Monte-Carlo Based Computer Player For the Game of Go

Student Name: A. Go

Supervisor Name: S. Dantchev

Submitted as part of the degree of Bsc Natural Sciences to the

Board of Examiners in the School of Engineering and Computing Sciences, Durham University.

Abstract –

Context/Background
The game of Go has been one of the long-standing grand challenges in artificial intelligence. Due to its extremely large branching factor and state space combined with the difficulty of designing good static evaluation functions, traditional brute force methods were inadequate for the purposes of creating strong Go playing AIs. Therefore, new techniques such as Monte-Carlo Tree Search needed to be developed before significant progress could be made. More recently, the use of neural networks and deep learning have led to large jumps in the playing strength of computers as demonstrated by programs such as AlphaGo.

Aims
This project aimed to implement an agent that plays Go using Monte-Carlo Tree Search. Different versions of this agent were implemented by extending the basic agent using different types of heuristics. The aim was then to evaluate and compare the different agents in order to see which heuristics were able to provide the most benefit.

Method
The different agents were all be extensions of the same base MCTS player. After tuning any necessary parameters, the agents competed against each other in a round-robin tournament. The best version of the agent also played several games against human opponents to try and produce an estimate of its absolute strength.

Results
We found that using the all-moves-as-first heuristic to improve the early accuracy when estimating the value of moves was the most effective way to extend the MCTS algorithm. In addition, Go-specific heuristics also provided value both in biasing the search and in improving the quality of the random simulations. A fully-connected single-layer neural network was found to be insufficient for being able to improve the computer's play.

Conclusion
We show that the MCTS algorithm can be extended using a variety of techniques in order to improve its performance at the game of Go, with success coming from both domain-dependent and domain-independent approaches. When all the techniques were combined, the resulting computer player was able to consistently win games against a 9kyu human player on a 9x9 board when taking a four stone handicap.

Keywords – AI, Go, Monte-Carlo, heuristics, neural networks

I. INTRODUCTION

The simplicity of the rules of Go combined with the long-standing inability of computer programs to beat the best human players has made Go one of the grand challenges of artificial intelligence, and one that stood until very recently (Silver et al., 2016). This introduction provides a brief summary of the rules of Go and describes the reasons why it is such a difficult game for computers to master. We then outline the aims and deliverables for this project.

Rules of Go

Go is an ancient, perfect information, abstract strategy game, in which players alternate turns placing a stone on an empty intersection of a 19x19 grid, with the ultimate goal being to surround as much territory (empty intersections) as possible. If a stone is adjacent to another stone of the same colour, they form a single block. The empty points adjacent to the block are the block's liberties, and if all of these become filled by enemy stones, the block of stones is captured and removed from the board. It is not allowed for a player to place a stone such that any of their own groups run out of liberties (i.e. suicidal moves are forbidden). The game ends when both players decide there are no more advantageous moves left to be played, and therefore both pass. There is one additional rule (the 'ko' rule) which prevents infinitely repeating board positions from recurring by forbidding moves which would return the board to the exact same state as it was the move previously.

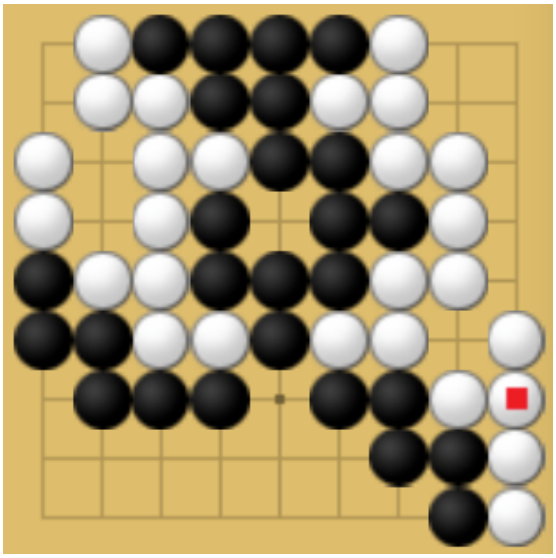

Figure 1. A completed game of Go on the smaller 9x9 board. Under Chinese rules, each player scores a point for each of their stones on the board, as well as for any of the empty intersections in the territory that they have surrounded. In this example, black has won by a score of 42-39. White will often receive 'komi' worth around 6 points as compensation for playing second.

An important concept in Go is that of 'life and death', which refers to the question of whether a given group can eventually be captured (in which case it is dead) or if the group will be able to survive until the end of the game. The feature of groups that distinguishes

between these two categories is whether or not the group can make at least two 'eyes' or not. An eye is an empty intersection inside a group which the opposing player will never be able to play on unless that point is the group's last remaining liberty, in which case playing there would capture the whole group. Due to the suicide rule, if a group has two eyes, the opponent will never be able to remove these liberties and so the group is 'immortal' and will live forever (barring the circumstance where a player fills in one of their own eyes, nearly always a huge blunder).

Go has a natural ranking system where the difference in two players' ranks corresponds to the number of handicap stones (stones placed on the board before the start of the game) that are required for the game to be even. Beginners start somewhere in the 30-kyu range, with their rank decreasing as they get stronger until they reach 1-kyu. After the kyu levels come the dan levels, going up from 1-dan to 9-dan. (Professionals are also ranked on a separate dan system. An amateur 9-dan will have similar strength to a professional 1-dan).

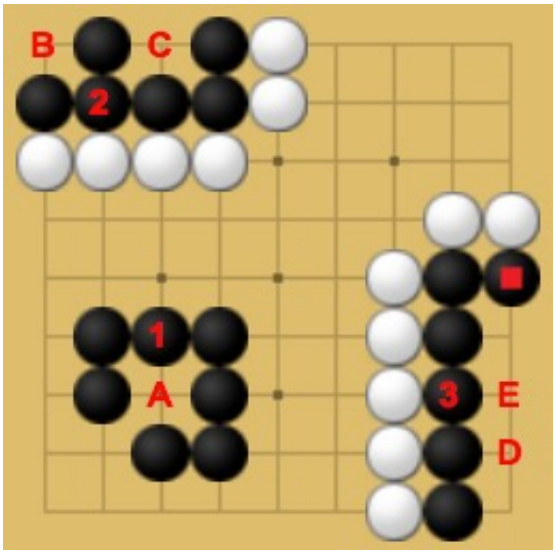

Figure 2. Examples of eyes and life and death.
The black group marked 1 currently has only one eye at A. White cannot play at A immediately (since that would be suicide) but can play there to capture the group once all of the outside liberties have been taken away.
The group marked 2 has two eyes and is therefore immortal. White will never be able to play at B or C, and so the group can never be captured.
The group marked 3 is also considered alive even though it doesn't yet have two clear eyes. E and D are considered *miai*; if white plays at one of the points, then black can play the other one to make two eyes. Therefore, black has no need to make an additional move here unless white plays there first.

