5-2020

# A Mathematical Approach to Gomoku

Oscar Garcia
*The University of Texas Rio Grande Valley*

# A MATHEMATICAL APPROACH

# TO GOMOKU

A Thesis

by

OSCAR 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

May 2020

Major Subject: Computer Science

# A MATHEMATICAL APPROACH

# TO GOMOKU

A Thesis
by
OSCAR GARCIA

COMMITTEE MEMBERS

Dr. Zhixiang Chen
Chair of Committee

Dr. Bin Fu
Committee Member

Dr. Dong-Chul Kim
Committee Member

May 2020

# ABSTRACT

Garcia, Oscar, A Mathematical Approach to Gomoku. Master of Science (MS), May, 2020, 34 pp., 3 tables, 4 figures, 7 references, 16 titles.

This goal of this thesis is to design and implement a light weighted AI for playing Gomoku with high level intelligence. Our work is built upon an innovative algebraic monomial theory to help assess values for each possible move and estimate chances for the AI to win at each move. With the help of the monomial theory, we are able to convert winning configurations into monomials of variables that represent the underlying board positions. In the existing approaches to building an AI for playing Gomoku, one common challenge is about how to represent the present configuration of the game along with the history of the moves of the two players. Compared with the usual 2D matrix of the board positions, our monomials can make the AI easily understand the current state and the history of the game, and they also allow the AI to compute the potential values for future moves from the current state and the history of moves made by the players. In addition, when we adopt the Monte Carlo Tree Search to probe for a possible winning strategy for the AI, those monomials help reduce the search space, in addition to help estimate rates for exploration of the historical moves and exploitation of the future moves. Based on the proposed algebraic monomial theory, we have implemented a lightweight powerful AI that is capable of playing Gomoku at highly competitive level. At this stage, our AI can win top rated AIs (up to top 7) from the most recent Gomocup rating.

TABLE OF CONTENTS

LIST OF TABLES

Page

LIST OF FIGURES

# CHAPTER I

## INTRODUCTION

Gomoku is a strategy board game traditionally played on a GO Board. The game is played by two players alternatively placing a black or white stone on the intersection positions on the board. The goal of a player is to form a chain of 5 stones either horizontally, vertically, or diagonally, while simultaneously preventing the opponent from doing so.

At an initial glance, Gomoku appears to be a simple game similar to Tic-Tac-Toe. With rules so simple that you can explain to a three year old and play with anyone who just knows how to place stones on a board, you would be amazed by the fact that Gomoku can be so hard to master. Surprisingly under its simplicity guise, Gomoku is a highly complex game. As is well-known, Go is the most challenging board game in the world. In 1980, Lichtenstein and Sipser [6] proved that Go is PSPACE-hard. In 1983, Robson [7 ] proved that Go is EXPTTIME-hard. In comparison with the computational complexity of Go, Reisch [4] proved in 1980 that Gomoku is PSPACE-complete. To make the matter more puzzling, Gomoku is asymmetrical, meaning that the black player seems to have a bit more advantage to win. In fact, in 1994, Allis [5] proved in his Ph.D. dissertation that there is a winning strategy for the black player for Gomoku. Such an asymmetry would add more challenge to build an AI to win when it plays white. One must know Allis' result does not mean that there is an efficient way for the black player to find the winning

strategy. Finding a strategy to win is still a quite gruesome task due to the sheer amount of possible states the game may generates as the players continue playing. Notably, the branching factor of Gomoku is 210, while the branch factor for Go is 250.

Despite the vast computational complexity challenges, there have been remarkable achievements on building AIs to play the most difficult games. AlphaGo [1] made its historical win over Lee Sedol, one of the best Go Masters in March 1996. The later version AlphaGo Zero is even more powerful. AlphaGo has been quoted to cost around $25 million dollars solely for the hardware [2]. Training a network for some powerful AI like AlphaGo is another incredibly hard task that requires huge amount of game data. Even for Gomoku, when two highly competitive players play the game, it would easily last over 10 minutes to finish. Thus, generating and collecting game data for training and self-learning of building AI would be a tremendous task.

