12-2020

Neural Network Development in an Artificial Intelligence Gomoku Program

David Garcia
The University of Texas Rio Grande Valley

NEURAL NETWORK DEVELOPMENT
IN AN ARTIFICIAL INTELLIGENCE
GOMOKU PROGRAM

A Thesis

by

DAVID GARCIA

Submitted to the Graduate College of
The University of Texas Rio Grande Valley
In partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

December 2020

Major Subject: Computer Science

NEURAL NETWORK DEVELOPMENT IN AN ARTIFICIAL INTELLIGENCE GOMOKU PROGRAM

A Thesis
by
DAVID GARCIA

COMMITTEE MEMBERS

Dr. Zhixiang Chen
Chair of Committee

Dr. Bin Fu
Committee Member

Dr. Andres Figueroa
Committee Member

Dr. John Abraham
Committee Member

December 2020

ABSTRACT

Garcia, David, Neural Network Development in an Artificial Intelligence Gomoku Program. Master of Science (MS), December, 2020, 46 pp., 13 figures, 7 references.

The game of Gomoku, also called Five in a Row, is an abstract strategy board game. The Gomoku program is constructed upon an algebraic monomial theory to aid values for each possible move and estimate chances for the artificial intelligence program to accomplish a winning path for each move and rounds. With the utilization of the monomial theory, winning configurations are successfully converted into monomials of variables which are introduced to represent board positions. In the artificial intelligence program, an arduous task is how to perform the present configuration of the Gomoku game along with the past moves of the two players. The monomials utilized can generate the artificial intelligence to efficiently interpret the current state and the history of the game. They can also acquiesce the artificial intelligence to generate the potential values for future actions from the present state and history of decisions made by the individuals. In extension, the Monte Carlo Tree Search pertaining to the monomial theory has been implemented to examine an achievable winning approach for the artificial intelligence. The particular monomials aid to reduce the search capacity in order to benefit estimate rates for analysis of the historical moves and analysis of the future actions. The artificial intelligence developed for our Gomoku program with algebraic monomial theory is efficient and highly competitive. In this current version of our program, the artificial intelligence can defeat its predecessor and defeat the top rated AI (Wine) which is ranked 7th in the Gomocup rankings.

ACKNOWLEDGMENTS

Thank you, Dr. Zhixiang Chen, chair of my thesis committee, for being persistent and mentoring me throughout my graduate studies. In addition, thank you, my committee members: Dr. Bin Fu, Dr. Andres Figueroa, Dr. John Abraham.

TABLE OF CONTENTS

Page

LIST OF FIGURES

Page

CHAPTER I

INTRODUCTION

1.1 The Origin of Gomoku

The game of Gomoku, also called Five in a Row, is an abstract strategy board game. The Gomoku board game was brought to Japan around 270BC, with the name of Kakugo (meaning preparedness, resolution and readiness). Japanese chronicles indicated at the time of the late 17th and early 18th century, Gomoku was at its height of popularity, being played by young and old alike. The first modern volume of the board game - called Kakugo appeared in 1858. The name "Gomoku" is from the Japanese language, in which it is referred to as gomokunarable. Go means five, moku is a counter word for pieces and narabe means line-up. The game is popular in Korea where it is called omok which has the same structure and origin as the Japanese name. In the nineteenth century, the game was introduced to Britian where it was known as Go Bang. The ancient strategy board game is more than 4000 years old and that its rules were developed in ancient China. But historians have indicated that the board game with the same rules have been found in ancient Greece and pre-Columbian civilizations of America.

1.2 Background

Gomoku utilizes two players, indicated by different stone colors (black or white), alternating turns placing stones on a checkerboard, using a 9×9 or 15×15 board, or a 19×19 board. One player uses the white stones and the other, black. The players take turns placing the stones on the vacant intersections of a board. Once placed on the board, stones may not be moved. In order to win, a player must first form his/her stones into an unbroken line of five horizontally, vertically, or diagonally. Despite its simple rules Gomoku is a strategy and logic board game, more complex and

difficult than Tic Tac Toe, Chess, and Connect Four.

1.2.1 Computers and Gomoku

Researchers and practitioners have been applying artificial intelligence techniques on playing gomoku for decades. In 1994, Victor Allis [5] developed the algorithms of proof-number search (pn-search) and dependency based search (db-search), and proved that when starting from an empty 15 15 board, the first player has a winning strategy using these searching algorithms. Victor's winning strategy applies to both free-style gomoku and standard gomoku without any opening rules. It indicates that the black player can win, in theory, on larger boards too. The computer olympiad began with the gomoku game in 1989, but gomoku has not been in the list since 1993. The Renju World Computer Championship was started in 1991, and held for 4 times until 2004. The Gomocup tournament is played since 2000 and taking place every year, still active now, with more than 30 participants from about 10 countries. The Hungarian Computer Go-Moku Tournament was also played twice in 2005. Not until 2017 were the computer programs proved to be able to outperform the world human champion in public competitions. In the Gomoku World Champion 2017, there was a match between the world champion program Yixin and the world champion human player Rudolf Dupszki. Yixin won the match with a score of 2 - 0.

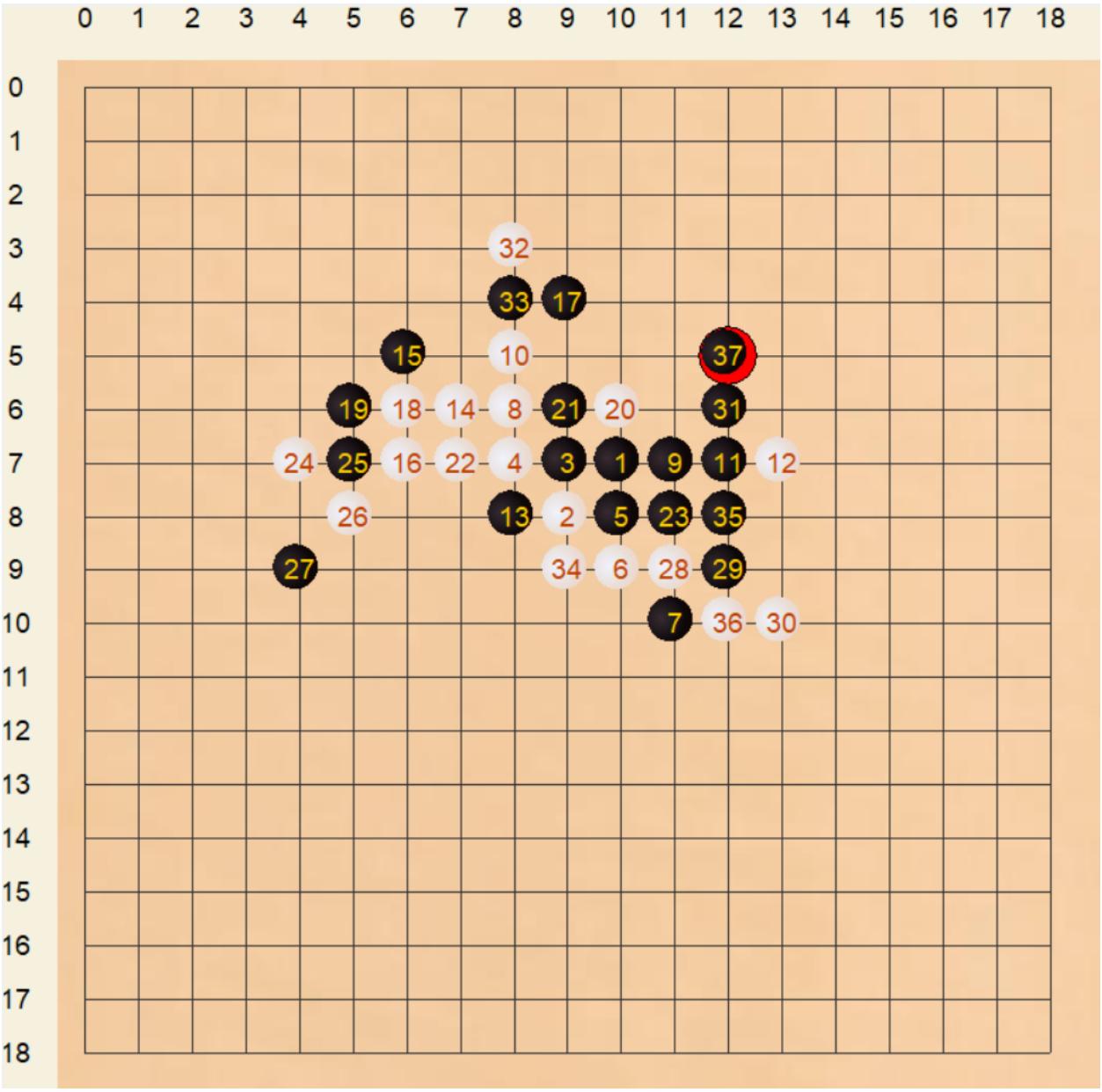

Figure 1.1: Example Board of Gomoku: Black Wins

CHAPTER II

METHODOLOGY

2.1 Monte Carlo Tree Search

The Monte Carlo Tree Search (MCTS) is a method for finding optimal decision in a given domain by taking random samples in the decision space and building a search tree according to the outcomes. It is a probabilistic and heuristic driven search algorithm that combines the classic tree search implementations alongside machine learning principles of reinforcement learning. In a tree search there are always possibilities that the current best action is actually not the most optimal action. For MCTS algorithm [6] it becomes useful as it continues to evaluate other alternatives periodically during the learning phase by executing them, instead of the currently perceived optimal strategy. This is called the "exploration-exploitation trade-off". MCTS not only exploits the actions and strategies that are found to be the best till now but also continues to explore the local space of alternative decision and finds out if they can replace the current best.

Exploration helps in exploring and discovering the unexplored parts of the tree, which results in finding a more optimal path. Exploration is useful to ensure that MCTS is not overlooking any potentially better paths. But it can quickly become inefficient in situations with large number of steps or repetitions. To avoid that, it is balanced out by exploitation. Exploitation sticks to a single path that has the greatest estimated value. This is a greedy approach that will extend the tree's depth more than its breadth. For this characteristic, MCTS becomes useful in making optimal decisions in Artificial Intelligence (AI) problems.

2.1.1 Monte Carlo Tree Search Algorithm

In MCTS, the nodes are the building blocks of the search tree. These nodes are generated based on the outcome of a number of simulations. The process of the Monte Carlo Tree Search can be broken down into iterations of four distinct steps or phrases: Selection, Expansion, Simulation and Backpropagation. These four phases will be performed repeatedly until the terminal state such as win or timeout is reached. The process of the iterations can be explained as follows [6]:

1. *Selection*: In this step, the MCTS algorithm traverses the current tree from the root node using a specific strategy. The strategy uses an evaluation function to optimally select nodes with the highest estimated value. MCTS uses the Upper Confidence Bound formula applied to trees as the strategy in the selection process to traverse the tree. It balances the exploration-exploitation trade-off. During the tree traversal, a node is selected based on parameters that return the maximum value. The parameters are then characterized by the formula that is typically used for this purpose is given in Figure 2.1. When traversing a tree during the selection process, the child node that returns the greatest value from the above equation will get selected. During traversal, once a leaf node is selected, the MCTS moves into the expansion step.

2. *Expansion*: Expanding is used to increase the options further in the game by expanding the selected node into its children nodes. These children nodes are the future moves that can be played in the game.

3.*Simulation*: From a newly added state, a simulation is then run based on a searching policy called default policy. The default policy selects the next action randomly, and is performed repeatedly until a terminal state is met.

4. *Backpropagation*: The result achieved by the simulation phase is transferred into a reward value, and is back propagated through selected states in the selection phase to update their statistic values.

