UNSW
SYDNEY

Simple Go playing using Minimax and Monte-Carlo methods

Chan Victor Yathong

z5262347

Abstract

In this paper, we explore the game of Go, the complexity and historical attempts to tackle Go by computers. Minimax and Monte Carlo methods are discussed, followed by implementations of the algorithms on a simplified game of Go.

Content

Introduction

Humans like playing games. We play all sorts of games among ourselves, with players or computers, with or without money, be it cards, chess, monopoly or Go. Recently, a new show about Chess came out on Netflix and caused a shortage of chess boards in stores.

With a history of more than 2500 years, the game of Go is said to be the oldest of all, that is still played in its original form. The rules of Go is simple. Every turn, two players place stones on a set of grid lines, with a strategic goal to capture enemy stones by surrounding them, and ultimately claim the territory that is surrounded by their own stones. The score is calculated at endgame. Given the simplicity, one would think that it is not hard for a computer to master. On the contrary, it is widely accepted as incredibly complex. In 2000s, researchers said that the chance for a computer to best humans by 2020 is at most 50/50 [1]. In fact, with advancement in algorithms, Chip Engineering and Artificial Intelligence, the human champion Lee from Korea was defeated 4-1 in 2016 with world-wide media coverage [2].

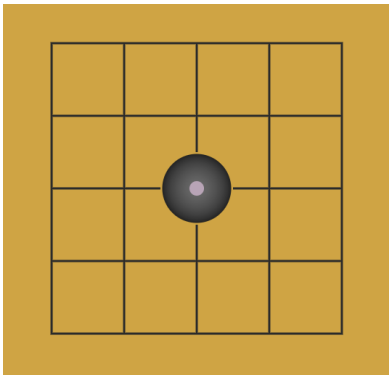
As a computer science student, I am therefore interested to delve deeper into the Game of Go, about the complexity, forms, and Algo strategies that has emerged. I will provide an overview of the A.I. programs developed by Deep Mind, and then implement a few algorithms on a simplified Go game myself.

Rules

The most important Rules in Go are as follows:

1. Board (19x19) is empty when game starts; Black moves first.
2. Each player move by placing a stone on an empty section of the board
3. A liberty is defined as the number of adjacent free spaces a stone has
4. A connected group of stones gets captured when all liberty is blocked; They will be extracted and free up the space. The active player takes priority in capturing.
5. Play that revert the game to a previous position is not allowed
6. Taking the captured points into account, the player who surrounds more area wins

Different regions of the world plays Go with subtle rules and scoring differences. But the rules above is universal.



(Figure 0. A black piece with 4 liberty)

Patterns

Due to the vast possibilities under these rules, a lot of special cases are involved, and rules in different region score them differently. Here, I will document a few interesting patterns I encountered that are also useful to understand the complications involved.

Ponnuki (*An opened flower*)

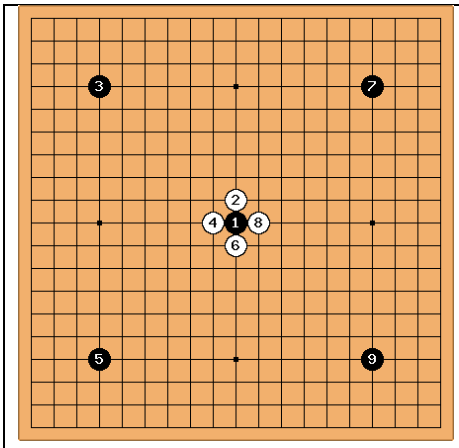


Figure 1. An example of Ponnuki

The Ponnuki is a pattern formed by capturing an opponent stone by four of your stones. In the example, white forms a Ponnuki by surround and captures the black stone in the center of the board.
Ponnuki is highly sought after by Go experts due to the shape having excellent offensive and defensive properties as the game plays on. An old saying goes: "A Ponnuki is worth 30 points."

Seki (*Mutual living*)

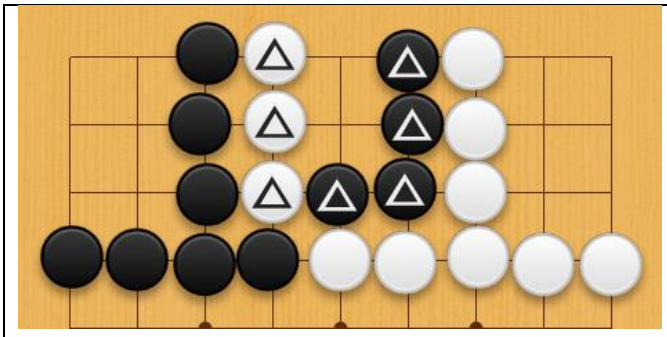


Figure 2. An example of Seki

The Seki is a special group of patterns that means "Mutual living" in Japanese. It usually involve 2 liberty shared in an enclosed area of black and white stones.

If either player place a stone in one place, the opponent could easily capture the other players' stones (triangles) by playing immediately to the other. In such a situation, no player will play first into this group of stones, forming a "frozen" area on the board. The scoring of this part is calculated differently in japan or Chinese rules.

Ladder

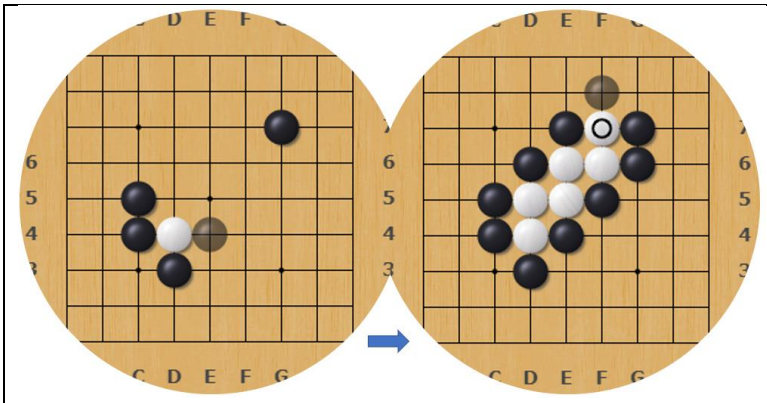


Figure 3. Example of ladder

A ladder is an offensive strategy to take when you have a number advantage.

