



Online Planning in MDPs Using Monte Carlo Simulations

Jonathon Mitchell

School of Computing and Communications

B.S.c Computer Science

Lancaster University

A project report submitted for the degree of

Computer Science

24th March 2023

Declaration

I certify that the material contained in this dissertation is my own work and does not contain unreferenced or unacknowledged material. I also warrant that the above statement applies to the implementation of the project and all associated documentation. Regarding the electronically submitted work, I consent to this being stored electronically and copied for assessment purposes, including the School's use of plagiarism detection systems in order to check the integrity of assessed work. I agree to my dissertation being placed in the public domain, with my name explicitly included as the author of the work.

Name: Jonathon Mitchell

Date: 24th March 2023

Abstract

The following report provides an insight into the effect of reducing the environment complexity on the MCTS algorithm. Monte Carlo Tree Search is an artificial intelligence algorithm designed to solve large Markov Decision Processes (MDPs) efficiently. Monte Carlo Tree Search (MCTS) is a decision-making algorithm that uses Monte Carlo simulations to search for optimal moves in a game or problem space. It works by building a tree of possible moves and outcomes, and then selectively exploring the most promising branches. By simulating many possible outcomes, MCTS gradually learns which moves are most likely to lead to success and selects them accordingly. This paper describes how the algorithm was designed and configured to its environment. As part of its implementation, it focuses on the algorithm's configuration: rewards shaping, and the number of simulations and how they affect the algorithm's performance. As such, the environment used was that of a Tic Tac Toe game board. This algorithm was developed in Java alongside an implementation of Tic Tac Toe, developed for this project. The experiments demonstrate the impact of different algorithm configurations on the algorithm's accuracy and efficiency. The results provide insights for future research on MCTS's scalability in simpler MDP domains.

Contents

1	Introduction	1
1.1	Project Aims	1
1.2	Project Overview	2
2	Background	4
2.1	History of Monte Carlo Tree Search and its evolutions	5
2.2	Literature Review	6
2.3	Research Overview and Implications	9
2.4	Theoretical Framework	10
2.5	Limitations	11
3	Design	13
3.1	Designing the Algorithm	13
4	Development	18
4.1	Implementation Framework	18
4.2	Data Structures and Classes	19
4.3	Algorithm Integration	20
4.4	Implementation Issues	28
5	Experiments and Results	29
6	Conclusions	33
6.1	Review of Aims	33

6.2	Possible Design/ Implementation Revisions	35
6.3	Closing Remarks	36
7	References	37
	Appendix A Project Proposal	39
	Appendix B Additional Tables	44

List of Figures

3.1	An illustration of the 4 stages of the Monte Carlo Tree Search Algorithm [4]	14
3.2	Class Diagram showing the Algorithm incorporated with the problem environment	17
4.1	An image of the MCTS agent playing against a human player	21
4.2	An image of the MCTS agent playing against a human player, a couple moves later	22
5.1	The table of results from the test case where the agent made the first move, and rewards shaping was disabled	30
5.2	The table of results from the test case where the agent made the first move, and rewards shaping was enabled	30
5.3	The table of results from the test case where the agent's opponent made the first move, and rewards shaping was disabled	30
5.4	The table of results from the test case where the agent's opponent made the first move, and rewards shaping was enabled	31
5.5	A scatter graph showing the relationship between average simulation time and the number of simulations	31
5.6	A scatter graph showing the relationship between the number of wins and the number of simulations	32

B.1 This table is an example table used when attaining the statistics from each match. The outcome of the match is represented by 'y' or 'n' and the average simulation time per game is calculated at the bottom. 44

Chapter 1

Introduction

Since its creation in the late 1940s, Monte Carlo methods have had much success in AI algorithms, especially those used to play digital games. Monte Carlo Tree Search (MCTS) in particular could be one of the best current AI algorithms in this particular field. Achieving levels of performance degrees higher than humanly possible in games such as Go and Ms Pac-Man whilst also beating all other game-playing AI in multiple competitions [1][2], this algorithm has proven its capability in large MDPs thus far. However, The aim of this project is to evaluate the performance and computational efficiency of the MCTS algorithm in lower complexity MDPs. A Markov Decision Process (MDP) is a sequential decision problem for a fully observable, stochastic environment with a Markovian transition model and additive rewards [3]. This meaning, the function of the algorithm is to build a tree of states in which transitions between states are caused by taking actions. The project will also explore the effect of different algorithm configurations, including the number of simulations, the depth of the search tree, and rewards shaping, on the performance of the algorithm.

1.1 Project Aims

- To develop a MCTS algorithm that can make decisions in its environment

- To discover whether the algorithm can achieve the desired reward state consistently
- To discover whether the algorithm can maintain computational and time efficiency
- To determine the effects of rewards shaping on the algorithm's competence and efficiency
- To determine the effects of adjusting the limits on the algorithms framework: the number of simulations, and the depth of the search tree.

1.2 Project Overview

- Chapter 2: The Background section of this report will provide a thorough overview of the related work in this area, including the existing literature on MCTS and its variations. It will also discuss the technical challenges and approaches used in this project and how they compare to other systems.
- Chapter 3: The Design and Research section of the report will detail the process researching and developing the design iterations of this project. It will also discuss major design decisions and provide thorough justifications.
- Chapter 4: The Development and Implementation section will provide insight into how the algorithm was created, including any problems faced and how they were addressed.
- Chapter 5: In the Experiments and Results section, the report will present the results of experiments on the algorithm's performance as complexity is adjusted, including the quality of its decisions and its efficiency in making decisions.
- Chapter 6: Finally, in the Conclusions section, the report will summarise the results and discuss the conclusions that can be drawn from them in the context

of the aims set at the start of this report, along with any lessons learned or after thoughts from the project.

Chapter 2

Background

Monte Carlo Tree Search (MCTS) is a simulation-based algorithm used for decision-making problems, especially in situations where the state space is large and complex. MCTS works by building a search tree of possible actions and their outcomes and then using simulations to explore the tree to find the best action to take. The algorithm is particularly useful in situations where there is uncertainty or incomplete information (MDPs), as it can model the possible outcomes of different actions and update its estimates of the value of each action based on the results of the simulations. MCTS has a wide range of applications in different domains such as gaming, robotics, and artificial intelligence. In gaming, it is commonly used for game-playing AI agents to make strategic decisions in complex games like Go, chess, and poker. In robotics, MCTS can be used to plan paths or actions for robots operating in complex and uncertain environments. In artificial intelligence, MCTS can be used for decision-making in areas such as natural language processing and recommendation systems. MCTS has proven to be very competent when solving complex decision-making problems in a variety of domains, and its ability to handle uncertainty and incomplete information makes it especially useful in many real-world applications where conditions are often unpredictable.

2.1 History of Monte Carlo Tree Search and its evolutions

Monte Carlo Tree Search (MCTS) was first introduced in the early 2000s as a simulation-based algorithm for solving complex game-playing problems. The earliest known application of MCTS was in 2001, when it was used to play the game of Othello by Coulom. Coulom then extended his work with the MCTS algorithm in 2006 with his paper "Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search" [4]. Coulom proposes an approach to Monte-Carlo Tree Search (MCTS) that improves its efficiency and effectiveness in game playing. The paper presents a number of enhancements to the basic MCTS algorithm, including a new selection policy and backup operators, that improve its performance in game-playing scenarios. These enhancements were demonstrated through experiments in which Coulom applied his improved MCTS algorithm to the game of Othello, achieving impressive results against strong human opponents. However, the algorithm was not widely known until the introduction of the UCT (Upper Confidence Bounds for Trees) variant by Kocsis and Szepesvári in their 2006 paper "Bandit Based Monte-Carlo Planning" [5]. This paper presents the Upper Confidence Bounds for Trees (UCT) variant of the Monte-Carlo Tree Search (MCTS) algorithm. The UCT algorithm uses a bandit-based approach to balance exploration and exploitation during the tree traversal. This meaning each state of the search tree is viewed as a multi-armed bandit; an ML framework in which an agent has to select actions (arms) in order to maximise its cumulative reward in the long term. This variant has been shown to outperform several other state-of-the-art planning algorithms, including the standard MCTS algorithm, in a range of benchmark problems, including game playing scenarios. As I am implementing MCTS with the UCT variant, Kocsis and Szepesvári's work provides an important reference for understanding the strengths and weaknesses of the UCT algorithm. Over time, various extensions and improvements have been made to the basic MCTS algorithm. For example, the

UCT variant has been shown to be effective in many games, but it may not work as well in all situations. Researchers have developed modifications such as MCTS with progressive widening, which uses a heuristic to limit the number of actions considered at each node of the search tree. Other variants include MCTS with domain-specific knowledge, which incorporates expert knowledge to improve the search, and MCTS with continuous actions, which can handle problems with continuous action spaces. Overall, MCTS has evolved significantly since its introduction. Its effectiveness and flexibility have made it competent when solving complex decision-making problems in a variety of fields.

