

Computer Games Development

Project Report

Year IV

Jack Malone

C00236428

[Date of Submission]

**Contents**

Acknowledgements

Project Abstract

Project Introduction and/or Research Question

Literature Review

Evaluation and Discussion

Conclusions

References

Appendices

# Acknowledgements

Use this template when writing your research report. As a rule of thumb, the report should be of the order of 10 pages (about 250 words/page).

# Project 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.

# Introduction

Go is a two-player board game in which the aim is to surround more territory than the opponent.[[1]](#footnote-0) 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. For a 19x19 board, there is approximately 250 moves that each player can possibly play. If a game continues for 150 turns (average turn count), then there would be around 250150 possible moves. Given the possible moves 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, and use the formula bd where b is the branching factor and d is the required depth to find how many leaf nodes there are or how many different possible board states there could be.[[2]](#footnote-1)

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[[3]](#footnote-2). 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[[4]](#footnote-3). 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 viewed [here](https://www.youtube.com/watch?v=WXuK6gekU1Y&t=2855s). 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 which lead to the creation of Alpha-zero.

1. 250
2. 62,500
3. 953,674,316,406,250,000,000,000

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 can the Monte Carlo Tree search be used alongside neural networks or other AI techniques to make an AI that can play the game well.

# **Literature Review**

**Game Trees:[[5]](#footnote-4)**Each node of a game tree represents a particular position or state in a game. 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.

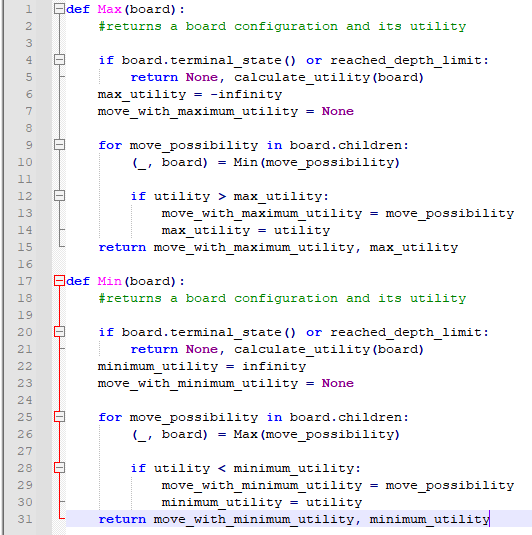
# The Monte Carlo tree search is a smarter kind of search when compared against an *uninformed search.*

**Uninformed search**: an uninformed search just searches through the search space without any knowledge of what the goal state is. As such, they will search through the entire tree before coming to a conclusion as to what the best way to traverse the tree is.

Two examples of uninformed searches are depth-first search and breadth-first search.

**Minimax:[[6]](#footnote-5)** Minimax is a decision making algorithm which is usually used in a 2 player, turn based game. The algorithm tries and finds the best next move in the game. In its implementation, one player is the minimiser and the other is the maximiser. If the evaluation of the current board state in the game is stored as a number, the maximiser will try go to a game state with the maximum score and the minimiser will try to get a game with the lowest score. 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 (increase chance of winning) results in the decrease of the score for the player (increase 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[[7]](#footnote-6). The algorithm takes into account three basic functions: *Maximise* and *Minimise*, as well as a *Utility Calculation*.

Possible implementation of the minimum and maximum functions for Minimax:



**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[[8]](#footnote-7). In the stochastic bandit problem, the rewards from each arm are from a probability distribution specific to that arm[[9]](#footnote-8). In the problem, there is a trade off between exploration. In machine learning, exploration stands for the acquisition of new knowledge, and exploitation refers to an optimised decision based on existing knowledge.

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 ϵ [K], rounds by t ϵ [T]. The reward for a specific arm a is Da, which is distribution supported on [0,1]. The expected reward is denoted by μ(a) := ∫01 x, dDa(x). The best expected reward is denoted by μ\* := maxaϵ[K]μ(a), and the best arm is a\* = argmaxaϵ[K]μ(a).

The cumulative regret in round t is defined as

R(t) = μ\*t-∑s=1t μ(as)

Where as 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. Let͞μ(a) be the average reward for arm a.
2. Exploitation phase: select arm â with the highest average reward â=argmaxaϵ[K]͞μ(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/>.

