University of Cincinnati

Date: 11/9/2016

I, Subrahmanya Srivathsava Sista, hereby submit this original work as part of the requirements for the degree of Master of Science in Computer Science.

It is entitled:

Adversarial Game Playing Using Monte Carlo Tree Search

Student's name: **Subrahmanya Srivathsava Sista**

This work and its defense approved by:

Committee chair: Anca Ralescu, Ph.D.

Committee member: Chia Han, Ph.D.

Committee member: Paul G. Talaga, Ph.D.

22414

Adversarial Game Playing Using Monte Carlo Tree Search

A thesis submitted to the

Department of Electrical Engineering and Computing Systems
of the
University of Cincinnati

in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in the School of
Computing Sciences & Informatics
of the College of Engineering & Applied Science
October 24, 2016

by

Subrahmanya Srivathsava Sista
B.Tech. (Computer Science and Engineering), Andhra
University, April 2013

Thesis Advisor and Committee Chair: Dr. Anca Ralescu

Abstract

Monte Carlo methods are a general collection of computational algorithms that obtain results by random sampling. Monte Carlo techniques, while great for simulation, have also found great application in the field of general game playing. We investigate the effectiveness of Monte Carlo methods as applied to general two player games (In this case we use a more interesting variant of the popular game Tic-Tac-Toe: fully observable, deterministic, static, single-agent environment). We set up AI agents, one using Monte Carlo simulation to play and the other using a more traditional mini-max setup. We compare and contrast their performance in all aspects, including efficiency, effectiveness, and cost in terms of memory/processing. After all the data collection and analysis we found that Monte Carlo Techniques tended to perform better relative to the Minimax algorithm when applied to a game of our choice and with restrictive time limits.

Acknowledgment

I offer my sincere gratitude to my advisor Dr. Anca Ralescu, for taking me in and offering the support needed to complete my thesis research.

I offer my thanks to my committee members, Dr. Chia Han and Dr. Paul Talaga for taking the time for my defense and their feedback.

A special thanks to Dr. Paul Talaga as I began my work under him and he offered nothing but encouragement towards my work.

Finally, my sincere gratitude to my parents, my brother, and my entire family for their support and help while I completed this work.

Contents

1 Introduction **1**

1.1 General Research Objective . . . 2

1.2 Specific Research Objective . . . 2

1.3 Research Methodology . . . 2

1.4 Contributions of this Research . . . 3

1.5 In This Document . . . 3

2 Overview of Our Algorithms **5**

2.1 An overview of Monte Carlo Tree Search . . . 5

2.1.1 Steps Involved in Monte Carlo Tree Search . . . 5

2.1.2 Upper Confidence Bound For Trees . . . 8

2.1.3 Characteristics And Popular Applications of MCTS . . 12

2.2 Variations of Monte Carlo Tree Search . . . 14

2.3 An Overview of Advanced Tic-Tac-Toe . . . 15

3 Implementation and Test Parameters **22**

3.1 Our Implementation of MCTS . . . 22

3.2 Our Implementation of Minimax 23
3.3 Parameters of Test 26
3.3.1 Machine Specs 26
3.4 Initial Observations 27

4 Results and Observations 29
4.1 Test 1 29
4.2 Test 2 30
4.3 Test 3 31

5 Conclusion and Future Work 34
5.1 Conclusion 34
5.2 Future Work 35

Appendices 37

A 38

B 40

List of Figures

2.1 Figure Explaining Monte Carlo Tree Search 6

2.2 Figure Explaining Monte Carlo Tree Search 7

2.3 Figure of an Ultimate Tic-Tac-Toe Board 16

2.4 Square in an Ultimate Tic-Tac-Toe Board 17

2.5 Winning a Board in an Ultimate Tic-Tac-Toe Board 18

2.6 Winning a game of Ultimate Tic-Tac-Toe 19

2.7 Playing In The Top Right Square 20

2.8 Playing In The Top Right Board 21

4.1 Scatter plot of the time taken per move and the average time taken for Test 1 30

4.2 Scatter plot of the time taken per move and the average time taken for Test 2 31

4.3 Scatter plot of the time taken per move and the average time taken 32

4.4 Plot Showing Convergence of Win Rates of Both Methods . . 33

List of Tables

4.1 Table of results for Test 1 29

4.2 Table of results for Test 2 30

4.3 Table of results for Test 1 32

Chapter 1

Introduction

Game playing in AI has always been a domain-specific problem. Depending on the type of game being played as well as its own rules and quirks, we often have to tweak or completely re-write the algorithms we plan to use to simulate playing it. While there do exist General Game Playing algorithms which attempt to play more than one game successfully, they often rely on a framework of rules being given to them which describes the game they are about to attempt playing [1]. That said, we have seen amazing success for AI in games where focused research is done. In Chess for example, the AI agents are already able to beat the top ranked players on a regular basis.

Now research has shifted to other games but also back to general game playing in an attempt to be able to create an AI that can act as an average player in the game, if not an exceptional one.

1.1 General Research Objective

The general research objective is to compare the approach and performance of a Monte Carlo approach to the thesis as opposed to a traditional Minimax approach.

1.2 Specific Research Objective

In order to achieve this accurate comparison of the two methods, we must also:

1. Select a game for these AI agents to play. Here we have chosen a fully observable, deterministic game with a fixed number of total moves.
2. Set up the framework and rules for the game (Here we use Advanced Tic-Tac-Toe, the rules of which are explained in TODO)
3. Set up and create different AI agents which follow a Monte Carlo approach as well as more traditional approaches (here we use Minimax approach)

1.3 Research Methodology

In order to achieve these research objectives, I took the following steps:

1. Study the current literature on Monte Carlo methods. There have been

several papers, both of original research and survey papers which cover Monte Carlo methods exhaustively.

2. Identify a standard of performance we can expect from the traditional approaches to the creation of an AI agent

3. Analyze the performance over time of the Monte Carlo approach and the traditional approach

1.4 Contributions of this Research

1. Finding the effectiveness of modern Monte Carlo methods as opposed to traditional heuristic-based methods for relatively simple games.

2. Finding ways to improve and optimize these Monte Carlo techniques depending on the demands.

1.5 In This Document

In part 2, we describe the game that we have used for this test as well as the standard use and working of the Monte Carlo Tree Search algorithm.

In part 3, we detail the rules we have set for ourselves in comparing Monte Carlo Tree Search to the Minimax algorithm as well as detailing the specifics of our implementation of each.

In part 4, we detail the results of our tests.