The Difficulty of Go for Computers

At the time that Deep Blue achieved its first victory over Kasparov, computer Go was still very much in its infancy (Funke, 2012). In 1998, a strong amateur player managed to defeat one of the strongest programs arounds despite it taking a 29 stone handicap (Muller, 2002), which would have put the strength of Go computers at the weak amateur level.

There are a couple of reasons why the techniques that allowed computers to conquer

chess, namely extremely large tree-searches aided by tools such as alpha-beta pruning, failed to crossover into the world of Go. Firstly, the state-space for Go is several orders of magnitude larger than that of chess. On a 19x19 board where almost every empty intersection is a legal move, Go has an average branching factor of around 200, compared to roughly 25 for chess. The game tree for Go is also deeper, with games that go to completion taking more than 200 moves, while the number for chess is closer to 80.

The other major factor is the lack of good heuristic functions for evaluating the state of incomplete games. While material and control of the center is easy to measure in chess and already provides a good measure of who is ahead, there is no easy way to examine such characteristics on an unfinished Go position without being able to read out the life and death statuses of different groups or having a good sense of global tactics.

The combination of these two reasons explain why exhaustive tree search failed to lead to the same breakthroughs for Go as they did for chess, which is why the best Go programs of the 20th century were more centered around the idea of expert knowledge. Monte Carlo Tree Search only came to prominence around the late 1990s and has since been the dominant paradigm in computer Go. The insight underlying MCTS is that moves can be evaluated by simulating many random games from a given position and looking at the average win rate. Since playing games using only random moves can be done very quickly and the score of finished games can also be calculated easily, many thousands of games can be simulated and the resulting information will be more useful than attempting to statically analyse different positions.

Aims and Deliverables

The aim of this project is to build a Go-playing agent which uses Monte-Carlo Tree Search, and to then use this agent to test various heuristics and extensions to this framework to see which ones would provide the most benefit. All of the evaluation is to be done on the smaller 9x9 sized boards, since the much smaller state-space allows for much faster running of experiments and a better chance of the computer agent being able to play well. The deliverables for this project were split into three stages:

Basic- Implement a Go-playing agent which relies on MCTS. The agent should be able to use this algorithm with a set number of simulations for each turn and be able to play at around the level of a human beginner on a 9x9 sized board.

Intermediate- Improve the MCTS agent by implementing a variety of heuristics, including Go-specific as well as domain-independent ones. Investigate which method of utilising heuristics and which parameter settings are most effective, and test and evaluate the heuristics by having the different versions of the agent compete in a tournament in order to see which ones provide the most benefit.

Advanced- Implement a neural network to be trained through self-play, and evaluate the effectiveness of the network when used as another type of heuristic to aid the basic MCTS agent.

II. RELATED WORK

The first example of a program that made use of statistical approaches to play Go was Gobble, developed by Bernd Brugmann, in which the results from random playouts were combined with simulated annealing methods to create a program that could play better than a human beginner without any explicitly encoded knowledge of Go (Brugmann, 1993). The key idea which Brugmann introduced is that since a random game of Go can be simulated very quickly and since the score at the end can be easily calculated, static evaluation functions can be replaced by a large number of random playouts, the average result of which can give a good idea of the quality of a move.

However, it wasn't until the use of UCT, Upper Confidence Bounds Applied to Trees, was introduced in the Monte-Carlo Tree Search algorithm that Monte-Carlo methods became the dominant paradigm in computer Go (Kocsis & Szepesvari, 2006). UCT is based upon the UCB1 algorithm (Upper Confidence Bounds 1) which was designed to solve the 'exploration-exploitation' dilemma in the Multi-Armed Bandit Problem. In this scenario, the player has a selection of different levers in front of them, each of which pays out a reward according to some unknown distribution. A winning strategy in this game is one which is able to balance 'exploitation', pulling the lever with the highest expected reward found so far to maximise profit, and 'exploration', investigating other levers to try and find one with an even higher expected payout. UCB1 does this by always selecting the action with the highest upper-confidence bound, which is the sum of two terms:

$$UCB1 = \frac{w_i}{n_i} + k\sqrt{2\frac{\ln(N)}{n_i}} \tag{1}$$

where w_i is the total reward from lever i thus far, N is the total number of actions taken, n_i is the number of times lever i has been pulled, and k is some empirically chosen positive constant. (Note that for the UCB1 value of an action to be well defined, it must have already been tried at least once. Therefore, before comparing the UCB1 values over a set of actions, each action must be 'initialised' with a single try.) The first term, w_i / n_i, is the expected payout based on the observed outcomes so far, whilst the second term becomes larger when lever i is tried less often relative to the other available actions. When applied to Go, this formula allows the search to be weighted towards moves which seem promising without being too greedy and risking under-valuing good moves due to early unlucky playouts.

A lot of the focus that has gone into improving MCTS programs has centered around developing heuristics that incorporate pieces of Go knowledge which can guide the search to make it more accurate or efficient. There are two distinct places where the use of heuristics has been explored, for example by Peter Drake and Steve Uurtamo (Drake & Uurtamo, 2007). The first is to bias the program so that moves which are deemed good by the heuristics are explored earlier and given more consideration. The second is in the use of 'heavy playouts', which alter the default policy during the simulation step in the hopes that more realistic games are played which result in higher quality information from each playout. Examples of simple heuristics that incorporate Go knowledge include playing moves that increase/decrease the number of liberties on particular groups which are involved in a fight, playing moves that capture enemy stones if possible, or playing moves that save your stones if they are in atari (one move away from being captured).

A popular extension to the MCTS framework is that of Rapid Action Value Estimation (RAVE), which is based upon the 'All Moves As First' (AMAF) heuristic (Gelly & Silver, 2011). AMAF is not a Go-specific heuristic, but it does rely on the property that the value of a move is somewhat independent of the time at which it is played. That is to say, a move which is a good move now, would still be a good move if it were played later in the game. Using the AMAF scores allows estimates about the quality of moves to be made more quickly, even though the scores would be less accurate than the actual MCTS values in the long run. RAVE provides a method of combining the MCTS and AMAF scores so that the AMAF scores are given more weighting early on, and then less weighting later on in the search when more information is available.

