60-371

Artificial Intelligence - Group Project

The Prisoner's Dilemma

Submitted to Professor Robin Gras
on March 20th, 2017

Stefanie Bodnar
Evian Kassab
Zach Shaver
Tyler Smith

Table of Contents

Abstract

This report evaluates two search algorithms, **genetic** and **hill-climbing**, that attempt to produce optimal strategies to compete within the “Iterated Prisoner’s Dilemma”. The research extracts findings from each in order to meet the objective of what algorithm would produce the best prisoner. The use of 1D and 2D lookups are a fundamental aspect of the research as the results varied between them. Both algorithms performed quite well in creating good strategies, and usually quite quickly. They were both able to produce good lookup strategies specifically against individual opponents and generally against a variety of opponents. Throughout the report, key findings will compare and contrast the two algorithms and their optimal search strategies.

1. Introduction to Prisoner's Dilemma

The Prisoner's Dilemma is an example of a game, invented by Merril Flood and Melvin Dresher in the 1950s. This dilemma was applied in a vast variety of fields, such as economics, political science and game theory. It can be used as a simulation of real world models that involve cooperative and game-like behaviour.

The game consists of two prisoners that are faced with a decision that could potentially reduce or increase their individual sentence based on the other prisoner's action. For example, two entities could benefit from cooperating or suffer from failing to do so. This game can be broken down into three components:

1) *Player*: Two criminals, have been caught and they are unable to communicate with each other.
2) *Actions*: These two criminals can both confess, and they will each get 10 years. If neither one were to confess, they will both get 1 year on lesser charge. However, if one confesses and accuses the other, then the confessor would go free and the accused would get 20 years.
3) *Payoff Function*: Gives the utility to each player for each combination of actions by all the players. In this game, the representation known as the strategic form and can be broken down in a matrix:

	Opponent **Cooperate**	Opponent **Defect**
Player **Cooperate**	-1, -1	-20, 0
Player **Defect**	0, -20	-10, -10

The motivation of each player is to achieve the best possible outcome which is the shortest jail sentence based on the constraints and viable outcome. From A's point of view, being a rational agent, they will try to receive no jail time by choosing to defect. However, if B also chooses to defect, then they both receive a sentence of 10 years. This is not what A (or B for that matter) wanted, and it would have been better for both of them to cooperate and only receive 1 year each. Herein lies the dilemma; choosing which move to make to minimizes the penalty or maximizes our reward. This report details two such algorithms for producing good strategies.

2. Strategies and Implementation

The research focused on using search algorithms to develop optimal prisoners. Strategies will be designed to make use of lookup tables with different dimensions and comparing their average scores. Although a Java library called IPDX will provide the basic programming interface for playing games, some requirements will have to be designed from scratch. The purpose of this section is to share technical design and programming details used to implement the interfaces that are not provided by the IPDX library, and ultimately optimize the algorithm's' effectiveness and speed.

History of Moves

To optimize quick lookup, historical moves are stored as a bit string in integer format. This is achieved by converting a string of moves (e.g. “DDDCD) to the appropriate binary (e.g. 00010), and storing as an integer (e.g. 2). This integer is used as the index for the associated history in the lookup table. Cooperation (C) is equal to a 1 bit and a defect (D) is equal to a 0 bit. The most recent move in the history is the least significant bit. In this fashion, up to 32 moves can be stored in memory and manipulated in an efficient manner, such as adding moves by bit shifting, or setting/flipping moves with AND/XOR operations.