**Upper Confidence Bound Action:**

*Exploration vs. Exploitation:[[10]](#footnote-9)*

* **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.

*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 estimate to drive exploitation.

At = argmaxa(Qt(a) + c)

Where t is equal to the time-steps and Nt(a) is the number of times action a is taken. Qt(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.

**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. The 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.

There are four steps that are applied each time the algorithm 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/Rollout:* during 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. *Backpropagation:* 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 aren’t 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)

**Training pipeline of Alpha-Go[[11]](#footnote-10):**

* *Supervised learning of policy networks:* The supervised learning policy network  
   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 traine a faster but less accurate rollout policy 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 has the same architecture as and initialised with the final weights of . 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 nueral network won 11% of games against Pachi and 12% against a slightly weaker program, Fuego.
* *Reinforcement learning of value networks:* Value network approximates the optimal value function by approximating . Trained on a large number of simulated games of 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 rollouts using the fast rollout policy .

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”**[[12]](#footnote-11)** which gives an outline of how the Monte Carlo Tree Search works as well as how it 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”[[13]](#footnote-12) 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*”.

# **Evaluation and Discussion**

Replace this text with Results and Discussion.

Describe the results using diagrams such as graphs etc. as appropriate, and discuss what the results mean.

Example: Results indicate that once the threshold gets over a certain point it significantly reduces player performance and player experience

# 

**Project Milestones**

Replace this text with Project Milestones.

Key project milestone dates and measurement on schedule, was project schedule adhered to, effectively planned for delivery on-time or ahead of schedule if appropriate.

**Major Technical Achievements**

What are your major technical achievements?

**Project Review**

What went right? What went wrong? What (if anything) is still outstanding/missing (i.e., still left to do)?  If starting again, how would you approach this project differently? What advice would you have for someone attempting a similar project in the future? Were your technology choices the right or wrong ones? If you chose the wrong technology, provide justifications for why you think this. What were the implications of your technology choices?

# **Conclusions**

summarise your work and findings.

**Future Work**

Indicate what might be some next steps to try (if a student next year was going to undertake a project in this area what might be an interesting thing for him/her to examine?).

# **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-deepmind-alphago/>

<https://towardsdatascience.com/monte-carlo-tree-search-an-introduction-503d8c04e168>

# **Appendices**

Replace this text with Appendices.

This might include ethics application and other relevant material e.g. copy of any questionnaires used.

1. <https://en.wikipedia.org/wiki/Go_(game)> [↑](#footnote-ref-0)
2. <https://towardsdatascience.com/monte-carlo-tree-search-an-introduction-503d8c04e168> [↑](#footnote-ref-1)
3. <https://en.wikipedia.org/wiki/Monte_Carlo_tree_search> [↑](#footnote-ref-2)
4. <https://deepmind.com/research/case-studies/alphago-the-story-so-far> [↑](#footnote-ref-3)
5. <https://www.analyticsvidhya.com/blog/2019/01/monte-carlo-tree-search-introduction-algorithm-deepmind-alphago/> [↑](#footnote-ref-4)
6. <https://www.baeldung.com/java-minimax-algorithm> [↑](#footnote-ref-5)
7. <https://towardsdatascience.com/how-a-chess-playing-computer-thinks-about-its-next-move-8f028bd0e7b1> [↑](#footnote-ref-6)
8. <https://en.wikipedia.org/wiki/Multi-armed_bandit> [↑](#footnote-ref-7)
9. <https://bochang.me/blog/posts/bandits/> [↑](#footnote-ref-8)
10. <https://www.geeksforgeeks.org/upper-confidence-bound-algorithm-in-reinforcement-learning/> [↑](#footnote-ref-9)
11. <https://becominghuman.ai/summary-of-the-alphago-paper-b55ce24d8a7c> [↑](#footnote-ref-10)
12. <https://arxiv.org/pdf/2103.04931.pdf> [↑](#footnote-ref-11)
13. <https://www.researchgate.net/publication/235985858_A_Survey_of_Monte_Carlo_Tree_Search_Methods> [↑](#footnote-ref-12)