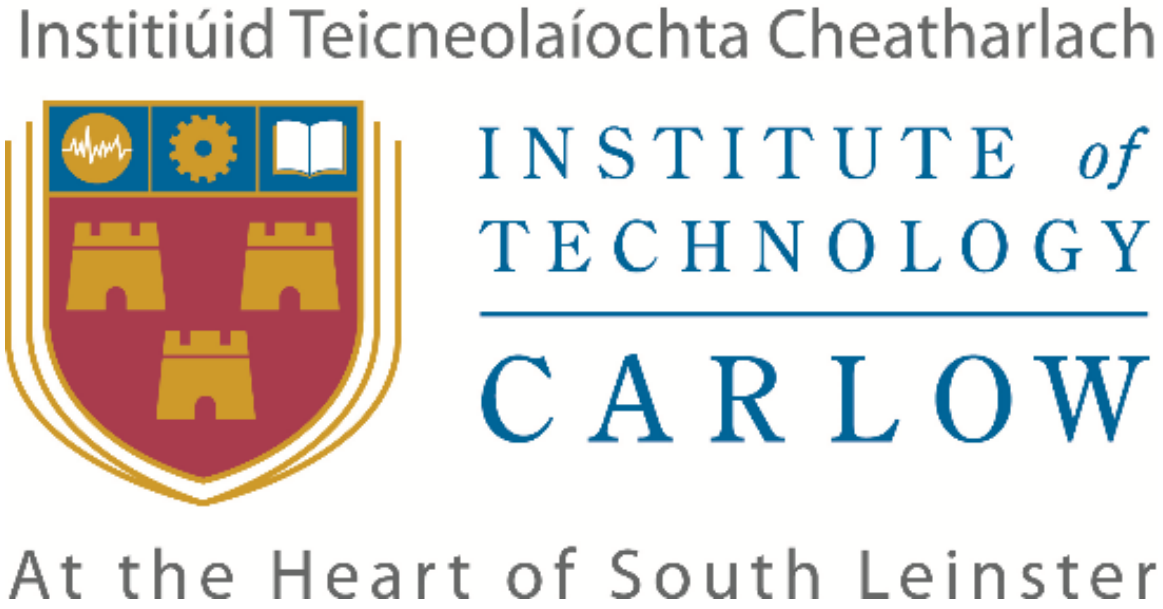
Computer Games Development
Project Report
Year IV

Jack Malone
C00236428

April 24, 2022



Contents

Acknowledgments

Abstract

The problem domain that I have chosen for this project is to be able to have an Artificial Intelligence be able to play the game of go or any other similar problem that requires the AI to search through a large search space to be able to figure out the best solution to the problem.

For this project, I would like to try and implement a version of Monte Carlo tree search such that I can create an AI that is able to efficiently solve a board state of the game of go. I would also like to test my application against a simpler algorithm such as MiniMax to compare how much quicker it is able to solve a position and against a stronger AI as well to see the difference between using it by itself as well with other algorithms that would help the tree search to find even better moves.

I might also try and research the use of neural networks alongside the Monte Carlo technique which aid the tree search in being able to find even better moves in the allotted time that the algorithm has to be able to find the optimal move. I'm interested in this research topic as I have an interest in different types of board games as well as how to implement AI that could be able to play these games so I would like to be able to test out different implementations of AI to see how well each type is able to play.

Chapter 1

Introduction

1.1 Basics of go

Go is a two-player board game in which the aim is to surround more territory than the opponent[14]. The game of go is usually played on a board that is 19x19 in size but can also be played on different sized smaller boards such as 9x9 or 13x13. For a 19x19 board, there is approximately 250 moves each turn that either player can play. If a game continues for 150 turns(approximate turn count), then there would be 250^{150} possible moves. Given the possible moves that each player can make per turn, it quickly becomes too much to be able to search through. If we use a game tree to describe how a normal game of go could develop, we would use the 250 possible moves as the branching factor[1], and use the formula b^d where b is the branching factor and d is the required depth[2] to find how many leaf nodes there are or how many different possible board states there could be[13].

1.2 Basics of Monte Carlo Tree Search

In 2006, Rémi Coulom described the application of the Monte Carlo method, which uses random sampling for deterministic problems which are difficult to solve with other approaches, to game-tree search and coined the term Monte Carlo tree search[15]. Before this was used, go playing programs used different techniques that were more similar to those used in games such as chess but are less suitable to the more complex game of go. Alpha-go was then introduced which uses two different neural networks which were able to beat world Go champion Lee Sedol in 2016 in South Korea with a lead of 4 games to 1[9]. It was first introduced in October of 2015 where it played its first match against European champion Fan Hui, with a score of 5-0. The documentary covering this story can be found here.

Alpha go first learned how to play go by learning from amateur games so that it knew how human players thought about the game through supervised learning. Alpha go was then retrained from games of self play through reinforcement learning which lead to the creation of Alpha zero.

The number of possible games in go increases exponentially as the game goes on. For example, on move 1 there are 250 possible moves, by move 2 there are 62,500 and by move 10 there are 953,674,316,406,250,000,000,000 possible moves.

[1] how many child nodes each current state has
[2] how many moves deep we are in to the search

1.3 Research Questions

There are a few different types of questions that I find could be interesting to look at when trying to implement the algorithm as efficiently as possible:

- How well does the Monte Carlo tree search do when compared with other more naive approaches to implementing an algorithm to solve the game of go.
- What other algorithms can we use alongside the algorithm to improve the speed and quality of move given.
- How well can alpha beta minimax solve the game of go on different sizes of boards and how this compares to Monte Carlo Tree search.
- What basic machine learning networks can be used instead of other algorithms.

Chapter 2

Literature Review

2.1 Basic Game Trees

2.1.1 Game Trees

Each node of a game tree represents a particular position or state in a game[4]. Whenever a player makes a move, such as placing a piece in go, this move will make a transition to one of the child nodes from the current state node. This is similar to decision trees where nodes with no children are referred to as leaf nodes.

The Monte Carlo Tree search is a smarter kind of search when compared against an *uninformed search*.

2.1.2 Uninformed Search

An uninformed search searches through the search space without any knowledge of what the goal state is. Two examples of uninformed searches are **depth-first search**[1] and **breadth-first search**[2].

2.1.3 Mini-max

Mini-max is a decision making algorithm which is usually used in a two player, turn based game[1]. The algorithm tries to find the best next move in the game. In its implementation, one player is the minimiser and the other player is the maximiser. If the evaluation of the current board state in the game is stored as a number, the maximiser will try to go to a gamestate with the maximum score and the minimiser will try to get a game with the lowest score possible. The algorithm is based on the *zero sum game* concept where a utility score is shared between the players and as a result an increase for one player (increased chance of winning) results in the decrease in the score for the opposing player (increased chance of losing). Two assumptions of the game is that each player is playing optimally so that they will usually try and pick the best move possible for them. Also the game should not have an element of chance[5]. The algorithm takes into account three basic functions: *Maximise* and *Minimise*, as well as a *Utility Calculation*.

