6-28-2017

Game Specific Approaches to Monte Carlo Tree Search for Dots and Boxes

Jared Prince
Western Kentucky University, jared.prince133@topper.wku.edu

GAME SPECIFIC APPROACHES TO MONTE CARLO TREE SEARCH FOR DOTS AND BOXES

A Capstone Project Presented in Partial Fulfillment

of the Requirements for the Degree Bachelor of Science

with Honors College Graduate Distinction at

Western Kentucky University

By

Jared A. Prince

May 2017

*****

CE/T Committee:

Dr. Uta Ziegler, Chair

Dr. James Gary

Siera Bramschreiber

ACKNOWLEDGEMENTS

I would first like to acknowledge my thesis advisor Dr. Uta Ziegler, without whose diligent support and expertise this project would not have been possible. She has been instrumental in both the inception and development of this project. I would also like to acknowledge my second reader, Dr. James Gary, and the Honors College staff who have facilitated this project. Finally, I would like to recognize my friends and family who have spent a great deal of time listening to me talk about this project and allowed me to brainstorm out loud, even when they had no clue what I was talking about.

ABSTRACT

In this project, a Monte Carlo tree search player was designed and implemented for the child's game dots and boxes, the computational burden of which has left traditional artificial intelligence approaches like minimax ineffective. Two potential improvements to this player were implemented using game-specific information about dots and boxes: the lack of information for decision-making provided by the net score and the inherent symmetry in many states. The results of these two approaches are presented, along with details about the design of the Monte Carlo tree search player. The first improvement, removing net score from the state information, was proven to be beneficial to both learning speed and memory requirements, while the second, accounting for symmetry in the state space, decreased memory requirements, but at the cost of learning speed.

Keywords: Monte Carlo tree search, dots and doxes, UCT, simulation, impartial games, artificial intelligence

VITA

February 16, 1995……Born – Indianapolis, Indiana

2009-2013…….Meade County High School, Brandenburg, Kentucky

2017……Presented at Annual WKU Student Research Conference

2017…….Fruit of the Loom Award for Exceptional Undergraduate Computer Science Major Recipient

FIELDS OF STUDY

Major Field 1: Computer Science

Major Field 2: Philosophy

CONTENTS

LIST OF FIGURES

LIST OF ALGORITHMS AND FORMULAS

LIST OF TABLES

1. INTRODUCTION

The goal of this project is to develop an artificial intelligence player for dots and boxes (a simple children's game) which improves upon standard methods. Dots and boxes has proven more difficult to work with than other simple games. Even games whose rules are much more complicated – chess, for instance – have seen great success with the standard methods, such as minimax and alpha-beta. However, these approaches have not worked well with dots and boxes due to the difficulty of evaluating a given board and the large number of possible moves.

To overcome these problems, a relatively new method of guiding gameplay is used: Monte Carlo tree search (MCTS). MCTS has recently been successful in Go players, which previously had been extremely weak. MCTS was used to overcome the inherent difficulties that arise in Go because of the massive number of possible moves and board configurations in a standard game [6]. Because the two games share the features which make other approaches unsuccessful and because of its success in Go, MCTS seems to be the ideal candidate for playing dots and boxes. This project applies MCTS to dots and boxes and offers two potential improvements to the standard MCTS.

New methods, algorithms, and strategies developed for use in simple environments such as games can often be translated for use in broader real-life fields. The advantage of testing such methods in a game setting is that one can evaluate the performance in closed systems with simple rules that are easier to write algorithms for and where the optimal result can usually be calculated using known methods for comparison.

The remainder of this thesis is organized as follows. First, an overview of dots and boxes (rules, strategies, etc.) is given. Then an overview of previous work in similar games. The next section outlines the theory and practice of Monte Carlo tree searches. Then an account is given of the implementation of Monte Carlo tree search used in this thesis and details of the approaches made to improve the implementation. The next section analyzes the results of these approaches. Finally, potential avenues of future work are explored and concluding remarks are offered.

2. DOTS AND BOXES

This section presents an overview of dots and boxes, including the rules, common strategies for playing the game, past work, and computational features. Dots and boxes is a two-player game in which players take turns drawing a line either vertically or horizontally to connect two dots in a grid. These lines are called edges, and the squares of the grid are boxes. The player who draws the fourth line of a box captures the box. When this happens, the player gains a point and must take another turn. If the line drawn is the fourth line for two connected squares, the player gets two points, but still gets only one extra turn. At the end of the game, the player with the most points wins. Figure 1 shows the moves of a simple 2x2 game, in which player one (blue) is the winner with three points.

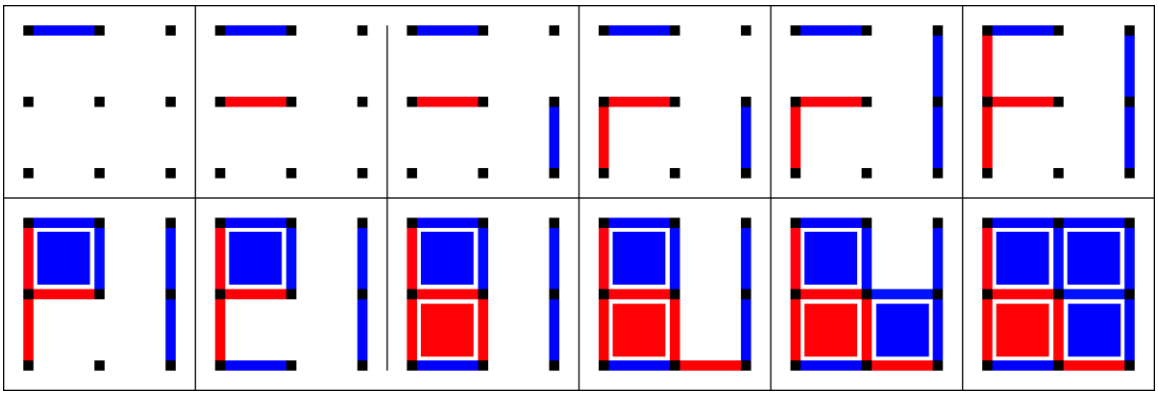

Figure 1: A simple 2x2 game with the first player as the winner

There are several things which make dots and boxes unusual. It is impartial, meaning that the current score and which players drew which lines does not affect the possible moves. In other words, given the same board configuration, either player could make exactly the same moves. It is also a zero-sum game, which means that the gain from a move for one player is exactly equal to the loss for the other player. Since there are a finite number of points available, each point one player gains is a point the other

player cannot have. It is fully observable, meaning that both players can see the entire game at all times (there is no information that only one player knows). It also has a set number of turns – though not a set number per player – equal to the number of edges on the board.

The most common strategies for playing dots and boxes involve taking boxes where possible and avoiding drawing a third edge on a box (which would allow the opponent to take it). These are not universally optimal rules – sometimes it is useful to sacrifice a box to an opponent or to avoid taking a sacrificed box – but they are generally valid. Because most players avoid drawing a third edge on a box whenever possible, most games consist of players drawing two lines per box until there is no choice but to draw a third line. This leads to a board which is filled with a series of chains (multiple boxes, each with two edges free, connected by these edges) and loops (chains whose ends connect back together). This type of board configuration is called a 'loony endgame' [3]. Figure 2 shows two example loony endgames. Notice that there are already some boxes taken in the first endgame.

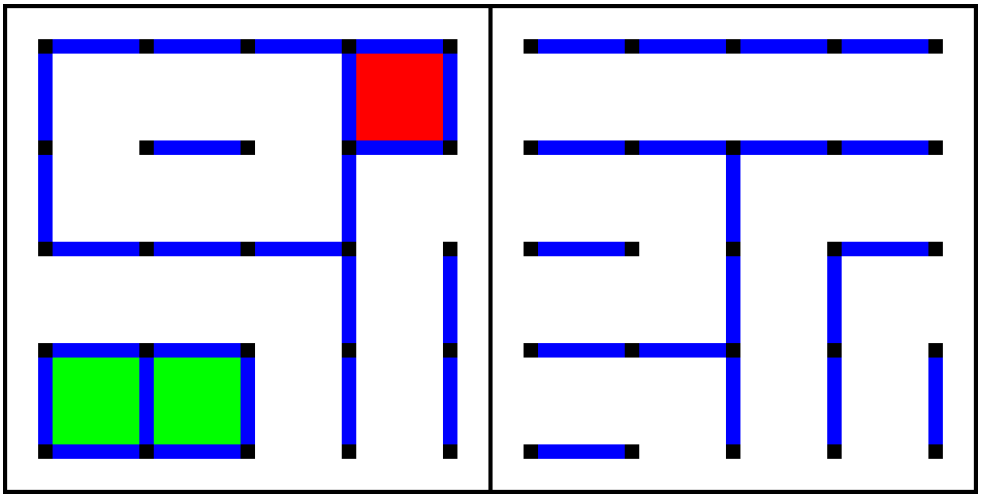

Figure 2: Two loony endgames

Once a loony endgame is reached, the first player (let's call the player A) is forced to draw the third edge of a box. This "opens" the chain or loop the box belongs to, allowing the next player (let's call that player B) to take all the boxes. Then player B is, in turn, forced to open the next chain. Player B, however, can avoid opening the next by sacrificing the last two boxes (for a chain) or four boxes (for a loop) and ending their turn. This forces player A to take the remaining boxes and open the next chain. By doing this, the player who started his turn with an opened chain can control the rest of the game, taking all but two (of four) of the boxes for each and taking all for the last chain (or loop). An important caveat of this approach is that when controlling a chain of less than four boxes or a loop of less than eight boxes, the player sacrifices more boxes than s/he gains. In such a case, the player might stay in control and yet lose the game. So it is sometimes better to take all the boxes in a chain – losing control, but gaining more points.

In optimal endgame play, any solitary boxes are evenly split between the two players because there is no opportunity for sacrifice. Likewise, chains of length two are split between the players because the player who opens the chain can prevent a sacrifice by taking the edge connecting the two boxes. Since both players can enforce the alternating order of two-chains and it is a zero-sum game, the order is enforced. If the player to open the chain does not want the opponent to sacrifice the chain, s/he can draw the opening line in the center of the two boxes. If s/he does want the opponent to sacrifice the chain, then the opponent does not want to sacrifice, so s/he does not sacrifice either way. Figure 3 shows the three possible 2-chain configurations. Red edges are the edges in each chain that the opener can draw to block a sacrifice.

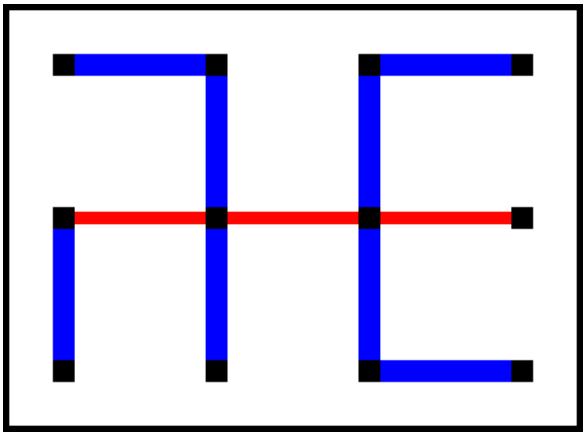

Figure 3: 3 different 2-chain configurations with sacrifice-blocking edges in red

The computational simplicity of a loony endgame makes it easy to determine optimal play. In fact, Buzzard and Ciere have developed efficient algorithms for determining the winner and the optimal move in a simple loony endgame, a variation of the loony endgame which has only chains or loops of three or more boxes [3]. If all boxes on the board have two or more edges, the optimal move can be easily calculated without requiring a search. In other words, it does not need to "look ahead" at the possible paths the game could take.

However, this does not mean that dots and boxes strategy is simple. In a simple loony endgame, for instance, the first player to 'open' a chain is usually the loser, because the other player can control the rest of the chains. Thus, it is important for a given player to engineer the endgame such that they are not the player to open a chain. Drawing a third line on a box before the endgame to allow the opponent to take a box early (and thus take an extra turn) can allow a player to be the player in control of the endgame. In a non-simple loony endgame, there can be chains of only three boxes, which can cause the controlling player to lose points. If the endgame is filled with 3-chains, then the first player to open a chain may still get more points than the opponent. So, while

optimal play in a loony endgame is (relatively) simple, there are complex strategies necessary to make these optimal moves result in a win for a given player.

A board's size is measured by the number of boxes per side. For instance, a 2x3 board has a height (in boxes) of 2 and a width of 3, for a total of 6 boxes. The number of edges in a board is given by ((height + 1) * width) + ((width + 1) * height), or 2 * width * (width + 1) for a square board. The boards can be of any size larger than 1x1, which is fully determined (neither player can change the outcome). Most games, however, are played on square boards.

While the rules of the game are simple, it has an extremely large number of board configurations of 2^p, where p is the number of edges. A naïve search of the game – which checks every possible sequence of moves that can be made – would be incredibly time consuming because there are $p!$ ($p * p\text{-}1 * p\text{-}2 \ldots * 1$) different possible games. Even for small games, there are a massive number of configurations, but an even more incredible number of distinct games, and increasing the board size grows those numbers exponentially. Square boards of 2, 3, or 4 squares per side have 12, 24, or 40 edges on the board, respectively. They have a total of 4096, 16 million, or 1 trillion configurations, with about 500 billion, $6 * 10^{23}$, or $8 * 10^{47}$ distinct games, respectively. This growth makes calculating optimal moves practically impossible for most boards. Even assuming every game becomes a loony endgame (which it does not) and the endgame begins at roughly the halfway point, the number of possible games is still massive. Figure 4 below shows two endgame scenarios which do not meet the criteria for a loony endgame. In both games, any move results in one or more boxes having a third line. However, each game contains an intersection of chains, which [3]'s algorithms do not take into account.

