# 9x9 Computer Go using Monte Carlo Tree Search

**Tom Brady**

Third Year Project Final Report 2024
Supervised by Magnus Wahlstrom
Royal Holloway University of London

# Contents

# 1 Abstract

The quality of Computer Go playing algorithms have drastically improved due to the development of the Monte Carlo Tree Search[4]. In this project, I will be exploring the performance of Monte Carlo Tree Search combined with techniques including UCB1 and its further enhancements in the context of playing 9x9 Capture Go, a game that has caused a new wave of ideas as to how programs are built to succeed in defeating experienced human players, primarily through the balance of the exploration-exploitation problem. I will discuss further the theory and functionality of bandit processes, the rules of Go, the method behind the Monte Carlo Tree Search and how these things in conjunction can be applied to my project.

# 2 Introduction

A game is said to have perfect information if there is no hidden information and both players have access to all the information that has occurred since the beginning of the game [9]. In a similar way to how games with perfect information can be played by two human players, such as tic-tac-toe, one player can also be substituted for a computer that can make moves in the game, given that it is aware of the rules. A structure of all possible moves that can be made can be visualised into a *game tree* which expands upon every choice that is taken. Algorithms have been developed over the years to decide which move would be most desirable in the context of the current state of the game it is playing. The Monte Carlo Method dates to the 1940s as a process to generate a random variable which has the expected value of the solution to a particular problem, which proved to be particularly useful in evaluating mathematical expressions [1]. This method, along with the min-max algorithm, was used in the creation of the Monte Carlo Tree Search by Rémi Coulom in 2006 [4]. The board game of Go has been particularly difficult to master for AI due to its complexity and the incomprehensibly large set of possible combinations of play in a single game. This challenge has been well tackled many times, but famously so when a team from Google Deepmind created AlphaGo, a 19x19 Go playing AI that was able to beat the European Go Champion 5-0 [15]. In my project to create an effective AI to play 9x9 Go, Monte Carlo Tree Search in combination with the Upper Confidence Bound algorithm will be used to produce the game tree structure and allow intelligent decisions to be made. The application should have the possibility for Go gameplay without the AI as well as playing against it. This report will outline my motivations and objectives along with:

- An overview on the current state of the art
- Theory behind the Upper Confidence Bound and Monte Carlo Tree Search
- Present enhancements being used in the literature
- Rules and explanation of Go

- My programs including the design and implementation of them
- An analysis of performance, followed by a critical reflection of results

## 2.1 Motivation for the Project

The aim of this project is to create an application that allows an AI to play a game of 9x9 Go and generate a sequence of moves that lead it to victory. When the project began, I was considering the choice to work on Connect4, Chess or 9x9 Capture Go. Not knowing what the game of Go was, I began researching before making my decision and quickly came across the story and execution of AlphaGo [15], which deeply interested me to pursue research in how Monte Carlo Tree Search was used to assist in the important milestone of conquering Computer Go and surpassing human players. I was enthralled at the idea of a new challenge, and decided to begin development on the project to create an AI that would be able to beat me in Computer Go, and perhaps go further by competing well with other Go-playing programs that have been developed in the past. I chose the project regarding Monte Carlo Tree Search as I have been previously interested in the nature behind Monte Carlo Simulations and their applications in a variety of industries including Physics, Finance and Computing.

## 2.2 Original Objectives for the Project

This section outlines a summary of my objectives throughout both the first and second term.

- Understand and implement the UCB1 algorithm (See Section 3)
- Experiment with the results of UCB1 compared to other techniques (See Section 3.6)
- Implement a playable 9x9 Go application that supports Player vs. Player (See Section 7)
- Write an algorithm to perform moves in Computer Go underpinned by the use of UCT and Monte Carlo Tree Search (See Section 8)
- Enhance the UCT algorithm with UCT-RAVE
- Explore heuristics to further improve the efficiency of the algorithm

## 2.3 Literature Analysis

There are many resources that exist to research in order to aid in the development of this project, as many of them are sourced throughout this report. One of said materials is the paper in support of the development of *Crazy Stone*[4], one of the first Computer Go playing algorithms to incorporate Monte-Carlo Evaluation which went on to defeat Kaori Aoba, a professional Japanese Go

player ranked at 4 dan. Crazy Stone stands to be one of the most monumental points of reference for the development of Computer Go playing for many other papers, as it was a novel creation at the time. The algorithm uses probability to determine the likelihood of a simulated game producing a move better than the current best move, as well as its own new constant representing the urgency of a move, based on the past mistakes of the algorithm. Particularly, it places more value on moves that are in *atari* (See Section 6.2) as they are more likely to be reacted to by the player. The algorithm described has long been outperformed by reiterations of similar methods, including *AlphaGo*[15] which showed particularly the importance of Monte Carlo Tree Search when used in a wider context to defeat World Champion Lee Sedol in 2016[2]. AlphaGo incorporated more than just a tree search, including a reinforcement learning driven policy network and a neural value network, showing that the steps in Monte Carlo Tree Search can be modified with other tools and prove to be even more useful than before. Futhermore, there are a plethora of heuristics that have been researched[13],[17] to improve Monte Carlo Tree Search and can be applied to game playing Computer Go which are further discussed in Section 5. Assessing the academic literature as a whole, a fairly heavy list of methods for playing Computer Go and enhancements are described in[3], outlining the kinds of heuristics that have been combined with MCTS and have shown positive results, such as All-moves-as-first's incorporation in RAVE (see Section 5). The most significant development in more recent years would be the creation of *MuZero*, which expands beyond Computer Go but to learn games like Chess and Shogi without prior knowledge of the rules, domain and without human data to support it's decisions[14]. It uses a more advanced model applied to a Markov Decision Process for which the tree search iterates over. Judging from most recent sources, it appears that methods of improving AI game playing is focused more on intelligent value networks and deep reinforcement learning to work with a tree search algorithm that chooses the best move, similarly to the simpler UCT but through more complex means.

# 3 Theory of Bandit Processes

## 3.1 The Multi-Armed Bandit Problem

Bandit processes were created to solve the Multi-Armed Bandit Problem, which has the general outline of a set of $n$ machines with *arms* that can be pulled repeatedly in any order. The only rules are that only one arm can be pulled at one time, and arm pulls either result in the generation of a reward or a failure[6]. Each machine has it's own probability distribution of generating a reward, but the algorithm is unaware of what those probabilities are. We are then given the task to maximise the amount of rewards returned over a certain number of iterations, for which bandit processes have been developed to solve. Common approaches mostly follow a *Greedy* model, in which the machine with the highest mean probability of generating a reward is sought out. This means

that the machine with the highest current potential of producing a reward is always chosen.[17]

## 3.2 Exploration and Exploitation

An important aspect of extracting the machine with the greatest potential is to balance *exploitation* and *exploration* in the algorithm. Exploitation refers to choosing the best option based on past experiences, whereas exploration refers to making new, potentially more attractive choices at the expense of possibly not getting a better result. The reason for its importance in the context of Go is that without some exploration, we are limiting the quality of moves performed by the algorithm and the opportunity for a greater outcome may be lost. A lot of the 'intelligence' of the program comes from this concept as it is very human.

## 3.3 Epsilon-Greedy

Epsilon Greedy is a fairly simple and one of the most common methods in multi-armed bandit problems. The basic principle includes a value $\epsilon$ that can be empirically assigned any probability between 0 and 1, and a chance to either exploit the best machine so far or to explore by pulling a random machine.

$$A(S) = \begin{cases} \arg\max Q(S,a) \; with \; p = 1 - \epsilon \\ uniform \; random \; action \; in \; S \; with \; p = \epsilon \end{cases}$$

- If $p < \epsilon$ then perform random action $A$ in $S$
- Otherwise pull current best machine

The above equation outlines the general performance of the $\epsilon$-Greedy method. It states that an action $A$ on the set of possible actions $S$ has two cases, The first being the machine that has produced the maximum reward thus far with a probability of $1 - \epsilon$, and the second being a completely random action on $S$ with a probability of $\epsilon$. The value of $\epsilon$ is our exploration parameter, and controls the frequency in which we choose to explore rather than continue to exploit the machine with the greatest potential. It can be generated each round with a separate mathematical function or changed empirically, however 0.1 is a fairly popular value.

## 3.4 UCB1 (Upper Confidence Bound 1)

UCB1 is the main method that will be used in this project to make decisions on where stones should be placed in the game of Go. The underpinning theory behind UCB1 works in the same way that Epsilon-Greedy does, attempting to balance exploration and exploitation. The method to achieve this is different however, as it is aimed to pull machine $i$ that maximises the following equation:

**Algorithm 1** Pseudocode for the $\epsilon$-Greedy method

```
t ← 1
ε ← input()
rounds ← input()
for 1 < t < rounds do
    chance ← random[0, 1]
    if chance < ε then
        pull() on random machine
    else
        pull() on max machine
    end if
end for
```

$$\bar{x_i} + \sqrt{\frac{2 \ln t}{n_i}}$$

- where $\bar{x_i}$ is the mean reward obtained from machine $i$ so far
- $t$ is the current time step
- $n_i$ is the number of times machine $i$ has been pulled so far

An important feature to note about UCB1 is that it will generate a set of values for each machine in order to pull the arm of the machine with the maximum value. As $\ln t$ increases logarithmically whereas $n_i$ increases linearly, the nature of a machine's UCB value is that it will decay as the number of times its arm is pulled increases, while the mean value will continue to increase so long as machine $i$ continues to generate rewards. This is in an attempt to balance the exploitation of the best machine with the exploration of other opportunities. Machines with a lower mean will have a higher UCB value than the machines with greater means as the timestep increases and the number of pulls on the machine stays low, giving it a boost. Similar to the explorative parameter in $\epsilon$-Greedy, the $\sqrt{2}$ coefficient in the numerator of the UCB can be adjusted empirically to affect the amount of exploration that is performed (the higher this value, the more exploration is performed).