Besides game data generating, Gomoku has three unique properties that would prevent us to from applying existing powerful AI like AlphaGo directly, even though the framework of AlphaGo is quite universal. (1) As shown by Allis in 1994 [5], the strategies for black and white players are asymmetrical. (2) The game length of Gomoku is on average 30 moves, while the average game length for Go is 150. Such a short game length allows far less tolerance for an AI for Gomoku to make a mistake. (3) It is true that Gomoku values locality more than global view, but how to utilize the local potentials without composing the global potentials?

## Method

In this thesis, we will develop an innovative approach to building an AI for Gomoku. We first present an algebraic monomial theory to present the game states and to potential values for each move. Unlike other AIs that have treated the board as an image, such as AlphaGo [1], we instead treat the monomials to represent game configurations and histories. This monomial approach is less intensive for an AI to process when compared to layers of 2D matrices needed for history recordings. Our new approach also allows for more information to be extracted when compared to a series of points on the game board. The most important aspect of our monomial solution lies on its ability to scale so that a light weighted AI can be developed and run efficiently on usual computing devises such as PCs, laptops, tablets and smart phones, without compromising its competitiveness.

# CHAPTER II

## MONOMIALS

The foundation of the work in this thesis is based on algebraic monomials. A monomial is a product of algebraic variables such that the degree of every variable is 1. For example, $x_1x_2x_3x_4$ is a monomial, but $x_1x_2^3x_3x_4$ is not because the degree of $x_2$ is 3. We know that we can play Gomoku on a $15 \times 15$ board or a typical $19 \times 19$ Go board. For simplicity, we focus on a $19 \times 19$ board, and all the work can be easily applied to a $15 \times 15$ board.

A $19 \times 19$ board has 361 grid points, and for each of these points we introduce a variable to represent it. Since the goal of a player is to form a list of 5 consecutive points with the player's stones, either horizontally, vertically or diagonally, we can create a monomial to represent each of such winning configurations. Such a monomial is simply the product of five variables for those five consecutive points. In general, for an $n \times n$ board, if we define a list of c consecutive points, either horizontally, vertically or diagonally, with the second colored stone as a winning configuration, then we devise a formula as shown in Figure 1 to show the total number of possible winning configurations for the board.

$$2(n\,(n - c + 1) + (n - c + 1)^2)$$

Figure 1. Formula used to calculate monomial amount.
n = board size
c = chain size

For the typical $19 \times 19$ Go board, according to our formula, we have a total of 1020 monomials representing all possible winning configurations for a player.

The roles of the algebraic variables and the monomials are symmetrical for the black and white players. Thus, for simplicity, we just need to discuss how these variables and monomials can help a black player play the game. At the beginning of the game, the black player will have a set of 1020 monomials. When the black player places a stone on a point represented by a variable $x_1$, in terms of algebra, this is the same as the black player sets the variable $x_1$ to 1. Considering that the black player has a monomial $x_1x_2x_3x_4x_5$ before setting $x_1$ to 1, then after setting $x_1$ to 1, the monomial $x_1x_2x_3x_4x_5$ becomes $x_2x_3x_4x_5$. If the white player ignores $x_2, x_3, x_4, x_5$, the black player can keep setting $x_2, x_3, x_4, x_5$ to 1 in four additional moves, thus the monomial is changed to 1, implying that the black payer wins the game. In reality, the white player is not so naïve. After the black player sat $x_1$ to 1, the white player could place a white stone on the point represented by $x_2$. In algebra, this is the same as setting $x_2$ to 0. By doing so, the white player basically sets the monomial $x_2x_3x_4x_5$ to 0, thus completely eliminating a possible winning configuration for the black player. The effect of setting a variable to 1 is symmetrical for black and white 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; when the white player sets a variable to 0 to harm the black player, symmetrically he sets the variable to 1 to further his chance to win.

A monomial for a black player is active (or alive), if none of the variable in the monomial has not set to 0 by the white player. A monomial is inactive (or dead), if at least one of its variables has been set to 0 by the white player. A variable in a monomial is called free, if it has not been set to 1 or 0 by a player.

As can be understood from the above discussion, at each move, the black 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 prevent the white player from doing so. As the game progresses, the black player needs to assess his offense strategy and defense strategy. For offense, he needs to find a way to move one his alive monomials to 1. For defense, he needs to eliminate any possible monomial that can be changed to 1 by the white player. The black player needs to find some competitive policy to balance the offense and defense strategies.