In this example, black plays at E4 to threaten a capture, which prompts a defensive move of white at D5. Black can keep up the pressure by following-up, carrying the battle to the right side of the board. All whites are captured by the help of a black stone at G7.

Difficulty

Go is played under perfect information; Each player knows exactly the scores and the position of each stone. For this reason, it can be viewed as a Markov Decision Process, where the perfect decision can be found by investigating the current game state The move history of getting to this state is irrelevant. In theory, a best move exists in every state.

However, knowing this does not make solving Go easier. Given the amount of legal positions, It should come as no surprise that Go is notoriously hard for a computer to play well. On a standard 19x19 board, a proper game can contain around 10^{160} state spaces [1]. To put that into context, chess has a state space of 10^{40} , and scientists estimate around 10^{80} atoms in the universe. The search space of Go simply makes these numbers miniscule. With this amount of freedom, brute-forcing all choices is out of the question, albeit the powerful GPUs we have nowadays.

One would seek to simplify such problem with a divide-and-conquer approach, i.e. breaking the problem of Go into smaller sub-problems. To start with, most played Go games has three phases: an Early game of around 20 moves where players play their stones quite evenly into each region to secure a territory; A mid game where most of the invasion and skirmishes take place, starting with one part and extend to the centre; An end game where most areas are established, and outcomes of areas are made based on mid game's residual. While it may be possible to write an algorithm for the early game, it is hard to decide even when the mid-game has started. In rare occasions, the fight start in one area before every area is marked. Moves placed in an area first gives you an advantage, but evaluating that influence using any hard-coded algorithm is unreliable, if possible.

On the other hand, one might seek to divide a Go board into smaller zones. A normal go board can be treated as an NxN grid of areas, and we could model each area as a smaller sub-problem. This was the fundamental idea of the first computer program to play Go [1]. Unfortunately, this approach struggles to beat good players, because it is hard for quantify inter-relations and proximity of different zones. When humans realize the program did not play well in certain places, they turn it to their advantage. Something happening at the boundary of this area will probably affect the way you deal with the area nearby, as players rush to make strong connections during the mid game.

To be more concrete, take the ladder formation as an example; Under normal circumstances, forcing a ladder is deemed unwise due to the loosely enforced outer-shape, which could pose a lot of weakness if the opponent managed to escape by connecting one of their regions. However, an experienced player could chip in moves in other areas before starting the ladder, to maximize the likelihood of a capture. These moves, while legit in their own regions, does impact the game in other areas greatly. This inter-dependency makes the attempt of dividing sub-areas hard.

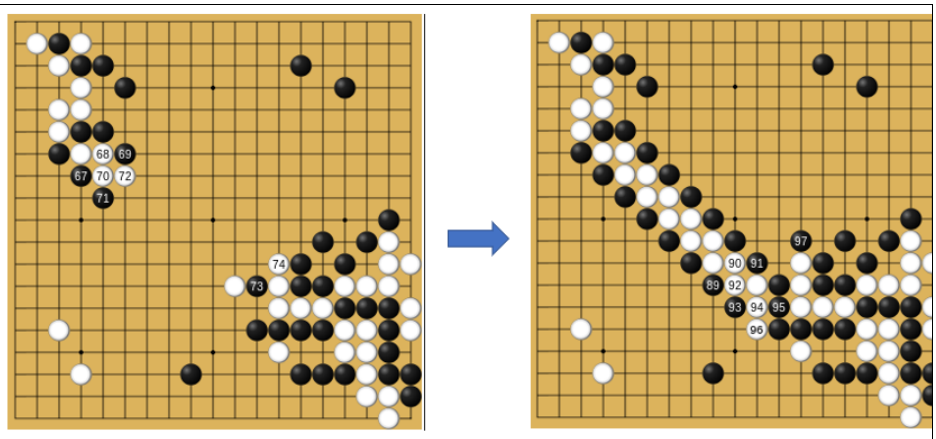


Figure 4. Use of Ladder in a high-level Tournament. White resigned at the end of the game.

Albeit the hardness in making an algorithm, researchers have tried to find an upper-bound of the difficulty by proving the hardness. At the endgame where a game's state space collapses to smaller sub-games, the overall endgame is still a **PSPACE-hard** problem of many polynomial-size game trees [3]. Determining whether a ladder results in a capture is a **PSPACE-complete** problem. As a whole, Go is a game that combines many of these hard problems into a problem of **EXPSPACE-hardness** [3]. This means as the size of the board grows, the space needed to compute a solution grows quickly out of reach of any computer in today's standards.

For these reasons, there has not been a successful algorithm that claims to play perfect Go in reasonable time; The game is far too complicated to be generalized as high-level logic. Until recently, Computer Go can hardly beat a professional player.

Minimax Tree Search

Minimax is a game playing algorithm that is used in a lot of game playing machines [1]. Given enough resources, it can find an optimal solution of a turn-based, two player game. It was also the underlying concept of DeepBlue, which was a chess-playing engine made by IBM to defeat GrandMaster Garry Kasparov.

By treating every game state as a node, the algorithm constructs a game tree from the current state, denoting possibilities that the game could progress. Then, it investigates the tree level by level. It alternates between two roles: maximizing the gain of your levels, and minimizing the gain of your counterpart. In a zero-sum game, a loss of your counterpart is your gain, and you assume the opponent also plays optimally. Thus, this strategy maximizes your points while trying to minimize potential points of your opponent, increasing your likelihood to win.

The trick of a good minimax search comes with the metric, the evaluating function. It should evaluate a game state and decides accurately if the state is beneficial to the player. This metric is crucial because it serves as the deciding factor of picking a move over the other. Imagine playing a boardgame like Chess, where trading pieces with your for an advantage is common. How do you place the value of your Knight with the opponent's Pawn? If you values the Knight more, what if the Pawn is close to your side, and it has a chance to promote to the queen or a rook? To learn about these parameters, DeepBlue started with a generalized function, and slowly tune them by analysing thousands of Master Games. It learns about importance of pieces at different positions at different stages, and was fine-tuned by a Chess Grandmaster. In real-time, it can generate and search for 200 million positions per second on a parallel supercomputer [4].