### 3.5 Performance of Bandit Processes

There are multiple measures for how optimal a bandit process is. The simplest being *Asymptotic Optimality*, measuring whether or not the bandit process finds the arm with the greatest probability of reward.[19] The performance of Bandit Processes is also most frequently measured by the *regret*. Regret is defined as the difference in the maximum theoretical amount of reward that could be generated from a process against the actual generated reward from the algorithm, therefore meaning that the lower the regret is, the more effective the

**Algorithm 2** Pseudocode for the UCB1 method

```
rounds ← input()
for first n terms do
    pull() on each machine
end for
for k < t < rounds do
    for i machines do
        calculate μ_i + UCB_i
    end for
    pull() on machine that maximises μ_i + UCB_i
end for
```

algorithm. UCB1 is theoretically known to outperform Epsilion-Greedy as the number of turns increases due to it's logarithmic regret bound[11], which makes it a better choice for this program. Specifically, the UCB1 regret bound can be expressed as:

$$O(\sqrt{KT \log T})$$

- where $K$ is the number of actions that can be taken in a single round and $T$ is the total number of rounds.

### 3.6 Experiments with Bandit Processes

In my code, I have run some experiments with the UCB1 algorithm to assess the performance and optimality against other methods. Firstly, I began to test the UCB1 algorithm as a standalone (See Table 1), initialising three bandits each with experimental means $\mu_1 = 0.35$, $\mu_2 = 0.5$ and $\mu_3 = 0.65$ respectively. It should be noted that the % reward acquired is calculated as the mean total reward after 10 trials with $n$ rounds. The experiment shows that increasing

Table 1: Table showing the % of reward acquired by the UCB1 algorithm (out of the maximum possible rewards)

| number of rounds $n$ | % reward acquired |
|---|---|
| 10 | 41% |
| 100 | 51.8% |
| 1000 | 59.6% |
| 10000 | 63.8% |
| 50000 | 64.5% |

the number of trials gives the algorithm more time to establish which moves have a greater chance of producing a reward or not. Accuracy grows rapidly as you increase for smaller values of $n$ but begins to slow as $n$ gets larger. In all trials I found that UCB1 successfully found the ideal machine $\mu_3$, making

it Asymptotically Optimal (see Section 3.5), and that for larger values of $n$ the mean reward tends towards the mean chance of reward for the best machine. I then ran the same experiment following the method of pulling a completely random machine arm at each time step with the same bandits and parameters.

Table 2: Table showing the % of reward acquired by random selection

| number of rounds $n$ | % reward acquired |
|---|---|
| 10 | 31% |
| 100 | 48.9% |
| 1000 | 48.6% |
| 10000 | 50% |
| 50000 | 49.9% |

We can see from random selection that even at smaller values for $n$ we are tending towards the probability of $\mu_2$, since 0.5 is the average of all machine probabilities. There is a chance of this naive strategy outperforming UCB1 for small values of $n$, but it does not have the capability of finding the most reliable arm to pull that UCB1 does. Lastly, I ran the experiment with $\epsilon$-Greedy following the policy discussed in Section 3.3, with a value of $\epsilon = 0.1$. As shown

Table 3: Table showing the % of reward acquired by performing $\epsilon$-Greedy

| number of rounds $n$ | % reward acquired |
|---|---|
| 10 | 57% |
| 100 | 62.2% |
| 1000 | 63.4% |
| 10000 | 63.3% |
| 50000 | 63.4% |

from the experiment with $\epsilon$-Greedy in Table 3, the algorithm finds the machine with the best probability of producing a reward faster than UCB1. One could even say that it performs better for smaller values of $n$. So why not use this method instead of UCB1 for Go if it produces better results here? An important aspect of modelling this problem in the context of Computer Go is the fact that moves in Go at a certain state $S_1$ will not necessarily have the same value at state $S_2$, meaning it is vital to be able to alter the balance between exploration and exploitation in order to adapt to the current state of the game. While you can empirically choose the value of $\epsilon$ which is the explorative parameter, if set too high, this would cause the algorithm to deviate from the best possible move at the current state $S_i$ more often due to having a greater chance to choose a random machine.

# 4 Monte Carlo Tree Search

## 4.1 Monte Carlo Method

The Monte Carlo method[1] is an umbrella term for a set of mathematical techniques to predict the possible value(s) of an uncertain event through random sampling. These methods are specifically useful in contexts where finding an analytical solution takes too much work or is not possible at all.

## 4.2 MCTS (Monte Carlo Tree Search)

Monte Carlo Tree Search incorporates the random sampling from the Monte Carlo Method and applies it to operationing on a game tree. This is directly adapted through the use of *rollouts*, where the entirety of a game is simulated all the way to the end through uniform random play. These rollouts are then used to update weightings on nodes in the game tree in order to improve the quality of the future rollouts that are used to further simulate play. For Computer Go, a game tree is constructed at every instance in which we want to find the best possible move for the current game state. (I.e. when it is the AI's turn to move)

### 4.2.1 Selection

For every iteration of the tree search, selection begins at the root, which in this case represents the current state of the board. There are many methods for selecting nodes, examples of which include $\epsilon$-Greedy and UCB1 as discussed prior, or other heuristic methods to determine the optimal node. The commonality between them is that eventually we will reach a leaf node where no more traversal can take place.

### 4.2.2 Expansion

Expansion is the decision as to whether or not we should append to the leaf node we have traversed to. If the leaf node has never been visited before, it means that the move the node represents has not been simulated, and therefore we can simulate immediately from that state. However, if a node has already been explored, we append a new node to it for every action available from that state. For example in a board game like Go, the expanded nodes will contain the possible actions for the next turn relative to the node that has been expanded.

### 4.2.3 Simulation

Simulation will always occur on a leaf node. After through either expansion or traversal, a new node that has not been visited before will be found, which will be subject to the simulation. The simulation step incorporates the Monte Carlo Method, for which randomness is used to determine a result. We will continue to place random available moves until the game reaches an end state and return the result of the game, either being a win for black or for white.

#### 4.2.4 Backpropagation

Lastly, we update the new child node as well as all nodes that were included in traversal to the child node with the result of the simulation step. The update includes adding to the visit count of all nodes included as well as updating the wins of nodes that correspond to the colour of the victor (i.e. only black nodes will have their wins increased if black wins the current simulation). This process repeats until we reach the root node, where we can begin again with traversing the updated tree to find another leaf node for expansion or simulation.

### 4.3 UCT (Upper Confidence Trees)

UCB1 is exceptionally useful in the Selection stage of the MCTS. We can consider each selection of nodes to be its own Multi-Armed Bandit Problem as we can attribute each internal node to a bandit. [10], updating the UCB1 equation to the following:

$$\frac{w_i}{n_i} + c\sqrt{\frac{\ln N_i}{n_i}}$$

- where $w_i$ is the number of wins recorded for the current node at round $i$
- $n_i$ number of simulations on the current node at round $i$
- $c$ an empirically chosen exploration parameter
- $N_i$ total number of simulations on the parent node at round $i$

It should be noted that the value of $c$ is theoretically equal to $\sqrt{2}$ as shown in the earlier discussed UCB1 algorithm, however it can be chosen empirically in an effort to optimise the balance on exploration with exploitation. The fraction $\frac{w_i}{n_i}$ is the calculation of the mean which is our exploitation term.

### 4.4 Pseudocode for MCTS

**Algorithm 3** Pseudocode for Monte Carlo Tree Search

```
procedure ROLLOUT
    while True do
        if S_i is TERMINAL then
            return S_i
        else
            A_i ← random(A, S_i)
            S_i ← simulate(S_i, A_i)
        end if
    end while
end procedure
current ← S_0
while current NOT leaf node do
    current ← arg max(current_child(UCB1_i))
end while
if (current, n_i) = 0 then
    rollout(current)
else
    current ← new current_child
    rollout(current)
end if
```

The MCTS pseudocode begins at a root state $S_0$ which we assign to a node *current*. We then want to perform selection with UCB1 until we reach a leaf node, so we pick child nodes of *current* that maximise the UCB1 value. Once we have reached a leaf node we can no longer perform selection, so we must choose to either rollout on *current* if it has never been visited before, or to expand a new child node off of current and perform a rollout on the new node. In both scenarios we perform a rollout, which loops forever until we are able to return an end state. So long as the game is not in a terminal state, we continue to perform random actions that are available at state $S_i$. Once we have reached a terminal state we simply return the result where it can be backpropagated and MCTS can be run in the next iteration.

## 5 Enhancements of MCTS

### 5.1 Light and Heavy Rollouts

As simulation in traditional Monte Carlo Tree Search is completely random, there are some modifications that can be made to improve the efficiency of the search. The most common is to apply the results of previous searches as a heuristic for future searches (called a *History Heuristic*)[16]. A *light* rollout refers to performing Monte Carlo Simulation without heuristics to achieve a

generally accurate result, whereas a *heavy* rollout describes one that incorporates heuristics such as the History Heuristic to generate an even more accurate result.

