

Final Year Project Report

Full Unit – Final Report

GAME PLAYING WITH MONTE-CARLO SEARCH

Hyunwoo Kim

A report submitted in part fulfilment of the degree of

BSc (Hons) in Computer Science

Supervisor: PROF CHRIS WATKINS



Department of Computer Science
Royal Holloway, University of London

March 31, 2023

Declaration

This report has been prepared on the basis of my own work. Where other published and unpublished source materials have been used, these have been acknowledged.

Word Count: 13027 words

Student Name: Hyunwoo Kim

Date of Submission: 31 March 2023

Signature: Hyunwoo Kim

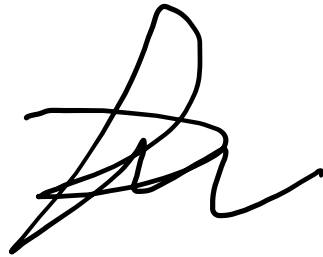
A handwritten signature in black ink, consisting of a large, stylized 'H' followed by a cursive 'K' and a trailing flourish.

Table of Contents

Abstract	6
Chapter 1: Introduction.....	7
1.1 Aims and Goals of the Project	8
1.2 Project Motivation	8
1.3 Milestones Summary	8
Chapter 2: Turn-Based Games and MCTS	10
2.1 What is the Turn-based games?.....	10
2.2 Alpha-beta pruning and its limitations.....	10
2.3 Connect 4.....	11
2.3.1 Rules of Connect4	12
2.3.2 Suitability of connect4 for AI Using MCTS	15
Chapter 3: Methods for Bandit Problems.....	16
3.1 What are the bandit problems?.....	16
3.2 Bandit Algorithms.....	16
3.2.1 Greedy Algorithm.....	16
3.2.2 E(Epsilon) - Greedy Algorithm.....	17
3.2.3 UCB (Upper Confidence Bound) Algorithm.....	17
Chapter 4: Algorithm for Monte-Carlo Search.....	19
4.1 Algorithm.....	19
4.1.1 Basic approach	19
4.1.2 Pseudo-code	20
4.2 Using UCB in MCTS	20
Chapter 5: Proof of Concept Development.....	22
5.1 Software Engineering	22
5.1.1 Test Driven Development.....	22
5.2 Connect Four built in Python	22
5.2.1 Text – Based Connect Four.....	23
5.2.2 Connect Four using GUI.....	25

5.2.3	Game playing against AI using MCTS	25
5.3	Source Code Documentation.....	26
5.3.1	main.py	26
5.3.2	game.py	27
5.3.3	board.py	28
5.3.4	mcts.py	30
5.3.5	node.py	33
5.3.6	gui.py	34
Chapter 6:	Experiment and Result Analysis	36
6.1	Finding the optimal exploration constant in UCB.....	36
6.1.1	Experiment setup.....	37
6.1.2	Procedures of experiment	37
6.2	Effect of other parameters	41
6.2.1	Number of simulations / rollouts	41
6.2.2	Number of iterations of MCTS process	43
6.3	Performance evaluation of the MCTS algorithm.....	44
6.3.1	Analysis parameters: number of iterations and number of simulations / rollouts 44	
6.4	Discussion of the results and their implications	45
6.4.1	Summary of results.....	45
6.4.2	Optimal parameters	45
Chapter 7:	Conclusion and Further Work	47
7.1	Summary of the achievements	47
7.2	Professional issues	48
7.2.1	Licensing.....	48
7.2.2	Usability	48
7.2.3	Time Management.....	48
7.2.4	Plagiarism	49
7.3	Possible further research and development	49
7.3.1	Exploration of Optimal Parameters for Different Games	49
7.3.2	Enhanced Experimentation for Parameter Optimization	50

7.3.3 Improved Simulation Algorithms.....50

7.3.4 Scalability and Parallelization50

Appendix..... 51

 User Manual 51

 System Requirements51

 How to run.....51

 Link to demo video.....51

Bibliography..... 52

Abstract

A Game AI (Artificial Intelligence) has been a prominent research topic in computer science for decades, with the goal of creating intelligent agents capable of playing games and solving complex problems. The development of game AI aims not only to increase the win rate of game-playing agents but also to improve the foundational algorithms of AI and enhance its ability to engage more challenges.¹

In recent years, turn-based strategy board games, such as Chess, Shogi, and Go, have emerged as popular targets for game AI research. These games share common attributes: they have simple yet clear rules and well-defined win/loss conditions, making them excellent testbeds for AI algorithms. Since the introduction of the minimax algorithm by John Von Neumann in 1928, there have been significant advancements in AI algorithms. However, more complex games like Go necessitated the development of more sophisticated algorithms due to the uncountable number of possible moves at each step.

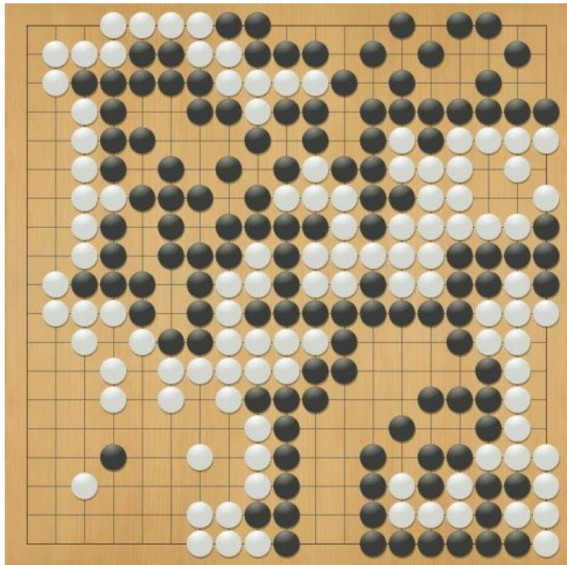
²This challenge led to the introduction of the Monte Carlo Go by Bernd Brügmann in 1993, which leveraged the Monte Carlo method as a primary search strategy for complex board games. The Monte Carlo method uses random numbers to determine the proportion of each step. ³Researchers have since applied this method to tree search, resulting in the Monte Carlo Tree Search (MCTS) algorithm. The MCTS algorithm aims to select the best move for expansion based on samples from random playouts. It comprises four fundamental stages: 1) Selection - selecting a node for expansion, 2) Expansion - creating child nodes, 3) Simulation - completing a rollout from a selected node, and 4) Backpropagation - updating the result to the root.⁴

In this project, we aim to develop a simple turn-based strategy board game, such as Connect 4, and create a game AI that employs the MCTS algorithm. The game AI will be utilised in experiments to measure its performance through gameplay. The experiments will include varying the number of iterations, simulations, and exploration constants to assess their impact on the performance of the MCTS algorithm. By analysing the results, we will evaluate the effectiveness of the MCTS algorithm and determine the optimal parameters for the MCTS through experimentation.

In conclusion, this study seeks to join the ongoing advancement of game AI programming by evaluating the performance of the MCTS algorithm in a simple turn-based strategy board game and identifying optimal parameters for its application. By building upon these findings, it will be possible to continue to develop more efficient and effective AI solutions for a wide range of complex problems, leading the way for the next generation of intelligent game-playing agents.

Chapter 1: Introduction

'Go' is a turn-based board game with a rich history, having originated in China approximately 2,500 years ago. Despite its seemingly simple rules, Go has been widely recognised as one of the most complex and strategically difficult board games in existence. This property of Go has posed significant challenges for researchers and AI developers, who have worked tirelessly to create intelligent agents capable of competing against professional human Go players. Despite advanced improvements in AI and game theory, the development of a Go AI program that can consistently outperform human experts remained as an elusive goal for many years.



Year	Handicap	Human level	Computer program
2008	9	8 dan	MoGo
2008	8	4 dan	Crazy Stone
2008	7	4 dan	Crazy Stone
2009	7	9 dan	MoGo
2009	6	1 dan	MoGo
2010	6	4 dan	Zen
2012	5	9 dan	Zen
2013	4	9 dan	Crazy Stone
2015	0	2 dan	AlphaGo Fan

Figure 1. The 1st 19*19 computer Go program won against professional human Go player.⁵

However, the performance of Go AI has made significant transformation with the advent of a new program called 'MoGo' in 2007. This marked a historic milestone as MoGo became the first AI to defeat a professional human player on a small-sized (9x9) Go board. This breakthrough rapidly accelerated the development and performance of Go AI programs. By 2008, MoGo had achieved another notable victory, triumphing over a human professional Go player on a full-sized (19x19) board with nine handicaps.

Seven years later, another Go AI called 'AlphaGo' made global headlines when it demonstrated an unprecedented level of performance by defeating a professional player without any handicap. This remarkable achievement showed the rapid advancements in AI and game-playing algorithms that had taken place in just a short period of time.

One of the most crucial factors that distinguished Go AI programs developed after 2007 from their previous models was the implementation of the Monte Carlo Tree Search (MCTS) algorithm. This innovative approach has awakened the field of AI for complex strategy games, enabling the creation of more sophisticated and competitive game-playing agents. The success of MCTS-based AI programs, such as MoGo and AlphaGo, has demonstrated the enormous potential of this powerful algorithm to engage even more challenging game environments⁶

1.1 Aims and Goals of the Project

In this project, the primary objective is to develop a turn-based game, 'Connect 4,' and implement an AI capable of playing the game using the Monte Carlo Tree Search (MCTS) algorithm to determine its moves. The summarised goals of the project are as follows:

1. Define the game rules of Connect 4.
2. Define formula of UCB in terms of game tree in Connect 4.
3. Make successful implementation MCTS for Connect 4
4. Apply MCTS and develop an AI for Connect 4.

1.2 Project Motivation

The implementation of MCTS has been a critical factor in the remarkable advancements of Go AI programs since 2007. As a result, today's Go AI programs have evolved to the point where they can consistently defeat professional human players on a full-sized Go board without any handicaps. Consequently, it is widely believed that human players are no longer capable of achieving victory against AI opponents in Go.

The dramatic and rapid improvement of Go AI, largely contributable to the implementation of MCTS, had attention by numerous researchers and scientists. Consequently, understanding the principles of MCTS and its applications has become increasingly important over time. Given the versatility of MCTS and its potential use in various fields, it is now crucial to engage into the study of MCTS and explore its possible applications.

1.3 Milestones Summary

The following is a summary of the key milestones for this project:

- **Build proof of concept programs – Connect4**
 - Build Connect4 that enables 2 human players to play
 - Build Connect4 that enables playing against AI
 - Apply GUI for game play
- **Build AI for selected game – Connect4 AI with Monte-Carlo Search**
 - Setup classes for AI (Node, MCTS)
 - Setup UCB: calculates UCB to choose the best valuable node
 - Setup MCTS algorithm for Connect 4
 - Implement fully functioning AI using MCTS for connect 4
- **Evaluation – Parameters of MCTS**

- Exploration constant (known as c-value)
- Number of simulations / rollouts
- Number of iterations of whole MCTS process

Chapter 2: Turn-Based Games and MCTS

2.1 What is the Turn-based games?

Game playing has been a popular area of research in artificial intelligence (AI), as it provides a challenging and interesting domain in which to test and develop AI algorithms. Turn-based games, in particular, have been a popular focus of research due to their simple and well-defined rules, and the fact that they are more amenable to game-tree search algorithms such as alpha-beta pruning and Monte-Carlo Tree Search (MCTS). In this section, we will provide an overview of game playing and AI, focusing on turn-based games as a prominent domain for AI research.