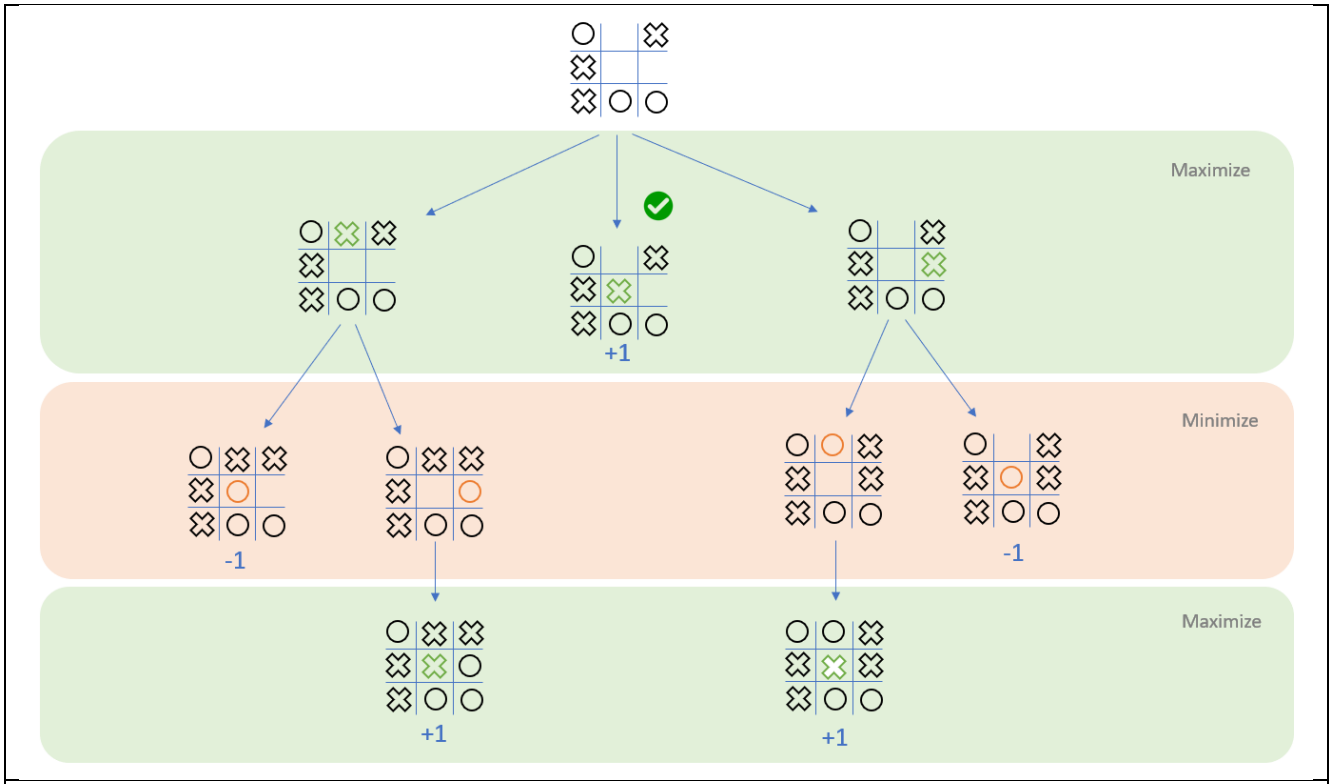


Figure 5. Playing tic tac toe with minimax tree as the Cross player

Pruning

To decrease the search space, most implementation of the minimax algorithm comes with Alpha-Beta pruning, an early-stopping strategy [1]. Although there are different implementations, the concept is the same. In essence, it ceases to continue the search from a subtree that is known to produce a worse outcome than a previously explored subtree, cutting off that branch completely.

In Alpha-Beta minimax search, α is set as the minimum score that our player is assured of, β as the maximum of opponent's. They are assigned as negative infinity and positive infinity respectively, which corresponds to the worst starting value. During expansion of the current state, α and β are updated during at each level. If after evaluating a subtree it is found that $\beta > \alpha$, that subtree will be pruned.

In most cases, it does not affect the choice of the current step, although its efficiency depends on the order of node traversal, much like quick sort. In case you traverse all nodes in increasing order of value, it is no quicker than not pruning. Still, you save time for later expansion by cutting some branches early.

Although Minimax coupled with pruning achieve good performance in a lot of games, applying it to Go has not seen much success. As we have mentioned, the number of search space increases exponentially, so searching an extra step requires an extra magnitude of computational resources. To be concrete, the branching factor of a Tree easily exceeds 300 on a 19x19 board, which makes minimaxing infeasible until very late into the game [1]. To prune earlier, expert knowledge is often applied to approximate the position's value, which could be biased from a scientific perspective [2]. Still, with all these applied, a depth-limited minimax search can look no further than 10 steps ahead in Go.

Monte Carlo Tree Search (MCTS)

Monte Carlo method is a broad class of methods that rely on random sampling. It draws inspiration from statistical physics about study of materials and atomic particles [5]. I will recap an example from the paper, published by a researcher in the Max-Planck-Institute of Physics. Imagine you are measuring the temperature of a metal in cooldown, but you can only do so by measuring one particle's velocity at a time. Atoms collide and transfer energy all the time, and single measurements could be inaccurate. By measuring repeatedly, however, we obtain a better understanding of the process. We can also predict the temperature at rest quite accurately after some trials. This concept provokes the angle that the author was looking at the Go problem: *"How would nature play Go?"*. A wide range of research came, and MCTS has since become the core of every good Go-playing computer program [6, 7].

