaslot, 2008).

The algorithm was first brought into popularity amongst the computer Go community with its mention in a paper by Rémi Coulom (2006). Ever since its introduction, all of the best performing programs have used Monte Carlo search methods. Basically, the Monte Carlo tree search method

has 4 steps (selection, expansion, simulation and backpropagation) which continue to execute consecutively until a specific time limit is elapsed (so that an answer is returned within a reasonable time limit). The algorithm is applied to a game tree, which is a directed graph that illustrates a game as it progresses (as illustrated below).

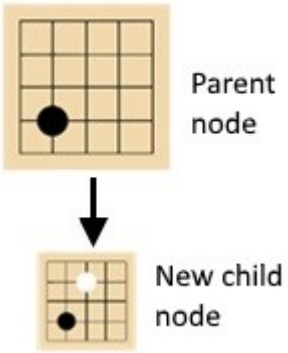


Selection:

In this phase of the algorithm, starting from the current game state, the algorithm selects a random, legal move. If this move has not already been explored, then that move is selected. If the move has already been explored then another random legal move is selected from the game state resulting from that move, until a non-explored move is selected.

Expansion:

Next, a new node is added to the tree that represents the move selected in the selection phase. The node is added as a child node of the node from which the move is being made from (as is shown in the diagram to the right).



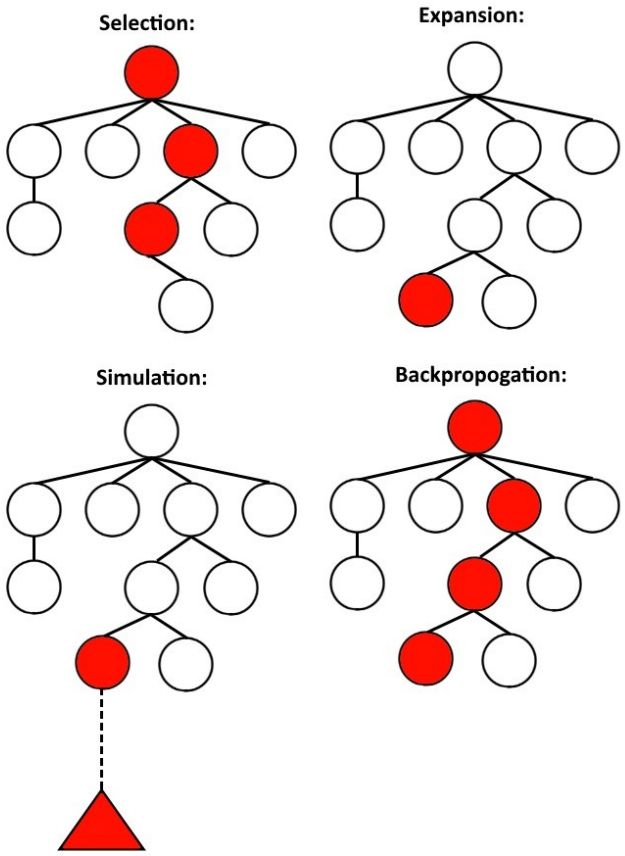
Simulation:

From the new child node, a random game of Go is simulated to completion and the score is calculated (the computer picks random moves for both teams during these simulations). If the computers team wins the random game then it is valued at 1, losing it is valued at -1 and drawing it is valued at 0. Some programs use hard coded “policies” by which moves are picked in the simulation based on logic encoded usually by professional players. This is better than evaluating a move based on a randomly simulated game as a random game is not likely to match a game being played from this point (as the random moves take no account of strategy) and thus this would better reflect the chances for the computer to win/lose/ draw from the board state being evaluated.

Backpropagation:

During backpropagation, the computer, starting from the recently created and evaluated child node, updates the values stored by each node along the route back to the initial node (i.e. the node representing the actual, current game state).

Once the time limit has been elapsed then the move with the best percentage of wins to simulations is selected and the computer makes that move. (The below diagram roughly shows application of the MCTS algorithm).