Recently there has been renewed interest in the use of neural networks to help create strong Go programs using 'deep learning' methods. The combination of neural networks trained with self-play and temporal difference learning is notable for having produced the first expert level backgammon player, capable of beating the best human players in the world using only a two-ply look-ahead (Tesauro, 1994). Previously, small neural networks using just a single hidden layer had been used in Go to try and evaluate positions by estimating final territory, although this resulted in only a very weak agent when used to play games directly (Schraudolph et al., 1994).

More complicated neural networks with up to twelve hidden layers have also been used to predict expert moves after being trained on expert game records, and these have been improving in accuracy (Maddison et al., 2015). Combining the results of a neural network that suggests moves in this way with MCTS is a current promising approach to improving state-of-the-art Go programs (Tian & Zhu, 2015).

The current leading Go program, AlphaGo, is reliant on such deep convolutional neural networks. The strength of AlphaGo comes from having two neural networks, one doing move prediction and one doing evaluation, trained initially through supervised learning using large databases of high-level amateur games, and then through reinforcement learning through self-play. By themselves, the neural nets were able to defeat the state-of-the-art Monte-Carlo programs. When combined with MCTS, AlphaGo was able to win five out of five games against the European champion, the first time a computer has beaten a human professional without taking a handicap (Silver et al., 2016). AlphaGo went on to defeat one of the top Korean professionals, Lee Sedol, 4-1 in a subsequent match.

III. SOLUTION

This section describes the design and implementation of the solution, and discusses the algorithms used in detail.

Architecture

The programming language chosen for this project was C++. There were two main reasons for this, the first being that speed was an important consideration. In the Monte-Carlo Tree Search algorithm, greater speed translates directly into being able to simulate more games in a given amount of time, which is one of the most important factors affecting the effectiveness of the algorithm. The second reason is that since the aim of the project is to extend an MCTS agent using a variety of heuristics, an object-oriented programming language which allows inheritance is highly desirable as this would allow all of the various agents to be based on the

same basic MCTS program.

Each player was designed as a class which would accept as an input a particular board position and would return the best move it could find. The project was structured so that the first task was to implement a player which would just run MCTS. Once this had been done, the different heuristics and extensions could be developed which could be selectively added to the base MCTS player by creating new subclasses incorporating the new features as required.

Monte-Carlo Tree Search / UCT

For the reasons described earlier, it is impossible to build out a complete game-tree for Go to any meaningful depth. Monte-Carlo Tree Search therefore uses a partial game tree which grows with each simulation that it plays out. Each iteration of the MCTS algorithm has four steps:

1) *Selection*: Starting at the root of the tree (which represents the current game state), the algorithm repeatedly selects a child node using its 'tree policy' (the simplest of which is to select the child with the highest winning percentage) until it reaches a leaf node. Since Go is an adversarial game, this is a minimax search as the tree policy acts from the point of view of the player about to play.

2) *Expansion*: Once a leaf node has been found, the tree is expanded by adding a child node for one of the possible successor game states.

3) *Simulation*: From this new game state, the 'default policy' is used to simulate the rest of the game, with the simplest policy being to just play random moves until a terminal state is reached. Only the eventual outcome of the simulation is important; the specific moves that were played can do not need to be remembered.

4) *Backpropagation*: The outcome of the simulation is used to update the information in each of the nodes that were visited during step 1.

These steps are then repeated as many times as possible in the time allowed. The child node which has been passed through the most number of times is then selected as the actual move to be played.

Even though the margin of victory is easy to calculate at the end of a game of Go, it is more robust to use a binary win/loss metric instead. This leads the agent to naturally carry out desired behaviours such as playing more conservatively when ahead and aggressively when behind, at the expense of making some moves that might seem unnatural to human players such as not caring about playing a move that loses a few points because it doesn't affect whether the computer will eventually win or not.

Growing the tree dynamically in this way allows the algorithm to spend as much of its time as possible searching along the paths that look the most promising and to not waste time exploring bad moves. However, a tree policy that is too greedy during step 1 can easily lead to good moves being ignored due to some unlucky playouts early on in the exploration. This is why the tree policy used by MCTS programs during the selection step is based on UCT, as described in Section II.

An implementation of UCT requires almost no tactical knowledge of how to play Go.

Only the rules must be known to ensure that the random playouts are all legal. The one exception is that some knowledge about eyes is usually encoded, as moves that fill in a player's own eyes are often suicidal with no tactical benefit and more importantly, may prevent a simulation from halting in reasonable time.

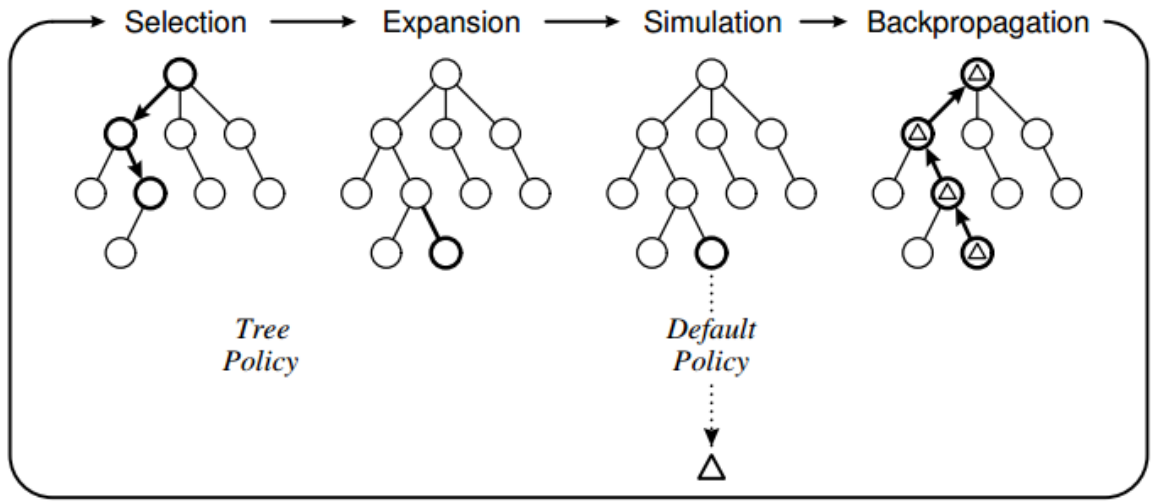

Figure 3. The steps of the MCTS algorithm.