Endgames may have many such intersection, and until they are removed, they do not become loony endgames.

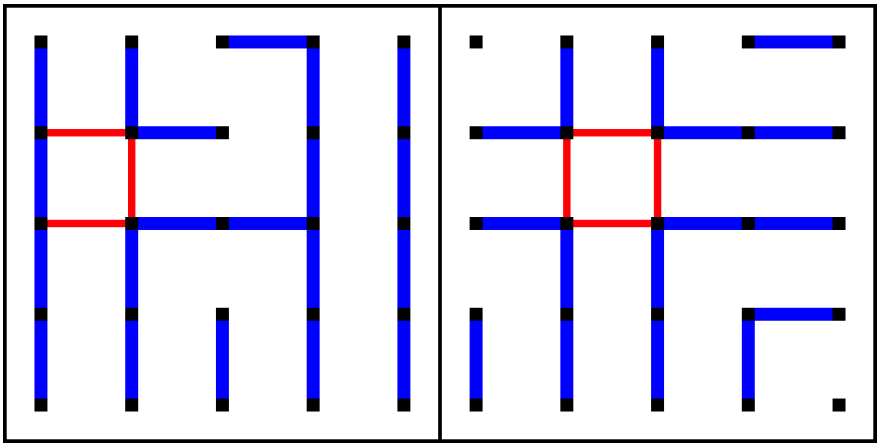

Figure 4: 3-way and 4-way intersections not covered in the loony endgame algorithm

Apart from developing programs to play the game (to understand how a computer can be told or can learn how to play a game well), the most common avenue of research into games such as dots and boxes is *solving*. Solving a game refers to conclusively determining the winner of the game in optimal play (possibly including the winning score), or deriving an algorithm for optimal play at every state. For complicated games, such as dots and boxes, this is often a computationally intensive process. To date, for dots and boxes, only boards up to 4x5 have been solved [1]. Table 1, below, shows the optimal result for the player which makes the first move for several different board sizes, as well as the computation time needed to solve them.

Board Size	Net Score for Player One	Computation Time
1x1	-1	0
2x2	2	.002 seconds
3x3	-3	54.8 seconds
4x4	0	3 hours
4x5	0	10 hours *130 on Wilson's solver

Table 1: Solved Games

Wilson's dots and boxes solver has been used to solve the game up to 4x4. This solution not only gives both the winner and the final score (in optimal play), but shows the optimal move at every state in the game [9]. The 4x5 board has been shown by [1] to be a tie in optimal play, but the computation took 10 hours, even with the authors' solver being as much as 175 times more efficient than the Wilson's solver. And although the prevalence of loony endgames often makes optimal endgame play straight forward to calculate or search for, exhaustive searches in the early to mid-game are incredibly time consuming.

3. PREVIOUS ARTIFICIAL INTELLIGENCE APPROACHES

This section presents a short explanation of some common artificial intelligence game playing techniques, including their limitations for use in dots and boxes. For a computer, playing the game consists of determining, from the selection of legal moves, which move to play at a given point in the game. To determine this, the computer builds a game tree, which consists of a series of 'nodes' connected by 'edges'. Each node represents a single state (a description of a possible point in the game). The edge connecting two nodes represents the action that is taken to get from the state of the first node to the state of the second node. The set of all possible states the game could reach is the state space. A complete game tree has nodes representing every state in the state space with some states appearing more than once. The root (the first, 'topmost', node) of the complete game tree is the node representing the starting configuration of the game. If any node A is connected to a later node B by a potential action, A is a parent of B and B is a child of A. Every node in the tree has an edge (connecting it to a child node) for every possible action that is legal in the game configuration represented by the node. There is a unique sequence of actions leading from the root node to each leaf node in the game tree, representing a distinct game. Figure 5 shows a simple example of a tree.

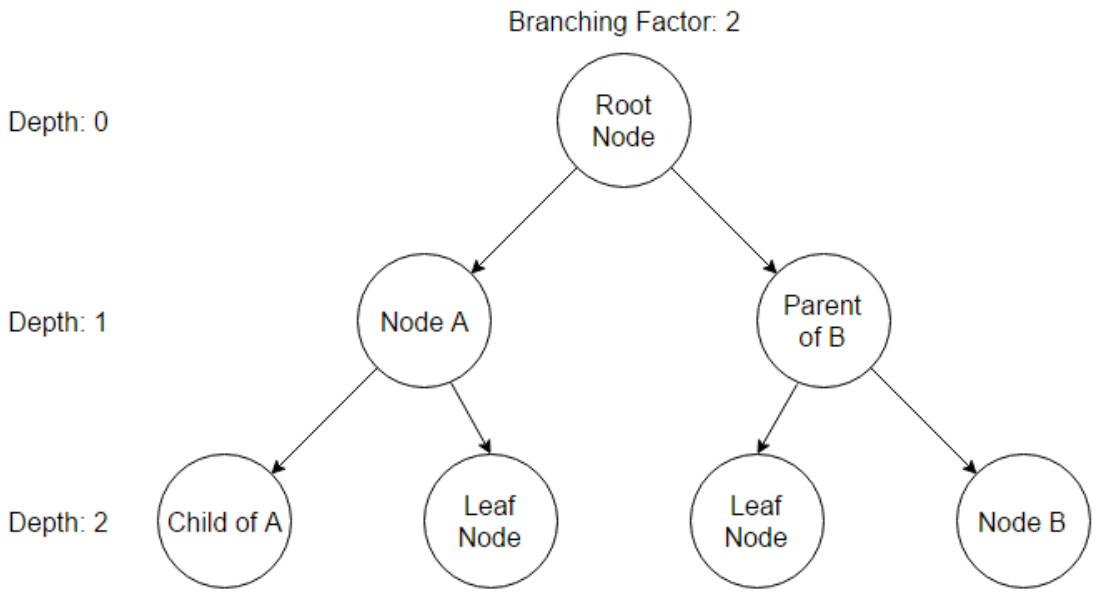

Figure 5: Tree Example

The most common artificial intelligence algorithm for a computer to participate in two-player combinatorial games is called ‘minimax’. It works by looking ahead from the current game configuration a certain number of moves to see all the possible states the game could reach, building a game tree that is complete to a certain depth (the number of moves made). To compare how desirable various states in this semi-complete game tree are, an ‘evaluation’ function is used. An evaluation function is a heuristic – that is, game-specific information which is used to guide searches such that an approximate solution can be found to a problem when an optimal solution is too difficult or too time-consuming to compute. The evaluation function is used to estimate how ‘good’ the state in each leaf node of the semi-complete game tree is for the player. Minimax then uses these estimates to work backwards and determine the best move for the player.

Because it is a zero-sum game, the advantage to player A is a disadvantage to player B. The best move is determined by maximizing the advantage at each move in which it is player A’s turn – selecting the move with the highest advantage – and

minimizing the advantage on turns that belong to player B – selecting the move with the lowest advantage [2]. In other words, minimax works backwards by determining the optimal move at each state for the player in control and assuming every turn is played optimally. It determines the best advantage player A can force player B to accept. A common improvement on the minimax algorithm is the addition of Alpha-Beta pruning. The goal of alpha-beta pruning is to decrease the number of branches the search must check by using information already learned in the search. When it proves a given move is worse than a previous move, it stops evaluating that move. For instance, if one of the available moves on a maximizing level leads to a score of 5, the highest score that is known to be possible is 5. If the first child of the next move results in a score of 2, the algorithm discards the move because the minimizing level selects a move which leads to a score of less than or equal to 2 [8]. The minimax algorithm with alpha-beta pruning was used to great success by the chess playing program, Deep Blue, which beat chess world champion Garry Kasparov in 1997 [4].

For minimax and alpha-beta pruning to work well for a game, the computer must be able to look ahead enough moves to direct the play in a valuable way, and it must be able to meaningfully evaluate the relative advantage of a given state. The algorithm's maximum depth is dependent upon the time given and the branching factor (the number of states that can be reached in a single move from an average state) of the tree. The larger the branching factor, the more time is required to reach a certain depth. The evaluation function must be an adequate assessment (generally) of which player has the advantage and by how much. Without these capabilities, minimax or alpha-beta pruning is not able to form a high-level player for the game.

In dots and boxes, however, the branching factor is so large that the number of moves that can be predicted is small. Moreover, there is a lack of any clear method of evaluating the advantage of a position apart from score. However, score is not always indicative of who is winning, since players often take many boxes in a row. With the no-third-edge strategy, players do not take any boxes until the endgame, so until that point there is nothing to compare between the players and it is difficult to evaluate who is "ahead". Even if a player sacrifices a box (by mistake or design), the resulting bump in score likely remains constant for many moves, so no move stands out as being better than any other. Lacking both the ability to look ahead many moves and the ability to accurately evaluate who has the advantage, the minimax cannot find purchase in any but the smallest dots and boxes games.

The problems described above for using well-established artificial intelligence approaches for dots and boxes were also encountered in the game Go. The large number of possible moves per turn and possible board configurations, as well as the number of average turns per game made traditional methods like minimax unfeasible, and it is difficult to accurately evaluate the value of a given state. Traditional methods of searching have produced Go players that play at only a beginner level [6]. In the last ten years, however, great strides have been made in Go by applying the idea of a Monte Carlo tree search to guide the search through the game tree [7]. Monte Carlo methods use random or pseudo-random sampling of the search space to calculate statistically reasonable approximations for deterministic problems that are difficult to calculate exactly using other methods. The more samples used, the more accurate the results [2].

The authors of [10] even use Monte Carlo tree search combined with artificial neural networks to play dots and boxes. In their implementation, the MCTS used as its selection policy an artificial neural network (ANN). The ANN was trained to predict the result of a game, give a board configuration and net score. Once the ANN learned to predict the outcome, it was given the possible board configurations after each move and its prediction was used to select the ‘best’ action. Their program, QDab, which used this MCTS player, performed better in tests than players using minimax [10].

4. MONTE CARLO TREE SEARCH

4.1 THEORY

The Monte Carlo tree search (MCTS) is one common approach to problems with such a large search space that a normal search, such as minimax, is ineffective. MCTS is a variation of Monte Carlo methods, which attempt to approximate solutions that are extremely difficult to calculate by collecting data from many simulated samples and averaging the result over the number of simulations [6].

One valuable aspect of MCTS is that it requires little to no domain knowledge. In other words, the only information it requires is how the state changes when a particular move is made. It does not need to know any of the strategy involved in the game; a random default policy is effective in directing the growth of the tree to advantageous branches. The search also does not need to be able to evaluate intermediate moves in the game. Because the simulations are played until the end of the game and it is the result of the game which is used to update the values of previous states, it only needs to be able to evaluate the winner at the end of the game [2].

The MTCS is a tree search which builds a portion of the game tree using simulated games. Each node on the tree contains three pieces of information: the number of times that node was reached (N(s), where s is the state), the number of times an action was chosen (N(s, a), where a is the action), and the total reward for each action from every simulation in which that action was picked (W(s, a)). The use of s (or state) in the MCTS formulas and algorithms refers to the node on the tree which represents that state. In reality, the node is what is located on the tree, and a state is only one piece of

information contained in the node. The reward for a single simulation may be binary values simply denoting a win/loss, or more specific and varied values denoting the margin of victory/defeat. From the number of times an action *a* was chosen in state *s* and the total reward from choosing this action the value Q(*s*, *a*) = W(*s*, *a*) / N(*s*, *a*) is computed, which represents the average reward from choosing action *a* in state *s*.

A MCTS game has two parts: gameplay and simulation. The gameplay is the actual series of moves that the players make – the game itself. At each of its moves, however, the MCTS player performs a certain number of simulations, starting from the current state of the game. There may be tens of thousands or hundreds of thousands of simulations per move. Each simulation is used to update the Q(*s*, *a*) value of the states and actions which were used during the simulation and which are represented in the tree. Because the values of a node are updated, each simulation increases the information available to the next. Thus, each successive simulation performs better (in theory). After all the simulations for a move are finished, the MCTS player chooses a move based on the updated values [7]. Algorithm 1 shows the process by which a MCTS game is played. In the algorithm, the notation $\max_a(Q(s, a))$ is used to denote the action that leads to the highest value of Q(*s*, *a*) for a given state s.

```
play_game:
  state ← initial_state

  while state is not a terminal_state
    if MCTS_player_turn
      simulate(state)
      action ← max_a(Q(state, a))

    else
      action ← opponent's move

    make move action
    state ← perform action in state
  end while
end
```

Algorithm 1: A Monte Carlo tree search game

4.2 SIMULATION

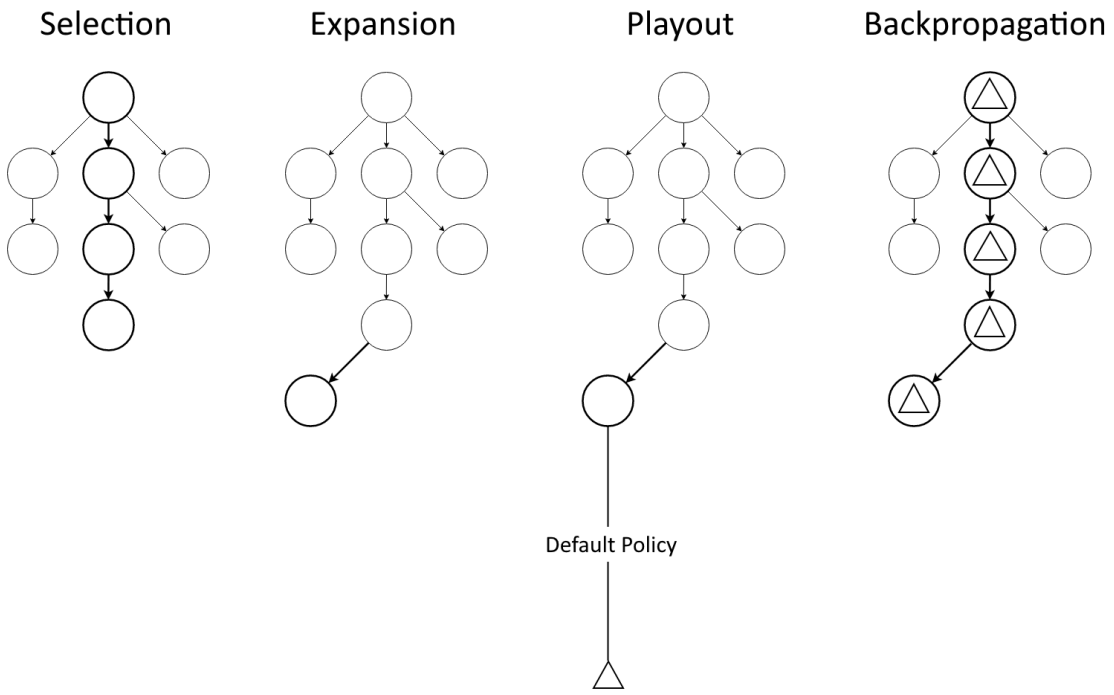

Figure 6: A Monte Carlo tree search simulation

Each simulation has four stages: selection, expansion, playout, and backpropagation. These stages are shown in Figure 6, with Δ representing the reward for the simulation. In the **selection stage**, the existing tree is traversed using a selection policy at each node *s*. Many different selection policies exist, the most intuitive of which are to select the action which has been selected the most ($a = \max_a N(s, a)$), or the action with the best average reward ($a = \max_a Q(s, a)$). The simplest policy usually chooses the best action based on the average result from making a given move at the current state ($a = \max_a Q(s, a)$) [7].

