Parallelization of the Monte Carlo Tree Search Approach in Dots-and-Boxes

Pranay Agrawal

Supervisor

Dr. Uta Ziegler

Western Kentucky University, May 2019

Faculty of Engineering and Applied Sciences

School of Engineering and Applied Sciences

Abstract

This project aims to explore how various factors affect the win ratio performance of a game learned through a parallelized Monte Carlo Tree Search approach. The Monte Carlo Tree Search [2] is a method designed to solve difficult problems by performing random simulations and storing the results in a tree in order to distinguish decisions based on their quality. The win ratio performance is determined by taking the number of wins over the number of total games. A parallelized approach divides the learning process among various process nodes on a high-performance computing platform. The data is shared periodically after a fixed number of simulations. The most useful information to share is the number of times each state or (state, move) pair was visited together with cumulative information whether the game was won or lost since this is used to compute the quality of each move of a given state. In this project, we determine how manipulating the different resources can impact the learning process of a parallelized Monte Carlo Tree Search. In order to achieve this, we use the game Dots-and-Boxes. An algorithm is presented which shares information from three tree levels, along with details of the Monte Carlo Tree Search implementation and results.

Contents

1 Introduction

Technology has given us a possible future in the field of Artificial Intelligence in today's rapidly changing world. The Oxford Dictionary defines Artificial Intelligence (AI) as the theory and development of computer systems able to perform tasks that typically require human intelligence. One significant leap that AI innovators took is the Monte Carlo Tree Search (MCTS) approach [2]. Currently, it is well understood that developing an innovative solution for every new situation is time-consuming and ineffective. Machine learning using the Monte Carlo Tree Search approach, however, can continuously learn new techniques and grow more efficient over time.

Since it is often difficult to determine the effectiveness of algorithms in complex environments, it is often more advantageous to develop strategies in simple environments such as games that can then be translated for use in broader real-life fields. Furthermore, a simple environment also provides scalability in complexity, which is valuable when testing algorithms. A recent, well-known problem in the field of AI was solving the game Go [6]. The difficulty with solving Go is due to its sheer number of possible moves, the high number of turns per game, and the impact of late-game moves on the result of the game [10]. In 2016, however, AlphaGo Zero [6] was able to break this complex barrier. The algorithm was provided with minimal prior knowledge about the game and learned good moves through self-play.

Because MCTS proved useful in Go, we chose to research MCTS in the game Dots-and-Boxes. To date, only boards up to 4x5 have been solved [5]. We implemented the slow-tree parallelized [8] MCTS for Dots-and-Boxes because it is a game similar to Go. The reason we chose to investigate performance in a closed system with simple known rules is that it can easily be compared with other performances in the same system, making it easier to determine what was bad, good, or even optimal. Then, through testing various factors and analyzing the results, we were able to determine the win-ratio performance $\frac{\text{wins}}{\text{total games}}$ in the slow-tree parallelized Monte Carlo Tree Search.

The remainder of this research paper is organized as follows. First, some background information on the following topics are given: Dots-and-Boxes (rules), general definitions/descriptions (terminology and relationships of trees and graphs), MCTS (concept and steps to a simulation), parallelization (concept and the three common types), and finally the purpose (research question). The next section describes in detail the synchronization algorithm. The next section analyzes the results of these approaches and some unforeseen difficulties. Finally, potential future work is discussed and a conclusion is provided.

2 Background

2.1 Dots-and-Boxes

Dots-and-Boxes is a two player game that starts with a rectangular array of unconnected dots [3]. The two players take turns creating an edge by joining two horizontal or vertical adjacent dots. If a player finishes a square (completes the fourth side), that same player must go again. The game ends when all boxes are taken up (no more lines can be drawn); the player who has completed the most squares is declared the winner. The board size is predetermined by the players and can be specified, for example, as 2x2 to indicate a 2 row by 2 column board of boxes. Figure 1 shows a 2x2 Dots-and-Boxes game where the red player has currently taken two boxes.

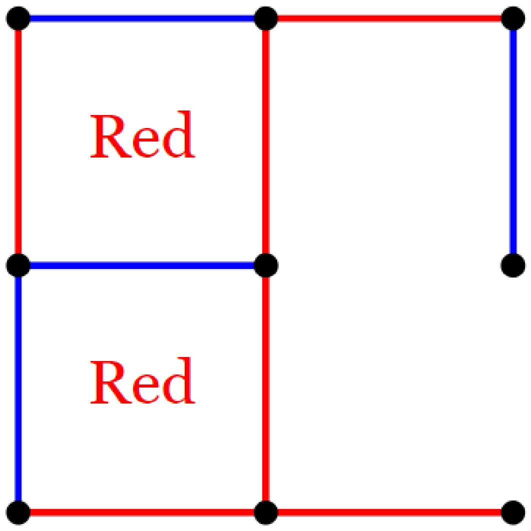


Figure 1: A 2x2 Dots-and-Boxes board where the starting player, player red, is up 2-0

2.2 Definitions/Descriptions

In order for an algorithm to learn the game Dots-and-Boxes, there must be some relationship between the game and the simulations. The first player, player 1, will first think about moves, decide on a good move, and then play it. The second player, player 2, will then think about moves, decide on a good move, and finally play it. This sequence will continue, with some potential turn-based exceptions due to game specific rules, until the game is over. The thinking process behind the move is composed of many simulations. Simulations are the computer version of a person playing out different games in his or her head, where one simulation would be just one game played out. Therefore, in order for the algorithm to learn about the game, each player will run a specific number of simulations and then make a decision in the actual game . The more simulations ran, the more accurate the results [2].

The MCTS is a tree search that is used to show the progression of games. In our implementation, we store certain information that can then be used to select good moves and play them. The variation of MCTS used in this paper uses a directed acyclic graph (DAG), which is a directed connected graph with no cycles. This means that a move in the game cannot be reversed or removed, making the game finite. The state associated with the root node (configuration of the current board) is the top/first node in the tree search. From here, the state connects to the next state(s) through separate action(s). An action is a legal move that connects one node to the next node. Taking an action will result in a new node with the previous node and the new action combined. In Dots-and-Boxes, taking an action is the addition of a horizontal or vertical edge on the board, which will combine the current node with the edge, resulting in the next node.

Nodes that do not have actions or have actions that do not connect to other states are called leaf nodes. A child node can be reached from a parent node through one discrete legal action. In a graph, there may be multiple ways to reach the same state, so a child may have multiple parents. The level of the child node is one greater than the parent node as it is one level deeper in the graph. In the context of Dots-and-Boxes, this means that level $n + 1$ has one less action available than level n as one action must have been selected to advance to the next level. Figure 2 shows an example of two parent states connecting to the same child state through separate actions.