## 5.2 Backup Strategies

While there are a myriad of strategies to improve simulation and traversal, strategies for backpropagation have also proven to be useful. [4] describes how nodes can be split into internal and external nodes, based on whether or not they have been visited more times than the current amount of points in the game. Internal nodes (ones that are higher than the current point score) are more likely to be chosen and dominate the others, so an effective backup strategy allows the algorithm to converge to this best move faster. The paper argues that when the number of moves are high and simulations are low, backing up the maximum node (the node with the most visits) does not reflect the best node but rather the luckiest one, and that other heuristics should be applied in order to ensure that nodes are picked based on probability of winning the game.

## 5.3 RAVE (Rapid Action Value Estimation)

Rapid Action Value Estimation was an algorithm proposed for Computer Go in the pursuit to improve the quality of Monte Carlo Tree Search. The following descriptions and ideas for these functions can be found in [17]. The RAVE algorithm incorporates the *All Moves As First* heuristic, which follows the notion that there should be one general value for each move, regardless of when it is played. This allows for fast and accurate estimation of values before simulating with Monte Carlo in an attempt to reduce the computational overhead of simulating all possible moves from all states.

### 5.3.1 All Moves as First

To understand how RAVE can be applied to the Monte Carlo Tree Search, It is important to understand how Monte-Carlo simulation can be used to estimate the *all-moves-as-first value*. This value represents the mean of all simulations in which action $a$ is selected at any turn after state $s$ is encountered. It can be defined by:

$$\bar{Q}(s,a) = \frac{1}{\bar{N}(s,a)} \sum_{i=1}^{N(s)} \bar{\Pi}_i(s,a) z_i \tag{1}$$

- where $\bar{\Pi}_i(s,a)z_i$ is an indicator function returning 1 if state $s$ was encountered at any step $t$ of the $i$th simulation **and** action $a$ was selected at any point after step $t$
- $z_i$ is the result of the $i$th simulation

- $\bar{N}(s,a)$ represents the total amount of simulations used to estimate the AMAF value.

#### 5.3.2 MC-RAVE

MC-RAVE is an enhancement to RAVE to calculate the overall value of a move as a weighted sum of the all-moves-as-first value as well as an $MC\text{-}Value$, which allows for balance between accuracy (from MC) and speed (from AMAF). The function for calculating the overall value of a node $n(s)$ can be denoted by:

$$Q_*(s,a) = (1 - \beta(s,a))Q(s,a) + \beta(s,a)\bar{Q}(s,a) \tag{2}$$

- where $\beta(s,a)$ is a weighting parameter that provides a schedule for combining the MC-Value with the AMAF-Value
- $Q(s,a)$ is the MC-Value
- $\bar{Q}(s,a)$ is the AMAF-Value

$\beta(s,a)$ is a function stored in each node that decides the weight of both values. The general scheduling policy is that for less simulations performed on node $n(s)$, we want a higher AMAF-Value and that for higher simulations on $n(s)$ we want a higher MC-Value.

#### 5.3.3 UCT-RAVE

MC-RAVE can then be appended to UCT in order to give the algorithm an explorative bonus, giving us the final function:

$$Q_*^{\Theta}(s,a) = Q_*(s,a) + c\sqrt{\frac{\log N(s)}{N(s,a)}} \tag{3}$$

Similarly to how UCT functions, we will then want to choose an action $(s,b)$ that maximises the equation above.

# 6 Computer Go

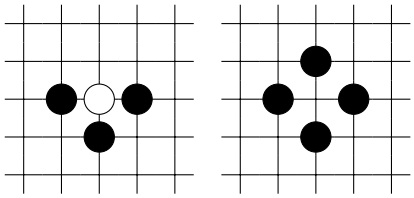
## 6.1 The Challenge of Computer Go

Unlike Chess, which saw great successes with alpha-beta searching being able to dominate human players[7], Computer Go presented a new challenge, which helped push the pioneering of a new class of algorithms relating to simulation-based search. Despite 9x9 Go having around 1 x $10^{38}$ legal positions which is less than Chess' 1 x $10^{40}$, the positions in 9x9 Go are too dynamic for a static evaluation like alpha-beta search to be effective[4]. The recent developments of MCTS has caused a revolution in AI Go-playing development in the attempt to create algorithms that surpass even the most experienced human Go players.

## 6.2 Rules

The game of Go revolves around placing stones and controlling territories. Black and white stones are played in alternating turns, starting with black. Collections of stones that are adjacent to each other on the board are referred to as groups, and territories can be formed by stones surrounding vacant areas of the board, or by completely surrounding a group of opposing stones. Stones also have the property of *liberties* which are the adjacent positions on the board that are unoccupied. To capture an opponents stone or group of stones, you must reduce the group to zero liberties.

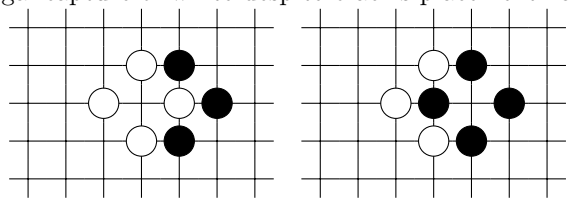
Figure 1: A simple capture is made as black reduces the white stones liberties to zero.



We can say that the white stone in Figure 1 is in *atari*, meaning that it only has one liberty remaining. Furthermore, it would be an illegal move for white to then play inside black's surrounding area as the stone would have no liberties. This rule is not always applied however as there are situations where a stone can be placed in a location with zero liberties given that it gains one or more liberties from a capture upon placement. Naturally, stones placed on the sides or edges of the board require less stones to reduce them to zero liberties, and can be considered more vulnerable.

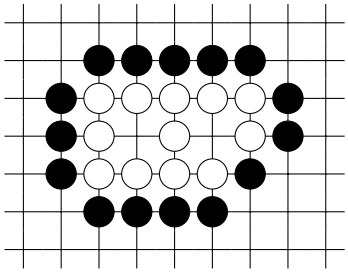
Despite black having a legal move in Figure 2, it is illegal for white to immediately re-capture the territory it lost due to the rule of *ko*, which states that no board position can be repeated, as it would allow the game to continue on forever. Life and death is an important concept in Go, as groups of stones can be surrounded but are still considered *alive* due to the presence of *eyes*.

Figure 2: A legal capture of white despite black's placement having 0 liberties



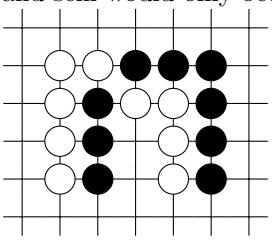
Without *eyes*, completely surrounded groups can be considered *dead* even before they are captured as it is inevitable that they will be.

Figure 3: White has a set of eyes, therefore the surrounded group is considered *alive*.



There is however one exception in which two eyes are not required for a group of stones to be alive, which is *seki* (mutual life). This can occur in many ways, but always when groups of opposite colour are adjacent and also share liberties. The territories stay empty as neither player will want to move as doing so will allow the opponent to capture their pieces. No points are provided to either player for territories that exist as a result of seki.

Figure 4: A simple *seki* example. Both players will abstain from playing inside the territory. It should be noted that the last line of stones placed signify the bottom of the board, as a valid seki would only occur there.



### 6.2.1 Komi

Komi is the term used to reference the points that white gains because of the advantage black has from moving first. It is a different value in different rule sets and game sizes, but it is widely known as 6.5 in both Japanese and Korean rule sets on a 19x19 board. The reason for the value being a decimal and not a whole is to reduce the frequency for which ties occur. In 9x9 Go, different organisations use different values but for this project I will be adhering to the AGA (American Go Association) with 5.5 points for white as default.

# 7 Programming Computer Go

Programming Computer Go has proven to be quite difficult given the complexity that comes with the game on its own. Compared to a Monte Carlo Tree Search

project regarding a similar game such as Connect4, I had to take more time to build the foundation for which I built the algorithm on. I chose Python as the language for the project since it is my most experienced language, as well as having knowledge on libraries I have used in the past for processing large amounts of data that might be useful later on. During development, I aimed to follow proper software engineering practices such as *Test Driven Development* as well as maintaining my code on *GitLab*. TDD is the practice of writing tests before working code to emulate the feature that you wish to implement, followed by writing the minimal amount of working code to achieve that goal. This helped massively with building classes and testing methods for objects on both general and more specific edge cases. I was presented with a new challenge during testing however, as some methods that incorporate the Monte Carlo Method produce probabilistic results due to their randomness, rather than deterministic ones. This made checking for results sometimes ambiguous, but I worked around it by forcing the game into situations where only one result could be produced. As I was using Python, I decided on the *unittest* module to handle testing as it integrated with my development environment (VS Code), felt more lightweight and followed the routine of creating test classes, similar to the classes created for working code. While managing code on GitLab, I aimed to separate the code based on theory and the code used in playing Go as they do not require interaction. I kept the main branch protected by only merging once a working release had been made, once for the end of the interim, and once now for the final submission. While I did not enforce a checkstyle or linting, I rigorously documented my classes and methods with Python documentation and followed a strict variable naming convention through snake case, along with including comments where necessary for lines that require further explanation. I also found making TODO notes useful throughout as they would prevent me from forgetting where a piece of code still required attention.

