COMPARISON OF GENETIC ALGORITHM TRAINING METHODS AS APPLIED TO TIC-TAC-TOE

A research paper submitted to the
Faculty of the Philippine Science High School –
Cordillera Administrative Region Campus
in fulfillment of the course requirements in
Science, Technology, Engineering and Mathematics Research 3

VASH PATRICK B. ANCHETA
DIEGO SULAYMAN R. PASCUA
RESH VNZI S. TOGUEÑO

21 May 2020

APPROVAL SHEET

In fulfillment of the requirements in Science, Technology, Engineering and Mathematics Research 3 (STR 3), this research entitled, "COMPARISON OF GENETIC ALGORITHM TRAINING METHODS AS APPLIED TO TIC-TAC-TOE" is submitted by Vash Patrick B. Ancheta, Diego Sulayman R. Pascua and Resh Vnzi S. Togueño on 21 May 2020

KAYE MELINA NATIVIDAD B. ALAMAG
Research Adviser

This research paper is hereby accepted by the Research Council.

CONRADO C. ROTOR, Jr., Ph.D.
Chair

MELBA C. PATACSIL
Co-chair

JAY JAY F. MANUEL
Member

MARITES P. RIVERA
Member

RICARIDO M. SATURAY, Jr.
Member

FREDA M. WONG
Member

ACKNOWLEDGEMENT

We are grateful for our friends and family for their continued support in our continuous lives. We are also thankful to our teachers, research teacher, and research adviser for their unwavering assistance in having the research completed. Without these people, this research would never be successful.

ABSTRACT

Vash Patrick B. Ancheta, Diego Sulayman R. Pascua and Resh Vnzi S. Togueño. Philippine Science High School – Cordillera Administrative Region Campus, 21 May 2020.
"COMPARISON OF GENETIC ALGORITHM TRAINING METHODS AS APPLIED TO TIC-TAC-TOE"

Adviser: **Kaye Melina Natividad B. Alamag**
Machine learning methods are algorithms where machines are not explicitly programmed to do what is tasked but rather, learns how to perform the task. An *m, n, k* game is a game where there is an $m \times n$ grid and two players alternate turns trying to earn k pieces adjacent to each other horizontally, vertically or diagonally. The *m, n, k* game to be used in the research to test the MLMs is Tic-Tac-Toe, configured as 3,3,3. The study utilizes an existing genetic algorithm to be used as a control setup. This genetic algorithm is modified to be controlled by move generators using the Controlled Elite Preservation operator and the resulting genetic algorithms are compared with regard to performance. Using an ANOVA Test at $\alpha = 0.05$, no significant difference in performance was found between the unmodified and modified genetic algorithms. This study provides a backbone for research involved in the transmission of knowledge between "smart" artificial intelligence and "naive" intelligence, raising the question on whether the evolution of a genetic algorithm can be better affected by a move generator in other components.

TABLE OF CONTENTS

ABSTRACT **iii**

List of Figures **v**

List of Tables **vi**

List of Listings **vii**

CHAPTER I: INTRODUCTION **1**
Background of the Study 1
Objectives of the Study 1
Significance of the Study 2
Scope and Limitations of the Study 2

DEFINITION OF TERMS **3**

CHAPTER II: REVIEW OF RELATED LITERATURE **4**

CHAPTER III: MATERIALS AND METHODS **10**
Research Design 10
Locale of the Study 10
Materials and Research Instruments 10
Procedures 11
Treatment of Data 14

CHAPTER IV: RESULTS AND DISCUSSION **15**

CHAPTER V: CONCLUSION AND RECOMMENDATIONS **19**

LITERATURE CITED **20**

APPENDICES **22**
Appendix A: Project Plan 22
Appendix B: Raw Data 25
Appendix C: Statistical Tables 26
Appendix D: Documentation 27

LIST OF FIGURES

Figure	Title	Page
1	Game tree	5
2	Flowchart of GA training	8
3	Procedural Flowchart	11
4	Genome of an organism	12
5	Notched box plot of treatments	15
6	Population-best fitness graph of control	16
7	Population-best fitness graph of RMG-controlled	17
8	Population-best fitness graph of UMG-controlled	17
9	Network chart	22

LIST OF TABLES

Table	Title	Page
1	Example of Organisms Sorted Through a Classifier System	9
2	Task Lists and Duration	22
3	Task Schedule Management and Personnel Assignment Plan	23
4	Material and Equipment Sourcing Plan	24
5	Risk Management Plan	24
6	Generations per Treatment	25
7	ANOVA Table	26

LIST OF LISTINGS

Listing	Title	Page
1	Base code of genetic algorithm	27

CHAPTER I
INTRODUCTION

Background of the Study

Machine learning (ML) is vast—it is used in different situations such as spam detectors, web search engines, photo tagging applications and game development (Sharma, 2016). There have been researches that are aimed at improving the implementation of ML in various games. A category of games under investigation through ML is the set of *m, n, k* game games, comprised of games where there is an $m \times n$ grid and two players alternate turns trying to earn k pieces adjacent to each other horizontally, vertically or diagonally (Hayes & Loge, 2016). Among the most common examples of *m, n, k* games are Gō, Othello, and Chess. Tic-Tac-Toe, the game under investigation in this study, is an example of an *m, n, k* game. A Tic-Tac-Toe board is composed of three rows and three columns, and requires three adjacent pieces of the same player to render a win, thus it is considered to have a $3, 3, 3$ configuration.

Improvements in ML have lead to the development of artificial intelligence (AI) players that can beat even the most competitive human players around the world. Machine learning methods (MLMs) are algorithms where machines are not explicitly programmed to do what is tasked. Rather, similar to its namesake, MLM-trained machines are capable of performing tasks given its own internal code without any human interference. In short, the machine *learns* (GeeksforGeeks, n.d.). An example of an MLM is the genetic algorithm (GA).

This study aims to develop multiple GAs with different elite preservation methods and compare their performance in Tic-Tac-Toe based on the possible situations.

Objectives of the Study

General Objective

- To compare the effectiveness of trained genetic algorithm (GA) organisms among each other as applied to Tic-Tac-Toe

Specific Objectives

1. To implement known heuristics into Python code
2. To train organisms of an implemented GA using different move generators (MGs)
3. To compare the development of the performance of trained GA organisms among each other within 500 generations

Significance of the Study

This study contributes to the body of knowledge in ML. Through this study, more can be known about how information gained from one method of AI can be passed on to another mechanism of AI through training. This sheds light on how information from one AI player can be transmitted to an MLM such as GA. This, by extension, can improve the comprehension of how machines can learn strategies in games from one with greater skill.

Scope and Limitations of the Study

This study focuses only on Tic-Tac-Toe and not other games such as Chess or Gō because it is the simplest game to conduct the research on heuristics, namely the training of the GA under different move generators (MGs). The complexity of the board game is not relevant to the study because the focus of the research is to compare the effectiveness of trained GA organisms given an *m, n, k* game. Applying these heuristics on other *m, n, k* games however are beyond the time frame of the research. Only three GAs were developed in this study. The first is a Python implementation of the GA in the work of Bhatt et al. (2008). Using developed AI, the second and third are modified implementations of the same GA. The performance of each GA is based on how many generations it takes for the GA to find a no-loss first player for Tic-Tac-Toe. This basis for comparison of performance, specifically using the skill of an organism as a first player, is due to the fact that Python is known for being slow. In line with this, indices are 0-based in this paper, as they are in Python.