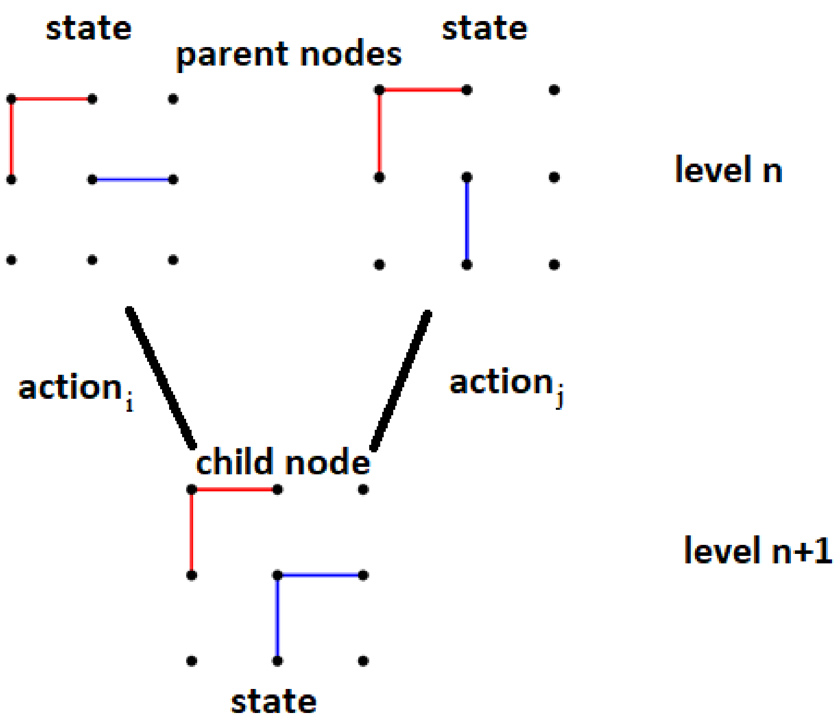


Figure 2: Two parent nodes connecting to the same child node through separate actions

A node that has multiple actions leading to it is called a transposition [1]. These are nodes that are combined into a single node, as they have the same state associated with the node. This not only helps reduce the number of nodes in the memory but also combines information learned from separate actions. For example, a 2x2 board with actions 0, 1, and 6 taken in that order is the same as actions 1, 0, and 6 taken in this order. In fact, there are 6 different orders to take these actions. In this case, there are 6 different orders that these actions can be taken in, which means that six separate states can be reduce into one state by using transpositions. A 1x1 board will drop from 16 states to 6 states, a 2x2 board will drop from 4096 to 570, and a 3x3 board drops from 16,777,216 to 2,102,800 [4]. Due to this, each simulation can do the work of many simulations, making the program more effective.

Each node keeps various information about itself and its actions to calculate its quality. In our implementation, $N_{(s)}$ is the number of times the simulations reached state "s." $N_{(s,a)}$ is the number of times the simulation used action "a" in state "s." $R_{(s)}$ is a total value of the simulation outcomes (win +1, draw +0, loss -1) in state “s.” $R_{(s,a)}$ is a total value of the simulation outcomes for action “a” in state “s.” Using $N_{(s)}$ and $R_{(s)}$, the quality of state “s,” $Q_{(s)}$, can be determined through the following equation: $Q_{(s)} = \frac{R_{(s)}}{N_{(s)}} = \frac{\text{rewards}}{\text{visits}}$. The greater the numerator (rewards), the greater the $Q_{(s)}$ value. Similarly, the smaller the numerator (rewards), the smaller the $Q_{(s)}$ value. Even though an update to rewards must result in an increment to visits by 1, an increase by 1 for both the numerator and denominator is still mathematically greater than the original value, meaning that the value of $Q_{(s)}$ will increase as expected. The quality of action “a” in state “s” is $Q_{(s,a)} = \frac{R_{(s,a)}}{N_{(s,a)}}$. The greater the $Q_{(s,a)}$ value, the better the action “a” from state “s.”

2.3 Monte Carlo Tree Search

The Monte Carlo Tree Search is a method that attempts to distinguish good decisions from bad ones in difficult learning problems by building a search tree node by node [2]. Thousands and thousands of simulations can be executed from the root node. The information learned in earlier simulations helps guide later simulations. This is done so actions that lead to undesired results do not waste resources that could have been better utilized exploring more promising actions. After a set constraint is complete, the computer weighs its options and chooses the best move in the actual game, progressing the real game forward.

Although uncertainty plays an enormous role, Kocsis and Szepesvari [7] showed that UCB (see below) converges to more reliable decisions with a larger number of simulations. Performing tons simulation for every move results in a highly effective search that can identify a good move sequence.

`One simulation in the MCTS can be broken down into four simple steps [9]:`

2.3.1 Selection

Starting at the root node, a node is recursively selected until a leaf node is reached. A decision is made on which node to select at each level by determining the node with the maximum selection quality. The Upper Confidence Bound applied to Trees (UCB1) is used to evaluate each child node in order to determine which node to select. UCB1 [8] takes into account a proper balance of exploitation and exploration. The effect that exploration has on the selection quality reflects the amount of uncertainty surrounding a specific action. The amount of uncertainty is based upon the number of simulations that a specific action receives compared to the number of times its parent node is reached. UCB1 uses an exploration constant [8] to vary the importance of exploration. The effect that exploitation has on the selection quality reflects the past success rate surrounding a specific action: moves that have a higher $Q_{(s)}$ value when compared to other actions from the state. For each action "a" from state "s," both exploitation and exploration are summed together using the equation [9], [10] $UCB1_{(s,a)} = \frac{R_{(s,a)}}{N_{(s,a)}} + C\sqrt{\frac{2\ln N_{(s)}}{N_{(s,a)}}}$, and the action with the greatest value is then selected. This process repeats until a leaf node is reached.

2.3.2 Expansion

A random legal action expands the leaf node, and consequently, connects the corresponding new node to the leaf node. This allows the tree to expand and increase in levels, discovering more potential options for future simulations. Although multiple nodes can be added to a tree, usually only one is added. Once this node is connected, the node is visited and a simulation is performed.

2.3.3 Default Simulation

A default policy is chosen to run the default simulation. There are many ways to pick the moves during a default simulation. These various ways are called the default policy. The simplest default policy, which is the one implemented in our program, is one that picks random valid moves until the game is over. Once the game ends, the result of the game is stored and backpropagation begins.

2.3.4 Backpropagation

The game result is traced up and the nodes and actions taken during the selection stage are updated. Each node taken prior to the expansion step increments its $N_{(s)}$ value by 1. Each action taken prior to the expansion step increments its $N_{(s,a)}$ value by 1 and updates its $R_{(s,a)}$ value depending on the outcome (win +1, draw +0, loss -1). After updating these values, the current simulation is complete and a new simulation starts with the newly calculated values.

2.4 Parallelization

A large number of simulations are needed to achieve reliable decisions, which can be time-consuming. The purpose of parallelization is to divide the time MCTS takes by running simulations simultaneously in hopes that the time gained outweighs the cost. This also divides the learning process. There are different types of parallelization that can be implemented for MCTS [2], [8]. Three main implementations along with their benefits and drawbacks are discussed below:

2.4.1 Slow-Tree Parallelization

The parallelization is split among process nodes. Each process node runs a separate MCTS from the same root state, resulting in different search trees. Information is synchronized periodically among all process nodes, updating the information for each process node. While slow-tree parallelization is relatively simple to implement, it may cause delays as all process nodes must wait until they reach the synchronization point. Furthermore, slow-tree parallelization can result in learning repetitive information as a process node may be learning a strategy that a different process node has already learned.

Slow-tree parallelization is the type of parallelization algorithm implemented in the program.

