Comparison of Genetic Algorithm Training Methods As Applied To Tic-Tac-Toe

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.

# 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 [@sharmaMachineLearningApplications2016]. 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 grid and two players alternate turns trying to earn pieces adjacent to each other horizontally, vertically or diagonally [@hayesDevelopingMemoryEfficient2016]. 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 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* [@geeksforgeeksMachineLearning]. 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 [@bhattSearchNolossStrategies2008]. 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.

# 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 [@aycockHowWinTicTacToe2002].

These examples of strategies were translated programmatically by [@barratRobbiebarratUnbeatableTictactoe2019]. 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. [@barratRobbiebarratUnbeatableTictactoe2019] 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 discussions gave occurred on how many distinct games one can play in Tic-Tac-Toe. [@schaeferHowManyGames2002] 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 [@schaeferHowManyGames2002], it is of more concern to count game states that end when there are 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 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 [@adamchikGameTrees2009]. A diverse game tree is optimal for exploring the capabilities of the different machine learning methods to be used in the study.

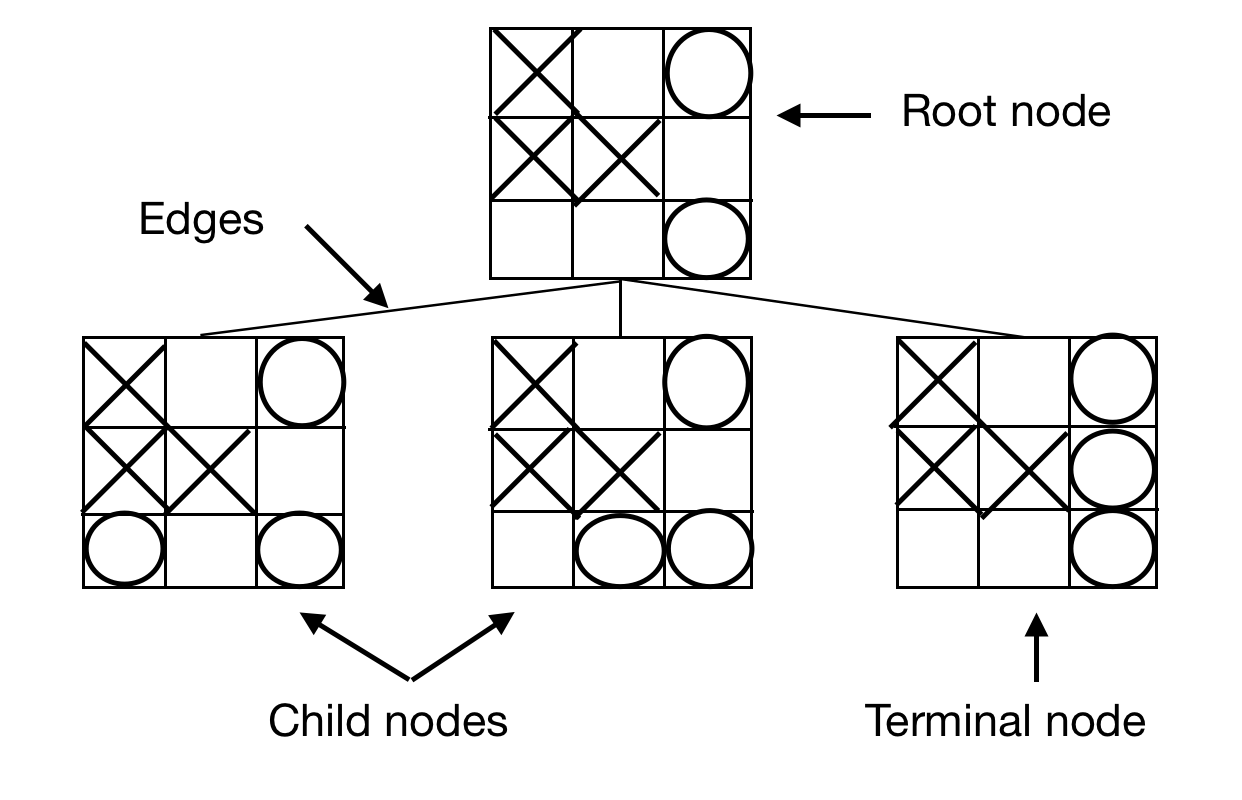


Figure 1: Game tree

A study by [@cranenburghTicTacToe2007] 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* [@hackerearthDepthFirstSearch2019]. According to [@cranenburghTicTacToe2007], 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 [@georgeGroupActionsWinning2016] specifies the rotational and reflectional symmetry of Ultimate/Super Tic-Tac-Toe. A study by [@lifshitzAIApproachesUltimate] 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 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 [@ortizMachineLearningHexapawn2017]. 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 [@gardnerMathematicalGames1958].