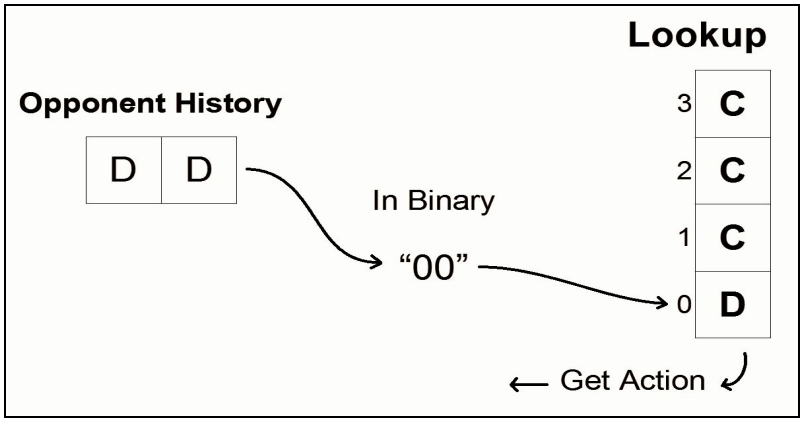
Array Lookups

Playing the game, the hill-climbing and genetic algorithms can utilize either a 1D (opponent history only) or a 2D (opponent and player history) array lookup to get their next move. For the 1D lookup, each action for a particular history is assigned a move to play next. The table lookup for Tit-For-Two-Tat (TFTT), which has a history of 2 moves, would look like:

Index	**0**	**1**	**2**	**3**
Value (Action)	D	C	C	C

The histories for each look up are the index values: **0** = DD, **1** = DC, **2** = CD, & **3** = CC.

Here, when the opponent's history is “DD” (0), the lookup returns a “D” and the strategy plays a defect, and for the rest (“DC”, “CD”, and “CC) it plays cooperation:



The size of array table increases exponentially as the history size increases by a power of two for every move. Because of this, it uses a lot of memory for large histories, but the advantage is that looking up actions is done in constant time - the array access time.

The 2D lookup just adds a second dimension to the array, which accounts for the player's history in addition to the opponent's. It works in the same way, but can take up much more memory.

Evaluation Function

To determine whether one array table is better than another, an evaluation function is required. This is used to find the better neighbors in the hill climbing algorithm, and to determine the fitness of the individuals within the genetic algorithm.

The lookup evaluation is implemented by playing 50 games, at 25 rounds a piece, against each individual strategy in the training set. The computed average score after playing against everyone in the training set is returned as the score of the lookup table.

A two (2) decimal place accuracy of the score is used as it presents the most consistent and usable value. Having less precision makes it hard to determine which lookup tables are better, and having more precision makes similar tables return a broader range of scores (lessening the effectiveness of their use as an evaluation).

Iterated Prisoner’s Dilemma

The implementation of the Iterated Prisoner’s Dilemma is based on each player having memory of the previous rounds played. The length of the memory allows the player to formulate their strategies in different ways. These strategies are deterministic which means players will always make the same move for a given history.

Genetic and hill climbing algorithms are the two algorithms that are being utilized and tested for the Iterated Prisoner's Dilemma. The goal is to get the best lookup table possible, using the previously mentioned evaluation function.

The training set for the evaluation function contains all opponents to optimize a lookup table against. In this report, the training set named "Variety" contains ALLC, ALLD, NEG, Pavlov, STFT, TFT and TFTT. It is possible to design a good lookup table that can be used to compete against a wide variety of opponents, as shown in the report. However, primary testing is also based on one training set comprised of only TFTT to ensure the most consistent results when judging an algorithm's effectiveness in establishing a good strategy.

See Appendix A for further information on the various strategies used.

The results collected throughout the research are using the following payoff matrix:

	Opponent **Cooperate**	Opponent **Defect**
Player **Cooperate**	3,3	0,5
Player **Defect**	5,0	1,1

Data Collection

Data collection is a vital step in the process for answering the objective of the research. By running the algorithms with various parameters and storing the data into a CSV file data collection was achieved. Upon obtaining the data, multiple graphs were rendered which are referenced by this report.

Unless otherwise noted, parameters in data collection were fixed to values that showed to be successful:

- Data collected for 1D lookups is based on opponent history only with a length of 5 and 2D lookups set both the player history and opponent history length to a length of three. Report is predominantly based on 2D lookups as they tend to present better results statistically (longer hill climbs, more diverse populations, etc.).

- For hill climbing, random restarts are set to ten with sideways moves also set to ten. Although, given the 2 decimal accuracy approach, it typically never makes sideways

moves.

- Implementation of the genetic algorithm is based on a population of size 100, generations of 150, and a mutation rate of 0.075 (7.5%).

3. Hill-Climbing

Introduction

The hill-climbing algorithm is a local search that keeps track of a single current state, and moves only to a neighbouring state if it is better than itself. It can be called a greedy local search because it grabs only the best neighbour state from its local neighbourhood. It can simply be put as a loop that continually moves in the direction of increasing value; uphill. Hill-Climbing terminates when no neighbour has a higher value which means it has reached a peak.