In part 5, we discuss the implications of the results and conclude with potential improvements and future work that may arise from what we have learned.

Chapter 2

Overview of Our Algorithms

2.1 An overview of Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a method for finding optimal decisions in a given domain by taking random samples in the search space and building a search tree according to the results [2].

2.1.1 Steps Involved in Monte Carlo Tree Search

The basic process of MCTS is a very simple concept. A tree is built in the search space, asymmetrically. For each iteration, it goes through 4 steps.[3]

1. Selection: An optimal node is selected from the tree based on the tree policy.
2. Expansion: If the selected node is not a terminal node, then the possible

child nodes are created and one of them is selected (call it C)

3. Simulation: A simulated playout of the game is run from C until the game is ended i.e. a terminal node is reached, based on the default policy.

4. Back-propagation: The result of the simulation is returned back over the tree. This could be a simple statement of win/loss or the final score if we also wish to determine the margin of a victory/loss.

A clear distinction must be made between the tree policy and the default policy. The tree policy determines the selection (or creation after expansion) of a child node from the nodes that already part of the tree, whereas the default policy determines the simulation of the game from the selected node.[2]

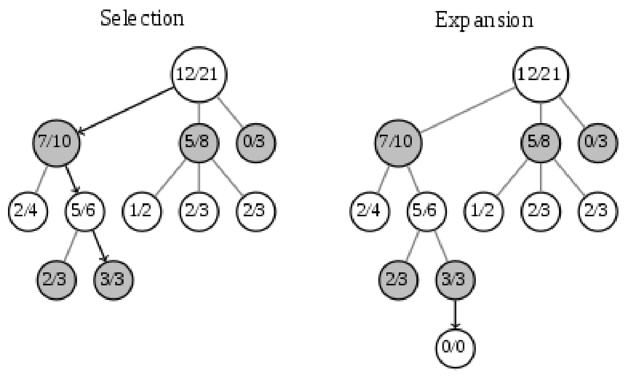

Figure 2.1: Figure Explaining Monte Carlo Tree Search

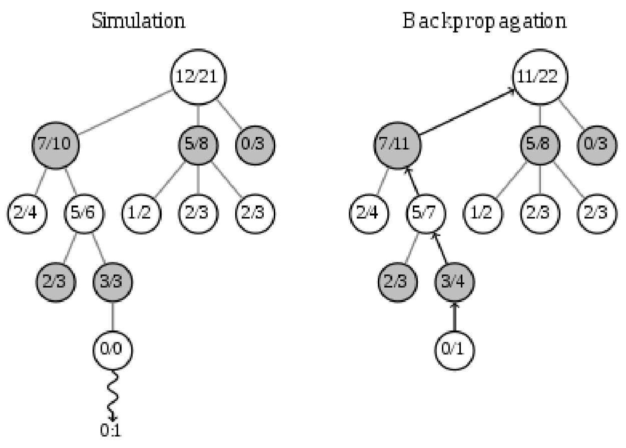

Figure 2.2: Figure Explaining Monte Carlo Tree Search

Figures sourced from Wikimedia Commons - by Mciura / CC-BY-SA / Split into two separate images.

The selection process relies on a tree policy. This policy must attempt to balance exploration (to find different paths in the tree and possibly stumble upon more optimal solutions) with exploitation (following what we know to be more optimal paths in order to achieve good results). At its most rudimentary stage, the tree policy would simply be random selection.

The expansion process again relies on random selection of a child node. This is intentional as MCTS works on the basis of fast, repeated simulations to get as much data as possible, as quickly as possible.

The simulation stage is where the bulk of the work happens, and it is really just a constant iteration of step 2 until we reach a node that no longer has any children. The back-propagation stage is the one which returns the result of that particular playout. In our case it will be a simple win/loss binary result.[3]

The framing of the tree policy is vital to obtaining a good result with the MCTS approach. A completely random approach will not yield results which are as good as a more carefully constructed method as it would occasionally ignore the paths that have statistically been proven by our information to work better.

2.1.2 Upper Confidence Bound For Trees

The most popular MCTS based algorithm is the Upper Confidence Bounds for Trees (UCT) algorithm.[4] It is in turn based on the UCB1 formula derived by Auer, Cesa-Bianchi and Fischer.[5] It frames out a simple formula for the selection of a node for the tree policy which provides a decent balance between exploration and exploitation.

```
Data: State of the board $s_0$
Result: The optimal move to make
Function UctSearch($s_0$) is
    create root node $v_0$ with state $s_0$;
    while within computational budget do
        $v_l \leftarrow TreePolicy(v_0)$;
        $d \leftarrow DefaultPolicy(s(v_l))$;
        Backup($v_l, d$);
    end
    return $a(BestChild(v_0, 0))$;
End

Function TreePolicy($v$) is
    while $v$ is non-terminal do
        if $v$ not fully expanded then
            return Expand($v$);
        else
            $v \leftarrow BestChild(v, C_p)$;
        end
    end
    return $v$
End
```

```
Function Expand(v) is
    choose $a \in untriedactionsfromA(s(v))$;
    add a new child $v'$ to $v$;
    $s(v') \leftarrow f(s(v), a)$;
    $a(v') \leftarrow a$;
    return $v'$
End

Function BestChild(v, c) is
    return $\underset{v' \in childrenofv}{\arg max} \frac{Q(v')}{N(v')} + c\sqrt{\frac{2 \ln N(v)}{N(v')}}$
End

Function DefaultPolicy(s) is
    while s is non-terminal do
        choose $a \epsilon A(s)$ uniformly at random;
        $s \leftarrow f(s, a)$;
    end
    return reward for state s
End
```

```
Function Backup(v,d) is
    while $visnotnull$ do
        $N(v) \leftarrow N(v) + 1$;
        $Q(v) \leftarrow Q(v) + d$;
        $d \leftarrow -d$;
        $v \leftarrow parentofv$;
    end
End
```

In this algorithm, each node has 4 fields of data associated with it.

$s(v)$ is the state of the board.

$a(v)$ is the next move from that node.

$Q(v)$ is the total reward at that node so far (in our case just the number of wins).

$N(v)$ is the number of times the node was visited (an integer greater than or equal to zero).

C_p is the constant which balances exploitation with exploration in the algorithm. By default its value is $\frac{1}{\sqrt{2}}$.

$d(v,p)$ represents the reward vector for player p at node v.

Once all the iterations are completed, the winning action is selected. This can be done in many ways.

