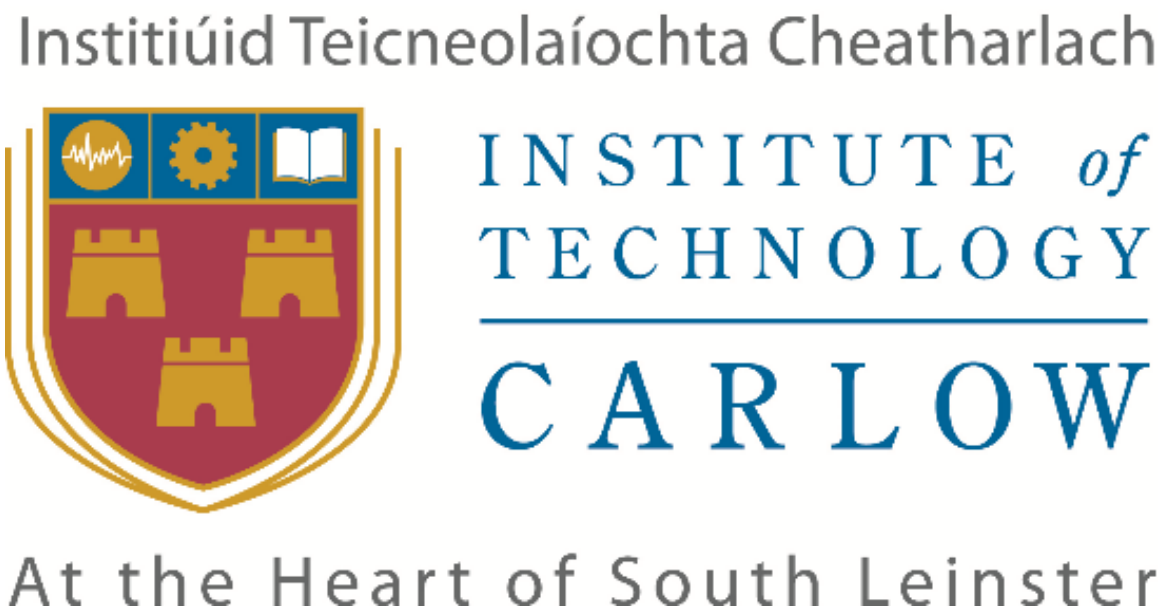
Computer Games Development
Project Report
Year IV

Jack Malone
C00236428

December 2021



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.

Acknowledgments

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[7]. 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[6].

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[8]. 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[5]. 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[3]. 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[4]. The algorithm takes into account three basic functions: *Maxmise* 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[9]. In the stochastic problem, the rewards from each arm are from a probability distribution specific to that arm[2]. 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.

2.2.2 Monte Carlo Tree Search

```
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)
```

2.2.3 Training pipeline of Alpha-Go

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

Chapter 3

Conclusion

Appendix A

Appendix

Bibliography

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

[2] Bo Chang. *Stochastic Bandits and UCB Algorithm*. Dec 8, 2018. URL: https://bochang.me/blog/posts/bandits/.

[3] 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/.

[4] Marissa Eppes. *Game Theory - The Minimax algorithm explained*. Aug 7, 2019. URL: https://towardsdatascience.com/how-a-chess-playing-computer-thinks-about-its-next-move-8f028bd0e7b1.

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

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

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

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

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