## 7.1 Code Structure

The structure of my code follows traditional *Object Oriented Programming*, in which objects in the game are split into their respective classes and are able to interact with each other. Object Oriented Programming is a programming paradigm that revolves around creating objects that have attributes (values) assigned to them, as well as methods that can be called after initialising the object. I knew Object Oriented Programming would be useful for this project since there is a clear hierarchy of objects as will be made clear in the breakdown of the classes. This was also useful for inheritance of View classes as actions at a user level could easily be delegated to the backend through subclasses.

### 7.1.1 Directory Structure

Below is a rough diagram of the directory structure I designed for this project. I wanted to split all Python code from any utility files, so I placed all python related code in the *py* directory. I then split the code by category and relevance,

in order to prevent any confusion. Each directory includes a dunder init file in order for it to be discoverable by unittest. I maintained a .gitignore file to ignore pycache when switching between branches and any unwanted txt log files that were generated during the optimisation stage.

```
.gitignore
readme.md
go.uxf
py
    assets
    go
    mcts
    tests
    main.py
```

### 7.1.2 Constants

I created a constants file in the beginning to act as a point of reference for information that would be important in the game, such as images of the stones, dimensions of the board and so on. This was particularly useful when creating the front-end, as RGB values are referenced frequently in presentation and the code becomes far more understandable when those values are replaced with the colour they represent. The use of $BLACK$ and $WHITE$ was particularly important as I used it to identify the colours of the stones in the backend also, allowing me to know the colour of a stone at any point in the game.

### 7.1.3 Stones

Stones are the most important part of Programming Go. In Go, stones are operated on the most out of all other objects considering they are placed many times a game, deleted many times a game, as well as tracked by colour. The stone class has the following attributes:

- board - The board that the stone is associated with
- coord - The coordinate the stone is situated at, for example (3,3)
- colour - The colour of the stone, always either black or white

The stone class comes with some helper methods, often used whenever the consideration of a new stone is placed. The first of which being a neighbours method, which returns the neighbouring coordinates of a stone and returns them, also taking into account the sides and corners of the board and removing the coordinates that are outside the range. This method is extremely important as it's used to identify stones in the neighbouring locations. Secondly, a method to utilise the method to get the liberties of a stone. If a stone (or group) is reduced to zero liberties by the opposition, the group dies. It's important for this reason to keep track of the liberties of each stone at any given point. When

a stone is created, it must immediately be assigned to a group if it has one. There are three cases in which a stone acquires a group, being the following:

- Case 1: The stone is isolated, creating a new group for itself
- Case 2: The stone is placed next to a group of the same colour, therefore joining the group
- Case 3: The stone is wedged between two groups of the same colour, therefore merging the two independent groups into a single group

The method begins by creating a list that will be used to store any groups that the new stone is next to, as well as finding out what stones are in the neighbouring positions of the new stone. If there are no groups in result, we can see that we have Case 1, where there are no groups surrounding the stone. If there is one group in result, we observe Case 2 and append the new stone to the group that already exists. Lastly, if there is more than one group in result, we continually merge the groups (using the merge method explained in 7.1.3) until there is only a single group left, then append the new stone to that group. This method is run in the constructor of a stone object to enforce that all stones but be associated with a group.

### 7.1.4 Groups

Group objects signify groups of stones that exist on the board. Stones have to be grouped together because captures can affect more than just an individual stone, and groups in Go essentially act as a single entity rather than a collection of stones that are all independent of one another. Taking this approach helped simplify the way groups are handled. A group object has just two attributes:

- board - The board the group is associated with
- stones - a set representing all the stone objects that are a part of the group

Group objects have two main methods, being the merge function as well as an update that checks the liberties of the group and removes it if it has zero liberties. The former merge method takes another group object as it's argument for the merge, and iteratively adds the stones of the second group to itself, then deleting the second group from the board. The latter method iterates over the set of stones and checks the liberties of each. If after the loop there are no stones with any liberties, the group deletes itself. Deletion is performed through a helper function, as deleting the group also has to delete the stone objects within the group, which is done through delegating the process to a helper function in the stone class that deletes stone objects.

### 7.1.5 Board

A board object is where the game is actually played. In the program, information is fed to board objects through placing stones on the window. In reality,

the board is being updated with new stones and responding to the changes appropriately. A board object has four attributes, being:

- turn - The current turn of the game
- white points - The total points accumulated by white so far
- black points - The total points accumulated by black so far
- gameover - Whether or not the game has finished yet

The constructor also initialises a new board object to have an empty list of groups, as well as an empty list of passes that will be used in tracking whether or not a player has forfeited. The board class contains a few methods, the most important of which being $find\_stones$, which takes a list of one or more coordinates and returns stones that exist in those coordinates. We build a list that is appended to with stone objects when the coordinate of a stone matches with any of the coordinates that we are searching for. This is an extremely helpful method in verifying whether or not it is suitable to place a stone in a specific location or not. Turn methods are also included, in order for the colour of stones that are placed to correspond to the current turn. In Go, black always moves first, meaning the turn can be tracked very simply by setting the turn to zero, incrementing it by one each time a stone is placed, and checking whether or not the turn is even or odd to return black or white respectively. Passing turns is tracked by building up a list of the turns for which a player has passed on. Two passes in a row signify that a player has forfeited the game, and two passes in a row in consecutive rounds mean that both players have passed and the game ends with no victor. For example, if a player passes on turn $x$ and they have already passed on turn $x-2$, they forfeit. Furthermore, if the other player has passed on turn $x-1$, the game ends and is considered a draw. Lastly, the board class has an update method that takes a single new stone as its argument, which updates all groups on the board as necessary so that groups respond in the correct way to the new stone being placed.

#### 7.1.6 View (Stones)

Naturally, stones have to be drawn onto the screen in order for the player to interpret the game at its highest level. StoneView is a superclass that inherits the stone class while adding some methods that allow it to be represented visually. Typically there should only be one board utilising the view classes and all other calculations involving stones should be abstracted from the view of the user. The view class only appends a few front end methods, and stones are immediately drawn to the window upon construction. I also wrapped the delete method from the stone class to draw the board over the stones to be deleted, removing them from view.

### 7.1.7 View (Board)

Similar to the stone view class, the board view class also immediately draws itself to the window upon its initialisation. A lot of the constants defined in the constants file become useful here for understanding the code, as longer lines of drawing pygame objects are reduced down significantly. The status of the groups can also be updated through the Board class here due to the inheritance.

## 7.2 Pygame

Pygame[12] is a set of Python modules used for creating video games, which I used in the creation of my Go application. I was aware of Pygame's existence prior to the project, as well as some of its capabilities. It is built off of the SDL library and has the aim of providing a simplistic experience to developing games in Python. This seemed like the obvious choice for an implementation in Go as very few front end components are required, given that Go is a game that does not need heavy animations or complex visual events. Pygame supports methods like drawing on the screen, handling input events such as clicking in order to place stones, and all general graphics. This allowed me to focus more of my concern on the structure of my Go application rather than how the front-end will interact with the back-end at a lower level.

## 7.3 Pygame-Menu

Further to simplifying the process in designing front end components for the user, I used the pygame-menu library which is an extension of pygame in order to generate the menu that opens upon running the main program. After beginning design for the AI game, I needed a way to distinguish player versus player and player versus machine. I encapsulated the two options into their own methods and used the pygame-menu methods to generate a simple menu for choosing which type of game you want to play.

## 7.4 GUI Choices

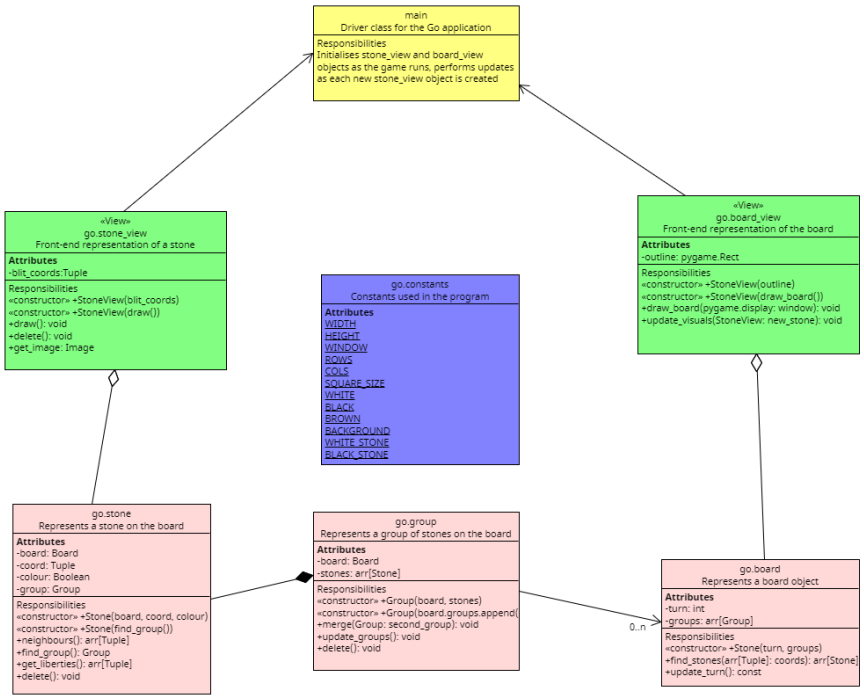
I had no plans to create an extravagant interface, moreso just a functional one with the necessary components. Despite this, I wanted at the very least for the Go board to mimic the look that it comes with in real life: the wooden board with black intersections. This allows those that have played Go before to know exactly how to begin the game. The pygame-menu class also comes with built-in GUI functions, along with themes and the ability to control the menu with the arrow keys, so I chose a simple theme too. The menu is presented when running the program and gives the user the option to play versus another player or to play against the computer. I also used the *PyAutoGUI* method to provide pop-up windows for players to know the score after one of them forfeited the match. Finally, I included a prompt that allows the player to set the amount of simulations that the computer will run whilst they play against the AI, in

addition to a prompt asking the user what they would like to set the komi value for in both player versus player and player versus computer games.