Put simply, Monte Carlo is about playing random games. When applied to a traditional tree search method, it increases the confidence level of a certain node by playing many games randomly. In each iteration, the most urgent node to explore is determined by a policy function. Then, a simulation is run from the selected node, consist of playing a number of games until the terminal state is reached, i.e. winning or losing. The way how these games are played vary, but the simplest case is just playing uniformly randomly. After this, the value of the current node is updated, with statistical figures like variance in mind. This change is then propagated back to the parents. The whole process can go on a few times to explore nodes that is worth exploring, or needs further information.

One would think that completely random actions are not ideal, because they do not portray real-world actions. In fact, you can learn something even by playing randomly, because some moves are great no matter when you play it [5].

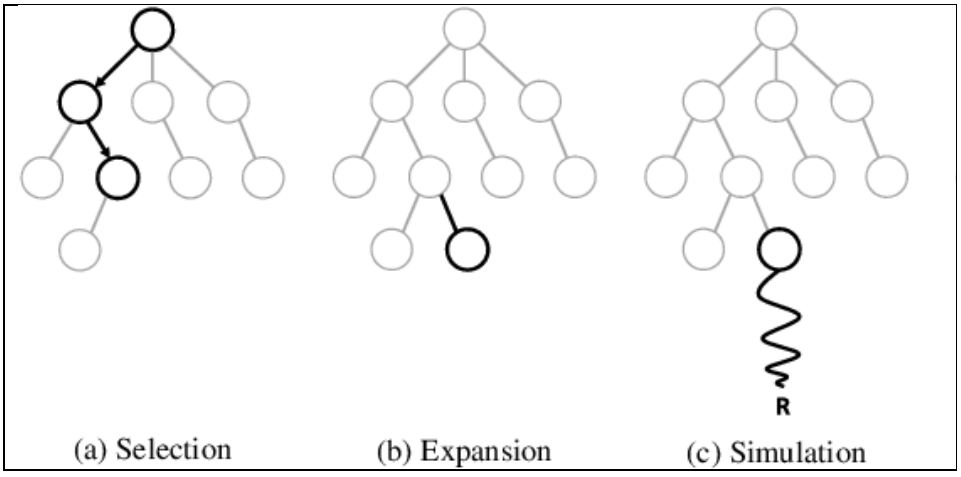


Figure 6. The 4 phases of a basic Monte Carlo Tree Search

Here, we look at an implementation of the MCTS in Crazy Stone, a very strong Go playing program.

In the algorithm, the expected value can be denoted by Q-value

$$Q(s,a) = \frac{1}{N(s,a)} \sum_{i=1}^{N(s)} l_i(s,a) z_i$$

where N(s,a) is the number of times action a is selected at state s, N(s) denotes the number of times a game has been played through state s, and $I_s(s, a)$ is 1 if a was selected at state s, and 0 otherwise.

With the Q-value calculated, the action is then selected with a probability proportional to

$$p_i = exp\left(-2.4 \frac{Q_0 - Q_i}{\sqrt{2(Q_0 - Q_1)}}\right) + C_i$$

Where Q values are sorted in descending order, so Q_0 is the maximum Q value. C_i is a constant to make sure the urgency never goes to 0, which is defined by

$$C_i = \frac{0.1 + 2^{-i} + a_i}{\mathrm{N}}$$

With these advanced metrics, Crazy Stone combined it with knowledge of a learned pattern library. It proceeded to become the first program to beat a professional player with less than 9-stone handicap [7].

AlphaGo

The famous champion-beating Go program combines MCTS with neural networks, a machine learning technique. By design, there is a policy network that output probabilities of how an expert will play at a given state, and a value network that evaluates the value of a board position. The policy network is trained using supervised learning of expert Go histories, followed by reinforcement learning, whereas the value network is only trained by reinforcement learning [8].

At play, the program performs random search quickly by selecting the best action output by policy network. When it reaches a leaf node, the leaf node may be expanded; It will be evaluated first by the value network, then by quickly playing to the terminal step. These values then combine to form a final value of the node. At the end of its search, all visited edges are updated, and the program chooses a best move based on the most visited position.

After AlphaGo vs Lee received a lot of attention, a stronger version, AlphaZero was trained. While still using MCTS for searching, it trained itself by pure reinforcement learning [2]. Through self-playing millions of games, it discovered some non-standard strategies and positions that no human player has known. In terms of playing strength, it is said to have attained an elo of 5185, a much stronger performance than the version that defeated Lee(3739).

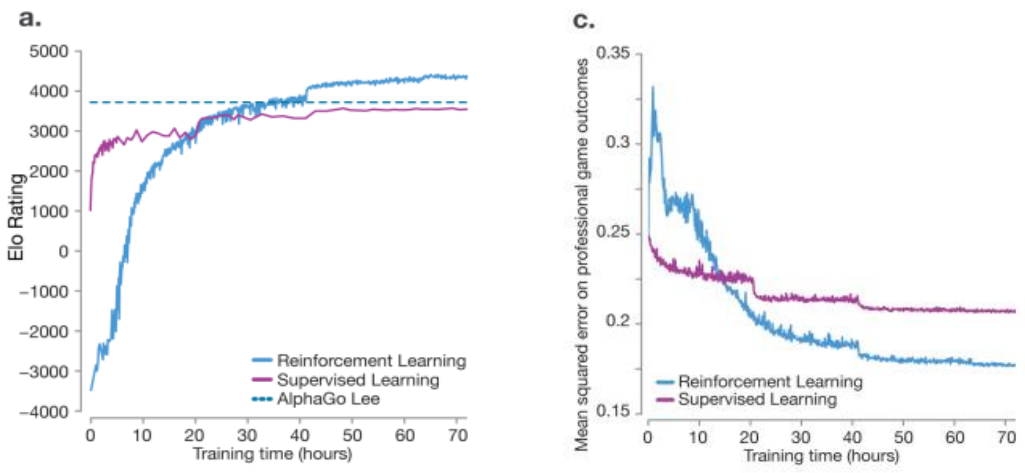


Figure 7. AlphaZero achieves unprecedented Go playing strength after playing with itself for 40 hours

Implementation

As a small study, a few experiments were conducted with the game-playing algorithms, notably the minimax and Monte Carlo searches. To do this the logic of GO rules has been programmed. A few hypothetical players have been constructed to compete with one another. Due to the constraint of resources, I must keep everything simple and quick. For this reason, I proposed a new set of rules for my simulation.