MCTS performs two kinds of search policies, which have different usages and are the key point in the algorithm. When all of the potential next actions are visited states, based on the statistical data stored in those states, the tree policy performs a calculation to determine the next

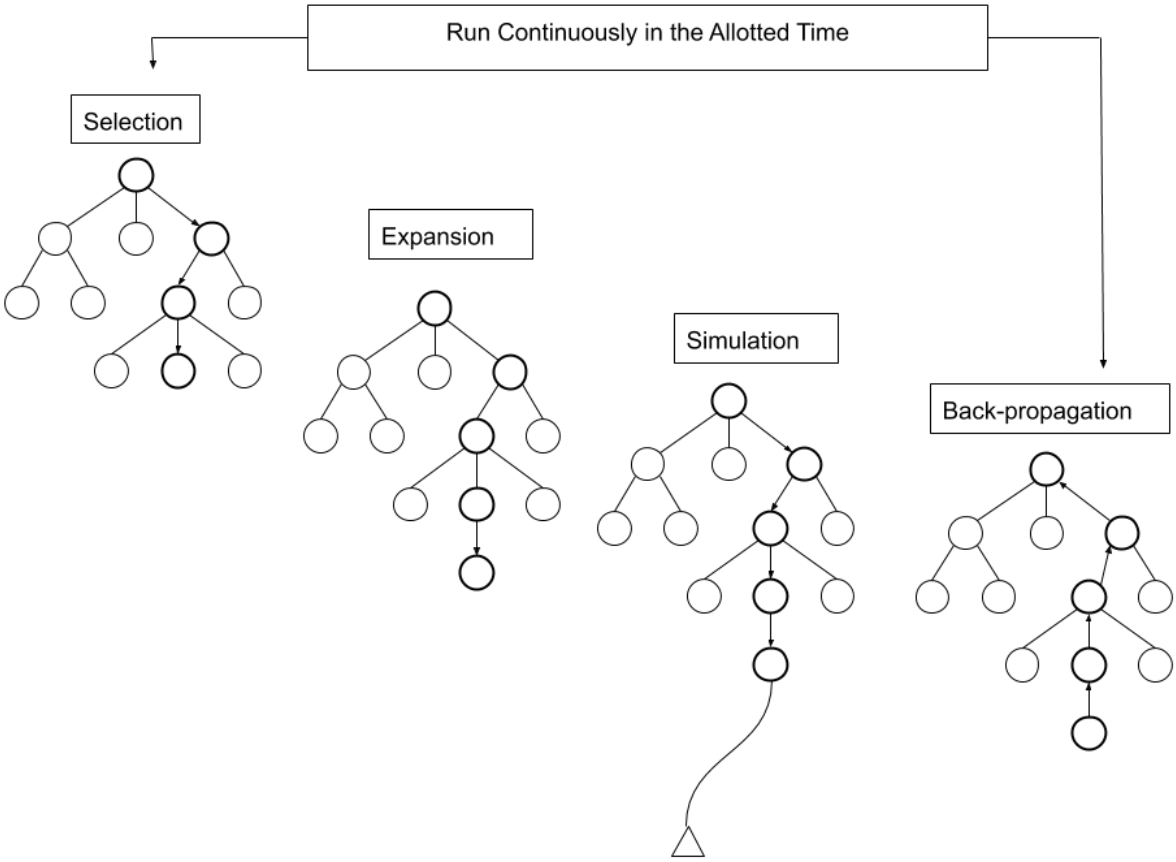

Figure 2.1: Monte Carlo Tree Search Algorithm

state. When the statistical data are not available, which means that the current state has un-visited children, the default policy will pick the next state randomly without performing any calculation.

MCTS has two advantages in the field of machine learning for playing games. Firstly, it is able to improve the performance of the computer game player through searching only a partial game tree instead of building a whole game tree. The latter will be inefficient and in some cases impossible due to the game's space complexity. Second, it is estimated that the value will become more accurate with the increase of the simulation times and nodes accessed, which indicates that the MCTS has the potential ability to improve itself. With more testing the MCTS is able to improve on itself to defeat higher competitors. MCTS has its own limitations, it relies on the randomness too much and is not efficient enough. Without the help of neural networks, MCTS cannot exhaust its maximum potential power when running a simulation.

These types of algorithms are particularly useful in turn based games where there are no element of chance in the game mechanics, such as Tic Tac Toe, Checkers, Connect 4, Chess, Go, etc. This type of algorithm is utilized by Artificial Intelligence Programs like AlphaGo by DeepMind (owned by Google and Alphabet Inc.) [6]. AlphaGo is the first computer program to defeat a professional human Go player, the first to defeat a Go world champion, and is arguably the strongest Go player in history.

2.1.2 Monte Carlo Tree Search Implementation

As mentioned previously the MCTS was succesfully used in AlphaGo [6]. The type of MCTS is also included in AlphaGomoku. This algorithm was implemented into our system to assist with decision making to help examine the offense and defense strategies utilized in matches. The MCTS is approximately identical to the one refined by AlphaGo, but with one missing item. A policy value network was not able to function by the deadline but a skeleton program was developed. Therefore, in this current stage, the program is inadequate of learning from past experiences to better guide the MCTS. Thus, human knowledge of the game has been integrated into the decision-making process to assist with MCTS to search for proficient strategies for offense or defense plays. During the search process, a search tree is assembled. Each node serves as a state the game board may

reside in. The children nodes produce states that can be reached from the parent node within one turn. The nodes are scored based on how many times they are frequent along with a reward value to identify how strong the node may perform when compared to its visit count. Each node can be classified based on whether it resides on an even or odd layer of the tree.

The root resides on layer zero, characterizing the state of the board when the MCTS is proposed. Afterward, all odd layer nodes for the AI's turn to play, while all even layer nodes for the opponent's turn. When a roll-out simulation arises, the result ends in one of two states: If a player is triumphant a score of +1 is provided to the current leaf node and back propagated to the root. This score alters each player's node abnormally as the score flips with each node alternatively layer by layer. In this situation, all winning player's nodes along the path will gain a score of +1, while all opponent's nodes along the path will receive a penalty score of –1. If the maximum depth for a roll-out simulation is reached, the result is treated as a tie, leading to all scores remaining unchanged for the nodes visited during the back propagation.

The only key difference is that the policy value score is treated as 1 in our approach due to the lack of a policy value network. Along with this setting, the MCTS is initialized with an exceedingly low temperature value to promote exploration for the initial 50 roll-out simulations. Afterwards, the temperature is raised, allowing all future simulation outcomes to be based primarily on the mean action value.

In its entirety, the implemented system functions with the help of two MCTS components. It performs simulations to find the best achievable strategy until a time limit is reached. Its counterpart, denoted as the winning MCTS, focuses on analyzing whether a winning strategy is displayed within the board. These two MCTS components function primarily in its present condition but differ in node selection for expansion, the winning MCTS tends to shorten nodes which fail to guarantee victory.

The winning MCTS has a selection of changes to perform its task accurately. The primary adjustment is the moves considered. Due to the nature of the game if one considers a wide array of options, then it would be impractical to simulate all the paths within a reasonable amount of

time, much less so if one wants to guarantee a successful outcome. With the goal of seeking for secured wins the focal point is to concentrate on the moves which force the challenger to react defensively. These actions, in most cases, would leave the opponent with 1 to 3 possible options. As the simulations progress nodes are shorten based on which player they attribute to. For a node representing the AI's move, the requirement is that all its children nodes contain a direction toward victory. This can be seen as encouraging that, no matter how the opponent may react with, there is a list of moves that would lead the AI to win. For the challengers nodes, on the hand, the requirement is that at least one child node contains a path to win. Nodes which fail to meet these requirements are cut from the tree. This appears in either a tree containing a path to win no matter how the opponent may react with, or a tree with no nodes left, implying that no guaranteed strategy to win has currently been found. Due to the extremely limited number of possible moves, this version of the MCTS find its conclusion on average within 2 seconds. This version is used to allow the system to identify when the main MCTS is no longer required, allowing for all prospect moves to be based solely on the output tree.

The essential MCTS, functions as a typical MCTS. It performs simulation after simulation until the maximum amount of time is reached. Once this develops, a point is chosen based on whichever it contains the maximum mean action value. As conventional for any MCTS, not all moves are considered. A few moves are selected based on assessment statistics of the AI's and the opponent's monomials. This effectively ignores moves with less importance, such as those with no nearby neighbors occupied. This selection procedure for potential moves is used all through tree node expansion in order to minimize the amount of resources spent on undesirable moves.

The primary and winning MCTS components collaborate with each other to find the most efficient strategy of winning. While the primary MCTS can administer an idea about which move boasts the best probability of winning, yet it is not adequate of considering all options to guarantee the winning outcome. This is crucial for the game of Gomoku because of its relatively short game length. A game can be set in stone as early as turn 9. Failing to win at the earliest chance simply leaves the opponent with more openings to win. The primary MCTS is called on average 4 to 8

times per game. At each turn, if the winning MCTS fails to find a guaranteed strategy of winning, the primary MCTS is called. This is duplicated until a winning path of moves is found, allowing for all future moves to be performed instantly. The implemented structure is able to perform, for every second, nearly 200 simulations of roll-out expansion with a maximum depth of 30 moves. With a limit set to 1 minute per turn, the AI can decide its final choice of moves that are backed by over 10,000 simulations.

Thousands of simulations are examined but a issue still exists with the final decision. Most considerably, when playing against the top ranked AI's, losses still appear for the AI. The losses are resulted from cases where multiple moves appear to have similar success rates. Such cases tend to lead the simulations to be split equally for those individual moves, such that not enough simulations are done on those moves to accurately assess which moves are truly more beneficial. Similarly, in some cases the lack of sufficient simulations may result in choosing a move with flaws because of the lack of exploration to certain tree paths.

The AI is capable of winning against other top ranked, highly competitive AI's. Further improvement on MCTS, such as implementing and training a network to assist with guiding exploitation of the unknown nodes and exploration of known nodes, is bound to be highly favorable.

2.2 Monomial Theory

A monomial is a commodity of algebraic variables such that the degree of every variable is 1. In this case, $x_1\ x_2\ x_3\ x_4$ is a monomial, but $x_1\ {x_2}^3\ x_3\ x_4$ is not because of the degree of x_2 is 3. It is understood that Gomoku can be played on a 15×15 board or a 19×19 Go board.

A 19×19 board has exactly 361 grid points, and for each particular points a variable is present. The objective of the player is to develop a horizontal, vertical or diagonal line to form 5 consecutive points. A monomial is created to represent each of the winning configurations. Such a monomial is simply the product of five variables for those five consecutive points. For an $n \times n$ board, if a list of c consecutive points is characterized, either horizontally, vertically or diagonally, with the second colored stone as a winning configuration, then we devise a formula as shown below. The formula used to calculate the monomial amount to show the total number of possible winning

configurations for the board.

$$\text{total winning configurations} = 2(n(n-c+1)+(n-c+1)2)$$

$$n = \text{board size}$$

$$c = \text{chain size}$$

For a 19×19 Go board, according to the formula a total of 1020 monomials are representing all achievable winning configurations for a player. The aspect of the algebraic variables and the monomials are symmetrical for the black and white (stones) players. In the beginning of the game, the black (stone) player will have a set of 1020 monomials. The black player positions its stone on a intersection represented by a variable x_1, this is equivalent to the black player setting the variable x_1 to 1. Once the black stone is positioned by the player the monomial $x_1\ x_2\ x_3\ x_4\ x_5$ then becomes $x_2\ x_3\ x_4\ x_5$. If the white (stone) player neglects x_2, x_3, x_4, x_5, the black (stone) player can keep positioning x_2, x_3, x_4, x_5 to 1 in four additional moves, thus the monomial is changed to 1, indicating the black (stone player) wins the game. After the black (stone) player positions x_1 to 1, the white (stone) player can position a white stone on the point represented by x_2. By accomplishing so, the white (stone) player sets x_2 to 0, hence the monomial $x_2\ x_3\ x_4\ x_5$ becomes 0, thus effectively eliminating a feasible winning configuration for the black (stone) player. The effect of setting a variable to 1 is symmetrical for black (stone) and white (stone) players. When the black player sets a variable to 1 on his own behalf, symmetrically he sets the variable to 0 to hurt the white player. Similarly, when the white (stone) player sets a variable to 0 to impair the black (stone) player, symmetrically he sets the variable to 1 in order to further his chance of victory. A monomial for a black (stone) player is operating, if none of the variables in the monomial have not set 0 by the white (stone) player. A monomial is inactive, if at leas one of its variables have been set to 0 by the white (stone) player. A variable in a monomial is called free, if it has not been set to 1 or 0 by a player. The black (stone) player finds a variable to set it to 1 with his goal to change one of his active monomials to 1, while at the same time to avoid the white (stone) player from doing so.