The **expansion stage** occurs when an action is selected which does not have a coinciding node on the tree. This is the stage in which new nodes are added to the tree. There is typically not much variation in this stage. The only usual variation is whether the

new node is always added to the tree or whether it is only added after the N(*s*, *a*) of the action that leads to it reaches a certain number. Another simple variation is to only expand nodes whose average reward Q(*s*, *a*) is greater than or equal to a certain value. When the expansion condition is met, a new node is created – representing the state of the game after the selected action – and added to the tree. After the expansion stage, playout begins [7].

In the **playout stage**, the simulation is continued until a terminal state is reached using a default policy. Default policies range from selecting random valid actions to selecting actions based on game specific strategy. Finally, when the game is finished, the values for each node in the selection stage are updated based on the result of the game. This is the **backpropagation stage** [7]. A variety of different approaches are possible for the backpropagation stage – from considering all simulations equal (formulas given above are for this case) to giving more weight to later simulations, since they have better information to make well-founded action selections. In a zero-sum game, if player A wins, for each node traversed in the selection stage in which it was player A's turn, the average value of the move player A made is increased and for the other nodes, the value of the move made by player B is decreased. Algorithm 2 shown below is the MCTS algorithm for simulating a single game during the MCTS player's turn. The simulate function runs the individual simulations, and the simulate_default function is used to finish a simulation in the playout stage.

```
simulate (state) :
  states_used ← empty list

  (*selection*)
  while state is not a terminal_state and state is on the tree
    action ← selection_policy(state)
    state ←
    add state to states_used
```

```
  end while

  add state to tree  (*expansion*)

  reward ← simulate_default(state)  (*playout*)

  (*backpropagation*)
  for each state in states_used
    N(state) ← N(state) + 1
    N(state, action) ← N(state, action) + 1
    W(state, action) ← W(state, action) + reward
  end for
end

(*playout*)
simulate_default (state) :
  while state is not a terminal_state
    action ← a random valid action
    state ← game after action is played
  end while

  return final reward of state
end
```

Algorithm 2: A Monte Carlo tree search simulation

4.3 UPPER CONFIDENCE BOUND FOR TREES

As described in the previous section, MCTS uses information from prior simulations to guide future simulations. This is too narrow an approach since it does not allow for MCTS to discover other – potentially better – strategies. A principle concern in MCTS is balancing exploitation (using the values and the portion of the tree already explored to bias the policy) with exploration (biasing the policy towards branches which have been explored less). Such a balance improves the estimates and expands the tree, which is necessary (to a point) in order to locate valuable paths.

A common implementation of MCTS which adds a bias towards exploration is Upper Confidence bound for Trees (UCT). UCT is a combination of standard MCTS and an upper confidence bound, which is an estimate of the maximum true reward, given the relative confidence of the current value estimate Q(*s*, *a*). Instead of using a policy during the selection stage which always chooses the action *a* in state *s* which has the best

estimated result Q(*s*, *a*), each estimate is given an uncertainty bonus, which ensures that exploration takes place. The larger the N(*s*) and N(*s*, *a*) of an action on a node, the more accurate Q(*s*, *a*) is. The smaller N(*s*) and N(*s*, *a*) are, the less accurate Q(*s*, *a*) is, and the more the uncertainty bonus is needed to ensure exploration. Thus, the value of an action is the average past reward of the action plus the uncertainty bonus (Q*(*s*, *a*) = Q(*s*, *a*) + U(*s*, *a*)) [7].

Here is an example which illustrates the problem with using only exploitation. Imagine a tree whose results are -1 for a loss and +1 for a win. In a simple greedy approach, if for a new node H with two possible actions *x* and *y*, MCTS selects *x* and loses, the expected value of that action is -1. If action *y* is chosen in state H in some later simulation and it results in a win, the estimate for y becomes +1. From then on, in state H action *x* is never chosen – even if every other simulation that picks action *y* in state H results in a loss – because the estimated value for action *y* in state H will always be slightly higher than -1.

Standard MCTS, explained in sections 5.1 and 5.2, leads to a tree in which valuable branches are left unexplored and in which the Q(*s*, *a*) of many actions are very inaccurate. The uncertainty bonus of UCT solves this problem by optimistically boosting the estimate of an action *a* in state *s* based on how many times it was chosen in state *s*. Formula 1 shows the uncertainty bonus for a given state, action pair, where *c* is a tuning constant greater than 0 (if *c* is zero the bonus is always equal to zero) [6].

$$U(s,a) = c * \sqrt{\frac{logN(s))}{N(s,a)}}$$

Formula 1: The uncertainty bonus of state s and action a

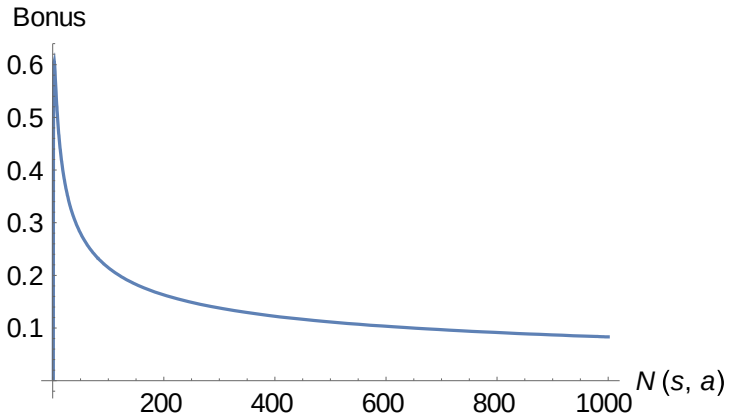

Figure 7: Graph of the uncertainty bonus of an action chosen x times

Figure 7, above, shows the uncertainty bonus applied to an action with N(*s*, *a*) and N(*s*) equal to *x* when *c* is equal to 1. As illustrated, the bonus applied decreases quickly as the number of times the action was chosen increases. This makes sense because as the number of times an action was chosen increases, its estimated reward becomes more certain. Thus, the bonus that can be charitably applied to it decreases. The more certain the estimate of an action is, the less its true value is likely to deviate from that estimate.

The upper confidence bound is an estimate of the maximum value the true reward could be, given the information already known and how much information that is; it represents the upward margin of error for the estimated reward of action *a* in state *s*. Using a greedy policy – which always chooses the option that has the highest value – with the upper confidence bound achieves a balance between exploration and exploitation. The UCT algorithm is consistent, meaning that (given enough time), it converges on optimal play. With enough simulations, all the value estimates for each node and action are guaranteed to approach the optimal reward [6].

5. APPLYING MONTE CARLO TREE SEARCH TO DOTS AND BOXES

The artificial intelligence player developed for this thesis applies the MCTS approach to the game dots and boxes. This section details the implementation of MCTS used in the current study. During the design of the player several issues needed to be addressed, including a representation of the board which conserves space and can be efficiently used to determine whether a box is taken, representing the game tree as a graph, and efficiently searching for a particular node in the tree.

Every node on the complete game tree represents a single unique state in the game, and every unique state appears on the tree – often more than once, since the same state can be reached with different sequences of actions. MCTS tries to only build a (relatively) small portion of the game tree. One aspect of the state is the board configuration – the layout of edges already draw by one of the players on the board. But this is not enough to distinguish between all possible states in the game. Imagine the final configuration of the game, in which all edges are filled, and thus all boxes are taken. Is that enough to determine the winner? No. The final board can represent a win or a loss for either player (or a draw on boards with an even number of boxes).

What is needed to separate these cases? The layout of which player finished which box would certainly work. But as already determined, which player took which box does not affect the game. The only thing that matters is how many boxes each player took. So perhaps the score for each player is all that is needed. But again, from knowing how many boxes were taken (which is given by the configuration) and how many boxes one player took, one can determine how many the other took as well (remember this is a zero-sum game). So, it seems all one needs is the score for a single player. For

simplicity's sake, one can use the net score – the number of boxes taken by one player minus the number taken by the other – for the player whose turn it is, which serves the same function but does not require the extra analysis of determining the number of boxes taken.

In this implementation, a single dots and boxes board configuration is represented as an integer. In binary form, each digit of the integer refers to an edge of the board. Zeros are edges that have not been drawn and ones are edges that have been drawn. For a given board configuration, the actions that can be taken are represented by the zeros in the binary representation. Figure 8 shows how the edges of the board are numbered. The edges are numbered from 0 to edges - 1, starting at the top left edge and increasing from left to right then top to bottom. When a move is made – that is, when another edge is added – the next board configuration is determined by turning the zero representing the taken edge to a one. To do so, the bit of the integer which represents the edge taken is 'flipped' to a one. To determine if a given box is taken when a move is made, each edge of the box is checked. For box *b* (numbered the same as the edges), Formula 2 gives the equations for the corresponding edges of *b*. It is important to note that in the formula for e_0, the division of *b* and width is rounded down to the nearest integer. This formula works for any rectangular board configuration.

$$
\begin{aligned}
e_0 &= ((\frac{b}{width} * ((2 * width) + 1)) + (b \ \ mod \ \ width)) \\
e_1 &= e_0 + width \\
e_2 &= e_1 + 1 \\
e_3 &= e_2 + width
\end{aligned}
$$

Formula 2: The edges which compose a box b

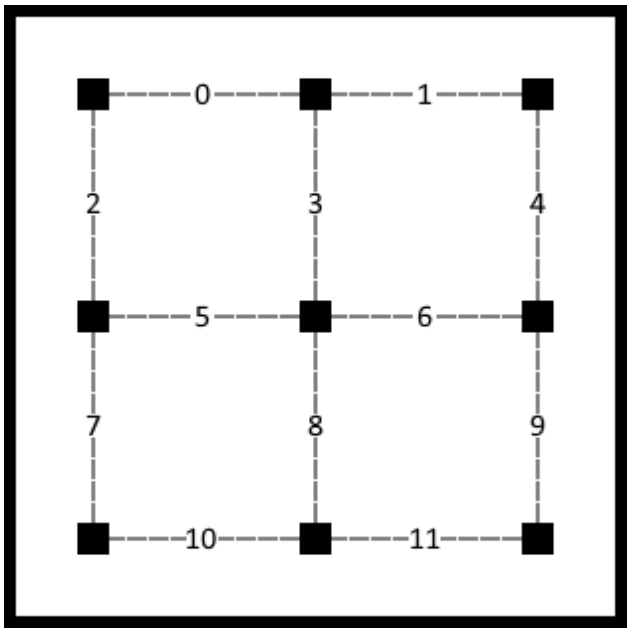

Figure 8: The edge numbering of a 2x2 board

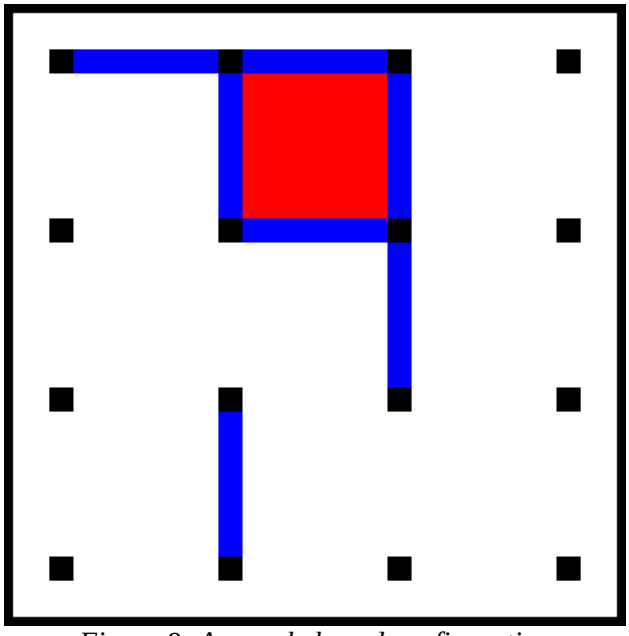

Figure 9: A sample board configuration

Figure 9 shows a sample board configuration. This board is represented in binary form as '1100110010001000000100000', and in integer form as 13,404,192. Pick a box to check, say box one. When 1 is plugged into Formula 2 as *b* and 3 is used as the width, the edges are 1 (e_0), 4 (e_1), 5 (e_2), and 8 (e_3). Because each of these digits in the binary

representation is 1 (starting at 0 and reading left to right), the formula tells us that box 1 has been taken.