## Benefits

### Potential Values for Board Points

At each move, the black player needs to know how to choose a point to increase his chance to win and meanwhile to diminish the white player's chance to win. To do so, the black player needs to find some good way to assess potential values for every remaining move from the current game state and the history of the game. It turns out that the monomials can help 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 $x$, we define the score for $x$ as follows: s(x) = 1, if x is a free variable; $s(x) = 2$, if $x = 1$, i.e., the black player has already placed a stone on the point represented by $x$; and $s(x) = 0$, if $x = 0$, i.e., the white player has already set a white stone on the point represented by $x$.

For any monomial m, the score of m is

$s(m) = \prod_{x\ in\ m} s(x)$.

At any step of the game, with the help of monomials, the value of any point $p$ is defined in the following:

$$v(p) = \sum_{m\ covers\ p} s(m)$$

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.

**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 streamlined 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, we only need to deal with up to 20 monomials, or 5 monomials per orientation. Also, as the game progresses, 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.

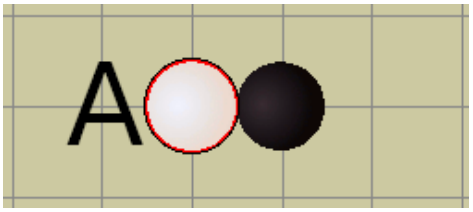


Figure 2. Example of a point no longer sharing a mutual monomial. If point A was to be taken by the black player, it would not influence the black point previously taken by the black player.

**Guidance for 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, we still need to utilize points to carry out locality analysis to for 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. As can be seen in Figure 2, if the black player is about to place a stone on point A, then this point A would no longer affect the previously occupied point, because the white stone created a local blockage between point A and the black stone on the right side.

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, we would need to check all adjacent points 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 checks required, such locality analyses would significantly increase the amount of resources needed to manage the game.

On the other hand, with the help of monomials, many types of local analysis, such as the one mentioned above, would become a breeze. As shown in Figure 2, 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.

**Locality and Incremental Property for Potential Value Updating**

It is worthy of noticing that 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 given below:

$$newValue(q) =$$

$$oldValue(q) + \sum_{m\ alive\ and\ coveres\ p\ and\ q} newScore(m) - oldScore(m)$$

The above locality and incremental properties make fast updating possible for re-calculating monomial scores and point potential scores with respect to a move made by the black (or white) player.

**Decision Making**

Monomials also provide the benefit of easily assessing the potentials for future moves from the current state and the game history data. For a simple example, consider that the black player has alive 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 one of the 2 free variables to win. Symmetrically, this applies to the white player so that the black 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. Take the case in Figure 3 for example. Assume that the AI plays black. If currently the AI has its turn to make a move, then among many possible moves, including in including A, B and C, choosing B 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.

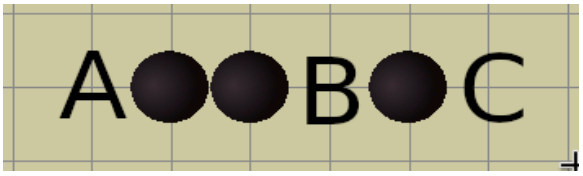


Figure 3. Example of choices the AI may have to decide on.
A, B, C are all valid options if the AI plays white. Otherwise B is the winning choice.

# CHAPTER III

# ARCHITECTURE

The overall architecture of our AI for play Gomoku has gone through many modifications throughout the span of the project. These versions share many features as each was built upon its predecessor. Thus, rather than going through all versions, in this chapter we will discuss 3 major revisions that occurred throughout the project.

## Version 1

The initial design of the AI was implemented with a rather simple architecture with less sophisticated decision making. As seen in Table 1 and 2, all decision makings are based on 2 simple tables containing the basic information pertaining to surviving monomials of the game.

| Active | Value | Orientation | Taken |
|---|---|---|---|
| bool | int | int | bool, bool, bool, bool, bool |

Table 1. Monomial Table.

In Table 1, Active attribute indicates whether a monomial is alive or dead; Value stores the score; Orientation tells the alignment of the monomials is horizontal, vertical, right diagonal or left diagonal; Taken indicates whether each variable of the five underlying variables is taken of not.