As the game advances, the black (stone) player need to assess his offensive strategy and defensive strategy. For offense, the individual needs to locate a way to maneuver one of the alive monomials to 1. For defense, the individual needs to terminate any possible monomial that can be altered to 1 by the white (stone) player. The black (stone) player need to locate an aggressive policy to balance the offense and defense strategy.

2.2.1 Board Points

For each consecutive move, the black (stone) player is demanded to know how to select a point to increase his/her chance to win and concurrently to diminish the white (stone) player's chance to win. The black (stone) player's objective is to assess potential values for every remaining move from the current game state and the history of the game. The monomials benefit with potential value assessing. For any point p that is represented by a variable x, we say that a monomial m covers p, if x is contained in m

For variable x, the score is defined for x as follows $s(x) = 1$, if x is a free variable; $s(x) = 2$, if $x = 1$, i.e., the black (stone) player has already set a white stone on the point represented by x. For any monomial m, the score of m is $s(m) = \prod_{x \text{ in } m} s(x)$. At any step of the game, with the assistance of monomials, the value of any point p is defined in the following: $v(p) = \sum_{m \text{ covers } p} s(m)$. It is easy to verify that for any a point p on the game board, the number of monomials covering it is between 3 and 20. The value $v(p)$ is a good indicator of $p's$ potential to help the black player to win the game. Symmetrically, we can define the potential value for any point on the game board for the white player.

2.2.2 Efficiency to Aid Search

Although there are more monomials (1020 total) than the number of points on the game board (361 total), those monomials allow a more up-to-date approach to handling the state of the game board. One of the key factors lies with the amount of calculations required for computing the potential values when compared to using points directly. For example, a point within the center region of the board would require the recalculation of up to 40 points. These 40 points consist of 10

adjacent points on each of the four orientations. With monomials, 20 monomials, or 5 monomials per orientation are dealt with in the game. Also, as the game advances, these monomials tend to become inactive, hence, to be discarded, at quite a rapid pace. A single move can wipe out up to 20 of the opponent's monomials while only a single point would be deemed inactive. Thus, with the monomial approach, the resources required to update the state of the board shrinks rapidly as the game progresses.

2.2.3 Locality Analysis

Another key feature of the monomials lies with the benefits provided for easier locality analysis. While dealing with the points directly may be less efficient, utilizing points to carry out locality analysis to identify vital patterns for offense or to detect critical pattern for defense. When used without guidance, locality analysis tends to be a resource intensive task to identify which points are required to be updated. If the black player is about to place a stone on point X, then this point X would no longer affect the previously occupied point, because the white stone created a local blockage between point X and the black stone on the right side. Shown on Figure 2.2. By relying solely on the points, it becomes quite difficult to manage the locality analysis as exhibited as above. In order to do this and other types of locality analysis, all adjacent points would need to be examined within a certain distance and identify if the white player occupies a point to enact some blockage or to launch an assault. Due to the sheer amount of point verification's required, such locality analyses would significantly increase the amount of resources needed to manage the game. The white stone eliminates all monomials covering it, including those covering the black stone and the point A. The surviving monomials would automatically encode the blockage between the point A and the black stone on the right, so no point check or testing is needed. Hence, it is evident that the monomials provide us with a simple and efficient solution to manage points with the history data that are readily available.

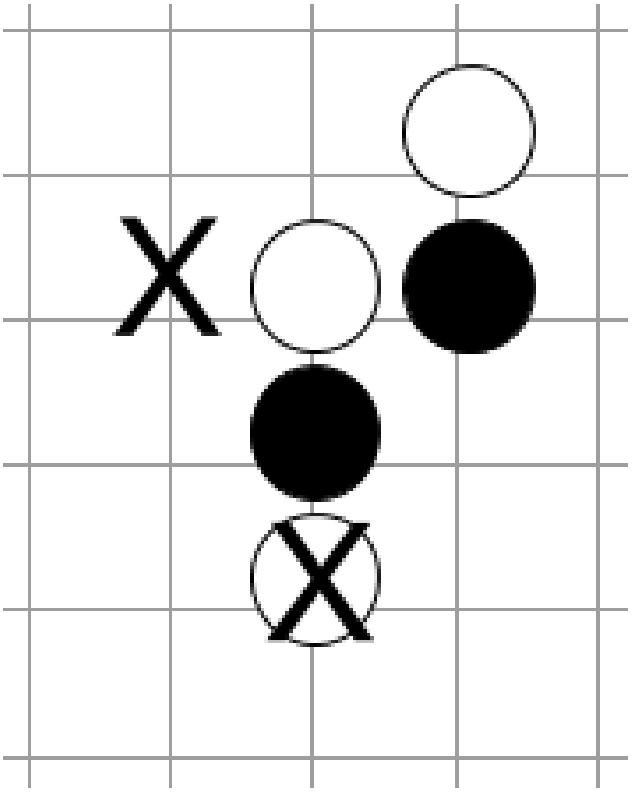

Figure 2.2: Utilizing Points: The black stone is no longer sharing a mutual monomial. If point X was taken on the left hand side of the white stone it is a wasted turn on the black stone player (unless it is trying to achieve a diagonal line). The black stone has also blocked a mutual monomial for the white stone player. As shown on the X placed on the white stone.

2.2.4 Potential Value Updating

The score updating process for monomials has incremental property and local characteristic. For the local characteristic, when the black player or the white players sets a value to a variable, i.e., places a stone at a point, this action only impacts the scores of the alive monomials containing the variable. Thus, we only need to update at most 20 monomials. For incremental property, when we need to update score for a monomial, we just need to multiple its old score by 2 in response to a move made by the black player, or set the score to zero to make the monomial dead in response to a move made by the white player. It is also worthy of noticing that the potential value updating for every active point q is incremental and local in response to a move made at p. The updating formula is: $newValue(q) = oldValue(q) + \sum_{\text{m alive and covers p and q}} newScore(m) - oldScore(m)$

The above locality and incremental properties make quick updating possible for recalculating monomial scores and point potential scores with respect to a move made by the black (or white) stone players.

2.2.5 Decision Making

Monomials also provide the aid of easily assessing the potentials for future actions from the current state and the game history data. For example, consider that the black (stone) player has a monomial m with score 8, i.e., $s(m) = 8$. This simple score of 8 tells us a lot of information, including the following facts: For 5 consecutive variables contained in m, 3 variables have been taken by the black players, 2 variables remain free, and importantly, there is a chance for the black player to place a black stone at one of the 2 free variables to win. Symmetrically, this applies to the white (stone) player so that the black (stone) player can detect from a monomial of score 8 that there is a potential threat from the white player. Those monomials scores can help us to build a decision tree to find competitive strategies for offense or for defense. Precisely, such a decision tree can easily guide the AI on deciding what moves should occur based on the given situation. Assume that the AI plays black. If currently the AI has its turn to make a move, then among many possible moves, including A, B, C, D, E, and F choosing B or D is a winning move, because B and the three

black stones will form an open four (i.e., four consecutive black (or white) stones with two free ends). Choosing A or C will not create an immediate threat to the opponent white player, because the white player can choose one of the remaining positions to put off the threat. On the other hand, if the AI was playing the white, and it is the AI's turn to move, then A, B, or C are rational choices because placing a white stone on any of them will eliminate the threat coming from the three black stones. It is highly unlikely that the black (stone) player will be able to achieve a horizontal line when playing with a human or AI but it still has different possibilities of winning. Shown in Figure 2.3 it can still form a diagonal line with the points D, E, and F. Placing the black stone on D gives the black player to initiate a win from either E or F.

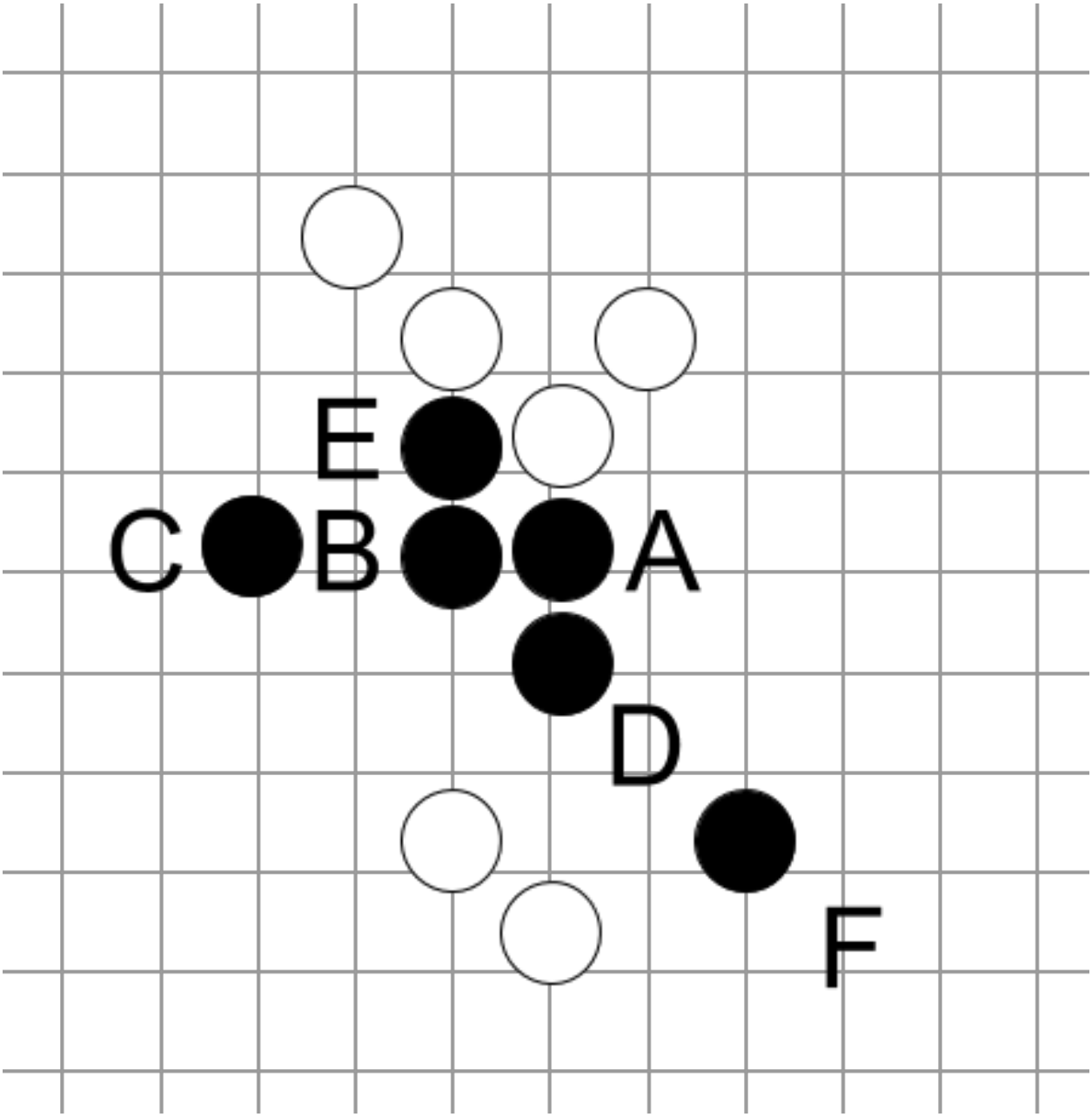

Figure 2.3: Decision Making:

CHAPTER III

GOMOKU VERSIONS

3.1 Version One

The original design of the system requires a large amount of resources to manage the game board. While acceptable at the beginning, this design leaves little room for new tools to be introduced. This is caused by the massive number of variables that need to be rewritten or updated as the game progresses. In Version 2, additional variables are added to help manage monomials and point tables in order to aid decision making and strategy searching. Without new improvements, those new variables will compromise system performance further. Hence, we need to devise new methods for efficient updating of monomials and variables.