1. Select the action with the highest reward ($Q(v)$).

2. Select the action with highest reward to playthrough ratio ($\frac{Q(v)}{N(v)}$).

3. Select the action with the highest number of visits i.e. the most robust. ($N(v)$)

4. Select an action with any customized parameter of your choice which suits your purposes. For example, one may choose to select the highest win rate action which also has a certain minimum number of visits.

2.1.3 Characteristics And Popular Applications of MCTS

The characteristic of MCTS that make it so promising and useful in the field of AI is that it is independent of a heuristic. In games where we do not have a particularly elegant way of evaluating the state of the game in order to determine the next move, MCTS comes in very handy as it does not rely on any such measurements. It takes quick, random moves to obtain statistical data. So the algorithm does not care about the reason that its moves are succeeding/failing. It simply uses the statistical data obtained to make a decision towards the one which is bringing it more success over time. In addition, this approach does not require much domain knowledge about the game itself and it is possible to create an agent for the game by simply having a knowledge of the game rules and not necessarily the tips and tricks needed to be a good player, as the agent eventually figures that out for itself.

In addition, Monte Carlo Tree Search is an "any time" algorithm. It can be halted at any point during the simulation and the most promising results obtained to that point can be used. This allows us to fine tune it for any situation and restrictions, whether they are time based or memory based. The algorithm can be configured to halt after the search tree reaches a certain size or after a certain amount of time has passed, or any combination of the two. This makes it more tolerant to failures and more flexible.[6]

MCTS also forms asymmetric trees in its exploration. Nodes or sections of the search tree which are found to be more promising are explored more thoroughly and too much computational power is not wasted on the less promising branches of the tree.

Lastly, MCTS is highly parallelizable. As each simulation runs independent of other, the algorithm does not require any time sensitive communication between multiple threads of the process. Parallelizing the algorithm also allows us to favour more exploration, possibly finding more optimal routes that may have been missed.[7]

The MCTS approach has seen great results for the popular game of Go, where it is now on a level with the best players of the world on smaller size boards. In October 2015, Google's AlphaGo which uses a Monte Carlo Tree Search based method run on knowledge learned from a deep learning network defeated Fan Hui, the European Go champion and a 2-dan Go professional, 5 games to nil. Dan is a rating system used for the top Go players, with a maximum rank of 9-dan. In March 2016, it went on to defeat Lee Sedol, a 9-dan player 4 games to one. [8]

2.2 Variations of Monte Carlo Tree Search

It is possible to modify and customize MCTS methods quite significantly. The tree policy and default policy can be replaced by a more informed and well constructed policy which can be based on any prior knowledge you may have. Work by many people such as Pellegrino and Drake[9] have investigated the performance of the heavy playouts of MCTS specifically applied to the game of Go.

Gelly and Silver[10] conducted tests on comparing basic randomized Monte Carlo Tree Search with hybrid techniques that involve game knowledge as applied to 9x9 Go. As one would expect, integrating domain knowledge to influence the tree policy lead to better results with lesser computation time using Monte Carlo Tree Search.

Parallelizing MCTS is also quite easy, and is done in a number of ways. A few of them include[7]:

1. Parallelizing from a certain leaf node. After the selection stage, multiple simulations are done over multiple threads and reported back to the main tree. This possibly leads to several duplications as wel as unnecessary exploration of nfruitful nodes, but is the easiest method to implement.

2. Parallelizing from the root. Independent game trees are constructed by the individual threads and combined at the end of all the simulations to get an overall result. Little to no communication is required and

therefore the threads can work more or less independent of each other.

3. Parallelizing the construction of the game tree itself. This involves the use of mutexes and other ways of thread synchronization to make sure the individual threads work on different sections of the tree. Lots of communication is required, so it somewhat reduces the speed of simulations and construction while increasing the chance of finding the most optimal solution. The "Fuego GO" program was modified by Enzenberger and Müller[11] to implement a lock-free method of tree parallelizing which would further improve the performance of the algorithm

2.3 An Overview of Advanced Tic-Tac-Toe

Advanced Tic-Tac-Toe is a humorous and more challenging alternative to the relatively simplistic game of Tic-Tac-Toe (which is known to always end in a draw when two well-informed players are playing). While a regular Tic-Tac-Toe game uses a 3x3 board, this game uses a 3x3 board which consists of 3x3 boards in each slot. While this may seem like a 9x9 board at first glance, each 3x3 board within the slot is independent by itself.

For the purpose of the explanation, I will call each individual position on the smaller board a square, and each individual Tic-Tac-Toe board a board.

At each turn, a player marks one of the squares.

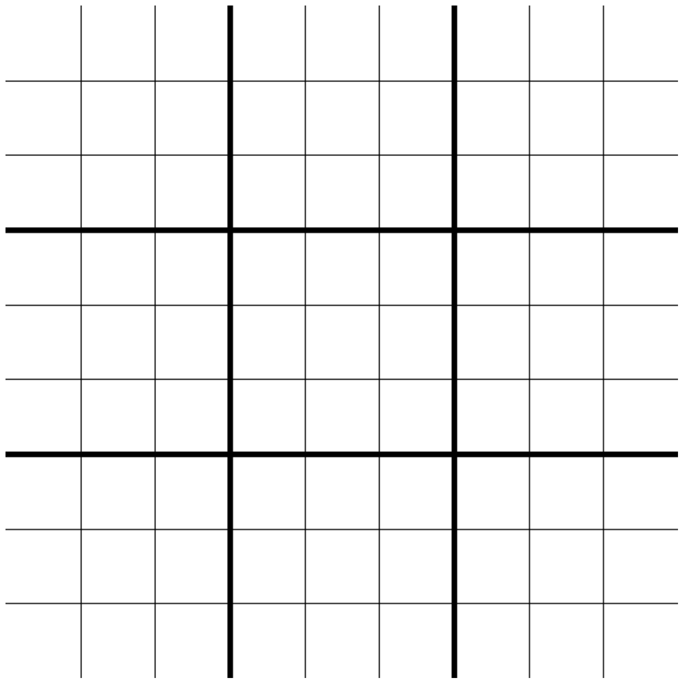

Figure 2.3: Figure of an Ultimate Tic-Tac-Toe Board

As with the regular rules of Tic-Tac-Toe, when a player achieves 3 squares in a row (vertical, horizontal and diagonal all count), he wins that particular board.

In our version, a player needs to win three of the boards in a row.