1) The board is limited to a 5x5 space. This number is obtained after trial-and-error of running the program on my laptop. It doesn't make it too slow to run multiple games, yet still provide some meaningful insights;
2) Each player's first move is pre-defined. They played first in the same game alternately
3) Players earn points by capture opponents' stones and surrounding them
4) The first player that captures 10 stones win

All the non-suicidal and no repeated position rules still apply.

Player1: The beginner

The beginner is someone who have just picked up Go as a hobby. He plays by the rules, but he has no idea what to look for. At any state, he has an equal chance to play on every legal position on the board. The beginner serves as our control

Player2: The Forward Looker (FL)

The forward looker uses minimax tree to look for best moves. Every time it is his turn to play, the Forward Looker looks a few steps ahead using a Minimax Search Tree. Then, it is going to choose the step that maximizes his likelihood of winning, rated by the expected number of captures. If there are more than one such paths, he will pick one at random.

The high-level logic of my minimax Tree is as follows:

- When a node is constructed, a 5x5 matrix is generated denoting the instant value of placing a stone in that area. The initial expected value of this position is represented by the maximum of this matrix.
- During a node expansion, every possible state is expanded to form a new node
- After the desired depth is reached, each node's value is updated in a back-propagation
- Every level has a depreciation factor of 20%. In other words, an early capture is preferred.

Player3: The Good player (GP)

The experienced player improves on the forward looker by digging deeper into the tree. On top of the process of forward looking, it picks the 2 best moves and analyze them further. From the two positions, it expands 2 steps ahead, expanding all nodes that guarantee a return, and others with a 30% chance. This iterative deepening process mimics the thought process of a player who shortlist a few good moves, then evaluate and compare his options.

Player4: Monte Carlo (MC)

Player 4 is special. He knows the basic rules, but he forms a strategy of his own. Every time it is his time to play, he plays a lot of games in mind, all moves at random. Because his brain is capable, he can play hundreds of games a second.

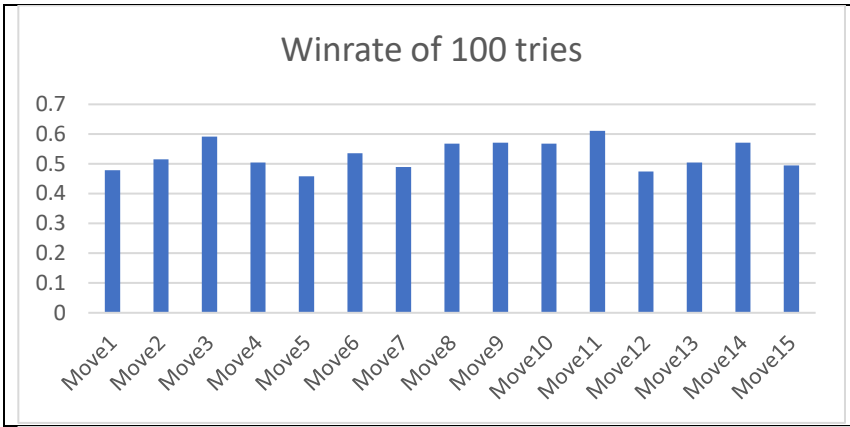
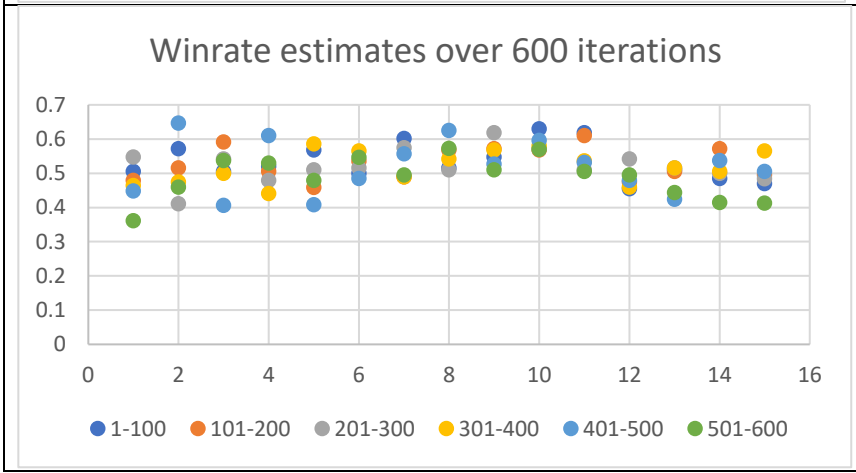
Our Monte Carlo implementation chooses a move based on random games played through a position. Only wins count, and draws are considered losing. To penalize moves that show a large variation in testing, the goodness of a move is calculated by the formula:

$$x = \frac{w}{\mathrm{N}}(1 - \partial^2)$$

N = total games played, W = total games won

∂ represents the standard deviation of results obtained over 4 batches

After evaluating the value for each move, the move representing maximum value is selected.

	The average winrate over 100 tries from a position. At first glance, the best move is unclear.
	Scatter plot of 600 tries from the same position, with intervals of 100. In this case, move 10 would be chosen due to its win rate and consistency.

Testbench and optimization

All code are written in Python 3.8 with numpy for matrix manipulation. Some testing and optimization took place to choose between different programming methods. Naturally, most of the run time is dedicated to the **val_matrix** function, which calculates the outcome of playing on different positions of the board. To mitigate this, an attempt was madeto change the logic from numpy matrix functions to simple python recursion. Due to most recursions happen only a few times on the 5x5 board, this has decreased the time spent on average for 70%.

The simulation was conducted on my Desktop computer, with i7-8700 as the CPU. Even with 32GB of ram, the minimax tree construction quickly exhausted all ram available. In contrast, my implementation of Monte Carlo does not use a lot of memory, but time to run increased quickly with amount of random games to play.