2.4.2 Tree Parallelization

A single tree is shared by multiple threads. Each thread works to add nodes to the tree and update information. Since the threads are updating the same tree, parts of the tree may need to be locked to ensure that separate threads do not simultaneously interfere with each other. Often an altered version of the selection algorithm known as Virtual Loss [8] is used to discourage interference. While certain implementations of tree parallelization prevent data corruption by locking portions of the tree, threads are unable to run as freely and sometimes have to settle exploring secondary actions. These other options may not be as beneficial to spend time or simulations on.

2.4.3 Leaf Parallelization

A single tree is shared with a single thread used to operate on the tree. Whenever a simulation run is required, the thread hands the simulation to another thread which runs independently. Introduced by Cazenave and Jouandeau [11], leaf parallelization is one of the simplest ways to parallelize MCTS. However, since simulations assigned earlier are completed and reused, leaf parallelization is slow as it cannot utilize high numbers of threads.

2.5 Purpose and Research Question

The purpose of this work described is to explore the question of how the amount of shared information affects the win-ratio performance of a game learned through a slow-tree parallelized Monte Carlo Tree Search. Since the size of the board can be scaled up, outcomes in simple environments can be separated distinctively. Since these results can be compared with each other, the game Dots-and-Boxes is utilized.

The following research question has been formed to guide this investigation:

Q: How does the amount of shared information affect the win-ratio performance?

The amount of shared information can be varied in four aspects:

a) How many levels of information in each tree is shared during synchronization?

b) How often are parallel process nodes synchronized?

c) How many process nodes are utilized?

d) How many simulations occur per move?

3 Method

The algorithm for parallelization utilizes slow-tree parallelization. When synchronization occurs, three levels starting from each process node's root node are synchronized. This involves $N_{(s)}$ and $N_{(s,a)}$ from levels 0, 1, and 2, and $R_{(s)}$ and $N_{(s)}$ from level 3. The information of all process nodes is combined and then redistributed to the individual process nodes. Once each process node has the updated $N_{(s)}$, $N_{(s,a)}$, and $R_{(s)}$ values, UCT3 [2] is used to calculate $Q_{(s)}$ values for states at levels 3 through 0, which are then used to calculate $R_{(s,a)}$ values for each action.

The purpose of the UCT3 algorithm is to help deal with transpositions in graphs because it appropriately allocates the amount of reward among the different actions leading to the transposition [1]. Since the quality of future states has an impact on the current state, the quality calculated for a parent node is based off a weighted distribution of its children's quality.

3.1 Synchronization Algorithm

Each node has a method to retrieve an array of the number of times its actions were visited and the new node from that action. By collecting $N_{(s,a)}$ for each action in the state, going to the child from that action, and repeating the process for three levels, the program can collect information for $N_{(s,a)}$ in levels 0, 1, and 2. If a node does not exist, the $N_{(s,a)}$ value for that node is equal to 0, indicating that it was visited 0 times. The algorithm must then calculate how many children and grandchildren actions are missing at greater levels. If we say that a node has Y children, which also means Y actions, then each one of those children will have $(Y - 1)$ children, which also means $(Y - 1)$ actions. This is because one action will be taken to connect the node to its children, and since that action cannot be played again, there will be one less option available. Knowing this information, we can calculate how many actions are missing if a child at some level does not exist. Furthermore, it also allows us to maintain a proper order in the array when combining process nodes. In order to minimize space, consecutive zeros in each array are combined together and represented as a negative number, with the value representing the number of zeros. For example, $[0, 0, 0, 5, 0, 0]$ would be represented as $[-3, 5, -2]$. This is the $combineZeros(var)$ method is the pseudocode below. Algorithm 1 shows pseudocode of the process explained above, which is implemented in the program. Note that $shareLevelX = N_{(s,a)}$ for level X.

```
input  : root node of current subtree
output: none; side effect: set global variables
        $shareLevel1, shareLevel2, shareLevel3$
1  $shareLevel0 \leftarrow$ an array of $N_{(s,a)}$ associated with root node;
2  $nact \leftarrow$ length of $shareLevel0$ (number of remaining legal actions);
3  foreach element i in shareLevel0 do
4      $child \leftarrow$ child $i$ of root node;
5      if $child = null$ then
6          $shareLevel1 \leftarrow$ append $(nact - 1)$ zeros;
7          $shareLevel2 \leftarrow$ append $(nact - 1)(nact - 2)$ zeros;
8      else
9          $shareLevel1 \leftarrow$ append array of $N_{(s,a)}$ associated with $child$;
10         foreach element j in (just appended) shareLevel1 do
11             $grandchild \leftarrow$ grandchild $j$ of child node $i$;
12             if $grandchild = null$ then
13                 $shareLevel2 \leftarrow$ append $(nact - 2)$ zeros;
14             else
15                 $shareLevel2 \leftarrow$ append array of $N_{(s,a)}$ associated with $grandchild$;
16 $combineZeros(shareLevel1)$;
17 $combineZeros(shareLevel2)$;
18 $combineZeros(shareLevel3)$;
```
Algorithm 1: Access and store $N_{(s,a)}$ for levels 1, 2, and 3 for each process node

In order to collect $N_{(s)}$ and $R_{(s)}$ for level 3, we generated all three edge combination from the remaining legal actions, and searched if the new node existed in the tree. If the node does not exist, then $N_{(s)}$ and $R_{(s)}$ are 0. If the node does exist, we sum the number of times each action was visited in the node associated with state "s" for $N_{(s)}$, using $N_{(s)} = \sum N_{(s,a)}$. We also sum the reward each action received in the node associated with state "s" for $R_{(s)}$, using $R_{(s)} = \sum R_{(s,a)}$. Algorithm 2 shows pseudocode for this process. Note that $shareLevelR3 = R_{(s)}$ and $shareLevelN3 = N_{(s)}$ for level 3.

```
input  : root node of current subtree
output: none; side effect: set global variables $shareLevelR3, shareLevelN3$
1  $comb3list$ ← an array of all possible three combinations from the remaining legal
     actions associated with root node;
2  $shareLevelN3$ ← array of $comb3List$ length;
3  $shareLevelR3$ ← array of $comb3List$ length;
4  foreach combination i in comb3list do
5      $tempNode$ ← new node of root node actions with combination i;
6      if $child = null$ then
7          $shareLevelR3$ ← append 0;
8          $shareLevelN3$ ← append 0;
9      else
10         $shareLevelR3$ ← append $R_{(s)}$ value associated with $tempNode$;
11         $shareLevelN3$ ← append $N_{(s)}$ value associated with $tempNode$;
```
Algorithm 2: Access and store $N_{(s,a)}$ for levels 1, 2, and 3 for each process node

After we combine the same index of each array together to make another array, the combined information is shared with all the process nodes. Figure 3 shows an example with two process nodes (referred to as learners in the figure) that share three levels of information, combine for shared info, and then send back to the learners.