So far, it seems to be just a larger game of Tic-Tac-Toe where it takes longer to achieve a result. However, where the strategy comes in is in the next rule: a player cannot choose which board to play in. This is determined by the previous players move. The position of the square in which he plays determines the position of the board which you must play in. For example, if the previous player chose to play in the top right corner square of his board,

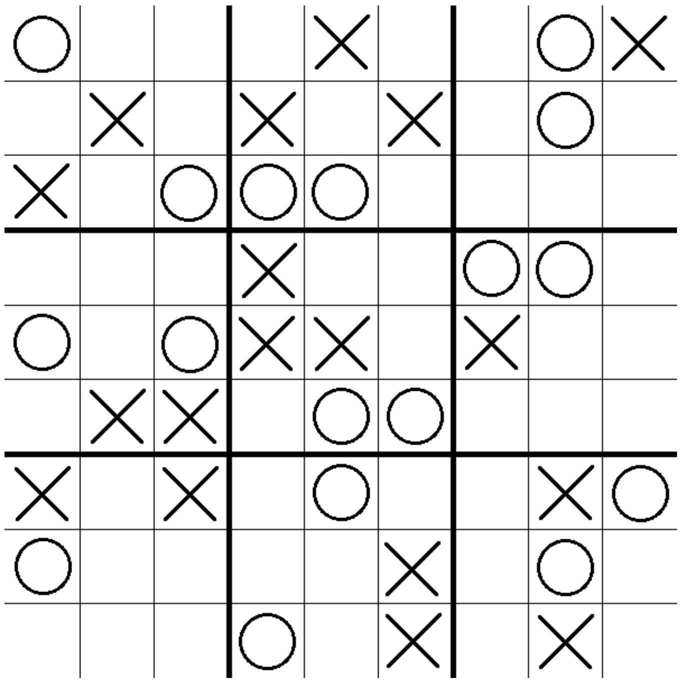
Figure 2.4: Square in an Ultimate Tic-Tac-Toe Board

then your next move must be made in a square of your choice in the top-right board only.

This adds an element of strategy and non-obvious solutions where you must plan ahead and not only try to win boards, but plan to send your opponent to different places in such a way that it benefits you in the longer run. This kind of a problem seems ideal for treatment by an AI agent using Monte Carlo methods as there is no particularly obvious heuristic that we can use.

A few other clarifying rules are used for the following scenarios:

1. What if one of the boards ends in a tie?

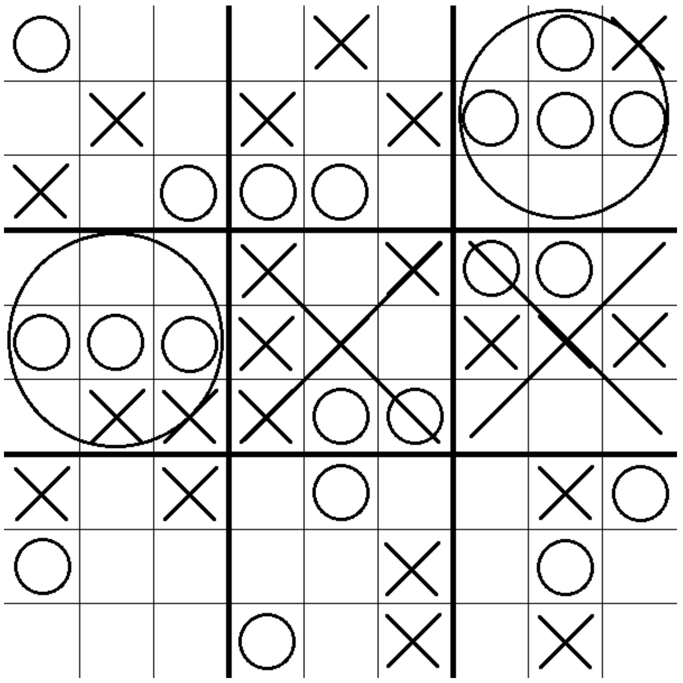

Figure 2.5: Winning a Board in an Ultimate Tic-Tac-Toe Board

As this is not an official game with rules laid out in stone yet, there is room for variants. We could consider a tied board as not counting toward either team. Or if we wanted we could say that it counts towards both. For the purpose of this thesis, I have counted a tied board as counting for both teams i.e. both sides can use this as one of the squares in their three-in-a-row.

2. What if the opponent sends me to play in a board that has already been won?

   The generally accepted rule in this case is that the player who has been sent to a finished board can choose to play in any board of his/her

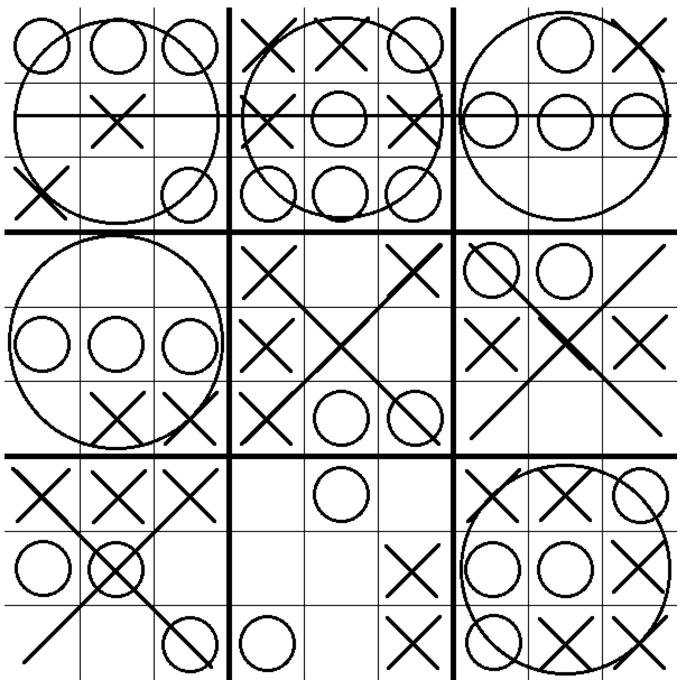

Figure 2.6: Winning a game of Ultimate Tic-Tac-Toe

choice. There is a more interesting variant in which the player who has won the board in question gets to choose the board that the next player chooses. So if player 1 ends up on a board that has already been won by player 2, player 2 gets to choose the next board which player 1 has to play on. In case of a drawn board, a coin flip can be done to determine which player gets to choose. However, we have chosen to stick to the general rule here.

In order to reduce the number of tied games, we are also using the rule that if at the end of the game, no clear winner can be determined, the one who controls the most boards is declared the winner.