In Table 2, we record the potential value for each point.

| Score |
| --- |
| int |

Table 2. Point Table.

We use the monomial table to identify whether a guaranteed strategy to win or lose was present given the current game state and the history of the game states. These strategies may be found based on checking monomial scores.

It follows from the monomial score definition given in Chapter 2 that a monomial may have scores from 0 to $2^5 = 32$. A score of 0 means that the white players has set one of the five variables contained in the monomial to 0, thus making this monomial dead. A sore of 32 means that the monomial achieves a winning case, thus making the black player win the game. As discussed in Chapter 2, once a monomial has a score of 8, then it sheds light for a player to win. With some additional analysis of the free variables in that monomials, we can help the player to follow the light to win. The same can be applied to the white player. We shall check the scores of all active monomials and search for a list of moves to win or to shatter the opponent player's chance to win. If configured correctly, monomials allow for finding an offensive (or a defensive) strategy, a list of several decisive moves. The monomial table has enough information for us to detect those offensive or defensive strategies. As exhibited in Figure 3, if an active monomial with a sore of 8, then an immediate action has to be taken to prevent the potential loss or to guarantee a win, depending on which player has right to move at the moment.

While the monomial table is used to help re-calculate the potential values for points and to search for offensive or defensive strategies, the point table will be used to decide all low priority moves. The potential values of the free points are used to decide which move would be most beneficial. A point with high potential scores signifies that it is covered by multiple active

monomials with relatively higher scores. Choosing such a point will increase the scores of those active monomials, thus creating better chance of victory, because the possibility of generating an offensive or defensive strategy is getting larger.

The above decision making is good and has a tendency to choose an aggressive offensive play style initially, simply because the decision-making focuses on the attack moves. This play style incurs a variety of issues, often ending up losing the game quickly to a competitive opponent player. The main issue is the repetitive decision making that is possible to leave some pattern to be detected by a competitive opponent. Once the opponent detects such a pattern of decision making, the opponent can find ways to deceive the AI so as to defeat it. On the defense side, while the AI can essentially predict and prevent a loss three turns ahead of the current state, it cannot do so farther ahead three or more turns. Moreover, the AI lacks the ability to adaptively learn from the opponent's moves. Based on our experiments, when the opponent finds a way to win, he can simply recreate such a winning play scenario to repeatedly beat the AI.

Another apparent issue is the speed of the architecture. While the AI is capable of making decisions moderately fast, but the AI tends to spend the majority of the time on redundant calculations. These calculations simply consist of updating, in response to each move, the information within the tables such as monomial scores and status changes of points (or variables) contained in monomials. While these calculations are simple and each can be performed quickly, yet the sheer amount of calculations needed would incur a significant impact on the overall throughput of the system. Based on our experiments, these calculations take up to about 80% of the overall time.

## Version 2

Having learned from the previous design and implementation of Version 1, in this section we present major revisions aiming at overcoming the flaws of the previous version. Our priority focuses on improving the speed of the architecture, because the time performance is a bottleneck for carrying out the implementation of all features. Additionally, we hope to further improve the AI's ability to detect decisive moves beyond 3 turns. Alongside this ability for probing ahead, improvements on strategies for decision making are required.

### State Tables

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, we shall introduce additional variables to help manage monomial 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 shall devise new methods for efficient updating of monomials and variables.

### Monomials

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 5 consecutive points horizontally or diagonally. Often, it is good to know the status of the two end points of the line segment. Hence, to maximize the benefits of

monomials, we add two end points to a monomial such that a monomial is represented with 7 points. On the border of the game board, there may 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 choose to 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 a monomial may have. The monomial table, on the other hand, is repurposed as a $1020 \times 1$ 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 mover, thus resulting in a significant boost in speed.

As can be seen in Table 3, 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 we simply denote it as monomial $M^{00}$. There exist 4 statuses for this monomial configuration in terms of 4

| Point 1 | Point 2 | Point 3 | Point 4 | Point 5 | Left Side Occupied | Right Side Occupied |
|---|---|---|---|---|---|---|
| False | False | False | False | False | False | False |
| False | False | False | False | False | False | True |
| False | False | False | False | False | True | False |
| False | False | False | False | False | True | True |
| False | False | False | False | True | False | False |
| False | False | False | False | True | False | True |
| False | False | False | False | True | True | False |
| False | False | False | False | True | True | True |
| . . . | | | | | | |

Table 3. Row 1-8 of a simplified monomial state table (only configuration information displayed).