[1]This algorithm starts at the root note and explores as far as possible along each branch before backtracking

[2]This algorithm starts at the root node and explores all nodes at the current depth before looking at all of the nodes at the next depth level

Possible implementation of the minimum and maximum functions for Minimax.

```
def Max(board):
    #returns a board configuration and its utility

    if board.terminal_state() or reached_depth_limit:
        return None, calculate_utility(board)
    max_utility = -infinity
    move_with_maximum_utility = None

    for move_possibility in board.children:
        (_, board) = Min(move_possibility)

        if utility > max_utility:
            move_with_maximum_utility = move_possibility
            max_utility = utility
    return move_with_maximum_utility, max_utility

def Min(board):
    #returns a board configuration and its utility

    if board.terminal_state() or reached_depth_limit:
        return None, calculate_utility(board)
    minimum_utility = infinity
    move_with_minimum_utility = None

    for move_possibility in board.children:
        (_, board) = Max(move_possibility)

        if utility < minimum_utility:
            move_with_minimum_utility = move_possibility
            minimum_utility = utility
    return move_with_minimum_utility, minimum_utility
```

2.2 Monte Carlo and Alpha Go

2.2.1 Multi Armed Bandit Problem

The multi-armed bandit problem is a problem in which a fixed limited set of resources must be allocated between alternative choices that maximise their respective gains, without full knowledge of all of the choices, and may become better known over time as it is looked at or by allocating resources to the choice[16]. In the stochastic problem, the rewards from each arm are from a probability distribution specific to that arm[3]. In the problem, there is a trade off between exploration and exploitation[3].

Maths behind the Multi Armed Bandit Problem

K is the total number of arms and T is the total number of rounds/moves. Both of these are known. Arms or branches are shown by $a \in [K]$, rounds by $t \in [T]$. The reward for a specific arm a is D_a, which is supported on $[0, 1]$. The expected reward is denoted by $\mu(a) := \int_0^1 x, dD_a(x)$. The best expected reward is denoted by $\mu^* := max_{a \in [K]\mu}(a)$, and the best arm is $a^* = argmax_{a \in [K]\mu}(a)$. The cumulative regret in round t is defined as

$$R(t) = \mu^* t - \sum s = 1^{t\mu}(a_s)$$

Where a_s is the chosen arm in round s. The goal of the algorithm is to minimise regret. To start searching through possible moves, you first start by exploring arms equally and pick an arm that is best for exploitation.

1. **Exploration phase:** try each arm N times. Let $\bar{\mu}(a)$ be the average reward for arm a.
2. **Exploitation Phase:** select arm $\hat{a} = argmax_{a \in [K]\bar{\mu}}(a)$. We then use this for all remaining rounds.

N is chosen in advance in relation to T and K. Other bounding functions can be found on `https://bochang.me/blog/posts/bandits/`.

2.2.2 Upper Confidence Bound Action

Exploration vs. Exploitation[10]

Definitions

- **Greedy Action:** When an agent chooses an action that currently has the largest estimated value. The agent exploits its current knowledge by choosing the greedy action.
- **Non-Greedy action:** When the agent does not choose the largest estimated value and sacrifice immediate reward hoping to gain more information about the other actions.
- **Exploration:** It allows the agent to improve its knowledge about each action. Hopefully leading to a long term benefit.
- **Exploitation:** It allows the agent to choose the greedy action to try to get the most reward for short term benefit. A pure greedy action selection can lead to sub optimal behaviour.

[3] In machine learning, exploration stands for the acquisition of new knowledge, and exploitation refers to an optimised decision based on existing knowledge.

Description

Upper confidence bound action selection: upper-confidence bound action selection uses uncertainty in the action-value estimates for balancing exploration and exploitation. Since there is inherent uncertainty in the accuracy of the action-value estimates when we use a sampled set of rewards thus UCB uses uncertainty in the estimates to drive exploitation.

$$A_t = argmax_a(Qt(a) + c\sqrt{\frac{ln(t)}{N_t(a)}})$$

Where t is equal to the time-steps and $N_t(a)$ is the number of times action a is taken. $Q_t(a)$ represents the current estimate for action a at time t. We select the action that has the highest estimated action-value plus the upper confidence bound exploration term.

2.2.3 Monte Carlo Tree Search

MCTS has two fundamental ideas: That the true value of an action can be approximated using random simulation and that these values may be used efficiently to adjust the policy towards a best first strategy. The algorithm progressively builds a partial game tree, guided by the results of previous exploration of that tree. Th tree estimates the values of moves and this becomes more accurate as more of the tree is built up. The algorithm consists of iteratively building a search tree until some predetermined time, memory or iteration constraint, at which point it returns the best move/root that it was able to find in that time. Each node of the tree represents a specific board state/ state of the domain, and child nodes are subsequent game states.

Four Stages of the algorithm as it looks for a move

1. *Selection:* In this step, we use tree policy to construct a path from the the root node (current board position) to the most promising leaf node. A leaf node is a node that has unexplored child nodes. The tree policy is an informed policy used for node selection in the explored part of the game tree. In this stage the algorithm also has to consider exploration vs. Exploitation, which can be solved by using an upper confidence bound algorithm. The algorithm keeps on selecting child nodes recursively through the tree until the most urgent expandable node is reached. A node is expandable if it represents a non-terminal state and has unvisited children.

2. *Expansion:* In this step, one or more child nodes are added to expand the tree randomly, according to the available actions.

3. *Simulation/Roll-out:* In this step, a simulation is run from the new node(s) with reward accumulated for each simulation. Roll-out policy is normally simply or even pure random such that it is fast to execute. A win could give a reward of +1, a draw 0, and a loss -1.

4. *Back-propagation:* In this step, the simulation result is backed up through the selected nodes to update their statistics. The back-propagation step does not use a policy itself, but updates node statistics that inform future tree policy decisions. The values of nodes are not updated during the roll-out step because we need to focus on the vicinity of the root node (snow-cap), based on which we need to make decisions of next moves. Whereas the values outside of snow-cap is not relevant for such decision, nor computationally efficient to store and calculate.