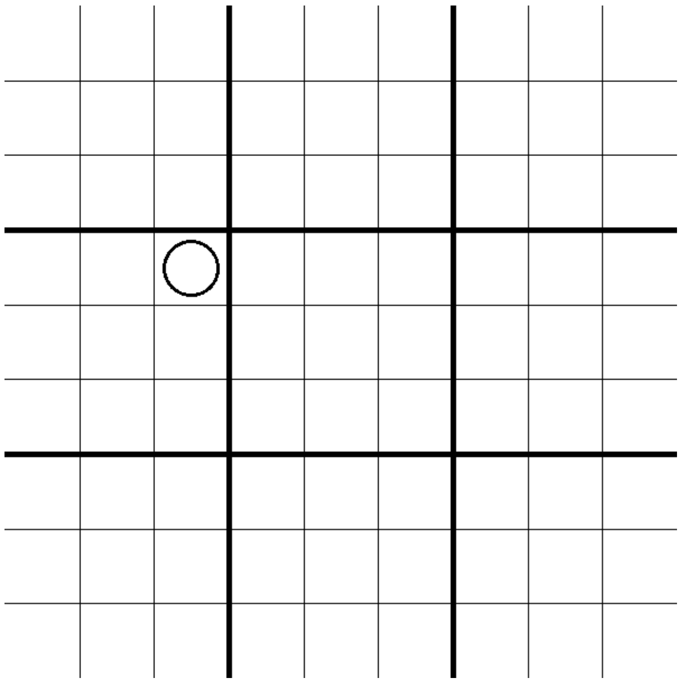

Figure 2.7: Playing In The Top Right Square

It's unknown exactly who is to be credited for inventing this variant of Tic-Tac-Toe, but it seems to have been popularized by an article in 2013 by Ben Orlin in his blog "Math With Bad Drawings". [12]

The properties of this game that make it suitable for our purposes is that it:

1. Has limited depth: Every game takes a maximum of 81 moves to reach completion

2. Has perfect information: Both players have the complete board state visible to them. Thus it is possible to compute not only your own optimal move, but also your opponent's.

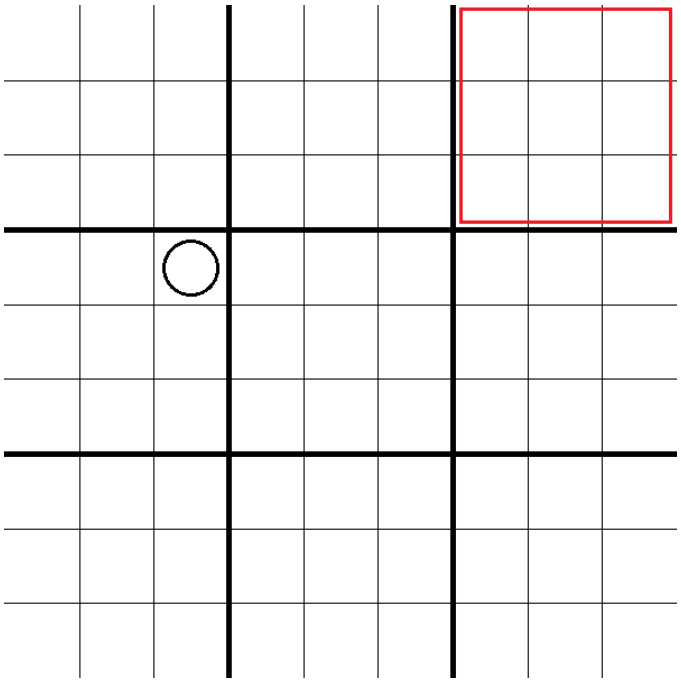
Figure 2.8: Playing In The Top Right Board

3. Is turn based: There is no real time decision making involved, you can react to your opponent's moves one by one.

4. No randomization: There is complete certainty in the moves we make, there is no dice rolling or card drawing to introduce random elements to the game.

This vastly reduces the amount of computation needed as a lot of permutations are cut down on.

Chapter 3

Implementation and Test Parameters

3.1 Our Implementation of MCTS

Monte Carlo Tree Search is a very versatile algorithm that can be implemented in different ways. There are so called heavy playouts which include an evaluation function to manipulate the tree policy to favour more optimal choices, as well as light playouts which rely on randomized moves. For the purpose of our thesis, I have used a light playout to see its effectiveness versus a method that does use an evaluation function (the Minimax approach). As such I will be using the basic formula proposed by Kocsis and Szepesvari. [4]

$$v = \frac{w_i}{n_i} + c\sqrt{\frac{\ln t}{n_i}} \tag{3.1}$$

Our algorithm tries to maximize the value of v, thus finding the node which gives us the optimal mix of exploration of the nodes to find new branches and exploitation of branches which we already know to be positive.

Here w_i is the number of wins after i iterations.
n_i is the total number of simulations aftr i iterations.
t total number of simulations for the node. It is equal to $\sum n_i$.
c is a constant which is used to balance exploration and exploitation. Its value is generally chosen empirically based on what suits one's needs. Theoretically its value was found to be $\sqrt{2}$ [4]

In this formula, the first component of the equation $\frac{w_i}{n_i}$ represents the exploitation component of the equation. It represents nodes with a high reward to visit ratio. The second component $\sqrt{\frac{\ln t}{n_i}}$ represents the exploration component. It is high when there are nodes that have a low number of visits.

It was also proven in this same study that given enough simulations, the error or false report rate (i.e. the chance of selecting a sub-optimal move from the available list) of the UCT algorithm falls to zero, thus proving that it eventually converges with that of the best possible Minimax algorithm.

3.2 Our Implementation of Minimax

The minimax algorithm for two player games relies on the framing of an evaluation function or heuristic which represents how well the state of the board benefits a player. The algorithm seeks to maximize the benefits of the player while trying also to minimize the corresponding heuristic of the opponent.

Here, more specifically, we use a depth-limited Minimax algorithm in order to have a measure of control over the amount of time that the algorithms takes per turn.