Given any monomial, a point (or equivalently, a variable representing the point) contained in the monomial has a binary status, active or inactive (i.e., free), corresponding to being occupied or unoccupied by a player. Recall that from the game board, a monomial represents a line segment of five consecutive points horizontally or diagonally. Often, it is good to know the status of the two end points of the line segment. To maximize the benefits of monomials, two end points are added to a monomial such that a monomial is represented with seven points. On the border of the game board, there may be no end points for a monomial. In such cases, a virtual end point is added to conform to a uniform representation for all monomials. With the addition of two end points, a monomial has $2^7 = 128$ possible configurations, a big number but manageable.

Rather than continuing on the same path toward organizing monomials as before, here in our new approach we pre-generate all monomial configurations that can help us to know how monomials may interact with each other. Those configurations can be used readily throughout the game with the addition of two tables – the state table and the monomial table. The state table consists of all

the monomial configurations with related data generated before a game commences. This table remains static throughout the game and is only used as a reference to identify the characteristics that a monomial may have. The monomial table, on the other hand, is re-purposed as a 1020 1 table. This table is used to store a monomial's current state by storing the index of the corresponding configuration found within the state table. Rather than performing a large, tedious amount of calculations, with these tables, only one index entry needs to be updated per monomial that is affected by a move, thus resulting in a significant boost in speed.

The state table shows a simplified version of the monomial state. The monomials are generated in a binary format as mentioned previously. For example, the first monomial generated is the base monomial where all points are unoccupied, and simply denote it as monomial M_{00}. There exist 4 statuses for this monomial configuration in terms of 4 possible statuses of its two end points. The first row with both end points as False indicates that the two end points of M_{00} are unoccupied. Analogies between the binary values the two end points and their statuses are easy to see for the next three rows. In binary representation of Boolean and integer values, False equals 0 and True equals 1. And, two bits 00 represents integer 0, and 01, 10 and 11 represent integers 0, 1, 2 and 3, respectively. Thus, first four row of the state table can be denoted as M_{00}, M_{01}, M_{02} and M_{03}, In general, let M_{xy} to denote the $I(x,y)$-th row of the state table, where $I(x,y)$ is called the index, or the row number, $0 \leq x < 32$, and $0 \leq y < 4$. Precisely, x encodes the 5-digit binary representation of a monomial configuration and y represents the 2-digit binary representation of two end point statuses. Furthermore, it is easy to verify that

$$I(x,y) = 4x + y, \tag{3.1}$$

It follows from the above analysis, when a monomial in configuration M_{xy} needs to be updated to a new configuration M_{xy}, the changes are in row index $I(x,y)$ to the row of the new configuration

$$I(x',y') = 4x' + y' \tag{3.2}$$

from x and y it is calculated, because x' is resulted from one bit-flip of x or y' is resulted from one bit-flip of y.

The state table brings up enhancements in the system speed, because now it can efficiently update the game board statuses. In the previous version of the system, the majority of the information found within a row would have to be revised and updated, leading to many values to be overwritten, thus overall status updating is a time-consuming task. With the state table of pre-generated information and easy index calculation of configurations, the new method is able to release a significant amount of resources.

3.1.1 Gomoku Points

What has been improved for monomials is to devise new methods to function with the point table. The point table is split into 2 tables, one table stores the states while the other stores all information related to these states. Unlike the monomials, certain changes are needed for representing points.

While the monomial table is transformed into a 1020×1 table, the point table is, instead, transformed into a 361×4 table. Every point has a configuration corresponding to each orientation type (horizontal, vertical, right diagonal, or left diagonal). A point on the game board border or near the border does not have all four orientation types, but to conform with an easy approach, virtual orientations are added for those points. 4 configurations are stored for every point.

Second, a configuration of any given variable can be treated as the concatenation of 1 to 5 monomials. Take for example, assume that the horizontal configuration of a point x_5 is the list of 9 consecutive points on the line segment centered at x_5, denoted from left to right as $x_1x_2x_3x_4x_5x_6x_7x_8x_9$. This configuration is composed of 5 monomials $x_1x_2x_3x_4x_5$, $x_2x_3x_4x_5x_6$, $x_3x_4x_5x_6x_7$, $x_4x_5x_6x_7x_8$ and $x_5x_6x_7x_8x_9$. If point x_3 is taken by the opponent, then the horizontal configuration for point x_5 becomes $x_4x_5x_6x_7x_8x_9$, which is composed of two monomials, $x_4x_5x_6x_7x_8$ and $x_5x_6x_7x_8x_9$. If the AI occupies some points in the configuration, say, $x_7$and x_9, the configuration turns into $x_4x_5x_6x_8$ which is still composed of two monomials $x_4x_5x_6x_8$ and $x_5x_6x_8$. Those configurations are similarly generated using binary bits, but they must keep point x_5 free. If the

opponent player takes x_5, then all the active monomials contributing to the configurations of x_5 will become dead, thus the configurations are no longer helpful for the AI. On the other hand, if the AI takes x_5 then the configurations can help the AI to create threat to the opponent player. The above analysis can be generalized to any point.

Using these configurations, we can generate all possible states that may occur throughout the game. The number of states for any point is given by the formula. For, illustration, we consider the configuration $x_1x_2x_3x_4x_5x_6x_7x_8x_9$ for point x_5 again. Here, the number of active points in this configuration is 9. When we focus on point x_5, we would like to know how many different states its neighboring 8 active points can form? We end up with 4×2^8 possible states. 2^8 is derived from two possible choices available for each of the 8 points to participate in a state, either yes or no. the factor 4 is derived from the additional consideration for the left end of x_1 and the right end of x_9, and we need to know whether these two end points are blocked (or taken) by the opponent, thus leaving us with a factor of 4 more possibilities. Recall that the leftmost monomial of the configuration is $x_1x_2x_3x_4x_5$ and that the rightmost is $x_5x_6x_7x_8x_9$. We only need to consider the left end point of x_1 and the right end point of x_9, because the other end points of the 5 monomials of the configuration are already included in points x_2 to x_9.

For a configuration of length less than 9, it is difficult to estimate the exact number of states that can be derived from it. For example, continuing with point x_5 and consider a configuration of length 8 for x_5. Following the discussions above, we have two possible cases: (a) the configuration is $x_2x_3x_4x_5x_6x_7x_8x_9$; and (b) the configuration is $x_2x_3x_4x_5x_6x_7x_8$. This leads to the conclusion that the opponent has already taken point x_1 in case (a) or x_9 in case (b). Once again, x_5 is our focus point, hence it must remain free. Thus, we have 2^7 possible states for either case (a) or case (b). Like before, we still need to consider end points. For case (a), since x_1 is already taken by the opponent, we only need to consider the rightmost endpoint, which is right next to point x_9. Thus, this endpoint leaves us with a factor of 2 more possibilities. Similarly, for case (b), we only need to consider the leftmost end point, which is left next to x_1, leaving us with a factor of 2 more possibilities. In summary, there are $2^7(2+2) = 2^7(2 \times 5 + 2 - 8)$ possible states with respect to a

configuration of length 8.

Continuing on point x_5, let us consider one more configuration of length 7. Here, we have three cases for such a configuration: (a) $x_1x_2x_3x_4x_5x_6x_7$; (b) $x_2x_3x_4x_5x_6x_7x_8$; and (c) $x_3x_4x_5x_6x_7x_8x_9$. As for configurations of length 8, each of cases (a) and (c) corresponds to one end point with two possibilities. But case (b) implies that the two end points are already taken by the opponent, leaving no more addition possibilities. Hence, the total number of states that can be derived from configurations of length 7 for point x_5 is $2^6(2+2+1) = 2^6(2\times 5+2-7)$.

Now, a general trend of the number of states that can be derived from configurations for point x_5: For configurations of length k with $5 \leq k \leq 8$, the total number of states is $2^{k-1}(2\times 5+2-k)$; and for k = 9, the total number is $4\times 2^{k-1}$. These observations can be generalized to any given point and to monomials of length n, $n \geq 5$.

For any given point, finding all possible states that can be derived from its configurations can be troublesome. Furthermore, it is even more of a hassle to calculate ID's of those states, because doing so would require considering binary statuses of the points involved in configurations of variable lengths, and whether the opponent has already occupied the end points or not. Therefore, rather than performing the search tasks guided by the formula in the previous paragraphs up to 32 times per player per turn, we instead store the information needed within the state table. At every row of the state table, each configuration is augmented with the ID of every state that may be reached within 1 move. This allows us to look up and identify the next ID easily from the current ID, and the processing time would be similar to that of updating the monomial tables. These state table provides an efficient manner of analyzing the data found within the board. The main advantage of the state table is storing of concatenated configurations of all active monomials with respect to all free points. These configurations allow fast discovery of features possessed by a free point. For example, these configurations can help the AI the detect whether a point can form a "combo" by its two configurations on two different orientations. "Combos" are threatening moves and will be discussed in detail later.

3.1.2 Selection

As mentioned previously the decision making of the initial version of the AI is simple-minded, so that the opponent player is able to predict the AI's moves after seeing many of the AI's moves. A number of games have been played and analyzed in order to identify what exactly would lead the AI to lose. We want to know whether a loss of the AI is caused by a mistake of the opponent player and by a list of well calculated moves. One of the major observations tells us that the AI is not able to learn from the same mistakes. For example, the opponent may create two threatening monomials simultaneously, leaving the AI in a vulnerable status without knowing any defensive move. Using this knowledge learned from analysis, the AI is enhanced with an ability to detect these threats which are referred as "combos." A "combo" consists of two monomials that lie on two different orientations but interest at a free point p with following property: Once a player plays a stone at the point p then the two monomials will represent two winning configurations so that the opponent become defenseless.

By the definition, when a player finds a combo, the player can create a threat to the opponent. The combos can be found by searching for monomials of different orientations which share a common point and are one turn away from becoming a threat. While searching for combos is a resource intensive task, yet the new point table provides enough resources for this task. If a union exists between 2 of the intersecting monomials on two different orientations, a combo is formed, leading to a win or loss within 5 turns. The combos ultimately allow the AI to look ahead by an additional two turns for wins and losses.

In addition to combos, another crucial enhancement of the AI is the inclusion of Monte Carlo Tree Search (MCTS). As stated earlier, our AI is not able to probe many moves ahead. The new resources of organizing monomials and points make it possible to incorporate a MCTS component, a resource intensive process. This MCTS component can effectively perform many simulations of roll-out expansion until either a max depth is reached, or a player achieves victory. This MCTS quickly resolves most of the original faults of our AI and helps significantly enhance its competitiveness.

3.1.3 Performance

This implementation was overall a huge success. The performance of the new version of the AI is increased significantly, making it ever much more difficult for a human player to win. With the combination of tools at its disposal, the AI has become extremely efficient against human players, because no other human that has played against it has been able to win. Due to this challenging situation, it becomes necessary to test our AI against other top-rated Gomoku AI's. While our AI can easily win a human player among those we know, including ourselves, wining over some top-ranked Gomoku AI's is no easy task. Although the AI can win some of the top-ranked Gomoku AIs, yet this version has not been able to beat any one of the top-10 ranked Gomoku AI's from the latest Gomocup list.