Heuristics

Rapid Action Value Estimation / All-Moves-As-First

The all-moves-as-first heuristic allows more information to be extracted from each simulation by allowing related nodes in the tree to share information (Gelly & Silver, 2011). It is not a Go-specific heuristic, but it does rely on a particular property present in Go, namely that the value of a move is somewhat independent of the time at which it is played. That is to say, a move which is a good move now, would still be a good move if it were played later in the game. The greater the extent to which this property holds, the greater value the heuristic will provide.

Instead of each node in the tree keeping track of only regular wins and visits as in UCT, each node will also keep count of its AMAF wins and visits. When using AMAF, all the moves played out during the simulation phase need to be remembered. When it then comes to updating the tree, the algorithm not only updates the nodes that were visited, but also increments the AMAF scores for any sibling nodes which represent moves that were played at some point further on in that game.

The AMAF values for each node can therefore be thought of as counting how many times that move has been played by that player *at any future point* during the simulations. and how many of those times that move has been part of a series of moves that led to a win. Since around half the points on the board are played by a given player during every game, nearly half the nodes will have their AMAF scores updated after every playout, so a lot more information is extracted per simulation than would be through regular UCT.

The RAVE (Rapid Action Value Estimation) algorithm is used to combine the AMAF values with the regular Monte-Carlo scores. If the Monte-Carlo score is M, and the AMAF score is A, then the RAVE score (R) for each node is given by the weighted sum:

$$R = (1-\beta)M + \beta A \tag{2}$$

When the total number of simulations run is still small, then β should be close to 1 so that most of the weighting is given to the AMAF value. As more simulations are run, we want the more accurate Monte-Carlo value to take over and so β should decrease.

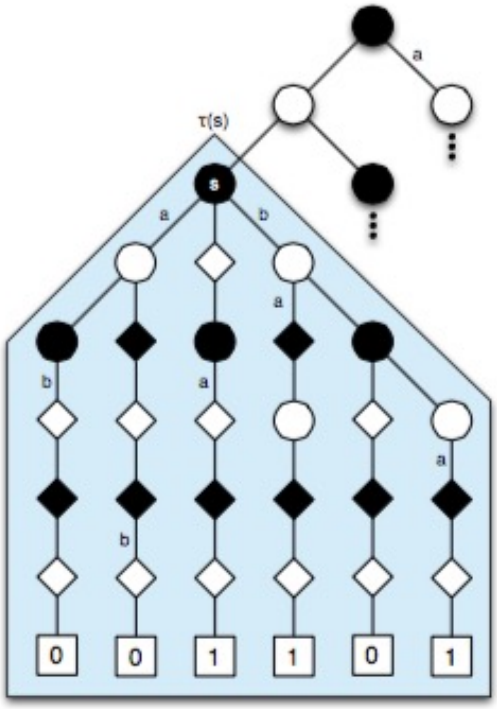

Figure 4. The All-Moves-As-First heuristic (image taken from Gelly & Silver, 2011). In this example, black is looking to examine the value of the moves *a* and *b* from state *s*. The simulations in which *a* is played directly have resulted in two losses, whereas playing *b* directly has resulted in one win and one loss. However, looking at all the simulations in which *a* has been played at any point gives three wins and two losses, whereas *b* has only achieved two wins and three losses. Therefore, the AMAF heuristic would prefer *a* to *b* at this stage of the algorithm. (The difference between circles and diamonds can be ignored for the purposes of this diagram.)

The cooling schedule we use takes the form:

$$\beta=\left(\frac{k}{3N+k}\right)^{\frac{1}{2}} \tag{3}$$

This cooling schedule is parameterised by an equivalence parameter k, which is the number of simulations that need to be run for the Monte-Carlo and the AMAF scores to be given equal weighting, and can be chosen empirically.

As before with UCT, we can also introduce a variance term into the RAVE algorithm to encourage more exploration:

$$UCT\text{-}RAVE = (1\text{-}\beta)\cdot M + \beta\cdot A + k\sqrt{2\frac{\ln(N)}{N_i}} \quad (4)$$

Go-Specific Heuristics

There are two places in an MCTS algorithm where domain specific heuristics can be employed (Drake & Uurtamo, 2007). The first is to bias the program so that moves which are deemed good by the heuristic are explored earlier and given more consideration. The second is in the use of 'heavy playouts', which alter the default policy during the simulation step in the hopes that more realistic games are played which result in higher quality information from each playout.

Move-ordering takes place at the expansion step of the MCTS algorithm. When a new child node is created, the heuristics may give some prior idea of whether or not the move is likely to be a good one and whether it is worth exploring early. This can be done by artificially adjusting the count of the wins and visits to that node. For example, if a given move is thought to be an especially good one, then instead of initialising the node with a count of zero wins and zero visits, it can be given a value of say 150 wins over 200 visits. The value that the visits count is set to should correspond to the confidence in the accuracy of the heuristic, so that a heuristic which is known to be better has a greater and longer-lasting influence in guiding the search.

In the regular MCTS algorithm, the simulation step runs by applying the default policy of just playing random moves until the game ends. This works because even though the random games are poorly played and unrealistic, averaging the outcome over enough simulations still produces useful information. A natural way to proceed would be to try to improve the quality of these simulations by guiding the playouts using some simple heuristics, in the hope that the better quality information emerging from each playout will outweigh the cost of the extra time that these simulations would take.

Another danger in using heavy playouts aside from increased time, is that you may end up with a bias in the sample of games which you simulate, which could lead to the information being less accurate than using regular MCTS. Indeed it has been found that stronger default policies (policies that win more often when applied directly to choose a move) can result in weaker agents when used for the purpose of heavy playouts (Gelly & Silver, 2007).

This project will run heavy playouts using heuristics using a *best-of-n* method. Instead of selecting just a single random move to play at each stage, the new policy will select *n* moves, compute the heuristic values for each of them, and play the move with the highest value. This method keeps down the time that needs to be spent doing heuristic calculations, and keeps the playout from being completely dominated by the heuristics. The exact value of *n* can be empirically chosen.

The domain-specific heuristics which were implemented and tested in this project were:

- Proximity heuristic - Favour moves in the local area of the last move which was played, since a new enemy stone in the area is likely to change the situation and often requires an urgent response.