## 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.

![Figure 2: Flowchart of GA training](data:application/pdf;base64,)

Figure 2: Flowchart of GA training

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 [@hochmuthGeneticEvolutionPerfect2003]. Fitness scores are usually calculated based on the ratio of games won, as shown in Equation [[eqn:fitness]](#eqn:fitness). 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.

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 [@hollandGeneticAlgorithms].

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 [2.1](#tab:ex_classifier), 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.

Example of Organisms Sorted Through a Classifier System

|  |  |  |  |
| --- | --- | --- | --- |
| 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 |

# 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 . 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 [9](#apx:documentation). The developmental computer is equipped with 4 GB of Random Access Memory (RAM). Various personal computers were used to perform the training.

## Procedures

![Figure 3: Procedural Flowchart](data:application/pdf;base64,)

Figure 3: Procedural Flowchart

### 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 [@bhattSearchNolossStrategies2008], 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 [@barratRobbiebarratUnbeatableTictactoe2019]. 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 [2.2](#fig:genetic), the operators are adapted from [@bhattSearchNolossStrategies2008]. The only aspect of the GA that differs from the implementation of that in [@bhattSearchNolossStrategies2008] is the organism representation.

###### Organism Representation

Organisms are represented with genomes, as shown in Figure [3.2](#fig:genome). Each gene in the genome represents a game state that considers rotation and symmetry using an implementation of the game-base used by [@bhattSearchNolossStrategies2008]. These genes have a corresponding allele that represents the move the organism will take.

![Figure 4: Genome of an organism](data:application/pdf;base64,)

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 [@bhattSearchNolossStrategies2008] was used to modify the implementation of the Stochastic Universal Selection operator by [@panchapakesanStochasticUniversalSampling2014]:

where is the number of organisms with the same fitness. Using as a factor produces a niching effect [@bhattSearchNolossStrategies2008]. The quantity produces the opposite pattern compared to Equation [[eqn:fitness]](#eqn:fitness), 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 [@bhattSearchNolossStrategies2008]. In this study, half of the population is selected via SUS.

###### Crossover Operator

Similar to that of [@bhattSearchNolossStrategies2008], 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, genes are mutated based on the best fitness in the population. This basis is formalized by [@bhattSearchNolossStrategies2008] in the following equation:

###### 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 are chosen:

###### 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 [[eqn:fitness]](#eqn:fitness) 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 [[eqn:fitness]](#eqn:fitness)) or no-loss solution is recorded for each simulation.

## Treatment of Data

### Statement of Hypotheses

### Analysis of Data

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

# RESULTS AND DISCUSSION

![Figure 5: Notched box plot of treatments](data:application/pdf;base64,)

Figure 5: Notched box plot of treatments

A notched box plot of treatment results is shown in Figure [4.1](#fig:boxplot). 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](data:application/pdf;base64,)

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 [4.4](#fig:fitgraph_umg). 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 [4.2](#fig:fitgraph_ctrl) and Figure [4.3](#fig:fitgraph_rmg). Figure [4.4](#fig:fitgraph_umg) 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 [@nist/sematechBartlettTest2013]. This test resulted in with a -value of 0.07801. Said -value is greater than , 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 -value of 0.768. The -value is greater than , therefore the null hypothesis that the means of the treatments are equal is not rejected.

![Figure 7: Population-best fitness graph of RMG-controlled](data:application/pdf;base64,)

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

![Figure 8: Population-best fitness graph of UMG-controlled](data:application/pdf;base64,)

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

It can be noted in Figures [4.2](#fig:fitgraph_ctrl), [4.3](#fig:fitgraph_rmg), and [4.4](#fig:fitgraph_umg) that the GAs stop on or before 15 generations. This is because the GAs can find no-loss solutions before even surpassing the 500 generation mark. The 10-generation worst case tabulated by [@bhattSearchNolossStrategies2008] agrees with these findings. The results for each GA are comparatively lower than the 500-generation upper bound used by [@bhattSearchNolossStrategies2008]. This can be explained by the fact that the first player is more likely to win in Tic-Tac-Toe [@cranenburghTicTacToe2007], 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 [4.4](#fig:fitgraph_umg). 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.

# 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.

# Project Plan

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 |

# Raw Data

Generations per Treatment

|  |  |  |
| --- | --- | --- |
|  |  |  |
| 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 [9](#apx:documentation).