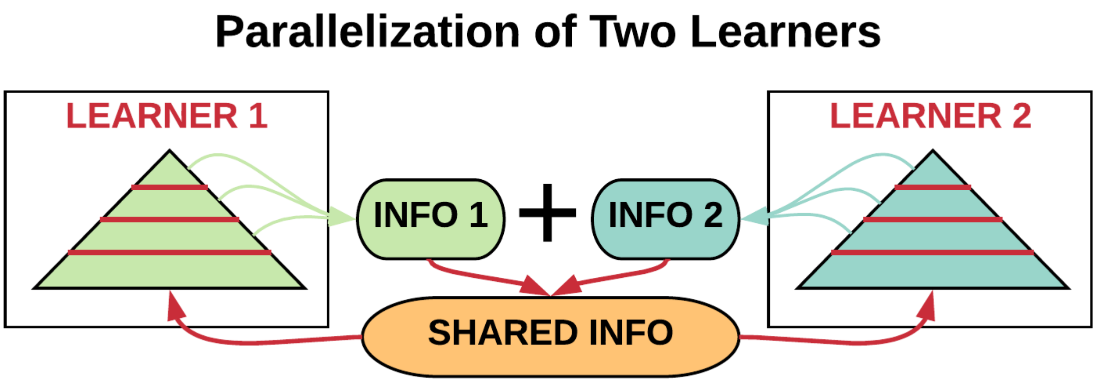


Figure 3: Two individual learners (process nodes) receiving the combined information once shared

Each individual learner computes $Q_{(s)}$ for all existing nodes in level 3, 2, and 1. $Q_{(s)}$ in level 3 is computed by dividing $R_{(s)}$ by $N_{(s)}$. The levels 2 and 1 use a weighted distribution for the percentage of times a certain action was chosen in the node being computed. In order to calculate $Q_{(s)}$ for levels 2 and 1, $N_{(s)}$ for levels 2 and 1 are also needed. This is calculated by summing up certain values from the combined $shareLevel2$ and $shareLevel1$, respectively. This fraction value is then multiplied by the $Q_{(s)}$ value of the node from the certain action. Finally, $R_{(s,a)}$ is calculated for each node in levels 3, 2, and 1 by multiplying $N_{(s,a)}$ with $Q_{(s)}$ for each action "a" in each state "s."

$Q_{(s)}$ `is calculated for each node in levels 3, 2, and 1:`

level 3 states: $Q_{(3s)} = \frac{R_{(3s)}}{N_{(3s)}}$

level 2 states: $Q_{(2s)} = \sum \dfrac{N_{(2s,a)}}{N_{(2s)}} * Q_{(3s)}$

level 1 states: $Q_{(1s)} = \sum \dfrac{N_{(1s,a)}}{N_{(1s)}} * Q_{(2s)}$

$R_{(s,a)}$ `is calculated for each node in levels 2, 1, and 0:`

level 2 states: $R_{(2s,a)} = N_{(2s,a)} * Q_{(3s)}$

level 1 states: $R_{(1s,a)} = N_{(1s,a)} * Q_{(2s)}$

level 0 states: $R_{(0s,a)} = N_{(0s,a)} * Q_{(1s)}$

In order to explain the UCT3 algorithm further, an example below is provided. The figures below assume that collecting information and receiving combined information was done properly. The first figure, Figure 4, shows the information that all process nodes currently have. The nodes are the rectangular boxes and the actions are the directed arrows. The value within the node labeled N^x_y, where x is the level and y is the position in the level, is the number of times the state "s" associated with the node was visited

$N_{(s)}$. The value at the last level within the node labeled R^x_y, where x is the level and y is the position in the level, is the reward value of the state “s” associated with the node $R_{(s)}$. The directed arrows labeled $N^x_{y,z}$, where x is the level, y is the y number of the parent node, and z is the position in y, is the number of times the action “a” in state “s” associated with the node was visited $N_{(s,a)}$. Directed arrows that come from the same parent node are the same color. Using this information, $Q_{(s)}$ for levels 3, 2, and 1 is calculated. Once all the $Q_{(s)}$ values are calculated, $R_{(s,a)}$ for levels 2, 1, and 0 is calculated. Note that the figure below does not scale the values, which would have just kept the same proportions in smaller values.

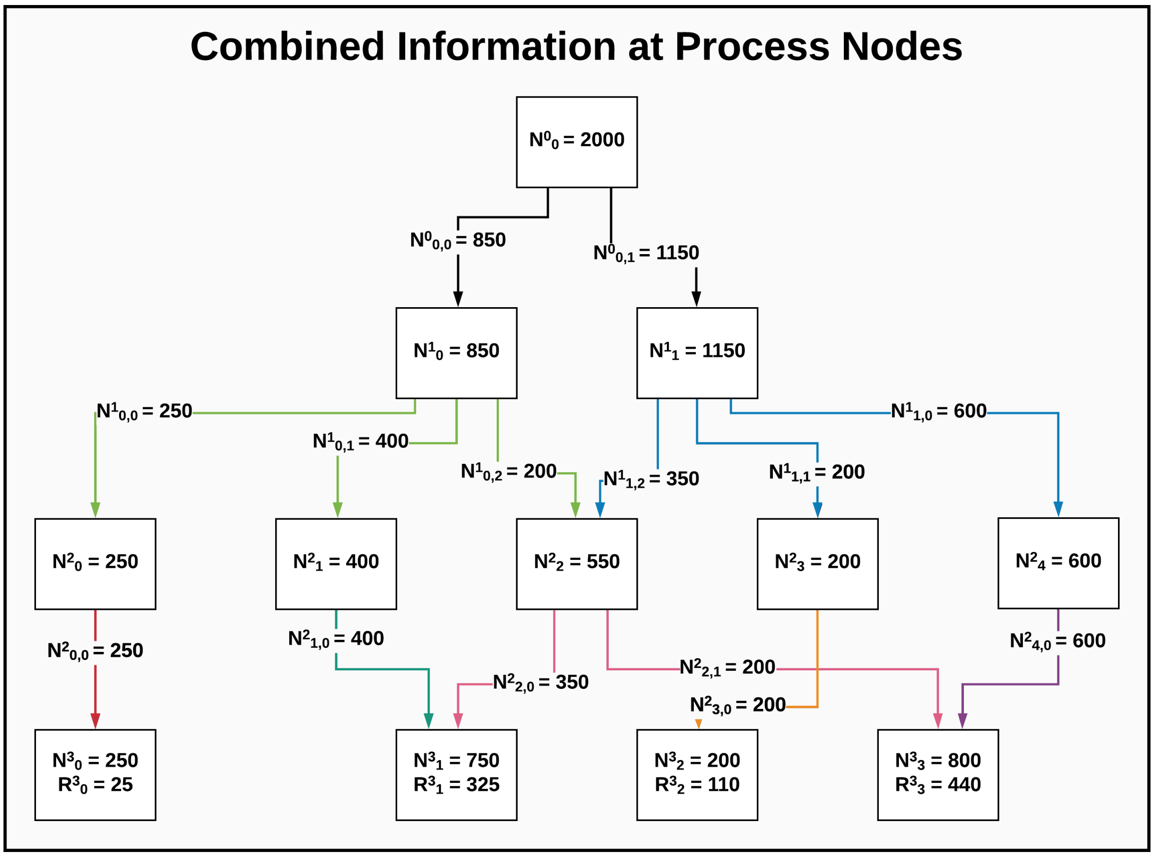


Figure 4: Process node information after receiving the combined information

The following $Q_{(s)}$ values are calculated (based on information from Figure 4) for every node in the particular level that is reachable from the root node:

$$Q_0^3 = \frac{25}{250} = 0.10 \quad Q_0^2 = \frac{250}{250} * Q_0^3 = 0.10 \quad Q_0^1 = \frac{250}{850} * Q_0^2 + \frac{400}{850} * Q_1^2 + \frac{200}{850} * Q_2^2 = 0.35$$

$$Q_1^3 = \frac{325}{750} = 0.43 \quad Q_1^2 = \frac{400}{400} * Q_1^3 = 0.43 \quad Q_1^1 = \frac{350}{1150} * Q_2^2 + \frac{200}{1150} * Q_3^2 + \frac{600}{1150} * Q_4^2 = 0.53$$

$$Q_2^3 = \frac{110}{200} = 0.55 \quad Q_2^2 = \frac{350}{550} * Q_1^3 + \frac{200}{550} * Q_2^3 = 0.48$$

$$Q_3^3 = \frac{440}{800} = 0.55 \quad Q_3^2 = \frac{200}{200} * Q_2^3 = 0.55$$

$$Q_4^2 = \frac{600}{600} * Q_3^3 = 0.55$$

Figure 5 below is similar to Figure 4, with the exception that Q_y^x, where x is the level and y is the position in the level, is shown for each state. Note that the actual implementation does not store $Q_{(s)}$ values.