DEFINITION OF TERMS

***m, n, k* game** – a game where there is an $m \times n$ grid where two players alternate turns trying to earn k pieces adjacent to each other horizontally, vertically or diagonally

Artificial Intelligence (AI) – program that simulates human actions, can simulate a human player in a game.

Allele – the configuration value of a gene given a unique organism

Genetic Algorithm (GA) – algorithm that simulates natural selection and biological reproduction to produce solutions to a problem

Gene – representation of a unique and distinct situation in a game given the game rules

Genome – mapping table of genes with corresponding alleles

Machine Learning (ML) – a heuristic where a program learns rather than strictly follow a given instruction

Machine Learning (MLM) – algorithms used in machine learning

Organism – an algorithm represented by a genome

Probability Valuation (PV) – classical probability of a player winning at a given game state

Tic-Tac-Toe – *m, n, k* game configured as 3,3,3

CHAPTER II
REVIEW OF RELATED LITERATURE

Effective Computer Algorithms on Tic-Tac-Toe

There have already been precedents in investigating the proper method of winning classic Tic-Tac-Toe. Examples such as prioritizing the center or placement of pieces in the opposite cell of the opponent's previous move are frequently cited as techniques to beat the opponent (Aycock, 2002).

These examples of strategies were translated programmatically by (Barrat, 2019). Using data gathered from manually played games, an "unbeatable" AI was coded. The result was an AI that could not be beat by a "smart" player. Barrat (2019) took feedback from interested players that were able to find exploits in his system, patching them up as the code matures. The code involves random choices as well, leading to an AI that is not easily predicted. The AI itself is effective as the first player and was unbeatable when tested. When tested as a second player, however, the AI has a small chance of being beat due to its random nature. This was discovered after playing it against a completely random–moving player.

Multiple studies have commenced on how many distinct games one can play in Tic-Tac-Toe. Schaefer (2002) argues that the common answer of 9! or 362,880 possible games is misleading for it ignores the symmetrical properties of the game. There are games that have exactly the same pieces placed on the board but are oriented differently in relation to the board. For example, a gametree turned 90° is considered a distinct game state in the 9! calculation. According to Schaefer (2002), it is of more concern to count game states that end when there are k adjacent pieces. Using this method of counting game states, a total number of 765 distinct game states was computed.

Numerous studies have been performed on the most effective algorithms for a computer program to win in Tic-Tac-Toe. In organizing the results of a computer algorithm, one of the most common methods of organization is the use of a game tree. A game tree is a collection of all possible game states arranged in chronological order. The root node represents the

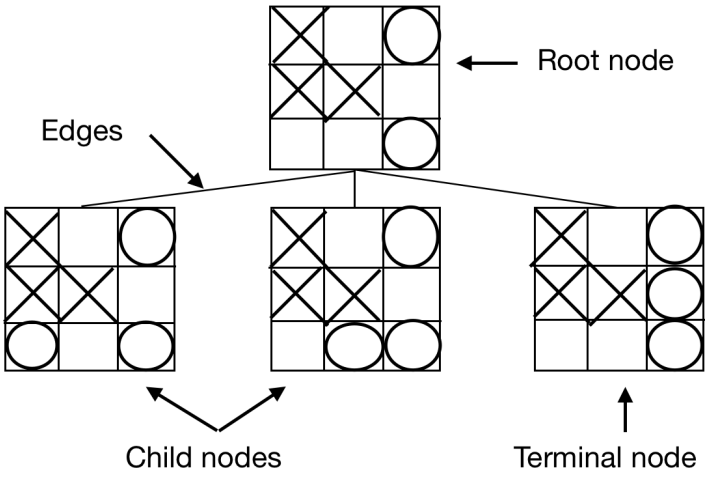

Figure 1. Game tree.

current state, its child node the set possible game states. The edges represent the moves and the terminal nodes represent game states that indicate a completed game. The game tree begins from the root node and branches out into nodes that have their own children (Adamchik, 2009). A diverse game tree is optimal for exploring the capabilities of the different machine learning methods to be used in the study.

A study by Cranenburgh et al. (2007) is concerned with the use and implementation of a heuristic known as depth-first search. Depth-first search is a machine learning method that utilizes *backtracking*. When the algorithm encounters a terminal node, it returns to the previous nodes to find other possible nodes, hence the term *backtracking* (HackerEarth, 2019). According to Cranenburgh et al. (2007), a higher depth search with more game states, leads to a higher win rate or at least, leads the program to reach a draw better.

Researches that deal with the more complex Tic-Tac-Toe variant Ultimate/Super Tic-Tac-Toe seek to find the patterns and implementing these patterns into AI. Analogous to how classic Tic-Tac-Toe is symmetric, a study by George and Janoski (2016) specifies the rotational and reflectional symmetry of Ultimate/Super Tic-Tac-Toe. A study by Lifshitz and David (n.d.) deals with the use of a mixture of heuristics and algorithms such as MiniMax (an algorithm that traverses the whole game tree) and ExpectiMax (an algorithm that analyzes

the expectations of the opponent's moves). Given a controlled set of parameters, ExpectiMax won against a random algorithm most of the time. When played against MiniMax however, ExpectiMax was more likely to lose. This is in agreement with the aforementioned conclusion that an increased depth produces better results. A caveat for this machine learning method is the drastic duration of time the algorithm requires with each increase in depth. The higher the depth of the depth-first search, the longer the time needed for the algorithm to provide the most optimal move.

Machine Learning Methods

Various methods of machine learning have been discovered prior. One of the earliest examples of machine learning methods is Hexapawn formulated by Martin Gardner. The game is composed of a 3×3 board with two sets of three pawns in a row on opposite sides of the board, and the players move alternately. The game is typically played by a human (who always goes first) and an AI player. A player wins the game by accomplishing one of three goals: move a pawn to the opposite edge, capture all enemy pieces, or leave the enemy with no moves. Because of the small scale of the game and its symmetry, all the possible game states can be represented in 24 cards. Each of these cards contains a move performed by the human player and arrows of different colors representing the possible moves the AI player can take (Ortiz, 2017). The AI player contains a matchbox for each card representing a game state. These matchboxes contain beads that correspond to the possible moves the AI player may make. During play, the beads are chosen at random and the AI player moves according to the bead taken. If the move made by the AI player leads to the loss of the AI, the bead corresponding to the move is taken away. Otherwise, the bead is returned to the matchbox. After 50 games, the AI player is practically unbeatable as game states with high probability of loss is reduced (Gardner, 1958).

Genetic Algorithm

Living organisms exhibit a level of problems solving that is almost impossible to be recreated by a computer scientist. Even more, the complexity that living organisms show is one that is incredibly enviable to a computer scientist wanting to achieve some problem solving program; problems that computer scientists have spent endless amounts of intellectual effort on have been solved by living organism without any thought, relying on the process of evolution.