As for the evaluation function, the one we have chosen here is relatively simple. The primary objective is to try and get as many squares in a row as possible on the current grid. The secondary heuristic is to attempt to play in squares that will send your opponent to grid where you own more squares. The reasoning behind this is that it limits the number of moves your opponent can make and diminishes his ability to dictate the flow of the game. Obviously though, the priority remains winning the grid that is currently being played in. As such, our implmentation of the Minimax algorithm looks like this:

```
Data: State of the board $s$
Result: The optimal move to make
Function EvaluationSelf(s) is
    for each empty node $n$ in current grid $g$ do
        if $number(n_d)$ OR $number(n_r)$ OR $number(n_c) = 2$ then
            $value \leftarrow 10$
        else if $number(n_d)$ OR $number(n_r)$ OR $number(n_c) = 1$
         then
            $value \leftarrow 9$
        else
            $value \leftarrow gridStrength(n)$
    end
End

Function EvaluationOppo(s) is
    for each empty node $n$ in current grid $g$ do
        if $number(n_d)$ OR $number(n_r)$ OR $number(n_c) = 2$ then
            $value \leftarrow -10$
        else if $number(n_d)$ OR $number(n_r)$ OR $number(n_c) = 1$
         then
            $value \leftarrow -9$
        else
            $value \leftarrow -gridStrength(n)$
    end
End
```

Here $number(n_d)$ refers to the number of the player's own marks in the diagonal where n is located. Similarly, $number(n_c)$ refers to the number of marks in the column of n and $number(n_r)$ is the number of marks in the row where n is located. $gridStrength(n)$ returns the number of squares marked in the grid corresponding to the position of the node n

3.3 Parameters of Test

In order to fairly and properly test the two methods against each other, we needed to give a more or less equal amount of time to both methods. As such, I first measure the time taken by the Minimax approach on my machine at various different depth limits. At first, we limit the depth of the Minimax search tree to 3, then 4 and finally to 5.

I measured the average time taken by the Minimax by running it against a random player (by which I mean an agent which simply makes a random available move on the board), and finding the average amount of time per move (over a sample size of 100 moves).

Hence, we provide a mostly similar amount of time for MCTS which can of course be stopped at any time and so acts as the control for our experiment.

3.3.1 Machine Specs

We are running our test on a Amazon AWS EC2 Ubuntu Machine with a single core 2.5 GHz processor and 1 GB of RAM. Our search tree is depth limited so it does not occupy much space. If any bottleneck were to exist, it would be in our processor.

3.4 Initial Observations

Even before running any tests, we can make a few statements about the working and efficiency of our algorithms. Since our Minimax is essentially a depth-limited depth first search at its core, the time complexity would amount to $O(b^d)$. The space complexity would be $O(bd)$ where b is the branching factor of our tree and d is the depth reached. The space complexity for MCTS would remain $O(bd)$, however in practice this value would be higher than what is being used by our Minimax approach as there is no limit placed on the depth. Now, as we analyze each step of out algorithm, we can see that:

The number of iterations that the algorithm runs through (n) is quite different each time. It depends on a lot of constraints, such as the computational budget assigned (in our case, the amount of time given to the algorithm. This number is highly variable and cannot be reasonably calculated.

The expansion stage of the algorithm runs at a time complexity of approximately $O(b)$ where b is the branching factor, as the node has to be expanded into the child nodes of the given root node.

The simulation stage runs at the time complexity of $O(d)$ because the computation actually done in choosing the next node is a random move, and it is done in linear time, corresponding to the depth of the tree.

The back-propagation similarly occurs in linear time, updating the play-out status of all nodes leading up to the root node i.e. $O(d)$

Thus, the overall time complexity of the algorithm can be said to be $O(nbd)$

Chapter 4

Results and Observations

4.1 Test 1

For our first test, we limit the depth of the Minimax algorithm to 3. This gave us an average of 5.11 seconds taken per turn. As such, we time limit our Monte Carlo Tree Search to 5 seconds per turn.

The standard deviation of our data is about 1.77.

Doing this test for 500 games yielded the following results.

Table 4.1: Table of results for Test 1

	Number of wins	Win rate
Monte Carlo	285	57%
Minimax	202	40.4%
None	13	2.6%

As we can see, Monte Carlo Tree Search appears to have some advantage over a more shortsighted Minimax approach.

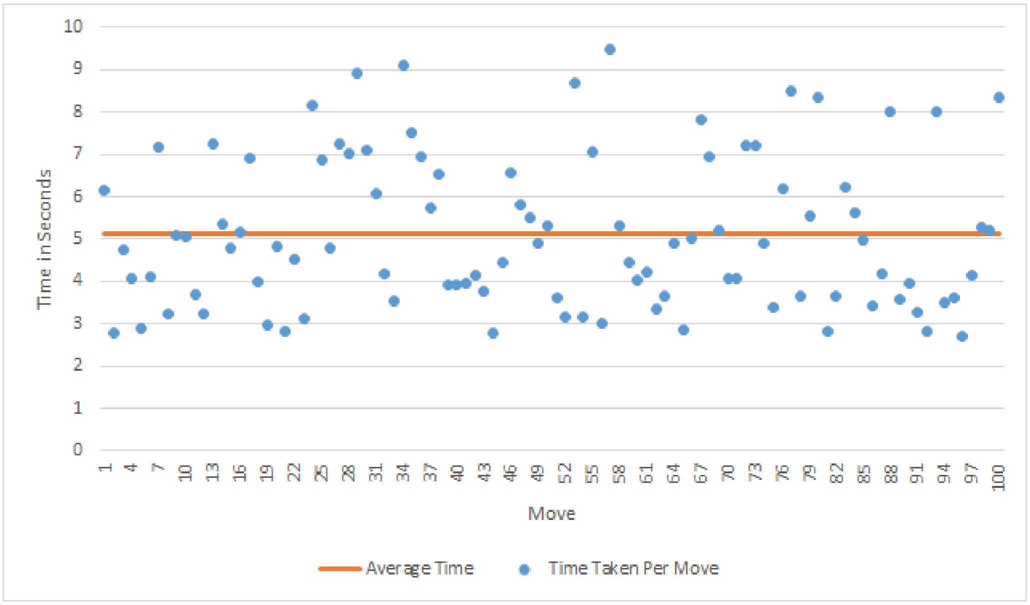

Figure 4.1: Scatter plot of the time taken per move and the average time taken for Test 1

4.2 Test 2

For the second test, we limited the depth of the Minimax algorithm to 4. This gave us an average of 16.55 seconds taken per turn. We limited the MCTS approach to 16 seconds per turn and got the following results

The standard deviation of our data is about 1.88.

Doing this test for 500 games yielded the following results.