2.2 Literature Review

In 2010 David Silver and Joel Veness proposed a Monte Carlo planning algorithm for large partially observable Markov decision processes (POMDPs)[6]. The authors addressed the scalability issue of POMDP planning by using Monte Carlo Tree Search (MCTS) with two key modifications: (1) a particle filter for belief state estimation. This works by representing the belief state as a set of weighted particles, where each particle corresponds to a possible state of the system, and the weight of the particle represents the probability of that state being the true state. At each time step, the particles are updated based on the agent's observations and actions, and resampled according to their weights. The particles with higher weights are more likely to be selected for the next iteration, while the particles with lower weights are less likely to be selected. (2) UCT with a heuristic backup policy. The heuristic backup policy is a rule-based policy that is used to select actions when the search has not explored a sufficient number of times a certain state-action pair. In particular, the backup policy selects actions that have the highest reward or the lowest cost, depending on the task. This allows the search to focus on promising branches of the search tree while still exploring less promising branches. Their approach achieved state-of-the-art performance on various benchmark problems, including a grid-world

navigation task and a robotic object search task. This approach achieved high-level performance in a wide range of problem environments. Thus, the algorithm proposed by Silver and Veness provides a useful reference for evaluating the performance of other MCTS-based MDP solvers, including those used in my own implementation of MCTS.

In the research paper "A Survey of Monte Carlo Tree Search Methods" by Browne et al. (2012), Browne provides a comprehensive survey of MCTS methods, including an overview of the various MCTS algorithms, including UCT and its variants, as well as their applications in various problem domains such as games. The survey covers both theoretical and practical aspects of MCTS, including the properties of optimal trees, the use of heuristics and domain knowledge, and the selection of appropriate parameters. The challenges and limitations of MCTS are also discussed, such as the trade-off between exploration and exploitation, the need for efficient state representation and evaluation functions and the difficulty of handling large state spaces. In terms of relevance to the specific research area of this report, the survey provides a useful reference for evaluating the performance of MCTS algorithms and for designing custom MCTS methods. The authors' analysis of the strengths and weaknesses of various MCTS algorithms can be used to guide the implementation of MCTS on tic tac toe, as well as to inform future research on MCTS in other decision-making domains.

Pepels et al. (2014) propose a real-time Monte-Carlo Tree Search (MCTS) algorithm for playing the game Ms. Pac-Man. The authors use MCTS to select actions based on simulations of the game and apply several optimisations to improve the search efficiency and the quality of the game-playing strategy. This paper demonstrates the use of MCTS in a real-time setting and showcases the potential of MCTS for improving game-playing performance. This paper also provides valuable insights into techniques for optimising MCTS algorithms, which can be applied to Tic Tac Toe as well as a variety of other problem domains. Thus, this paper provides a useful reference for evaluating the performance of different MCTS algorithms and

for designing custom MCTS methods in games such as Ms. Pac-Man and Tic Tac Toe.

In "The Impact of MCTS Playout Budget in Monte-Carlo Go" [9], Enzenberger evaluates the impact of the number of simulations on the performance of MCTS in the game of Go. Enzenberger found that by increasing the number of simulations, the performance of MCTS improved, but the rate of improvement diminished simultaneously. Despite this paper focusing on the game of Go, its insights and conclusions are relevant as they highlight the importance of balancing exploration and exploitation in MCTS, which is essential in games with large branching factors such as Go or Tic Tac Toe. Furthermore, the paper discusses the importance of efficient simulation algorithms and the use of domain-specific knowledge in MCTS. Thus, this paper is a good reference for evaluating the impact of different playout budgets on the performance of MCTS algorithms in Tic Tac Toe. This effect, will likely inform the design of an optimal playout budget for Tic Tac Toe and improve the both efficiency and accuracy of the algorithm in selecting moves.

"Effective Pruning of Monte Carlo Tree Search Using Progressive Bias and Rollout Horizon Control" by Nagai [10] proposes a pruning technique for MCTS that uses progressive bias and rollout horizon control to improve the efficiency and effectiveness of the search. The authors do this by introducing a new selection criteria that places greater weight on selection towards more promising nodes and limits the the depth of the rollouts based on the node's estimated value. This paper is relevant in terms of the subject matter of this report in that it addresses the issue of improving the efficiency of the MCTS search method by 'pruning' less promising branches. This summary of this paper concludes that it can provide insights into the design and implementation of this particular MCTS algorithm, as it can benefit from the proposed pruning technique. The selection criterion and rollout horizon control can be adapted to Tic Tac Toe and other problem domains to improve the efficiency and effectiveness of the MCTS search.

Overall, the literature reviewed provides a wealth of insights into the strengths

and limitations of MCTS, its various variants and applications, and potential areas for future research.

2.3 Research Overview and Implications

Monte Carlo Tree Search (MCTS) has been extensively researched in the fields of gaming, robotics, and artificial intelligence. Some of the strengths of MCTS include its ability to handle large search spaces, its flexibility in handling different types of problems, and its ability to learn and improve over time. Some of the weaknesses of MCTS include its computational complexity, its sensitivity to parameter settings, and its potential to get stuck in local minima. Previous research on MCTS has focused on developing and improving the algorithm and its variants, applying it to various problems in different fields, and comparing it with other search algorithms. Some studies have proposed modifications to the basic MCTS algorithm, such as MCTS with progressive widening or domain-specific knowledge, to improve its performance on specific problems. Other studies have explored the use of MCTS in robotics, natural language processing, and recommendation systems. In terms of gaming, MCTS has been used to develop AI agents for games like Go, chess, and poker, and has achieved remarkable performance on these games. For example, AlphaGo, an AI agent developed using MCTS, defeated the world champion in the game of Go. In robotics, MCTS has been used to plan paths or actions for robots in uncertain environments, such as autonomous driving or industrial automation.

My research builds upon the existing work in MCTS by applying it to a low complexity problem domain. Specifically, my research aims to investigate the computational and time efficiency of MCTS when applied to a low complexity environment; a problem domain that has not been extensively explored in the literature. Furthermore, my research aims to investigate how configuring search parameters, such as: the number of simulations and horizon control, and the implementation of rewards shaping, affect the efficiency and accuracy in this low

complexity environment. In summary, previous research on MCTS has identified its strengths and weaknesses, developed and improved the algorithm and its variants, and applied it to various problems in different fields. My research builds upon this work by applying previously researched algorithm modifications to the basic MCTS algorithm in new environments.

2.4 Theoretical Framework

Monte Carlo Tree Search (MCTS) is a decision-making algorithm that is based on the principles of Monte Carlo simulation and tree search. MCTS can be used to solve a variety of problems, including games, robotics, and optimization, and is particularly effective in problems with large search spaces and uncertainty. At a high level, MCTS works by constructing a search tree that represents the possible actions and outcomes of a decision problem. The tree is built incrementally, by simulating the outcomes of actions and updating the statistics of the nodes in the tree based on the simulation results.