Hill-Climbing in the Prisoner's Dilemma

An illustration of the algorithm is out lined in the following pseudo code:

```
Input: Max Restarts, Max Sideways
Output : Lookup that is a Local Maximum

Lookup = new Random Lookup.

Repeat: {
        Neighbour = getBestNeighbour().
        if ( Neighbour's Score > Lookup's Score)
                Store Neighbour as Lookup
        else if ( Neighbour's Score < Lookup's Score OR Out of Restarts)
                return Lookup
        else
                if ( Sideways Moves Available )
                        Do a Sideways Move
                else
                        Reset Sideways Moves
                        Do a Restart
}
```

First, a random lookup is created. Then the algorithm enters a loop, searching its neighbourhood for better tables, replacing itself with them, and repeating.

getBestNeighbour() is the operator function for the hill climbing. In the prisoner dilemma problem, it takes the current lookup, and returns the best neighbour. It does this by flipping each action in the lookup table, getting the score, flipping the action back - keeping the table

with the largest score. This allows exploration of only the immediate neighbourhood. An example neighbourhood for a 1D lookup table with history size 2 is shown below:

Original Table:	CCCC (Score : 2.51)	
Neighbour 1:	CCC**D** (Score : 2.48)	
Neighbour 2:	CC**D**C (Score : 2.33)	
Neighbour 3:	C**D**CC (Score : 2.65)	<= Best Neighbour
Neighbour 4:	**D**CCC (Score : 2.56)	

In the event of a locating a better neighbour, the lookup is changed to this neighbour's state. Our version stores the lookup to disk if it is greater than the best lookup found, even after restarts.

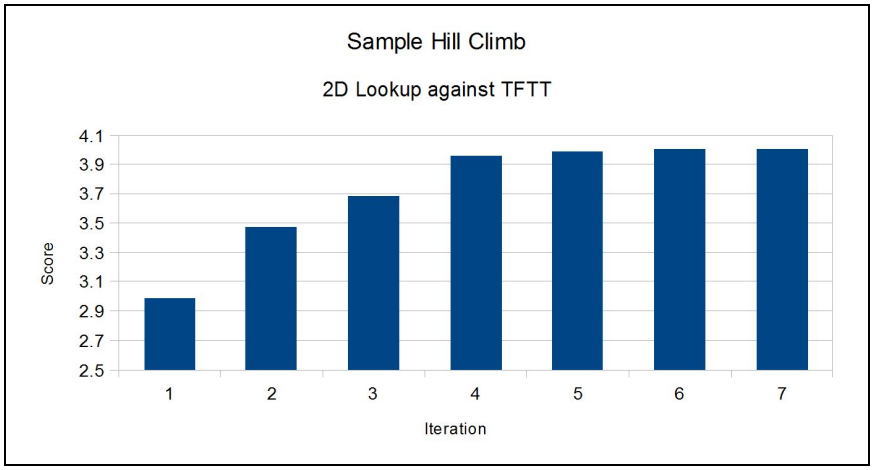
If the best neighbour has a worse score than the current state, or it happens to run out of restarts, it will load the best lookup from disk and return it as the solution.

If the best neighbour's score is equal to the current state, a sideways move is permitted. If there are no more available sideways moves, then a restart will randomize the lookup table, effectively placing the table in a completely different area of the search space, and the loop will continue.

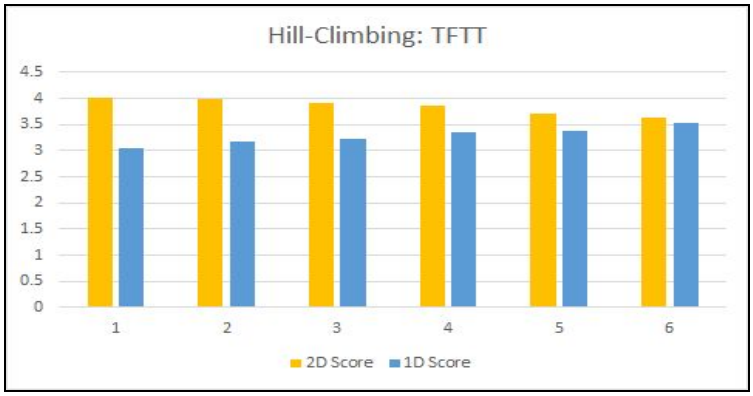
Findings

The data collection demonstrates that the 2D lookups are more consistent in reaching the optimal lookup. When trained against TFTT, it typically scores a 4.0, which is the highest possible when competing with it. The 1D lookup tends towards the local max being achieved relatively quickly, whether or not it's optimal, so it possibly return a subpar table. The 2D lookup version was more consistent in finding the most optimal lookup, with longer hill climbs. A possible reason for this outcome is that having two histories provides more diverse neighbours. There is bound to be at least one that has a better score, if it exists in the search space.