Table 4.2: Table of results for Test 2

	Number of wins	Win rate
Monte Carlo	254	50.8%
Minimax	228	45.6%
None	18	3.6%

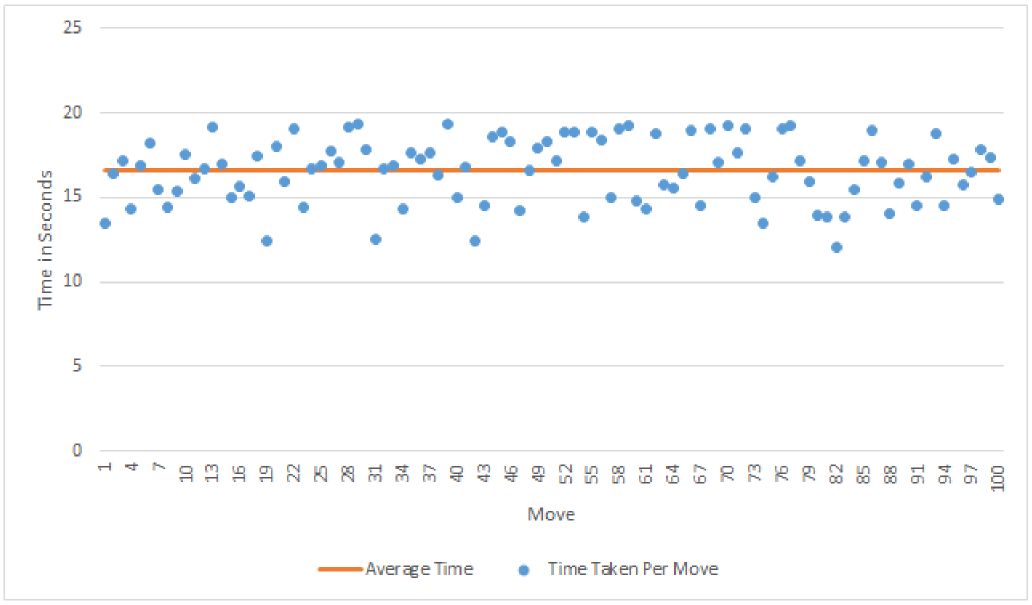

Figure 4.2: Scatter plot of the time taken per move and the average time taken for Test 2

4.3 Test 3

For the second test, we limited the depth of the Minimax algorithm to 5. This gave us an average of 28.25 seconds taken per turn. We limited the MCTS approach to 28 seconds per turn and got the following results. We only ran 100 games due to time constraints, as nearly 30 seconds per turn multiplied by around 50 turns per game game to about 25 minutes taken per game played.

The standard deviation of our data is about 2.61.

Doing this test for 100 games yielded the following results.

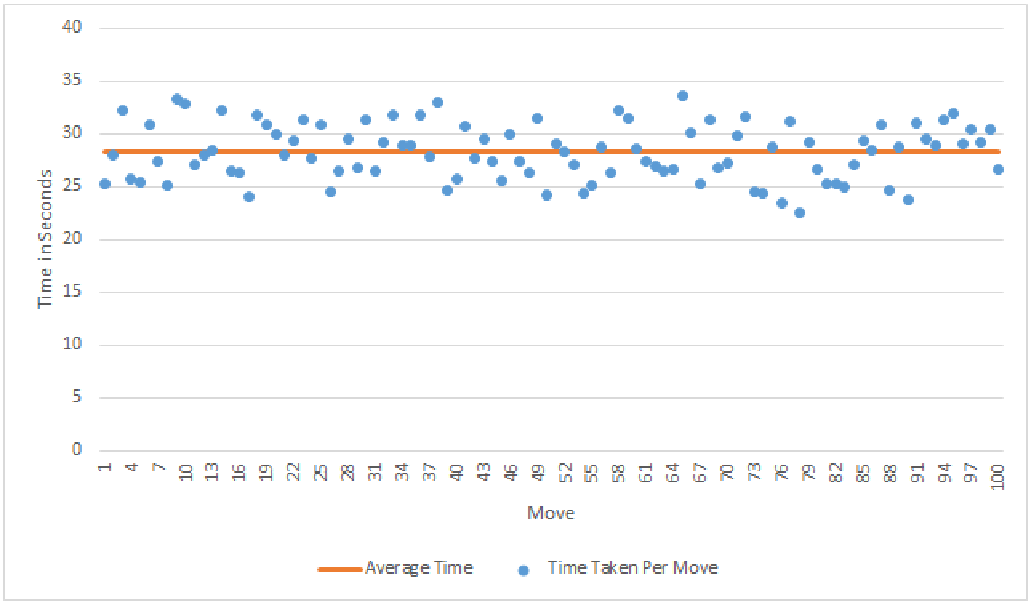

Figure 4.3: Scatter plot of the time taken per move and the average time taken

Table 4.3: Table of results for Test 1

	Number of wins	Win rate
Monte Carlo	50	50%
Minimax	47	47%
None	3	3%

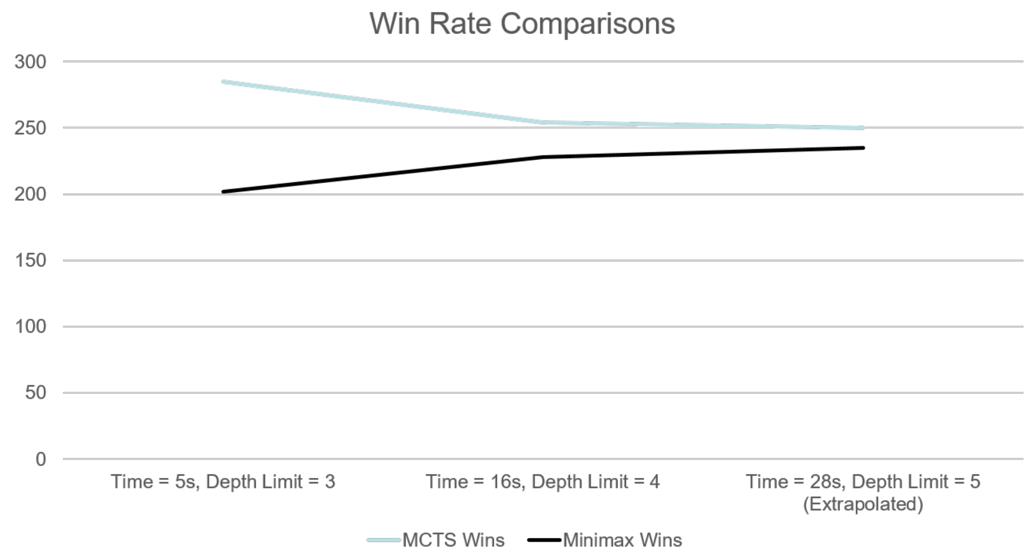

Figure 4.4: Plot Showing Convergence of Win Rates of Both Methods

Chapter 5

Conclusion and Future Work

5.1 Conclusion

In this experiment, we essentially pitted our MCTS AI agent against the one utilizing the Minimax approach, gave them both more or less an equal amount of time and

As it is in fact possible to further optimize both these approaches, it is not suitable to offer a definitive conclusion as to which approach would work better. Also, our experimentation does not weigh the advantages of certain more intangible characteristics of using MCTS such as not needing domain knowledge, ability to parallelize as well as any computational benefits that may arise depending on the structure of our tree.