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 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, $1 \leq x \leq 32, 1 \leq y \leq 4$. 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

$$I(x, y) = 4x + y.$$

It follows from the above analysis, when a monomial in configuration $M^{xy}$ needs to be updated to a new configuration$M^{x'y'}$ , we only need to change its row index $I(x, y)$ to the row index of the new configuration

$$I(x', y') = 4x' + y',$$

which is easy to calculate from $x$ and $y$, 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 remarkable enhancement in system speed, because now we 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, our new method is able to release a significant amount of resources.

**Points**

Like what have been improved for monomials, we will devise new methods to work 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.

First and foremost, points have multiple configurations. While the monomial table is transformed into a $1020 \times 1$ table, the point table is, instead, transformed into a $361 \times 4$ 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, we shall add virtual orientations for those points. By this way, we need to store 4 configurations 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 $x_1\ x_2\ x_3\ x_4\ x_5\ x_6\ x_7\ x_8\ x_9$. This configuration is composed of 5 monomials $x_1x_2x_3x_4x_5$, $x_2\ x_3\ x_4\ x_5\ x_6$, $x_3\ x_4\ x_5\ x_6\ x_7$, $x_4x_5x_6x_7x_8$, and $x_5x_6x_7x_8x_9$. If point $x_3$ is taken by the opponent, then the horizontal configuration for point $x_5$ become$x_4\ x_5\ x_6\ x_7\ x_8\ x_9$s, which is composed of two monomials, $x_4x_5x_6x_7x_8$, and $x_5x_6x_7x_8x_9$. When the AI occupies some points in the configuration, say, $x_7$ and $x_9$, the configuration turns into $x_4\ x_5\ x_6\ x_8$, which is still composed of two monomials $x_4x_5x_6x_8$, an$x_5x_6x_8$d. 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 shall generate all possible states that may occur throughout the game. The number of states for any point is given by the formula in Figure 4. For illustration, we consider the configuration $x_1\ x_2\ x_3\ x_4\ x_5\ x_6\ x_7\ x_8\ x_9$ 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

$4 \times 2^8$ 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 of 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 $x_5\ x_6\ x_7\ x_8\ x_9$. 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 trickier to estimate the exact number of states that can be derived from it. For example, let us continue 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 $x_2\ x_3\ x_4\ x_5\ x_6\ x_7\ x_8$. 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) $x_1\ x_2\ x_3\ x_4\ x_5\ x_6\ x_7$; (b) $x_2\ x_3\ x_4\ x_5\ x_6\ x_7\ x_8$; and (c) $x_3\ x_4\ x_5\ x_6\ x_7\ x_8\ x_9$. 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 additional 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, we can see a general trend of the number of states that can be derived from configurations for point $x_5$: For configurations of length $k$ with $5 \le k \le 8$, 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 $n, n \ge 5$. We present the total number of states in the general setting in Figure 4.

$$f(x) = 4\times 2^{k-1} + \sum_{k=n}^{2n-2} 2^{k-1}(2n+2-k)$$

Figure 4. The total number of states derived from configurations for any given point $x$ on a single orientation. $n$ refers to the size of the monomials. For Gomoku, $n = 5$.

As one can tell from the estimate given in Figure 4, for any given point, finding all possible states that can be derived from its configurations can be quite 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 Figure 5 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 for the point provide us with 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 threating moves and will be discussed in detail later.

**Decision Making**

As mentioned previously the decision making of the initial version of our AI is too 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 some fluke 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 2 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 area  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 1 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 2 turns for wins and losses.

In addition to combos, another crucial enhancement of our 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 rollout 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.

**Performance**

This implementation was overall a huge successful. The performance of the new version of our AI is increased significantly, making it ever much more difficult for a human plyer to win. With the combination of tools at its disposal, we soon realize that it become quite difficult to test our AI with human player, because we can no longer to find a human player to win it. 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 by no means an easy task. Although our 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.