3.2 Version Two

The goal for the Gomoku program is to build an AI to win the top-10 ranked Gomoku AIs from the latest Gompcup list. With this in mind, we shall overhaul the previous architectures and also include Monte Carlo Tree Search (MCTS) to find a competitive strategy ahead of many moves. The successes of AlphaGo[1] and the existing Gomoku AIs provide insight that MCTS is a valuable tool to be included in the system. After many experiments, the realization is that the reliance of MCTS on combos for its search becomes an issue. While combos may provide a winning strategy, the AI must also take the entire board into account. Sometimes, when the AI attempts to create a combo, the opponent player is able to find a set of moves to overtake the combo. This will force our AI to immediately turn onto the defensive mode, and in the end the AI may lose the game because of a single mis-calculated move. The mistake occurs, because once a combo is identified the AI tends to over prioritize the winning move, so that the AI may simply overlook some of the opponent's possibility to win. Thus, rather than concentrating on combos, the final version will invest the good share of its resources on MCTS. While the combos are still used, yet they are mainly used to guide the MCTS toward finding a competitive strategy for offense or defense.

The first major change is to introduce additional information to the monomial state table.

Instead of having a score based on 2^0 to 2^5, new scores are provided for a monomial to indicate whether the AI needs to take an action for the monomial, such as if the opponent occupies a point in a monomial, whether this will force the AI to react. The new scores are implemented in order to make up for the removal of the point tables. The new scores help provide a clearer picture than the previous scores that are no longer used for points.

In the previous approach, recall that a unique score is assigned to every monomial. For the new design, we assign two rank scores to every monomial. We do not need to concern with any monomial with an old score of 2^5, because such a score means a win or loss depending on which player owns the monomial. We also do not concern with any monomial with an old score of 0, which means that the monomial is dead or inactive. Hence, we shall address monomials with old scores from $2^0, 2^1, 2^2, 2^3$ and 2^4. An old score of $2^i, 0 \leq i \leq 4$, implies that an active monomial has i variables occupied by the player who owns it.

The new rank scores range from 0 to 9. A monomial of an old score of 2^4 is assigned a new rank score of 0 or 1, with 0 representing that this monomial guarantees a win, but with 1 representing that a win is possible but can be blocked.

A monomial of an old score of 2^3 is assigned a new rank score of 2 or 3, and this assignment continues for a monomial of an old score of 2^2 or 2^1. Finally, for a monomial of an old score of 2^0, it is assigned a new rank score of 8 or 9. An odd rank score represents a monomial can be blocked, while an even rank score represents that a monomial cannot be blocked.

The new rank scores are favorable for building a decision tree. Previously, a decision tree may have located a monomial of an old score of 2^3, but additional checking to determine if additional action is required to move toward a victory. However, with the new rank scores, such a monomial may have a rank score of 2 or 3, with 2 meaning no additional action is needed but 3 meaning further action is needed. Hence, there is no need to do additional checking. The rank score system helps streamline the decision-making processes for the decision tree and the MCTS.

In the implementation, with the help of the new rank scores for monomials, an increase of the search tree depth is more than twice for MCTS. This increase of the search depth allows far

broader search for a wining strategy. In addition, for scenarios where the search was solely focusing on guaranteed winning possibilities, now the AI has the resources to consider all possible options that may be taken by the opponent. Evidently, this additional ability allows the AI to decide, with higher confidence, whether a move would truly help achieve victory. These new changes, while seeming simple, require a variety of setting ups to be done. For example, to help MCTS increase the search tree depth, a naïve approach to simply double the search tree depth will slow down the speed of the previous MCTS a factor of four. With the new system, since the point table is not being utilized and the reliance on points is removed, the speed of the new MCTS is more than quadrupled despite the doubling of the search tree depth.

3.2.1 Result

While the AI was previously at a standstill to obtain a single win against any of the top-10 ranked Gomoku AI's from the latest Gomocup ranking, the new version of our AI is capable of defeating the top-10 ranked AI's. For example, for the top 7th ranked Gomoku AI, Wine, the previous versions of our AI achieved only a single win against it during a number of competitions in a span of several months. However, the new version of our AI has achieved a 30% winning rate over Wine when our AI plays first.

CHAPTER IV

EXPERIMENTS

The gomoku program was able to make significant improvements from version one to version two. The first version was only able to win once against the top 7th ranked Wine AI. The second version is now able to win Wine with a ratio of 30%. The monte carlo tree search was improved to make efficient decisions to be able to win against other AI's. The figures below are the wins and losses against the top 7th ranked Wine. In order to get to this point over 400+ matches were manually completed against the top 10 AI's in Gomocup. The iterations were all completed by one individual compared to AlphaGo [6] by DeepMind it does not come near to the amount of iterations that it produced by them. AlphaGo is able to produce over millions of iterations. They are able to produce such high amount because they consist of a large team that compartmentalizes the project. For this Gomoku AI it was produced by one individual with one Hewlett-Packard "HP" laptop. Limitations were met due to the lack of equipment but progress was still made.

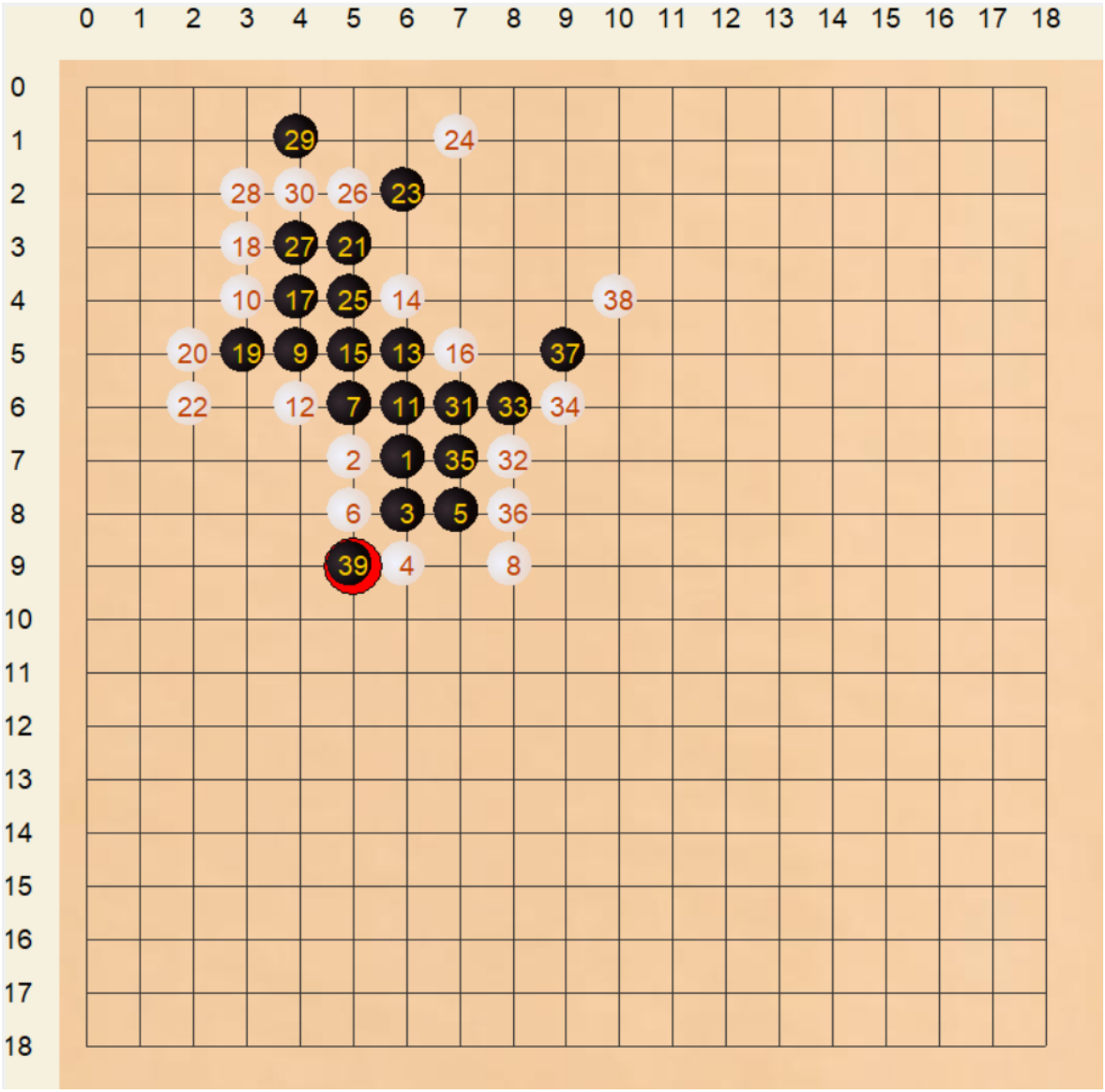

Figure 4.1: Version 2 - Number 1 Win: The artificial intelligence program is able to establish a win because it was aggressive on it's offense and able to capitalize on the mistakes of Wine

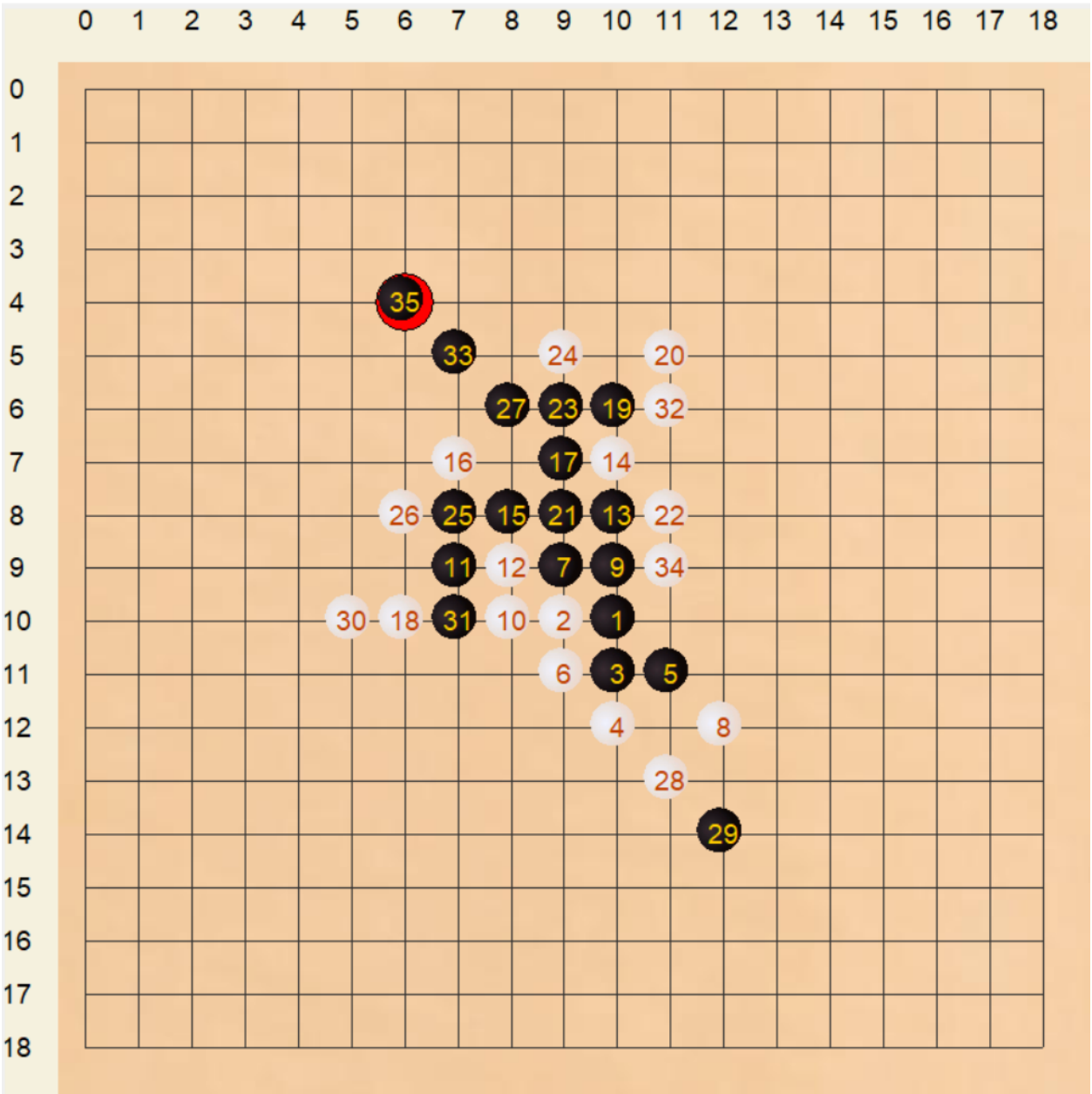

Figure 4.2: Version 2 - Number 2 Win: The artificial intelligence program begins with a defensive approach because Wine starts with a strong offensive but the AI is able to establish a win from 27 and on.