Pseudo-code for back-propagation

```
def run(node, num_rollout):
    #one iteration of select->expand->simulation backup
    path = select(node)
    leaf = path[-1]
    expand(leaf)
    reward = 0
    for i in range(num_rollout):
        reward += simulate(leaf)
    backup(path, reward)
```

2.2.4 Alpha-Go

The Training pipeline[7]

- *Supervised learning of policy networks:* The supervised learning policy network $\rho\_\sigma := \rho\_\sigma(a|s)$ is a 13-layer convolutional neural network trained on 30 million moves of human experts. The SL-policy network achieved 57% accuracy compared to the best accuracy of 44.4% by other research groups. Also trained a faster but less accurate roll-out policy ρ_π trained on a set of 8 million moves; this achieved an accuracy of 24.2%, using 2 rather than 3 ms so that approximately 1000 complete games could be simulated per second on each processing thread.

- *Reinforcement learning of policy networks:* The reinforcement learning policy aims to improve the SL-policy through self play. RL-policy network ρ_ρ has the same architecture as ρ_σ and initialised with the final weights of σ. This adjusts the policy towards the correct goal of winning games rather than maximising predictive accuracy. When played head-to-head the RL-policy network won more than 80% of games against the SL-policy network. Against the strongest open-source Go program Pachi, RL-policy network won 85% of games whereas the previous state-of-the-art program based on supervised learning of convolutional neural network won 11% of games against Pachi and 12% against a slightly weaker program, Fuego.

- *Reinforcement learning of value networks:* Value network $v\_\theta(s)$ approximates the optimal value function by approximating $\hat{v}\{\rho\_\rho\} \approx v^*(s)$. Trained on a large number of simulated games pitted against each other. The network was trained on 30 million moves sampled from distinct games of self-play by the RL-policy. The values produced by this network were consistently more accurate than values produced by Monte Carlo roll-outs using the fast roll-out policy ρ_π.

2.2.5 Extensions To Monte Carlo Tree Search

There are two extensions mentioned in the paper "Generalised Monte-Carlo Tree Search Extensions For General Game Playing"[6] that I would like to implement alongside the normal Monte Carlo tree search algorithm to see how they improve the efficiency of the algorithm. The extensions in this paper generalise how the algorithm gains knowledge so that knowledge about the specific domain that it is being used for is not required beforehand. The two extensions that I will be looking at are *Early Cutoffs* and *Unexplored Action Urgency*.

```
def early_cutoff(self):
    if not useEarlyCutoff:
        return False
    if playout_steps < minimum_steps:
        return False
    if self.is_goal_stable():
        return playour_steps >= cutoff
```

Listing 1: Pseudo Code for deciding cutoffs for the early cutoff extension

Early Cutoffs

Without having to encode a lot of domain knowledge into the Monte Carlo Tree search the search may end up using a lot of time and computation on specific paths in the simulation that end up being not useful in the play-out phase of the algorithm. If the simulations run for a long time, there is a possibility that a good move could be missed or that the algorithm will end up just going through the same sets of moves over and over. Instead, it would be better if we would be able to predict when going through a certain simulation if it is not worth continuing through it and using that time to look through a different simulation with a higher probability of generating better moves for us. There are also cases where if one of the players in the game has a big lead it might not be worth continuing to simulate the rest of the different possibilities as this outcome is unlikely to change.

This extension is used for two cases when simulations can be terminated early. The first case is when a simulation can be evaluated to a finalised score before getting to a terminal state. The other case is when the simulation has taken a certain amount of moves that are more than any actual game for the game that it is implemented in would actually take without reaching a terminal state.

Cutoffs should only be used in applicable games. This can be figured out by observing normal simulations of the game and getting information on the results from the games such as the depth of terminal states and if it needs naturally terminated simulations. It is also required that it is also possible to evaluate a specific state accurately or if the current state does not accurately correlate to the terminal state. This can be done by using an algorithm that estimates the score for each player at the current board state. We then check the stability of the game. Stability is based on the assumption that if a value calculated from a state changes gradually throughout the course of the game it is in some way correlated with what is actually happening in the game as it progresses. If it varies widely as the game progresses then it is unrelated to the game progression. To calculate the stability value the variance of the changes in score obtained from the goal relation is observed throughout the simulations and then averaged over them. If the resulting stability value is lower than a predetermined threshold but greater than 0 then this score is marked as stable and the extension is used.

Unexplored Action Urgency

UCT is commonplace in the selection phase to control the exploration/exploitation balance in MCTS. Usually UCT never exploits on the edges of the game tree, it always selects amongst the unexplored actions available. It is not until all unexplored actions have been looked at that attention is given to exploitation in the state. If domain specific knowledge is available unexplored actions can be evaluated and bad ones can be ignored. However, in Go, even with domain specific knowledge it may be difficult to determine a bad move straight away. If the knowledge is accurate however, the bad moves can be ignored.

```
def unexplored_action_urgency():
    if state.explored == 0:
        return playout_strategy(state.unexplored)
    action = selection_strategy(state.explored)
    if state.unexplored == 0:
        return action
    exploit = action.uct_value
    discount = state.unexplored_size / len(state.actions)
    urgency = 50 + C_p * sqrt(ln(state.visits())) * discounts
    if exploit >= urgency:
        return action
    else:
        return playout_strategy(state.unexplored)
```

Listing 2: Pseudo Code for unexplored action urgency extension

With this extension it is possible for the agent to exploit actions on the edge of the game tree if there is a reason to do so.With this extension it is possible for the agent to exploit actions on the edge of the game tree if there is a reason to do so. Without domain specific knowledge, we are still able to look at how the rewards of an action accumulate. If the reward is consistently very good the action estimated average will stay high no matter how often we select. We will prefer this action over any unexplored action.

Without domain specific knowledge, we are still able to look at how the rewards of an action accumulate. If the reward is consistently very good the action estimated average will stay high no matter how often we select. We will prefer this action over any unexplored action found in the state while the number of simulations traversing the state is low. Instead of the default action where unexplored actions take priority a quick decision is added when the search is on the edges of the tree. The decision of selecting an unexplored action is given a value that can be compared with the highest UCT value of the already explored actions and the higher value is taken.

2.2.6 Research papers

A good research paper that I have found that is similar to the research topic that I want to learn about is "Monte Carlo Tree Search: A Review of Recent Modifications and Applications"[8] which gives an outline of how the Monte Carlo Tree Search works as well as how it is used with machine learning and implemented for specific use cases.

Another research paper that I have found that gives an outline of Monte Carlo Tree Search is "A survey of Monte Carlo tree search methods"[2] which gives an outline of the different parts inside of the algorithm as well as how it can be used in different types of games including go but also certain connection games and different combinatorial games such as *Clobber* and *Othello*. Pseudo Code for unexplored action urgency extension

With this extension it is possible for the agent to exploit actions on the edge of the game tree if there is a reason to do so. With this extension it is possible for the agent to exploit actions on the edge of the game tree if there is a reason to do so. Without domain specific knowledge, we are still able to look at how the rewards of an action accumulate.