A turn-based game is a game in which each player takes turns making moves, with the objective of achieving a certain goal or outcome. Examples of turn-based games include chess, connect four, and tic-tac-toe. These games are often studied in AI research because they have clear and well-defined rules, which makes them an ideal domain for developing and testing AI algorithms.

One of the most popular techniques for game playing is the minimax algorithm, which is a search algorithm that seeks to minimise the maximum possible loss. In other words, it assumes that the opponent is playing optimally and selects a move that minimises the maximum possible loss for itself. However, the minimax algorithm has limitations, such as its inability to handle games with large branching factors, where the number of possible moves is too large to search exhaustively.

To overcome the limitations of the minimax algorithm, researchers have developed more advanced search algorithms, such as alpha-beta pruning and MCTS. Alpha-beta pruning is an extension of the minimax algorithm that uses heuristics to prune parts of the game tree that are unlikely to lead to a better outcome. This reduces the number of nodes that need to be evaluated and improves the efficiency of the search. Alpha-beta pruning has been used successfully in many turn-based games, such as chess.

MCTS, on the other hand, is a newer search algorithm that has become increasingly popular in recent years, particularly in the domain of board games. The basic idea behind MCTS is to simulate many random games from the current position, and then use the results of these simulations to guide the search for the best move. Unlike alpha-beta pruning, MCTS does not require any domain-specific knowledge or heuristics, making it more flexible and adaptable to different types of games.

In addition to search algorithms, AI researchers have also developed other techniques for game playing, such as machine learning and decision trees. Machine learning involves training a computer program to learn from past games, and use this knowledge to improve its playing strategy. Decision trees, on the other hand, are a type of rule-based system that can be used to make decisions based on the current game state.

In this project, a turn – based board game called Connect 4 is used to show the proof of concept program.

2.2 Alpha-beta pruning and its limitations

Alpha-beta pruning is a classic game-playing algorithm that has been widely used in computer games and artificial intelligence research. The basic idea of alpha-beta pruning is to eliminate the need to search the entire game tree by cutting off branches that are known to be irrelevant. Alpha-beta

pruning is an improvement over the minimax algorithm, which searches the entire game tree to determine the best move.

The alpha-beta pruning algorithm works by maintaining two values, alpha and beta, which represent the best values found so far for the maximizing player and the minimizing player, respectively. At each level of the game tree, the algorithm recursively explores the possible moves and updates the alpha and beta values accordingly. If the algorithm finds a move that leads to a worse outcome than the current best move, it cuts off that branch and returns immediately. This saves computation time by avoiding the exploration of irrelevant branches of the game tree.

While alpha-beta pruning is an effective algorithm for many games, it has several limitations. One of the main limitations of alpha-beta pruning is its reliance on a static evaluation function to estimate the value of the game state. The evaluation function is used to evaluate the strength of a game state and to compare it to other game states. The evaluation function must be carefully designed to capture the important features of the game and to be efficient to compute. However, this is a challenging task and may require significant domain expertise. In contrast, Monte Carlo Tree Search (MCTS) does not rely on a static evaluation function and can learn to evaluate game states based on simulated playouts.

Another limitation of alpha-beta pruning is that it assumes that both players are playing optimally. This is not always the case in practice, as players may make mistakes or may have different strategies. As a result, the alpha-beta pruning algorithm may prune a branch of the game tree that would have led to a better outcome if the opponent had made a mistake.

Additionally, alpha-beta pruning can be limited by the depth of the search. As the search depth increases, the computation time required to search the entire game tree becomes prohibitively large. Therefore, alpha-beta pruning must be limited to a certain depth, which can lead to suboptimal moves if the depth limit is too shallow.

Despite its limitations, alpha-beta pruning is still widely used in game playing and AI research. Many variants and improvements of alpha-beta pruning have been proposed to address its limitations, such as transposition tables, quiescence search, and aspiration windows. However, these improvements can add significant complexity to the algorithm and may not always be effective.

In contrast, MCTS is a newer algorithm that does not rely on a static evaluation function and can learn to evaluate game states based on simulated playouts. MCTS has been shown to be effective in many games, including Go, which has a much larger branching factor than other board games such as Chess and Connect4. MCTS has also been shown to be robust to imperfect information and stochasticity in the game, making it a promising algorithm for many real-world applications.

2.3 Connect 4

⁷Connect 4, released by Milton Bradley Company in 1974 under name ‘Connect Four’, is a turn – based game played by two players. Most of the formats in Connect 4 are similar to the game ‘Tic – Tack – Toe’.

The game Connect 4 consists of three main features; Board, Pieces, and Players.

The ‘board’ of Connect 4 has 6 rows and 7 columns.

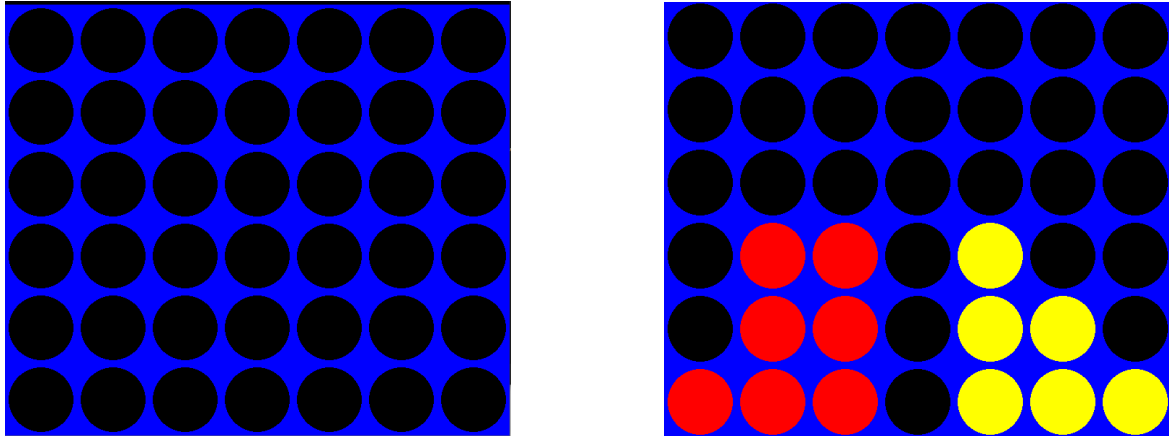


Figure 2. Empty Connect 4 board(left) and board filled with few Red and Yellow stones(right).

The game is run by two ‘players’, who has different colour of ‘pieces’ each other; for example, Figure 2 shows the players using different colours – in case of Player 1 went first and uses Red Pieces, all the Red pieces are placed by Player 1 and all the Yellow pieces are place by player 2.

In this project, the Player 1, who can place a piece first, uses Red colour and Player 2 uses Yellow pieces. The Players cannot change either the colour of the piece which is already placed on the board, or will be placed on the board.

Each Players can place only 1 piece at each turn. If one places a piece, the turn changes and another Player should place a piece for next. Unlike the games ‘Go’ or ‘Gomoku’, the Players are not allowed to add their pieces at any empty places, but can only ‘drop’ the pieces vertically to the lowest row of the chosen column. If the column is full of pieces, the player needs to choose from other columns.

2.3.1 Rules of Connect4

The object of Connect 4 is to make four same colours of pieces in a row. The row could be in 4 different shapes; 1. Horizontal [()], 2. Vertical [-], 3. Diagonal (1) [/], and 4. Diagonal (2) [\] (See Figure 3).

2.3.1.1 Winning states

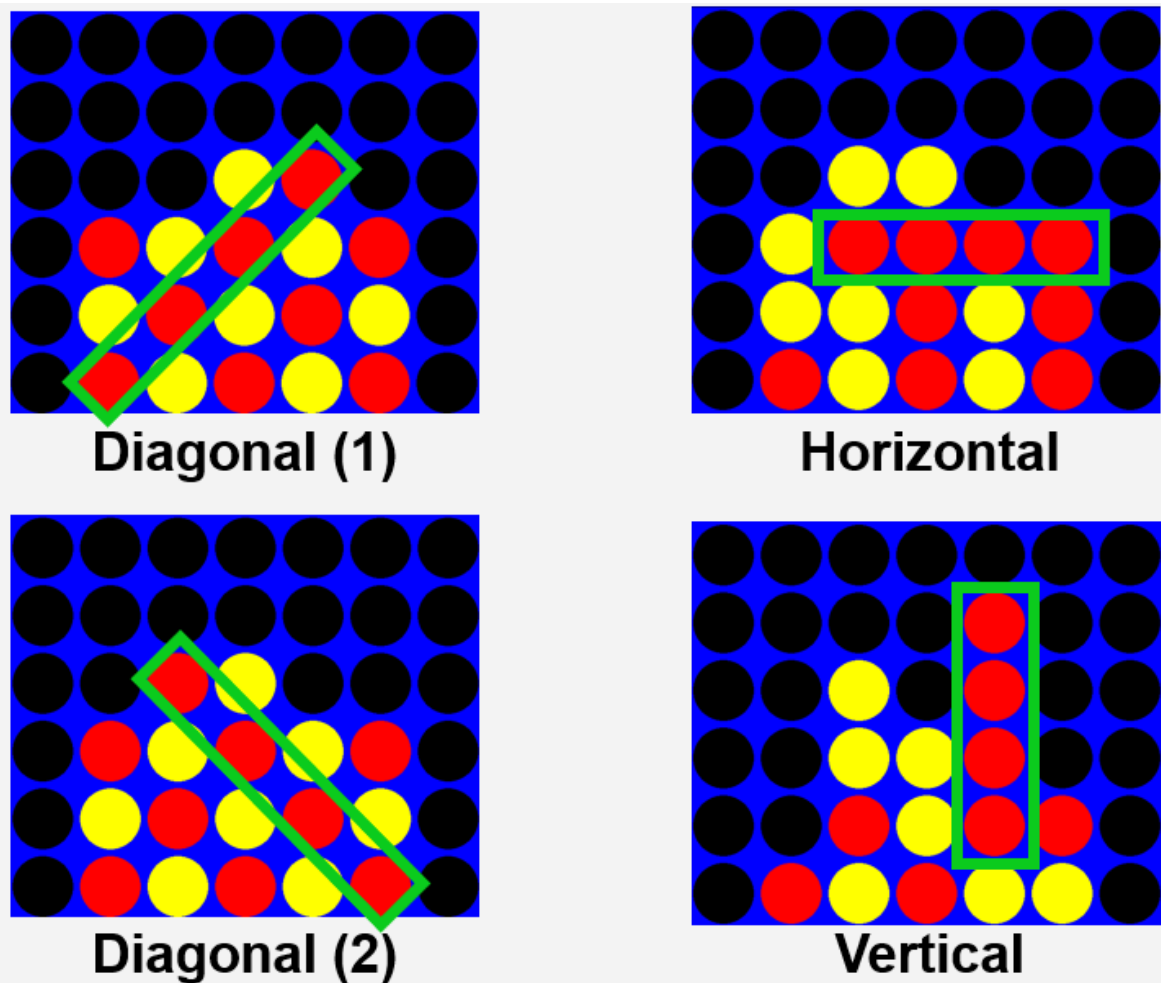


Figure 3. 4 different cases of winning states in Connect 4

If a Player makes a row of 4 same coloured pieces in any shape described above, the Player wins and the game overs.