Due to this, many researchers have began emulating the process of evolution on their algorithms, subjecting their own programs to a process of reproduction and natural selection. This method is widely used in the realm of machine learning, since it eliminates a great hurdle in software engineering: programmers need not to specify the individual aspects of each program and the means that actions are carried out.

There are two aspects of the evolution process that computer scientists want to emulate: natural selection and reproduction. "Natural selection" is the process of choosing which solutions survive to make a new generation of algorithms, the survivors govern the characteristics of next generation. Reproduction is the recombination of genes from parent to offspring, making the "child" algorithm of an organism have similar characteristics to their two parent algorithms.

Programming natural selection is done by giving an algorithm a test of fitness, going through numerous iterations, then giving a fitness score based on that performance (Hochmuth, 2003). Fitness scores are usually calculated based on the ratio of games won, as shown in Equation 1. The higher the fitness score, the higher the changes of winning with the solution or the move done in the game. Algorithms with low fitness scores are discarded and those with high fitness scores move on to reproduction.

$$f(x_i) = \frac{n_{\text{lost}}}{n_{\text{games}}} \tag{1}$$

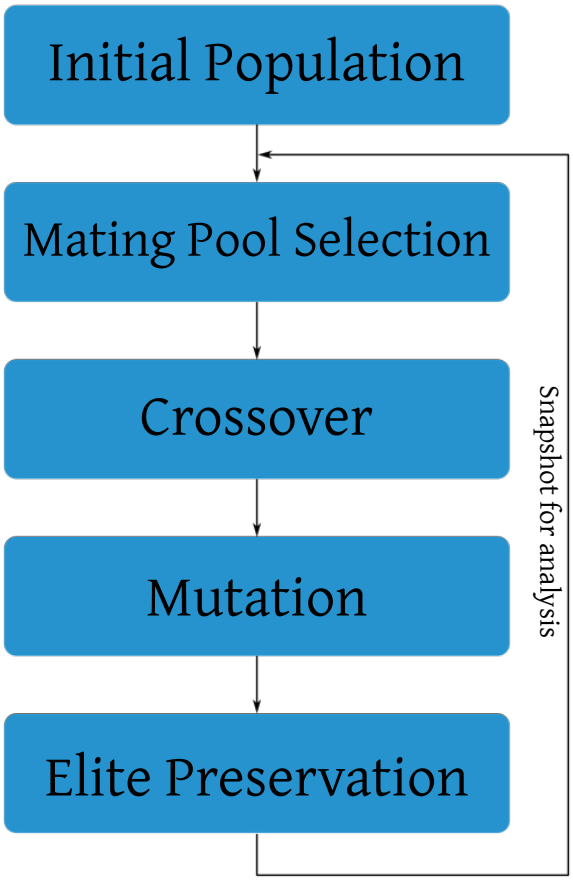

Figure 2. Flowchart of GA training.

The emulation of reproduction is far more complicated. Originally, it was based on random mutation of the algorithms, however this oftentimes produces algorithms that would not run, or are drastically different from the intended purpose of the program. Later development focuses on adding together characteristics from the two parents. This was also limited since this could only be done to characteristics that could be added meaningfully (Holland, n.d.).

Currently, reproduction is done by means of a classifier system. A classifier system is a system where conditions and actions are represented by strings of ones and zeroes corresponding to the presence or absence of that characteristic. For example, as shown in Table 1, since humans have eyes and opposable thumbs and require oxygen, but do not have wings or gills, humans can be recorded as [10011], while the only recorded characteristic of the bacterium that is present is that it requires oxygen, so it is recorded as [00001]. Reproduction can now be done on these "genes" by recombining different genes and making new offspring. In the computer science world, these aspects are usually very basic and primitive, but with strings with lengths reaching tens of thousands of bits long.

Aspect	Human	Fish	Bacterium
opposable thumbs	1	0	0
wings	0	0	0
gills	0	1	0
eyes	1	1	0
requires oxygen	1	1	1

Table 1. *Example of Organisms Sorted Through a Classifier System*

CHAPTER III
MATERIALS AND METHODS

Research Design

This Developmental Research is composed of two components: Software Development and Data Collection. In Software Development the code for the move generators and genetic algorithms are initialized, and the efficiency of its implementation is optimized. In Data Collection the genetic algorithms were simulated and after that, the fitness data collected was analyzed using *R*. The independent variable is the elite preservation method. The dependent variable is the number of generations the specific GA takes to find a no-loss solution. Extraneous variables such as the software specifications can be held constant by the use of the same software such as the operating system and the same Python version (3.8.3 64-bit). The study was not affected by hardware specifications as it concerns the number of generations instead of the time taken on the system.

Locale of the Study

The software was developed and data was collected mainly at the Philippine Science High School – Cordillera Administrative Region Campus. The training and data collection occurred at the aforementioned location on various personal computers.

Materials and Research Instruments

The software was developed with Python 3.8.3 on KDE Neon 5.18 using Visual Studio Code 1.45.1 and hosted on GitHub. A link to the repository hosting the software is located in Appendix D. The developmental computer is equipped with 4 GB of Random Access Memory (RAM). Various personal computers were used to perform the training.

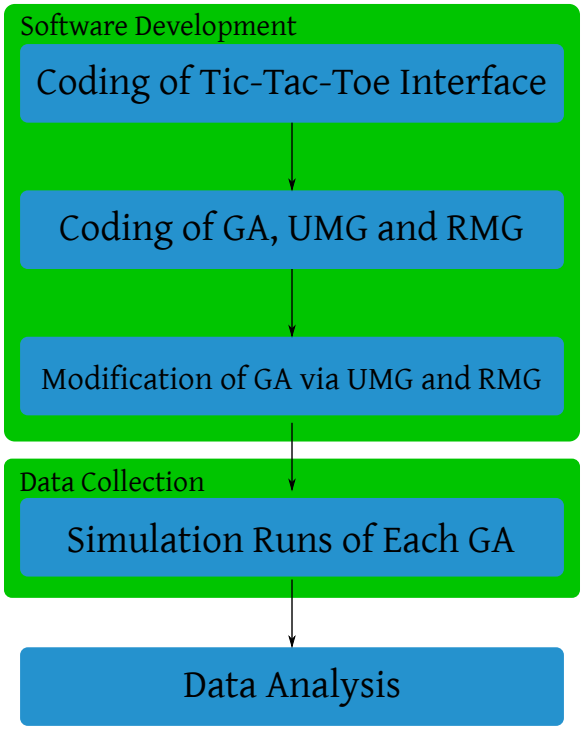

Figure 3. Procedural Flowchart.

Procedures

Software Development

The study aims to compare the performance of the genetic algorithms with different elite preservation methods. The three are: an unmodified Python implementation of the work of Bhatt et al. (2008), and two with modified elite preservation methods based on MGs. A preliminary for the accomplishment of such task is a game engine for the interaction between trained GA organisms. This was coded in Python, as application programming interfaces (APIs) for machine learning have already been implemented in Python.

After the development of a platform for play of Tic-Tac-Toe between trained organisms, the development of the required MGs followed. The Random Move Generator (RMG) returns one of all possible moves in Tic-Tac-Toe with equal probability. Another MG is Unbeatable Move Generator (UMG).