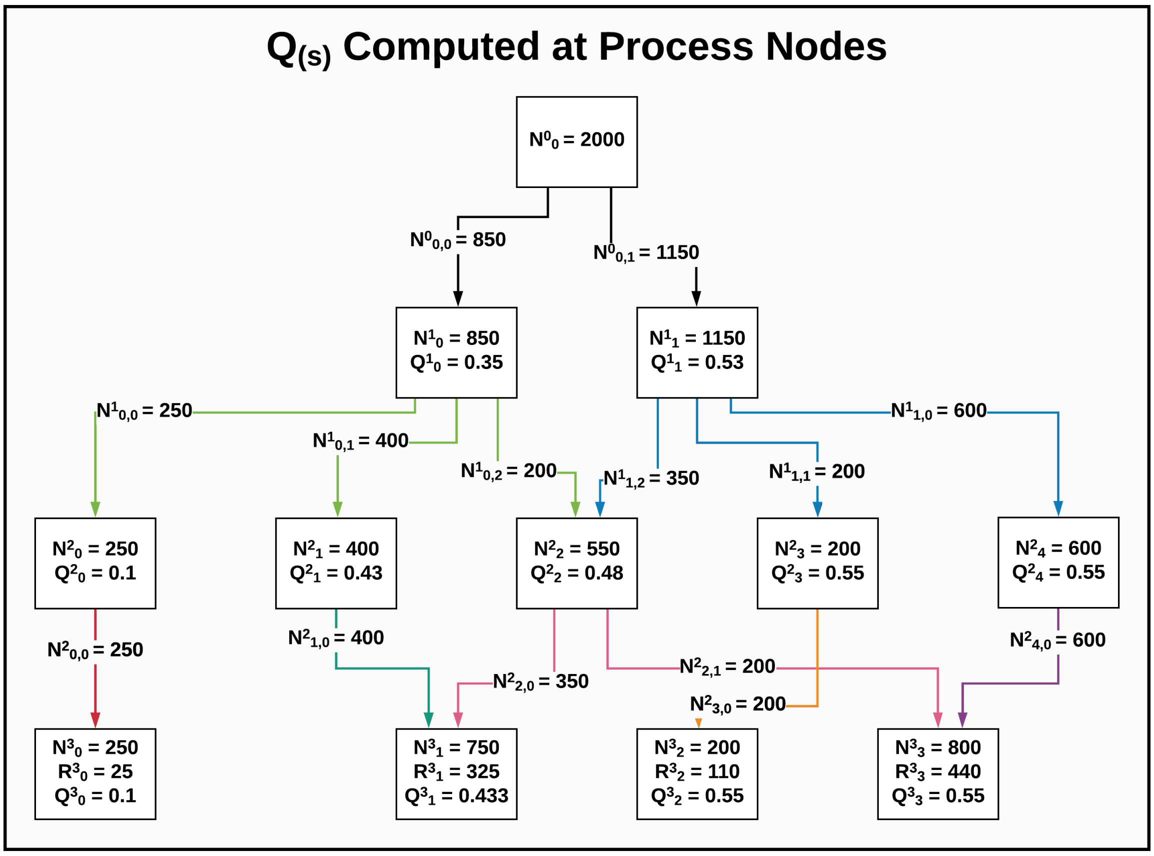
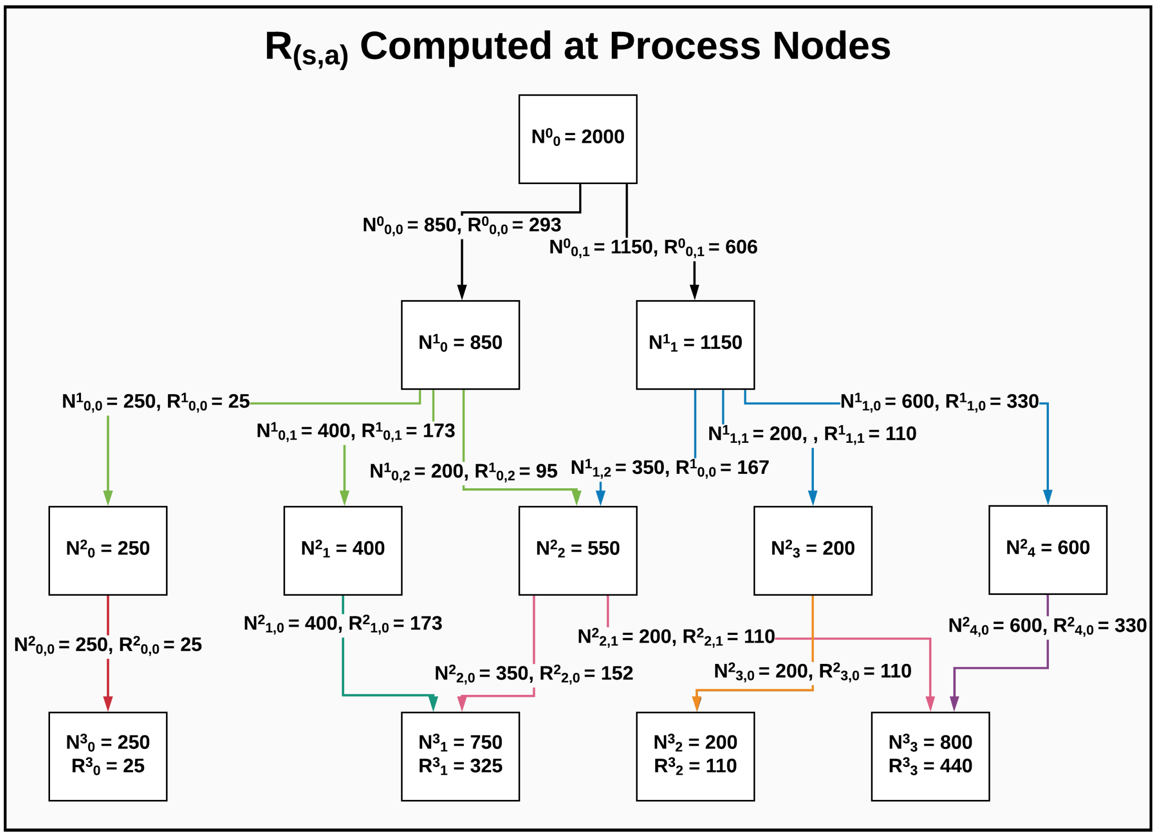