At each player's end of the turn, the game checks if there is a 'winning state' on the board. To be more effective, the verification of whether there is a '4 pieces in a row' differs on the shape of the row.

```
def check_status(board, player):
    """Checks whether there is any win state on the board. ...

    # 1 Horizontal (-)
    for col in range(COLUMN - 3):
        for row in range(ROW):
            if board[row][col] == player and board[row][col + 1] == player and \
                board[row][col + 2] == player and board[row][col + 3] == player:
                return True

    # 2 Vertical (|)
    for col in range(COLUMN):
        for row in range(ROW - 3):
            if board[row][col] == player and board[row + 1][col] == player and \
                board[row + 2][col] == player and board[row + 3][col] == player:
                return True

    # 3 Diagonal (/)
    for col in range(COLUMN - 3):
        for row in range(ROW - 3):
            if board[row][col] == player and board[row + 1][col + 1] == player and \
                board[row + 2][col + 2] == player and board[row + 3][col + 3] == player:
                return True

    # 4 Diagonal (\)
    for col in range(COLUMN - 3):
        for row in range(3, ROW):
            if board[row][col] == player and board[row - 1][col + 1] == player and \
                board[row - 2][col + 2] == player and board[row - 3][col + 3] == player:
                return True
```

For instance, in case of horizontal row: There is no need to check all 7 pieces; It only needs to check based on first 4 pieces from left on the horizontal row as there are only 4 cases of possibility as described below:

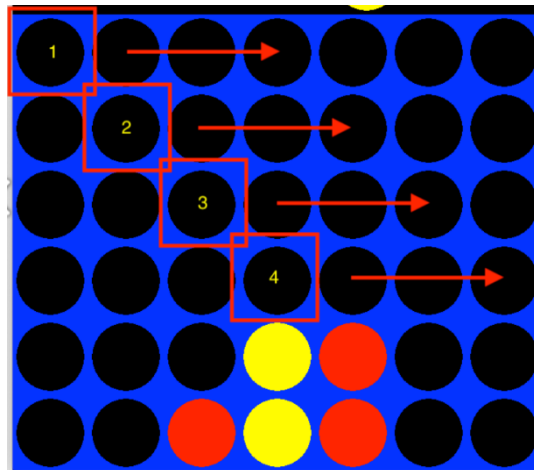


Figure 4. 4 cases of verification for horizontal winning state.

Similarly, the verification for a vertical row could be done by checking only 3 pieces from bottom in a vertical row.

The diagonal states could be divided into two shapes; Diagonal (1), which has its piece on the left corner lower than the piece on the right corner, and Diagonal (2) which is opposite; left corner to be higher than the pieces on the right. (see Figure 5).

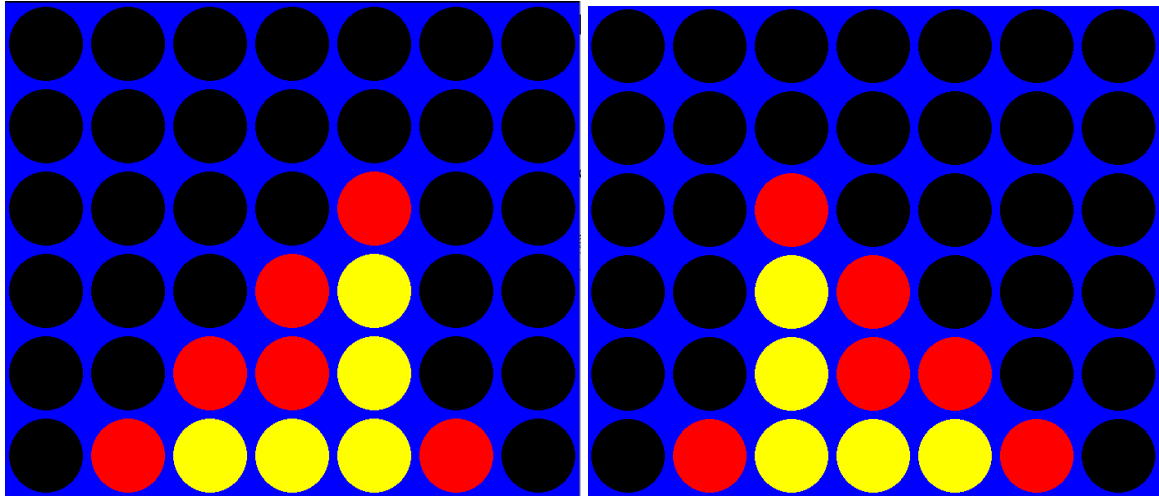


Figure 5. Red's winning state by Diagonal 1 (left) and Diagonal 2 (right).

2.3.1.2 Draw state

The Game ends draw either in situations described below:

1. There are no empty places left to put pieces and no 'winning state' on board.
2. Although there are some empty places left on board, there is no such a way to make 'winning state' by either Player 1 or Player 2.

2.3.2 Suitability of connect4 for AI Using MCTS

In this project, the game of Connect 4 has been selected as the basis for a proof-of-concept program, and several factors contribute to this decision. A primary consideration is the relatively simpler gameplay mechanics in comparison to other turn-based games, such as the ancient game of Go. The reduced complexity of Connect 4 lends itself to a more manageable environment for applying algorithmic approaches.

For instance, the number of potential moves in Go is directly proportional to the number of empty positions on the board. Given that the Go board comprises a 19x19 grid, the initial player has precisely 361 distinct options for their opening move. In contrast, Connect 4 allows players only to drop their respective game pieces vertically within a selected column. Consequently, the maximum number of available moves corresponds to the total number of columns on the game board, which amounts to seven.

As a result, when implementing the Monte Carlo Tree Search (MCTS) algorithm in the context of Connect 4, the number of nodes requiring selection and evaluation is significantly reduced. This diminished computational costs with respect to the number of iterations and time required to achieve meaningful results.

Chapter 3: Methods for Bandit Problems

Before the development of Monte-Carlo Tree Search, there has been a number of studies to solve bandit problems. Not only for game AI, but also for other problems or algorithms, the development of bandit algorithms was crucial to improve their efficiency.

3.1 What are the bandit problems?

Choice and regret: Which one to choose to get **best rewards**?

Imagine a player visiting to a casino and try to play on slot machines. The player wants to get the best result as possible as he can. The player lacks of chips to try all the possible cases on every slot machine. Still, the player wants to know which machines are best to choose to get best rewards.

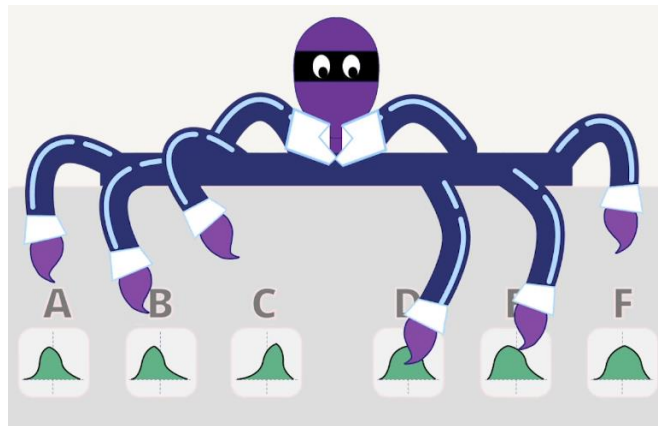


Figure 6. Illustrates bandit problems choosing machine from A to F (Beck, Jana).

⁸The Bandit algorithms could ‘recommend’ what would be the best options to choose for the player according to the calculation of explorations and their exploitations.

3.2 Bandit Algorithms

There are many different algorithms for multi-armed bandit problems. Each algorithm could be used in different environment as they all have advantages and disadvantages compare to each other. Below will describe 3 different bandit algorithms as examples; Greedy Algorithm, E-Greedy Algorithm, and UCB Algorithm.

3.2.1 Greedy Algorithm

Tests every machine and select the best one – not enough exploration.

```

GREEDY( $S$ )
  parameter( $s$ ):  $S$  – set of blocks
  output: superstring of set  $S$ 
  while  $\|S\| > 1$ 
    do  $\begin{cases} \text{choose } s_1, s_2 \in S \text{ such that } \text{overlap}(s_1, s_2) \text{ is maximal} \\ S \leftarrow (S \setminus \{s_1, s_2\}) \cup \{\text{merge}(s_1, s_2)\} \end{cases}$ 
  return (remaining string in  $S$ )
  
```


Figure 7. Pseudo code for Greedy Algorithm.⁹

The algorithm simply explores all the possible options(machines) at once, and compares which one returned the best rewards to decide a machine. If a machine is decided, the algorithm considers the machine as a ‘Best machine ever’ and will spend every coin to play on that machine only.

The main problem of Greedy algorithm is that was not enough exploration for choosing one ‘Best Machine’. For instance, consider Machine 1 and Machine 2. The Machine 1 is set to 80% of win rate and the Machine 2 is set to 30 % of win rate. The player tried both machines, and got Machine 1 lost and Machine 2 win in the first round. If the player uses greedy algorithm at this moment, it will choose Machine 2 as the best machine since Machine 2 gave higher rewards than Machine 1 based on explorations it had. Further exploration may differ the result, the selected machine may not be actual best machine.

3.2.2 E(Epsilon) - Greedy Algorithm

Test every machine and select the best one, but try another one by the rate of ‘epsilon’

- Still need to explore although already found best machine.

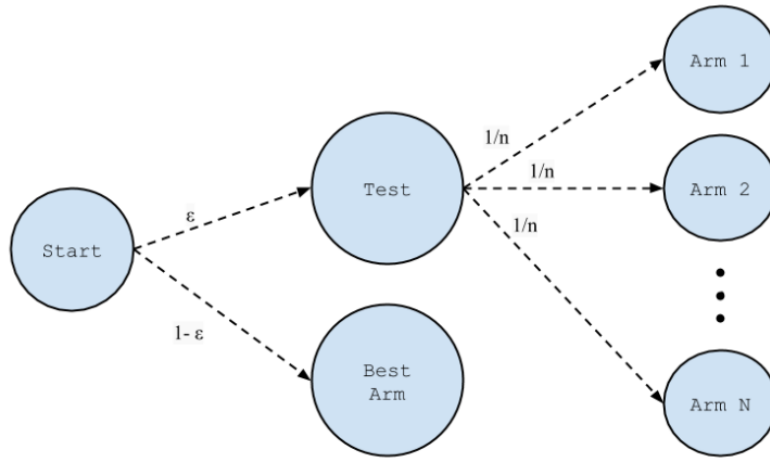


Figure 8. Simple illustration about how the Greedy algorithm works.

Unlike Greedy algorithm, E – Greedy algorithm keeps try to explore. The algorithm spends coins to the ‘best machine until now’ by the rate of $(1 - \epsilon)$, and explores other machines by the rate of ϵ . However, since the exploration is random as rate of ϵ , there would be still lack of exploration.

3.2.3 UCB (Upper Confidence Bound) Algorithm

To make better algorithm for exploration towards unexplored options, ‘UCB (Upper Confidence Bound) algorithm’ could be suggested. The UCB algorithm considers not only the number of rewards, but also the number of visits(simulation) it had. To make the UCB algorithm as a formula, it can be described as below;

$$A_t = \operatorname{argmax}_a \left(Q_t(a) + c \sqrt{\frac{\ln(t)}{N_t(a)}} \right)$$

Figure 9. basic UCB formula.