## 7.5 MVC Model

For the game of Go, I have implemented the methodology of *Model*, *View*, *Controller*, which involves splitting the code base into distinct purposes, being the back end model of of Go board, the visuals that are presented to the user, and the controller that updates the model upon user input and returns a visual response through the view. To accomplish this, I split the board and stone classes into model and view so that the *Driver* (main) class is able to communicate with the rest of the program in an organised manner. Details of this are expanded upon in the UML diagram below.

## 7.6 UML Diagram for Go




The UML diagram breaks down the class structure for the program further. The relationship between stones and groups is a composition as stones are assigned to a group when they are created, meaning that stones cannot exist without their group. Similarly, when a group is reduced to zero liberties and is removed from the board, all of the stone objects that exist within the group are also removed from the model. 0 to $n$ groups can exist on the board, as in the initial state of the board there are no stones placed at all, however groups of stones

can accumulate on the board up to $n$ times. Stones and the board are inherited by their respective views to allow updates to the model to be visualised in the application. The surface level interaction between the user and the interface is handled in the main class, where *stone_view* and *board_view* objects are created. Finally, I have listed the constants that are used throughout the files where necessary, although they do not have any direct impact or association with any particular class.

# 8 Programming MCTS for Go

Programming Monte Carlo Tree Search for Go has proven to be quite difficult in both implementation and optimisation. Due to the high complexity of Go already, it was difficult to create a functioning program as there are so many factors that define good and bad play. Not only did the programming for MCTS cause me to refactor the driver class to work with the new method of play, but I also had to refactor the structure in the underlying code to ensure a decent level of performance.

## 8.1 Nodes

Nodes are the core part of MCTS. A single node holds a large amount of information including the current state of the board that it represents. It is also important to track certain statistics of the node for backpropagation as many updated searches will need to take place. The attributes of the node class consists of the following:

- children - initialised as an empty array but is filled with nodes that it the node expands later on
- parent - the parent node in the tree
- state - the board state stored in the node
- wins - the amount of wins that the node has
- simulations - the amount of times the node has been simulated, used to ensure that no node is simulated more than once
- visit count - the amount of times we have traversed through the specified node
- expansion move - if the node has been expanded upon, the action that was generated for that new node
- colour - the colour of the node, either black or white, which represents the turn of play
- actions - a list of possible actions

All of the nodes attributes are used in some way for the search, so it is important to keep all of these up to date for every iteration. As we are using UCT for the traversal, the node class has a getter method that calculates its current UCB value given the current point in the game. It uses the same equation as mentioned in Section 4.3, but in the context of the attributes, can be expressed like this:

$$\frac{wins}{visit\_count} + 2\sqrt{\frac{\ln timestep}{visit\_count}} \tag{4}$$

Nodes also need to have a list of actions that is available to them. I have implemented this as a list comprehension in the constructor to generate all 81 positions that are possible in a game of 9x9 Go. This list is exhausted in a method to generate random positions, in which a random choice is made and then removed from the list of actions. It is removed because the move will either be chosen or discarded by the MCTS. I have included a try catch block to catch when the actions list becomes empty, in which case the function just returns null. I could have generated the list of actions dynamically such that each node has only the actions available considering its current state, however due to random choices from a list taking constant time I thought that the impact would be fairly negligible.

## 8.2 Copying

When creating new nodes for expansion or rollout nodes for simulation, copying the current state of a node is essential. Without copying the state, I would have to manually input all the moves up to that point, which would be quite difficult to achieve. For MCTS operations I used the copy module, and more specifically, the *deepcopy*() function. I used deep copies as opposed to shallow copies as editing the attributes and objects encapsulated by the copy holds references to the original and will edit the original object. In the case of rolling out nodes, I want the rollout node to be completely separate from the node I am choosing to simulate from, but also for the rollout node to have an initial state that is a copy of the state of the leaf node, which is where *deepcopy*() comes in very usefully.

### 8.2.1 Example Difference Between Shallow and Deep Copies

Given a list x, and respective shallow and deep copied lists x_shallow and x_deep, changing the contents found in $x[1][0]$ in the x_shallow list causes the original to also be edited, however the same changes on x_deep are not. This is an important consideration when copying nodes as performing a shallow copy and then editing the contents of the board within the shallow copy would have adverse effects on the original list.

```
x = [[1,2,3], [4,5,6]]
x_shallow = copy.copy(x)
x_shallow[1][0] = 7
print(x)
```

Produces the following output:

```
[[1,2,3], [7,5,6]]
```

Whereas doing the same with a deepcopy produces the following results:

```
x_deep = copy.deepcopy(x)
x_deep[1][0] = 7
print(x)
print(x_deep)

[[1,2,3], [4,5,6]]
[[1,2,3], [7,5,6]]
```

## 8.3 Traversal

I have coded the tree traversal to work recursively, checking whether the current node in the function call is a leaf or not. If the node has no children, it means that it's a leaf node. If not, find the node in the list of children with the highest UCB value and call the traverse function on that node until we reach a leaf.

## 8.4 Expansion

Expansion, like traversal, also has one of two cases. If the visit count of the node we are looking to expand is zero, then there is no need for any expansion and we can return that node for it to be simulated. If the node has been visited before however, we will start by generating positions for every action held in the actions array for the node. For each position, we will check whether or not a stone exists there, as well as that the generated position is not null. If no stone exists and the position is valid, we will generate a new node object, initialised with zero wins, simulations and visits, with a state attribute that is a deep copy of the parent state. The expansion move attribute will then be assigned to the generated valid position. A stone will be placed for the action and the board is updated for that node, now making it one action ahead of the parent node. In normal Monte Carlo Tree Search, we choose the node to simulate on after expansion randomly, but since the actions were generated randomly, I just return the first child that was generated in the expansion. This does not interfere with future traversals, as nodes are only expanded once.

## 8.5 Simulation

The rollout method begins by increasing the number of simulations for that node. I then create a new node for the rollout that has a deep copy of the state of the node to simulate from. A copy needs to be made here so that the simulation has no effect on the actual node in the tree. While the game has not ended, new positions are generated in the same way as expansion, except this time, when there are no more actions left to take, the player in the simulation

passes. If the action is valid, place the stone and continue the game until one player passes twice. While in the real game forfeiting signifies a voluntary loss, forfeiting here is more like the computer agreeing that there are no more possible moves to be played, so rather than giving the loss to whichever player forfeited first, the loss goes to the player with the least points after the forfeit. The method increases the total number of wins for either colour across the entire tree, and returns the victor for the simulation.

## 8.6 Backpropagation

Backpropagation is the last method in the MCTS process. I have implemented it recursively in a similar way to traversal, utilising the parent attribute of nodes. Every time the method is called, it increases the visit count of the current node. It then checks whether the colour returned in the simulation result is the same as the colour of the node in the tree, and if so, increments the wins of that node by one. Lastly, if the parent of the node is not null, call the backpropagate method again on the parent. This keeps the method running until we reach the root, fully updating the tree.

## 8.7 Kernel Methods

The *play*() method is the method that is called by the main class when the player plays their first move. It runs by creating a new node to act as the root, with no parent and one visit, in order for it to immediately expand to options for the computer's response. It then runs the four components of the MCTS a given number of times, and finally chooses the node in the root's children with the highest visit count with the *attrgetter* method from the *operator* module, which allows for efficient lookup on objects. It then returns the selected node, representing the best move one ahead of the previous board state, and plays it.

## 8.8 Main

The main class immediately loads the window and presents the user with the menu. Based on the button pressed, the main method directs the player to the PvP method or the AI method. PvP initialises a single board_view object for the two players to play against each other on. The AI method however, generates a board_view object as well as a backend only board for the game tree to be built off of, which is tracked and updated with the live game.

## 8.9 Optimisations

Because of the nature that thousands of nodes are created for each move, it takes a lot of computational work. It became painfully clear after implementing Monte Carlo Tree Search for the first time that I needed to give optimisation some attention in order for the search depth to be sufficient. I did not want the player waiting for too long for each move to be processed but still wanted the

quality of the move to be enough that it could be considered intelligent. It is no secret in the literature that optimising Computer Go is a challenge, and there are heuristics discussed in Section 5 on strategies to mitigate that.

### 8.9.1 cProfiler

Python profiling was recommended to me by my supervisor and has served as an important tool in determining what functions need optimising. The cProfile module allows you to run files while tracking the amount of function calls and cumulative time, producing a list of statistics on those metrics when the program has finished. As I wanted to test the entirety of a game played with the AI, I frequently used the following command in the Python shell:

```
python -m cProfile py/main.py > x.txt
```

I stored the logs produced by cProfile in text files as they were too long for the terminal and were easier to analyse through searching the document.