The core of the MCTS algorithm involves four steps: selection, expansion, simulation, and backpropagation. During the selection step, the algorithm traverses the tree from the root node to a leaf node, selecting nodes based on a predefined selection policy. The algorithm adopts the UCT policy [5], proposed by Kocsis and Szepesvári in 2006, in which the agent balances the exploration of new nodes, with the exploitation of previously visited promising nodes. Once a leaf node is reached, the algorithm expands the node by adding child nodes that represent the possible actions that can be taken from that state. The algorithm then simulates the outcome of the game or problem by selecting actions randomly from the expanded nodes until a terminal state is reached. Finally, the results of the simulation are backpropagated up the tree, updating the statistics of the nodes visited during the selection step.

2.5 Limitations

Although Monte Carlo Tree Search (MCTS) has shown great success in a wide range of applications, there are still some limitations and areas for future research. One of the limitations of MCTS is its high computational cost, which can make it challenging to apply in real-time applications or on resource-constrained devices. This limitation has influenced the direction of this research paper by deriving a need for a more efficient tree search. This project aims to do that through the use of rewards shaping and the implementation of configuration restrictions (number of simulations, maximum tree depth). These algorithm modifications could provide a more efficient search by preventing the algorithm from generating too many unpromising nodes through the use of reward shaping, and by finding the optimal algorithm configurations; using a number of simulations and a maximum tree depth, that balances efficient computational usage with accurate decision making. In terms of ongoing research areas, this limitation has inspired more efficient variants of MCTS, such as parallel MCTS [11] or MCTS with deep neural networks [12], which can reduce computation time while maintaining the accuracy of the search.

Another limitation of MCTS is its tendency to focus on short-term rewards, which can lead to suboptimal solutions in many problem domains. This limitation has also influenced this research project as rewards shaping is a tool that can be used to prevent MCTS from focusing on short-term rewards. Rewards shaping does this by modifying the reward signal that the agent receives during training to incentivise desired behaviour. In MCTS, the agent’s reward signal is based on the outcome of the game/ task, which may not be available until the end of the game/ task. By shaping the reward signal during the intermediate steps of the game, the agent can learn to make decisions that will lead to the desired outcome in the long term, as opposed to just optimising for short-term rewards. For example, in tic tac toe, the agent can be rewarded for placing 2 “X’s” in a row, or by blocking an opponent’s attempt to form a row. This limitation has also led to developments in the ongoing research areas, producing MCTS variants that incorporate long-term planning, such

as Monte Carlo Tree Search with Value Iteration (MCTS-VI) or Monte Carlo Tree Search with Rollouts and Value Expansion (MCTS-RAVE) [13].

Chapter 3

Design

In this design chapter, we will explore the various components of the Monte Carlo Tree Search (MCTS) algorithm and how they can be implemented and customised for specific applications. The process of designing the MCTS algorithm can be broken down into the different components that it is made up of: the tree search algorithm, the state, action and observation spaces, and the node structure. Each of these elements provide different features which, collectively, allow the algorithm to learn how to act critically. Overall, this chapter aims to provide a comprehensive understanding of the design principles and techniques behind this implementation of the MCTS algorithm.

3.1 Designing the Algorithm

I began by developing the Monte Carlo Tree Search class. This involved initialising the root node with the initial state of the game board passed in as a parameter. The initial state is passed into the root node's constructor because this node represents the starting point of the game. From this, the node can expand and produce child nodes of every legal action at the start of the game. This initialises the search tree and sets the algorithm up to begin simulating. The search method's function is to run the 4 stages of the Monte Carlo Simulations required to expand the search tree.

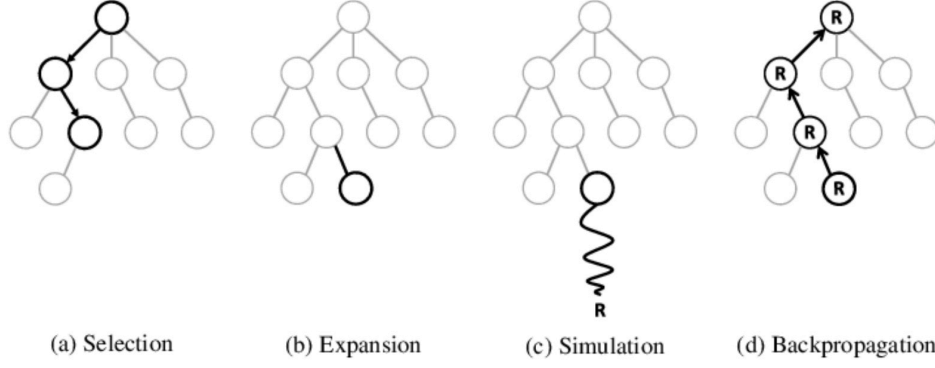


Figure 3.1: An illustration of the 4 stages of the Monte Carlo Tree Search Algorithm [4]

The 4 stages are: selection, expansion, simulation and backpropagation, as shown in figure 3.1. The algorithm functions by looping through these 4 stages to build up a tree of states, each of which having a visit count and reward value. These values are used by the algorithm to determine which actions are the best in terms of reaching the desired reward state once the algorithm has finished simulating. The visit count represents how many times that node has been visited, and the reward count, represents how many times that node has led to a winning state. These values can then be used to compute the score of that particular node. This is done by dividing the reward value of that node by the visit count. The selection phase requires defining a selection policy such as the Upper Confidence Bound on Trees (UCT) policy. This policy uses the UCB1 algorithm to decide which actions to take [5].

$$Q(s, a) = c\sqrt{\log N(s) \div N(s, a)} \quad (3.1)$$

$Q(s, a)$ is the score of the action a applied on the state s , c is the scalar constant which determines the ratio of exploration to exploitation, $N(s)$ is the parent node, and $N(s, a)$ is the child node created when a is applied on s . If c is 0, the algorithm will act greedily, favouring exploitation over exploration. This means that the algorithm will be much more likely to choose nodes with a higher visit count as opposed to less visited or unexplored nodes. The algorithm may want to do this because the likelihood is that nodes with a higher visitation lead to winning terminal states. After selecting a node to expand, the next phase in the algorithm is expansion. The expansion phase involves generating new child nodes from the selected node. These child nodes represent all the possible states that agent can transition into from its current state. In order to implement this phase, the algorithm obtains all the legal moves from its current state. New states are then generated with these moves applied, thus creating list of child nodes, of which the algorithm can simulate from. Following the expansion phase is the simulation phase. First the algorithm chooses one of these nodes to rollout. During the rollout period, the algorithm loops, choosing random actions and forming new states (without creating new nodes) until a terminal state has been reached. Once the terminal state is reached, the reward is backpropagated up the tree. During the backpropagation phase, the statistics (reward value and visit count) are updated for each node that led to that terminal state until the root node is reached. However, as this algorithm is being applied on a 2 player game, the sign of the reward value is switched at each node. This is because, at each node, it is a different player's turn meaning that a win for one player is a loss for the other, and this is represented by the reward value (negative for loss, positive for win).

The function of the state space is to maintain all the of the state variables from the environment the agent is placed in, for example: the positions of the points in the game board in Tic Tac Toe, the player whose turn it is to make a move, the amount of moves that have been played up to the current state, etc. This is so that the agent has access to these variables without having to access them directly from

the environment's class. This allows the agent to run simulations, in which moves can be made, and these state variables can change. From the simulations, the agent can then know what moves will lead to winning states without having to actually make moves. When the agent has this ability to plan, it can then choose what it computes are the best moves and then make them within environment. I started implementing the State space by defining variables to represent the same variables within the game class such as the game board, the current player, the move that led to this state and the number of moves up to this state. I then added accessors and mutators so that these variables could be both accessed and changed from outside the class. These methods would then be used in both the MCTS class and the Node class in methods such as simulate, and expand. I also added functionality similar to that of the game class, such as methods to check for wins, etc. This was so that the during simulations, the algorithm would be able to detect winning states, and could then back propagate the results up the tree that of nodes that led to the terminal state.