The above graph is based on the resultant lookups of 50 different 2D Lookup Hill Climbs, with no restarts, when training the evaluation function against all variety of strategies. It is evident that due to the large search space, the hill climb will hit many local maxes that are not optimal tables. Hill-Climbing is limited to its local search as the space to explore. This results in only reaching a certain potential. The solution to this problem, is to allow sideways moves and restarts, which is implemented by our algorithm. This allows the algorithm to continue after hitting the local max, and it often finds the optimal table. An sample of a single hill run is shown below, hitting the local max and moving sideways 3 times:



The graph below shows the average scores when trained against TFTT at different history lengths (memory depths). Smaller memory depths tend to do better when trained against TFTT, most likely do to TFTT's memory depth being only 2 moves. Interestingly, when trained against a variety of strategies the 2D does better with smaller histories and the 1D does better with larger histories:



4. Genetic Algorithm

Introduction

The genetic algorithm attempts to solve a family of constrained and unconstrained optimization problems, and it works very well with the Prisoner's Dilemma problem. It tries to emulate nature's way of optimizing creatures and creating spinoffs that are better and overtake them. Starting with a random population, successive populations are generated by combining parent states rather than modifying a single state. Each state is rated by an objective function called the fitness function, which is essentially the evaluation function detailed earlier. These fitness functions are used in order to rank the initial population and allow fitter individuals to mate more often. We accomplish this through a biased selection process, followed by a crossover which is the reproduction of the two pairs mating. These offspring are subject to mutation with a small probability.

Genetic Algorithm in Prisoner's Dilemma

Below is the algorithm in pseudocode:

Input: Population Size, Number of Generations, Number of Children per Couple
Output : Fittest Individual after Evolution

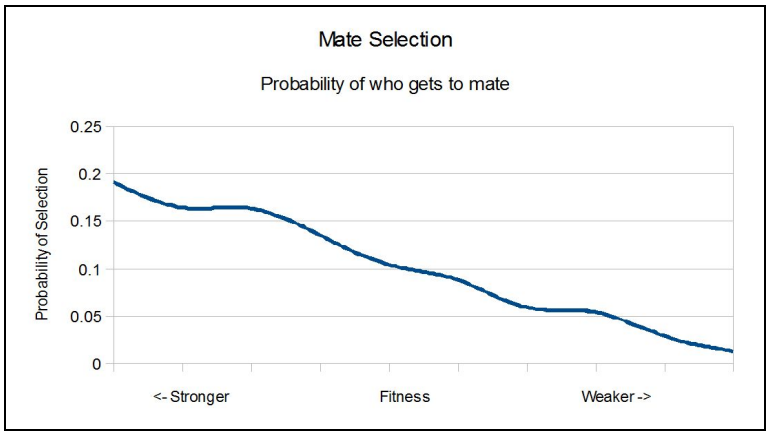
Population = *new* ***Population of Random Lookups***

For: (E*ach Generation*) {
    NextGeneration = Empty Population.
    For: (**Population size**) {
        Father = *selectRandomIndividual().*
        Mother = *selectRandomIndividual().*
        Child = *reproduce*(**Mother, Father***).*
        Mutate **Child** with a small probability.