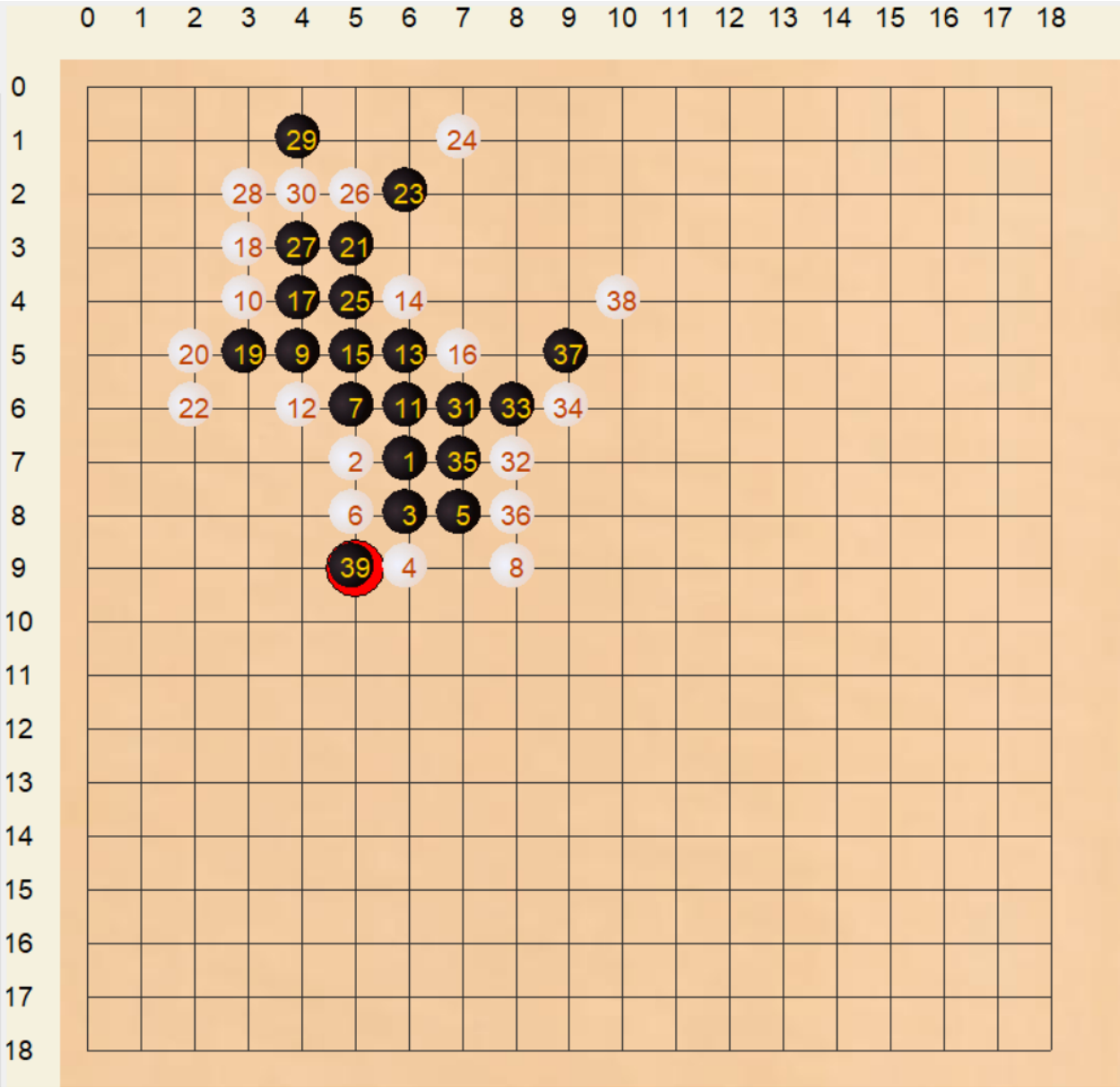

Figure 4.3: Version 2 - Number 3 Win: The artificial intelligence program begins with an aggressive offense. It is able to establish a inside position by doing so it has a higher probability of winning. As shown in the image the AI has various connections of four.

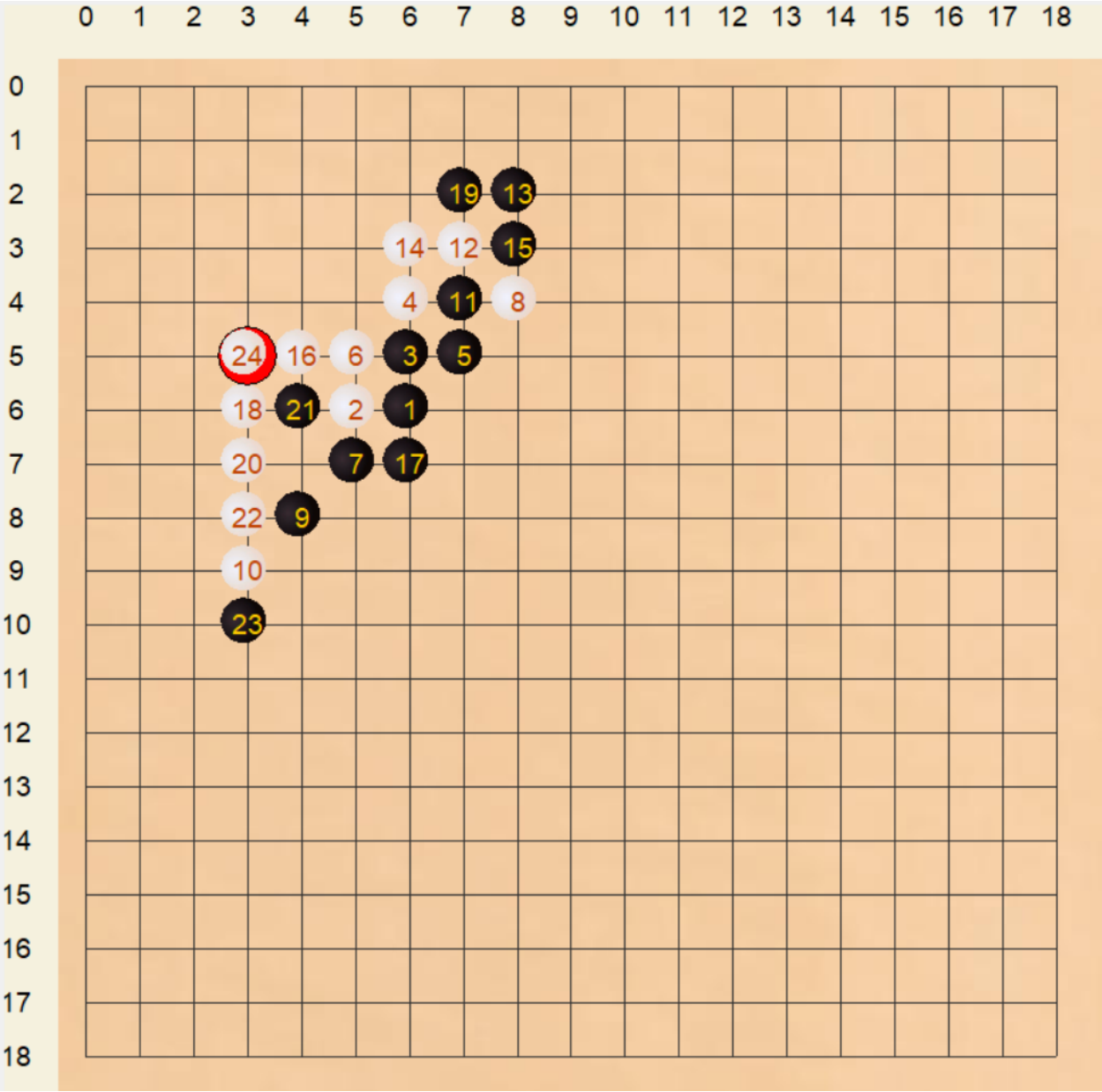

Figure 4.4: Version 2 - Number 1 Loss: Wine was able to establish an easy win.

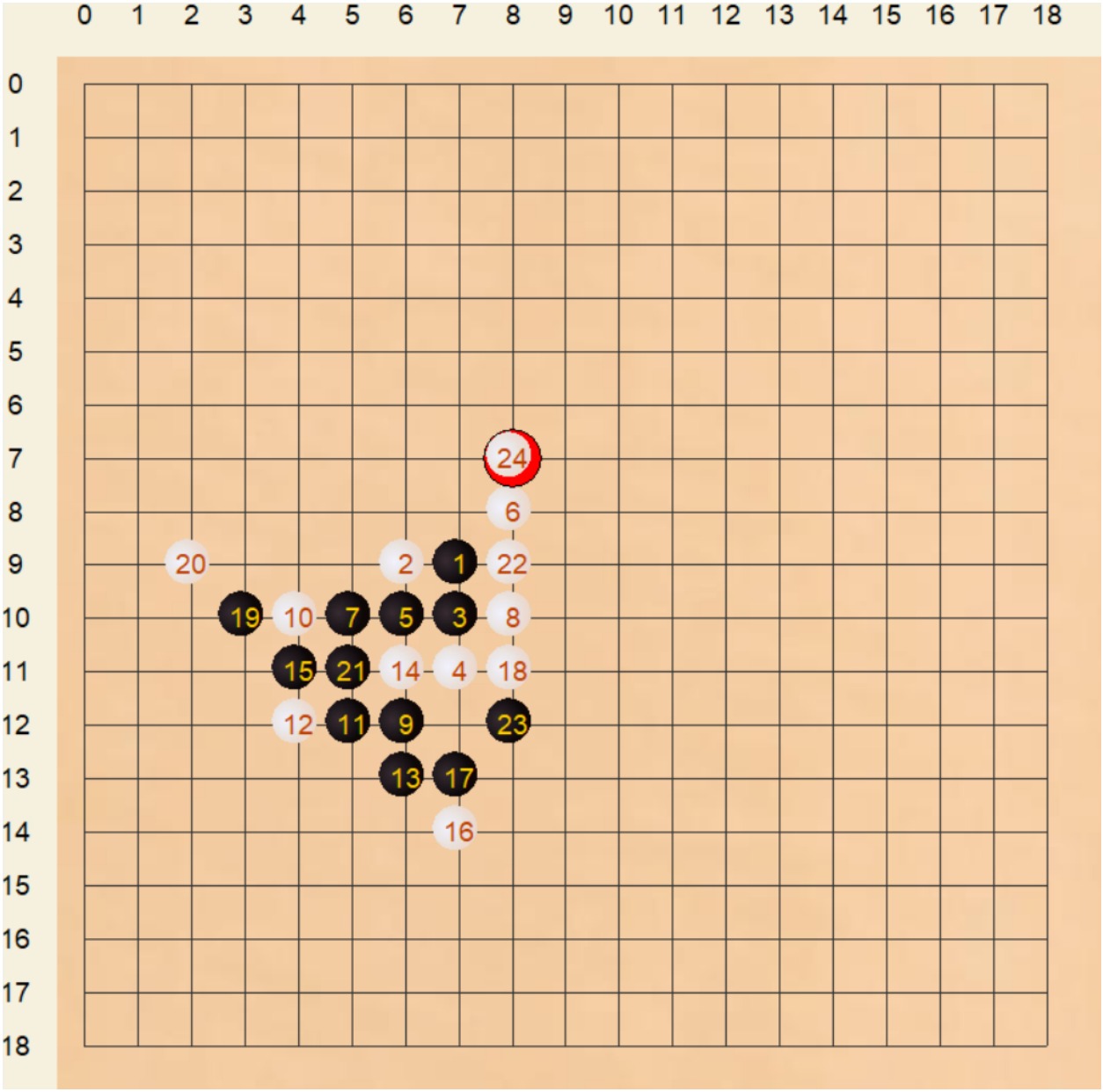

Figure 4.5: Version 2 - Number 2 Loss: The Gomoku AI was enable to be aggressive on its defense and Wine was able to win the match by the 24th move.