The responsibility of the Node class is to act as a node within the search tree. This involves maintaining variables such as: the state that that node represents, a list of child nodes, the parent node, the reward count and the visit count (both of which are updated whenever a simulation leads to a terminal state). The Node class also requires methods that play a part in the MCTS algorithm, including: expand, selectChild and updateStats. The expand method generates a list of all the legal moves in the current state. These moves are then applied to the current state generating new states which are passed into the constructors of the newly instantiated child node objects. Finally, these objects are added to the list of child nodes of the expanded node. The selectChild node holds the functionality of the UCT policy, thus computing which child is going to be expanded next. The updateStats method is used during the backpropagation phase, in which is called for each node that led to the terminal state during the simulations.



Figure 3.2: Class Diagram showing the Algorithm incorporated with the problem environment

Chapter 4

Development

This chapter focuses on the development of my own implementation of MCTS, its design, and its integration with the specific problem domain. The chapter begins with an overview of the problem, followed by a detailed description of the MCTS algorithm and its basic components. The implementation of my MCTS algorithm will be discussed in detail, highlighting the key decisions and trade-offs made during the development process.

4.1 Implementation Framework

For the development of my MCTS implementation, I decided to use Java as the programming language. Java is a language I am familiar with and have experience in, which made it a natural choice for the project. Additionally, Java has several features that make it well-suited for implementing computationally intensive algorithms like MCTS. For example, it is a compiled language, which means that it can be optimised for performance, and it has a built-in JIT compiler that can further optimise code at runtime. One of the main advantages of using Java is its support for object-oriented programming (OOP) concepts like encapsulation, inheritance, and polymorphism. This allowed me to create a well-structured and maintainable codebase, which was especially important for implementing the algorithm. To

implement the MCTS algorithm, I used basic Java libraries such as Random, List, ArrayList, and Scanner. These libraries provided the necessary functionality for generating random numbers, manipulating lists, and reading input from the user. For the development environment, I chose to use Eclipse IDE. Eclipse has a package feature that allows me to organise my code into logical groups, making it easier to navigate and maintain. This was particularly useful as the project grew more complex. One decision I made during the implementation process was to avoid using rendering to display the simulations. Instead, everything was printed to the console. This choice was made for several reasons. Firstly, it allowed me to focus on the algorithm development itself, without the distraction of creating a graphical interface. This made it easier to debug and optimise the algorithm, which was especially important as the project grew more complex. Additionally, by avoiding rendering, I was able to reduce the complexity of the development process, which reduced the risk of errors and bugs. Overall, this approach allowed me to develop and test the MCTS algorithm more efficiently and effectively.

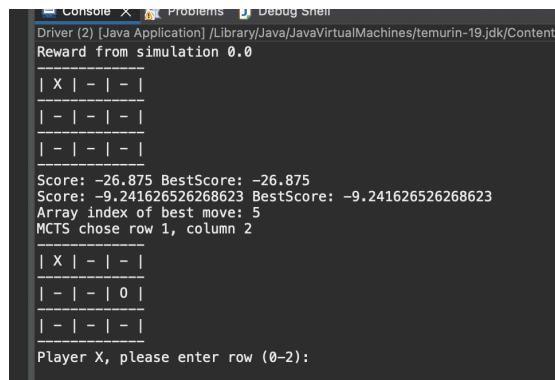
4.2 Data Structures and Classes

In this implementation of the Monte Carlo Tree Search (MCTS) algorithm, there are four main classes: MCTS, Node, State, and TicTacToe. Each of these classes has specific data structures that are used to store information and facilitate the search process. The MCTS class holds the methods needed to run the MCTS algorithm. It utilises an `ArrayList<Integer>` to store legal moves. I chose to use an ArrayList because it is a dynamic data structure that can be resized as needed. Additionally, it allows for efficient random access to its elements, making it easy to select a child node to explore based on some criteria, such as the Upper Confidence Bound for Trees (UCT) score. Furthermore, it is part of the Java Collections Framework, which makes it easy to use, and it is type-safe, which prevents type errors at runtime. The Node class stores information about each game state, such as the current game

board, its parent node, its children nodes, and an `ArrayList<I>` of legal moves. We use an `ArrayList<I>` for legal moves and children because of the aforementioned reasons. The `State` class represents the current state of the game, storing information about the current game board. I use a 2D `char[][]` array to represent the game board. This data structure is easy to understand, space-efficient, flexible, and type-safe. Additionally, accessing and modifying elements in a 2D array of characters is very fast, making the game run faster and more efficiently. Reducing the computational complexity is vital when trying to measure the efficiency of the algorithm. The `TicTacToe` class represents the Tic Tac Toe game and uses a 2D `char[][]` array to represent the game board. This data structure is easy to understand, space-efficient, flexible, and type-safe, making it an ideal choice for representing the game board. In conclusion, the chosen data structures and classes facilitate the MCTS algorithm's search process, making it efficient and easy to implement. The choices made were based on their dynamic size, efficient random access, ease of use, performance, and type safety.

4.3 Algorithm Integration

The problem environment being solved by MCTS is Tic Tac Toe, a two-player game that is played on a 3x3 grid, as shown in figures 4.1 and 4.2. The objective of the game is to get three marks in a row, either vertically, horizontally, or diagonally. The players take turns placing their marks on the game board, with the first player using one type of mark (usually an 'X') and the second player using a different type of mark (usually an 'O'). The game continues until one player successfully gets three marks in a row or until the game board is completely filled with marks without either player winning. The rules of the game are as follows. Players cannot place their marks outside the bounds of the game board or on spaces that already contain marks. Furthermore, at each turn, the current player can only place one mark in one space. These rules ensure fair play and prevent any unfair advantages for each



```
Driver (2) [Java Application] /Library/Java/JavaVirtualMachines/temurin-19.jdk/Contents/
Reward from simulation 0.0

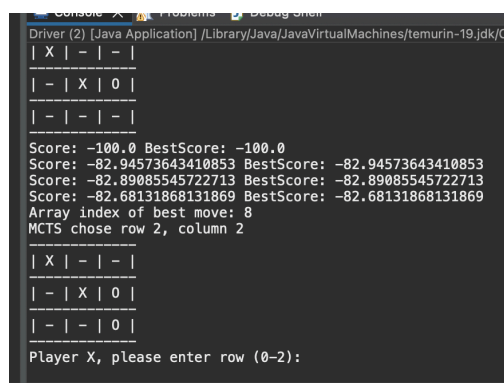
| X | - | - |
| - | - | - |
| - | - | - |

Score: -26.875 BestScore: -26.875
Score: -9.241626526268623 BestScore: -9.241626526268623
Array index of best move: 5
MCTS chose row 1, column 2

| X | - | - |
| - | - | 0 |
| - | - | - |

Player X, please enter row (0-2):
```

Figure 4.1: An image of the MCTS agent playing against a human player



```
Driver (2) [Java Application] /Library/Java/JavaVirtualMachines/temurin-19.jdk/C
| X | - | - |
| - | X | 0 |
| - | - | - |
| - | - | - |
Score: -100.0 BestScore: -100.0
Score: -82.94573643410853 BestScore: -82.94573643410853
Score: -82.89085545722713 BestScore: -82.89085545722713
Score: -82.68131868131869 BestScore: -82.68131868131869
Array index of best move: 8
MCTS chose row 2, column 2
| X | - | - |
| - | X | 0 |
| - | - | 0 |
Player X, please enter row (0-2):
```

Figure 4.2: An image of the MCTS agent playing against a human player, a couple moves later

player.

Tic Tac Toe is a relatively simple game with a small number of possible states, but it still poses a challenge to traditional algorithms due to its branching factor and the possibility of multiple optimal moves. Traditional algorithms rely on exploring the entire game tree to determine the best possible move, which can be computationally expensive and impractical for larger games or more complex game states.

The main challenge with Tic Tac Toe is the large branching factor. This refers to the number of possible moves available at each turn. In Tic Tac Toe, there are nine possible moves at the beginning of the game, which quickly expands to over 7000 possible game states, after only four moves, excluding invalid moves. This means that traditional algorithms need to explore a large number of game states to determine the optimal move, which can be inefficient both in terms of computation and time.