Results

The simulation is ran in batches. Due to the difference in complexity, run time for the Good Player and the Monte Carlo limit the number of games we can simulate. Still, we managed to run at least 20 games for every pair of players.

***FL (n)**: The forward looker who plays by building minimax trees n steps ahead*

***GP (n, r)**: A good player that looks n steps ahead, then r steps further into the best 2 nodes*

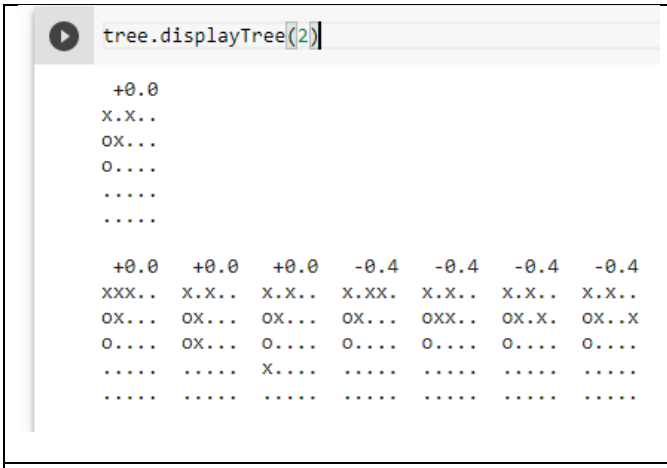
***MC (n)**: Monte Carlo that plays **n** games randomly from possible positions*

	Games Played					
	against					
by	Beginner	FL(1)	FL(2)	GP(2,2)	MC (20)	MC (200)
Beginner						
FL(1)	*200*					
FL(2)	*200*	*100*				
GP(2,2)	*100*	*100*	*100*			
MC (20)	*100*	*100*	*100*	*50*		
MC (200)	*50*	*50*	*20*	*20*	*20*	

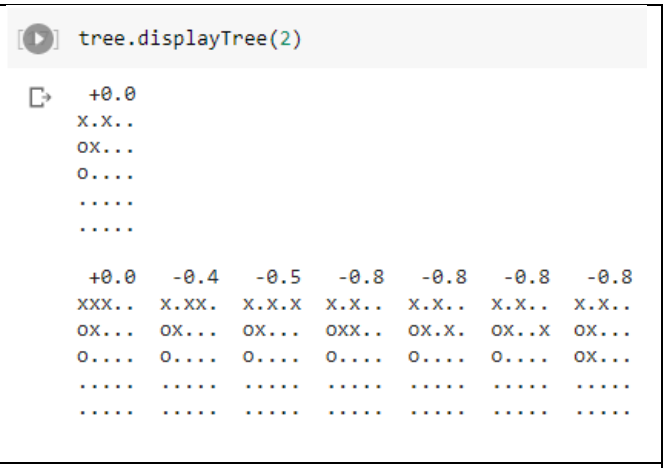
	Win rate					
	against					
by	Beginner	FL(1)	FL(2)	GP(2,2)	MC (20)	MC (200)
Beginner						
FL(1)	80.50%					
FL(2)	88.50%	72%				
GP(2,2)	100%	83%	68%			
MC (20)	66%	45%	32%	10%		
MC (200)	84%	54%	50%	20%	65%	

Evaluation

Our evaluation starts by comparing each algorithm with the control player. Just by planning one step ahead, FL(1) beats the beginner more than 80% of games. By planning one step further, the rate increases to 89%. The Good Player wins all the games against the beginner. This proves that the ability to plan far ahead is crucial to playing Go. An example produced by our visualization illustrates how expanding further into the tree produce better approximated expected values.



Expected values looking 2 steps ahead



Look 1 step further and expanding promising nodes

Our Monte Carlo algorithm scored quite well even with only 20 games played, indicating our algorithm is effective. Interestingly, the performance is quite consistent. It equates to a player that's slightly below the level of a FL(1) player. As we swap to 200 games Monte Carlo, performance improved, albeit with a rate that is lower than expected. With 10 times the original games played, MC(200) barely draws with FL(2), and struggled to play with the Good Player, losing 16 out of 20 games.

To investigate further, the play style of the Monte Carlo player and game histories was studied. Interestingly, the algorithm tends to aim for long term strategic targets, forming chains and omits small captures that minimax trees are proficient at. This can be illustrated by a game it played with FL(2) which I included in the appendix. Also, it is observed that there is a tendency for the MC to play poorly at the edge of the board, which was mentioned as a shortcoming of the flat MCTS. I attribute this to the insufficient number of games explored.

Limitations

This simulation was intended as a self-study of game-playing algorithms. It was not an attempt to make new discoveries. Still, there are several areas it could be improved. Firstly, the number of games played is insufficient for statistical significance; Also, more starting positions could be tried. The number of levels to expand in Trees is insufficient to draw a good comparison with Monte Carlo; The same can be said for Monte Carlo random plays. In fact, most simulations like this are now be done on cloud GPUs, as they are much more powerful in matrix calculations.