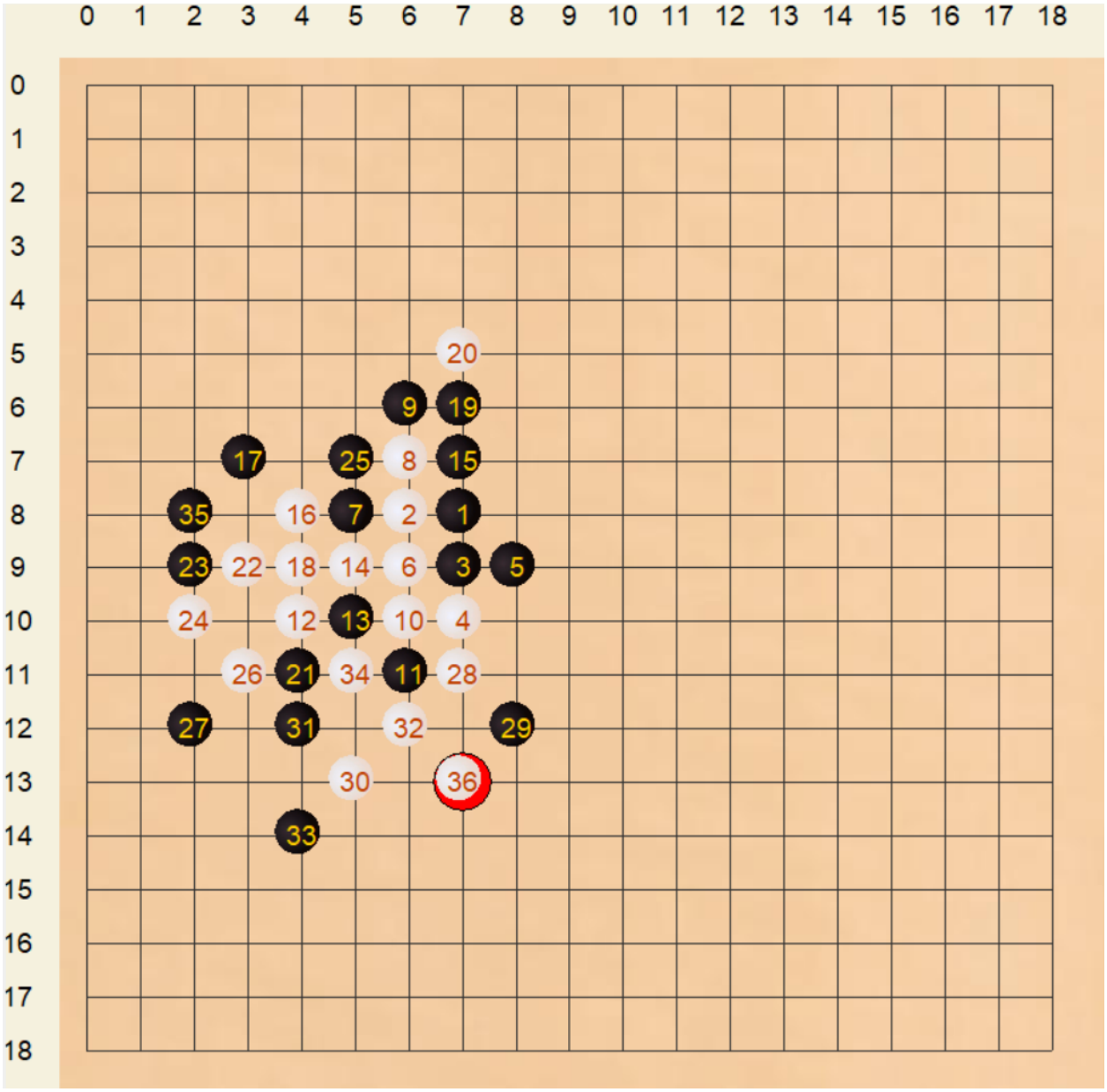

Figure 4.6: Version 2 - Number 3 Loss: In this match the Gomoku AI is able to establish a good defense. It was able to suppress Wine in the beginning but Wine had to possibilities of winning.

CHAPTER V

RELATED WORK

5.1 AlphaGo

In 2016, AlphaGo [6] established a win against Lee Sedol, a 18-time Go world champion, in a five game match, with the score of 4:1, which was the highest achievement of a computer game engine in the longtime human-computer Go challenges. Previously, AlphaGo would only play Go at a strong amateur level. Being able to defeat the highest professional human player made AlphaGo a state of the art Go engine at its time.

AlphaGo [6] introduces a nonlinear way by using a neural network to make better use of human experts knowledge. During the MCTS simulation process, AlphaGo is not looking for the matching of the handcrafted patterns, but it utilizes a convolutional neural network model to learn the MCTS default policy. This approach has its advantages. The network models are trained from human plays, thus the neural network enhances the original default policy by significantly reducing its randomness. The useful information from human plays is extracted by the neural network with a high degree of nonlinearity, which is much more efficient than doing this by hand. AlphaGo [2] trains a policy network and a value network to perform the default policy calculation. The training contains a 13-layer policy network with 30 million positions from the KDS Go Server with an accuracy of 57.0 percentage. in order to further improve the performance of the policy network, a reinforcement learning framework is applied. The framework lets different policy networks selected from different training iterations play with each other until a terminal state is reached with a reward value of +1 for winning and -1 for losing. Based on the reinforcement learning plays, a value network with the same structure as the policy network is trained. With policy and value networks in hand, a final approach then combines them into the structure of MCTS to actually play Go at

$$\frac{Q(v)}{N(v)} + \frac{P(v|v_0)}{1+N(v)}$$

Figure 5.1: AlphaGo Selection Process

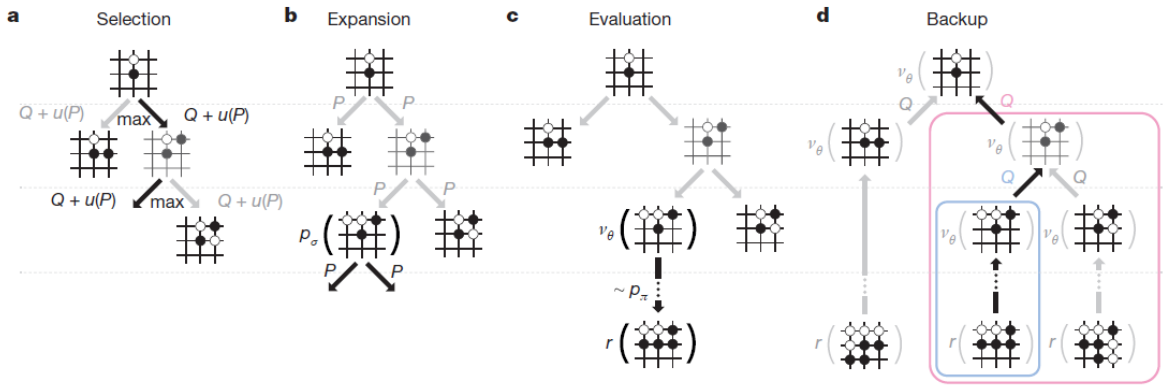

Figure 5.2: Monte Carlo Tree Search Structure in AlphaGo

AlphaGo's level.

The process of AlphaGo's MCTS [2] has two improvements compared with the usual MCTS algorithm. Inside the evaluation (simulation) process, instead of using the previous default policy to choose next move randomly, AlphaGo uses a pre-trained policy network to perform the select. Inside the selection process, instead of using the traditional UCB method, AlphaGo uses a new function shown in Figure 5.1 to take policy and value networks into account. In Figure 5.2 it displays the MCTS search structure that is utilized in AlphaGo [2].

Combining MCTS with deep neural networks, AlphaGo reached the professional level of playing Go. The newest version of AlphaGo reveals that, without the help of human knowledge, computer game engines can perform even better.

5.2 AlphaGo Zero

AlphaGo Zero [3] became the state of the art computer Go engine that outperforms any other engines. Similar to AlphaGo, AlphaGo Zero combines MCTS with neural networks. However, instead of training the network models from human plays, AlphaGo Zero trains them from scratch. This indicates that at the beginning, the policy and value networks have no knowledge about the

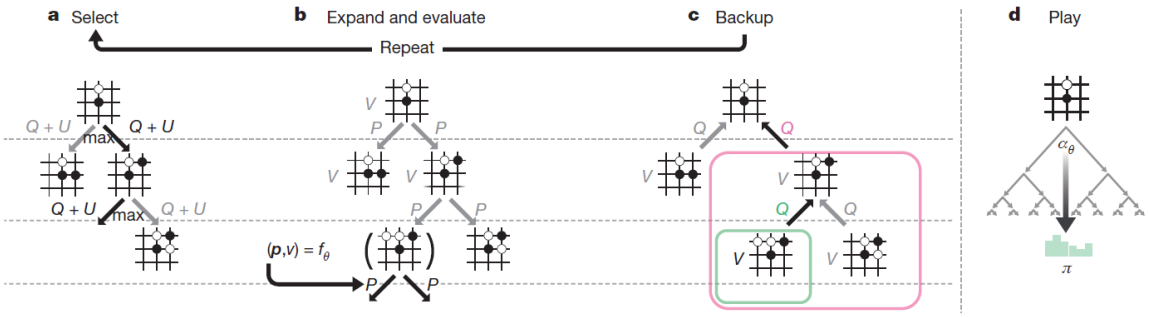

Figure 5.3: Monte Carlo Tree Search Structure in AlphaGo Zero

game. The engine then puts these models into a reinforcement learning (RL) framework, and trains them based on their own results. The approach achieved huge success.

The training strategy of AlphaGo [3] can be described as follows. The policy and value networks are initialized randomly, a MCTS structure is then using these models to play against itself. Moves are selected based on the search probability π, which is the output value of the policy network, and the scalar value v, which is the output value of the value network. Once the terminal state s and the winner z are recorded, a training process will be held in parallel. The purpose of the training is to take s as input and output its own search probabilities p and scalar value r, and to maximize the similarity between π and p, while minimizing the error between r and the real winner z. After several training iterations, the RL framework will start a competition between the newly trained network with the current MCTS network, and the winner's weights will be updated into the MCTS's network.

Adapting the training changes, AlphaGo Zero [3] also modifies the MCTS structure. It joins the tree policy and default policy into one, which guides MCTS's four stages, the difference between the expand stage and the evaluation (simulation) stage is now disappeared. As shown on Figure 5.2 the randomly default policy is discarded, so instead of letting the previous simulation stage complete the whole play as often as possible, now the four stages are performed repeatedly to finish just one play. AlphaGo Zero combines the policy and value networks into one residual network model with two branches. The architecture outperforms the other approaches, such as, separate residual network, dual convolutional network and separate convolutional network.