Because in dots and boxes each state may occur in many different games (the order in which the edges were taken does not matter if the net score is the same), there may be multiple paths from the root node of the game tree to the node representing that state. This thesis uses a representation of the state space which avoids multiple copies of the same state and is called an 'acyclic directed graph'. Additionally, one can only add edges, never remove edges already drawn. An acyclic directed graph is a graph in which a node may have multiple parents, in which every connection between nodes is one-way. No state can be reached twice in the same path (starting at the root). A node *w* has a child for each edge that remains undrawn and a parent for each edge that is drawn. Each parent of *w* has one less edge, and is connected to *w* by the action of drawing that edge. Each child of *w* is the result of drawing one of the remaining edges. Thus, each node has a total of *n* parents and children (where *n* is the number of total edges on the board). A node at depth *d* has *d* parents and *n* - *d* children. Any node with *n* edges is at depth *n* in the tree (*d* = *n*). Because one can never remove edges that are drawn, a node cannot be the parent of a node that is at a higher depth. More specifically, because each parent of *w* has one less drawn edge and each child has one more drawn edge, parents and children of *w* must be at a depth of one less than *w* and one greater than *w*, respectively.

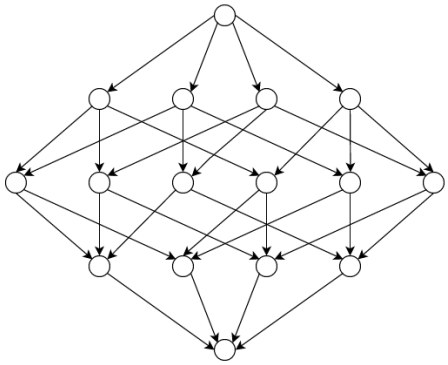

Figure 10: Complete tree of the 1x1 board

Figure 10 shows the directed acyclic graph of the 1x1 board (for simplicity). Notice that it has a defined diamond shape because up to the midpoint each level has successively more nodes and afterwards successively less nodes. There is an equal number of nodes at corresponding depths ($d_0 = d_4$, $d_1 = d_3$ for the 1x1 board). Notice also that the lower half of the graph is symmetrical with the upper half, when reflected vertically; the lower half and the upper half are the same when one is reflected both horizontally and vertically. Although the game tree is represented as a graph, for simplicity, the rest of this thesis will continue to refer to it as a tree.

Because the game tree is represented as an acyclic directed graph, a new node to be added in the expansion stage of the simulation may already be in the tree in another location. To avoid adding the same node twice, the tree must be checked to determine if the appropriate node already exists. In order to keep from having to search the entire tree, a hash table is used to store the references to all nodes on the tree. Before a node is added to the tree as a child of node *w*, the hash table is checked. If the corresponding node already exists in the hash table, then the old node is used as the child of *w* instead. In other words, a connection is made between *w* and the old node, rather than creating a new node and making a connection between *it* and *w*.

The realization of MCTS for this thesis implements the following for the four stages of each simulation: UCT is used for the selection policy, expansion occurs the first time a node not at the edge of a tree is traversed, and playout uses a policy of random moves. For backpropagation, all simulations are treated equally. The rewards given for a game are -1 for a loss, 0 for a draw, and +1 for a win.

Because dots and boxes is impartial, one tree can be used to make moves for either player. Given a board configuration and a net score (for the player in charge), which player is in charge is irrelevant. The strategy for both players (given the same net score) is the same. Therefore, moves that were made by the opponent can inform the decisions of the MCTS player when it reaches the same state in a later game.

6. POTENTIAL IMPROVEMENTS

This section explains methods implemented in this thesis to improving MCTS. Although they all share a goal of improving the performance of the search, improvements to Monte Carlo tree searches come in a few standard types. Because MCTS has been shown to converge on optimal play, the only limitations are the time and space required. The first type of improvements are enhancements to the simulation algorithms (selection and default policies, backpropagation methods, etc.). The main goal of these improvements is to decrease the time needed to learn optimal play by increasing the power of each simulation (and thus decreasing the number of simulations needed). Other methods attempt to either speed up the computation time for each simulation or increase the algorithm's ability to select valuable paths [2].

One method of speeding up MCTS is to parallelize the simulations across multiple processes or multiple machines. This is possible because multiple simulations can be run simultaneously on different threads and can be combined later. It is not a perfect method, however, because each simulation informs the next, so separating the simulations loses a bit of the power. The authors of [5] were able to achieve a speedup of 14.9 for 16 processing threads.

The following improvements are of another type: methods aimed at refining the complete game tree itself. They use information and methods specific to dots and boxes (or games like it) to decrease the search space of the game. These improvements are generally focused on improving the accuracy of the value estimate Q(s, a) of actions more quickly and decreasing the number of nodes MCTS adds to the tree (without negatively impacting the accuracy of value estimates over time). For the problems in

which they are applicable, these improvements can be very powerful. The use of a graph to represent the game tree is the first method of decreasing the search space.

This chapter addresses two improvements with the goal of reducing the search space of the game. The first is the use of unscored states, sacrificing a small amount of information to reduce the number of states to be represented in the game tree. The second is to account for symmetrical board configurations and consider these the same state. This sacrifices no game information and still reduces the size of the game tree.

6.1 UNSCORED STATES

The first improvement implemented is the use of unscored states, rather than the scored states discussed in the previous section. An unscored state has only the board configuration, rather than the board configuration and the net score. This decreases the search space by a significant amount (about 50 percent for a 4x4 board).

The massive number of board configurations for even small boards have already been shown. If each state consists of the board configuration and the net score, each board may have many different associated states. Specifically, for a board of n total boxes, there are between 1 and $n + 1$ different net scores (and thus states) per configuration. A configuration with n boxes captured has $n + 1$ possible net scores because a player may have any number of boxes from 0 to n (inclusive) and both players' scores add up to n. There are only a few exceptions to this rule, including boards with exactly 4 edges taken, making up one box. These boards can only have net scores of +1 because only the second player could have taken the box. The exact number of scored states for a given board size is difficult to determine. For a 2x2 board, there are an

estimated 5120 scored states, compared to 4096 unscored states (a roughly 25 percent increase). For a 3x3 board, there are about 26.2 million scored states, compared to the 16.8 million unscored states (a roughly 56 percent increase). For a 4x4 board, there are approximately twice as many scored states as unscored states. Although the totals for 5x5 boards or even larger boards are nearly impossible to determine, the percentage of increase is expected to be larger still. These estimates were calculated by iterating over each configuration, counting how many boxes were taken in each one, and adding $n + 1$ to the total.

However, the net score does not affect the optimal strategy. The optimal move depends only on the board configuration because once a box has been taken, it no longer affects the rest of the board. The optimal move is the one that achieves the most boxes in the rest of the game for the player who makes the move. This is not affected by how many boxes the player took in the earlier portion of the game. Buzzard and Cierre's loony endgame algorithms, for instance, disregard the score at the start of the loony endgame when determining optimal play [3]. Figure 11 shows the same board configuration with and without showing who took which box. Observe that since the taken boxes are separate from the rest of the board, only the edges that connect to an untaken box are relevant. The optimal move does not depend on knowing the score for either player; it does not affect how many boxes either player can take in the rest of the game.

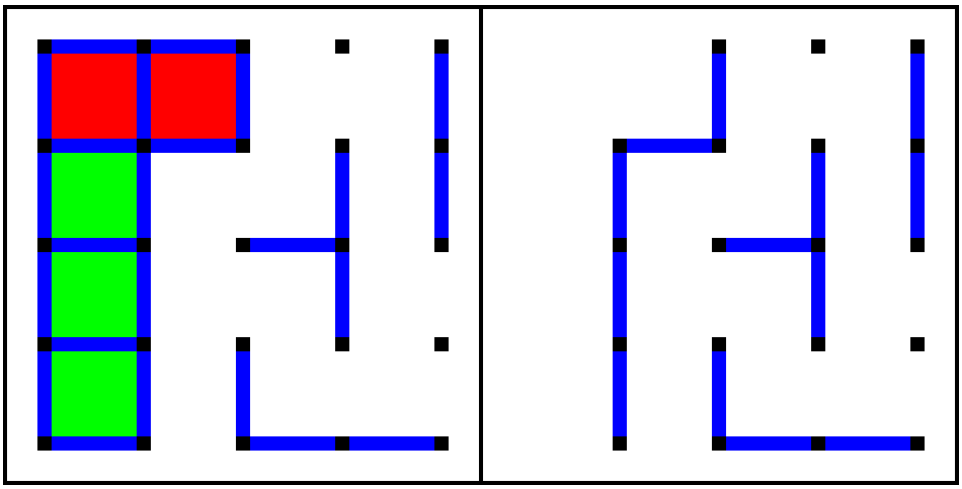

Figure 11: The same board with and without the score

One concern with removing the net score from the state is whether a board configuration B, which is reached with a bad net score, is given an artificial boost to its value estimate when the same board configuration is reached with a good net score. This concern stems from the use of a graph to represent the game tree, which allows two paths to lead to the same state, as opposed to the normal tree representation, which does not. Imagine a board configuration in which the net score is either -5 or +5 (for simplicity's sake, ignore the other net scores on such a board). Consider the path of board configurations pb = [s0, s1, s2, …, sn, B] that lead to a score of -5 the 'bad' path and the path of board configurations pg = [s0, s1', s2', …, sn', B] that lead to a score of +5 the 'good' path. If pb is taken and the game results in a loss, does this not make the pg seem less desirable because the board B it leads to has led to a loss? No. Although they lead to the same board B, only the path that was taken is be updated. That means that actions along the pb have decreased values, and actions along pg are not be updated. The values in a node are updated only based upon the results of choosing an action, not on the values of later nodes. So even when the game tree is represented as an acyclic graph, different paths are treated the same way they would be if represented as an actual tree.

However, this does give rise to another concern. Although path pb is not affected by path pg (or vice versa), the value of each action possible in connecting configuration B and later are affected by which path was taken. This is where the acyclic graph representation varies from the simple tree representation. In the tree representation, even without net score as part of the state, no two paths would ever connect. Using a graph representation allows two paths to do so if they lead to the same state. Imagine there are only 9 boxes on the board (and so the bad path always results in a loss and the good always results in a win). Further assume that B was reached along the bad path. Whichever action is selected in B – and all the configuration in the rest of the game – does not affect the outcome of the game. Thus, the actions made in the rest of the game will be random, because they all lead to a loss. In games where the outcome is not determined (only influenced) by the net score this could lead to skewed the values.

This concern is partially mitigated by the MCTS itself. The use of unscored states potentially adds some random or semi-random static to the move selection from the meeting point for paths pb and pg (configuration B), but the Q(*s*, *a*)'s of the configuration and actions in pb (the bad path) decrease with every negative result and their uncertainty bonuses decrease as well (It is important to note that Q(*s*, *a*) is a feature of the tree, so the Q(*s*, *a*) of an action in B refers to the node that represents B in the tree). So the MCTS algorithm tends to start avoiding configuration along pb. After pb is no longer traversed, only pg remains, so the Q(s, a)'s of B and its children are no longer be skewed by pb and again approach the optimal moves.

Even if the MCTS would not stop selecting pb because of its undesirability, pb should not affect the optimal move. Because the optimal move is the one which nets the

current player the most points and this is not affected by the earlier game, the optimal move for pb is the same as for pg. Because pb is worse than pg, each of the children of B is worse for pb than pg. However, because the decrease in their value for pb relative to pg is uniform, the Q(*s*, *a*) values of all children decrease uniformly. This means whether pg or pb is taken, what MCTS learns for B and the remaining configurations in the game gravitates toward optimal play. It is possible that it would take MCTS more simulations to learn optimal play, but the current expectation is that significantly more time is saved by decreasing the search space than is lost due to the additional simulations needed to reach optimal play.

6.2 NON-SYMMETRICAL STATES

The second implemented improvement is representing the board configuration in a way that treats symmetrical configurations as the equivalent states. In other words, symmetrical board configurations are represented as the same state. This decreases the search space to roughly 1/8 the original size, which in turn decreases the number of simulations needed to achieve the same results and decreases the number of nodes on the tree.

Although there is a vast number of configurations for most boards, many of these states are symmetrical. Each configuration has horizontal and vertical symmetry. Likewise, square boards have rotational symmetry. In other words, each board can be rotated 90 degree 3 times for a total of four states. Then it can be reflected either horizontally or vertically and rotated again. In total, on a square board, every configuration is symmetrical to 7 other states. Some of these states are identical (when the board configuration itself is symmetrical), but in general, for a board with *n* possible

configurations there are roughly *n*/8 states, none of which are symmetrical to any other. For instance, a 1x1 board drops from 16 states to 6 states, a 2x2 board drops from 4096 to 570, and a 3x3 board drops from 16,777,216 to 2,102,800. These numbers were calculated by iterating over each board configuration and comparing it to its symmetrical configurations. If the state was the 'canonical state' – the only one of the 8 states which is used in non-symmetrical states – the total was incremented. If not, it was discounted. For the remainder of this thesis, states or nodes which represent all their symmetrical counterparts are considered 'non-symmetrical'. Likewise, a MCTS player using a tree which accounts for symmetries is a non-symmetrical player. A MCTS player using a tree which does not account for symmetries is a symmetrical player. The number of non-symmetrical configurations for board sizes of 4x4 and greater have not been calculated since a 4x4 board would take more than 300 hours, and a 5x5 board would take more than 36,000 years, given the current resources. This is based on the number of configurations on these boards and the time it takes to check a single configuration.

```
get_canonical_config (config) :
  current_config ← config

  for 1 to 4
    transformed_config ← config after rotation
    if current_config < transformed_config
      current_config ← transformed_config
    transformed_config ← config after reflection
    if current_config < transformed_config
      current_config ← transformed_config

  end for

  return current_config
end
```

Algorithm 3: Getting the canonical representation of a board configuration

To determine the canonical configuration of a given configuration, the integer representing the board configuration *b* is converted to its binary representation *b*'.