2.2.7 Zobrist Hashing

Zobrist hashing, names after its inventor Albert Zobrist, is a technique to represent game board positions, from games like chess or Go, as a hash value. It is mainly used with transposition tables, a special kind of hash table that is indexed by a board position and used to avoid analysing the same board position more than once.[12] The main purpose of Zobrist hash codes in board game programming is to get an almost unique index number for any board position, with a very important requirement that two similar positions generate entirely different indices. These index numbers are used for faster and more space efficient hash tables or databases.

2.2.8 Machine Learning

Reinforcement Learning

Reinforcement learning is the training of machine learning models to make a sequence of decisions. It learns to make the best decisions by trying to maximise the reward that it receives after making a decision in a certain environment. In reinforcement learning an agent tries to manipulate the environment and travels from one state to another. The agent receives a reward on a success but does not receive any reward on failure. In this way the agent learns from the environment. Compared to supervised learning, where the model would be given a lot of direction in terms of getting labelled input as well as expected output, reinforcement learning receives less supervision which depends on the agent in determining the output. They must learn independently to discover the sequence of actions that maximise the reward. The quality of actions is measured by not just the immediate reward they return, but also the delayed reward they might get. As it can learn the actions that result in eventual success in an unseen environment without the help of a supervisor, reinforcement learning is a very powerful algorithm. The formal framework for Reinforcement Learning borrows from the problem of optimal control of Markov Decision Processes.[11]

Reinforcement Learning Terminologies

- **Agent:** The learner and the decision maker.
- **Environment:** The space where the agent is able to take actions and learns from them.
- **Action:** Some set of actions that an agent is able to take inside of an environment. These are usually somewhat limited to make it easier to go through what the agent is able to do and what they have to learn.
- **State:** A specific state of the agent in the environment. In Go, this will be after a certain move has been played or the starting board state.
- **Reward:** Any action that the agent does in a state will give them a reward which is usually a number, the higher the reward the better they did. **Policy:** The decision making function of the agent. **Value function:** The value of a state represents the long term reward achieved starting from that state and executing a particular strategy. **Temporal difference algorithms:** this is a class of learning methods, based on the idea of comparing temporally successive predictions. **Model:** this is the view that the agent has in to the environment. This maps state action pairs to probability distributions over states. Not all agents use a model of its environment.

Markov Decision Process

In mathematics, a Markov decision process is a discrete time stochastic control. It provides a mathematical framework for modelling decision making in situations where outcomes are partly random and partly under the control of a decision maker. At each time step, the process is in some state s, and the decision maker may choose any action a that is available in state s. The process responds at the next time step by randomly moving into a new state s', and giving the decision maker a corresponding reward $R_a(s, s')$. A Markov decision process is a 4-tuple (S, A, P_a, R_a) where:

- **S** is a set of states called the *state* space.
- **A** is a set of actions called the action space.
- $P_a(s, s') = Pr(s_t + 1 = s'|s_t = s, a_t = a)$ is the probability that action a in state s at time t will lead to state s' at time $t + 1$.
- $R_a(s, s')$ is the immediate reward received after transitioning from state s to s', due to action a.

Q-Learning

Q learning is a model free reinforcement learning algorithm to learn the value of an action in a particular state. Q learning can identify an optimal action selection policy for any give Markov decision process, given enough exploration time and a partly random policy. When Q-learning is performed we create what's called a q-table or matrix that follows the shape of [state, action] and we initialise the values to zero. We then update and store the q-values after an episode. An episode is when you complete an environment once. This table becomes a reference table for the agent to select the best action based on the q-value. The next step is then for the agent to interact with the environment and make updates to the state action pairs in the q table Q[state, action]. The agent interacts with the environment using exploration and exploitation to make sure that it is exploring the environment but also when it finds a good action it uses it properly. Updating the q-table occurs after each action until a given episode is done. The final state for the game of go is when either both players decide to pass or when there is no legal moves to play, I have decided to use the no legal moves as a cutoff as well as a hard cutoff after a certain number of moves to simulate both players passing at the end of a game. The core of the algorithm can be described as the following:

$$Q^{new} \leftarrow (s_t, a_t) + \alpha * (r_t + \gamma * maxQ(s_t + 1, a) - Q(s_t, a_t))$$

Q learning variables

- **Learning rate:** determines to what extent newly required information overrides old information.
- **Discount Factor:** is used to balance immediate and future reward.
- **Reward:** is the value received after completing a certain action at a given state. A reward can happen at any given time step or only at the terminal state.

2.2.9 Deep Q learning

To calculate the TD error, we calculate the difference between the Q target and the current estimate of Q.

$$\Delta w = \alpha[(R + \gamma max_a \hat{Q}(s', a, w)) - \hat{Q}(s, a, w)]\nabla_w \hat{Q}(s, a, w)$$

As the real value for TD is not known, it needs to be estimated. Using the Bellman equation, the TD target is the reward of taking an action plus the discounted highest Q value for the next state.

$$Q(s, a) = r(s, a) + \gamma max_a Q(s', a)$$

Policy optimisation methods

In policy optimisation methods the agent learns the policy function directly that maps state to actions. Deterministic policy maps state to action without uncertainty when there is a deterministic environment.

Proximal Policy Optimisation (PPO)

PPO is a policy gradient method[4] with either discrete or continuous action spaces. It trains a stochastic policy in an on-policy way. The idea behind PPO is to avoid having too large of a policy update. To do that, a ratio is used that tells us the difference between the new policy and the old policy and this ration is usually between 0.8 and 1.2. It also introduces a new way of training an agent by running K epochs of gradient descent over a sampling mini batches.

[4]Policy gradient methods are a type of reinforcement learning techniques that rely upon optimising parameterised policies with respect to the expected return by gradient descent

Chapter 3

Results

3.0.1 Monte Carlo Tree Search vs. Alpha Beta

To contrast the two different algorithms that I decided to create, I saved data from multiple games played between the different algorithms. I then kept track of how many moves they were both able to calculate within the time limit given as well at the end of the game I also calculated the number of pieces captured as well as territory that each of the algorithms was able to get. This would then show which AI won the game as well as on average how well each algorithm did against each other.

From my findings in the Jupyter Notebook that I created was that the Monte Carlo Tree Search was able to search through a lot more moves both in the smaller boards as well as scaling up to the larger boards. It was able to reach its max exploration space when the board was 5x5, whereas the alpha beta search was not able to go through that many moves on the board and generally resulted in it picking non optimal moves. Unique to the 5x5 board, it appears as if the alpha beta algorithm actually ended up calculating less moves which makes sense as it has less moves that it needs to go through as there's only so many places you can play.