⁹The t stands for the number of selection of corresponding slot machine at certain time. $Q_t(a)$ stands for the total rewards of (a) at the time t . c is the hyper parameter which can decide the balance

between exploitation and exploration(higher c means more exploration). $N_t(a)$ stands the number of selection of corresponding slot machine(node) until time t .

¹⁰Now, to be used in a tree system, such as Monte Carlo Search Tree, UCT (Upper Confidence Boundary of Tree) could be used as tree policy. The UCB1 algorithm could be used in a tree system to be UCT:

UCB1 formula:

$$\text{UCB1} = \bar{X}_j + \sqrt{\frac{2 \ln n}{n_j}}$$

Figure 10. UCB1 Formula for UCT.

\bar{X}_j stands for the exploitation, which is the average reward from the child node j . This basically means the win rate of choosing the node j .

The term on right hand stands for the rate of exploration of child node j ;

n_j stands for number of visiting of child node j , and n stands for the number visiting of parent node of child node j .

Since n_j is the denominator, although it has smaller value of \bar{X}_j , which means the win rate of child node j is low, if the number of visiting (n_j) is low, the whole UCB1 value can increase and the node j could be selected at high rate.

Chapter 4: Algorithm for Monte-Carlo Search

In terms of game AI programming, if the ‘depth’ of a game tree and its number of branches are considerably low, techniques such as min-max can enhance the performance for the program. However, these methods, during their process, may increase the ‘memory size’ of game tree significantly. To mitigate this drawback, more efficient pruning methods, such as Alpha-Beta Pruning, have been proposed. Nonetheless, for games characterised by nearly infinite depth and an immense number of branches, exhaustive search approaches are infeasible. To overcome this crux, Monte – Carlo Tree Search has suggested as a new method.

4.1 Algorithm

The Monte Carlo Tree Search employs simulation to approximate the values of randomly chosen child nodes. As a best-first search method that incorporates randomised exploration, MCTS can determine the most accurate estimated solution without relying on a position evaluation function. The precision of these approximated values is variable upon the number of simulations conducted; consequently, a greater quantity of simulations yields higher accuracy in the resulting estimates.

4.1.1 Basic approach

The MCTS records the result of explorations to setup the game tree gradually. To apply MCTS in a game, the game should satisfy the following conditions;

1. There should be limitation of reward value: the game should have minimum and maximum score value
2. The game should have game rules, and it should be complete information game
3. There should be time limitation for the game so that the simulation could be ended in certain time.

To perform the MCTS algorithm the program should have;

1. Search Tree: saves information to be used for simulation
2. Game Tree: saves information for nodes which are created during the game playing.

¹¹The MCTS starts by initialising the game tree, and continues to perform iterations of the following 4 steps of algorithm which are described in the figure 11 in the given time. The steps are; 1. Selection, 2. Expansion, 3. Simulation, and 4. Backpropagation.

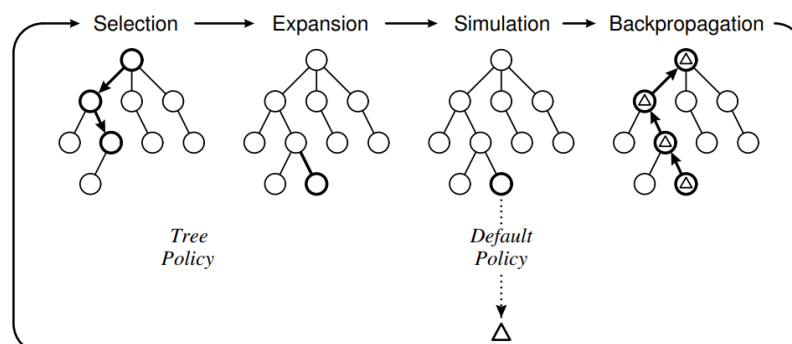


Figure 11. One iteration of the MCTS algorithm.

1. Selection: The algorithm starts from the root node. By using child selection policy, it recursively evaluate nodes which are not visited(unexpanded) and not in terminal state, and finally find the most urgent node.
2. Expansion: The node which is selected from Selection step are added to the Search Tree and ready to expand.
3. Simulation: Using default policy, the simulation is run from the node that is added in Expansion.
4. Backpropagation: It returns and updates the result from the simulation and send through the route as the node added.

4.1.2 Pseudo-code

The following figure describes simple Pseudo-code for the MCTS algorithm:

```

function MCTS(Position, NoOfSimulations)
  for each child n available from Position
    reward[n] = 0
    for i from 1 to NoOfSimulations
      POS = copy(Position)
      while (POS is terminal node)
        MCTS(POS, random child from POS)
      end while
      reward[n] += result(POS)
    end for
    reward[n] /= NoOfSimulations
  end for
  return(child i with highest reward[i])
end function

```

Figure 12. Simple Pseudo-code describing MCTS.

4.2 Using UCB in MCTS

Nowadays, UCB1 algorithm is generally used in MCTS as described in the figure 13 below;

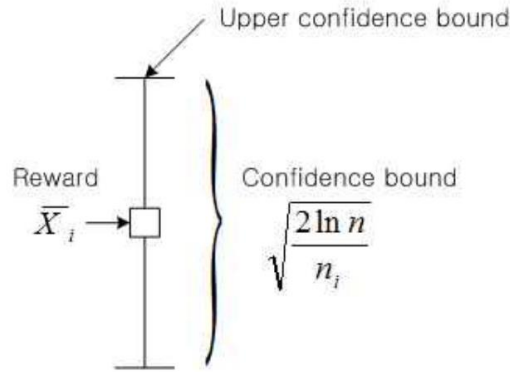


Figure 13. Rewards and Confice bound in UCB1.

\bar{X}_j is used to calculate the actual win rate of the selected child node, and the confidence bound

$\sqrt{\frac{2 \ln n}{n_j}}$ is used to calculate the rate of ‘how valuable is the node’ to be visited.

Particularly, in this project, the acutal UCB algorithm has written as following code;

```
def calc_uct(self, c=None):
    """Calculates UCB value for the node. ...
    if self.visits == 0:
        return None

    if c == None:
        c = np.sqrt(2)

    ucb = (self.wins/self.visits) + c * np.sqrt(np.log(self.parent.visits)/self.visits)
    return ucb
```

The \bar{X}_j is calculated as the selected node’s (amount of wins / number of visits), and $c * \sqrt{\frac{\ln n}{n_j}}$ is calculated as $c * \sqrt{\frac{\log(\text{number of visits from parent node})}{(\text{number of visits from the child node})}}$.

Chapter 5: Proof of Concept Development

5.1 Software Engineering

5.1.1 Test Driven Development

In this project, a development method called ‘TDD (Test Driven Development)’ is mainly used to program the Connect 4.

¹²There are 3 phases for Test Driven Development;

1. Red phase: Write a test that doesn’t work.
2. Green phase: Make the failed test work. Faking the test is allowed.
3. Blue (Refactor) phase: Remove duplicated code and make the code work by refactoring.

The most important factor during the TDD process is that the actual working code should not be written before writing test that fails, and then refactor the code to be working. There are several advantages of using Test Driven Development:

1. Decreases time taken in debugging: Same reason as using unit test, since the Test Driven Development working similar to unit test, it is easier to find specific errors.
2. Saves time taken in setting up structure: Since TDD writes test first than the actual code, the structure of the code starts from basic concepts. Therefore, it would have lower chance of changing the whole structure of the code in the future.
3. Stable code: Since all the code is continuously written under actual tests, the code is stabilised than the code which is written without tests.

5.2 Connect Four built in Python

The Python programming language supports numerous powerful libraries and tools such as Numpy. There are 4 main reasons why the program is built in Python;

1. Simple: Since the code written in Python language is simple and clear, it is relatively easier to be read and used. Thus, a number of deep learning or machine learning projects could have been written in Python recently.
2. Fast: Python code is relatively short compare with other languages such as C or Java. Due to this aspect, it could be learnt in less time and this could lead to the result in faster development of the program overall.
3. Code Test: Python supports various code tests such as unit test. Since the AI project needs an environment which is able to test countless simulations and checks if there are any system faults, Python has been chosen to make sure of faster tests and better accuracy.

4. Visualisation tools: Various libraries in Python supports visualisation for the developers with tools which are relatively easier and simpler to be used.

5.2.1 Text – Based Connect Four

Initially, the goal was set to develop Connect Four game which enables to be played by 2 human players without AI movements.

The program consists of 2 main parts; Board, and Game.

1. Board.py: The board is designed to represent the actual game board of Connect 4. Basically, the board.py defines the length of rows and columns, size of the board, and the number of stones placed on the board during the game play. The board is a type of numpy.ndarray representing 2-dimensional array of list built in size of 42 blanks; 6 rows and 7 columns. The board.py contains basic functions that use the game board in connect 4, such as;
 - i. Creates an empty board.
 - ii. Prints the board on the terminal console.
 - iii. Gets the next player.
 - iv. Gets the index of row which is empty in the given column.
 - v. Gets the indices of column which has at least 1 empty space.
 - vi. Checks if the board has any 'winning state'.
 - vii. Adds a stone to the board.
 - viii. Checks if the board is full or not.

```
1  import numpy as np
2
3  ROW = 6
4  COLUMN = 7
5
6  > def create_board(): ...
13
14 > def print_board(board): ...
32
33 > def next_player(board): ...
47
48 > def get_free_row(board, column_index): ...
62
63 > def get_free_columns(board): ...
79
80 > def check_status(board, player): ...
118
119 > def add_stone(game_board, column_index): ...
154
155 > def is_board_full(board): ...
165
166 > def get_player_marker(board): ...
```

2. Game.py: The game object is initialised by creating a board. Due to the input from main.py, it selects which version of connect 4 will be used. It contains 5 different ways of running connect 4 in terms of players and GUI:
 - i. Enables game play by 2 human players in command line(text version)
 - ii. Enables game play by 2 human players with GUI
 - iii. Enables playing against AI using MCTS in command line (text version)
 - iv. Enables playing against AI using MCTS with GUI
 - v. Enables 2 AI players play against each other using MCTS in command line


```

class Game:
    """Runs the connect 4 game in 3 ways.

    There are 3 versions of connect 4:
    1. Play by 2 human players, 2. Play against AI, 3. AI against AI
    """
    def __init__(self): ...

    def play_text(self): ...

    def play_gui(self): ...

    def play_gui_mcts(self, first_player): ...

    def play_mcts(self, first_player, n_of_iterations, c, n_of_simulations=3): ...

    def auto_mcts(self, n_of_games, n_of_iterations, c1, c2, n_of_simulations1=1, n_of_simulations2=1): ...

```

5.2.2 Connect Four using GUI

In terms of better user experience, the Pygame is used to visualise the game play for the players. The Gui.py contains variables that define; colours, font, size of boxes, screen width, screen height. It is designed to get position of mouse pointer from the player, to make it as an input column index when the player clicks the left button of the mouse.

```

class Gui:
    """Run GUI version of Connect 4.
    Uses Pygame to draw Graphics on pop up screen.
    """
    pygame.init()

    def __init__(self): ...

    def drawScreen(self, x_pos, winner, player, using_AI=False, wrong_input=False): ...

    def is_over(self, winner): ...

    def run_mcts(self, first_player): ...

    def runGame(self): ...