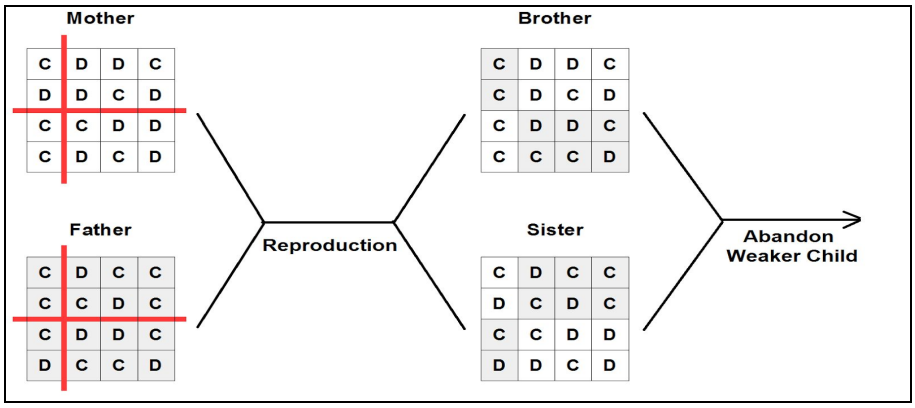
```
                    Add the Child to NextGeneration.
            }
            Population = NextGeneration
}
Return the Fittest Individual in Population
```

A population is simply an array of lookup tables. All are randomized initially.

selectRandomIndividual() selects a random individual from the population for mating. The randomness is skewed towards selecting the fittest individuals. This is accomplished by sorting the individuals by fitness and applying a skewed random distribution to it, as shown below:



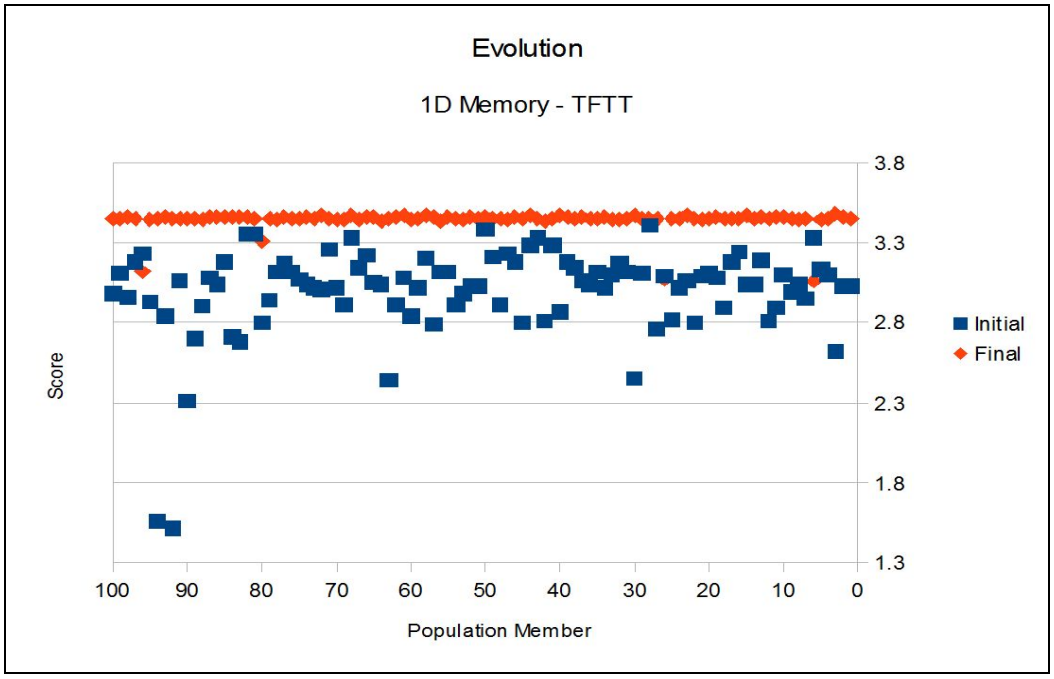
By generating a new population from the current one according to this distribution, the algorithm will hopefully evolve the population to a better lookup table over time. The two members selected for reproduction are viewed as the parents. *reproduce(***Mother, Father***)* takes the **Mother** and **Father**'s lookup tables, splits them at a random index close to the middle (for a 2D, it does this for rows and columns), and exchanges their parts to form 2 children as shown below for the 2D lookup version. The children are scored, and the child lookup with the best score is added to the next generation.