On the larger boards you can see that the alpha beta algorithm starts to calculate less and less moves as it has a larger search space to go through as it increases in the amount of moves as the games go on as the game progresses as they don't have to go through as many moves for each possible moves but it was not far enough through the game for the number of free spaces left on the board would reduce the amount of moves that it calculated.

For the 13x13 and 19x19 boards I was not able to go through as many moves in the game as it took a lot longer to generate so that the moves displayed are mostly from the first half or so of the game.

Results

From the data that I was able to generate for both the Alpha Beta algorithm as well as the Monte Carlo Tree Search, the Monte Carlo Algorithm was able to outperform the Alpha Beta algorithm on all of the board sizes that I created, however they were a lot closer on the 5x5 board as there are a lot less moves for the Alpha Beta algorithm to go through on this board. Looking at the graph for the number of moves the algorithm was able to process, the number of moves goes down as the the move number increases. This shows that at the start of the game the algorithm was going through all of the possible moves that it could and the reason that it is going through less towards the end of the game is that there was not as many legal moves that it could calculate. For the 13x13 board, the algorithm was only able to start processing more than a handful of moves towards the end of the game, again where there is a lot less possibilities for it to go through. Starting on this board the algorithm mostly just played on the first valid spot that it calculated as it did not have time to calculate more. It was not able to calculate any moves for the larger boards. Comparing this to the Monte Carlo Tree search, this algorithm was also able to go through its maximum amount of moves for the 5x5 board, but it started to have less moves for the 13x13 board and the 19x19 board. However, on these boards, even for the starting moves where it had to go through a lot more states, it was able to go through a lot more moves as it did not have to go and calculate all of the possible states that the board could be in, but only the ones that it thought would likely be chosen. From the graphs showing the amount of territory as well as captures that both of the algorithms had, there is a clear increase in the number of captures and territory that the Monte Carlo Tree search had compared to the Alpha Beta search. Monte Carlo seemed to prefer captures over territory where the Alpha Beta search seemed to prioritise the opposite.

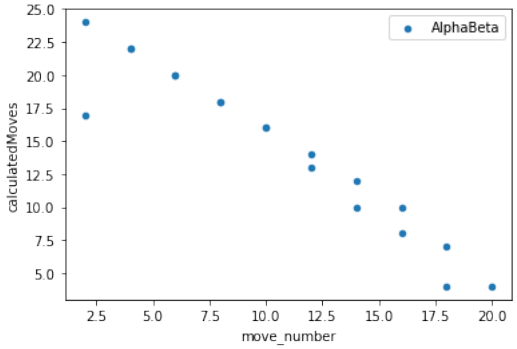


Figure 3.1: Total number of moves that the algorithm was able to calculate throughout a game on a 5x5 board

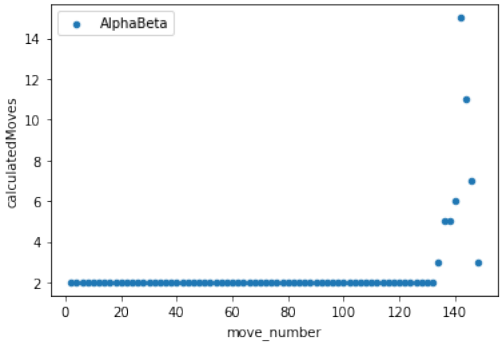


Figure 3.2: Total number of moves that the algorithm was able to calculate throughout a game on a 13x13 board

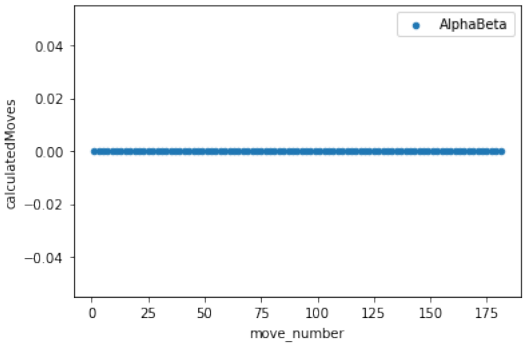


Figure 3.3: Total number of moves that the algorithm was able to calculate throughout a game on a 19x19 board

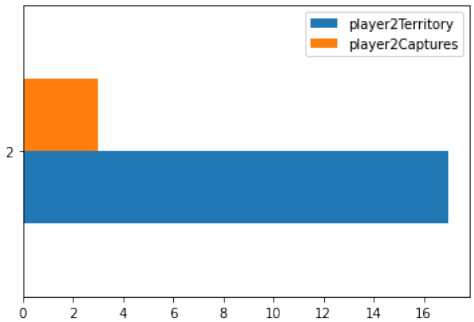


Figure 3.4: Total territory and captures that the Alpha Beta algorithm received on a 13x13 board

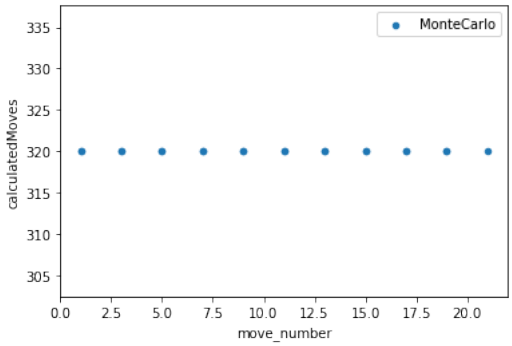


Figure 3.5: Total number of moves that the algorithm was able to calculate throughout a game on a 5x5 board

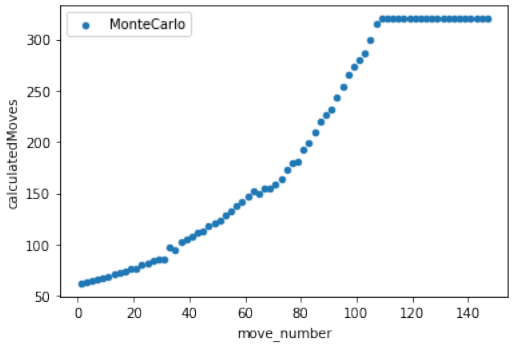


Figure 3.6: Total number of moves that the algorithm was able to calculate throughout a game on a 13x13 board

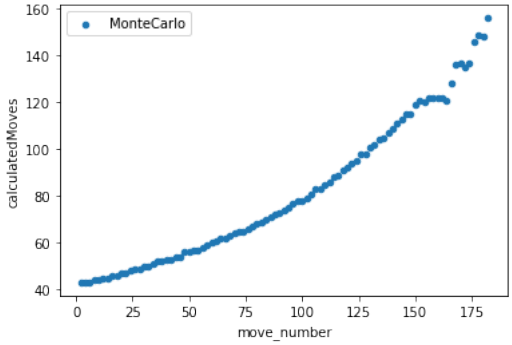


Figure 3.7: Total number of moves that the algorithm was able to calculate throughout a game on a 19x19 board