Algorithm 3 shows the process by which the canonical version of *b* is obtained. The value *b*' is transformed seven times (three rotations, a reflection, and three more rotations) to encompass the seven symmetrical configurations b_i' (for i = 1, … ,7). To determine where each edge is located after a rotation or reflection, a 'map' of reflected or rotated edges is kept for each board size. Each transformed configuration b_i' is compared to *b*', one digit at a time; the first digit which is different between b' and b_i' is the determining digit. If *b*' has a zero and b_i' has a one, the *b*' becomes b_i' and the process continues. Because *b*' and b_i' are represented as an integer, this can be efficiently done by comparing the numbers themselves. The larger number becomes *b*' in every comparison. After all seven have been checked, *b*' is the canonical configuration. In other words, the canonical configuration is the one in which the drawn edges are closest to the top and closest to the left of the board. To determine which actions are symmetrical for a board, each state that is the result of an action *a* is checked to see if it is canonical. If it isn't, *a* is discarded. If it is, *a* is a symmetrical action. Figure 12 shows eight symmetrical states. On the top row from left to right, the board is rotated 90 degrees at a time. From top to bottom, the board is reflected horizontally.

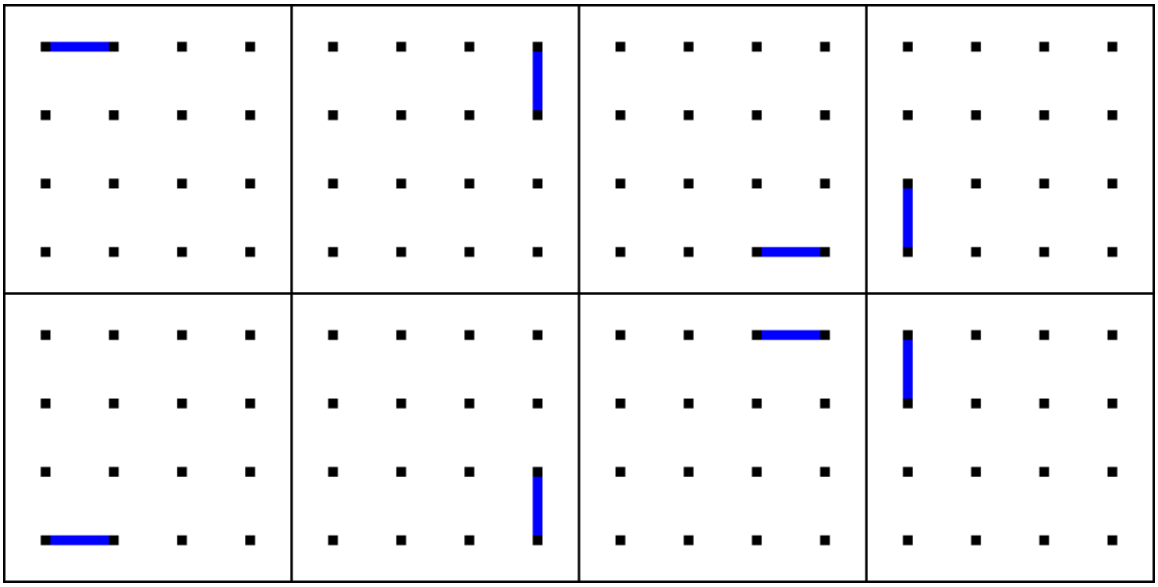

Figure 12: Tracking rotation and reflection to show 8 symmetrical states

Computationally, there is no difference between the 8 states which share rotational or reflective symmetry. Since the orientation of the board does not affect the score, the available moves, or the results of these moves, any two boards which share symmetry also share those traits, and thus share optimal moves. This means that storing each of the symmetrical states as a separate node on the tree (with separate paths to reach them and separate paths from them to final states) adds no information that would not be present if only one of the eight states were part of the tree. The only difference comes from the application of this information. Much like the use of unscored states, removing symmetrical states from consideration decreases the number of states in the tree, which likewise decreases the number of simulations needed to access a significant portion of the tree and the space required to store the tree data. It also increases the potential for a single simulation's power to update the value of the nodes it traverses.

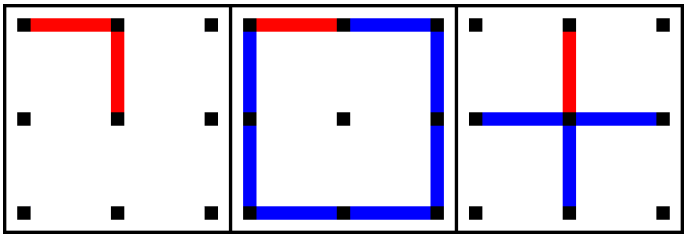

Figure 13: The symmetrical opening moves of the 2x2 board

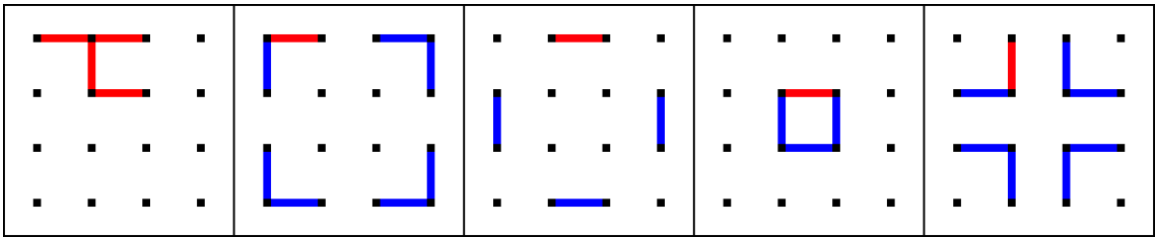

Figure 14: The symmetrical opening moves of the 3x3 board

Figures 13 and 14 above show the opening moves in a 2x2 and 3x3 board. In each figure, the first board shows the non-symmetrical opening edges (the edges that all lead to different non-symmetrical configurations in the first move). The remaining boards in each figure show each non-symmetrical edge (in red) with its symmetrical counterparts (in blue). Notice that, by some combination of reflection or rotation, the red edges on each board can be transformed to become each of the blue edges. So, from an initial 12 moves, there are only two which are non-symmetrical for a 2x2 board. For the 3x3 board there are 4 non-symmetrical edges among the possible 24 initial edges on the board. While the first non-symmetrical move for a 2x2 board (shown in the middle board in Figure 13) has the full 8 symmetrical states, the other move (connected to the center point of the board) has only four, since its reflection is also one of its rotations. It is states like that which make the number of non-symmetrical states slightly higher than 1/8 the total number of states.

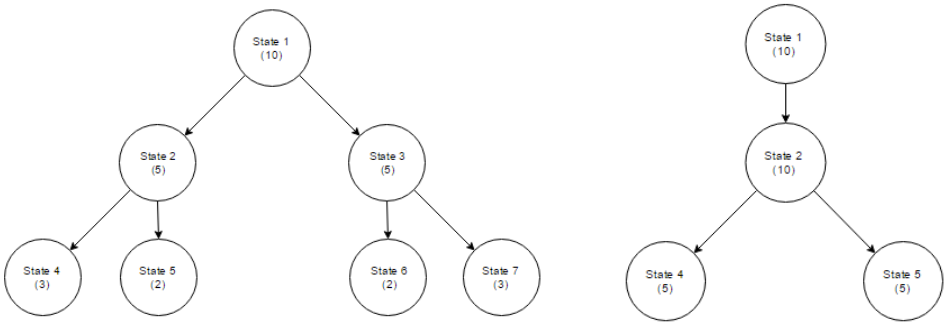

Figure 15: A tree with two symmetrical children and with symmetries combined

The benefit of considering only these two board configurations (instead of the 12 total) is clear. For a given number of samples, the number of times Monte Carlo tree search would select an available action depends (in part) upon how many actions there are. Imagine a configuration with two symmetrical children (shown in the left tree of Figure 15, above, as states 2 and 3). Because the states are symmetrical, the average reward Q(s, a) of the states should remain close to the same value. Thus, each of the two states is sampled about the same number of times. However, if each child is considered the same state (as in the right tree in the figure above), then all samples are used on the same child. Thus, the value is updated twice as fast. And this means that all the children of states 2 and 3 (4, 5, 6, and 7) are updated twice as fast. This change would propagate through the tree. The first state would update twice as fast and the symmetrical children of the state would update twice as fast again because they would be combined. The improvement in updating speed would grow exponentially by depth. Because most nodes have 8 symmetrical states, a tree with non-symmetrical states is expected to update roughly 8 times as fast as a tree with symmetrical states.

7. RESULTS

To examine the benefits of the two approaches implemented in this thesis (unscored states and non-symmetrical states), a variety of tests were performed on players using some combination or scored or unscored states and symmetrical or non-symmetrical states. In each of the figures shown, the results for players who were tested using the 'default' setting are shown as a solid line, while results for a player using the 'improved' setting are shown as a dashed line. In other words, in tests to determine the viability of unscored states, a scored player's results are shown as a solid line, while an unscored player's results are shown as a dashed line. Likewise, in tests of the non-symmetrical states, the symmetrical player is shown with a solid line and the non-symmetrical player is shown with a dashed line. In addition, the four square board sizes tested (2, 3, 4, and 5) were each assigned a color (blue, orange, green, and red). In each figure, the board size represented in the test is shown with the corresponding color.

In each test, a series of games were played between two separate MCTS players, using two separate trees. To maintain the accuracy of the averages, the number of games played depended on the size of the board on which the games were played. Board sizes of 2, 3, 4, and 5, were played with 200, 500, 1000, and 2000 games, respectively. In each game, each player started with a tree containing only the root node. Before each move was made, the player to move ran a certain number of simulations. When all simulations had finished, the player selected an action using their tree. Once an action was selected, the trees of both players were advanced with that action. In this way, the current position of each player on the separate trees remained the same.

7.1 UNSCORED STATES

The first step in evaluating the unscored player was to quantify the difference between the number of nodes found during a game for an unscored player and a scored player. Figure 16 shows this difference with the average number of nodes located in the tree for each player at the end of the game. As expected, each of the scored players had more nodes in the final tree than the unscored player with the same board, and the gap between the scored and unscored players increases as the number of simulations increases. It is likely that the unscored player would reach the point at which more simulations do not increase the average number of nodes faster than the scored player. However, since data was only collected for up to 150,000 simulations, it is unknown how much of a final difference there would be. Figure 17 shows the data for the 2x2 board on a visible scale. The rate of increase in the number of nodes found starts to shrink rapidly at around 20,000 simulations.

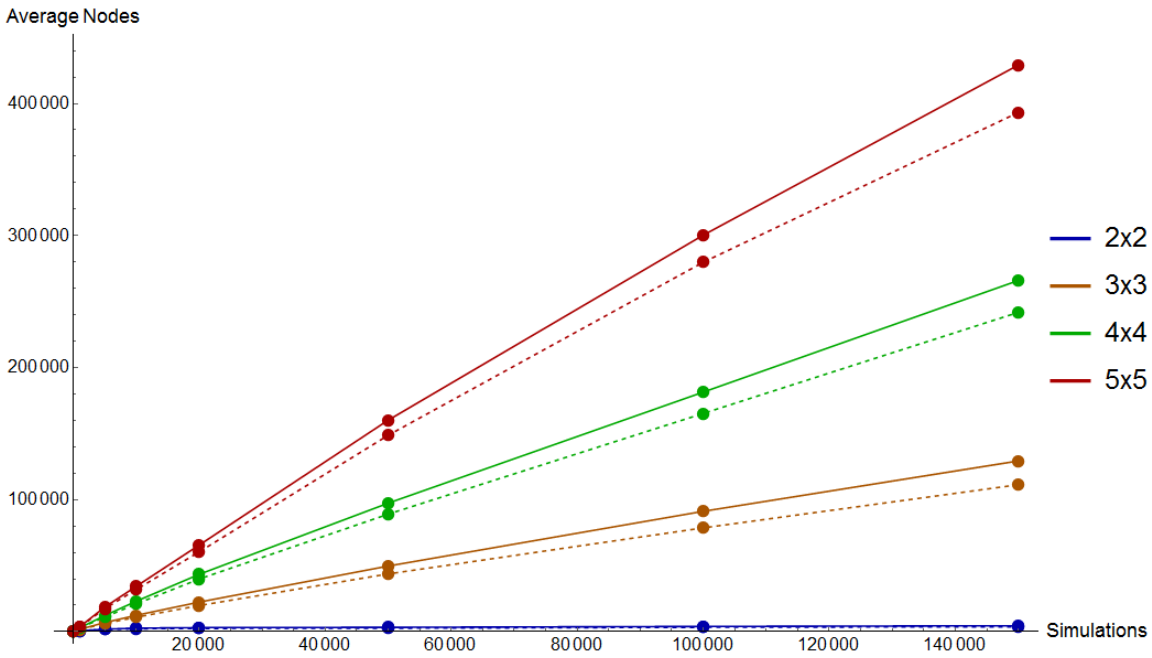

Figure 16: The average nodes in the final tree for scored (solid) and unscored (dashed) players