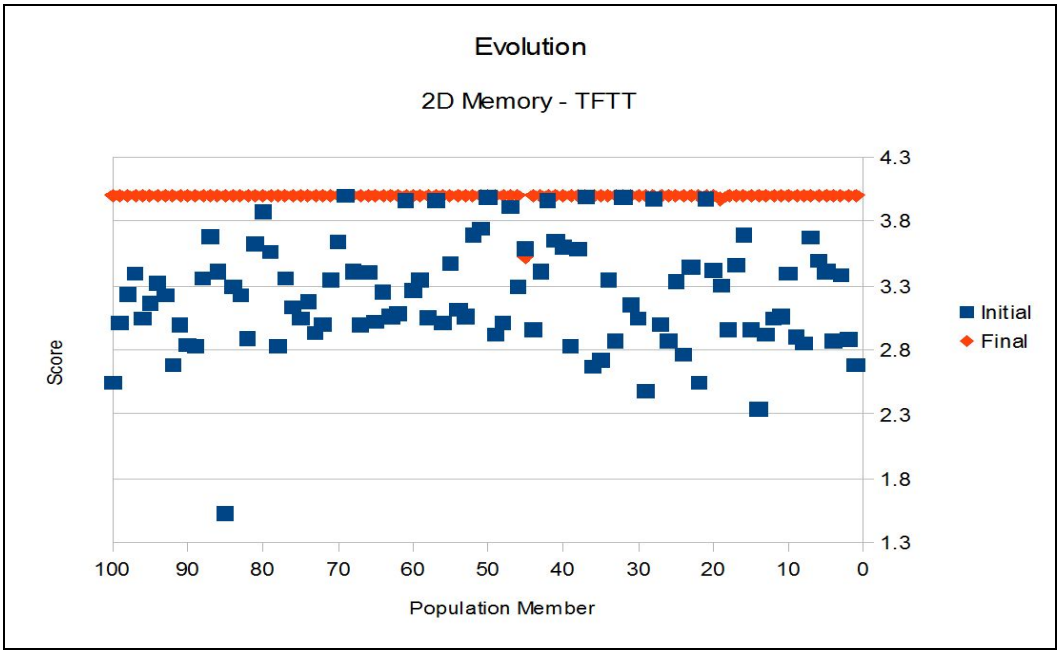


Our version of the genetic algorithm includes 2 modes: a) 1 child per couple and b) 2 children per couple. The only difference between the two is upon reproducing; only the best child is added to the next generation for the 1 child per couple mode. This proved to be the most successful version and was used in testing.