5.3 Fuego

Fuego [1] is an open-source framework for developing game engines for two-player board game, with a focus on the game of Go. The software framework enhances MCTS with a playout policy and prior knowledge trained to predict human expert moves. Alongside with a lock-free multithreaded environment. Fuego improves MCTS's performance, while limiting its randomness.

The modifications made by Fuego [1] is to reduce MCTS's search tree further by taking human experts experiences into account. The software framework holds a small set of hand selected 3 * 3 patterns built by human experts. During the simulation process, a move will be selected if it matches one of the patterns and is adjacent to the previous move; this is referred to as play-out policy. Only in the case where no adjacent pattern is found, Fuego lets the traditional MCTS default policy to select the moves randomly. Since the default policy is less efficient than play-out policy, in order to increase the latter's proportion, Fuego developed a replacement policy which attempts to move tactically bad moves to an adjacent point, which indicates that a move based on the pattern match is more likely to happen. The prior knowledge and the move filter, are used to narrow the search to a set of moves with probabilities. The prior knowledge enables the game engine to reward and punish certain moves based on its features. After finishing every pattern match of the playout policy, based on the current selected move, a rewarding credit will be added to all of its neighbors up to a certain distance.

5.4 Gomoku with Adaptive Dynamic Programming and Monte Carlo Tree Search

Researchers designed a Gomoku engine in 2016. The programs utilized are Adaptive Dynamic Programming (ADP) to train a neural network [4], combined with MCTS, and the results indicated that the engine outperformed the Gomoku engine with a single ADP algorithm.

The functions of the programs are described as follows [4]:

1. A shallow neural network with 3 layers which is trained by the ADP algorithm is provided.

2. From any given checkerboard state, using the pre-trained network to output 5 candidate moves with their winning probabilities.

3. Using these 5 moves with their associated states as the root state of MCTS respectively, the MCTS then performs these 5 choices, and outputs 5 MCTS based winning probabilities.

4.Using a weight value λ to balance the network based winning probabilities and MCTS based winning probabilities, the highest one will be chosen as the next move.

When comparing with AlphaGo's algorithms, the Gomoku engine with ADP and MCTS has two different approaches. Firstly, [4] the network model is not embedded into the structure of MCTS. Instead, several iterations of MCTS are performed separately, and the outputs are then combined to selected the best move. Secondly, since it is MCTS that actually performs the simulation and plays the game, the policy network is not needed any more, only a value network which outputs the winning probabilities is trained.

CHAPTER VI

FUTURE WORK

6.1 Neural Network

Neural networks belong to a class of models within the machine learning spectrum. A neural network entails a specific set of algorithms that can often improve machine learning results. Mathematically ,neural networks are general function approximations, which is why they can be applied to almost any machine learning problem about learning a complex mapping from the input to the output space.

Neural Networks have been widely used in domains like Natural Language Processing, Image Classification, Reinforcement Learning, etc. They can analyze the input features in a non-linear way, and approximate a transformation function between the input features and the target values with high confidence. If utilizing policy strategies to reduce MCTS's randomness, neural networks usually perform better than traditional methods in gathering those policies.

6.2 Policy - Value Neural Network

The key to overcome an enormous search space when performing a simulation is to combine two deep neural networks - the "policy network" and the "value network". Each of the two networks contains many layers with millions of neuron connections. The "policy network" predicts the next move, and is used to narrow the search to consider only the moves most likely to lead to a win. The "value network" is used to reduce the depth of the search tree - estimating the winner in each position in place of searching all the way to the end of the game.

6.3 Convolutional Neural Network (CNN)

In 1998, Yann LeCun [7] and collaborators developed a recognizer for handwritten digits called LeNet. It used back-propagation in a feedforward net with many hidden layers, many maps of replicated units in each layer, output pooling of nearby replicated units, a wide net that can cope with several characters at once, even if they overlap. It was later formalized under the name "Convolutional Neural Networks".

The objective of a Convolutional Neural Network is convolution. Each convolutional layer has multiple small sizes of kernels where each of them is related to a pattern. Convolving [7] this kind of kernel to the input features means to extract that specific pattern from the input. This is done by moving the kernel through the whole input and performing the convolution operation continuously. As a result, the original picture is transformed into a set of pictures consisting of different patterns, meaning it has been abstracted in different ways. Compared with traditional fully connected layer which converts the input picture into a 1-dimensional vector, the convolutional layer converts it into a more hierarchical and clearer format, and makes the future analysis easier. Convolutional neural networks are suitable for applications where the input value is large and has depth. This makes it a good candidate for machine checkerbaord game players. Each individual square needs to preserve different information such as a player's number, turn to play, etc.

6.4 Residual Neural Network (RNN)

Convolutional Neural Networks focus on training the network model in a efficient way, while the Residual Neural Networks [3] are trying to fully use CNN's potential ability to enable a network model to contain as many convolutional layers as possible. Some unsolved technique problems, such as gradient vanishing, imply that the network model can't be expanded to a large number of layers, otherwise relevant information will be lost during the long back propagation process. There is no guaranteed method to prevent information vanishing, but researchers have developed RNN to bypass it. A RNN consists of residual blocks. The key issue of a residual block is that it keeps adding shortcuts around at least each two convolutional layers. This indicates that

before the actual convolution operations, it preserves a copy of the input value, and this copy is then added back to the output after the convolution operations are completed. By implementing this, it ensures that even if the useful information is lost during the calculations, that information will be added back at the end of the calculation. Since most of the useful information will be preserved during the training, the network model can be designed much deeper with more convolutional layers, such a network model will have greater performance.

RNN's can be effective in two cases. The first case [3] is where the training is complicated and could not be done by a shallower network model, such as checkers. The second one is when there are multiple models that are in need to learn the same features during a period, and in this situation models can be incorporated into one RNN model. They learn the same features since they are in one model, and when they need to learn different avenues, the RNN model gives each of these models a shortcut and leads them to different outputs. with an accumulation of neural networks and MCTS, a strong checkerboard game engine will be built, where the neural network can help the MCTS reduce its randomness, and provide the network model a platform which can further enhance its performance. This approach can be even further improved with reinforcement learning.

6.5 Deep Q-Learning Framework

The reinforcement learning framework can be incorporated into a game engine to learn and play games by itself autonomously. The more games self-played and more errors generated during the autonomous learning process, the better will the game engine slowly begin to get after seeing the more rewards. A combination of MCTS and neural network model [2] equips a game engine with an ability to continuously improve itself. The framework that is used in the domain of machine checkerboard game is called Deep Q-Learning (DQN) framework. The core of a Q-Learning framework [2] is a table Q(S,A). It represents a function that outputs the reward value for a state S if it takes the action A. For each state S, only the action with the highest Q value will be chosen. The statistical data stored in states is updated after each play, so the Q value continuously becomes more accurate, and gives the player a better prediction. Instead of the table Q(S,A), DQN uses a neural network to approximate the reward function.

For the machine checkerboard games, the reward Q function is divided into two network functions, a policy function and a value function. The policy function returns a potential next action based on a given state. Wile based on the same state, the value function returns the probability score that indicates to what extent this state will win. The outputs from these two functions are then balanced by a constant λ to output the final reward Q value. By processing this, the DQN ensures that the machine player balances the local game state and the sense of the bigger picture.

6.6 Self-Play Reinforcement Learning

To generate self-play data, saving both the current latest model and the historical optimal model obtained through evaluation, and the self-play data is always generated by the optimal model, which is used to constantly train and update the current latest model, and then evaluate the current latest model and optimal model at regular times to decide whether to update the historical optimal model. This process is simplified, by saving only the current latest model, self-play data is generated directly from the current latest model, and is used to train to update itself.

6.7 Longer Training Duration

With longer training times and a machine with efficient GPU, the Gomoku program can train the machine player to be more intelligent. With stronger GPU support, the speed of the training process will be faster, and in a single month over 100,000 iterations can be performed, which indicates exceptional offense and defense strategy to win against other AI's.

6.8 Symmetric State

After creating a new board, a horizontally symmetric board is generated, hence one board is expanded into two boards. The consequence of this is that the player selected a bottom-up fill up game strategy, which indicates it only considers the squares located in the current row, either on the same side of the current piece, or on the horizontally symmetric side.

6.9 Gomoku AI Implementation

What fell short in this experiment was a Policy-Value Neural Network , Self-Play Reinforcement Learning and Longer Training Duration. If time permitted the items listed above would have been implemented into the Gomoku AI. The policy network would have assisted the prediction of the next move in the Gomoku match, and able to narrow the search to consider only the moves most likely to lead to a win. The value network would have assisted to reduce the depth of the search tree. The function of the value network is to estimate the winner in each position in place of searching all the way to the end of the game. These two networks would have been implemented into the already existing Gomoku AI. The self-play reinforcement learning would also have been a key factor to having a efficient AI. Implementing this into the program would be able to generate self-play data. By utilizing the current latest model, self-play data is generated directly from the current latest model, and is used to train to update itself. With longer training times the Gomoku program could have generated more iterations in a span of months. A single computer would have been dedicated to the program to run iterations daily. This would have significantly improved the AI to win against the top five AI's in Gomocup.

CHAPTER VII

CONCLUSIONS

By utilizing Gomoku algebraic monomials, a successful artificial intelligence has been created that is adept of playing the game at a highly competitive level. The mechanics and architeture develped for the artificial intelligence are all established on monomials. The artificial intelligence gomoku program has been tested against top-ranked Gomoku AI's from the latest Gomocup list. The most recent version of our artificial intelligence is able to win up to the top 7th ranked AI.

The artificial intelligence gomoku program can be improved with policy-value neural network, self-play reinforcement learning, deep q-learning framework, longer training duration and multi-threading for the Monte Carlo Tree Search. As displayed by AlphaGo all of the functions listed above can be incorporated to provide a more efficient and effective artificial intelligence gomoku program.

BIBLIOGRAPHY

[1] M. ENZENBERGER, M. MULLER, B. ARNESON, AND R. SEGAL, *Fuego–an open source framwork for board games and go engine based on monte carlo tree search*, IEEE Transactions on Computational Intelligence and AI in Games, (2010).

[2] D. SILVER, A. HUANG, C. J. MADDISON, A. GUEZ, L. SIFRE, G. V. D. DRIESSCHE, J. SCHRITTWIESER, I. ANTONOGLOU, V. PANNEERSHELVAM, AND M. LANCTOT, *Mastering the game of go with deep neural networks and tree search*, Nature, (2016).

[3] D. SILVER, J. SCHRITTWIESER, AND K. SIMONYAN, *Mastering the game of go without human knowledge*, Springer Nature, (2017).

[4] Z. TANG, D. ZHAO, K. SHAO, AND L. LUV, *Adp with mcts algorithm for gomoku*, The State Key Laboratory of Management and Control for Complex Systems Institute of Automation, Chinese Academy of Sciences, Beijing, (2016).

[5] V.ALLIS, *Searching for solutions in games an artificial intelligence (ph.d.thesis)*, University of Limburg, Maastricht, The Netherlands, (1994).

[6] Z. XIE, X. Y. FU, AND J. Y. YU, *Alphagomoku: An alphago-based gomoku artificial intelligence using curriculum learning*, Likelihood Lab, (2018).

[7] R. ZHANG, *Convolutional and recurrent neural network for gomoku*, Stanford University, (2016).

BIOGRAPHICAL SKETCH

David Garcia graduated from the University of Texas at San Antonio in 2016 with a bachelor's in Art. Following graduation, he began working at The University of Texas Health Science Center - MD Anderson Cancer Center. Once resigning from a full time position at MD Anderson Cancer Center he pursued graduate school in 2019 at The University of Texas Rio Grande Valley and earned a Master of Science in Computer Science in December 2020. Contact info: Davidgar152@gmail.com