## Version 3

In order to eliminate the flaws encountered in the development of the two previous versions of the architecture, we decide to remove the point tables. This leaves the final version of the architecture to solely rely on monomials to manage the board.

### Goal

Our goal for this thesis 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 make us believe that MCTS is a valuable tool to be included in our system. After many experiments, we realize 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 our AI may loss 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 our 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 this 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 quite beneficial for building a decision tree. Previously, a decision tree may have located a monomial of an old score of $2^3$, but we have to do additional checking to determine if additional action is required to move to 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 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 our implementation, with the help of the new rank scores for monomials, we are able to increase the search tree depth 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 4. With the new system, since the point table is no longer needed and hence reliance on points is removed, the speed of the new MCTS is more than quadrupled despite the doubling of the search tree depth.

**Result**

While our AI was previously unable 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 beating some of the top-10 ranked AI's. For example, for the top $7^{th}$ ranked Gomoku AI, Wine,

the previous versions of our AI achieved only a single win against it during a number of competitions done in a span of several months. However, the new version of our AI has achieved a 20% winning rate over Wine when our AI plays first.

# CHAPTER IV

## MONTE CARLO TREE SEARCH

Monte Carlo Tree Search (MCTS) has been successfully used in AlphaGo [1]. This type of MCTS is included in AlphaGomoku [3], which is a rather direct application of the AlphaGo algorithm. To help probe for competitive offensive and defensive strategies ahead of many moves, we have implemented a similar MCTS component in our system to assist with decision making. Our implementation of MCTS remains practically identical to the one developed in AlphaGo [1], but there is one major difference. Unlike the MCTS used for AlphaGo, in our system a policy value network has not been introduced to work alongside with the MCTS. Thus, at the present stage, our system is incapable of learning from past experiences to better guide the MCTS. To make up for this lack of self-learning ability, human knowledge of the game has been integrated into the decision-making process to assist with MCTS to search for better strategies for offense or defense.

In all other aspects the MCTS in our system functions identical to the standard approach. During the search process, a search tree is built. Each node represents a state the game board may reside in.  The children nodes similarly represent states that can be reached from the parent node within one turn. These nodes are primarily scored based on how many times they are

visited along with a reward value to identify how well the node may perform when compared to its visit count.

Each node can be categorized based on whether it resides on an even or odd layer of the tree. The root resides on layer zero, representing the state of the board when the MCTS is initiated. Afterward, all odd layers represent nodes for the AI's turn to play, while all even layers represent nodes for the opponent's turn. When a rollout simulation occurs, the result ends in one of two states: If a player is victorious a score of +1 is provided to the current leaf node and back propagated to the root. This score affects each player's node differently as the score flips with each node alternatively layer by layer. By this way, 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 rollout simulation is reached, the result is treated as a tie, leading to all scores remaining unchanged for the nodes visited during the back propagation.

All equations such as the upper confidence bound remain the same as those used in Alpha Gomoku [3]. 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 extremely low temperature value to promote exploration for the initial 50 rollout simulations. Afterwards, the temperature is raised, allowing all future simulations outcomes to be based primarily on the mean action value.

In total, the implemented system functions with the help of two MCTS components. The primary MCTS functions like that of AlphaGo [1]. It performs simulations to find the best possible strategy until a time limit is reached. Its counterpart, denoted as the winning MCTS, focuses on identifying whether a winning strategy is present within the board. These two MCTS

components function primarily the same way but differ in node selection for expansion, the winning MCTS tends to prune nodes which fail to guarantee victory.

The winning MCTS has a variety of tweaks to perform its task efficiently. The main change is the moves considered. Due to the nature of the game if one considers a wide variety of options, then it be unfeasible to simulate all the paths within a reasonable amount of time, much less so if one wants to guarantee a winning outcome. With the goal of searching for guaranteed wins we shall concentrate on the moves which force the opponent to react defensively. These moves, in most cases, would leave the opponent with 1 to 3 possible options. As the simulations progress nodes are pruned based on which player they refer to. For a node representing the AI's move, we require that all its children nodes contain a path toward victory. This can be seen as assuring that, no matter how the opponent may react with, there is a list of moves that would lead the AI to win. For the opponent's nodes, on the hand, we require that at least one child node contains a path to win since these nodes referring to the systems turn. Nodes which fail to meet these requirements are pruned from the tree. This results 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 outcome 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 future moves to be based solely on the output tree.