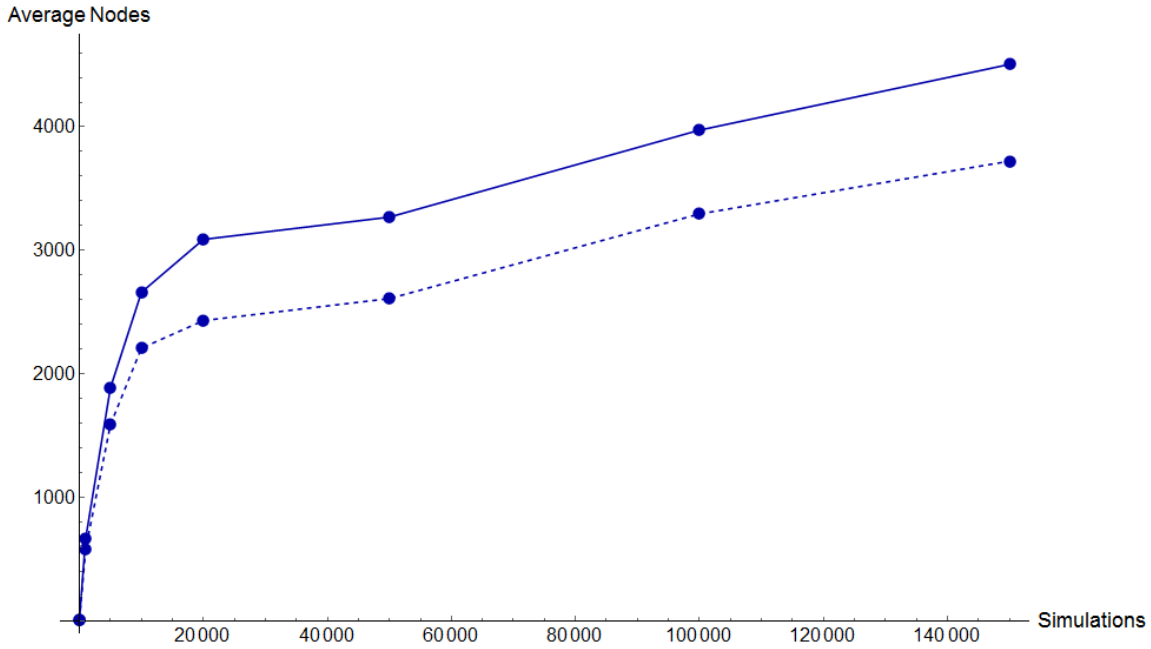

Figure 17: The average nodes in the final tree for scored (solid) and unscored (dashed) players on a 2x2 board

A player using unscored states was tested against another unscored player. To examine the effect of the number of simulations given per turn, the test was performed with 6 different numbers of simulations for the first player (0, 1000, 5000, 10000, 20000, and 30000) and a static number for the second player (5000). The same tests were performed with scored players playing against other scored players. The goal of this test was to verify that the unscored player could learn to win as quickly against another unscored player as a scored player learns against a scored player.

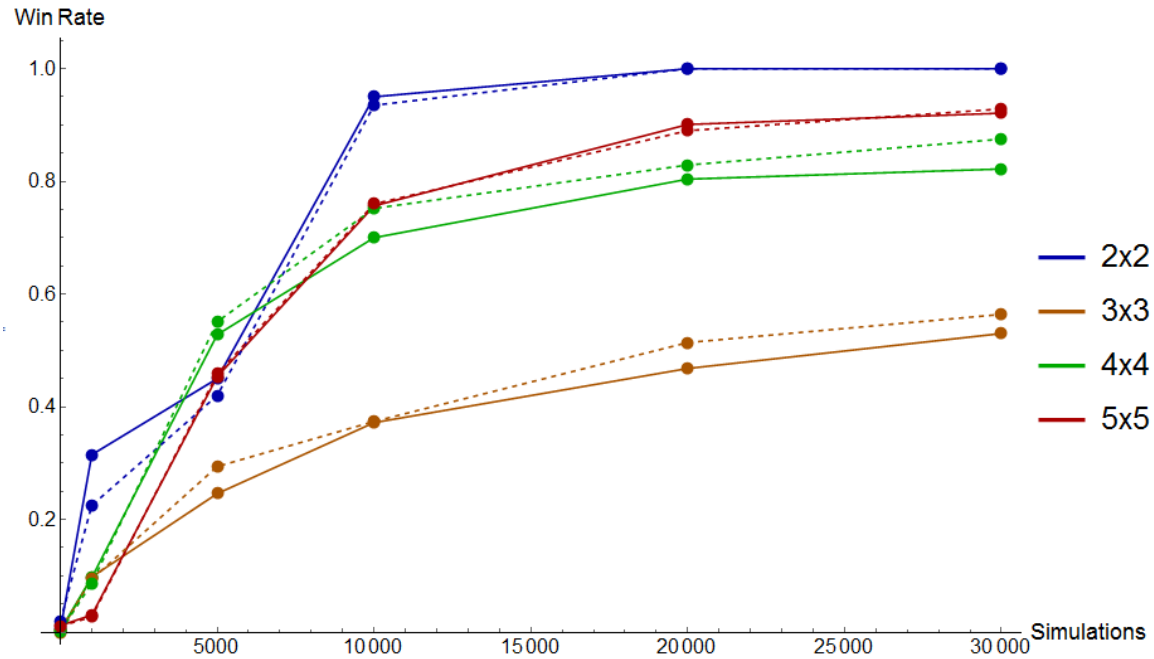

Figure 18: The win rates for scored (solid) and unscored (dashed) players as player one's simulations increase and player two's simulations remain static with both players facing equivalent opponents

Figure 18 shows the win rate for both scored and unscored players on square boards of length 2, 3, 4, and 5. The *x* axis shows the number of simulations given to player one during each turn and the y axis shows the win rate for player one as a value between 0 and 1. As expected, the win rate for zero simulations is nearly zero for every board size; the opponent has 5000 simulations, so they have a significant advantage against a player making random moves. At 5000 simulations (when the players are equally matched), player one wins between 40 and 60 percent of the time for board sizes 2, 4, and 5. A 3x3 board is a loss for player one in optimal play, so it makes sense that even with an equal number of simulations player one loses significantly more than half the time. A 2x2 board is a win for player one and a 4x4 board is a draw, so the win rate makes sense. As the number of simulations increases all sizes improve, with all but the 3x3 board (which is known to be at a disadvantage) approaching a win rate close to 1.

As expected, the rate of learning for scored and unscored players are roughly equal. For a 5x5 board, the difference between the two is virtually nonexistent, likely because with so few simulations the larger state space of the scored player did not have an impact. For a 2x2 player, the state space is so small that most or all the states were found by both players, giving the scored player (whose information is perfect) a slight advantage. For the 3x3 and 4x4 boards, the unscored player learned slightly faster than the scored player. This shows that the loss of information to the unscored player does not negatively impact its ability to learn.

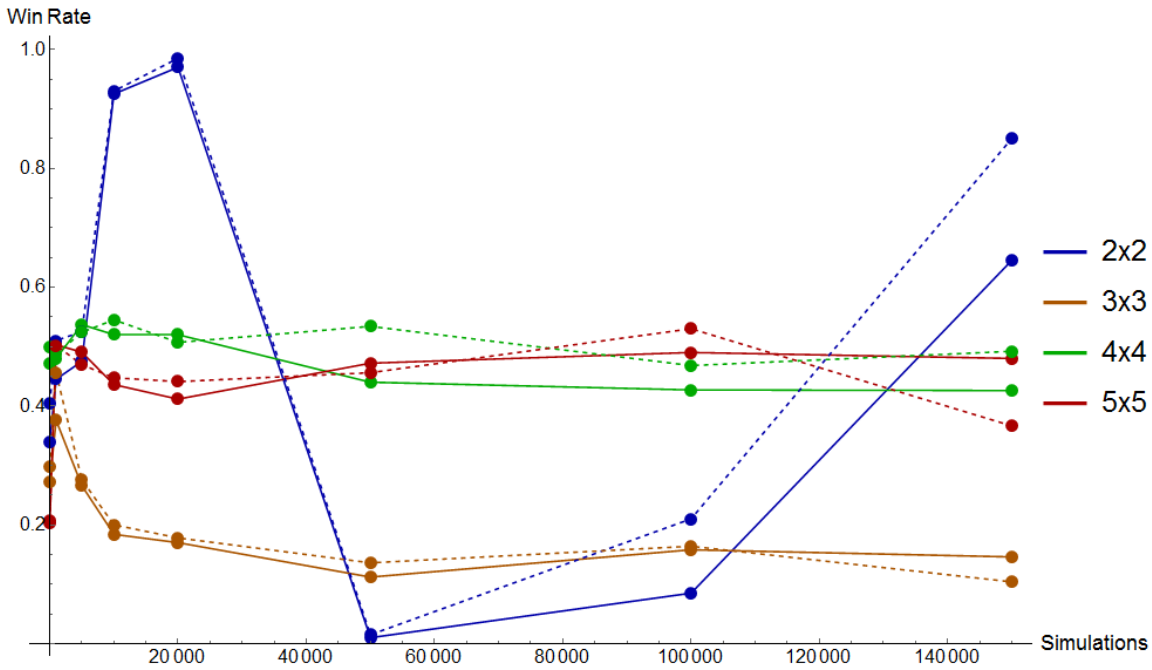

Figure 19: The win rate for scored (solid) and unscored (dashed) players against and unscored player

Figure 19 shows the results of similar games played by scored and unscored players. In this test, each player was pitted against a player using unscored states and both the tested player and the opponent were given an equal number of simulations (shown on the y axis). Because each player faces the same opponent, any difference in the win rates between the scored and unscored players should be the result of a difference in learning

ability between the two. Again, the win rates for the players using unscored states are shown by dashed lines and the win rates for players using scored states are shown by solid lines. From the original predictions, it was expected that the unscored players would play better (achieve higher win rates) than their counterparts using scored states. However, each pair of players seems to be evenly matched. For each board size where the optimal result is known, the win rates seem to match that result. The first player eventually learned to consistently win on a 2x2 board, the players consistently tied on the 4x4 board, and the second player learned to win most of the time on a 3x3 board. For larger numbers of simulations, the unscored player performed slightly better than the scored player on boards of size 2 and 4. For board sizes 3 and 5, the scored and unscored players performed almost equally.

On the 2x2 board, the first player suffered a severe decrease in win rate between 20,000 and 50,000 simulations. This is likely the result of the first player initially (in smaller numbers of simulations) learning a path that was very likely to result in a win. However, when the second player was given enough simulations, it was able to learn a way to beat player one's strategy, at which point player one started to lose almost every game. The more times the first strategy resulted in a win, the more entrenched it became in player one's gameplay. That is why it took more than 100,000 extra simulations to recover from that strategy.

7.2 NON-SYMMETRICAL STATES

Although accounting for symmetries reduces the size of the search space and the number of nodes added to the tree (as well as improving the number of simulations needed to converge on optimal results), it requires some extra computation, which

negatively impacts the time needed for each simulation. The extra computation needed for each simulation has the potential to lessen or even counteract the time improvement from needing fewer simulations. If accounting for symmetries decreases the number of simulations needed to reach some benchmark of accuracy by half, but increases the time needed for each simulation to 3 times, accounting for symmetry in fact increases the overall time.

To examine this possibility, both a symmetrical player (one which does not account for symmetries) and a non-symmetrical player (one which removes symmetries) were timed in separate sets of 100 games played against another player with the same settings on a 3x3 board with 10,000 simulations per move. In all the non-symmetrical tests, players used unscored states. Both played as the first player in their respective games, and the number of milliseconds taken by the player during each turn was recorded. The vast majority of the time taken for a move is spent performing the simulations. At the end, the time taken by the players in each turn was averaged over the number of times they took that turn (turn one, two, etc.). In other words, if the player took a turn 90 times in the 100 games, the total time for that turn was divided by 90 to get the average time that turn took. This is necessary because a player does not make the same turns in every game. For example, the first player might make the 7^{th} move 50 percent of the time.