# Statistical Tables

ANOVA Table

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

# Documentation

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

Listing lst:ga.py: 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)  
 base\_case\_repr = f"{repr(base\_case)}\n"  
 game\_repr = f"{repr(game)}\n"  
  
 # print(base\_case, base\_case\_repr, game\_repr, sep="\n")  
  
 with open("python\_code/game-base.txt") as file:  
 for index, line in enumerate(file):  
 if line == base\_case\_repr:  
 base\_case\_move = int(self.genome[index])  
 # print(base\_case\_move)  
  
 if base\_case\_repr == game\_repr:  
 return base\_case\_move  
  
 for rotation in GAME\_ROTATIONS:  
 temp\_list = []  
 for index in rotation:  
 temp\_list.append(base\_case.tiles[index])  
  
 if f"{repr(Game(temp\_list))}\n" == game\_repr:  
 # print(Game(temp\_list))  
 for index, base\_case\_index in enumerate(rotation):  
 if base\_case\_index == base\_case\_move:  
 return index  
  
 def mutate(self, num\_to\_mutate: int):  
 positions = random\_sample(range(1, 766), num\_to\_mutate)  
 for position in positions:  
 game = Game([int(x) for x in file\_getline(  
 "python\_code/game-base.txt", position).split()])  
 if not game.winner:  
 self.genome = f"{self.genome[:position-1]}{random\_choice(tuple(i for i in range(9) if not game.tiles[i]))}{self.genome[position:]}"  
 else:  
 self.mutate(1)  
 self.fitness = None  
 self.get\_fitness()  
  
 def get\_fitness(self):  
 if self.fitness is not None:  
 return self.fitness  
 else:  
 self.win\_x = 0  
 self.draw\_x = 0  
 self.loss\_x = 0  
  
 empty\_game = deepcopy(Game())  
 empty\_game.set\_tile(self.move(empty\_game))  
 anal\_queue = deque([empty\_game])  
  
 while len(anal\_queue):  
 to\_analyze = anal\_queue.popleft()  
  
 if to\_analyze.winner:  
 if to\_analyze.winner == 1:  
 self.win\_x += 1  
 elif to\_analyze.winner == 2:  
 self.loss\_x += 1  
 else:  
 self.draw\_x += 1  
 else:  
 for empty\_tile in (i for i in range(9) if not to\_analyze.tiles[i]):  
 to\_analyze\_clone = deepcopy(to\_analyze)  
 to\_analyze\_clone.set\_tile(empty\_tile)  
 if not to\_analyze\_clone.winner:  
 to\_analyze\_clone.set\_tile(  
 self.move(to\_analyze\_clone))  
 anal\_queue.append(to\_analyze\_clone)  
  
 self.fitness = (self.loss\_x) / (self.win\_x +  
 self.draw\_x + self.loss\_x)  
  
 return self.fitness  
  
 def \_\_eq\_\_(self, other):  
 return self.get\_fitness() == other.get\_fitness()  
  
 def \_\_lt\_\_(self, other):  
 return self.get\_fitness() < other.get\_fitness()  
  
 def \_\_repr\_\_(self):  
 return f"Organism <{self.genome}>"  
  
  
def crossover(parent1: Organism, parent2: Organism):  
 cross\_sites = sorted(random\_sample(range(1, 766), 50))  
 pre\_cross\_sites = deque(cross\_sites)  
 pre\_cross\_sites.appendleft(None)  
 post\_cross\_sites = deque(cross\_sites)  
 post\_cross\_sites.append(None)  
 parent1\_genome = parent1.genome  
 parent2\_genome = parent2.genome  
 offspring1\_genome = []  
 offspring2\_genome = []  
 alternator = True  
 for pre, post in zip(pre\_cross\_sites, post\_cross\_sites):  
 if alternator:  
 offspring1\_genome.extend(parent1\_genome[pre:post])  
 offspring2\_genome.extend(parent2\_genome[pre:post])  
 alternator = False  
 else:  
 offspring1\_genome.extend(parent2\_genome[pre:post])  
 offspring2\_genome.extend(parent1\_genome[pre:post])  
 alternator = True  
 return Organism("".join(offspring1\_genome)), Organism("".join(offspring2\_genome))  
  
  
def population\_after\_mating(mating\_pool):  
 rotated = deque(mating\_pool)  
 rotated.rotate(-1)  
 ret\_list = []  
 for parent\_pair in zip(mating\_pool, rotated):  
 ret\_list.extend(crossover(\*parent\_pair))  
 return ret\_list  
  
  
def population\_after\_mutation(population\_to\_mutate):  
 probability\_of\_mutation = min(o.get\_fitness()  
 for o in population\_to\_mutate)  
 num\_to\_mutate = int(250\*probability\_of\_mutation + 10)  
 for organism in population\_to\_mutate:  
 organism.mutate(num\_to\_mutate)  
 return population\_to\_mutate  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 # for i in range(10000):  
 # organ = Organism()  
 # print(organ)  
 # organ.mutate(5)  
 # print(organ)  
 # boi = GameController(RandomMoveGenerator(1), organ, verbose=True)  
 # boi.start()  
 # organ = Organism()  
 # for i in range(50):  
 # print(organ.get\_fitness())  
 parents = [Organism(), Organism()]  
 for org in parents:  
 print(org.genome)  
 print(org.get\_fitness())  
 # print()  
 # for org in population\_after\_mating(parents):  
 # print(org.genome)  
 for mutated in population\_after\_mutation(parents):  
 print(mutated.genome)  
 print(mutated.get\_fitness())