### 8.9.2 Sets

At the point of the interim, I had implemented Go boards as a list of groups, each containing a list of stones. The structure of my program involves a lot of lookup operations, particularly in *find_stones*, *neighbours* and *get_liberties*. As these were all stored as lists, lookup can be slow and averages at $O(n)$ time complexity, whereas sets are represented as hash tables, and therefore have an average lookup time of $O(1)$. While this increase in speed is not guaranteed since worst case lookup in hash tables is also $O(n)$, in most scenarios, changing the list of stones that a group object has to a set of stones improved the speed quite a lot. Another example of using sets to improve the efficiency of the algorithm would be the direct improvements I made in the *find_stones* method.

Listing 1: Previous find_stones() implementation

```
result = []
for group in self.groups:
    for stone in group.stones:
        if stone.coord in coords:
            result.append(stone)
return result
```

As can be observed from the code above, we are building a small list based off of comparisons between the given coordinates list and the stones on the board. We are essentially scanning the entire board here, and comparing the coordinates list each time, leading to an $O(n^3)$ time complexity: quite inefficient. I ran an experiment using cProfile to see how long a set number of simulations would take given this implementation. The constants for this experiment were that I ran 160 simulations on the same computer, placing a single stone in position (5,5) and closing the program after the algorithm responds to that move.

```
ncalls  tottime  percall  cumtime  percall function
 160    0.024    0.000    6.072    0.038 rollout
 547336 3.209    0.000    3.285    0.000 find_stones
```

The log shows that the *find_stones*() method was called 547336 times, and that the total time spent in the function was 3.209 seconds. Considering the cumulative time (total time in a function including time in it's subfunctions) of the rollout was 6.072 seconds, this shows we're spending half the time of the simulation in the *find_stones*() method.

Listing 2: Current *find_stones*() implementation

```
coord_set = set(coords)
result = [stone for group in self.groups for stone
in group.stones if stone.coord in coord_set]
return result
```

Changing the implementation to the above improves the efficiency as the lookup comparison in coord_set becomes $O(1)$, and the implementation of a list comprehension rather than two for loops makes the generation of the result array faster. From these improvements, I ran the cProfile module again under the same conditions, and observed the following results:

```
ncalls  tottime  percall  cumtime  percall function
 160    0.025    0.000    5.156    0.032 rollout
 547085  0.282   0.000    2.334    0.000 find_stones
```

The experiment shows that now the function runs much faster, a whole second in fact, while still being called approximately the same amount of times.

## 8.10 Performance Analysis

Now that the code for MCTS has been optimised, I can perform an analysis on the runtime of the search tree using different parameters for the MCTS operation. Starting with simulation count, I examined the time taken for the algorithm to return a move for 100, 250, 500, 1000 and finally 2000 simulations. The move was the same every time, a black stone on (5,5), using the same computer. The method in question is the *play*() method. It is clear that there

Table 4: Table showing the runtime of MCTS based on the number of simulations

| number of simulations | time spent in *play*()(s) | % time increase |
|---|---|---|
| 100 | 1.962 | n/a |
| 250 | 6.026 | 207.1 |
| 500 | 9.149 | 51.8 |
| 1000 | 16.773 | 83.3 |
| 2000 | 34.119 | 103.4 |

is an increase in the amount of time to return a move based on the amount of simulations. What is interesting about the percentage time increases is the drastic increase between 100 simulations and 250 simulations. The reason for this is due to the amount of expansions performed at 100 simulations compared to at 250. At 100, the program is just beginning to expand to depth 3, meaning it has less nodes to consider compared to 250 iterations. I expected to see close to a 100% increase when changing the simulation count from 1000 to 2000, seeing as most of my programming tends towards $O(n)$ in the average case. Considering the case where we have 250 simulations, we can look deeper into what methods take the most time during the decision to place a stone, down to the built-in methods in Python. Breaking down the 6.026 seconds taken by

Table 5: Table showing runtime of methods in play()

| function name | time spent in func (s) |
|---|---|
| traverse() | 0.024 |
| expand() | 1.396 |
| rollout() | 4.604 |
| backpropagate() | 0.001 |

the *play*() method, we can see that the rollout took the most time. Traversal and backpropagation take almost negligible time, as they only have to traverse a limited amount of nodes. Traversal does take a little more time however as it is calculating the UCB score of each node at every level. Further decomposing the expand() and rollout() methods shows us that very little time is spent in the method itself, and rather on the board operations that are being performed at each step. It is difficult to pinpoint how long certain functions like find_stones() and get_liberties were running for for expand() and rollout() as they both share these functions, however a large 4.106 seconds was spent in update_board(), 3.364 seconds of which were spent in get_liberties(), the function responsible for checking whether or not a group of stones has been reduced to zero liberties.

# 9 Reflection

## 9.1 Achievements

Overall, I am quite content with the product. Although it doesn't have all the features I set out to implement at the beginning of the project, it still serves as an application with multiple methods of playing Computer Go. The tree search algorithm operates in the correct manner and produces a result on the screen for the player to interact with. I am particularly pleased with the code structure, as I believe all methods are well documented, understandable and efficient. I think I made good decisions in the construction of different classes and how they interact with each other, creating the necessary attributes for larger function and removing redundancies. The GUI also has a solid design with clear options

and uses user input to communicate with the backend on how AI games should be played.

## 9.2 Difficulties

Difficulties are expected with every project and I understood that there would be challenges during the process. One of the main challenges during the second term was that the step up in complexity between Player versus Player and Player versus AI was a far greater leap that I imagined. Now that the largest scale object I had created in the first term (the Board class) had become an attribute in Nodes, an object that is duplicated thousands of times, the complexity of assuring that each board in each node had accurate data was a challenge. One error I had made in programming was creating the BoardView class to only be initialised once but not enforcing it, and then trying to copy it into the nodes, which interfered with the window in which the real game was being played. In hindsight, it would be better to implement the BoardView class as a Singleton pattern so that once one instance is created, another cannot be made. This higher level of structure made testing quite difficult as well, as verifying that node objects had the correct information, I would have to look into the board for each node, and sometimes stones within the groups of that board to confirm the board had changed in the expected manner, which produced quite verbose results. The amount of planning and thought going into new functionalities grew progressively longer, even sometimes causing an overhaul of code that had previously been written.

## 9.3 Limitations

Monte Carlo Tree Search, and especially in the context of Computer Go, comes with computational limitations. For example, AlphaGo's AI processing was not run on just a single computer[15]. It implemented its search policy in parallel across 8 GPUs in 40 different threads, as well as performing simulations across 48 CPUs. Without an efficient value network, taking an exhaustive route and simulating all possible moves proved to be rather slow no matter how much the code was optimised. The main limitation when running MCTS in the context of Go is raising the search depth in the early to mid stages of the game. Considering a board with a single stone on it, the search algorithm would expand and consider 80 moves on the first layer, and assuming that the algorithm simulates all of them, each of those 80 nodes will produce 79 children, which will all be exhaustively simulated. This means that in total, the program would perform 6320 simulations just for the first three levels, which is already computationally demanding. This begins the question on how certain moves, or even games can be pruned in order to reduce the number of simulations that need to be done, but even then, if we were to reduce the amount of simulations by 50%, it still would not be long before we would run into the same issue with a search depth over five.

### 9.4 Future Enhancements

To enhance the project further, some kind of policy network should be incorporated in order to prune the amount of simulations that are occurring. The two ways to make the search more 'intelligent' is to either increase the depth of the search or to make more optimal decisions in the traversal phase (like RAVE, adding to the UCT algorithm). This may reach beyond the scope of the project as machine learning would be a good research point from here. An example of a policy I think that would be helpful to solve the current issues would be a set including a large amount of simulated games, each with their own results. When the AI makes a move, it can compare with this set to see the results of games from that point, and if over a certain threshold of win percentage (i.e. if the AI appears to lose frequently from here) then we can prune that section of the game tree. As more games are played with this methodology, the set size would increase and progressively improve the performance of the AI in the game. Another enhancement that would be useful moving forward would be a backup policy, potentially one using probability to determine the best move instead of a deterministic result, in addition to a threshold constant used to prune unfavourable subtrees that are producing a winrate below the threshold and can be considered to be wasting computational effort.

### 9.5 Conclusion

In this report, I have reviewed the theories and methods surrounding Monte Carlo Tree Search and it's applications in 9x9 Computer Go. Documentation on it's implementation and capabilities, along with an unbiased reflection has also been provided. Personally, I have thoroughly enjoyed exploring the research surrounding the project and writing my own programs, along with gaining experience in working on long-term programming goals.