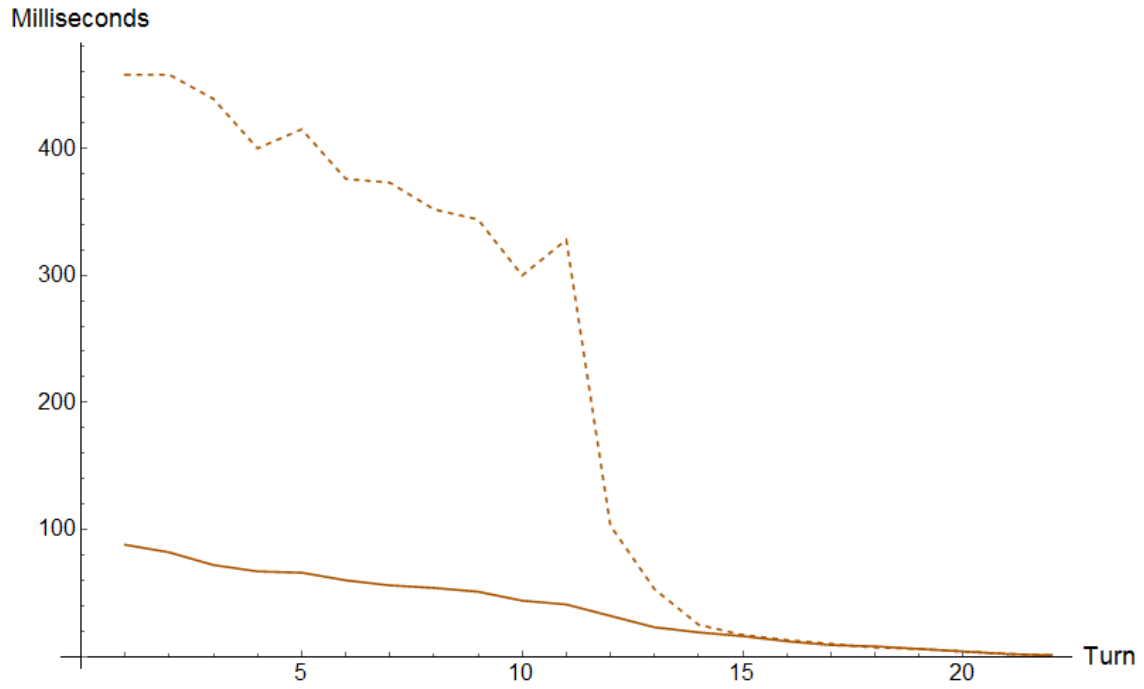

Figure 20: Average turn times for non-symmetrical (dashed) and symmetrical (solid) players on a 3x3 board

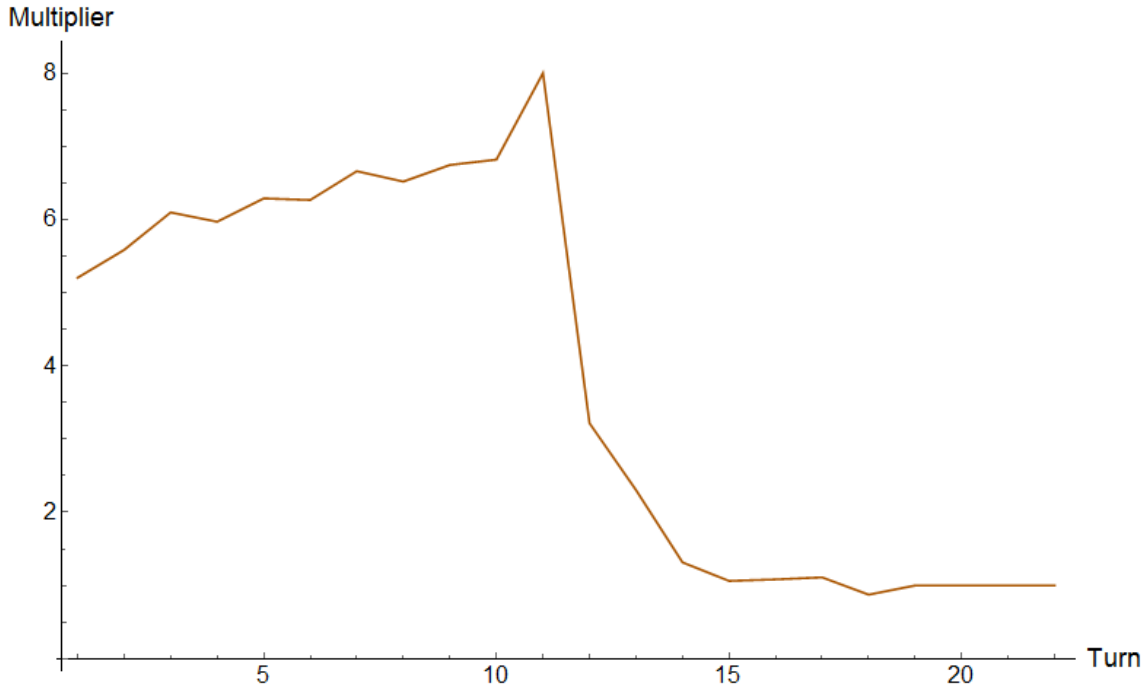

Figure 21: The average turn times for a non-symmetrical player as a factor of the average times for a symmetrical player on a 3x3 board

Figure 20 shows the average times per turn of the symmetrical and non-symmetrical players. In the first half of the moves, the non-symmetrical player takes

considerably longer than the symmetrical player. In the second half of the game, the non-symmetrical player becomes approximately as fast as the other. Figure 21 shows the same results with the average time for the non-symmetrical player as a factor of the average time for the symmetrical player. In other words, the y-axis of Figure 21 represents the number the symmetrical average must be multiplied with to reach the non-symmetrical average. Clearly, the non-symmetrical player takes longer than could be hoped. For the first half of the game, the non-symmetrical player takes roughly six times longer than the symmetrical player, rising to a peak of eight times longer. In the last half of the game, it rapidly drops to take roughly the same time as the symmetrical player. Although not ideal, the factor never surpasses eight, which means that the loss of time from the extra computation does not surpass the expected decrease in the number of simulations needed. Figures 22 and 23 show the same comparisons for a 4x4 board. In this case, the multiplier from the average time of the symmetrical player to the average time of the non-symmetrical player again rises from six to eight before dropping to approximately one. However, the drop to a multiplier of one took a larger percentage of the game than on a 3x3 board. This seems to indicate that for larger board sizes, the increase in computation time may overcome the decrease in simulations needed.

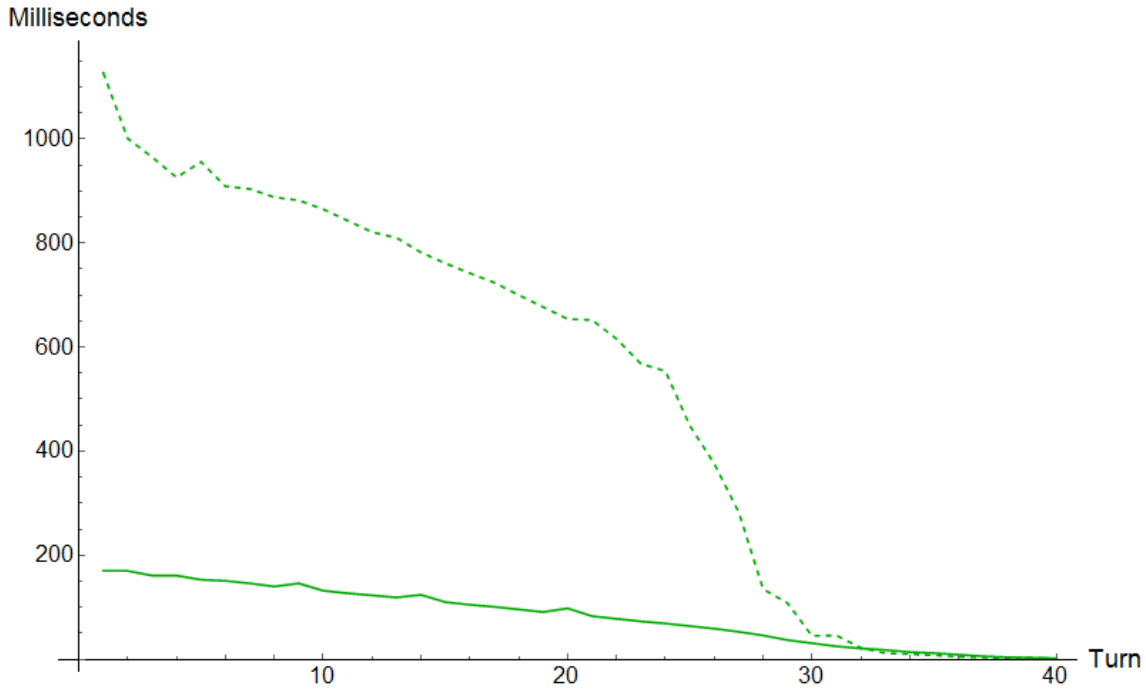

Figure 22: Average turn times for non-symmetrical (dashed) and symmetrical (solid) players on a 4x4 board

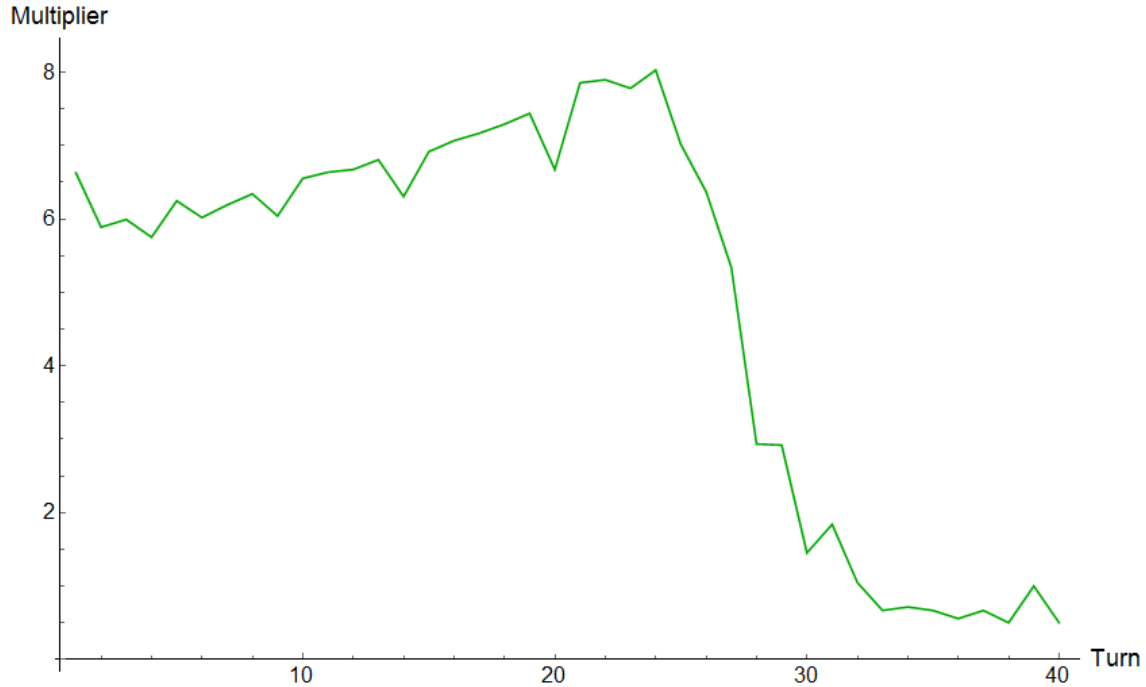

Figure 23: The average time for a non-symmetrical player as a factor of the average time for a symmetrical player on a 4x4 board

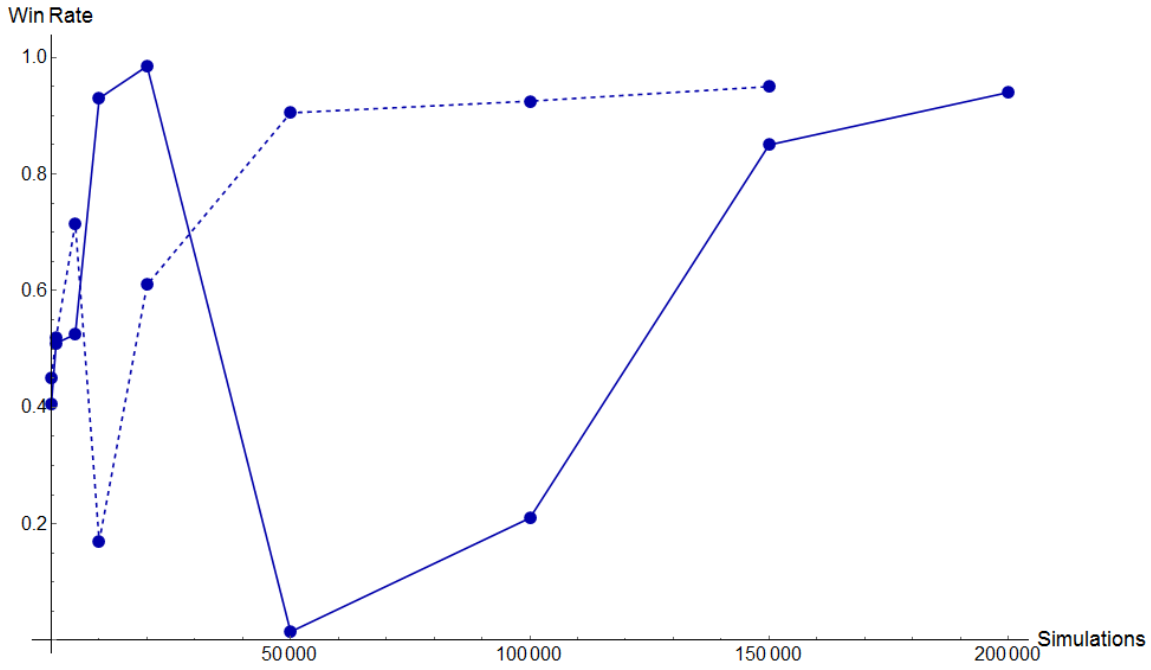

Figure 24: Win rates for symmetrical (solid) and non-symmetrical (dashed) players on a 2x2 board playing against equivalent opponents

Figure 24 illustrates the results of games played between two symmetrical players and between two non-symmetrical players. Because the non-symmetrical player has a smaller search space, one would expect it to exhibit the same behavior as the symmetrical player, but over fewer simulations. This figure shows the symmetrical player quickly learning to win most of the time before dropping to a win rate of nearly zero and slowly rising back up to a high win rate. The dashed line shows the same behavior being exhibited by the non-symmetrical player. It first learns to win most often, before suddenly dropping to a low win rate. Then it slowly learns to win at a high rate again. However, while the symmetrical player takes 200,000 simulations to recover its approximately 90 percent win rate, the non-symmetrical player does so with only 50,000 simulations.