```

5.2.3 Game playing against AI using MCTS

Based on the game created in 5.2.1, it adds two files to apply MCTS; node.py and mcts.py.

The object Node in node.py consists of 7 variables; board, won_player(if there is any winner in this node), col(column index of the node), parent(parent Node), children(children Nodes), wins(number of wins from this Node), and visits(number of visits in this Node). In node.py, it contains important functions for node, such as;

1. Calculating UCB value of the node with give exploration constant
2. Choosing a child node which has the highest win rate.

```

class Node:
    """A class to represent nodes for the game tree in MCTS.
    """
    def __init__(self, board, won, col, parent):
        """Init by setting the attributes for selected node...
        self.board = board
        self.won_player = won
        self.col = col
        self.parent = parent

        self.children = None
        self.wins = 0
        self.visits = 0

    def calc_uct(self, c=None): ...

    def add_child(self, child): ...

    def choose_node(self): ...

    def get_node(self, col): ...

```

The mcts.py defines the whole process of Monte Carlo Tree Search. It contains methods for each phase; 1. Selection, 2. Expansion, 3. Simulation, 4. Backpropagation.

```

from connect_four.board import *
from connect_four.node import Node

> def random_move(input_board): ...

> def selection(node, c): ...

> def expansion(node, free_columns): ...

> def simulation(node, n_of_simulations=1): ...

> def backpropagation(node, simulation_winner): ...

> def mcts(root=None, c=None, n_of_simulations=1): ...

```

5.3 Source Code Documentation

5.3.1 main.py

Enables the player to select the game play either playing with two human players or against AI.

Player can select whether play on terminal (text – based version) or GUI.

For future experiment, it also enables two AI with different MCTS parameters to play against each other automatically.

5.3.2 game.py

Run the connect 4 games in 3 ways.

There are 3 versions of connect 4:

1. Play by 2 human players, 2. Play against AI, 3. AI against AI

1. `play_text(self)`

Run the text-based Connect 4 on command line.

Play by 2 human players

2. `play_gui(self)`

Run the GUI version of Connect 4 powered by Pygame.

Play by 2 human players.

3. `play_gui_mcts(self, first_player):`

Run the GUI version of Connect 4 powered by Pygame.

Play against AI.

Args:

`first_player (int)`: 1 if the human player play first, -1 if AI play first

4. `play_mcts(self, first_player, n_of_iterations, c, n_of_simulations=3):`

Run the Text version of Connect 4 for playing against AI

Args:

`first_player (int)`: 1 if the human player play first, -1 if AI play first

`n_of_iterations (int)`: number of iterations

`c (float)`: exploration constant(`c_value`)

`n_of_simulations (int, optional)`: number of simulations. Defaults to 3.

5. `auto_mcts(self, n_of_games, n_of_iterations, c1, c2, n_of_simulations1=1, n_of_simulations2=1):`

Run the Text version of Connect 4 for AI against AI

Play by 2 AI against each other automatically.

Args:

`n_of_games (int)`: number of games to be played

n_of_iterations (int): number of iterations

c1 (int): exploration constant(c_value) for player 1

c2 (int): exploration constant(c_value) for player 2

n_of_simulations1 (int, optional): number of simulations for player 1. Defaults to 1.

n_of_simulations2 (int, optional): number of simulations for player 2. Defaults to 1.

Returns:

winner_cnt (list): list of number of wins for each player

p1_time_taken (list): list of time taken for each move by player 1

p2_time_taken (list): list of time taken for each move by player 2

5.3.3 board.py

Describes the game board for connect 4.

1. create_board():

Create a numpy array size of (6,7)

Returns:

numpy.ndarray: board size of 6*7.

2. print_board(board):

Print the game board on command line with column numbers.

Args:

board (numpy.ndarray): board size of 6*7

3. next_player(board):

Return the next player as an integer

Args:

board (numpy.ndarry): board size of 6*7

Returns:

int: 1 if the next player is 'O' else, -1d

4. get_free_row(board, column_index):

Return the position of free(empty) row in the given column index

Args:

board (numpy.ndarray): board size of 6*7

column_index (int): selected column index (from 0 to 6)

Returns:

int: position of empty row index

5. get_free_columns(board):

Return a list of column indices

The column indices inside the list should have at least 1 empty space

Args:

board (numpy.ndarray): board size of 6*7

Returns:

free_columns (list): list of column indices

6. get_free_row_index(self, column_index)

Returns the position of empty row index in the column

Args:

column_index (int): column number from 0 to 6

Returns:

int: empty row index (from 0 to 5) or -1 if the column is full

7. check_status(board, player):

Checks whether there is any win state on the board.

There are 4 types of winning states; Horizontal(-), Vertical (|), Diagonal(/) and another Diagonal(\).

If the board has any types of winning state described above, returns True.

Args:

board (numpy.ndarray): a board size of 6*7

player (int): integer 1 as a player 'O' and -1 as a player 'X'

Returns:

boolean: True if there is a winning state, if not, False.

8. add_stone(game_board, column_index):

Place a stone in the selected column.

It checks if there is an empty space in the given column index.

If there is, it places a stone and checks if there is any win pattern.

It returns a board after placing a stone and player if the player wins.

It returns 0 for player if there is no winner

It returns 2 for player if the board is full.

Args:

game_board (numpy.ndarray): a board size of 6*7

column_index (int): selected column index (from 0 to 6)

Returns:

board (numpy.ndarray): a board size of 6*7

player (int):

0 if there is no winner,

1 if player 'O' wins,

-1 if player 'X' wins,

2 if the board is full.

9. is_board_full(board):

Check if the board is full or not.

Returns:

Boolean: True if the board is full, if not, False.

10. get_player_marker(board):

Return next player as a marker 'O' or 'X'

Args:

board (numpy.ndarray): a board size of 6*7

Returns:

String: 'O' if the next player is 1 else 'X'

5.3.4 mcts.py

Defines the process of MCTS in python.

1. random_move(input_board):

Place a stone in input board until the winner is found(final state).

Place a stone in a random column index of the input board and return player.

If player 'O' wins, return 1

If player 'X' wins, return -1

If the game ties, return 2.

If there is no winner, return 0.

Args:

input_board (numpy.ndarray): a board size of 6*7

Returns:

player (int): 1 if player 'O' wins, -1 if player 'X' wins, else 0

2. selection(node, c):

Selection process in MCTS

Select a child node with the highest UCB value.

If there is no UCB value in the child, return a random Node from children.

Args:

node (Node): root node in selection process

c (float): exploration constant for mcts (c_value)

Returns:

node (Node): child node with the highest UCB value.

3. expansion(node, free_columns):

Expansion process in MCTS

Expands every expandable columns from the selected node.

Then add all the expanded node as a child node for the selected Node.

Args:

node (Node): root node before expansion

free_columns (list): list of expandable column indices

4. simulation(node, n_of_simulations=1):

Simulation(rollout) process in MCTS

From the selected Node, simulate by randomly add stones

until a winner is found(game over).

If there is any child node which already has winner,

it returns winning child node immediately.

Returns original node before simulation and winner from the simulation.

Args:

node (Node): root node before simulation

n_of_simulations (int, optional): number of simulations/rollouts. Defaults to 1.

Returns:

node (Node): root node before simulation

simulation_winner (int): winner player(1 or -1 or 0(no winner)) from the simulation

5. backpropagation(node, simulation_winner):

Backpropagation process in MCTS

Backpropagate the results from the previous process by recording the nubmer of wins and visits.

Args:

node (Node): child node that has win/visits data

simulation_winner (int): winner player(1 or -1 or 0(no winner)) from the simulation

6. mcts(root=None, c=None, n_of_simulations=1):

MCTS process for Connect 4.

Start by setting the root node with empty board.

The process of MCTS consists of 4 phases:

1. Selection, 2. Expansion, 3. Simulations/rollouts, 4. Backpropagation

Args:

root (Node, optional): root Node. Defaults to None.

c (float, optional): exploration constant(c_value). Defaults to None.

n_of_simulations (int, optional): number of simulations. Defaults to 1.

Returns:

root (Node): root Node

5.3.5 node.py

A class to represent nodes for the game tree in MCTS.

1. `__init__(self, board, won, col, parent):`

Init by setting the attributes for selected node.

Args:

`board (numpy.ndarray)`: board size of 6*7

`won (int)`: integer value for estimated winning player.

`col (int)`: positional column index of the node.

`parent (Node)`: parent node of the current node.

`children (Node)`: children nodes.

`wins (int)`: number of wins.

`visits (int)`: number of visits.

2. `calc_uct(self, c=None):`

Calculates UCB value for the node.

Uses UCB formula to calculate the values. If there is no c value given, original c value from `uct1 (sqrt(2))` will be used.

Returns:

float: ucb value for the given node.

3. `add_child(self, child):`

Add child node.

Args:

`child (Node)`: Node that will be a child node for selected node.

4. `choose_node(self):`

Return a Node with the highest win rates.

If there is no child Node exist for the selected Node,

return None for both Node and column index.

If there is a winning Node, return the winning Node immediately.

Returns:

`best_node (Node)`: Node with the highest win rate

`best_node.col (int)`: column index of the best_node

5. `get_node(self, col):`

Return a child Node in the selected column index.

Returns None if there is no children for selected Node.

Args:

`col (int)`: column index

Returns:

`node (Node)`: child Node in the column index

5.3.6 gui.py

Run GUI version of Connect 4. Uses Pygame to draw Graphics on pop up screen.

1. `__init__(self):`

Init by creating a board set up attributes.

2. `drawScreen(self, x_pos, winner, player, using_AI=False, wrong_input=False):`

Draw connect 4 game play by graphics on the screen.

Args:

`x_pos (int)`: current horizontal position of mouse

`winner (int)`: an integer indicates who's winner

`player (int)`: current player

`using_AI (bool, optional)`: True if it is played by an AI. Defaults to False.

`wrong_input (bool, optional)`: True if the column at `x_pos` is full. Defaults to False.

3. `is_over(self, winner):`

Check if there is winner in game.

Args:

`winner (int)`: winner player(1 or -1 or 0(no winner))

Returns:

Boolean: True if there is any winner or draw, else False

4. `run_mcts(self, first_player):`

Play Connect 4 GUI version against AI

Get input from mouse pointer position which the player controls.

Args:

first_player (int): 1 if human player play first, -1 if AI play first

5. runGame(self):

Run Connect4 by using Pygame.

Get input from mouse pointer position which the player controls.

Chapter 6: Experiment and Result Analysis

In order to implement MCTS effectively, it is essential to determine the optimal parameters for the algorithm. A variety of parameters exist for MCTS, including the number of MCTS iterations, the number of simulations (rollouts), and time limits, among others. However, the most crucial parameter for optimizing MCTS is the exploration constant in the UCB algorithm.

This chapter will focus on conducting experiments to determine the optimal exploration constant for the UCB algorithm. After determining the ideal exploration constant, further experiments will investigate the impact of the other parameters on the efficiency of the MCTS algorithm.

All the experiments were conducted on a standard laptop computer with the following specifications:

- CPU: Intel Core i7 – 11800H @ 2.3GHz
- RAM: Samsung DDR4-3200 16GB (8GB * 2)
- Graphics: NVIDIA GeForce RTX 3060 Laptop GDDR6 6GB VRAM

The connect 4 MCTS program was implemented in Python 3.10.7.