## 10 Professional Issues

Projects of any size require adequate planning and thought. Projects of this scale however, cannot and must not be planned once but instead must be revised throughout the entirety of their generation. Accurate management is required for all projects, both before and during development, and the ability to adapt to changes is pivotal in order to output great products. At the start of a project, a dedicated amount of time and resources should be allocated to gather knowledge on the task that is trying to be achieved, whether through research, communications with stakeholders or recruitment of staff in order to lower the chances of large roadblocks appearing later on. Companies must communicate well with stakeholders in the beginning to earn investments in their projects. Particularly for start-ups, it is important that these investments are spent wisely on the appropriate staff and infrastructure so that business value can be created. An example of a failure regarding this would be the music streaming service CrowdMix[5], which failed to spend its investments wisely and ended up shutting

down even before creating a public working product. Not enough attention was paid to hiring those that would create the service for them, but rather to individuals that would raise the 'profile' of the company. This led to £14m of stakeholder money going to waste, providing a lesson for other companies to take appropriate action in making sure promises on creating a product are delivered. It was stated that CrowdMix employees did not have a clear description of what the product was actually supposed to be, which is even more worrying from a management standpoint as it leaves no direction to be made for the company. When working in a technology based team, I believe it is best to incorporate smaller goals through Agile or through Kanban boards in order to ensure the necessity to adapt to a new revision for a project is maintained, otherwise you end up being at risk of falling behind and losing business value. The main difficulty during this project was the continuous re-evaluation of the amount of time it would take to perform a task or complete a certain implementation. A lot of research was made in the theory behind the processes and I gained a vast understanding of how algorithms worked for the product to be successful, however the journey from understanding to implementation took more time than expected when I considered all the other moving parts in the program, particularly in the later stages of the project. Some time had to be spent refactoring 9x9 Go and writing new methods in order to put the pieces together for MCTS before writing the code for it. Making sure the core components of the project (I.e. 9x9 Go application and MCTS implementation) were my top priority. As with the requirements that come with the project, they can be compared to stakeholders or clients demands when faced with a project in the real world, and therefore have to be given the highest priority and most attention. I think a good solution for this management issue in the workplace is a software like Jira by Atlassian. Not only does it provide a kanban board for managing work, but pieces of work can be tagged to people, given a priority level and a deadline. Furthermore, another key professional issue that is closely tied into the development of Go playing algorithms is the ever increasing worry that intelligent computers will replace human work. During the exhibition series of games between Lee Sedol and AlphaGo in 2016, many fans of Go thought that it would be completely out of the question for Lee Sedol to lose as he was the world champion, and Go playing algorithms in the past had not come close to the percieved level of Sedol. A large number of people thought the task of mastering Go to be impossible many years ago, and yet it was done and moreover enhanced upon in recent years. Particularly in the employment field of technology, people are becoming increasingly fearful that AI will replace them, as companies have already begun work on models to automate the workflow of a software engineer[18]. Considering the improvement in performance compared to AI models that were implemented just a few years prior, we can see a similar pattern from the improvements made from AlphaGo to MuZero in just four years and can perhaps expect to see a similar change in the coming years. Despite not directly 'replacing' Go players, it has certainly come to light that a computer can outperform humans with relative ease, and there are serious ethical issues that arise when the same concept is placed different contexts such as medicine

or law. Things lead to an ethical issue because it can be argued that AI does not always make human like decisions. Often in the adoption of AI in Computer Go, the AI would make moves that even some of the best players in the world would even consider not to be good. If this kind of behaviour is transferred to a medical diagnosis or a legal decision, it can be difficult for developers and the public to endorse programs replacing humans if it makes unruly decisions, and moreso puts companies at risk of taking responsibility for those actions.

# References


[1] W.F. Bauer. *The Monte Carlo Method*. 1958. URL: https://www.jstor.org/stable/2098715.

[2] BBC. "Artificial intelligence: Google's AlphaGo beats Go master Lee Sedol". In: *BBC Technology News* (2016). URL: https://www.bbc.co.uk/news/technology-35785875.

[3] Cameron B. Browne and Powley et al. "A Survey of Monte Carlo Tree Search Methods". In: *IEEE Transactions on Computational Intelligence and AI in Games* 4.1 (2012), pp. 1–43. DOI: 10.1109/TCIAIG.2012.2186810.

[4] Rémi Coulom. "Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search". In: *Computers and Games* (2007). DOI: https://doi.org/10.1007/978-3-540-75538-8_7.

[5] Eamonn Forde. "Anatomy of a Catastrophe: the fall of CrowdMix". In: *MusicAlly* (2016). URL: https://musically.com/2016/08/05/anatomy-of-a-catastrophe-the-fall-of-crowdmix/.

[6] J.C. Gittins. *Bandit Processes and Dynamic Allocation Indices*. 1979. URL: http://www.jstor.org/stable/2985029.

[7] The Guardian. "Deep Blue computer beats world chess champion". In: *The Guardian Chess Archive* (2021). URL: https://www.theguardian.com/sport/2021/feb/12/deep-blue-computer-beats-kasparov-chess-1996.

[8] Jaap van den Herik Guillaume M J-B Chaslot Mark H.M. Winands. "Parallel Monte-Carlo Tree Search". In: *Computers and Games, 6th International Conference* (2008). URL: https://dke.maastrichtuniversity.nl/m.winands/documents/multithreadedMCTS2.pdf.

[9] Yurii Khomskii. *Infinite Games*. 2010. URL: https://www.math.uni-hamburg.de/home/khomskii/infinitegames2010/Infinite%20Games%20Sofia.pdf.

[10] Szepesvári Kocsis L. "Bandit Based Monte-Carlo Planning". In: *Lecture Notes in Computer Science Vol. 4212* (2006). DOI: https://doi.org/10.1007/11871842_29.

[11] Paul Fischer Peter Auer Nicolò Cesa-Bianchi. "Finite-time Analysis of the Multiarmed Bandit Problem". In: *Machine Learning 47, 235-256* (2002). DOI: https://doi.org/10.1023/A:1013689704352.

[12] *Pygame Wiki*. URL: https://www.pygame.org/wiki/about.

[13] J. Schaeffer. "The history heuristic and alpha-beta search enhancements in practice". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 11.11 (1989), pp. 1203–1212. DOI: 10.1109/34.42858.

[14] Hubert T. et al. Schrittwieser J. Antonoglou I. "Mastering Atari, Go, chess and shogi by planning with a learned model". In: *Nature 588, 604–609* (2020). URL: https://www.nature.com/articles/s41586-020-03051-4.

[15] Maddison C. et al. Silver D. Huang A. "Mastering the game of Go with deep neural networks and tree search". In: *Nature 529* (2016). DOI: https://doi.org/10.1038/nature16961.

[16] Maciej Świechowski and Jacek Mańdziuk. "Self-Adaptation of Playing Strategies in General Game Playing". In: *IEEE Transactions on Computational Intelligence and AI in Games* 6.4 (2014), pp. 367–381. DOI: 10.1109/TCIAIG.2013.2275163.

[17] David Silver Sylvain Gelly. "Monte-Carlo tree search and rapid action value estimation in computer Go". In: *Artificial Intelligence Volume 175* (2011). DOI: https://doi.org/10.1016/j.artint.2011.03.007.

[18] The Cognition Team. "SWE-bench technical report". In: (2024). URL: https://www.cognition-labs.com/post/swe-bench-technical-report.

[19] Kevin Hanbo Zhao. "Regret-Based Algorithms for Multi-Armed Bandits". In: *Bachelor's Thesis* (2020). URL: https://nrs.harvard.edu/URN-3:HUL.INSTREPOS:37364663.

# 11 Appendix

**Word count excl. Diary transcript, captions, figures, tables and code is approximately 12100**

## 11.1 User Manual

The setup to run the code for this project is fairly simple. However, there are some package that need to be installed in order for the program to run properly. To do this, you will first have to install pip, a Python package manager. Pip is automatically installed if you have downloaded Python, but if not, it can be installed using the following command:

```
py -m ensurepip --upgrade
```

Once pip has been installed, there are a few packages that are required to be installed, namely pygame, pygame-menu and PyAutoGUI, which can easily be installed through the following commands:

```
pip install pygame
pip install pygame-menu
pip install pyautogui
```

Once installed, the code can be ran simply by running the main class.

```
python py/main.py
```

## 11.2 Interim Reflection

### 11.2.1 Original Term 1 Plan

Following is the original plan that was made at the beginning of term. I have shortened the plan to the weeks that are relevant here, being term one. A reflection of the original plan in comparison to current progress, as well as an updated plan for the second term can be found in Section 7.

- Week 1 - Research MCTS
- Week 2-3 - Develop UCB1 method for bandits
- Week 4-5 - Code playable Go program
- Week 6-7 - Have 'pure' MCTS execute random rollouts
- Week 8-9 - Attempt to improve UCB1 with RAVE
- Week 10 - Compare performance of UCB1 with RAVE
- Week 11 - Prepare for report and presentation

### 11.2.2 Term 1 Reflection

Reflecting on the plan, it is clear that the volume of functionality that I wanted to achieve by the end of term was not met. I believe that the perceived difficulty of certain aspects such as implementing MCTS for Go as well as comparing standard UCT with UCT-RAVE were beyond the scope of the time frame for the first half of the project. Despite this, my expanded knowledge through working on the core elements of the project and research will allow me to refine my plans for the next term.

### 11.2.3 Original Term 2 Plan

For the second term, my first course of action will be to implement MCTS with UCT as fast as possible. This will allow me to have some form of AI game playing no matter how unrefined it may be. From here I can begin enhancing the code so that it runs more efficiently, starting with implementing UCT-RAVE. Furthermore, I'd like to look more into heuristics for the selection and simulation phases for MCTS, specifically since RAVE has a close relationship with the

*History Heuristic* used in alpha-beta search for minimax trees[13]. I plan to research further into this during the winter break, which will allow me to begin the second term with a clear mind for what heuristics I'd like to experiment with. Furthermore, I have interest in the concept of *Tree Parallelisation* in the context of Computer Go, which involves running MCTS through multiple threads[8]. With this in mind, below is an updated Project Plan for the second term.