UCT (Upper Confidence bounds applied to Trees):

UCT is the application of UCB (upper confidence bounds) methods to, in this case, game trees (Browne, et al, 2012). The algorithm affects which nodes are selected in the selection phase of the MCTS algorithm. The advantage of using this algorithm to select which nodes to explore is that is balances the necessity to explore moves of which the computer has not yet searched and to search further into moves that seem to have promising results. It does this by treating the choice of possible random moves as a "multi-armed bandit problem". An example of a multi-armed bandit problem is a gambler in a casino. The premise being that the gambler can pick to play any of a large variety of slot machines, each time the gambler does so he/she gets a certain amount of money back, the amount of which is dependent on the slot machine. The gambler aims to find the slot machine with the biggest pay back amount and thus accumulate the largest amount of money. This can be likened to the selection phase of the Monte Carlo search algorithm in which each possible node can be valued by its likelihood in resulting in a winning game situation and the goal is to find the move with the maximum value, which is the optimal play. It does this by giving each node a calculated UCT value and choosing to explore the node to explore with the highest value, as calculated below (Rimmel, et al., 2010):

$$UCT = \acute{X}_j + C_p\sqrt{\frac{\ln(n)}{n_j}}$$

Where $\acute{X}_j$ is the expected value of making the child node's move (i.e: $\acute{X}_j = \frac{W_j}{n_j}$, the number of times that simulating the child nodes as resulted in a victory divided by the total simulations of that child node); C_p is a constant in the range $0 < C_p < 1$ (that dictates the amount of exploration compared to exploitation, in my program I used $\sqrt{2}$. I explain why I have used this variable in the *Implementation and Analysis* chapter of this essay); n is the number of times that the parent node, of the child node in question, has been visited and n_j is the number of times that the child node in question has been visited.

RAVE (Rapid Action Value Evaluation):

Whereas, the pure MCTS algorithm assigns value to game states indicative of the score of the computer at the end of a simulated game, simulated from this game state, the RAVE algorithm makes the assumption that a move is just as valuable at any point in the game and thus assigns values to individual moves. *(Rimmel, et al., 2010)*. This means that whenever a move is evaluated by simulation the program checks for similar moves elsewhere in the game tree and updates the values of these nodes too. A lot of the time in Go two moves made at different points in the game are just as equally valuable. For example, the below diagram shows a move made by white (X) that looks promising as it challenges black piece:

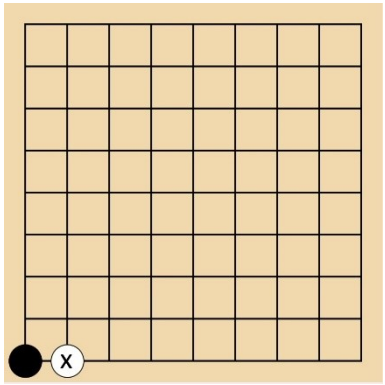
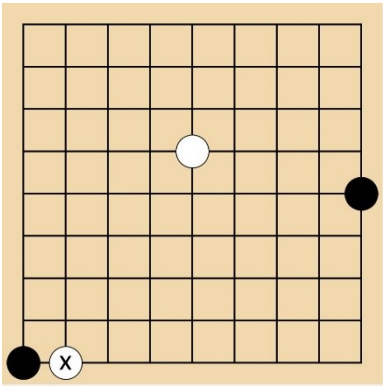


Figure 1, left

Figure 2, right



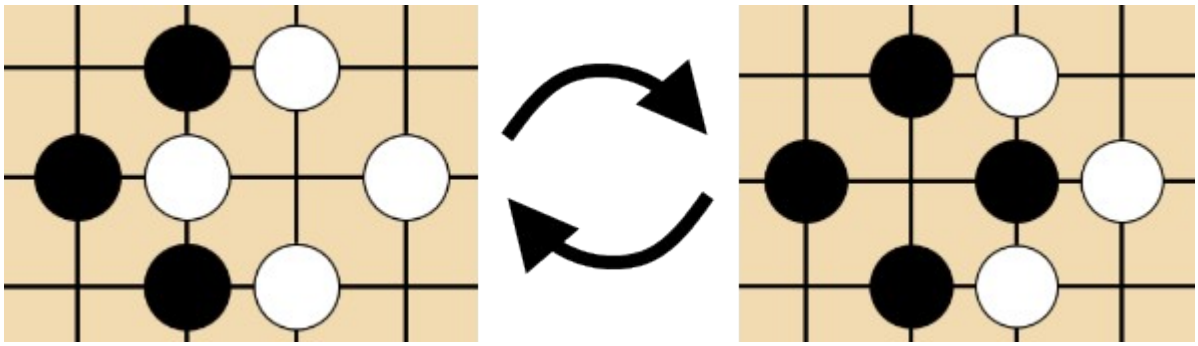
In figure 1 the move is made immediately but in figure 2 black doesn't play the move after placing another stone somewhere else. However, the position at X is still as valuable in both cases (in that it challenges the black stone as opposed to other places on the board that do not). Thus, during propagation, the program finds nodes in other parts of the tree and manipulates their values too, depending on the score of the recently simulated move. If the UCT algorithm is implemented alongside RAVE then the count of wins and number of times passed through a node will be changed. This rapidly gathers more information about moves thus increasing the computer's ability to find optimal moves.