Another challenge in Tic Tac Toe is the possibility of multiple optimal moves. Unlike other games, such as Chess or Go, Tic Tac Toe does not have a large number of possible moves or a deep game tree, so it is possible for multiple moves to be equally optimal. This makes it difficult for traditional algorithms to choose the best move, as they may need to explore a large number of game states to determine which move is truly optimal.

To address these challenges, Monte Carlo Tree Search (MCTS) has emerged as an effective algorithm for playing Tic Tac Toe. MCTS is a heuristic-based algorithm that uses random simulations to build a tree of possible game states and then selects the best move based on the simulation results. This approach allows MCTS to quickly explore the most promising moves and avoid wasting computational resources on exploring less promising moves. Overall, while Tic Tac Toe may seem like a simple game, its high branching factor and multiple optimal moves make it a challenging problem for traditional algorithms.

MCTS is well-suited to solving game environments, such as Tic Tac Toe, efficiently, for a number of factors. One of these is MCTS's ability to handle

incomplete information. Despite not knowing what some of the opponent's moves might be in advance, due to the agent's simulation-based approach, it has the ability to plan its own moves and predict its opponent's possible moves at any state with only limited information. Its simulation-based approach also helps deal with large branching factors (large number of possible moves) as it can quickly identify the most promising moves and avoid wasting computational resources and time on less promising moves. Another reason MCTS is well-suited to solving this game environment is due to its trade-off between exploration and exploitation to make decisions. In a game where there could be multiple optimal moves, such as Tic Tac Toe, MCTS balances the need to explore new branches of the game tree with the need to exploit the most promising moves that have already been explored, thus giving it an advantage in terms of finding some of the optimal moves. Adaptive to different opponents: MCTS can be adapted to play against different opponents with varying levels of skill. By adjusting the simulation parameters and the exploration/exploitation trade-off, MCTS can learn to play optimally against opponents of different strengths. Overall, MCTS is a powerful algorithm for solving Tic Tac Toe due to its ability to handle incomplete information, balance exploration and exploitation, handle large branching factors, and adapt to different opponents. These advantages make it a useful tool for solving other games and problems as well.

In my implementation of Tic Tac Toe, I integrated the Monte Carlo Tree Search (MCTS) algorithm to enable an agent to play the game against a human opponent or a random number generator. I designed a method to run the game which instantiates an object of the MCTS class and passes the current game state into its constructor. The method then enters a loop, as shown in Algorithm 1, until an end game state is reached, during which the players take turns making moves. When it is the user's turn, they are prompted to enter the coordinates of the space on the board where they want to move. However, when it is the agent's turn, the agent's search method is called, which initiates the MCTS algorithm consisting of four stages: selection, expansion, rollout, and backpropagation. When simulating, the agent comes across

Algorithm 1 Search Method in MCTS class

```
1: for i to simulation count do
2:   Check if the current node is a leaf node
3:   while current isn't leaf do Current = selectNode(Current)
4:   end while
5:   If the current node is a leaf node and has not been visited yet
6:   if c then current node visits == 0
7:     simulate(current)
8:   else
9:     current.expand()
10:    current = current.getChildren()[0]
11:    simulate(current)
12:  end if
13: end for
```

a getReward method at a terminal state. In this method, checks are done to see if the state is a winning, losing or drawing state. There are also extra checks as part of a rewards shaping implementation which check for 2 pieces in a row. The aim behind this functionality, is to influence the agent to adapt aggressive strategy and to try to get 2 marks next to each other. The agent has incentive to do this because if it achieves it a small reward is backpropagated up the tree. Once the agent has finished backpropagating the results from the simulation, its 'get best move' method is called which computes the best move to make by dividing the total reward by the visit count for each of the nodes; a higher value indicates a node that is more likely to reach a winning state. This method returns the move the agent selects as an index of the position on the board.

There are many possible variations, modifications and adaptations that can be made to the MCTS algorithm. In this particular implementation, the state representation was modified to better suit this game environment. This was done through simplification. As opposed to making the algorithm deal with multiple

coordinates, when understanding moves, making moves, updating states with moves, etc. I combined the 2 coordinates into a simple index value representing the position on the board. This allowed the agent to fully understand which moves it can make, etc without the complexity of getting it to understand 2 separate coordinates and work with them. The row and column values can easily be computed from this index value by dividing it by the amount of rows, and applying the modulo operation with the number of columns respectively. The state representation was also augmented in terms of adding additional methods to aid the decision-making process. This was done by adding methods to check for win states. Doing this allows the agent to determine which terminal states are wins, losses or draws during the simulation phase. This is vital in the decision making process as this determines which reward values are backpropagated up the tree, thus enabling the agent to choose the best/near optimal moves when it has finished simulating.

Another modification made to the MCTS algorithm was the UCT formula. Adding this in allows the agent to balance exploration and exploitation. This is vital for this particular game environment because, as stated earlier, there are many potential optimal moves within the game of Tic Tac Toe. Balancing exploration with exploitation gives the agent an advantage in finding multiple near-optimal solutions as it can either choose to play moves that it knows are likely to lead to winning states, or it can choose to explore new game trees where more optimal moves could reside. If another selection policy was used such as the Best-First Search, this may not be the optimal search policy. The best-first search strategy prioritises nodes with the highest estimated value, rather than the highest UCT value. This can be useful for problems where the goal is to find a single optimal solution rather than a distribution of solutions. In problem environments such as that of the Tic Tac Toe environment, there may not be a single optimal solution at any game state, so it would likely be computationally impractical to employ a search function such as this.

Another possible search strategy could have been Thompson sampling. Thomp-

son Sampling is a Bayesian approach to MCTS that is based on sampling from a probability distribution to estimate the value of each node. While it can be effective in some cases, Thompson Sampling can be computationally expensive, especially if the number of possible moves or the complexity of the game is high. On the other hand, UCT is an effective selection strategy in MCTS for Tic Tac Toe due to its ability to balance of exploration and exploitation while converging to one of the optimal solutions quickly and efficiently. While Thompson Sampling is also a valid selection strategy, it may not be as efficient for problems like Tic Tac Toe that have a relatively small search space.

During the development process of my implementation of Monte Carlo Tree Search (MCTS) on Tic Tac Toe, I encountered a couple of challenges related to getting the simulations to work effectively. Firstly, I discovered that the structure of the search method needed to be revamped since I was calling 'select' on the root node before it had any children. To fix this, I had to reevaluate how the agent should run and expand the root node once it was initialised. Then, during simulations, I had to ensure that the 'current node' was a leaf node before running simulations, selecting a child based on the Upper Confidence Bound (UCT) policy, and repeating the process until a leaf node was found. If the leaf node had been visited, simulations were run. Otherwise, the leaf node was expanded, and simulations were run from the first child node. Secondly, I encountered complexity issues related to applying the algorithm on rendered games. Thus, I decided to create my own implementation of the Tic Tac Toe environment and use console inputs and outputs to represent the game board, as well as inputting moves. This helped me separate the game state from the class that runs the game, thereby simplifying the process of enabling simulations. These changes allowed me to overcome the challenges in the MCTS implementation on Tic Tac Toe, enabling the agent to explore the search space effectively and converge towards the optimal solution. Overall, the modifications improved the performance and efficiency of the MCTS algorithm on Tic Tac Toe, making it a viable solution for other similar games and problems.