However, as we can see from our results, Monte Carlo Tree Search performs quite well against our implementation of Minimax. Obviously the results may change quite a bit depending on how we frame our evaluation function for the Minimax, as well as using a different, more efficient tree policy for our MCTS approach. However, these experiments have given us a good idea at how effective MCTS can be, as even a randomized, unoptimized approach is able to do better than a reasonably well made Minimax AI.

We performed repeated tests for different levels of "difficulty" for our AI agents, and we also found that Minimax seemed to begin reaching the performance of our MCTS approach over time. This makes sense as it was found by (TODO REF) that ultimately, with enough simulations, the decision tree of MCTS converges upon that of Minimax even with random playouts.

Other results using MCTS have shown that while it may not necessarily be the most efficient approach to a problem, particularly when the problem is small enough (i.e. with a low branching factor so that brute force or a strong evaluation function can work better), it is versatile enough that it can used for most games without requiring any domain knowledge and also can itself be improved by the use of any evaluation functions that can be formulated.

5.2 Future Work

There are many improvements that can be made to our project.

1. Use of a better evaluation function for the Minimax approach

2. Use of a better default policy for our MCTS approach so as not to rely on random choice playouts (i.e. heavy playouts vs light playouts)

3. Parallelization of the MCTS approach. Using a multiple core machine and multithreading our approach would hugely improve the performance of our MCTS approach and possibly lead to better results.

4. Application of MCTS to other problems than board games. MCTS can be applied to great effect in field such as cryptography and security as it can be used as a tool to find flaws by repeatedly attempting to crack the existing security measures. It can also be applied to various other popular problems such as the Traveling Salesman Problem, Multi-Armed Bandit knapsack problem etc.

Appendices

Appendix A

The following is a code snippet showing how the board was represented in our python code.

```
matrix=[[[[0,0,0],[0,0,0],[0,0,0]] for i in xrange(3)]
    for i in xrange(3)]
main_matrix=[[0,0,0],[0,0,0],[0,0,0]]
```

The 0 represents an empty cell. 1 represents an 'X' and 2 represents an 'O'.

Beginning the UCT algorithm, and timing it appropriately:

```
def run_uct(self):
    sims = 0
    begin = time.time()
    while time.time() - begin < self.calculation_time:
        #calculation_time is a constant set by the user
          self.run_simulation()
    sims += 1
    ...
```

Determining the best move after receiving the data:

```
def calculate_action_values(self, board_state, player,
     available_moves):
         actions_board_states = ((p, self.board.
             next_board_state(board_state, p)) for p in
             available_moves)
         return sorted(
             ({'action': p,
                 'percent': 100 * self.stats[(player, S)].
                     value / self.stats[(player, S)].visits
                     ,
                 'wins': self.stats[(player, S)].value,
                 'plays': self.stats[(player, S)].visits}
              for p, S in actions_board_states),
             key=lambda x: (x['percent'], x['plays']),
             reverse=True)
```

Appendix B

Despite being based on the basic version of Tic-Tac-Toe, our version seems to have very little in common with its more simple parent. Due to the nature of the game, often you see very bizarre situations and strategies arise where there seem to be several easily claimed boards available to a player but they are unable to take advantage of this as their opponent does not allow them to play on these boards on their terms.

In my experience, most games of Ultimate Tic-Tac-Toe lasted around 20-30 minutes when played with my friends. As such, the benchmark of 30 seconds given to the MCTS approach seems to be the best one to use, although unfortunately due to time constraints we weren't able to run more than 100 games as a simulation on that benchmark.

During my simulations, I found that the number of turns taken seemed to vary quite heavily. It should also be noted that the relatively high variance in time taken by the minimax algorithm to return values to use can be explained by the fact that it depends entirely on the number of available moves to the algorithm at the time. If it was directed to play in a grid where there were only two available moves, the simulation would be completed much quicker than if there were five or more. On the other hand, out Monte Carlo approach would dutifully continue its simulations until the alloted time expired, thus strengthening its own results.

With its well publicized success in the field of Go, MCTS has risen to the fore as the algorithm of choice for attempting to solve a large variety of problems, including games that are completely different from the flagship Go, such as the popular card game Magic: The Gathering, and even in the field of video games, with an MCTS based approach being used for the AI in "Total War: Rome II"

Bibliography

[1] Maciej Świechowski and Jacek Mańdziuk. Self-adaptation of playing strategies in general game playing. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(4):367–381, 2014.

[2] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2012.

[3] Guillaume Chaslot, Sander Bakkes, Istvan Szita, and Pieter Spronck. Monte-carlo tree search: A new framework for game ai. In *AIIDE*, 2008.

[4] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *European conference on machine learning*, pages 282–293. Springer, 2006.

[5] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2-3):235–256, 2002.

[6] Arnaud Doucet, Nando De Freitas, and Neil Gordon. An introduction to sequential monte carlo methods. In *Sequential Monte Carlo methods in practice*, pages 3–14. Springer, 2001.

[7] Guillaume MJ-B Chaslot, Mark HM Winands, and H Jaap van Den Herik. Parallel monte-carlo tree search. In *International Conference on Computers and Games*, pages 60–71. Springer, 2008.

[8] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou,

Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[9] Seth Pellegrino and Peter Drake. Investigating the effects of playout strength in monte-carlo go. 2010.

[10] Sylvain Gelly and David Silver. Combining online and offline knowledge in uct. In *Proceedings of the 24th international conference on Machine learning*, pages 273–280. ACM, 2007.

[11] Markus Enzenberger and Martin Müller. A lock-free multithreaded monte-carlo tree search algorithm. In *Advances in Computer Games*, pages 14–20. Springer, 2009.

[12] Ben Orlin. Ultimate tic-tac-toe. `https://mathwithbaddrawings.com/2013/06/16/ultimate-tic-tac-toe/`. Accessed: 2016-11-04.