6.1 Finding the optimal exploration constant in UCB

The exploration constant, also referred to as the 'c value' in MCTS, is a parameter that can be adjusted to strike a balance between the exploration and exploitation trade-off during the tree search process. This value determines the extent to which unexplored paths in the search tree are given weight compared to already explored nodes that show promising results.

Typically, the exploration constant is used during the selection phase of MCTS, where the algorithm selects the next node to expand by recursively traversing the tree from the root node. During this process, an Upper Confidence Bound (UCB) value is calculated for each child node of the current node, based on its average reward, the number of times it has been visited, and the exploration constant.

The UCB value is typically calculated using the formula:

$$UCB = Exploitation + C * \sqrt{\frac{\log(total_visits)}{(node_visits)}}$$

Figure ?. Simplified UCB algorithm.

where:

- exploitation is the average reward of the child node
- total_visits is the total number of times the parent node has been visited

- `node_visits` is the number of times the child node has been visited
- `C` is the exploration constant

The exploration constant determines the amount of exploration versus exploitation in the search. A higher value of `C` promotes more exploration by increasing the UCB value for nodes with fewer visits, while a lower value of `C` encourages more exploitation by prioritizing nodes with higher average rewards. The value of `C` should be tuned to balance these two objectives depending on the problem being solved.

Overall, the exploration constant is a crucial parameter in the effectiveness of MCTS, as it determines the balance between exploration and exploitation in the search process.

6.1.1 Experiment setup

In this step, the goal was set to determine the optimal value of the exploration constant (c-value) in the Monte Carlo Tree Search (MCTS) algorithm. To achieve this, the `'auto_mcts()'` method was utilised in `game.py`, where the MCTS algorithm was programmed to play against itself with different values of the exploration constant.

A range of exploration constant, varying from 0.5 to 2.0, was tested by having the algorithm play 100 games against each other for each value of exploration constants. This sample size was determined to provide enough statistical power to identify any significant differences in performance between different c-values.

The algorithm was evaluated based on its win rate, defined as the percentage of games won by the algorithm, and by examining the average number of nodes visited in the MCTS tree during a game, as well as the average time taken to make a move.

To ensure fairness, the algorithm played as both the first and second player an equal number of times. Additionally, other parameters such as the number of iterations of the MCTS process fixed at 50, and the number of simulations (rollouts) fixed at 20 per iteration, were kept constant to minimise bias.

6.1.2 Procedures of experiment

The exploration constant, also known as the c-value, is a critical parameter in the Monte Carlo Tree Search (MCTS) algorithm that can significantly affect the number of visits per node.

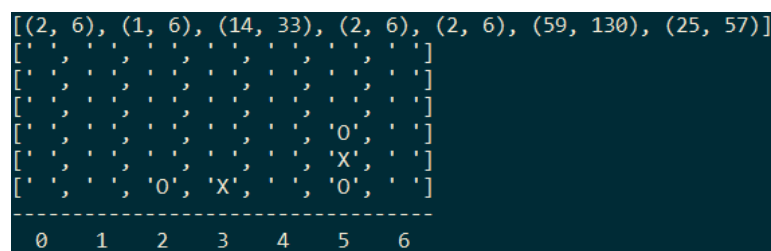


Figure ?. (wins, visits) per node(column) with c-value 0.1.

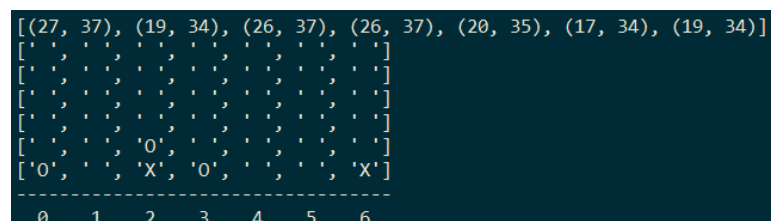


Figure ?. (wins, visits) per node(column) with c-value 10.0.

Figures ? and ? depict the visits of each node during the MCTS process with c-values of 0.1 and 10.0, respectively. The numbers in brackets represent the (wins, visits) per each column from 0 to 6

When the exploration constant is extremely low, as in Figure ?, the algorithm tends to visit only a few selected nodes, while avoiding exploring less visited nodes. This approach results in nodes with fewer visits being under-assessed, reducing the algorithm's ability to select a node with high potential, which could be crucial in the future.

Conversely, when the exploration constant is extremely high, as in Figure ?, the algorithm visits each node with relatively even numbers of visits, regardless of its importance. This approach leads to suboptimal performance because the algorithm keeps visiting unimportant nodes, wasting computational resources.

To find the best optimal value of exploration constant, a reasonable range of values ranging from 0.5 to 2.0 was established. An experimental approach was then employed to assess the performance of each value by having the values play against one another, with the aim of determining the best win rates.

As the optimal value of the exploration constant cannot be determined definitively, a simplified procedure was employed. Initially, a range of values for the exploration constant, denoted as c , was established. Four values within this range were then selected for testing. The performance of each value was assessed by having them play against each other in a series of tests. The two values that produced the highest win rates were then identified, and the range of values was reset between these two values. This process was repeated for 3 times until a reasonable result was obtained, indicating the optimal value for the exploration constant within the established range. It should be noted again that due to the lack of a definitive answer for the optimal value, this simplified approach was deemed appropriate for this study.

Initially, the experiment was conducted between four different values, namely 0.5, 1.0, 1.5, and 2.0. The results of the experiment were analysed to determine which of these values produced the highest win rates. The findings of the experiment are presented below.

Experiment result Details:

Detailed result:

- When AI with c-values of (0.5) were tested against (0.5, 1.0, 1.5) for 100 games each:

c-values:	1.0	1.5	2.0
As a First Player: 0.5	52 WINS / 0 DRAWS	53 WINS / 0 DRAWS	64 WINS / 0 DRAWS
As a Second Player: 0.5	46 WINS / 0 DRAWS	31 WINS / 0 DRAWS	44 WINS / 0 DRAWS

- When AI with c-values of (1.0) were tested against (0.5, 1.5, 2.0) for 100 games each:

c-values:	0.5	1.5	2.0
As a First Player: 1.0	54 WINS / 0 DRAWS	58 WINS / 0 DRAWS	58 WINS / 1 DRAWS
As a Second Player: 1.0	48 WINS / 0 DRAWS	32 WINS / 0 DRAWS	45 WINS / 0 DRAWS

- When AI with c-values of (1.5) were tested against (0.5, 1.0, 2.0) for 100 games each:

c-values:	0.5	1.0	2.0
As a First Player: 1.5	69 WINS / 0 DRAWS	68 WINS / 0 DRAWS	73 WINS / 0 DRAWS
As a Second Player: 1.5	47 WINS / 0 DRAWS	42 WINS / 0 DRAWS	42 WINS / 0 DRAWS

- When AI with c-values of (2.0) were tested against (0.5, 1.0, 1.5) for 100 games each:

c-values:	0.5	1.0	1.5
As a First Player: 2.0	56 WINS / 0 DRAWS	55 WINS / 0 DRAWS	58 WINS / 0 DRAWS
As a Second Player: 2.0	36 WINS / 0 DRAWS	41 WINS / 1 DRAWS	27 WINS / 0 DRAWS

Simplifying the tables above to indicate the win counts:

Total win rates:

Win rates	0.5	1.0	1.5	2.0
As a First Player	56.33 %	56.67 %	70.00 %	56.33 %
As a Second Player	40.33 %	41.67 %	43.67 %	34.67 %

The results of the experiment indicate that exploration constants with c-values of 1.0 and 1.5 were the most successful in terms of winning outcomes. Therefore, the range of exploration constants under consideration has been adjusted to include values of 1.15, 1.35, 1.55, and 1.75.

Starting from this point forward, tables will be presented in terms of win rates. (The experiment was conducted under the same conditions)

Second trial [1.15, 1.35, 1.55, 1.75]:

Total win rates:

Win rates	1.15	1.35	1.55	1.75
As a First Player	57.33 %	68.00 %	64.00 %	60.00 %
As a Second Player	32.67%	43.00 %	41.33 %	33.67 %

Based on the new results, it can be inferred that exploration constants with c-values of 1.35 and 1.55 demonstrated a higher frequency of successful outcomes. Consequently, the range of exploration constants under consideration has been narrowed to include values of 1.30, 1.40, and 1.50.

Third trial [1.30, 1.40, 1.50]:

Total win rates:

Win rates	1.30	1.40	1.50
As a First Player	54.50 %	69.00 %	61.00 %
As a Second Player	34.00 %	43.50 %	40.50 %

The experimental results indicate that exploration constants with a c-value of 1.4 yielded a higher frequency of successful outcomes. Therefore, the range of exploration constants under consideration has been further narrowed to include values of 1.35, 1.40, and 1.45.

Last trial [1.35, 1.40, 1.45]:

Total win rates:

Win rates	1.30	1.40	1.50
As a First Player	54.50 %	69.00 %	60.50 %

As a Second Player	41.50 %	41.50 %	40.50 %
-------------------------------	---------	---------	---------

After conducting the experiment with exploration constants of 1.35, 1.40, and 1.45, the results showed that the c-value of 1.40 had the highest frequency of successful outcomes. The average of each wins between played as first and between second became relatively even. At this moment, it can be concluded that the c-value of 1.40 is the most optimal value for the exploration constant in the current experiment.

Note that these findings may vary in context of different parameters. Further experiments could be conducted to investigate the effect of other hyperparameters, such as the number of rollouts per move, on the performance of the algorithm.

6.2 Effect of other parameters

In this chapter, it will assess the overall performance of a Connect4 game-playing program based under experimental conditions where a fixed exploration constant was employed. To this end, a series of experiments were conducted to investigate the impact of different parameters on the MCTS algorithm, with a focus on the exploration constant which was set to a c-value of 1.4 found in previous experiment.

Prior to conducting the experiments, the number of iterations of the entire MCTS process and the number of simulations, or rollouts, were measured to evaluate their effects on the MCTS algorithm in terms of win rate and time taken. By systematically varying these parameters and examining their interactions with the fixed exploration constant, we were able to gain insights into how different factors contribute to the overall performance of the MCTS algorithm.

6.2.1 Number of simulations / rollouts

To investigate the impact of the number of simulations or rollouts on the performance of AI, a series of experiments were conducted where AI players with varying numbers of simulations or rollouts were tested against each other.

The number of simulations or rollouts were assessed using values of 1 and 5, and combined with different numbers of iterations of the MCTS algorithm, set at values of 10, 50, and 100. The experiments consisted of 200 games for each player, with 100 games played for going first and 100 games played for going second.

Details of results: number of simulations 1 against 5, with number of iterations of 10, 50, 100.

- In case of number of simulations / rollouts: 1
 - ◆ Played first: 48 wins, 46 wins, 45 wins
 - ◆ Played second: 38 wins, 36 wins, 25 wins
 - ◆ Total time taken on first: 4.82 seconds, 28.59 seconds, 66.83 seconds
 - ◆ Total time taken on second: 4.32 seconds, 23.35 seconds, 59.66 seconds.
- In case of number of simulations / rollouts: 5
 - ◆ Played first: 62 wins, 64 wins, 71 wins
 - ◆ Played second: 50 wins, 51 wins, 51 wins

- ◆ Total time taken on first: 17.12 seconds, 85.31 seconds, 205.4 seconds
- ◆ Total time taken on second: 15.69 seconds, 86.87 seconds, 193.326 seconds.