UMG is based off of the work of Barrat (2019). It is stated to be unbeatable, however, the Move Generator has a small chance of being beat as the second player. This makes an

ideal opponent for trained organisms, given that the GA will find an exploit eventually. In line with this, the GA might be able to use said exploit to have better performance.

Genetic Algorithm. The organisms to be trained were implemented in Python as well. The GA was composed of unique genomes with genes for each possible game state with alleles that correspond to the moves to be taken. Following the flow represented in Figure 2, the operators are adapted from Bhatt et al. (2008). The only aspect of the GA that differs from the implementation of that in Bhatt et al. (2008) is the organism representation.

Organism Representation. Organisms are represented with genomes, as shown in Figure 4. Each gene in the genome represents a game state that considers rotation and symmetry using an implementation of the game-base used by Bhatt et al. (2008). These genes have a corresponding allele that represents the move the organism will take.

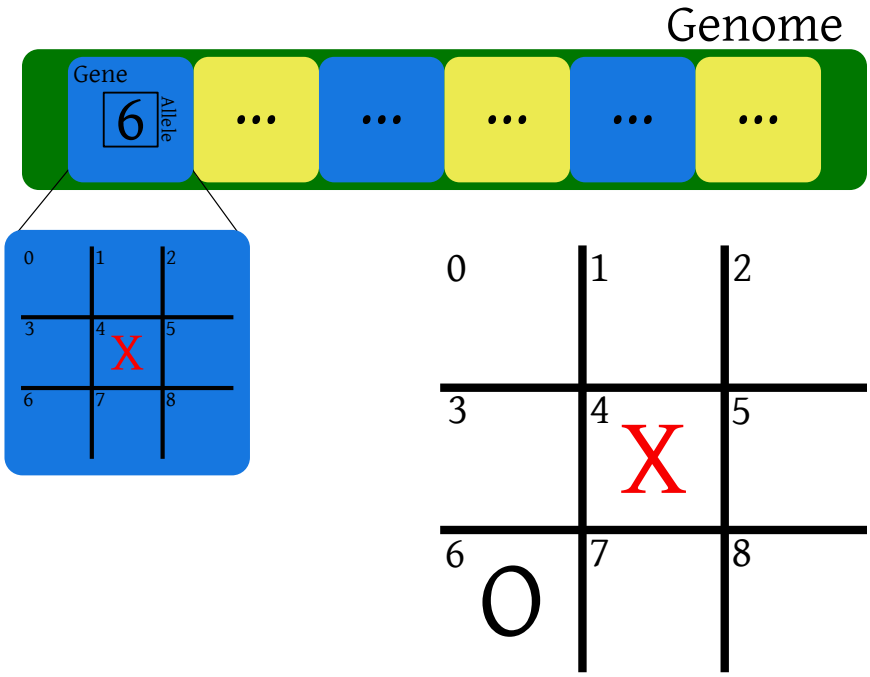

Figure 4. Genome of an organism.

Initial Population. For each game state represented by genes in the genome, the organism is given a random possible move for that game state. Any valid move in a given game state is considered in choosing the allele randomly.

Selection Operator. The following niched fitness equation (Bhatt et al., 2008) was used to modify the implementation of the Stochastic Universal Selection operator by Panchapakesan (2014):

$$f_{\text{niched}}(x_i) = \frac{1}{m} * [1 - f(x_i)]$$

where m is the number of organisms with the same fitness. Using $\frac{1}{m}$ as a factor produces a niching effect (Bhatt et al., 2008). The $[1 - f(x_i)]$ quantity produces the opposite pattern compared to Equation 1, where instead of zero as the perfect fitness and one as the worst fitness, zero is the worst fitness and one is the perfect fitness. This was done due to the fact that SUS is inherently maximizing (Bhatt et al., 2008). In this study, half of the population is selected via SUS.

Crossover Operator. Similar to that of Bhatt et al. (2008), 50 cross-sites are randomly chosen per parent pair which is used to take alternate series of alleles from each parent. This produces two new offspring with alleles from each parent. The mating pool is linear, with parent pairs formed from a mating pool individual and the individual adjacent to it. If the individual is last in the linear mating pool, it is paired with the first individual in the mating pool.

Mutation Operator. Using random reset mutation, n_m genes are mutated based on the best fitness in the population. This basis is formalized by Bhatt et al. (2008) in the following equation:

$$n_m = 250 * f_{\text{minimum}} + 10$$

Controlled Elite Preservation. In the control genetic algorithm, the organisms from the population before crossover, the population before mutation, and the population after mutation are sorted by fitness in increasing order. The solutions with sorted index $j - 1$ are chosen:

$$j(i) = i + 2N * \frac{(i-1)(i-2)}{(N-1)(N-2)}, \quad i = 1, 2, \ldots, N$$

Modified Controlled Elite Preservation. The following modifications were made to the elite preservation method to produce two other GAs: In UMG-Controlled Elite Preservation, the organisms are sorted by the number of losses of each organism when played against the UMG for 300 games. In RMG-Controlled Elite Preservation, the organisms are sorted by the number of losses of each organism when played against the RMG for 300 games.

Data Collection

In the beginning of the GA the organisms were behaving randomly. The fitness function for the organisms was based on the possible results based on an unpredictable opponent, thus every possible win, loss and draw is computed and plugged into Equation 1 to retrieve the fitness of an organism. Through natural selection the "best" organism prevails. When the "best" organism has zero fitness, the GA stops and snapshots the population for the last time. If no zero-fitness organism is found before 500 generations have passed, the GA stops itself. Using this method, 30 simulations per elite preservation method was run, with 100 organisms per population. The number of generations before a simulation encounters a zero-fitness (based on Equation 1) or no-loss solution is recorded for each simulation.

Treatment of Data

Statement of Hypotheses

$$H_0 : \mu_{\text{UMG}} = \mu_{\text{fitness}} = \mu_{\text{RMG}}$$

$$H_a : \text{The means are not all equal}$$

Analysis of Data

The data was analyzed through the R with the use of Analysis of Variance (ANOVA) which gives the p-value to test the hypotheses at a given confidence interval. The study utilizes $\alpha = 0.10$. Should the alternative hypothesis be true, Tukey Honest Significant Differences (Tukey HSD) is applied to locate the different mean.