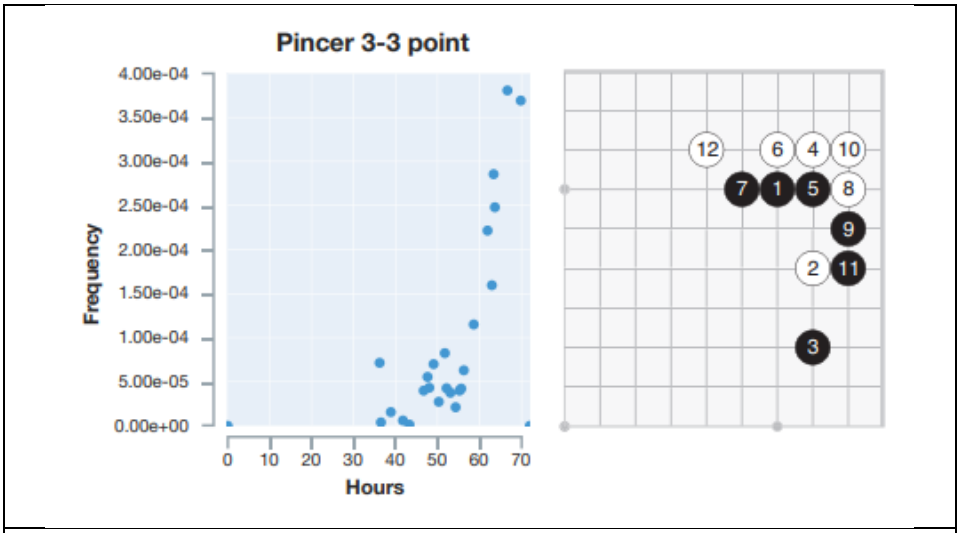
In terms of scope, more variations of the minimax tree could be investigated, along with different pruning strategies. More experimentation could be done with the discount ratio across levels, and the effect on long term and short term goals could be observed. Overall, the study was highly bottle-necked by hardware, and some of my optimization, although helpful, does not solve the core of the problem: Every level of the tree utilizes more than ten times the resource of the previous one. As a follow-up, coding it in a more efficient language like C could make it faster, but a huge improvement will not be expected.

Conclusion

Go is a board game that has a long history. In theory, every state of a Go game has an ideal move, although the computing power needed is astronomical. By reviewing literature, we investigated the minimax and the Monte Carlo algorithms, and some of the breakthroughs that came in recent years. The minimax plays the game by brute-forcing future states in a smart way; The Monte Carlo method obtains an estimate by playing as many random games as possible. After I tried to code a simulation by myself, it is understood that no matter how simple the game rules become, a Game like Go cannot be mastered by a computer without knowledge obtained by a lot of self-playing.

In this project, I have learned much about general Game-playing algorithms, as well as their respective applications in games. I go as far as to program a Go simulation from scratch, just to be met with realistic bottlenecks that simply smarter algorithms cannot help. At the end of the day, I have gained in knowledge and skills that I am sure will be useful in the future

I was partly motivated by the AlphaGo documentary, about the journey the Google DeepMind team who took up the challenge to beat the world champion in Go. When I watched it, I was moved by the way tension presented between the champion and the program, which has transcended above a normal Go battle. More importantly, it represents a new milestone in computer science. It has long been thought impossible to beat a professional player in Go, let alone the world champion. Witnessing such an event makes me wonder: If there is no task a computer cannot do better than us, then what are we good for? Chances are, we all have to learn from robots in the future.



A new move that discovered by AlphaZero after playing with itself millions of games

References

[1] T. C. Bruno Bouzy, "Computer Go: An AI Oriented Survey," *Artificial Intelligence,* no. 132, 2000.

[2] J. S. David Silver, "Mastering the Game of Go without Human Knowledge," DeepMind, London, 2017.

[3] R. A. H. Erik D. Demaine, "Playing Games with Algorithms: Algorithmic Combinatorial Game," 2008.

[4] M. Campbell, "Knowledge Discovery in Deep Blue," *Communications of the ACM,* vol. 42, no. 11, 1999.

[5] B. Brugmann, "Monte Carlo Go," Max-Planck-Institute of Physics, Munchen, 1993.

[6] G. K. B. R. Steven James, "An Analysis of Monte Carlo Tree Search," Council for Scientific and Industrial Research, Providence, 2017.

[7] E. P. D. W. Cameron B. Browne, "A Survey of Monte Carlo Tree Search Methods," *IEEE TRANSACTIONS ON COMPUTATIONAL INTELLIGENCE AND AI IN GAMES,* vol. 4, no. 1, 2012.

[8] A. H. David Silver, "Mastering the game of Go with deep neural networks and tree search," *Nature,* no. 16961, 2016.

[9] G, F, 203.

[10] R. Coulom, "Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search," Italy, 2006.

[11] D. S. Sylvain Gelly, "Combining Online and Offline Knowledge in UCT," in *Proceedings of the 24 th International Conference on Machine Learning*, Corvallis, 2007.

Appendix

1. High-level definition of the program (Will be uploaded to https://github.com/sholick?tab=repositories once clean-up and commented)

```
Go_no_go methods:

  def DoItAll( matrix ):                                  # return all 1 for the board, as soon as the place is empty

  def PromAndRandom( matrix, prob )                       # return 1 and 0 based on probability of trying, default probability is 30%

Misc:


  def compressor( board ):                               # compress a board state into a list, for quick comparison afterwards
      return [black, white]                              # [ [0,0,1,...], [1,0,1.....] ]


  def cluster( board, xy, liberty ):                     # Find the cluster starting from (x,y), loop the board to determine if the cluster is alive or dead
      return record, liberty                             # dict of nodes, True/False


  def outcome( board, x, y ):                            # Determine the impact of playing at (x,y) by checking surrounding enemies of (x,y)
      return result, new_Board                           # (int), numpy.matrix
                                                         # uses cluster

  def val_matrix( board, turn, history ):                # Determine the values of playing at all possible places on a board
      return result, outBoard                            # list of list [[5] * 5],  list of direct outcomes of boards [[5] * 5]
                                                         # uses outcome


class Game( history, board, turn, size, score ):        # represents a game of Go

  def init():           # init with empty board

  def play(x,y):        # proceed to play at x,y coordinate and update score if needed

  def display(b):       # Display a board

  def replay(hist):     # Display the history of the game from start
```

```
class node( board, matrix, path, children, parent, value, eV, turn, history ):

  def find( board ):           # Find the node with specific board from children

  def last():                   # Find the last step of this node

  def proceed( x,y ):           # Construct new node as if we play at x, y

  def expand():                 # FULL explansion. Proceed() every possible location, uses DoItAll() function

  def selective():              # proceed Only ony those look promise, uses the PromAndRandom() function

  def update():                 # Updates expected value of this step by subtracting the children values. 20% discount rate is applied
                                # A cascade of update() from children nodes is equivalent to a backward propagation


class GameTree ( state, head ):                         # upper manager for nodes, with various control and expand functions

  def _init_( game ):           # init the tree with a game

  def show( board ):            # Display the current board

  def move( x,y ):               # Proceed the game by playing at x,y

  def lookForward( n,r ):        # Starting from root, Fully expand n steps, followed by r random steps at 30%

  def digDeep( num, n, r):        # From the (num) most promising moves, dig deeper into each of them by full expand n followed by r loose steps

  def displayTree( depth, optimal)  # Display the tree nodes that are expanded with different settings
```