Figure 5: Process node information with computed $Q_{(s)}$ values

$R_{(s,a)}$ `values are calculated (based on information from Figure ??) for each node in levels 3, 2, and 1:`

$$
\begin{array}{lll}
R^2_{0,0} = Q^3_0 * N^2_{0,0} = 25 & R^1_{0,0} = Q^2_0 * N^1_{0,0} = 25 & R^0_{0,0} = Q^1_0 * N^0_{0,0} = 293 \\
R^2_{1,0} = Q^3_1 * N^2_{1,0} = 173 & R^1_{0,1} = Q^2_1 * N^1_{0,1} = 173 & R^0_{0,1} = Q^1_1 * N^1_{0,1} = 606 \\
R^2_{2,0} = Q^3_1 * N^2_{2,0} = 152 & R^1_{0,2} = Q^2_2 * N^1_{0,2} = 95 & \\
R^2_{2,1} = Q^3_3 * N^2_{2,1} = 110 & R^1_{1,2} = Q^2_2 * N^1_{1,2} = 167 & \\
R^2_{3,0} = Q^3_2 * N^2_{3,0} = 110 & R^1_{1,1} = Q^2_3 * N^1_{1,1} = 110 & \\
R^2_{4,0} = Q^3_3 * N^2_{4,0} = 330 & R^1_{1,0} = Q^2_4 * N^1_{1,0} = 330 &
\end{array}
$$

Figure 6 below with $R^x_{y,z}$, where x is the level, y is the position in the level, and z is the position in y, is shown for each action in each state. Note that the actual implementation does store $R_{(s,a)}$ values.



$R_{(s,a)}$ Computed at Process Nodes
$N^0_0 = 2000$
$N^0_{0,0} = 850$, $R^0_{0,0} = 293$
$N^0_{0,1} = 1150$, $R^0_{0,1} = 606$
$N^1_0 = 850$
$N^1_1 = 1150$
$N^1_{0,0} = 250$, $R^1_{0,0} = 25$
$N^1_{0,1} = 400$, $R^1_{0,1} = 173$
$N^1_{0,2} = 200$, $R^1_{0,2} = 95$
$N^1_{1,2} = 350$, $R^1_{0,0} = 167$
$N^1_{1,1} = 200$, , $R^1_{1,1} = 110$
$N^1_{1,0} = 600$, $R^1_{1,0} = 330$
$N^2_0 = 250$
$N^2_1 = 400$
$N^2_2 = 550$
$N^2_3 = 200$
$N^2_4 = 600$
$N^2_{0,0} = 250$, $R^2_{0,0} = 25$
$N^2_{1,0} = 400$, $R^2_{1,0} = 173$
$N^2_{2,0} = 350$, $R^2_{2,0} = 152$
$N^2_{2,1} = 200$, $R^2_{2,1} = 110$
$N^2_{3,0} = 200$, $R^2_{3,0} = 110$
$N^2_{4,0} = 600$, $R^2_{4,0} = 330$
$N^3_0 = 250$ $R^3_0 = 25$
$N^3_1 = 750$ $R^3_1 = 325$
$N^3_2 = 200$ $R^3_2 = 110$
$N^3_3 = 800$ $R^3_3 = 440$

Figure 6: Process node information with computed $R_{(s,a)}$ values

4 Results

We compared the treatment group (parallelized MCTS algorithm) to the controlled group (non-parallelized MCTS algorithm) for various numbers of simulations. In both groups, player one (the first player) played against a 5,000 simulations player (player two) on a 2x2, 3x3, and 4x4 board. We also kept track of the percentage of times an outside edge in the 2x2 board was played. We track the percentage of times an outside edge was taken because from previous research we know that this is the most optimal first move in a 2x2, and we can have another measure for how good our algorithm is based on this pick percentage. In the $UCB1$ formula, which combines both exploitation and exploration, the exploration constant (c-value) was 1.0. This means that exploration was weighted heavily in the algorithm, so simulations were pushed to explore much more often. Player 1 was tested at the following simulations: 100, 1k, 2k, 5k, 10k, 20k, 50k, 100k, 200k, 400k, 1m, 3m, and 5m. Each test consisted of 50 matches.

4.1 Non-Parallelization Performance

The non-parallel (base-line) performance was essential when determining the parallelization performance. In the Figures 7 and 8 below, the x-axis is the number of simulations, whereas the y-axis is the win-ratio percentage. The win-ratio percentage is calculated by the number of wins over the number of games. This means that ties and losses contribute equally to the win-ratio performance. Since most of the interesting learning occurred before 50k simulations, the graph is split into two. The first graph ranges from 100 to 50k simulations, whereas the other graph ranges from 50k to 1 million simulations. There appeared to be little significant difference as the number of simulations increased past the 1 million simulations mark. Each graph also has 4 lines plotted: 2x2 win-ratio, 3x3 win-ratio, 4x4 win-ratio, and 2x2 good first moves. The NxN win-ratio is the win-ratio percentage for the NxN board. The 2x2 good first moves graphs the percentage of times an outside edge was selected.