CHAPTER IV
RESULTS AND DISCUSSION

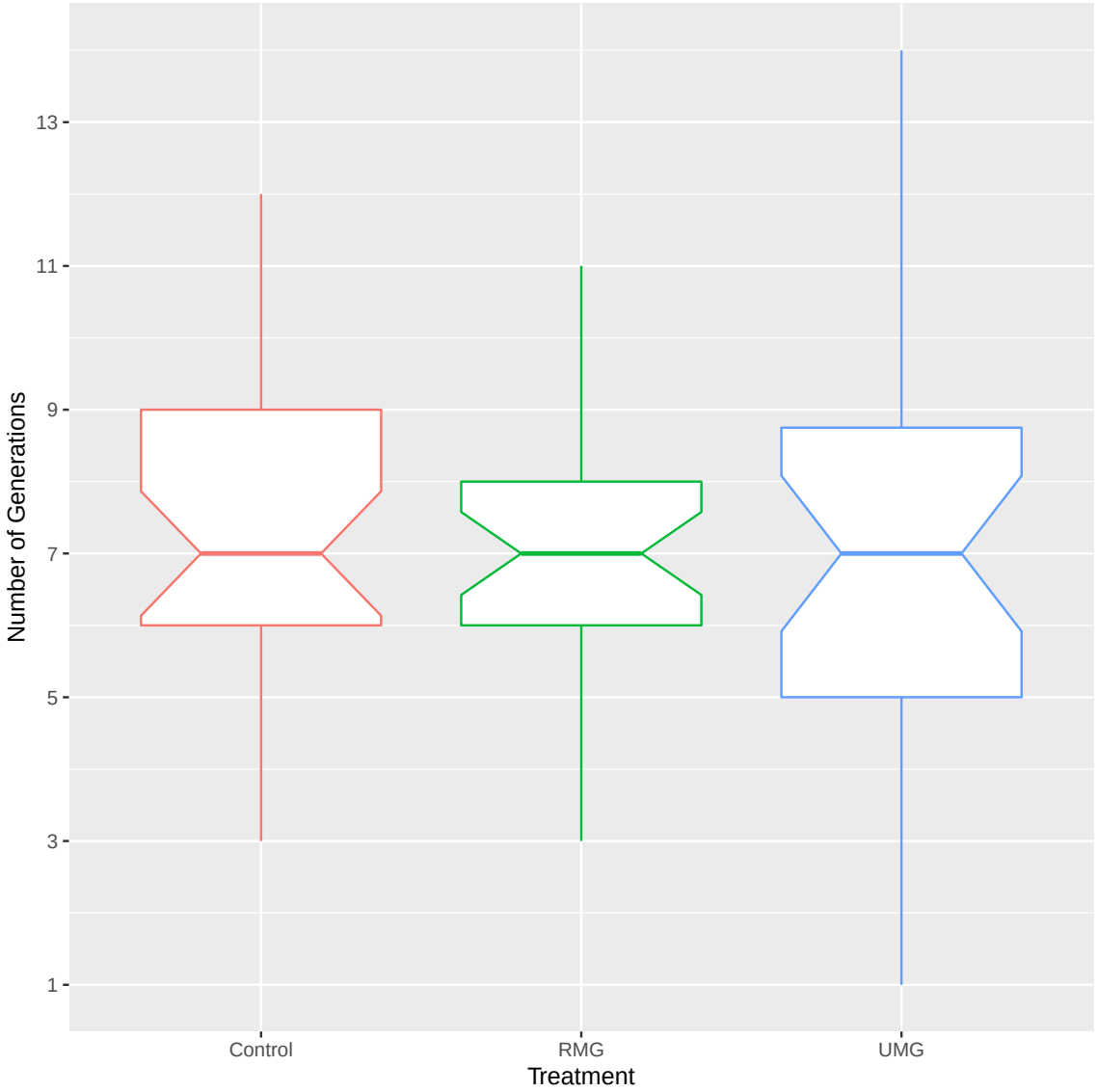

Figure 5. Notched box plot of treatments.

A notched box plot of treatment results is shown in Figure 5. This box plot denotes the interquartile range (IQR) where 50% of the data is located. It is shown that each treatment has a median of seven (7) generations before termination, and the population median is shown by the notches to be within six to eight generations. The UMG-controlled GA shows a larger spread while the RMG-controlled GA shows more consistent data points.

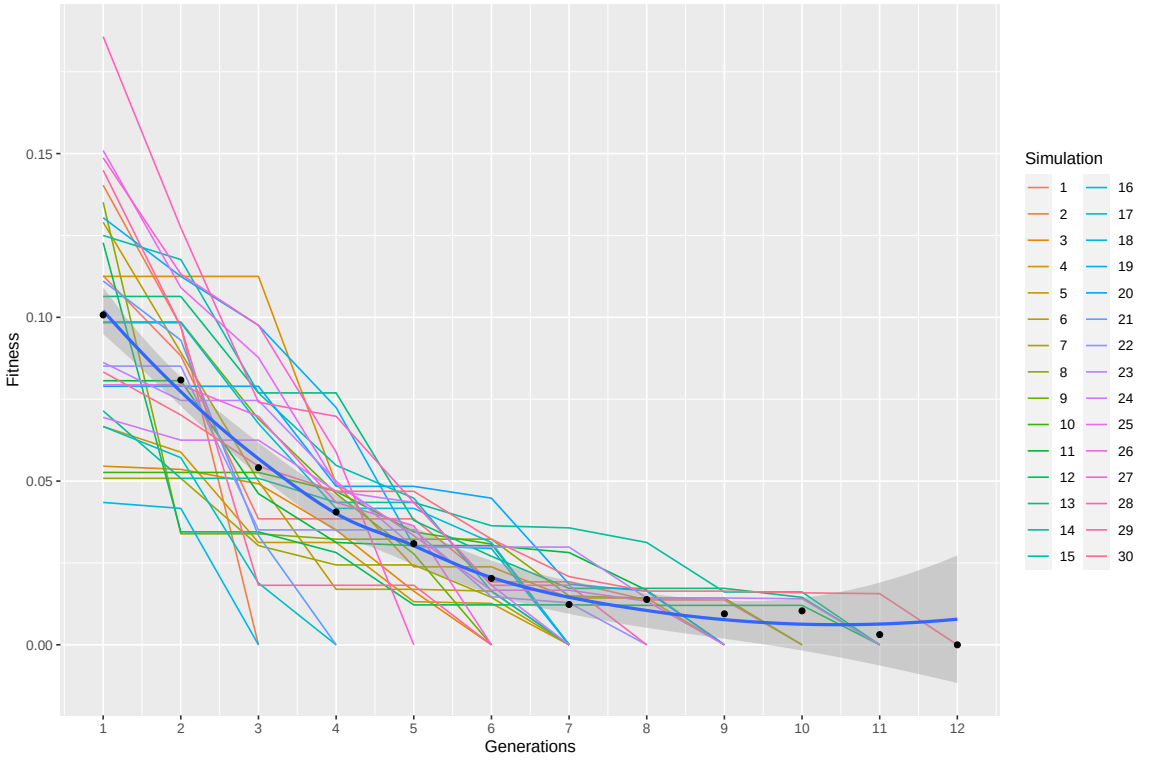

Figure 6. Population-best fitness graph of control.

The large spread in the data points of the UMG-controlled GA is also evident in Figure 8. It is easily shown by the wide standard error bar depicted along the smoothed regression model compared to the standard error bars depicted in Figure 6 and Figure 7. Figure 8 also depicts unique cases where population-best fitness suddenly increases by a large margin.

This evident spread might invalidate the ANOVA test, thus Bartlett's Test for Homogeneity of Variances was applied to the data via *R* (NIST/SEMATECH, 2013). This test resulted in $\chi^2 = 5.1019,\ df = 2$ with a p-value of 0.07801. Said p-value is greater than $\alpha = 0.05$, thus the assumption of homogeneity of variances that is necessary in ANOVA is maintained.

ANOVA was then used to test the null hypothesis that the means of the treatments are equal. This returned an F-value of 0.264 and a p-value of 0.768. The p-value is greater than $\alpha = 0.05$, therefore the null hypothesis that the means of the treatments are equal is not rejected.