To describe these results in a table in terms of win rates:

- When AI with number of simulations (1) were tested against (5) in terms of number of iterations (10, 50, 100) for 100 games each:

Simulations: 1	10	50	100
As a First Player	48 % (2 DRAWS)	46 % (3 DRAWS)	45 % (4 DRAWS)
As a Second Player	38 %	36 %	25 % (4 DRAWS)
Time Taken (First Player)	4.82 Seconds	28.59 Seconds	66.83 Seconds
Time Taken (Second Player)	4.32 Seconds	23.35 Seconds	59.66 Seconds

- When AI with number of simulations (5) were tested against (1) in terms of number of iterations (10, 50, 100) for 100 games each:

Simulations: 5	10	50	100
As a First Player	62 %	64 %	71 % (4 DRAWS)
As a Second Player	50 % (2 DRAWS)	51 % (2 DRAWS)	51 % (4 DRAWS)
Time Taken (First Player)	17.12 Seconds	85.31 Seconds	205.4 Seconds
Time Taken (Second Player)	15.69 Seconds	86.87 Seconds	193.32 Seconds

The results of the experiments showed that AI players with larger numbers of simulations or rollouts had a higher win rate compared to those with lower numbers, particularly when playing as the first player. Interestingly, the AI players with higher numbers of simulations also won more frequently when playing as the second player compared to those with lower numbers.

However, it is important to note that the AI players with higher numbers of simulations or rollouts also required significantly more time to complete their search process. Specifically, the time required for the higher number of rollouts was approximately 3 to 4 times greater than that required for the AI players with lower numbers of simulations.

These findings suggest that increasing the number of simulations or rollouts can lead to better performance of the MCTS algorithm, particularly in terms of winning outcomes. However, this improvement comes at the cost of increased computational resources and time required to complete the search process.

6.2.2 Number of iterations of MCTS process

To investigate the impact of the number of iterations of whole MCTS process on the performance of AI, a series of experiments were conducted where AI players with varying number of iterations were tested against each other.

The number of iterations assessed using values of 30 and 100, and combined fixed value of numbers of simulations/rollouts, set at values of 5. The experiments consisted of 200 games for each player, with 100 games played for going first and 100 games played for going second. The experiment was repeated for better results.

Details of results: number of iterations of MCTS process 30 against 100

- In case of number of iterations: 30
 - ◆ Played first: 29 wins, 31 wins
 - ◆ Played second: 23 wins, 21 wins
 - ◆ Total time taken on first: 57.50 seconds, 58.13 seconds
 - ◆ Total time taken on second: 51.80 seconds, 52.29 seconds
- In case of number of iterations: 100
 - ◆ Played first: 76 win, 78 wins
 - ◆ Played second: 69 wins, 68 wins
 - ◆ Total time taken on first: 190.86 seconds, 201.21 seconds
 - ◆ Total time taken on second: 174.16 seconds 176.35 seconds

To describe these results in a table in terms of win rates:

- When AI with number of iterations (30) were tested against (100) for 2 times for 100 games each:

Iterations: 30	1 st Test	2 nd Test
As a First Player	29 % (2 DRAWS)	31 % (1 DRAWS)
As a Second Player	23 % (1 DRAWS)	21 % (1 DRAWS)
Time Taken (First Player)	57.50 Seconds	58.13 Seconds

Time Taken (Second Player)	51.80 Seconds	52.29 Seconds
---------------------------------------	---------------	---------------

- When AI with number of iterations (100) were tested against (30) for 2 times for 100 games each:

Iterations: 30	1st Test	2nd Test
As a First Player	76 % (1 DRAWS)	78 % (1 DRAWS)
As a Second Player	69 % (2 DRAWS)	58 % (1 DRAWS)
Time Taken (First Player)	190.86 Seconds	201.21 Seconds
Time Taken (Second Player)	174.16 Seconds	176.35 Seconds

Consistent with the findings regarding the number of simulations / rollouts, the experiments showed that increasing the number of iterations of the MCTS process led to a higher win rate for the AI players, particularly when playing as the first player. Furthermore, the AI players with higher numbers of iterations also won more frequently when playing as the second player compared to those with lower numbers.

However, as with the results on the number of simulations or rollouts, increasing the number of iterations also required significantly more computational resources and time to complete the search process. Specifically, the time required for the higher number of iterations was approximately 3 to 4 times greater than that required for the AI players with lower numbers of iterations

6.3 Performance evaluation of the MCTS algorithm

6.3.1 Analysis parameters: number of iterations and number of simulations / rollouts

The number of simulations or rollouts refers to the total number of times the MCTS algorithm iteratively performs this process of generating and evaluating action sequences from the current game state (selected node).

During each iteration, the MCTS algorithm selects a node in the search tree, expands the node by generating one or more child nodes, selects a child node based on a selection strategy. Then, it simulates a sequence of actions from the selected child node to the end of the game, evaluates the outcome of the simulation, and back-propagates the result of the simulation to update the statistics of the nodes along the path from the root to the selected node.

As noted in earlier results, the number of simulations or rollouts is a crucial parameter in the MCTS algorithm, as it determines the value of the selected node. Increasing the number of simulations generally leads to better performance of the algorithm, but also increases the computational time required to search for an optimal solution. Similarly, the number of iterations is an important parameter that affects the consistency of the win rate in the MCTS algorithm. As with the number of

rollouts, increasing the number of iterations can result in improved performance, but at the cost of increased computational resources, time taken, and memory usage.

Taken together, these results suggest that the optimal performance of the MCTS algorithm is dependent on the appropriate combination of various parameters, including the number of simulations or rollouts, the number of iterations, and the exploration constant. However, achieving optimal performance requires a trade-off between the performance and the computational resources required for the algorithm to operate. Therefore, it is crucial to consider the balance between these factors when designing and optimizing MCTS-based game-playing programs and other AI applications. By carefully considering these trade-offs, we can develop more efficient and effective AI algorithms that can solve complex problems across a range of applications and domains.

6.4 Discussion of the results and their implications

6.4.1 Summary of results

In the context of the MCTS for a board game, efficiency refers to the algorithm's ability to make optimal decisions within a reasonable time frame, minimizing computational resources and providing a responsive user experience. The efficiency of the MCTS algorithm is vital, as it directly impacts the game's playability and overall user experience. This section discusses the efficiency of the MCTS implementation, considering the trade-offs between accuracy, response time, and computational resource usage.

One of the primary factors that influences the efficiency of the MCTS algorithm is the exploration constant used in the UCB algorithm. A higher exploration constant encourages the algorithm to explore less-explored game states, potentially leading to better decision-making. However, excessive exploration can also result in increased computation time, as the algorithm spends more time evaluating less-important nodes. Through experimentation, an optimal exploration constant was determined to provide a balance between exploration and exploitation, ensuring efficient decision-making while maintaining reasonable response times.

Another factor that contributes to the MCTS algorithm's efficiency is the number of simulations or rollouts. As the number of simulations increases, the algorithm explores more possible winners from the game states and potential outcomes, which can improve its decision-making accuracy. However, this increased exploration comes at the cost of increased computation time and resource consumption.

The efficiency of the MCTS algorithm also depends on the number of iterations of the MCTS process. More iterations enable the algorithm to refine its evaluation of game states and moves, leading to better decision-making. However, similar to the trade-off between the number of simulations and response time, increasing the number of iterations also increases computation time and resource consumption.

6.4.2 Optimal parameters

To find an optimal balance between accuracy, response time, and resource consumption, a series of experiments were conducted to fine-tune the MCTS algorithm's parameters. Assuming maximum 3 seconds as a response time, several experiments were conducted to discover the optimal balance. Through these experiments, optimal values for the exploration constant (c - value), the number of iterations, and the number of rollouts were identified, ensuring that the Connect 4 implementation delivers a satisfying user experience without sacrificing performance.

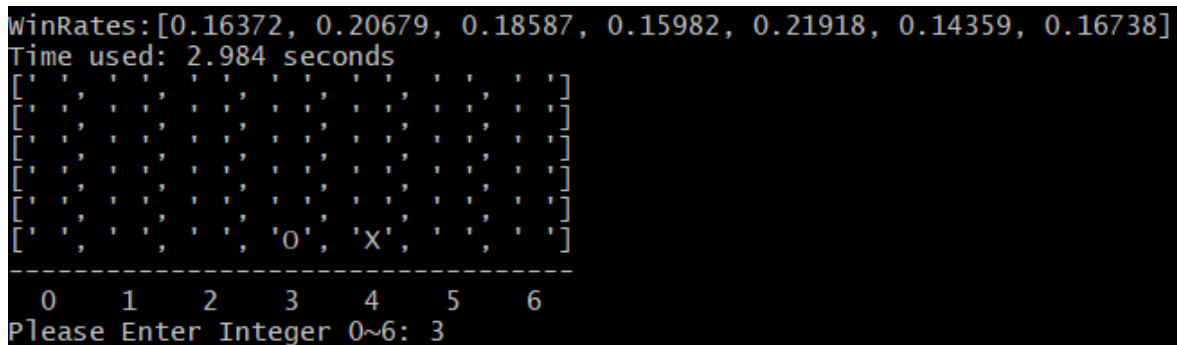


Figure ?. Maximum 3 seconds at early stage of game.

The optimal values found were as follows:

exploration constant (c-value): 1.4

Number of iterations: 1500

Number of rollouts: 4

These values were chosen based on their ability to fit within a 3-second time frame, which was deemed an acceptable response time for a turn-based game like Connect 4. This constraint ensures that the game remains engaging and interactive, as players do not have to wait excessively long periods for the AI to make its move.

Note that optimal parameter values identified in this project may not be universally applicable to other MCTS implementations or game scenarios. Factors such as the complexity of the game, the available computational resources, and the desired user experience may require different parameter values for optimal performance. Therefore, it is essential to conduct thorough experimentation and analysis when implementing the MCTS algorithm in other contexts to ensure that the chosen parameter values represent the best possible performance.

Chapter 7: Conclusion and Further Work

7.1 Summary of the achievements

Over the course of this final year project, several significant achievements were accomplished, by contributing to the successful implementation of the MCTS algorithm in the Connect 4 game. These achievements effected on various factors in terms of theoretical understanding, algorithm development, experimentation, and development skills. The followings summarise the major part of accomplishments and learnings throughout the project.

Theoretical Understanding: This project provided an opportunity to research deeply into the study of bandit problems and their associated algorithms, particularly the Upper Confidence Bound (UCB) algorithm. This exploration facilitated a comprehensive understanding of the principles behind bandit problems and the role of bandit algorithms in the MCTS algorithm. Gaining enough knowledge of these concepts was helpful in designing an effective MCTS implementation for the Connect 4 game.

MCTS Algorithm Development: The project offered valuable experience in designing, implementing, and optimising the MCTS algorithm to achieve optimal performance in the context of the Connect 4 game. This process involved understanding the complicated part of the MCTS algorithm, including its exploration-exploitation trade-offs, and developing efficient code to bring the algorithm to be effective. This meaningful experience has reinforced the understanding of the MCTS algorithm and its practical applications in game-playing AI for Connect 4.

Experimentation and Optimization: The project involved extensive experimentation to determine the optimal parameter values for the MCTS implementation, ensuring a balance between decision-making accuracy (win rates), response time (time taken for AI process), and resource consumption. This process involved conducting trials with various parameter combinations and analysing their performance. The ability to design and carry out experiments effectively and derive meaningful insights from the results was a critical skill and has been considerably enhanced through this project.