4.4 Implementation Issues

During the implementation of the MCTS algorithm, I faced a number of issues that required my attention. The following paragraphs will discuss these implementation issues and how I resolved them. The first issue I faced was getting the simulations to work correctly. After reviewing the structure of the search method, I realised that I was calling "select" on the root node before it had any children. To fix this issue, I reevaluated how the agent should run. I concluded that the agent cannot select a child node from the root node before it has been expanded. As a result, I had to expand the root node once it was initialised. Then, I checked that the "current node" was a leaf node before running simulations. If it wasn't, I selected a child based on the UCT policy and repeated the process until a leaf node was found. After finding a leaf node, I checked if it had been visited before. If it had, I ran the simulations; if it hadn't, I expanded the leaf node, chose the first child, and ran simulations from there. This approach resolved the issue and allowed the simulations to run correctly. The second issue I encountered was that the algorithm was originally applied to rendered games. This meant that separating the game state from the class that runs the game was required to enable simulations, which added unnecessary complexity. To address this issue, I made my own implementation of the tic tac toe environment and used console inputs/outputs to represent the game board and input moves. This approach simplified the implementation of the algorithm and enabled me to focus on optimising it without worrying about the game's implementation details. Overall, addressing these implementation issues was crucial in ensuring the successful implementation of the MCTS algorithm. By reevaluating the search method structure and creating a simplified environment, I was able to overcome these issues and develop a working implementation of the algorithm.

Chapter 5

Experiments and Results

The aim of my tests and experiments on MCTS in Tic Tac Toe were to investigate how adjusting the number of simulations and adding rewards shaping affects its performance in terms of efficiency and accuracy. My hypotheses were that a higher number of simulations would increase the algorithm's performance but decrease its efficiency, and that rewards shaping would improve both its accuracy and efficiency. I ran several experiments varying the number of simulations and the implementation of rewards shaping and recorded the win rate and average simulation time for each experiment. When evaluating the performance of my algorithm, I used 2 metrics: win rate (per 10 games) and mean simulation time (over 10 games). I gathered data for these metrics by running games, in which the agent would play against a random number generator which generated random coordinates for its moves. I then analysed the results and compared them to my hypotheses. The findings of the experiments are presented in the following chapter of this report.

When designing each test case, I applied a systematic approach in which there would only be 1 independent variable per test case. In total, there were 4 different test cases. For each test case I ran 10 games and recorded the number of wins, and the average simulation time per 10 games. There were 2 tests in which rewards shaping was enabled and 2 in which it was disabled (the algorithm solely relied on the rewards backpropagated from end game scenarios). For each of these there

was a version in which the agent started playing first and a version in which the random number generator started first. Within each of these test cases, the number of simulations varied from 10 per move, to 5000 per move to 10000 per move. This was to see how each of these variables affected its performance.

Agent Starts			
Rewards Shaping Disabled			
Number of Simulations	Win Rate (/10 games)	Average Simulation Time Per Game (ms)	Total Simulation Time (ms)
10	4	0.437172966	38.2640425
5000	7	0.034366073	1761.70245
10000	5	0.026061412	2606.1412

Figure 5.1: The table of results from the test case where the agent made the first move, and rewards shaping was disabled

Agent Starts First			
Rewards Shaping Enabled			
Number of Simulations	Win Rate (/10 games)	Average Simulation Time Per Game (ms)	Total Simulation Time (ms)
10	3	0.440129337	44.0129337
5000	3	0.032345171	1617.25855
10000	2	0.024300223	2430.0223

Figure 5.2: The table of results from the test case where the agent made the first move, and rewards shaping was enabled

Opponent Starts			
Rewards Shaping Disabled			
Number of Simulations	Win Rate (/10 games)	Average Simulation Time Per Game (ms)	Total Simulation Time (ms)
10	0	0.455706667	45.5706667
5000	2	0.031449567	1572.47835
10000	5	0.02476892	2476.892

Figure 5.3: The table of results from the test case where the agent's opponent made the first move, and rewards shaping was disabled

In terms of whether my results support my hypotheses, some where proven correct and others not. Apart from some anomalies, as the number of simulations increased in each test case, the win rate also increased, as well as the total simulation time over the 10 games, thus indicating that increasing the number of simulations increases performance, whilst reducing efficiency. This is because, as the algorithm runs more simulations, it gets a better understanding of the state space,

	B	C	D	E
6	Number of Simulations	Win Rate (/10 games)	Average Simulation Time Per Game (ms)	Total Simulation Time (ms)
7	10	1	0.382640425	38.2640425
8	5000	3	0.035234049	1761.70245
9	10000	1	0.028398852	2839.8852

Figure 5.4: The table of results from the test case where the agent's opponent made the first move, and rewards shaping was enabled

especially when new states are visited, thus allowing it to build a comprehensive list of promising moves to make. However, running more simulations, requires more computational costs, thus slowing the algorithm down and making it more inefficient. My implementation of Rewards Shaping did not seem to affect any of the performance metrics, as indicated by the graphs. This could be due to not having a large enough reward for getting 2 marks in a row compared to the end game reward. If the reward values in the reward shaping method, were higher, it is likely they would have influenced the agent to place more marks in a row.

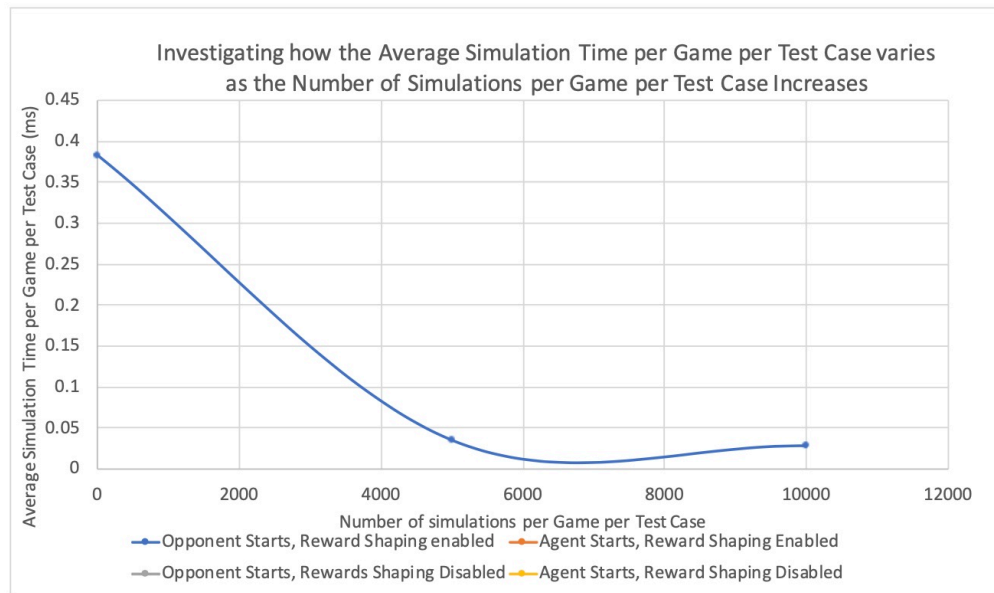


Figure 5.5: A scatter graph showing the relationship between average simulation time and the number of simulations

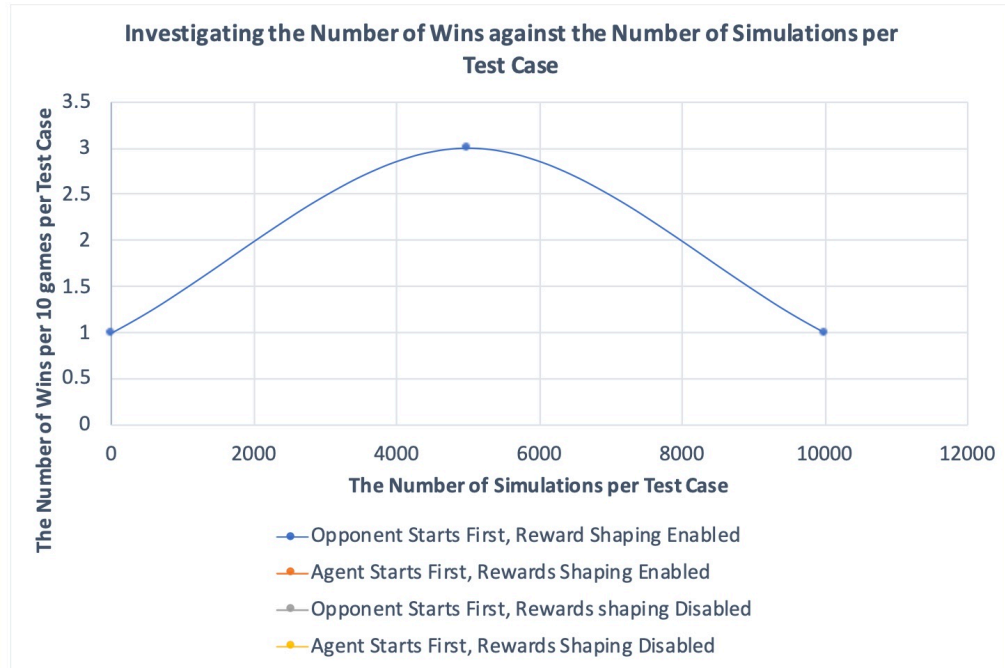


Figure 5.6: A scatter graph showing the relationship between the number of wins and the number of simulations