It can be noted in Figures 6, 7, and 8 that the GAs stop on or before 15 generations. This

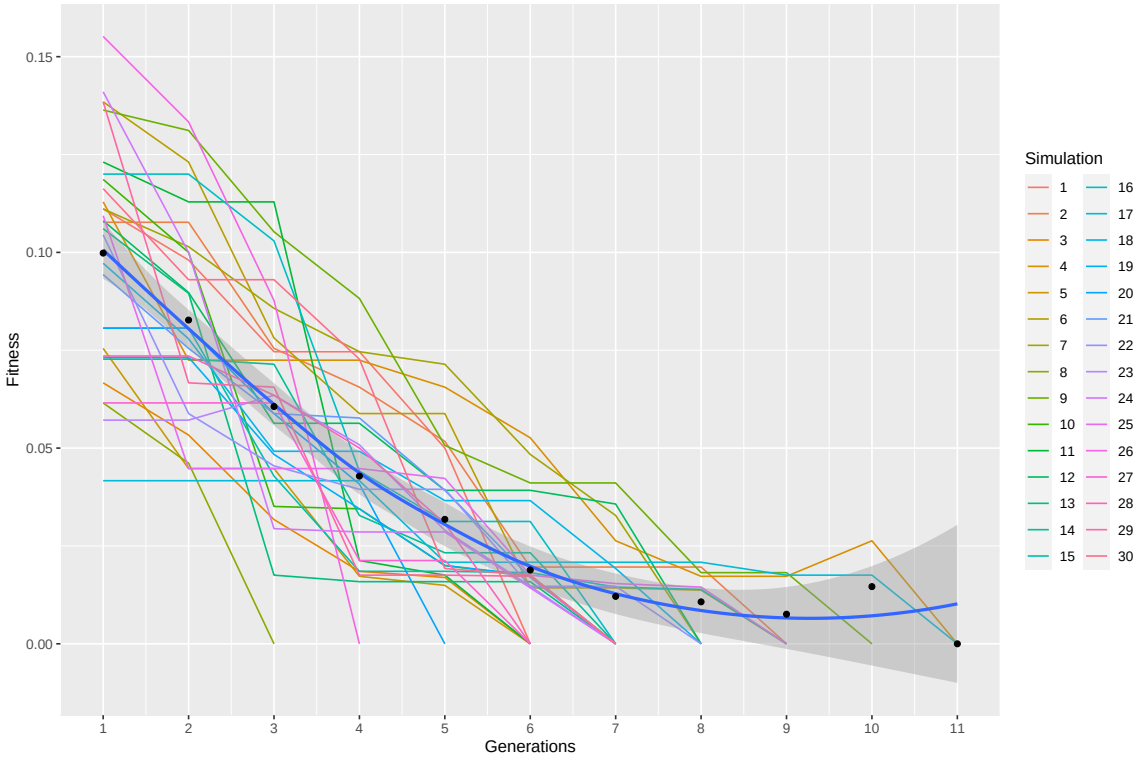

Figure 7. Population-best fitness graph of RMG-controlled.

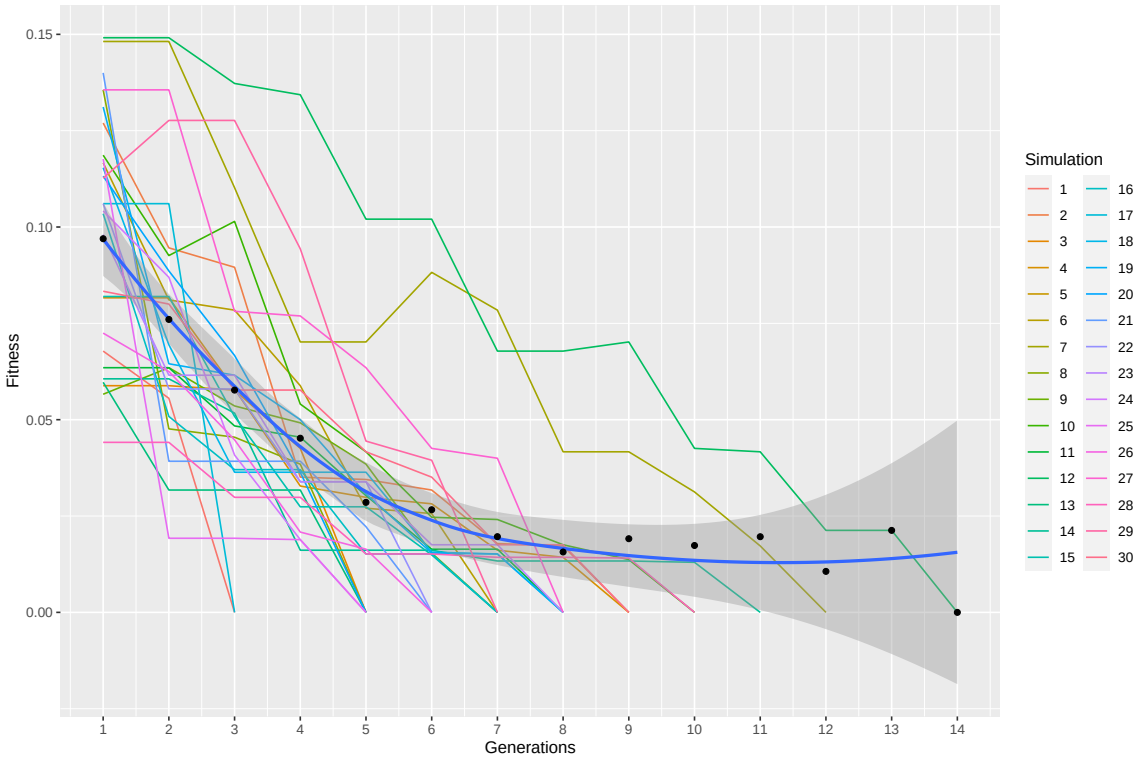

Figure 8. Population-best fitness graph of UMG-controlled.

is because the GAs can find no-loss solutions before even surpassing the 500 generation mark. The 10-generation worst case tabulated by Bhatt et al. (2008) agrees with these findings. The results for each GA are comparatively lower than the 500-generation upper bound used by Bhatt et al. (2008). This can be explained by the fact that the first player is more likely to win in Tic-Tac-Toe (Cranenburgh et al., 2007), thus the game is inherently biased towards the first player.

The ANOVA test results assert that the modifications have no effect on the number of generations a GA requires to find a no-loss solution. This means that the application of an unbeatable opponent did not promote the genetic evolution of the GA towards a no-loss solution. This can imply that either the modification was made in a less-effective component, or the application did not change any behaviour of the GA at all. The latter implication can be disproven by the discrepancies produced by the UMG-controlled GA demonstrated in Figure 8. This behaviour shows that in some cases, the alignments of the UMG goes against the alignments of the fitness function. Such behaviour demonstrates that the UMG modification does in fact, affect the evolution of the GA in no small way.

In the case that the UMG modification was placed in a less-effective component of the GA, it could be implied that there is no effective component in the GA that can accommodate the modification, or that there is one that is not specifically in the Controlled Elite Preservation operator. To prove whether there is an effective component in the GA that can utilize the "unbeatability" of the UMG or not requires extensive research far beyond the scope of this study.