Software development and Python Coding Skills: The project served as an excellent platform to apply and improve test-driven development skills in the context of Python programming. By adhering to TDD principles throughout the project, the codebase was developed in a structured and robust manner, ensuring the reliability and maintainability of the MCTS implementation. This experience has sharpened the ability to write high-quality and well-tested code, and demonstrated the value of TDD in software development projects.

Connect 4 Game Implementation: The project built up to the successful creation of a Connect 4 game powered by the MCTS algorithm, showing the effectiveness of the algorithm in making optimal results in a complex turn-based game. The game's development involved designing and implementing the game board, graphic or texture user interface, and AI components, further improving Python programming skills and providing valuable experience in game development.

In conclusion, the achievements accomplished during this project have not only resulted in a successful MCTS-powered Connect 4 game but have also significantly advanced the understanding of bandit problems, the MCTS algorithm, and their practical applications in AI game-playing. Furthermore, this project helped to develop critical skills in experimentation, test-driven development, and Python programming, which may lead to advanced work on my possible future projects and research in terms of software development.

7.2 Professional issues

During the development of a Monte Carlo Tree Search (MCTS) algorithm for Connect 4 presented various professional issues that required careful consideration and decision-making. These issues, which are associated to the field of computer science, are not only critical for maintaining professional standards, but also for ensuring the societal impact of computer technology is positive and meaningful. In this section, it will discuss four professional issues encountered during the project: open-source licensing, usability, management and plagiarism.

7.2.1 Licensing

One of the most significant professional issues that I have encountered during the development of the MCTS for Connect 4 project was the usage of open-source libraries and resources. Open-source programs offered numerous advantages, such as saving cost for time, increasing flexibility, and the potential for collaboration with a community of developers.

For this project, I utilised various open-source implementation of the MCTS algorithm for connect 4 as a reference for my own work. The open-source project I chose to was licensed under the MIT License, a permissive license that allows for the use, modification, and distribution of the code, as long as the original copyright notice and permission notice are included.

7.2.2 Usability

In terms of usability, one another professional issue I encountered during the development of the MCTS for Connect 4 project was ensuring the possible implications of using AI in board games. While developing an AI, it is essential to consider the broader social impact of implementing AI in board games and how it might affect human interaction and engagement.

The implementation of MCTS in Connect 4 introduces AI as a potential replacement for human opponents in the game. Definitely, AI can offer several benefits, such as solving a complicated and complex problems, providing a challenging opponent for experienced players or offering an available opponent when no human player is present. However, it also raises concerns about the broader implications of replacing human opponent in board games.

For an instance, it may impact on learning from a human player. When playing against a human opponent, players can learn new strategies and techniques by observing their opponent's moves, emotions, and adapting their style of gameplay. With AI opponents, the learning experience may be limited or even diminished, as AI algorithms can sometimes make moves that are difficult for human players to comprehend or replicate.

Moreover, AI opponents in board games may lead to an overreliance on AI in various aspects of life. This could result in reduced problem-solving skills and critical thinking abilities among players, who may become accustomed to relying on AI assistance rather than developing their skills and strategies.

7.2.3 Time Management

One of the most critical aspects of project management is time management and resource allocation. In the context of the time resources, it entailed the following:

- Setting milestones and deadlines: To maintain a structured approach to the project, I established clear milestones and deadlines for each phase, including research, design, implementation, testing, and documentation. These milestones allowed me to monitor progress and make necessary adjustments to stay on track.
- Prioritizing tasks: Throughout the project, I prioritised tasks based on their importance and dependencies. This strategy enabled me to focus on critical tasks while ensuring that all necessary prerequisites were completed in a timely manner.

- **Balancing time and quality:** To ensure a high-quality Connect 4 implementation without compromising the project timeline, I carefully balanced the time spent on each task with the desired quality of the final product. This approach allowed me to allocate resources efficiently and make informed decisions about trade-offs between time and quality.
- **Adapting to unforeseen challenges:** During the project, I encountered unexpected challenges, such as difficulties in implementing certain aspects of the MCTS algorithm or issues with integrating third-party libraries. To address these challenges, I adjusted my timeline, reassessed priorities and followed guidance from my supervisor.

7.2.4 Plagiarism

Another professional issue I confronted during this project was the potential for plagiarism. As I researched and studied existing MCTS implementations, related research papers, and other resources to gain a deeper understanding of the algorithm and its application to Connect 4, it was necessary to ensure proper citation and acknowledgment of these sources. To avoid plagiarism and demonstrate academic integrity, I took the following steps:

- **Properly citing all sources consulted:** In my project report, I included a comprehensive list of references, adhering to the required citation format. Whenever I discussed ideas or methods derived from these sources, I provided in-text citations to give credit to the original authors.
- **Using plagiarism detection tools:** To further ensure the originality of my work, I used plagiarism detection software to check my project report and other written materials. These tools helped me identify any potential instances of unintentional plagiarism and correct them before submitting my final project.

By addressing the issue of plagiarism and adhering to proper citation, I tried to keep the ethical standards of computer science professionals and demonstrated a commitment to academic integrity.

In conclusion, throughout the course of the final year project, I encountered various professional issues that required thoughtful consideration and management.

This experience has not only enhanced my technical skills but also deepened my understanding of the importance of professionalism in the field of computer science. By describing these professional issues in my project, I believe that I have become a more ethical and socially responsible in computing society.

7.3 Possible further research and development

The results of this study present several opportunities for future research and development in the field of game AI programming, particularly in the context of the Monte Carlo Tree Search (MCTS) algorithm. The following sections summarise improvable areas for further research, aiming to enhance the performance and efficiency of MCTS-based game-playing programs across various applications and fields.

7.3.1 Exploration of Optimal Parameters for Different Games

As demonstrated in this study, the exploration constant and other parameters of the MCTS algorithm can significantly impact its performance. However, the optimal settings for these parameters may vary depending on the game or problem being addressed. Further research could focus on identifying the best parameter configurations for MCTS in a wide range of games, taking into account their unique characteristics, such as game tree depth, branching factor, and game complexity. By examining the performance of MCTS under diverse conditions, researchers could develop more

generalizable strategies for optimising the algorithm to achieve best performance across various applications.

7.3.2 Enhanced Experimentation for Parameter Optimization

Due to the time and resource limitations, the number of tests conducted in this study was relatively minimised and simplified, potentially restricting the generalizability of our findings. Future research could involve more extensive and rigorous experimentation, including a larger number of tests, varying exploration constants, and more diverse combinations of iterations and simulations. By employing more comprehensive and robust experimental designs, researchers may obtain a better understanding of the relationships between the algorithm's parameters and its performance, resulting in the development of more efficient and effective MCTS-based game-playing AI programs.

7.3.3 Improved Simulation Algorithms

In this study, the simulations performed during the MCTS algorithm were based on random move selection. While this approach can provide valuable insights into the algorithm's performance, it may not fully cover the potential advantages of using more highly developed simulation algorithms. For example, incorporating heuristics or other manual algorithms capable of detecting winning moves and blocking opponent's winning moves by checking if there is any pattern of 3-in-row could lead to more accurate evaluations of nodes during the simulation process. By integrating more advanced simulation techniques, it could further enhance the performance of the MCTS algorithm, particularly in games where strategic decision-making with limited time or resource is crucial for higher win rates.

7.3.4 Scalability and Parallelization

As the complexity of games and problems increase, it becomes increasingly important to investigate the scalability of the MCTS algorithm. Future research could focus on optimizing the algorithm's performance in more complex environments, particularly those with larger game trees and higher branching factors and possible movement options. Moreover, techniques for accelerated execution of code such as exploring parallelization techniques for MCTS or using fast and lighter datatype could help enhance its efficiency and enable the algorithm to engage more demanding applications and computational challenges.

In conclusion, the potential for further research and development in the field of MCTS-based game AI programming is unlimited. By exploring these opportunities and building upon the findings of this project, it will be possible to continue advancing the study in game-playing algorithms, and finally, contributing to the development of more efficient and effective AI solutions for a wide range of complex problems.

Appendix

User Manual

System Requirements

- Python 3.10 or higher
- git installed
- pygame installed
- numpy installed

How to run

1. Download the code as a zip file and unzip. Or you can clone by

```
$ git clone https://gitlab.cim.rhul.ac.uk/zfac131/PROJECT.git
```

2. Move into the project folder

3. Run main.py

```
$ python main.py
```

4. Follow instruction from the command-line:

```
$ python main.py
pygame 2.1.2 (SDL 2.0.18, Python 3.10.7)
Hello from the pygame community. https://www.pygame.org/contribute.html

(1): 2 Human Players
(2): Play against AI
(3): Test_MCTS

Please Enter 1~3: |
python.exe*[64]:17784                                     « 221218[64]
```

Link to demo video

<https://youtu.be/w71iqCKam1I>

Bibliography

- ¹ Understanding the Role of AI in Gaming. (2020).AITHORITY [online]
Available at: <https://aithority.com/computer-games/understanding-the-role-of-ai-in-gaming/>
[Accessed 4 Oct. 2022].
- ² Brüggmann, Bernd (1993). *Monte Carlo Go*. Technical report, Department of Physics, Syracuse University. [PDF] Available at: <http://www.ideaest.com/vegos/MonteCarloGo.pdf>
[Accessed 5 Oct. 2022].
- ³ Metropolis, Nicholas, and Stanislaw Ulam. "The monte carlo method." *Journal of the American statistical association* 44.247 (1949): 335-341.
- ⁴ Chaslot, Guillaume M. JB, et al. "Progressive strategies for Monte-Carlo tree search." *New Mathematics and Natural Computation* 4.03 (2008): 343-357.
- ⁵ Ghoneim, Amr S. *On Competency of Go Players: A Computational Approach*. Diss. University of New South Wales, Canberra, Australia, 2012.
- ⁶ Lee, Byung-Doo. "Implementation of robust Tic-Tac-Toe game player, using enhanced Monte-Carlo algorithm." *Journal of Korean Society for Computer Game* 28.3 (2015): 135-141.
- ⁷ "Connect Four". United States Patent and Trademark Office.
US Serial Number: 73019915. US Registration Number: 1009552
- ⁸ Amadio, Brian (2020). Multi-Armed Bandits and the Stitch Fix Experimentation Platform. [blog]
Available at: <https://multithreaded.stitchfix.com/blog/2020/08/05/bandits/>
[Accessed 11 Nov. 2022].
- ⁹ Zaritsky, Assaf, and Moshe Sipper. "The preservation of favored building blocks in the struggle for fitness: The puzzle algorithm." *IEEE Transactions on Evolutionary Computation* 8.5 (2004): 443-455.
- ¹⁰ Powley, Edward J., Daniel Whitehouse, and Peter I. Cowling. "Bandits all the way down: UCB1 as a simulation policy in Monte Carlo Tree Search." *2013 IEEE Conference on Computational Intelligence in Games (CIG)*. IEEE, 2013.
- ¹¹ Browne, Cameron B., et al. "A survey of monte carlo tree search methods." *IEEE Transactions on Computational Intelligence and AI in games* 4.1 (2012): 1-43.
- ¹² Beck, Kent. *Test-driven development: by example*. Addison-Wesley Professional, 2003.