The primary MCTS, on the other hand, functions as a typical MCTS. It performs simulation after simulation until the maximum amount of time is reached. Once this occurs, a point is chosen based on whichever it contains the maximum mean action value. As usual for any MCTS, not all moves are considered. A handful of moves are selected based on assessment

statistics of the AI's and the opponent's monomials. This heuristic effectively ignores moves with less importance, such as those with no nearby neighbors occupied. This selection process for potential moves is used throughout tree node expansion in order to minimize the amount of resources spent on undesirable moves.

The primary and winning MCTS components cooperate with each to find the best strategy of winning. While the primary MCTS can provide us an idea about which move boasts the best probability of winning, yet it is not capable 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 opportunities 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 repeated until a winning path of moves is found, allowing for all future moves to be performed instantly. Our implemented system is able to perform, for every second, nearly 200 simulations of rollout expansion with a maximum depth of 30 moves. With a limit set to 1 minute per turn, our AI can decide its final choice of moves that are backed by over 10,000 simulations.

Even though thousands of simulations are examined, yet there still exist issues with the final decision. Most noticeably, when playing against top rated competitive AI's, losses still occur for our AI. These losses are likely 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 moves, such that not enough simulations are done on either of those moves to truly assess which moves is 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.

Overall, our AI is capable of winning against other top rated, 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 beneficial.

# CHAPTER V

## SUMMARY AND CONCLUSION

By transforming Gomoku algebraic monomials, we are able to successfully create an AI that is capable of playing the game at a highly competitive level. The architecture and tools developed for the AI are all based on monomials. We have tested our AI against the top-ranked Gomoku AI's from the latest Gomocup list. For the current version, our AI is able to win up to the $7^{th}$ top ranked AI. Precisely, for the $7^{th}$ top ranked AI, the winning chance of our AI is about 20% from two separate sets of the games.

Obviously, there is still much more to be done to continue the growth of this AI. First and foremost, the MCTS can be improved with the addition of multi-threading. This would significantly make much more simulations of rollout expansion to be done, so that it become possible to relieve the issue where simulations are equally split among moves showing equal potential. Similarly, the introduction of a neural network to guide the MCTS would be a huge addition. As proved by AlphaGo [1], such a policy value network can be incorporated with the MCTS to help an AI learn from self-play.

REFERENCES

[1] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, *et al*., Mastering the Game of Go with Deep Neural Networks and Tree Search, Nature, 529 (7587): 484–489, 2016

[2] B. Alpert, Artificial Intelligence's Winners and Losers. Nov 4, 2017. https://www.barrons.com/articles/artificial-intelligences-winners-and-losers-1509761253

[3] Z. Xie, X. Fu, J. Yu, AlphaGomoku: An AlphaGo-based Gomoku Artificial Intelligence Using Curriculum Learning, arXiv, 2018

[4] S. Reisch, Gobang ist PSPACE-vollständig (Gobang is PSPACE-complete). Acta Informatica. 13 (1): 59–66, 1980.

[5] V. Allis, Searching for Solutions in Games and Artificial Intelligence (Ph.D. thesis). University of Limburg, Maastricht, The Netherlands, 1994.

[6] D. Lichtenstein, M. Sipser, Go Is Polynomial-Space Hard, Journal of the ACM. 27 (2): 393–401, 1980.

[7] J. M. Robson, The complexity of Go, Information Processing, Proceedings of IFIP Congress. pp. 413–417, 1983.

# BIOGRAPHICAL SKETCH

Oscar Garcia graduated from Mission High School in 2013 and chose to continue his education at South Texas College. As two years flew by, he graduated with an Associate's Degree in Computer Science in the spring of 2015. Afterwards, he chose to continue his education at The University of Rio Grande Valley. An additional three years later, he obtained his Bachelor's Degree in Computer Science in 2018. Rather than stopping there, he once again chose to continue his education with his goal to receive a Master of Science Degree in Computer Science from The University of Rio Grande Valley, and his goal was achieved in May 2020. He can be contacted by his personal email of oscar3720@gmail.com.