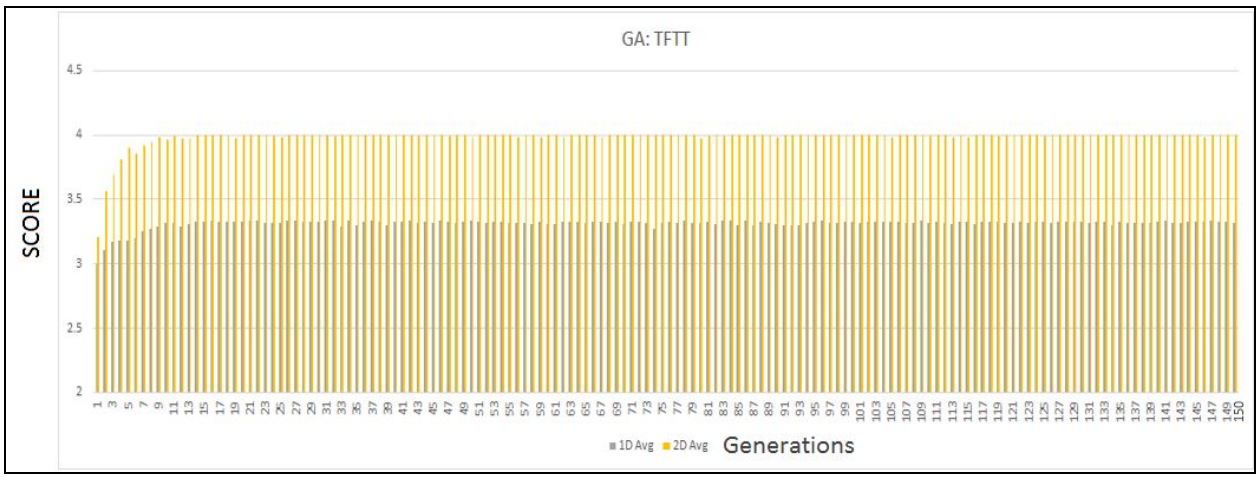
Findings

An interesting observation is that the ending population is dependent on the strategies that the they are being trained against - the training set. Given enough generations, the entire population will converge on whatever the optimal strategy is for countering the training set. When facing a single opponent, such as TFTT, the population quickly converges on the optimal solution for a 2D lookup. Usually there is an individual in the initial population who has the solution, so the rest of the population just has to catch up with it. When faced with a variety of opponents, there is more room for improvement and so the ending fittest individual is often better than the starting fittest individual. This is due either to the breeding itself making better tables, and/or mutated children that have better lookups and then become the dominant individuals themselves. The genetic algorithm performs very well, and even the worst tables in a population perform well after evolution. The graphs below shows how the initial population and final populations differ after evolution, when training against TFTT (See Appendix B for same graphs but with a variety of opponents):



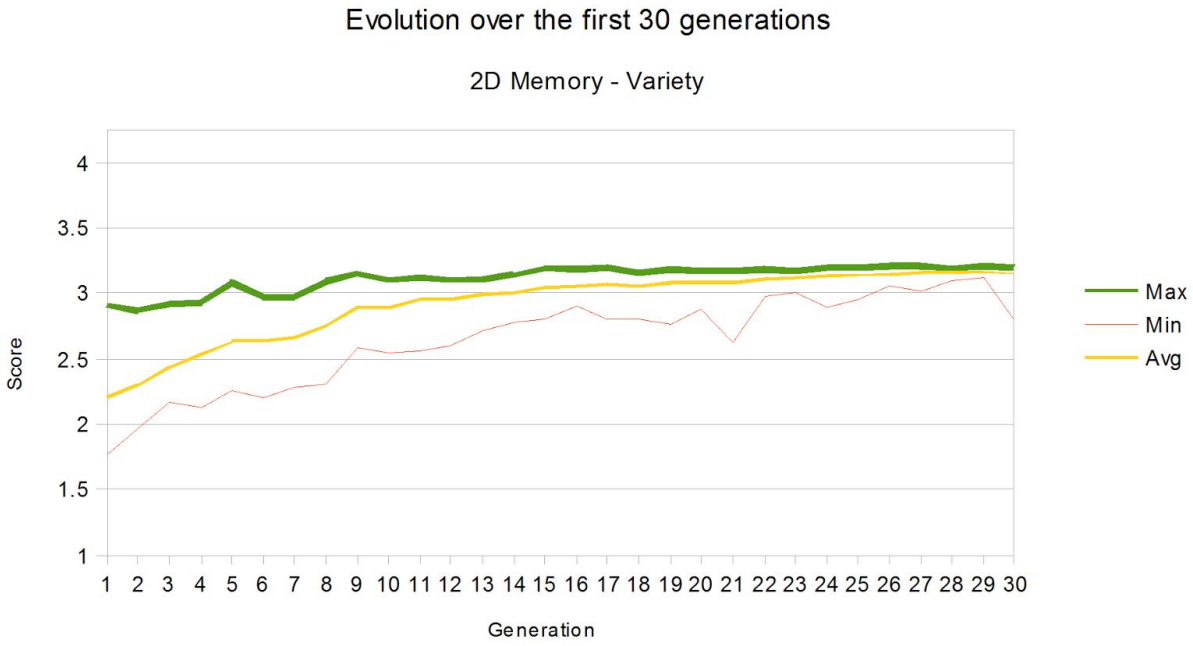


The graph below depicts evolution for 150 generations, when trained against TFTT. Once again the 2D remains the best lookup table to proceed with as the scores clearly are much higher. It can also be seen that the peak fitness of the population comes within the first 10 generations and then plateaus for the next 140. This shows that GA produces a fit average population very quickly. Thus the number of generations doesn't really affect the end result past about 10 generations. The fittest individual pulled after 10 generations is likely to be the same as after 150 when playing against a single strategy, due to them all converging on the best table. However, with a variety of strategies, the average population will get stronger over time most likely due to the diversity of solutions (see Appendix C).