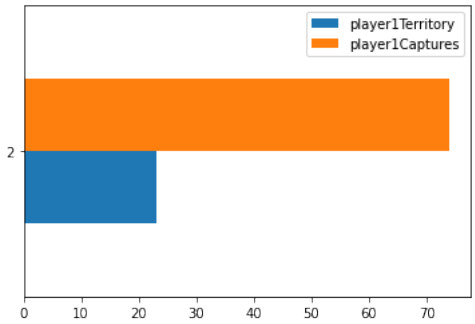


Figure 3.8: Total territory and captures that the Monte Carlo algorithm received on a 13x13 board

Proximal Policy Optimisation vs. Deep Q learning

Each graph for the rewards from deep Q learning show the average reward received from the model over an average of 100 games with the average score being on the y-axis and the game number on the x-axis. From the first game no proper reward was received over the games. This was a result of using a real reward system inside of the environment where the model would only receive a reward at the end of a game. As a result of this, they were not able to successfully figure out what moves were good or not as they were only likely to receive rewards after 400-500 moves if it was able to finish the game and then this would be likely to only give the reward for the last move instead of all of the moves leading up to it. This would result in that at the start of a game it would not be able to figure out what to do as it would not be able to back-propagate the reward to the very start of the game for a reward several hundred moves in the future. For the second attempt, I altered what reward system it would be using to gain reward to a heuristic method which would give reward based on the approximate result of the current board state so that it would be able to easier figure out if a move was worth taking or not. I also introduced a positive and negative reward based on whether or not it was able to play a valid or invalid move respectively. This resulted in a large variation in the reward given throughout the games that it had to learn resulting in the model not being able to fully learn what moves were good or not and it did not seem to have enough games to be able to pick a proper valid move. For the final training session, I decided to remove the negative rewards for playing an invalid move, so that it would just try and use the heuristic to figure out if a move was good or not. I also made it so that if the model played an invalid move, a random legal move would be chosen instead and then the reward would be given for this. The aim for this was that even if it did try and pick an invalid move, it would then have a proper move to use and hopefully after enough games it would be able to always pick a valid move. With this method towards the end there was a large variation in the rewards that it was getting for the moves, which would appear as if it was still picking invalid moves before it started to stabilise again towards a reward of 0. Overall, the deep q learning was not able to learn to be able to make legal moves in the starting setups for the game, this may be able to be changed if it was given a lot more games to be able to train in the game, but this would result in having to let the model train for several days, weeks and may not see great improvements. A better solution would likely to be to reduce the action space that the model was given so that it could only choose between the valid moves in the position. This would improve it dramatically in the algorithm being able to pick only valid moves but this might require changing how it is given the state when actually playing to only include valid moves which may retract from it learning to play valid moves on its own. The proximal policy optimisation algorithm(PPO) was able to start making valid moves fairly quickly and was given approximately 200,000 timesteps to train. This algorithm was able to pick valid moves a lot better than the deep q learning but it will probably need a lot more training to be able to reach a good level of playing against actual players. It was able to play similarly to how the Monte Carlo tree search played or how the alpha beta algorithm would be able to play given enough time, however once the PPO algorithm and deep q learning are trained they are able to give moves instantly instead of requiring time to figure them out.

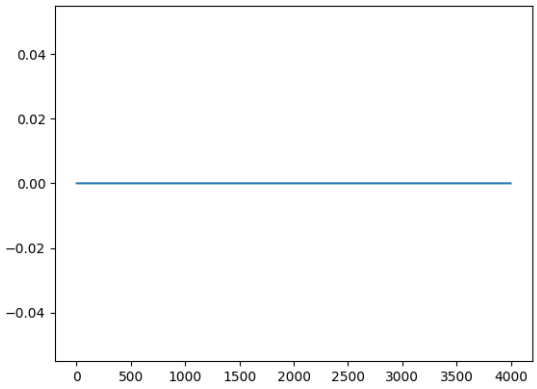


Figure 3.9: First attempt of training the deep q learning model

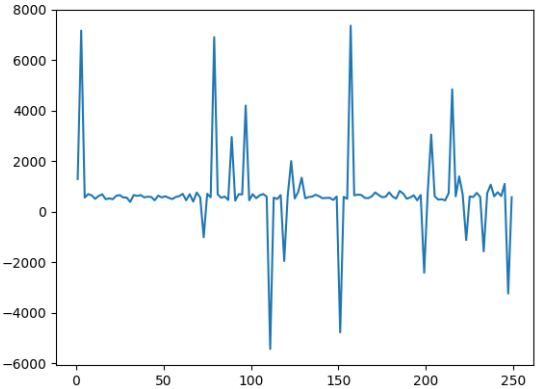


Figure 3.10: Updating reward for the model to reward legal moves more to attempt to get it to be able to make more legal moves

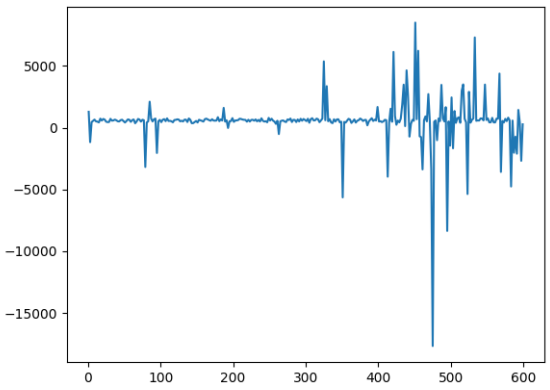


Figure 3.11: Final attempt of training the deep q learning model with 600 games at 500 steps per game