Implementation and analysis:

After thoroughly researching the algorithms mentioned in the last section, I began to construct my own computer Go program that would use these techniques to play Go. This section will summarize my experience of creating this program, specifically in implementing the aforementioned techniques. After creating a suitable interface, through which a human player could place pieces on a virtual board, I implemented all of the following algorithms in the order they appear below. The user interface also includes the ability for the player to select 9 by 9 or a 19 by 19 board to witness how much better the computer plays on smaller boards (due to the reduced state space). The program also allows the user to move back to undo moves and travel back to points in the game. This allows the player to witness the different decisions the computer could make in situations.

Monte Carlo tree search (MCTS):

The MCTS was the hardest method to implement in my program, as the other two methods simply alter the way in which the MCTS algorithm behaves. This means that in implementing MCTS I first had to design all necessary data structures (classes for game trees and nodes etc.) and then write the code to execute MCTS with these new data structures. Not only this, but whilst implementing the algorithm I realised how much would need to be separately handled for the algorithm to run without error. For example, whilst running simulations, used to evaluated board positions, I ran into the problem of "Ko". Ko is a set of situations (an example of Ko is shown below) in which a particular



set of actions result in repeating board states. This repetition could last indefinitely if the same moves are repeatedly made repeatedly (perhaps possible if these are the only legal moves left for the computer to simulate and hence must be picked by the computer). Whilst the above example could be easily spotted and thus avoided, more complicated forms of Ko exist that are much harder for a computer to recognize. This created a problem in the simulation phase as it could simulate a game indefinitely and thus crash the program. To avoid this, I added a timer that would record how long each simulation would take, if the time of that simulation exceeded 0.25 seconds then it is most likely that this simulation has encountered a state of Ko and thus the simulation is terminated and another simulation is started. This does mean that 0.25 seconds of search time is wasted if Ko is encountered but this is much more favorable than having the program crash due to infinite simulation.

(More information on Ko can be found at "Sensei's Library: Ko", 2014).

Another problem in the simulation phase was in the computer being able to determine when to pass. As stated in the last section, the simulation phase of MCTS is to run until the game is over but since in Go the game only ends when both players consecutively pass, how would the computer know when both players would probably pass? This proved to be a very hard problem to solve and in the end I resigned to having the computer simulate until there are no legal moves remaining. If a

method were developed to determine when both players were likely to consecutively pass then I think simulation time could be saved and more simulations could be run in this saved time (though it is possible that by evaluating whether both players would consecutively pass in every single move during simulation this method would actually use more processing time and power than it saved).

Having implemented this algorithm, my program could select moves with some strategy though performance was very poor. It was apparent that the computer was not very skilled in playing when both I, and a friend who had just learnt the rules of Go, easily beat my program. I believe that performance could have been improved if, during the simulation phase of MCTS, a policy could be implemented which would enable the computer to simulate moves that were not purely random but instead based on some logic. This would more accurately model a possible game from that state and would therefore make the value assigned to that state more accurate (because a game simulated from purely random moves is not very likely to show the strategic advantage of that position). However, I did not think it was possible to implement this as policies used by Go programs during simulation tend to be designed by professional players that have much more knowledge of the game than me.

UCT:

After implementing this algorithm, I noticed a slight improvement in my program's ability though not by much. The program seemed to recognize opportunities to capture stones more frequently it would often start a game by trying to capture the 4 intersections in the corners of the board (as to capture this positions require only 2 stones). I think the performance of this algorithm could be improved by implementation of a policy to make board state evaluation more accurate. This is because the UCT algorithm can only help with the balance of exploitation and exploration of nodes if those nodes are accurately evaluated. Another noteworthy milestone in implementation of this algorithm is the selection of the constant for C_p. My program uses the constant $\sqrt{2}$. I came to use this constant as many papers on the topic of UCT recommend use of the constant $\sqrt{2}$. Also, during a visit to the Computer Science department of the University of York I was able to talk to a final year student (Matt Bedder) who had researched Monte Carlo Tree Search applied to a multitude of games in depth for his final year project. He recommended that I initially use the constant $\sqrt{2}$ and then use a technique known as a "parameter sweep" which would gradually optimize the constant as the game progressed. However, the parameter sweep was far beyond my understanding and so I settled for the constant $\sqrt{2}$.