- Winter break - Explore more heuristic concepts that can be applied
- Week 1-2 - Implement MCTS with UCT for 9x9 Go
- Week 3 - Improve GUI to showcase and log considered moves
- Week 4-5 - Improve UCT with UCT-RAVE
- Week 6-7 - Experiment with MCTS running in multiple threads
- Week 8-9 - Adopt an optional heuristic for simulation (so it isn't just random play)
- Week 10 - Test algorithm performance against other Go playing algorithms and record results
- Week 11 - Prepare for final report and code submission

## 11.3 Diary Transcript

At the moment, this diary is being upheld physically and my original plan was to keep it that way, but I believe it would be best to include a transcript of what is covered, as well as to update this digital log whenever the physical book is updated.

In here contains key developments in code, relevant equations as well as a demonstration of how my thinking and strategies changed throughout the project.

Week 9th - 15th October 2023
Did lots of research on the UCB1 algorithm and found some good resources and visualisations on how UCB1 works.
There appears to be slight variations in the equations, such as UCB-Normal, and also Epsilon-Greedy. The process of UCB1 involves maximising a value that acts to balance exploitation with exploration.

The usual UCB1 equation runs as follows:

$$\overline{x}_i + \frac{\sqrt{2 \ln t}}{n_{i,t}} \tag{5}$$

- $\overline{x}_i$ is the estimated mean of machine $i$
- $t$ is the current time step

- $n_{i,t}$ is the number of times machine $i$ has been pulled at time step $t$

Week 16th - 22nd October 2023
Focused on setting up my work environment and getting some code/tests down for the UCB1 algorithm. I've decided to use Python for the project, and the standard unittest module for testing my programs. Later I will have to utilise other modules in order to express things graphically (especially for the game of Go).

Week 23rd - 29th October 2023
Wrote the majority of the UCB1 code during this week and confirmed that it works. I've noticed some interesting potential for the things that I can do with it, such as creating logging files to store results. Something that I believe would be useful for the future is to actually show the effectivity of the algorithm would be to add an input 'balance' and observe how it changes in UCB1 vs. naive strategies such as just picking on random bandit at a time.

Week 30th October - 5th November 2023
Started work on the Go game in Python without any AI moves. I'm using PyGame as a library which allows me to visualise the game. I'm vaguely familiar with the library but will definitely be reading documentation as I go on the functions that are available. In terms of planned code structure, I know that at the very least I will need a class to represent the board as well as a class to represent stone objects that exist on the board. I've also added a constants file that I'll use to track key values in the game and will import where needed.

Week 6th - 12th November 2023
This week began with me drawing the board on the screen and overall cultivating the board class. I've got the board pasting in the centre of the screen and it's the correct size. Thinking about the representation of the board in terms of backend, I thought it would be best to use a two-dimensional array to represent the stones and their respective coordinates on the board. I've also created the class to represent stone objects as well as initialise it with a row, column and its colour. I found some images online for the two stones that I will paste to the screen when they are placed.

Week 13th - 19th November 2023
For this week, I decided to devote a lot of time to the development of Go to ensure that a working two-player game is finalised in time for the interim report. For this reason, I will split this journal entry into the respective days where I made significant updates on the codebase.

November 13th, 2023
To understand the mechanics behind how PyGame prints objects to the screen, I wrote a method 'spawn_stone' that also takes the mouse position as a parameter in where the stone is placed. This allowed me to adjust the image sizes to

appear on the board smoothly.

November 15th, 2023
I fixed a front-end code smell. Before, I was referencing the entire PyGame window as part of the matrix in which Go will be played on. This is an issue as players should not be able to place stones outside the board, which is something I found could happen with the current implementation. I decided to create a fixed size as I know the window will always be 900x900 (that was my decision). I made the board 720x720, and the square sizes 90x90. This has many benefits, such as allowing me to expand the window in the future without interfering with the board.

November 16th, 2023
I began to think about what kind of information is necessary for a stone object. Naturally in the game of Go, each stone is either black or white, so I must include the colour. As it's a 9x9 grid, the stone's relative position is also important. I also included the board as I need a way to connect the stone and the board objects together. I then created the neighbours method, which very simply returns the adjacent coordinates of a stone's coordinates and pops them if out of range of the board. This is extremely important as Go play relies heavily on the interaction between stones and their adjacent neighbours.

November 17th, 2023
I created more tests for the neighbours method that prompted me to alter my code. Now that I am working strictly on backend, it is far easier to develop using TDD. Corrected what I believe to be a naive approach and a large error made on my behalf in regards to system design. Rather than having a board object with 81 arrays in two dimensions, I've edited the structure to follow a one-dimensional array of groups that contain stone objects. So long as each stone has the necessary information regarding its location and colour, there is no need for the former structure. Added a method that searches through a list of groups and checks whether a stone exists a specified coordinate. This was mainly to verify the new board implementation was effective, but the functionality is also useful as it will allow me to identify stones in neighbouring positions later on.

November 18th, 2023
Created a method to update and keep track of the turn in the game. Natural rules of Go imply that black goes first, then white and alternating from then on, which makes things very simple. Previously, I created the find_stone method to identify stones that exist at specific coordinates. I realised it would be rather ineffective to loop through what the neighbours method returns and calling find_stone for each one, so I updated it to take a list of coordinates rather than just one (although it does work with calling just one coordinate). Since I had find_stone implemented, it was fairly simple to update.

November 20th, 2023
Next, I needed a method that assigns a group for stones (or creates one) when the stone object is initialised. To begin slowly, I first assigned each stone to their own respective group regardless of location. I had to do some test refactoring as beforehand I was manually creating groups. I expanded on this method by allowing stones that are neighbours to others while having the same colour form a group of multiple stones rather than creating an entirely new group.

November 22nd, 2023
I created a method to calculate the 'liberties' of a stone. Liberties are essentially adjacent positions that are not occupied by stones. I want to have a facade class that interacts with the front end of the program, so I created a class to inherit the methods of a stone with some extra methods. I also made a method that returns the image associated with the colour of the stone object so it can be pasted on the board.

November 23rd, 2023
Despite wanting a facade class, I found some implementation issues regarding the image not appearing on the PyGame window. To circumvent this, I opted for including an optional parameter in the constructor of the stone class that defaults to 'None', allowing me to create stones that exist on and off the board.

November 25th, 2023
Created the method to merge groups together. I will then use this in scenarios where stones are placed in between groups.

November 26th, 2023
Changed find_group() method to work with the new ability to merge groups so that stones that have more than one adjacent neighbour of the same colour will merge the two groups together into one large group.

November 27th, 2023
Created a method that will update the groups on the board every time a new stone is placed. If a group is updated and found to have 0 liberties, it will be removed from the board. To do this, I had to override delete methods so that individual stones would also be deleted when a group is deleted.

November 28th, 2023
Added a method that updates the visuals of the game when a new stone is placed and groups are updated. Now that I have the application working at runtime, I found a few bugs that showed there were some errors in my tests, so I fixed them.

December 1st, 2023
Fixed a bug that caused the application to crash. At first I was unaware of the reason this was happening, but soon found it that it was when square shapes were being made. This was because a newly placed stone would have two neigh-

bouring stones of the same colour but they would also be from the same group. Due to my merge implementation the program would attempt to merge the same group object onto itself which would cause the crash, which I mitigated by checking first if a group has already been added to the list of groups to merge before attempting to perform the merge.

December 5th, 2023
Split the Go program into the MVC model to make my code cleaner, which was done by inheriting the stone and board class into respective views that then communicate with the main class.

December 6th, 2023
Fleshed out better experiments for the bandit processes and created a new branch to store them collectively. Implemented $\epsilon$-Greedy with a naive strategy to see how it compares to UCB1. I also worked on the UML diagram for Go to be put in the interim report.

Week January 14th - 21st
Began making plans to get 9x9 Go ready for simulation as more methods need to be created. I also fixed a bug where stones were allowed to be placed with the scroll wheel.

Week January 29th - February 4th
Made some GUI improvements and incorporated a points system along with passing, in order for games to be finalised as well as scored.

Week February 5th - 11th
Created methods necessary for players to forfeit the game through a double pass, and began work on the node class for MCTS. I'm planning on creating nodes that will be linked together and placed into a MCTS tree where utility functions can be used on them.

Week February 12th - 18th
Finalise the node class to include a calculation method for the unique UCB1 score of each node to be used later for traversal.

Week February 19th - 25th
Created the MCTS utility class and finished the traverse method. Begun work on the rollout method.

Week February 26th - March 3rd
Fully implemented the rollout method and conducted tests on it. Made a try-catch block to catch when a node's actions reached zero.

Week March 11th - 17th
Implemented the expand method as well as refactored the rollout method to

return a victor for backpropagation.

Week March 25th - 31st
Refactored and finalised the expand method and created the functionality for the backpropagation stage of the algorithm.

Week April 1st - April 7th
Produced a method that ran the entire MCTS process, created a menu to allow the player to choose between PVP and AI, and further refactored and optimised MCTS to improve the way it runs.