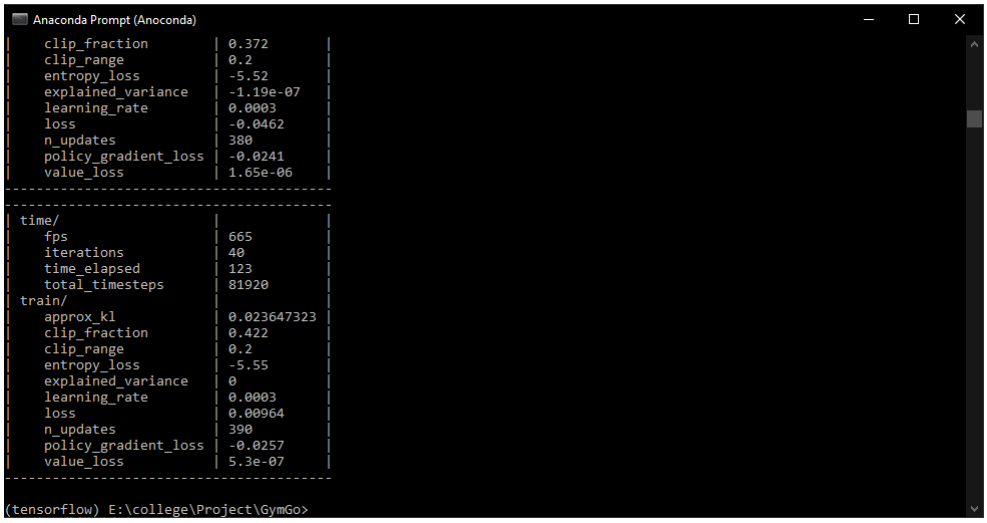


Figure 3.12: Result of training proximal policy optimisation model

Chapter 4

Project Milestones

4.0.1 Game

For the main game most of it was fairly on time. It took me a little bit longer to create the rules for the game than I expected so that took around one or 2 weeks longer than what I had expected. The main part of the rules that I had problems with was with making sure that when a player placed a piece that it would capture the opponents pieces correctly and if they captured an entire group then this group would be fully captured instead of just one or two pieces closest to where they placed down the piece. There was also the issue of if they placed somewhere that looked illegal to place but it would capture a group I had to allow it which took up some more time. The main piece that took the most time was for figuring out how to set up the game to keep track of Ko correctly. This is making sure that the same board positions cannot keep reoccurring in a row infinitely which usually happens when a piece is captured and the opponent can then capture the piece was placed down which could lead to infinite loops in some configurations. I got around this by setting a spot that could cause the issue to Ko so that they could not place at this position for the next move and then it would be removed again. The first part with the board for setting up the display as well as setting up user input was on time for making a player able to place pieces as well as this switching who's turn it was.

4.0.2 Algorithms

There was some parts of the research that I decided not to follow fully so that I could focus on other parts of the research. After completing the main Monte Carlo Tree Search algorithm as well as the Alpha Beta Algorithm I realised with the differences already apparent between the two that adding more extensions to the Monte Carlo Tree Search algorithm might be hard to be able to quantify in the data analytics as there was already a gap between the two algorithms. I also did not look at Zobrist Hashing much as for the smaller boards the Alpha Beta algorithm was already able to play reasonably well but for the bigger boards it would not give enough performance gain for it to be able to reasonably make a decision as to what move to pick. In terms of keeping on track for the time, I was able to get most of the Monte Carlo Tree Search done mostly in the time frame that I had given myself. I did have some issues with it in terms of the states being copied over incorrectly which took me a week or two to figure out which slightly slowed me down.

I had previously decided on working on two extensions for the Monte Carlo Tree Search algorithm as well as zobrist hashing for the Alpha Beta Tree Search. Towards the end of the project after I had finished writing both of the algorithms, I decided to change what I was working on and to

do something else so I did not end up doing any of these. I realised that most of these probably would have been fairly small changes to the algorithms and probably would have not aided greatly in comparing the two algorithms. I also had a decent amount of time left and I decided to work on something more challenging for the rest of the time that I did have. Instead I decided to work on the proximal policy (PPO) algorithm as well as deep Q learning. This involved finding a suitable environment from online that I would be able to use to train them. Finding this as well as setting it up to work properly with the algorithms took a little bit longer than expected as there was a few things inside of the environment that I had to change to get it working the way that I wanted it. This took a few days to do. After I was able to fix the issues with the environment itself, it did not take me too long to properly research as well as implement the PPO algorithm. After I got that implemented I focused on trying to get the Deep Q learning algorithm implemented. This took around a week and a half for me to be able to implement, after which I had a model that was able to train on the environment. However the model took a long time to train and it seems as if it was not able to make legal moves most of the time so the training of the model did not turn out too well but it might be able to be improved given more time to train and a better reward model for the game. I might also be able to improve how often the model remembers certain things with retraining and the learning rate as to get it to try out new moves instead of only doing the same moves.

4.0.3 Major Technical Achievements

The main technical achievement was getting the Monte Carlo algorithm to work fully with my implementation of the board game. It took me a while to both get the game fully working at a stage where the algorithm would be able to properly get the information that it needed from all of the states as well as for making sure that the algorithm could quickly enough create copies of the board state so that it would be able to check whether or not a move was a good one to pick. There was also some issues with making sure that the nodes for the algorithm were working correctly so that they would properly be stored in the root node for when the algorithm finishes to then be able to go through all of them to figure out what the best move was. I was also quite happy with how the interaction between the algorithm and the data collection was handled as it allowed me to be able to easily compare the two algorithms against each other in a jupyter notebook as well as look at the data and go through it in a database viewer.

Even though I was not able to get a fully working model with deep q learning, it was still one of the more difficult parts of the actual project and I think it still turned out well in the time that it had to work. I was able to also work with the environment that I found to set it up so that I would be able to use it more easily.

4.0.4 Project Review

I think that the main part of this project went pretty well in that I was able to set up the project so that I was easily able to compare the two main algorithms that I wanted to compare at the start of the project. I also got more work done on the data collection side of the project with being able to get a jupyter notebook set up so that I would be able to create graphs for all of the data that I had captured. This helped a lot with being able to create the final report as I was able to get proper data to be able to review at the end of the project instead of having to manually track both of the algorithms. The extensions of the algorithms that I had planned on doing at the start of the project did not end up going into the project at all in the end because I had other things that I wanted to add in to the project such as reinforcement learning that I though would be better suited to the project. If I was starting again I probably would have decided on doing the reinforcement learning a

lot sooner than I did instead of the extensions as this could have given me a bit more time to work on these algorithms and maybe even get one other reinforcement learning model into the project. The deep q learning worked okay in terms of me being able to try and train it, but I think if I was to start again, I would pick a different type of machine learning to do such as Supervised Learning as I think the Deep Q learning would require a lot more training than I was able to give or some extra information during training to really be able to work well. I think Supervised Learning could be a good fit if you were able to find a good set of data online that you were able to give rewards to which could result in a well trained model. However, this data set would have to be quite large to be able to be used effectively as well as being from good players so that the model would not pick up mistakes. I think python was a good choice in terms of being able to use it for the jupyter notebook as well as for the reinforcement learning models, however if I was starting again I might change to creating the game inside of Godot or another engine. Pygame was fairly easy to set up for creating the game, but I think an engine would definitely help in terms of speeding up how quickly I would be able to create the base game for it as well in that it would have better support for a good ui. The ui that I ended up going with was using an older library add on for pygame which was a bit more difficult to use, I would prefer to use a better library in the future or to just use the ui libraries that a game engine provides.