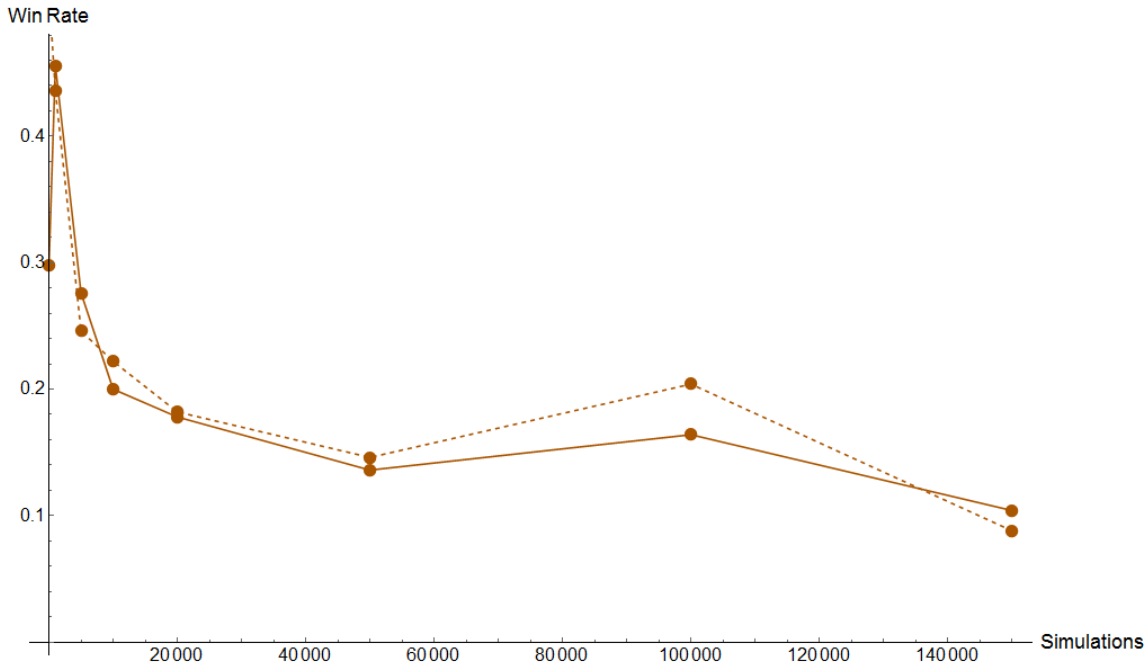

Figure 25: Win rates for symmetrical (solid) and non-symmetrical (dashed) players on a 3x3 board playing against equivalent opponents

Figure 25 shows the results of the same test on a 3x3 board. For this size, there is significantly less prominent graph behavior to judge. With more than 10,000 simulations, the win rate hovers between 10 and 20 percent all the way to 150,000 simulations. However, in the initial drop from about 50 percent at 1000 simulations, the symmetrical and non-symmetrical players are nearly the same. Because of the increase in computation time for the non-symmetrical player, the results of this test for board sizes of 4 and 5 could only be determined to 20,000 simulations. Even if more data was collected, however, win rates for symmetrical players on those boards remain about 50 percent even up to 150,000 simulations, so there is virtually no graph behavior by which an increase in learning speed could be measured.

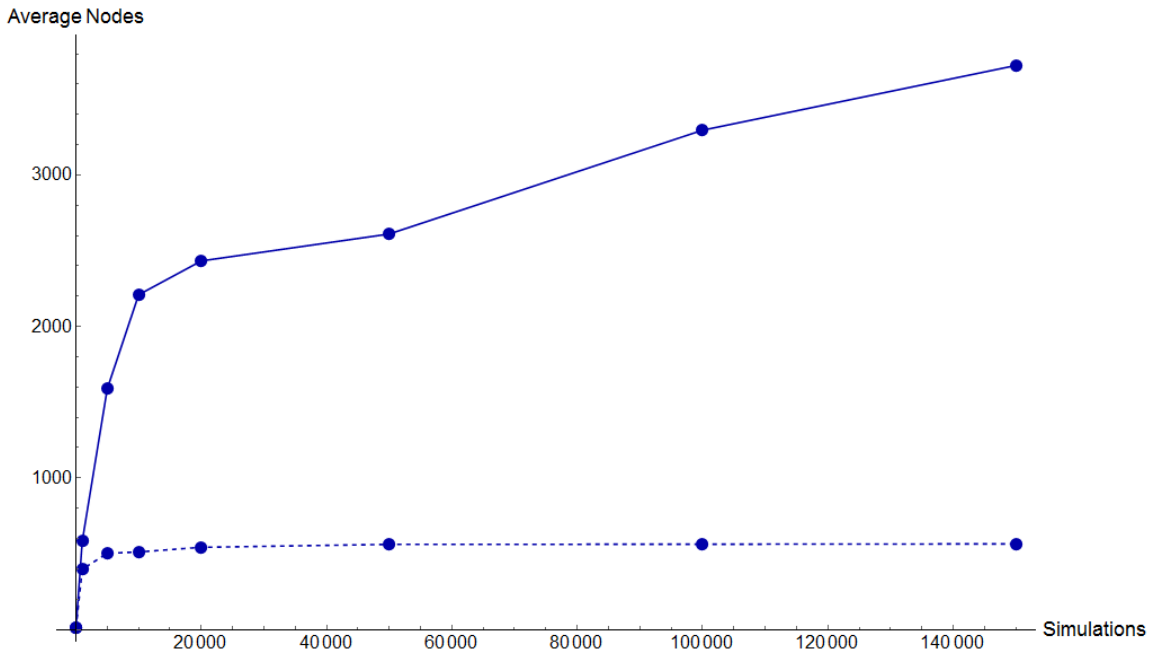

Figure 26: The average number of nodes for symmetrical (solid) and non-symmetrical (dashed) players on a 2x2 board

The number of nodes in the average tree was tested again for the symmetrical and non-symmetrical players. In Figure 26, it can be seen that even with only 5000 simulations, the non-symmetrical player has an average of 503 out of 570 possible nodes on its final tree. This is roughly the same percentage of the search space explored by the symmetrical player using 150,000 simulations. In fact, at 50,000 simulations, the non-symmetrical player explored about 99 percent of the search space. It seems that while the non-symmetrical player has a search space $1/8^{th}$ the size of the symmetrical player, it explores a larger portion of that search space. This is certainly helpful for players on a 2x2 board, where the search space is small enough to be completely explored, but it may be obstructive on larger board sizes. This may explain why Figure 25 does not show the non-symmetrical player's win rate decreasing any faster than the symmetrical player.

8. CONCLUSIONS

Dots and boxes, while appearing simple compared to many other combinatorial games, presents many challenges for computation and for artificial intelligence players. The vast search space and the difficulty of evaluating the advantage of a given state has made approaches that have worked well in similar games untenable for dots and boxes. The use of Monte Carlo tree searches represents a powerful new approach to the problem, and has worked well for Go, which has similar limitations.

Using unscored states did not negatively impact the search's ability to learn valuable strategies. In some cases, it even caused the search to perform slightly better. The average number of nodes in the final trees were also smaller for unscored players. The expectation was that the smaller search space would lead to substantially faster learning and better performance, although the difficulty in determining the search space for scored states made it problematic to make specific predictions of how much faster the learning would be. It was expected that the loss of information from the removing the score would be negated and surpassed by the benefits of updating individual nodes more quickly and the smaller search space. The data collected shows this not to be the case, though. It seems that the loss of information misleads the search on a scale roughly proportionate to the increased updating speed of the nodes. Overall, the use of unscored states does not detract from the MCTS player; the differences are generally beneficial, or at worst inconsequential, so it is a worthwhile approach to improving the search.

The exclusion of symmetrical states from the tree significantly decreased the search space without sacrificing any information, but the increase in computation time counterbalanced or even exceeded the benefit of this approach in practical applications

where time is a factor. Although it was predicted to increase the power of the search by as much as 8 times, the true potential in the current implementation is much smaller (perhaps as small as 3 times). The computation time, on the other hand, increased by as much as 8 times for large portions of the game on boards of size 3 and 4, with larger boards likely to see even higher rates. This computational slowdown may be decreased in future work as more efficient means can be used to calculate the canonical states, but in its current form, it seems to exceed the benefit of the smaller search space, so the non-symmetrical does not seem to be a valuable approach, unless storage space is much more limited than time.

Of the two approaches explored in this thesis, the removal of net score is the most promising. It results in either equivalent or slightly better gameplay with slightly fewer nodes. Non-symmetrical players will only be valuable if more efficient methods can be found for computing the symmetrical states. The Monte Carlo tree search itself seems to be a valuable method for a program to learn to play dots and boxes and future explorations into potential improvements of the search may produce significantly better players.

9. FUTURE WORK

There are many potential avenues of exploration for future work. One such avenue is varying the tuning constant of the uncertainty bonus based on the depth in the tree. Because dots and boxes has a specific number of moves per game, the tuning constant may be able to be based on the depth of the search, since the number of future paths decreases in a known manner every move. Imagine that you want to tune the uncertainty bonus so that a node is explored x times before its bonus drops to a negligible amount and its value is essentially based on the $Q(s, a)$. For the root node, x simulations might sample 50 percent of the tree while for a node deeper on the tree, x simulations might sample 90 percent of the tree. It might be beneficial to sample a specific node a specific percentage of the number of children it has or the number of total descendants. To do this, one could decrease the tuning constant with the depth.

Another possible avenue is the addition of transposition in the backup stage of the simulation. This would involve updating not only the path that had been taken in the simulation, but every possible path that could lead to the node. For instance, if a node x on the path that resulted in a win had two parents (one which was also on the path and one which was not), the $N(s, a)$, $N(s)$, and $W(s, a)$ of both parents would be updated as though each was a node on the path. Likewise, the parents of other nodes on the path taken in the simulation would be updated. Thus, all ancestors of a node on the path are updated. This allows each simulation to do the work of many simulations. The benefit for a MCTS tree with full state information (the board configuration as well as the net score) is clear. No matter how a state is reached, the probabilities of each possible outcome from that state are the same. As was explained in section 6.1, having the net score and the

board configuration uniquely identifies a state of the game and any more information (such as the path) is irrelevant. For an unscored state, which does not have full state information, transposition would not achieve equivalent results. This is because of a concern raised in section 6.1 about nodes with the same board configuration but different scores artificially skewing the backpropagated values for both for each other. That concern was invalid this project, but the addition of transposition would make it valid.

The exploration of transposition, then, would seek to answer two questions. By how much does transposition decrease the number of simulations needed to reach the same result in scored states? In unscored states, does the use of transposition positively impact the search or does the lack of score information skew the results in a negative way? It may be that the decrease in the number of nodes would offset the skew values of the transposed backpropagation.

There are some areas of code in the Monte Carlo tree search program used in this thesis which are relatively inefficient. In future work, it would likely be beneficial to alter these areas to improve their efficiency. In particular, the calculation of canonical states in non-symmetrical players has the potential to be drastically expedited, which would lead to a smaller difference in average simulation times between symmetrical and non-symmetrical players. This avenue could make the non-symmetrical approach a much more valuable improvement.

Finally, it would be valuable to collect mare data, both from larger board sizes and from larger numbers of simulations. The data collected for this thesis was for board sizes of only 2-5, with a maximum of 200,000 simulations per turn. Even for a 3x3 board, the largest search had an average final tree size of about 100,000 (less than one percent of

the search space). Larger numbers of simulations could reveal different behavior in the players (like how the 2x2 player completely reversed its win rate at 50,000 simulations and took until 200,000 simulations to recover). Likewise, larger board sizes might reveal more significant gaps between scored and unscored players (since with larger board sizes the percentage difference in the search space of the two is expected to increase).

REFERENCES

[1] J. Barker and R. Korf, "Solving Dots-and-Boxes," in *AAAI Conference on Artificial Intelligence*, Toronto, 2012.

[2] C. Browne, E. Powley, D. Whitehouse, S. Lucas, P. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis and S. Colton, "A Survey of Monte Carlo Tree Search Methods," *IEEE TRANSACTIONS ON COMPUTATIONAL INTELLIGENCE AND AI IN GAMES*, vol. 4, no. 1, 2012.

[3] K. Buzzard and C. Michael, "Playing Simple Loony Dots and Boxes Endgames Optimally," *Integers*, no. 14, 2014.

[4] M. S. Campbell, A. J. J. Hoane and F.-h. Hsu, "Search Control Methods in Deep Blue," American Association for Artificial Intelligence, 1999.

[5] G. Chaslot, M. Winands and H. J. van den Herik, *Parallel Monte-Carlo Tree Search*, Universiteit Maastricht.

[6] S. Gelly, L. Kocsis, M. Schoenauer, M. Sebag, D. Silver, C. Szepesvári and O. Taytaud, "The Grand Challenge of Computer Go: Monte Carlo Tree Search and Extensions," *Communications of the ACM*, vol. 55, no. 3, pp. 106-113, March 2012.

[7] S. Gelly and D. Silver, "Monte-Carlo Tree Search and Rapid Action Value Estimation in Computer Go," *Artificial Intelligence*, vol. 175, no. 11, pp. 1856-1875, 2011.

[8] "Minimax search and Alpha-Beta Pruning," [Online]. Available: https://www.cs.cornell.edu/courses/cs312/2002sp/lectures/rec21.htm. [Accessed 10 4 2017].

[9] D. Wilson, "Dots-and-Boxes Analysis Index," [Online]. Available: http://wilson.engr.wisc.edu/boxes/. [Accessed 25 3 2017].

[10] Y. Zhuang, S. Li, T. V. Peters and C. Zhang, "Improving Monte-Carlo Tree Search for Dots and Boxes with a novel board representation and artificial neural networks," *IEEE CIG*, pp. 314-321, 2015.