Chapter 6

Conclusions

This paper has investigated the impact of adjusting the configuration of the Monte Carlo Tree Search (MCTS) algorithm in Tic Tac Toe by varying the number of simulations and enabling rewards shaping. The key findings suggest that increasing the number of simulations resulted in an improvement in the algorithm's decision making ability, while having a light impact on its computational efficiency. On the other hand, enabling rewards shaping led to a slight reduction in the total simulation time, indicating an improvement in the algorithm's simulations by playing more moves that lead to winning states. However, there was no significant increase in the algorithm's decision making, which could be due to several reasons, such as the need for further rewards shaping or a lack of sufficient data. This research highlights the importance of exploring various configurations of MCTS algorithms to enhance their performance in lower complexity problem domains such as Tic Tac Toe.

6.1 Review of Aims

The first aim, which was to develop an MCTS algorithm that could select, expand, simulate, and backpropagate consistently, was achieved successfully. The algorithm demonstrated robustness in performing these functions during the testing phase of its implementation in which it had no issue obtaining the current state of

its environment, simulating into different states, and determining which move to make based on the value of the states computed from those simulations. However, the second aim, which was to discover whether the algorithm could achieve the desired reward state consistently, was not achieved. The experiments section showed that the algorithm did not achieve the desired reward state often, and in fact, consistently lost when playing against a random number generator. This could be due to variety of factors, for example: a lack of exploration; the algorithm could have been biased towards certain moves or strategies, leading to a failure to explore the full range of possibilities. It could also be due to suboptimal rewards shaping as mentioned before. In this particular environment there are a multitude of ways in which rewards shaping strategies could have been implemented beyond what already exists within this version. A more in depth reward function that encouraged more promising strategies and discouraged more bad moves, could have developed the agent's decision making ability to a great extent whilst increasing efficiency by discouraging selection, expansion and simulation of suboptimal nodes. Therefore, a promising solution to this issue, that could have achieved this aim, would be to develop a more refined rewards shaping function. The third aim, which was to determine whether the algorithm could maintain computational and time efficiency, was achieved successfully. The algorithm consistently had a low average simulation time across multiple experiments, not going above 3 seconds to make a decision despite a large variety in the number of simulations within the tests, thus indicating computational efficiency. Therefore, proving that despite the original intention of the algorithm to solve large, complex problem domains efficiently, it still maintains efficiency in environments significantly less complex than what it was designed for. In regards to the fourth aim, which was to determine the effects of rewards shaping on the algorithm's competence and efficiency, the algorithm did not seem to improve with added rewards shaping, only achieving a barely noticeable improvement in computational efficiency. Thus, this aim was not achieved. As stated before, a more refined reward function could have improved the agent's

decision making. Furthermore, it could have achieved this aim and given more insight into the potential utility of MCTS in lower complexity environments and possibly within computer architectures with lower computing capacity. Finally, the fifth aim was partially achieved, as the experiments showed that an increased number of simulations improves the algorithm's performance. However, the maximum depth of the search tree was not experimented with, thus limiting the full achievement of this aim.

6.2 Possible Design/ Implementation Revisions

If the algorithm were to be revised in terms of design/ implementation, there are few possible routes to go down. The first of which could be to adjust the exploration parameter. The UCT policy determines how much MCTS should explore new nodes versus exploit existing knowledge. Increasing the exploration parameter, thus making the algorithm less greedy, could lead to a greater variety of strategies being employed. By doing this, the algorithm could have a more diverse database of strategies to choose from. This could improve the agent's ability to win as it provides greater potential for playing in new ways. The experiments in this paper suggest that rewards shaping had a slight impact on computational efficiency but did not significantly improve decision making. Improving the reward function to incentivise blocking the opponent's moves or giving higher rewards for certain moves could improve the algorithm's ability to achieve the desired reward state in the same way as adjusting the exploration parameter. By developing a fully refined reward function, the agent could see massive improvements in decision making, due to good strategy actually being coded into the way the algorithm makes decisions. Lastly, the use of domain-specific knowledge in a simple game such as Tic Tac Toe which has a well-defined set of rules, could improve the algorithm further in the same way as the aforementioned revisions. Incorporating domain-specific knowledge, such as strategies for playing certain moves or identifying common patterns could

provide additional strategy in the planning process. The linking factor amongst these improvements is the fact that they all add to MCTS's ability to strategise and understand the problem domain in a deeper, more competitive way.

6.3 Closing Remarks

To conclude, I believe that this project has been a success in researching the MCTS algorithm in a novel way; with a low complexity environment and algorithm configurations that reduce computational complexity while increasing efficiency and effectivity. However, this is not to say that the project was a complete success, as some aims were not met, as evidenced in the experiments section. I would have liked to have added more algorithm configurations or even implement a different variation of MCTS, and to have also met all the aims of the project. The methodology applied in order to complete this project was a comprehensive study of the available relevant literature and textbooks followed by constant code iterations. The literature served, at first, as the main way of understanding how to begin implementing the code. Once I had built a good understanding and begun implementation, it served as the reference for fixing any bugs, or filling in gaps in my comprehension, during development. A positive of this methodology is that it enabled the successful implementation of the algorithm, and it helped in gaining a new understanding of how AI algorithms can work and their potential for future development. It also taught me how to research scientifically and develop both fundamental programming skills, such as debugging, as well as how to manifest a complex concept in code.

Chapter 7

References

1. "Mastering the Game of Go with Deep Neural Networks and Tree Search" (D. Silver et al, 2016)
2. "Real-Time Monte Carlo Tree Search in Ms Pac-Man" (T. Pepels et al, 2014)
3. "Artificial Intelligence: A Modern Approach 4th ed" (S. Russel et al, 2020)
4. "An Analysis of Monte Carlo Tree Search" (S. James et al, 2017)
5. "Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search" (R. Coulom et al, 2006)
6. "Bandit Based Monte-Carlo Planning" (L.Kocsis et al, 2006)
7. "Monte Carlo planning in large POMDPs" (D. Silver et al, 2010)
8. "A Survey of Monte Carlo Tree Search Methods" (C. Brown et al, 2012)
9. "Real-time Monte-Carlo Tree Search in Ms Pac-Man" (T. Pepels et al, 2014)
10. "The Impact of MCTS Payout Budget in Monte-Carlo Go" (Enzenberger et al, 2010)
11. "Effective Pruning of Monte Carlo Tree Search Using Progressive Bias and Rollout Horizon Control" (Nagai et al, 2015)

12. “Parallel Monte-Carlo Tree Search” (Chaslot et al, 2008)
13. “Mastering the game of Go with deep neural networks and tree search” (D. Silver et al, 2016)
14. “Monte-Carlo tree search and rapid value estimation in computer Go” (D. Silver et al, 2011)

Appendix A

Project Proposal

Monte Carlo Tree Search in Solitaire

Abstract

This project aims to investigate the Monte Carlo Tree Search and its capabilities in a Partially-Observable Markov Decision Process (POMDP).

The Monte Carlo Tree Search is a relatively new algorithm that has already proven to be very effective, in large, fully observable domains. Furthermore, a variant of this algorithm that could provide just as good a performance in only partially observable domains could contribute to the overall development of Artificial Intelligence algorithms.