RAVE:

After implementing this algorithm, I noticed further improvement in my program's ability. My program would often begin to construct a string of stones to gain territory and stop halfway to play at another section on the board. This made me question the nodes that the UCB algorithm chose to explore during simulation of MCTS. However, upon further testing and analyzation of the program I determined that this phenomenon was most likely hindered by inaccurate evaluation because the random simulations that occur during the simulation phase of MCTS rarely result in strategic maneuvers like the creation of groups of strings to capture territory.

Bouzy Influence Maps:

In an attempt to shorten simulation time, I developed functions to generate “Bouzy Influence Maps”. These are maps that use mathematical morphology principles to estimate how much dominance a player has over a certain region of the board (Bouzy, 2003). With these tools I hoped to end simulations sooner (once a large proportion of the board is strongly dominated by one team as opposed to when no possible moves are left) but I was not able to generate maps efficiently enough so this technique had to be abandoned as It resulted in longer simulations times than when simulating until no more moves were available. If I had a better understanding of programming I could fix this problem which would result in my program being capable of simulating more possible moves in less time and thus have a better chance of finding optimal moves during application of MCTS.

Conclusion:

The algorithms I have researched and implemented perform as expected and the only thing limiting their ability is how efficiently they run per my code. Thus throughout my program I have always been mindful of the efficiency of my code though I think it could be made faster if I had a deeper understanding of the game of Go. For example, with this understanding I could develop methods to better guess when a simulation during the MCTS simulation phase should end rather than simulating until there are no possible moves left. Thus saving time and allowing for more simulation.

Given time, a technique I would like to implement into my program is Zobrist hashing (an algorithm used to avoid simulating 2 identical board states independently of each other). Currently my program doesn’t simulate enough moves to warrant implementation of this algorithm, though after enough tweaking Zobrist hashing would significantly improve my program.

Having implemented the algorithms discussed in this essay, I am pleased with the current ability of my program. It is clear that my program shows some skill as it is capable of beating another computer player which makes purely random decisions.

Though, my program is reasonably skilled, considering the current ability of professional programs and the time and resources available to me, it is nowhere near as good as any professional computer Go program. This is apparent as my program can be easily beaten by even an unexperienced human player.

To conclude, I have achieved my target in creating a Go playing computer that, whilst not as competent as any professional program, is well made given the resources and time available to me, and clearly shows decision making ability in regards to the playing of Go.

References:

- Anon contributors, “Sensei’s Library: Rules of Go”, <http://senseis.xmp.net/?RulesOfGo>, 2015.
- Anon contributors, “Sensei’s Library: Ko”, <http://senseis.xmp.net/?Ko>, 2014.
- Bouzy, B., 2003. Mathematical morphology applied to computer go. International Journal of Pattern Recognition and Artificial Intelligence, 17(02), pp.257-268. Browne, C.B., Powley, E., Whitehouse, D., Lucas, S.M., Cowling, P.I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S. and Colton, S., 2012. A survey of monte carlo tree search methods. Computational Intelligence and AI in Games, IEEE Transactions on, 4(1), pp.1-43.
- Coulom, R., 2006. Efficient selectivity and backup operators in Monte-Carlo tree search. In Computers and games (pp. 72-83). Springer Berlin Heidelberg.
- Chaslot, G., Bakkes, S., Szita, I. and Spronck, P., 2008, October. Monte-Carlo Tree Search: A New Framework for Game AI. In *AIIDE*.
- Rimmel, A., Teytaud, O., Lee, C.S., Yen, S.J., Wang, M.H. and Tsai, S.R., 2010. Current frontiers in computer Go. Computational Intelligence and AI in Games, IEEE Transactions on, 2(4), pp.229-238.
- Gelly, S., Kocsis, L., Schoenauer, M., Sebag, M., Silver, D., Szepesvári, C. and Teytaud, O., 2012. The grand challenge of computer Go: Monte Carlo tree search and extensions. Communications of the ACM, 55(3), pp.106-113.

Bibliography:

- Anon contributors, *“Sensei’s Library: KSG Bot Ratings”* <http://senseis.xmp.net/?KGSBotRatings#toc1>, *2016*.
- *Bradberry*, *“Introduction to Monte Carlo Tree Search”*, *<http://jeffbradberry.com/posts/2015/09/intro-to-monte-carlo-tree-search/>*, *2015*.
- Li, et al., “Bouzy's 5/21 algorithm” <http://gnu.gds.tuwien.ac.at/software/gnugo/gnugo_14.html>, 2004.