2. Example of running the compete function

```
[111] playForRounds(MonteCarlo, beginner, 5, True)

      Game 0

      Score: B/0  W/1
      o o o o x
      o - o o -
      o o o o x
      x o x x x
      x x x x x

      Game 1

      Score: B/3  W/6
      x o o o o
      x x o o o
      x x o o -
      x x x o o
      x x - o o
```

3. The Monte Carlo player seems to focus on the long term strategic target by omitting the dying stone at move 4 and place at the center; This ultimately leads to a Victory.
   (Please omit the score board which does not change; it was a bug when I took this picture)

2: Score: B/4 W/1 - - - - - - x - - - - - - - - - - - o - - - - - -	3: Score: B/4 W/1 - - - - - - x - - - - - - - - - - - o - - - - x -	4: Score: B/4 W/1 - - - - - - x - - - - - - - - - - - o - - - o x -
5: Score: B/4 W/1 - - - - - - x - - - - - x - - - - - o - - - o x -	6: Score: B/4 W/1 - - - - - - x - - - - - x - - - - - o - - - o - o	7: Score: B/4 W/1 - - - - - - x - x - - - x - - - - - o - - - o - o
8: Score: B/4 W/1 - o - - - - x - x - - - x - - - - - o - - - o - o	9: Score: B/4 W/1 - o - - - - x - x - - - x - - - - x o - - - o - o	10: Score: B/4 W/1 - o - - - - x - x - - - x - - - - x o - - o o - o
11: Score: B/4 W/1 - o - - - - x - x - - - x - - x - x o - - o o - o	12: Score: B/4 W/1 - o - - - - x - x - - - x - - x - x o - o o o - o	13: Score: B/4 W/1 - o - - - - x - x - - - x - - x x x o - o o o - o
...	...	...
24: Score: B/4 W/1 o o o x o x x - x o x x x x o x x x o - o o o o o	25: Score: B/4 W/1 - - - x o x x x x o x x x x o x x x o - o o o o o	26: Score: B/4 W/1 - - o x o x x x x o x x x x o x x x o - o o o o o

4. Analysis of run time of the minimax Tree expansion process

```
         6387960 function calls in 9.027 seconds

   Random listing order was used

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
  1375722    0.213    0.000    0.213    0.000 {method 'get' of 'dict' objects}
   361136    0.106    0.000    0.106    0.000 {method 'update' of 'dict' objects}
  1127817    0.153    0.000    0.153    0.000 {method 'append' of 'list' objects}
    11114    0.003    0.000    0.003    0.000 {method 'extend' of 'list' objects}
        1    0.000    0.000    0.000    0.000 {method 'sort' of 'list' objects}
       12    0.000    0.000    0.000    0.000 {built-in method posix.getpid}
       17    0.000    0.000    0.000    0.000 {built-in method posix.urandom}
       17    0.000    0.000    0.000    0.000 {method 'acquire' of '_thread.lock' objects}
    64701    0.008    0.000    0.008    0.000 {method 'random' of '_random.Random' objects}
        2    0.000    0.000    0.000    0.000 {built-in method builtins.compile}
        2    0.000    0.000    9.027    4.513 {built-in method builtins.exec}
       12    0.000    0.000    0.000    0.000 {built-in method builtins.isinstance}
   660297    0.090    0.000    0.090    0.000 {built-in method builtins.len}
    10615    0.003    0.000    0.003    0.000 {built-in method builtins.max}
        6    0.000    0.000    0.002    0.000 {built-in method builtins.print}
        1    0.000    0.000    0.000    0.000 <ipython-input-150-62b8b9fa0010>:7(<module>)
    28016    0.955    0.000    8.110    0.000 <ipython-input-117-782b83d006db>:75(val_matrix)
    28016    0.062    0.000    8.206    0.000 <ipython-input-141-44fc476619bd>:13(__init__)
   210401    0.798    0.000    1.098    0.000 <ipython-input-117-782b83d006db>:97(compressor)
   182385    2.554    0.000    5.386    0.000 <ipython-input-117-782b83d006db>:26(outcome)
        1    0.000    0.000    9.027    9.027 <ipython-input-150-62b8b9fa0010>:5(<module>)
   182385    0.246    0.000    1.206    0.000 <ipython-input-117-782b83d006db>:67(checkHistory)
      121    0.003    0.000    0.368    0.003 <ipython-input-141-44fc476619bd>:59(expand)
    28036    0.264    0.000    8.617    0.000 <ipython-input-141-44fc476619bd>:45(proceed)
    28016    0.034    0.000    0.108    0.000 <ipython-input-141-44fc476619bd>:77(update)
    10993    0.064    0.000    8.523    0.001 <ipython-input-141-44fc476619bd>:68(selective)
        1    0.023    0.023    9.026    9.026 <ipython-input-141-44fc476619bd>:124(lookForward)
       11    0.000    0.000    0.000    0.000 <ipython-input-141-44fc476619bd>:144(<lambda>)
    10615    0.007    0.000    0.007    0.000 <ipython-input-141-44fc476619bd>:82(<listcomp>)
    10993    0.183    0.000    0.205    0.000 <ipython-input-137-230da75ee5d0>:1(PromAndRandom)
    10993    0.011    0.000    0.011    0.000 <ipython-input-137-230da75ee5d0>:3(<listcomp>)
   324794    2.201    0.000    2.428    0.000 <ipython-input-117-782b83d006db>:1(cluster)
    17401    0.004    0.000    0.056    0.000 /usr/local/lib/python3.6/dist-packages/numpy/core/_meth
    28016    0.030    0.000    0.111    0.000 <__array_function__ internals>:2(copyto)
```