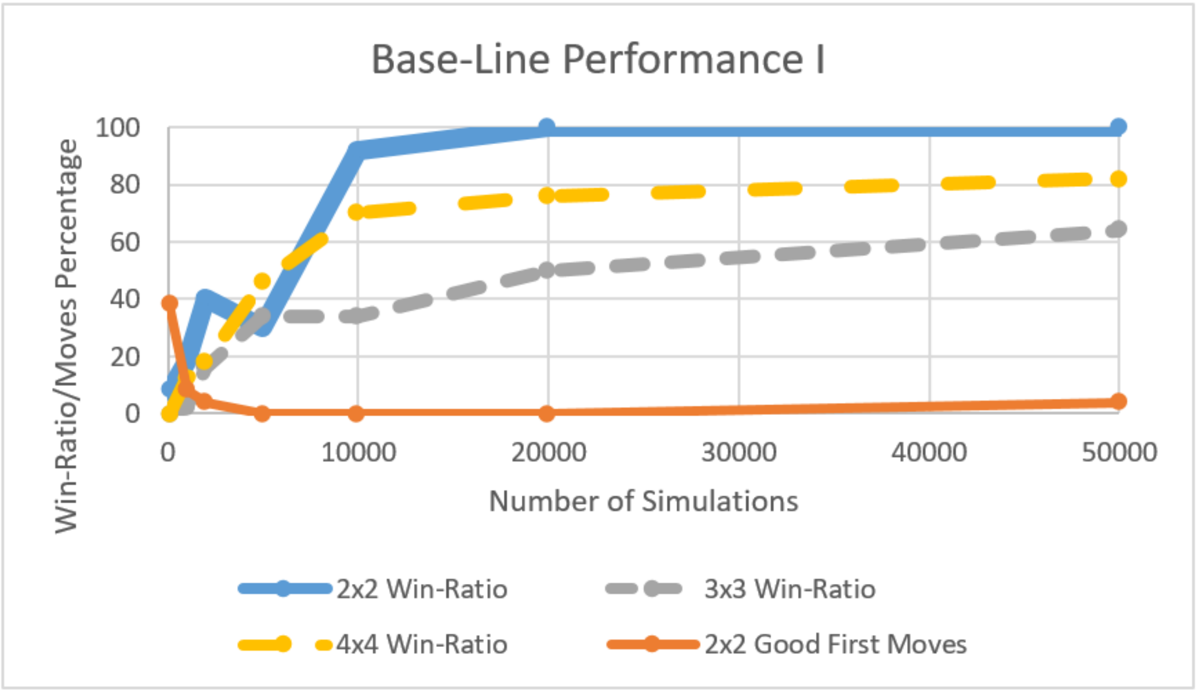


Figure 7: Non-Parallel performance I for 2x2, 3x3, and 4x4 boards

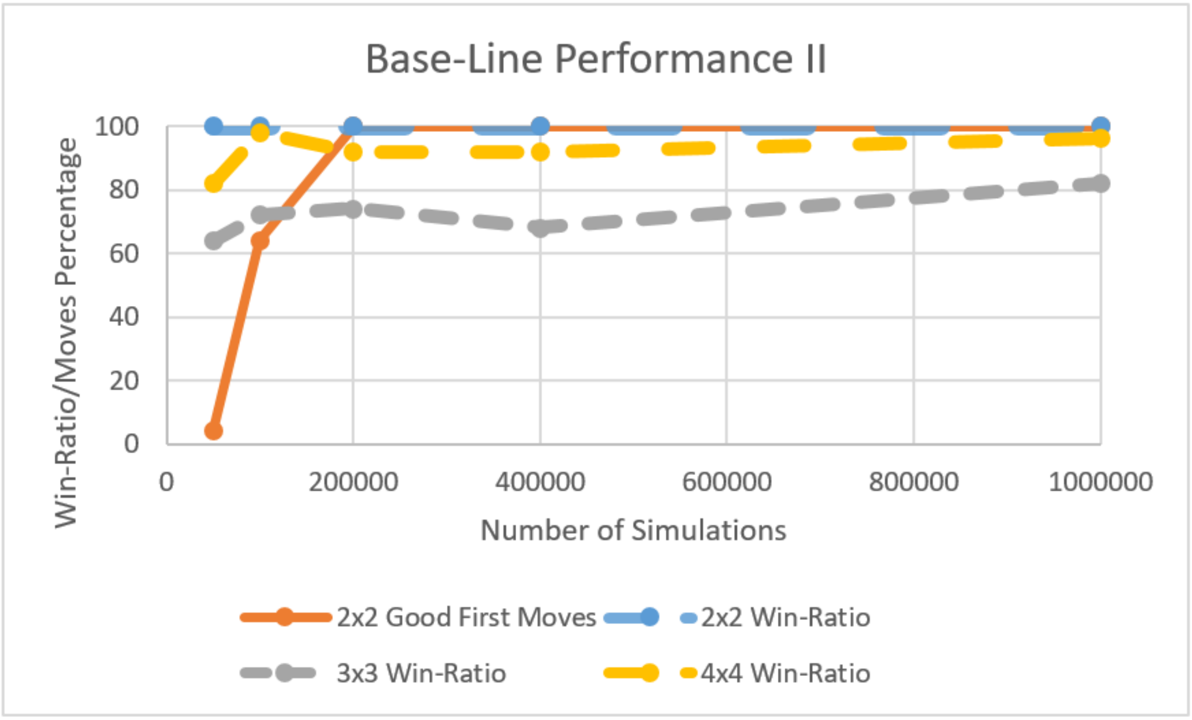


Figure 8: Non-Parallel performance II for 2x2, 3x3, and 4x4 boards

In Figures 7 and 8, the win-ratio percentage mostly increased as the number of simulations increased. There are at certain points where an increase in the number of simulations actually resulted in worse performance. These occur at 5k simulations in the 2x2 board, 40k simulations in the 3x3 board, and 200k simulations in the 4x4 board. From previous research, we know that when both players play optimally, the first player wins in a 2x2 board, loses in a 3x3 board, and ties in a 4x4 board. Therefore, it makes sense that the 2x2 performed the best, the 3x3 board performed the worst, and the 4x4 board performed better than the 3x3 board but worse than the 2x2 board. Even though the first player is supposed to lose in a 3x3 board, player two did not have enough simulations to play optimally, so player one still won games. Similarly, even though a 4x4 board is a tie when played optimally, the first player for simulations greater than 5k is stronger than its opponent, causing the win-ratio to surpass 50%.

The percentage of times a good first move was picked decreased from 40% to 0%, and then quickly increased to 100%. This means that the algorithm initially learned to pick an inside edge, and even though the win-ratio percentage continued to increase, after 50k simulations, the algorithm quickly learned to pick an outside edge. This is most likely because there are 8/12 outside edges, so the probability of picking one with little amounts of learning is likely.

4.2 Parallelization with Symmetry

When we observed not only a significant decrease in the parallelization performance but also no apparent trend as the number of simulations increased for each board size, we considered other complications with our initial solution. One potential solution may lie when accounting for symmetrical boards.

Symmetrical board states are game states that can be rotated or flipped to reach another different state. Even though the configuration of the board is different, the quality of the board state is the same. Therefore, since each process node learns independently, the overall learning may not benefit as process nodes learn different but equivalently good moves. In some cases, the process nodes when combined may impair each other's learning process. In Figure 9, there are eight different but symmetric 2x2 board configurations.

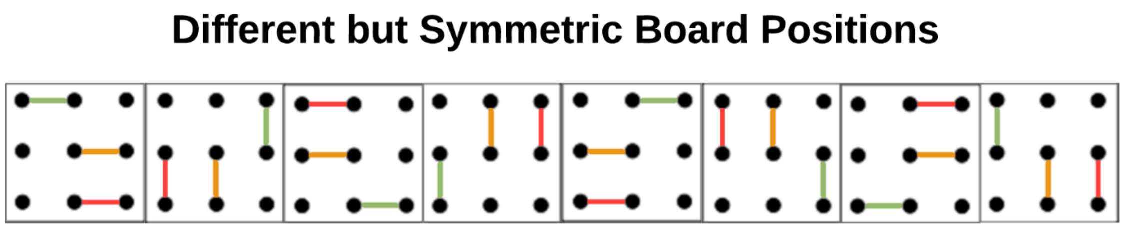


Figure 9: 2x2 Dots-and-Boxes boards with the same quality but different configurations

In Figure 10 we notice how two process nodes when combined could improve or diminish the quality of individual moves. The bar with a spectrum of color shows our representation of good moves (turquoise) and bad moves (strawberry red). On the first row of boards, the two process nodes (INFO 1 and INFO 2) have each learned a good move. When the information is shared and combined, however, the good moves are lost. In the next row of boards, the first board in rotated by 180 degrees. Combining this symmetric board with the original second board, the good moves are preserved.