- Avoid-the-first-two-lines heuristic - Moves that are played near the edge of the board

are often ineffective at exerting influence over the rest of the board or at taking large amounts of territory. Furthermore, they tend to be more vulnerable to attack, and so are often bad moves, especially in the opening.

- Capture heuristic - If a stone or a group of stones can be captured, then capture them. Although not all captures are good moves, they often provide tactical benefit and can frequently be 'urgent' moves since not capturing now could allow the opponent's stones to escape.

Neural Network

For this project, a simple feed-forward neural network with a single hidden layer was implemented and trained to predict the eventual winner of the game. These predictions could then be used to evaluate positions and guide the MCTS algorithm in the same way as other heuristics would.

Other networks have included higher-level features of the game as inputs to the network, such as the liberties on each group of stones, distance from the edge of the board, or distance from the last move that was played. All of this would be extra tactical information which the network could exploit without having to calculate the intermediate values. However, to keep the network as fast and as simple as possible, this project only only used the raw game-state as the network input. Specifically, there were 81 input vectors, one for each point on the board, each with three binary channels. The channels indicated whether a point was empty, occupied by white, or occupied by black. There was one final input indicating whose turn it was to play next. The whole network was fully connected, with the hidden layer consisting of 40 neurons, and a single output neuron returning a value between 0 and 1, indicating the likelihood that black would go on to win the game. Throughout the network, the logistic function was used to smooth the activation of each neuron to return an output between 0 and 1:

$$\varphi(x) = \frac{1}{1+e^{-k\Sigma w_i x_i}} \tag{5}$$

where k is some positive constant.

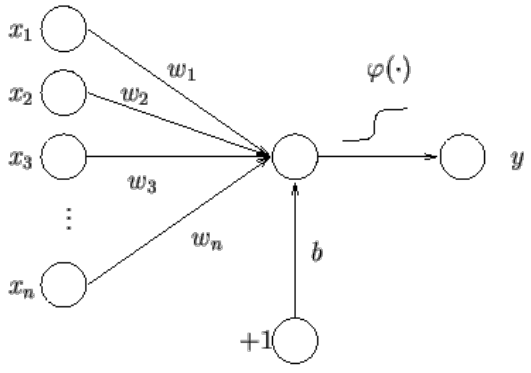

Figure 5. Diagram of a single neuron within the network. The activation level is calculated as the weighted sum of the inputs and the bias, which is then scaled to a value between 0 and 1 using the sigmoid function.

The neural network was trained through self-play and a method called Temporal Difference learning (TD), a standard method of reinforcement learning which was used to train the first expert level backgammon computer using only self-play (Tesauro, 1994). The TD algorithm works for tasks in which the aim is to predict the end result in a sequence. Instead of only performing updates when the final result is known, the TD algorithm updates its old predictions after each timestep to bring them more into line with its latest predictions. During the training period when the neural network is playing against itself, for each move the network will do a 1-ply look-ahead and evaluate all the possible successor positions. If the network were to just play the best move it found, then it would end up playing very similar games each time and so the training would be very slow. Therefore the moves are chosen stochastically with better moves having a higher probability of being chosen according to:

$$p_i = \frac{e^{k \cdot s_i}}{\Sigma e^{k \cdot s_j}} \tag{6}$$

where s_i is the score for move i and k is some positive constant. Higher values of k make it more likely that the moves with higher evaluations are chosen. The evaluation of the move which is eventually chosen is then used as the target value for the initial game-state, and the weights are updated accordingly.

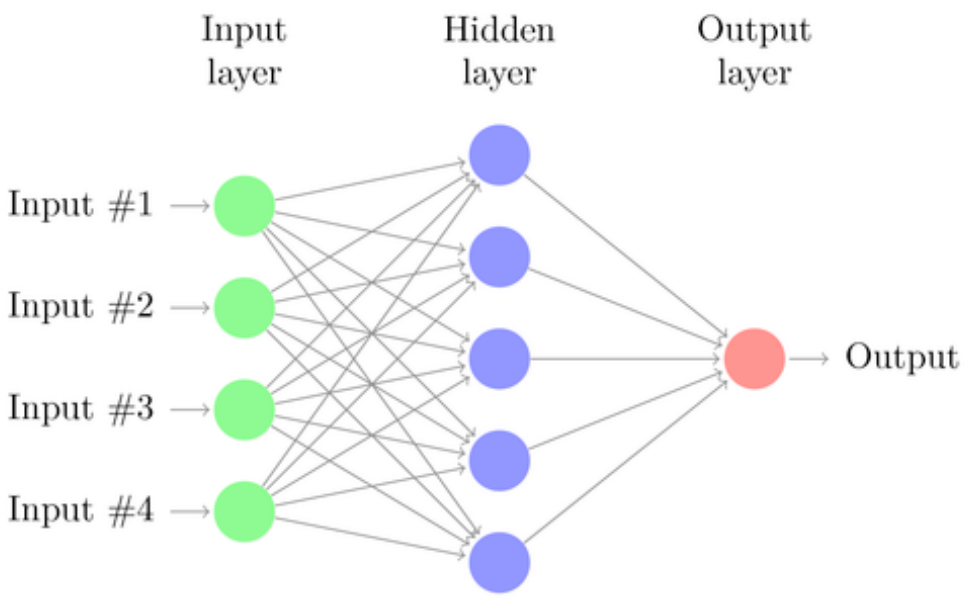

Figure 6. Example of a neural network with a single hidden layer. For this project, the network takes 82 inputs (specifying the configuration of a 9x9 board, as well as whose turn it is to play), and was trained to predict the eventual winner at the end of the game. This information could then be used as another heuristic to help evaluate different board positions.

The update is performed using standard backpropagation methods, where the change in each weight is equal to the learning rate multiplied by the partial derivative of the squared error with respect to each weight:

$$w' = w + \eta \frac{\delta(\text{target-output})^2}{\delta w} \tag{7}$$

IV. RESULTS

This section describes the results of the various experiments which were run. We first discuss the experimental settings and how the parameters were tuned to make the algorithms as strong as possible, and then go on to discuss the results of the different agents playing against each other. We conclude by looking at a match played between the strongest agent and a human player.

Tuning

Before any testing of the could be done, there were various parameters in the algorithms that needed to be tuned in order to optimise performance and determine the experimental settings. Two of the main parameters that required tuning were the equivalence parameter in the RAVE algorithm in (3), and the level of confidence in the heuristics.