Chapter 5

Conclusion

Compared to Alpha Beta Tree Search, the Monte Carlo Tree search is a great improvement in terms of calculation time as well as moves calculated and the quality of the move generated. In a time constrained or memory constrained environment, Monte Carlo Tree Search can far outperform Alpha Beta for environments where there is a large search space. This is as a result of the amount of processing that the algorithm needs compared to Alpha Beta to get a similar result but without having to search the entire tree to find the best possible move, instead it is focused on finding a move that is the best from what it has found so far. The algorithm also works better when it is cut off early as a result of time constraints, even if the algorithm was not able to be run to conclusion, it is still able to produce a reasonable result, even if it is not the best possible. When Alpha Beta is completed early however, we do not have a guarantee on whether or not it has found a good move as it is limited to linearly searching through the search space.

The reinforcement learning models that I used in my research, were able to reproduce similar results as the other algorithms, but with the added benefit as that as they are playing they use up a lot less time in calculating a move as well as not having to use as much computer resources. Hopefully in the future, the deep q learning model that I created can be extended to improve its decision making to be able to choose more valid moves as well as being able to make human like moves if supervised learning was also used to create a model instead of only using reinforcement learning.

Chapter 6

Future Work

6.0.1 Different Machine Learning techniques

One next step could be to further compare different types of machine learning against each other for trying to solve the game of go. One area that would probably be best to focus on would be supervised learning. A component ai could be created given a good data set to train the model on, however this could take a while to train the agent so some way of speeding this training up using multiple pcs or a different technique to use alongside the supervised learning would be needed. This would be similar to some of the ai that AlphaGo created as they were creating their neural network where they used a large number of professional games to try and create an ai that could play the game as well. Other games could also be chosen to see how these algorithms could be used in those games. Chess may be an easier game to create an AI for as the search space is a lot smaller, however it could take a bit longer to actually implement the game itself into an environment that you could use unless you use a pre-made one as the rules of Go are not as complex. For my implementation of the model, I did not use any external data apart from training it on playing the actual game, so giving the ai an opening book as well as puzzles as part of some kind of Supervised learning could improve the skill level of the ai that are generated as the skill level of the ai that I generated is not that high, especially for the larger boards as this is a much harder problem to solve, so someone could first try to make a good ai for the smaller boards and then work up to creating an ai that can play reasonably well compared to a human beginner at the game. There is also a book that shows you how deep learning can be used to solve the game which can be found here `https://www.manning.com/books/deep-learning-and-the-game-of-go`.

6.0.2 Incorporating with different implementations

My game was created without being able to also be used to play against ai created by other players. There is a standardised format for playing moves on a board as well as a way to format boards that could be used to also incorporate other ai into the game which could then be used to compare the ai against already created ai if needed.

Appendix A

References

https://www.youtube.com/watch?v=UXW2yZndl7U&t=625s
https://www.youtube.com/watch?v=lhFXKNyA0QA&t=7s
https://www.youtube.com/watch?v=l-hh51ncgDI
https://www.youtube.com/watch?v=62nq4Zsn8vc
https://en.wikipedia.org/wiki/Go_(game)
https://arxiv.org/abs/1611.00625
https://www.geeksforgeeks.org/ml-monte-carlo-tree-search-mcts/
https://www.analyticsvidhya.com/blog/2019/01/monte-carlo-tree-search-introduction-algorithm-dee
https://towardsdatascience.com/monte-carlo-tree-search-an-introduction-503d8c04e168
https://www.geeksforgeeks.org/ml-monte-carlo-tree-search-mcts/
https://int8.io/monte-carlo-tree-search-beginners-guide/

Bibliography

[1] Baeldung. *Introduction to Minimax: Algorithm with Java implementation*. 2021. URL: https://www.baeldung.com/java-minimax-algorithm.

[2] Cameron Browne et al. "A Survey of Monte Carlo Tree Search Methods". In: *IEEE Transactions on Computational Intelligence and AI in Games* 4:1 (Mar. 2012), pp. 1–43. DOI: 10.1109/TCIAIG.2012.2186810.

[3] Bo Chang. *Stochastic Bandits and UCB Algorithm*. Dec. 2018. URL: https://bochang.me/blog/posts/bandits/.

[4] Ankit Choudhary. *Introduction to Monte Carlo Tree Search: The Game-Changing Algorithm behind DeepMind's AlphaGo*. 2019. URL: https://www.analyticsvidhya.com/blog/2019/01/monte-carlo-tree-search-introduction-algorithm-deepmind-alphago/.

[5] Marissa Eppes. *Game Theory - The Minimax algorithm explained*. Aug. 2019. URL: https://towardsdatascience.com/how-a-chess-playing-computer-thinks-about-its-next-move-8f028bd0e7b1.

[6] Hilmar Finnsson. "Generalized Monte-Carlo Tree Search Extensions for General Game Playing". In: (2012).

[7] gwk. *Summary of the AlphaGo paper*. June 2017. URL: https://becominghuman.ai/summary-of-the-alphago-paper-b55ce24d8a7c.

[8] Bartosz Sawicki Maciej Swiechowski Konrad Godlewski and Jacek Mandziuk. *Monte Carlo Tree Search: A review of recent modifications and applications*. Mar. 2021.

[9] Alpha Go Team. *Alpha Go Description*. URL: https://deepmind.com/research/case-studies/alphago-the-story-so-far.

[10] *Upper Confidence Bound Algorithm in Reinforcement Learning*. Feb. 2020. URL: https://www.geeksforgeeks.org/upper-confidence-bound-algorithm-in-reinforcement-learning/.

[11] Piyush Verma and Stelios Diamantidis. Apr. 2021. URL: https://www.synopsys.com/ai/what-is-reinforcement-learning.html.

[12] Lars Wachter. *Zobrist Hashing*. Aug. 2020. URL: https://levelup.gitconnected.com/zobrist-hashing-305c6c3c54d0.

[13] Benjamin Wang. *Monte Carlo Tree Search: An Introduction*. URL: https://towardsdatascience.com/monte-carlo-tree-search-an-introduction-503d8c04e168.

[14] Wikipedia. *G0 (game)*. URL: https://en.wikipedia.org/wiki/Go_(game).

[15] Wikipedia. *Monte Carlo Tree Search*. URL: https://en.wikipedia.org/wiki/Monte_Carlo_tree_search.

[16] Wikipedia. *Multi-Armed Bandit*. URL: https://en.wikipedia.org/wiki/Multi-armed_bandit.