Computer Games Development
Project Report
Year IV

Jack Malone
C00236428

December 2021

Institiúid Teicneolaíochta Cheatharlach

INSTITUTE *of* TECHNOLOGY

CARLOW

At the Heart of South Leinster

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.

Contents

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[12]. 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[11].

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[13]. 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.

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[14]. 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. 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*.

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

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

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

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