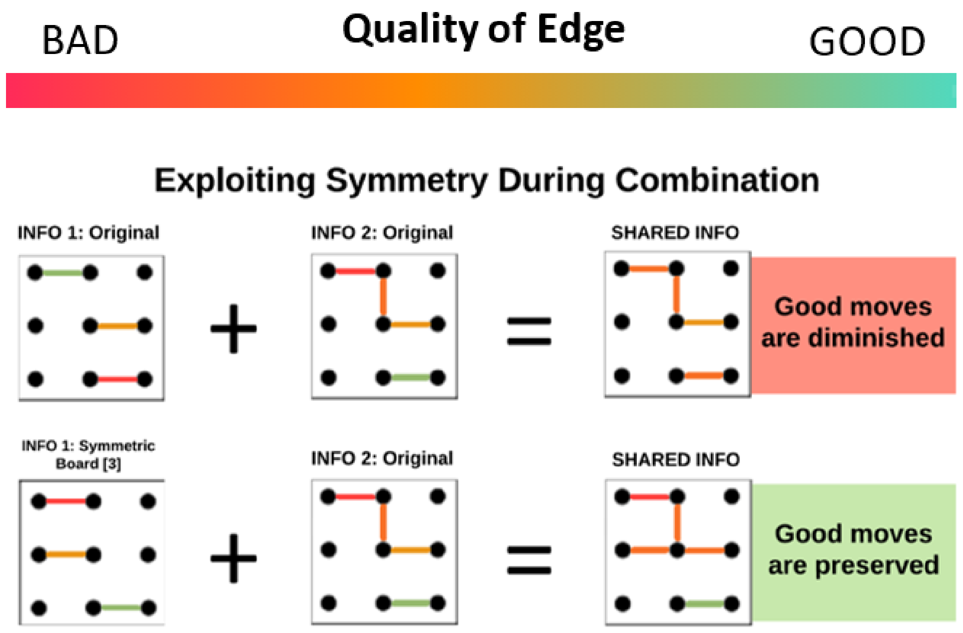


Figure 10: Utilizing symmetry to combine the independently learned information

5 Conclusion/Discussion

Due to the sheer high number of legal moves, high number of turns per match, and the impact of late-game moves on the match outcome, Dots-and-Boxes presents many challenges for artificial intelligence players. Monte Carlo Tree Search is meant to be an approach to the problems that arise in games like Dots-and-Boxes.

The non-parallel performance resulted in an overall increase in win-ratio performance as the number of simulations increased. At some specific points, there were dips in learning, meaning that the win-ratio actually decreased as the simulations increased. As expected, the 2x2 dominated the 3x3 and 4x4 win-ratio, and the 4x4 dominated the 3x3 win-ratio. Surprisingly, the 2x2 good first move (outside edge) initially decreased from 40%. In fact, player one did not learn that an outside edge was better until after 50k simulations.

Independent process nodes learn different aspects of the overall task with some repeated information, and combining it is similar to one learner learning the different aspects using more simulations. Parallelism is predicated on the idea of consistently increasing performance with more simulations. A decrease in performance with increased simulation indicates that the selection of the number of simulations for the current situation for each learner must be made with care.

Symmetry is an important factor to take into account when dealing with parallelization. There are a few preliminary solutions that could help deal with symmetry with varying efficiency. One solution is to rotate and flip the boards into all possible symmetric states, combine the results, and select the action with the best quality in all combination results. While this is the simplest solution, it is not very inefficient. Another solution is to translate the board into a canonical state, a common representation of the board. While this is more difficult, it is also more efficient.

Besides working on the symmetry, future work also needs to be conducted on the results. This includes investigating other ways that the amount of shared information can be varied. More specifically, results on the number of process nodes utilized, frequency of sharing for synchronization, and the number of levels of information synchronized needs to be collected. Once these are collected, the statistical significance should also be calculated. Furthermore, an analysis of the time requirement for the sharing and combining is needed, and an efficient algorithm to deal with symmetry must be discussed.

6 References

[1] Benjamin Childs, James Brodeur, Levente Kocsis, et al., *Transposition and Move Groups in Monte Carlo Tree Search*, 2008 IEEE Symposium on Computational Intelligence and Games, 2008.

[2] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, *A Survey of Monte Carlo Tree Search Methods*, Computational Intelligence and AI in Games, IEEE Transactions on, 2012, pp. 1–25.

[3] Elwyn Berlekamp, *The Dots and Boxes Game: Sophisticated Child's Play*, Peters, 2006.

[4] Prince, Jared, *Game Specific Approaches to Monte Carlo Tree Search for Dots and Boxes*, Honors College Capstone, Experience/Thesis Projects, paper 701, 2017.

[5] Joseph Barker and Richard Korf, *Solving 4x5 Dots-and-Boxes*, AAAI'11 Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence, San Francisco, 2011, pp. 1756–1757.

[6] Julian Schrittwieser, David Silver, and Karen Simonyan, *Mastering the game of Go without human knowledge*, Nature, 2017.

[7] Levente Kocsis and Csaba Szepesvari, *Bandit based Monte-Carlo Planning*, European Conference Machine Learning, 2006, pp. 282–293

[8] N. Septon, P.I Cowling, E.J Powley, and D. Whitehouse, *Parallelization of Information Set Monte Carlo Tree Search*, IEEE Congress on Evolutionary Computation, unknown, 2014.

[9] Sylvain Gelly and David Silver, *Monte-Carlo Tree Search and Rapid Action Value Estimation in Computer Go*, Artificial Intelligence, 2011, pp. 1856-1875.

[10] Sylvain Gelly, Marc Schoenauer, MichÃ¨le Sebag, Olivier Teytaud, Levente Koc, et al., *The grand challenge of computer Go: Monte Carlo tree search and extensions*, Communications- ACM, Association for Computing Machinery, 2012, pp. 106–113

[11] Tristan Cazenave and Nicolas Jouandeau, *On the parallelization of UCT*, Proceedings of the Computer Games Workshop 2007, 2007, pp. 93–101.

A Terminology Glossary

Node: A structure that contains certain values

State: A legal board position that can be achieved during the game

Action: A legal move that connects one state to the next state

Edge: A legal action in the game Dots-and-Boxes

Root Node: The current state (configuration of the current board)

Level: 1 + the number of edges between the node and the root node

Leaf Node: States that do not have actions or have actions that do not connect to other states

Parent Node: The board position that is one level above relative to its child node

Child Node: The board position that has one discrete legal action more relative to its parent node

Directed Acyclic Graph: A directed connected graph where the game is finite and an edge cannot be removed

Transposition: A node that has multiple actions leading to it

Rewards: The total of the wins (+1), draws (+0), losses (-1)

$\mathbf{R_{(s)}}$: The reward at state "s": $R_{(s)} = \sum R_{(s,a_i)}$

$\mathbf{R_{(s,a)}}$: the reward at action "a" in state "s"

$\mathbf{N_{(s)}}$: The number of times state "s" is visited: $N_{(s)} = \sum N_{(s,a_i)}$

$\mathbf{N_{(s,a)}}$: The number of times action "a" in state "s" is visited

$\mathbf{Q_{(s)}}$: The quality of state "s": $Q_{(s)} = \frac{R_{(s)}}{N_{(s)}}$

$\mathbf{Q_{(s,a)}}$: The quality of action "a" in state "s"

Win-Ratio Performance: $100 * \frac{\text{Wins}}{\text{Total Matches}}$

Simulations: The computer playing out games for the purpose of acquiring some type of data

Process Nodes: The individual learners that use the parallelized MCTS algorithm

Master Node: A special process node that interacts with all the individual process nodes

Parallelization: Divide the learning process among various process nodes on a high-performance computing platform to run simulations simultaneously

Synchronization: UCT3 algorithm; cycle of combining parallelized information to calculate new $R_{(s,a)}$ values