Figure 7 shows that best performance with RAVE is achieved using an equivalence parameter of at least several hundred, indicating that relying more on the AMAF estimate for the first several hundred simulations produces better results than only using the regular Monte-Carlo results. We found that increasing the value of the equivalence parameter much beyond this point had little further impact. Figure 8 shows the performance for different levels of confidence in the heuristics. Although it is clear that using heuristics for the purposes of move-ordering significantly improves performance, it can also be seen that if the heuristics are given too much weighting, then MCTS will not be able to correct for any mistakes in the heuristic values and so performance declines.

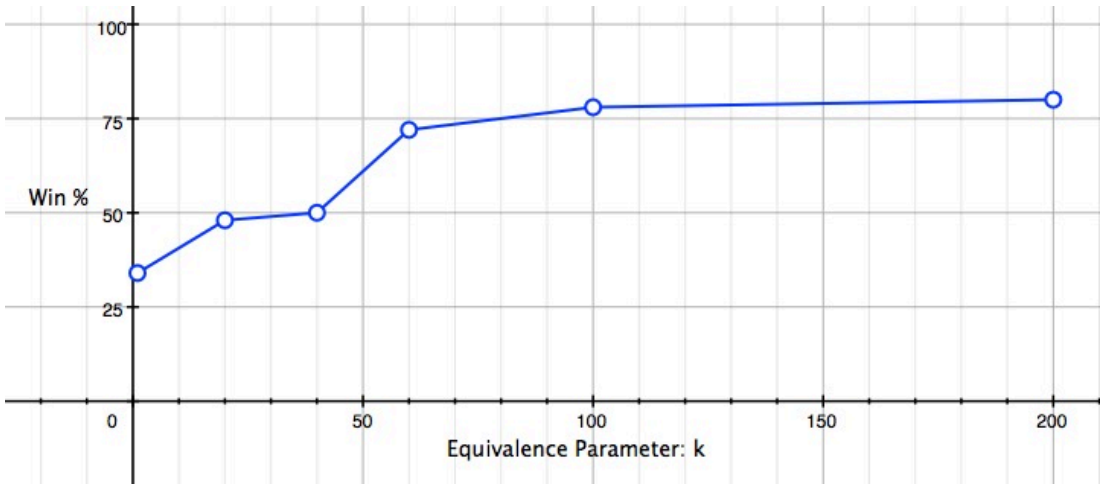

Figure 7. Graph showing the win percentage using the RAVE heuristic for different values of the equivalence parameter, k. The RAVE player used 2,000 simulations per turn against a regular MCTS opponent running 3,000 simulations per turn. Fifty games were played at each setting.

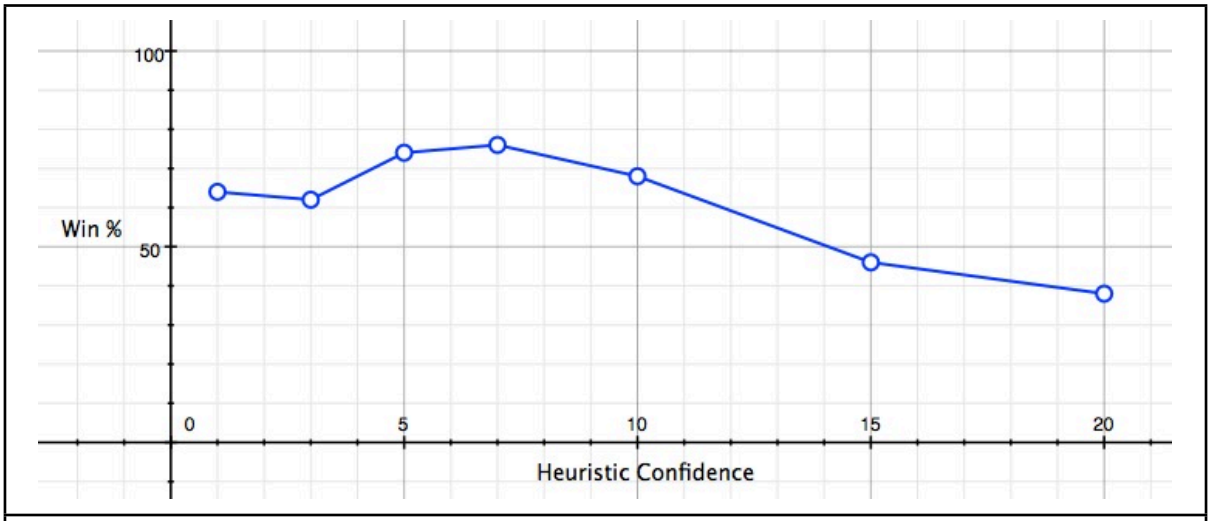

Figure 8. Graph showing the win percentage when using heuristics to do move-ordering for different values of heuristic confidence when playing against a regular MCTS opponent. Each player used 3000 simulations per turn, with fifty games played at each setting.

The values of the equivalence parameter and heuristic confidence to be used in the rest of the experiments were taken to be the best performing values in the tuning runs. Using too high a value for n in the *best-of-n* heavy playouts resulted in simulations taking an unreasonable length of time so a value of five was used.

The performance of the neural network as it was being trained is shown in figure 9, where the results of the network playing directly against a completely random player are displayed after repeated rounds of training through self-play. We found that not having a high enough value for k in (6) resulted in the network reaching a plateau earlier and being unable to improve its win rate. We used a value of 20 which was found to work reasonably well.

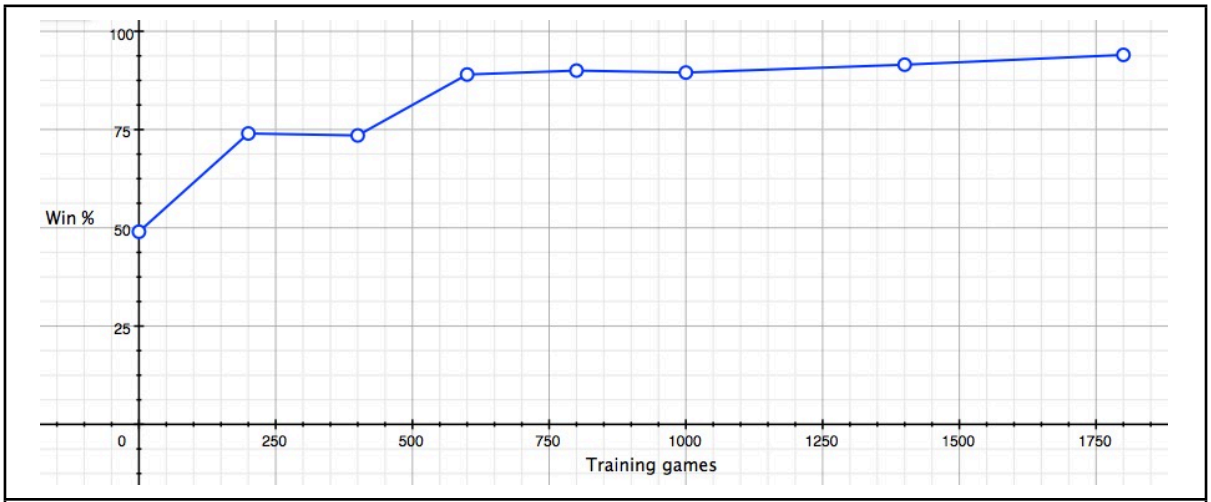

Figure 9. Graph showing the win percentage of the neural network when playing against a random opponent after increasing amounts of training. 200 games were played after each training stage.