Either way, this study shows that modifying the Controlled Elite Preservation operator of the GA does not significantly improve the performance of the GA. An observation that might be of use, however, is the fact that the UMG-controlled GA shows promise of increase in variability of the GA, which can be used in finding more diverse solutions in the search space.

CHAPTER V
CONCLUSION AND RECOMMENDATIONS

In essence, the stated objectives: to train organisms with MGs and to compare their performance, has been achieved through the modifications to the Controlled Elite Preservation operator. This study targeted the Controlled Elite Preservation operator specifically, thus further research is required to acquire more comprehensive data regarding the effects of having a "smart" opponent control the evolution of a GA.

The study resulted in proof that modifying the Controlled Elite Preservation operator does not improve the performance of the GA. This opens up more questions, on whether or not the application of MG will ever affect the behaviour of a GA. It is also recommended to involve testing the performance of the GA as a second player. Furthermore, this research serves as a foundation for more extensive investigations into the effects of "smart" AI on the learning of other AI.

LITERATURE CITED

Adamchik, V. S. (2009). *Game trees*. Retrieved October 19, 2019, from https://www.cs.cmu.edu/~adamchik/15-121/lectures/Game%20Trees/Game%20Trees.html

Aycock, R. (2002). How to win at Tic-Tac-Toe, 14.

Barrat, R. (2019, April 9). *Robbiebarrat/unbeatable_tictactoe*. Retrieved April 27, 2020, from https://github.com/robbiebarrat/unbeatable_tictactoe

Bhatt, A., Varshney, P., & Deb, K. (2008, July 12). In search of no-loss strategies for the game of tic-tac-toe using a customized genetic algorithm. In *Proceedings of the 10th annual conference on Genetic and evolutionary computation*. https://doi.org/10.1145/1389095.1389269

Cranenburgh, A. V., Samid, R., & van Someran, M. (2007). Tic-Tac-Toe.

Gardner, M. (1958). Mathematical games. *Scientific American*, *232*, 126.

GeeksforGeeks. (n.d.). *Machine learning*. Retrieved October 22, 2019, from https://www.geeksforgeeks.org/machine-learning/

George, W., & Janoski, J. E. (2016). Group actions on winning games of Super Tic-Tac-Toearxiv 1606.04779. Retrieved October 22, 2019, from http://arxiv.org/abs/1606.04779

HackerEarth. (2019). *Depth first search tutorials & notes*. Retrieved October 19, 2019, from https://www.hackerearth.com/practice/algorithms/graphs/depth-first-search/tutorial/

Hayes, N., & Loge, T. (2016). Developing a memory efficient algorithm for playing m, n, k games. https://www.micsymposium.org/mics2016/Papers/MICS_2016_paper_28.pdf

Hochmuth, G. (2003). On the genetic evolution of a perfect tic-tac-toe strategy. *Genetic Algorithms and Genetic Programming at Stanford*, 75–82.

Holland, J. H. (n.d.). *Genetic algorithms*. Retrieved October 22, 2019, from https://www2.econ.iastate.edu/tesfatsi/holland.gaintro.htm

Lifshitz, E., & David, T. (n.d.). AI approaches to Ultimate Tic-Tac-Toe, 5. https://cs.huji.ac.il/~ai/projects/2013/UlitmateTic-Tac-Toe/files/report.pdf

NIST/SEMATECH. (2013, October 30). *1.3.5.7. Bartlett's Test*. Retrieved May 19, 2020, from https://www.itl.nist.gov/div898/handbook/eda/section3/eda357.htm

Ortiz, A. (2017, July 10). *Machine learning with Hexapawn*. Retrieved October 19, 2019, from http://ai.aeortiz.com/2017/07/10/machine-learning-with-hexapawn/

Panchapakesan, A. (2014, March 30). *Stochastic universal sampling GA in python*. Retrieved May 12, 2020, from https://stackoverflow.com/a/22750088

Schaefer, S. (2002, January). *How many games of Tic-Tac-Toe are there?* Retrieved May 3, 2020, from http://www.mathrec.org/old/2002jan/solutions.html

Sharma, A. (2016, January 11). *Machine learning - Applications*. Retrieved October 22, 2019, from https://www.geeksforgeeks.org/machine-learning-introduction/

APPENDIX A
PROJECT PLAN

Table 2. *Task Lists and Duration*

Task	Task Description	Preceding Tasks	Duration (in days)
A	Development of Tic-Tac-Toe Game Platform and Implementation of Algorithms	—	30
B	Testing, Refinement and Optimization of Implemented Programs	A	31
C	Data Collection	B	60
D	Data Analysis	C	60

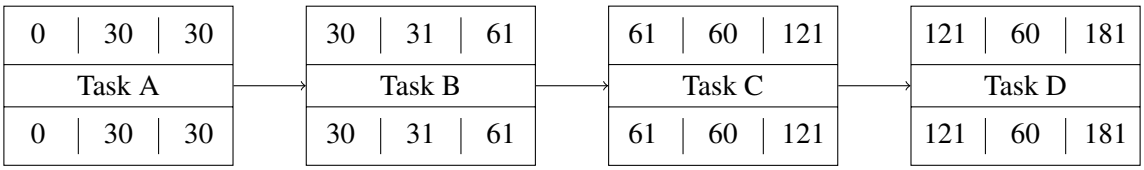

Figure 9. Network chart.

Table 3. *Task Schedule Management and Personnel Assignment Plan*

Task	Task Description	Personnel	Duration (in days)	EST	LST	ECT	LCT
A	Development of Tic-Tac-Toe Game Platform and Implementation of Algorithms	All Personnel	30	NOV 01 2019	NOV 30 2019	NOV 01 2019	NOV 30 2019
B	Testing, Refinement and Optimization of Implemented Programs	All Personnel	31	DEC 01 2019	DEC 31 2019	DEC 01 2019	DEC 31 2019
C	Data Collection	All Personnel	60	JAN 01 2019	FEB 29 2019	JAN 01 2019	FEB 29 2019
D	Data Analysis	All Personnel	61	MAR 01 2019	APR 31 2019	MAR 01 2019	APR 31 2019

Table 4. *Material and Equipment Sourcing Plan*

Protocol	Date/s Needed	Unit	Materials Needed	Potential Source	Remarks
Development of Tic-Tac-Toe Game Platform and Implementation of Algorithm	NOV-01 to 30	1	Laptop with Python	From Home	On Hand
Testing, Refinement and Optimization of Implemented Programs	DEC-01 to 31	1	Laptop with Python	From Home	On Hand
Data Collection and Analysis	JAN-01 to APR-31	1	Laptop with Python and R	From Home	On Hand

Table 5. *Risk Management Plan*

Risk	Safety Measure/Protocol
Development of Carpal Tunnel Syndrome	Frequent 5-minute breaks to relieve muscles
Electrocution	Proper usage of electronic devices
Loss of data	Upload of data into the cloud
Proprietary software trial expiry	Use of free and open-source software

APPENDIX B
RAW DATA

Table 6. *Generations per Treatment*

Number of Generations		
Control	RMG	UMG
10	6	3
3	9	9
6	6	5
9	11	9
7	6	1
7	9	7
7	8	12
10	3	5
6	10	7
7	7	10
9	6	8
11	8	14
11	7	5
11	7	11
7	9	7
4	7	7
7	11	3
3	8	8
7	7	8
9	5	5
4	7	6
8	8	6
11	7	8
7	7	5
9	9	5
6	4	6
5	6	8
8	6	10
6	7	7
12	7	9

The best fitness per generation data is in the GitHub Repository mentioned in Appendix D.