As the environment in Solitaire is only partially observable, a version of the Monte Carlo algorithm, called Partially Observable Monte-Carlo Planning (POMCP), will need to be used. The initial stage of this project will involve developing the POMCP. This algorithm will then be tested and developed in games of the computer version of the card game, Solitaire.

The results of this project are expected to show that the algorithm will be able to solve all 'winnable' games of Solitaire at a speed much faster than humanly possible. As some games are simply impossible to win in Solitaire, the algorithm is expected to complete as much as possible and then detect that the game is unwinnable.

Introduction

Since its creation, Monte Carlo methods have had much success in AI algorithms, especially those used to play digital games. It could be the best current AI algorithm achieving superhuman performance levels in games such as Go and Ms Pac-Man whilst also beating all other game-playing AI in multiple competitions [1][2].

A Markov Decision Problem (MDP) is a sequential decision problem for a fully observable, stochastic environment with a Markovian transition model and additive rewards [3]. A POMDP has the same elements as MDP (transition model, actions, reward function) and a sensor model due to its partially observable nature. To extend Monte Carlo simulation to POMDPs, a history-based rollout policy must be used [4].

This proposed system aims to expand on current MCTS implementations and to see whether it can thrive in partially observable environments as well. This will be achieved by implementing a POMCP and developing it to be able to not only play Solitaire, but to be able to beat it in superhuman times.

This project proposal will be split into the following sections: background, project plan, overall workplan and references.

The background section will provide a synopsis of the related work, how this project fits into the existing literature and comparison to the variations of this algorithm. The project plan section will detail the aims and objectives of this project and it will provide a detailed methodology of how this algorithm will be developed and how it will be evaluated. The workplan section will contain a plan for each task in the project's schedule and an estimate for each. The references section will include all references to any of the resources used.

Background

Many papers have been written on Monte Carlo Tree Search and its variations. One of these focuses on an implementation for the game Go [5]. In this implementation, Monte Carlo Tree Search is combined with policy and value networks to produce the algorithm for AlphaGo (the strongest playing Go AI).

This algorithm is right for Go because of the game's depth and complexity in terms of potential moves and sequences of moves. However, this issue differs from the one relating to this proposed project. In this project, the issue isn't so much to do with the complexity of the game but more so its partially observable nature. Therefore, it would be interesting to see how well the algorithm can cope with the lack of foresight in terms of not knowing which turned over cards are in which place.

Another paper tested the effectivity of MCTS in large POMDPs [4]. In this paper, an algorithm, POMCP, is developed which is a combination of Monte Carlo belief state updates and PO-UCT. This algorithm also shares the same simulations for both Monte Carlo simulations. Traditionally, POMDP planning focused on small problems with few states. This investigation used more challenging POMDPs, and POMCP was able to achieve high performance in these POMDPs with only a few seconds of online computation. This is because the Monte Carlo simulation provided an effective mechanism for tree search and belief state updates.

However, the environments in this investigation differ from the environment in the game used in this proposal despite them all being POMDPs. This is because the Solitaire environment isn't so much an environment within which moves can be made as it is a sorting game where the player must sort the cards into the correct pile efficiently whilst observing the rules and constraints. This is interesting because it suggests that despite it being the same kind of problem in that it isn't fully observable, it may require a different variation of the algorithm.

Specifically, this project will investigate the effects of applying MCTS to a different environment in terms of whether a new variation will be needed or whether it confirms that one of the algorithms suffices for the suggested environment. If a new variation is needed, this could potentially be a new development in this area of research.

Project Plan

Aims and Objectives

The overall aim for this project is to develop an algorithm based on the Monte Carlo Tree Search that can learn not only how to play a game within a partially observable environment but achieve a high level of proficiency in it. Specifically, this includes:

- Creating and developing the algorithm to be tested – the algorithm and each of its elements will all need to be created and configured to be able to play Solitaire.
- The next objective is to develop the algorithm to a point of mastery at which point it will be able to:
 - Complete each game quicker and more efficiently than humans are capable of.
 - It should also be able to discover when a game cannot be won just as easy or quickly as it wins them.

Methods

The method to this project will be composed of 2 distinct phases: creation, and development/ configuration.

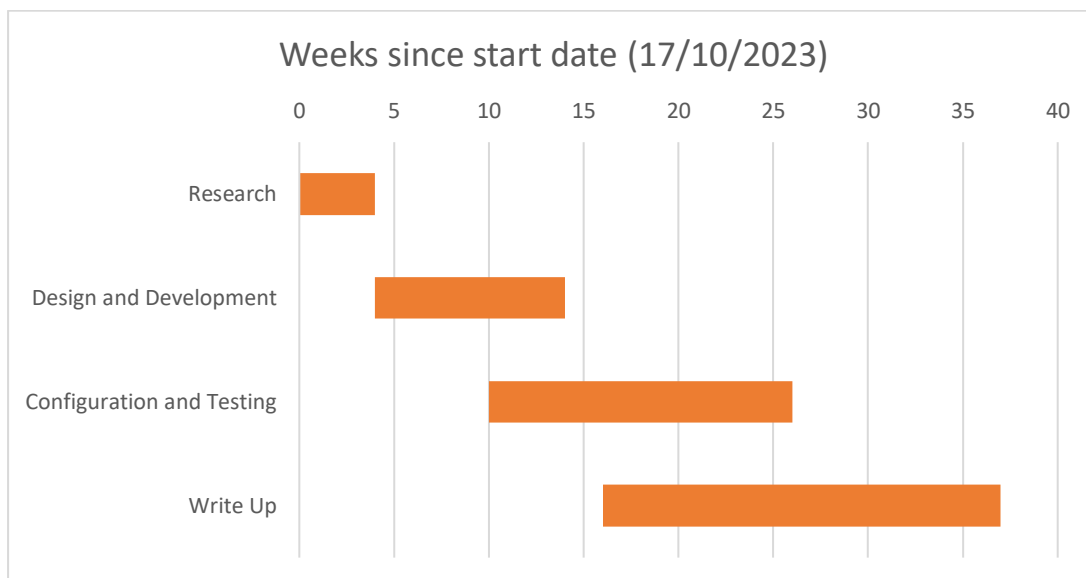
In the creation element of this project, the algorithm will be researched, designed, and coded. Research will help with understanding the logic of the algorithms and equations as well as how to design and implement each aspect of the overall algorithm.

In the development/ configuration element of this project, the algorithm should either be complete or not far off. It will then be tested, and its performances will be reviewed and analysed to see which aspects of the algorithm need further configuration. The program will then be retested, and this process will repeat until an optimal version of the program is found which ideally completes all the objectives. When analysing the results from each game, progress will have been made if the algorithm solves the game quicker or more efficiently (less moves).

Overall Workplan

This project will begin in early November 2022 and end March 2023. The work will be split into the following sections:

- Research
- Design and development
- Configuration and testing
- Write up of findings



References

1. Mastering the Game of Go with Deep Neural Networks and Tree Search (D. Silver et al, 2016)
2. Real-Time Monte Carlo Tree Search in Ms Pac-Man (T. Pepels et al, 2014)
3. Artificial Intelligence: A Modern Approach 4th ed (S. Russel et al, 2020)
4. Monte Carlo planning in large POMDPs (D. Silver et al, 2010)
5. Mastering the Game of Go with Deep Neural Networks and Tree Search (D. Silver et al, 2016)
6. A Survey of Monte Carlo Tree Search Methods (C. Brown et al, 2012)

Appendix B

Additional Tables

Test Number	Win count y/n	Average Simulation Time (ms)
1	n	0.465294933
2	y	0.385382325
3	y	0.38389728
4	n	0.536607525
5	n	0.4464779
6	n	0.48310525
7	n	0.3849189
8	n	0.46249695
9	y	0.38237868
10	n	0.470733625
Average Simulation Time Per Game (ms)		0.440129337

Figure B.1: This table is an example table used when attaining the statistics from each match. The outcome of the match is represented by 'y' or 'n' and the average simulation time per game is calculated at the bottom.