Tournament Results

To evaluate the relative strength of each heuristic and determine which provided the most benefit to overall performance, we conducted a round-robin tournament using five different versions of the basic player (all running at 3,000 simulations per turn):

1) The basic MCTS algorithm on its own.
2) Using the RAVE extension ($k = 200$).
3) Using heuristics to do move-ordering (*heuristic_confidence* = *7*).
4) Using heavy playouts ($n = 5$).
5) Using the neural network as a heuristic for move-ordering.

The results of the tournament are shown in Table 1.

Win % for column player	MCTS	RAVE	Move-ordering	Heavy playouts	Neural Net
MCTS		94	74	80	52
RAVE	6		28	36	12
Move-ordering	26	72		52	30
Heavy playouts	20	64	48		22
Neural Net	48	88	70	78	

Table 1. Results of the round-robin tournament between the different versions of the MCTS agent. Numbers show the win percentage for the column player after a 50 game match.

Games Against Human Players

The last step of experiments was to play the strongest possible version of the program against human players. This allowed us to get a feel for the absolute strength of the program and to give it an approximate rank. We used 30,000 simulations per turn for these games, and made use of all possible heuristics. As the program was still significantly weaker than the human opponents, handicap stones were given to the computer. With a four stone handicap, the computer was able to win almost all games against the author (a 9kyu player). With only three stones, the computer would lose most matches as it struggled to deal with the increased choice during the opening, and would subsequently not have large enough lead to try to hold on to going into the endgame. One of the four stone games which the computer won is shown in Figures 10-11 below.

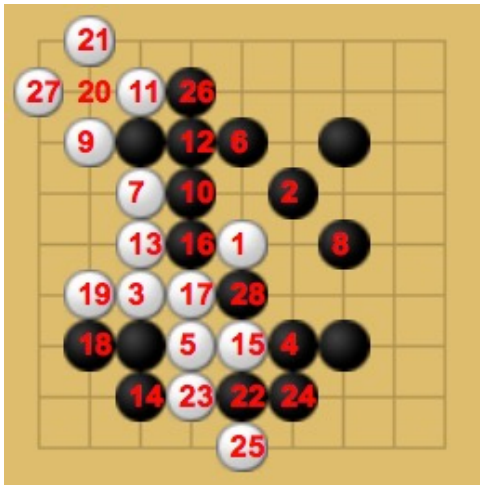

Figure 10. The opening moves of a game played between the computer and the author. Black plays solidly up to this point, sacrificing the stones in the lower left. White 25 was a mistake which Black would later punish. Black 28 is a bad move as White can capture straight away. Black has maintained a solid lead up to this point.

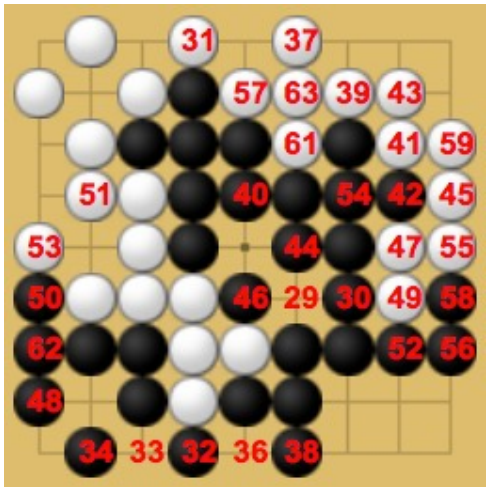

Figure 11. Moves 29-63. 35 at 50, 60 at 63. Black 32 was a very good move which brought the stones in the lower left back to life. Black 40 was a mistake. Blocking at 41 or 43 would have been much better and preserved Black's large lead. However, the game was still good for Black and there was little chance for White to catch up. Black's pointless sacrifice at 60 is an example of the computer freely giving up points if it does not affect the final outcome. Black wins the game by 2 points.

V. Evaluation

Outcome

Making use of the AMAF heuristic through the RAVE algorithm proved to be the most effective extension of the MCTS agent, defeating all other opponents in the tournament and producing the most number of wins against the basic MCTS opponent. The agent using heavy playouts was the next most successful in the tournament, although this observation should be balanced by the fact that this program was considerably slower than any of the others.

Nevertheless, the results show that an efficient implementation of this procedure would be a good way to improve performance, and that the use of heuristics is more valuable in the simulation of stage of MCTS than it is for doing move-ordering.

Even after large amounts of training time and being shown that it was much better than random, the neural net failed to provide much value for use as an extra heuristic, accumulating just eight more wins that the basic MCTS program over 250 games played, within the realm of statistical error. Although it is possible that more training would have yielded better results, it seems likely that the simplicity of the network's architecture limited the strength of the information that it could provide.

The computer player was able to win against the author when playing with a four-stone handicap, indicating play at the level of a human beginner (around 22 kyu). Despite playing some obviously bad moves, as well as moves which would likely not have been played by any human, the computer played well enough to maintain its lead and win the game comfortably.

Limitations

The overall system had several limitations affecting the strength of the findings. The first is that since the program was optimised mainly for simplicity instead of speed and there was only a limited time to run experiments, the number of simulations per turn had to be capped at the low thousands in order for the games to complete in a reasonable amount of time. Since the accuracy of the Monte-Carlo scores is directly linked to the number of playouts completed, being able to scale this number would likely have produced a much stronger player, and so this part of the design could have possibly warranted more attention in the early stages of development. This is also true for the implementation of heavy playouts, which despite demonstrating that it could improve performance significantly, ran very slowly in comparison with the other heuristics. This means that the value of n had to be kept small in the *best-of-n* implementation, so we were unable to see whether higher values of n (presumably corresponding to higher quality playouts) would have resulted in even larger benefits. The program's overall lack of speed also limited the number of games that could be played during experiments. Although 50 game matches were enough to reveal clear trends and effects, the small sample size leaves the possibility for a lot of noise and variance. This means that it is very difficult to accurately compare the sizes of various effects, particularly when very similar to one another.