APPENDIX C
STATISTICAL TABLES

Table 7. *ANOVA Table*

Source	DF	Sum of Squares	Mean Square	F	p-value
Treatments	2	3.0	1.478	0.264	0.768
Residuals	87	486.7	5.594		

APPENDIX D
DOCUMENTATION

GitHub Repository: https://github.com/MasterToast10/paleo-str-g12

Listing 1. Base code of genetic algorithm.

```
from random import choice as random_choice, sample as
    random_sample
from linecache import getline as file_getline
from engine import MoveGenerator, Game, GameController
from game_base import GAME_ROTATIONS, find_base_case
from rmg import RandomMoveGenerator
from collections import deque
from copy import deepcopy
from functools import total_ordering


@total_ordering
class Organism(MoveGenerator):
    def __init__(self, genome: str = None):
        self.fitness = None

        if genome is None:
            temp = bytearray()
            with open("python_code/game-base.txt") as file:
                for line in file:
                    game = Game([int(x) for x in line.split
                        ()])
                    if not game.winner:
                        temp.append(random_choice(
                            tuple(i for i in range(9) if not
                                game.tiles[i])))
                    else:
                        temp.append(9)
            self.genome = "".join(str(x) for x in temp)
        else:
            if len(genome) != 765:
                raise Exception("Invalid genome")
            self.genome = genome

    def move(self, game: Game):
        base_case = find_base_case(game)
```

```
34          base_case_repr = f"{repr(base_case)}\n"
35          game_repr = f"{repr(game)}\n"
36
37          # print(base_case, base_case_repr, game_repr, sep="\
                n")
38
39          with open("python_code/game-base.txt") as file:
40              for index, line in enumerate(file):
41                  if line == base_case_repr:
42                      base_case_move = int(self.genome[index])
43                      # print(base_case_move)
44
45                      if base_case_repr == game_repr:
46                          return base_case_move
47
48                      for rotation in GAME_ROTATIONS:
49                          temp_list = []
50                          for index in rotation:
51                              temp_list.append(base_case.tiles
                                    [index])
52
53                          if f"{repr(Game(temp_list))}\n" ==
                                game_repr:
54                              # print(Game(temp_list))
55                              for index, base_case_index in
                                    enumerate(rotation):
56                                  if base_case_index ==
                                        base_case_move:
57                                      return index
58
59      def mutate(self, num_to_mutate: int):
60          positions = random_sample(range(1, 766),
                num_to_mutate)
61          for position in positions:
62              game = Game([int(x) for x in file_getline(
63                  "python_code/game-base.txt", position).split
                        ()])
64              if not game.winner:
65                  self.genome = f"{self.genome[:position-1]}{
                        random_choice(tuple(i␣for␣i␣in␣range(9)␣
                        if␣not␣game.tiles[i]))}{self.genome[
                        position:]}"
66              else:
67                  self.mutate(1)
68          self.fitness = None
```

```
69          self.get_fitness()
70
71      def get_fitness(self):
72          if self.fitness is not None:
73              return self.fitness
74          else:
75              self.win_x = 0
76              self.draw_x = 0
77              self.loss_x = 0
78
79              empty_game = deepcopy(Game())
80              empty_game.set_tile(self.move(empty_game))
81              anal_queue = deque([empty_game])
82
83              while len(anal_queue):
84                  to_analyze = anal_queue.popleft()
85
86                  if to_analyze.winner:
87                      if to_analyze.winner == 1:
88                          self.win_x += 1
89                      elif to_analyze.winner == 2:
90                          self.loss_x += 1
91                      else:
92                          self.draw_x += 1
93                  else:
94                      for empty_tile in (i for i in range(9)
                            if not to_analyze.tiles[i]):
95                          to_analyze_clone = deepcopy(
                                to_analyze)
96                          to_analyze_clone.set_tile(empty_tile
                                )
97                          if not to_analyze_clone.winner:
98                              to_analyze_clone.set_tile(
99                                  self.move(to_analyze_clone))
100                         anal_queue.append(to_analyze_clone)
101
102             self.fitness = (self.loss_x) / (self.win_x +
103                                             self.draw_x +
                                                self.loss_x)
104
105             return self.fitness
106
107     def __eq__(self, other):
108         return self.get_fitness() == other.get_fitness()
109
```

```
110     def __lt__(self, other):
111         return self.get_fitness() < other.get_fitness()
112 
113     def __repr__(self):
114         return f"Organism <{self.genome}>"
115 
116 
117 def crossover(parent1: Organism, parent2: Organism):
118     cross_sites = sorted(random_sample(range(1, 766), 50))
119     pre_cross_sites = deque(cross_sites)
120     pre_cross_sites.appendleft(None)
121     post_cross_sites = deque(cross_sites)
122     post_cross_sites.append(None)
123     parent1_genome = parent1.genome
124     parent2_genome = parent2.genome
125     offspring1_genome = []
126     offspring2_genome = []
127     alternator = True
128     for pre, post in zip(pre_cross_sites, post_cross_sites):
129         if alternator:
130             offspring1_genome.extend(parent1_genome[pre:post
                ])
131             offspring2_genome.extend(parent2_genome[pre:post
                ])
132             alternator = False
133         else:
134             offspring1_genome.extend(parent2_genome[pre:post
                ])
135             offspring2_genome.extend(parent1_genome[pre:post
                ])
136             alternator = True
137     return Organism("".join(offspring1_genome)), Organism("
        .join(offspring2_genome))
138 
139 
140 def population_after_mating(mating_pool):
141     rotated = deque(mating_pool)
142     rotated.rotate(-1)
143     ret_list = []
144     for parent_pair in zip(mating_pool, rotated):
145         ret_list.extend(crossover(*parent_pair))
146     return ret_list
147 
148 
149 def population_after_mutation(population_to_mutate):
```

```
150     probability_of_mutation = min(o.get_fitness()
151                                   for o in
                                          population_to_mutate)
152     num_to_mutate = int(250*probability_of_mutation + 10)
153     for organism in population_to_mutate:
154         organism.mutate(num_to_mutate)
155     return population_to_mutate
156
157
158 if __name__ == "__main__":
159     # for i in range(10000):
160     #     organ = Organism()
161     #     print(organ)
162     #     organ.mutate(5)
163     #     print(organ)
164     #     boi = GameController(RandomMoveGenerator(1), organ
            , verbose=True)
165     #     boi.start()
166     # organ = Organism()
167     # for i in range(50):
168     #     print(organ.get_fitness())
169     parents = [Organism(), Organism()]
170     for org in parents:
171         print(org.genome)
172         print(org.get_fitness())
173     # print()
174     # for org in population_after_mating(parents):
175     #     print(org.genome)
176     for mutated in population_after_mutation(parents):
177         print(mutated.genome)
178         print(mutated.get_fitness())
```