In addition to AMAF and RAVE, this project also looked at a small number of domain specific heuristics. Although these heuristics were partly chosen because of their simplicity to implement, the experiments showed that they did manage to improve overall performance. However, several other heuristics which could potentially have provided greater benefit had to be overlooked due to the lack of time. These include using machine learning to build up a database of patterns to suggest possible good moves in particular local situations. There are also other domain-independent heuristics which have proven to be useful in computer Go, such as the history heuristic which similarly to AMAF, aims to favour moves which produce good results in several branches of the search tree (Schaeffer, 1989). Since this project found RAVE to be the best way to improve performance, heuristics which work along similar principles would be a good candidate area to look for similar improvements.

This project only implemented a neural network with a very simple architecture. Many of the recent strides forward in computer Go have resulted from creating more complex

networks with more hidden layers. Although implementing a network of this kind of scale would have been infeasible for this project, there is the possibility that a network architecture that was more suited to the problem at hand, such as a convolutional neural network, could have been trained more quickly and produced better results than the single-layer network that was used. It is also possible that different training methods might have proved more effective than using just self-play as done in this project. Databases of high-level human games could have been used instead, or the network could have been trained against opponents other than itself, or some combination of these strategies could have been used to either speed up the training of the network or to improve final performance.

Lastly, during the tuning and testing process, the different heuristics and extensions to the program were examined mostly individually. Due to the number of possible configurations being too high, it was not possible to test the different combinations of the improvements, although doing so may have found that certain aspects worked particularly well with each other or may have required different parameter settings to achieve best performance.

Project organisation

This project managed to complete its basic, intermediate and advanced aims and deliverables. The nature of the project and the design solution meant that after the initial development on the basic MCTS program had been completed, it was easy to add lots of extensions on top of the existing framework. However, since the core MCTS component was used by all of the players, it would have been worth spending more time on this part of the software, specifically with an eye towards optimising more for speed. Although this project managed to survey a number of improvements to the MCTS player, some of these extensions were only at a basic level, and larger benefits may have been found with a more focused approach which examined some of the aspects in greater detail.

VI. CONCLUSION

In this project, we implemented a computer player that could play the game of Go using a Monte-Carlo approach, and then investigated the use of different heuristics and extensions to the basic algorithm to see which ones would provide the most benefit. We found that Rapid Action Value Estimation, a technique which aims to extract more information from each simulation, was the most effective way to improve the Monte-Carlo agent, winning out against all the other agents in a tournament setting. The use of Go-specific heuristics was also found to be effective, both when used to bias the search towards particular areas, and when used in heavy playouts to increase the quality of the simulations. However, an efficient implementation of heavy playouts is required for the technique to be worthwhile, as otherwise the extra time spent improving the playout quality could be better invested into simply running more simulations. Despite the current breakthroughs in computer Go stemming primarily from the improvement of neural networks, we found that a single-layer neural network was unable to provide any practical benefit. This could be because a network with this simplicity takes too long to train, or the network architecture could be insufficient for learning enough information about the game. When all the techniques were combined, the resulting computer player was able to defeat the author when playing on a 9x9 sized board with a four stone handicap, achieving the aim of being able to play at a level equivalent to that of a human beginner.

There are several ways in which this project could be extended. Firstly, improvements in speed could be looked for, as any Monte-Carlo based agent will be able to benefit simply by increasing the number of simulations it is able to run, and this would also make heavy playouts a more viable option when looking for improvements. Secondly, since RAVE proved to be successful, similar heuristic techniques which improve the search without having to incorporate expert knowledge could be examined to see if they can provide similar benefit. The most interesting area for extending the work done in this project involves neural networks with improved architectures and training methods. For example, convolutional neural networks could be used to try and detect local patterns, and some of the training could be done by looking at high-level games instead of relying solely on self-play.

REFERENCES

Bouzy, B., Chaslot, G. (2006) “Monte-Carlo Go Reinforcement Learning Experiments”, Proc. IEEE Conf. Comput. Intell. Games, 187–194

Brugmann, B. (1993) “Monte Carlo Go,” Technical report. Physics Department, Syracuse University

Drake, P., Uurtamo, S. (2007). “Move Ordering vs Heavy Playouts: Where Should Heuristics be Applied in Monte Carlo Go.”, Proc. 3rd North Amer. Game-On Conf., 35–42

Drake, D., Uurtamo, S. (2007). "Heuristics in Monte Carlo Go", Proc. Int. Conf. Artif. Intell. , pp.171 -175

Funke, R. (2012) “From Gobble to Zen” Beyond AI: Artificial Dreams, 10–20

Gelly, S., Silver, D. (2007), “Combining online and offline knowledge in UCT”, Proceedings of the 24th international conference on Machine learning, pages 273–280

Gelly, S., Silver, D. (2011), "Monte-Carlo tree search and rapid action value estimation in computer Go", Artif. Intell. , vol. 175 , no. 11 , pp.1856 -1875

Tesauro, G. (1994). “TD-Gammon, a self-teaching backgammon program, achieves master-level play”, Neural Computation, 6(2):215–219.

Kocsis, L., Szepesvari, C. (2006). “Bandit based Monte-Carlo planning”, European Conference on Machine Learning (ECML), 282–293.

Maddison, C., Huang, A., Sutskever, I., and Silver, D. (2015) “Move evaluation in go using deep convolutional neural networks”

Müller, M. (2002). “Computer Go,” Artificial Intelligence, Volume 134, Issues 1–2, Pages 145-179

Schaeffer, J. (1989) “The History Heuristic and Alpha-Beta Search Enhancements in Practice” IEEE Transactions on Pattern Analysis and Machine Intelligence 11(11):1203-1212

Schraudolph, N., Dayan, P., Sejnowski, T. (1994) “Temporal difference learning of position evaluation in the game of go”, Advances in Neural Information Processing Systems, pp. 817–817

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

Tian, Y., Zhu, Y. (2015) “Better Computer Go Player with Neural Network and Long-term Prediction”, arXiv: 1511.06410