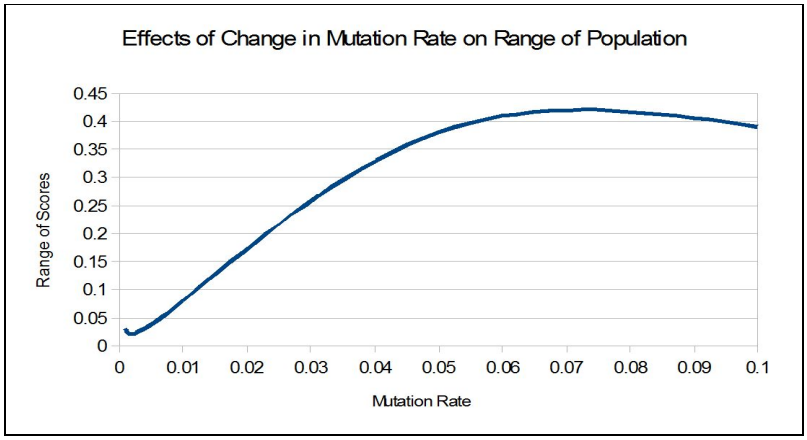


Below is a graph showing the population scores when their lookups are trained against a

variety of strategies, over the first 30 generations. See Appendix C for similar charts with longer generations. Compared are the minimum, maximum, and average scores of the population achieved. The fittest individuals get a little stronger, while the weakest and average individuals tend to catch up, resulting in a much stronger overall population. The spread between min and max also decreases, further corroborating with the theory that the entire population gets stronger as a whole. This could be viewed as speciation, as the entire population takes on a similar look, especially when trained against a singular strategy since there is 1 optimal strategy. Hypothetically, when trained against multiple strategies, if another population with better lookups was introduced to mix with them then it could diversify the lookups and make them even better.



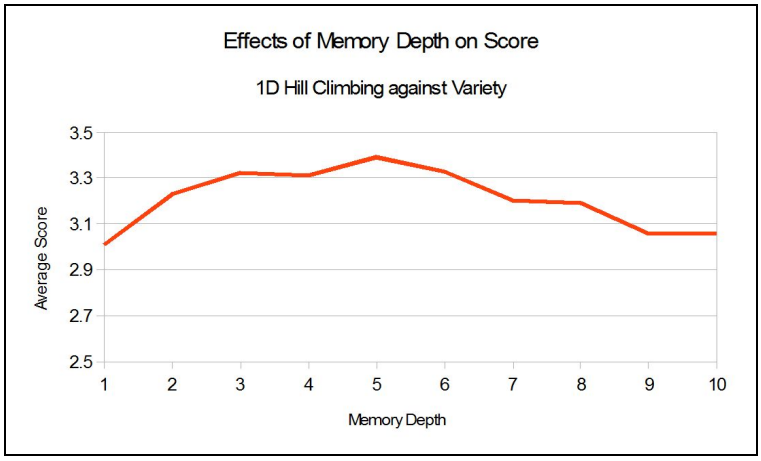
Changing the population size and number of generation do not have much effect on the resultant fittest individual when training against TFTT, as the best lookups in the initial population aren't that far off from optimal. However, change in the mutation rate creates a more diverse population, so the range of scores is greater in turn, as shown below.

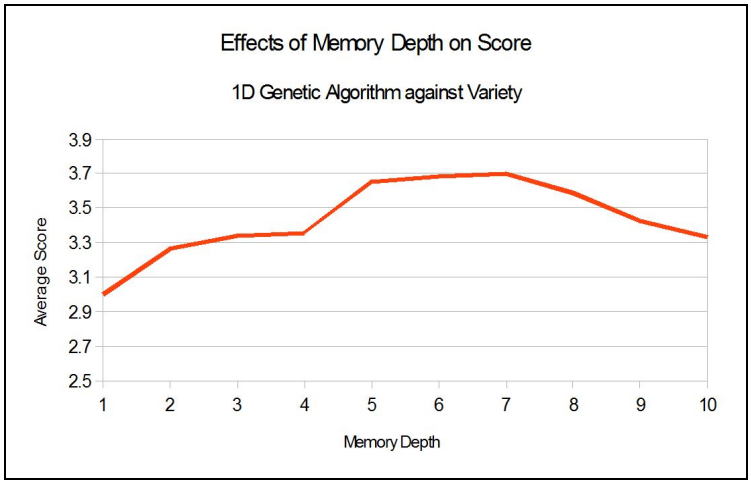


5. Conclusion

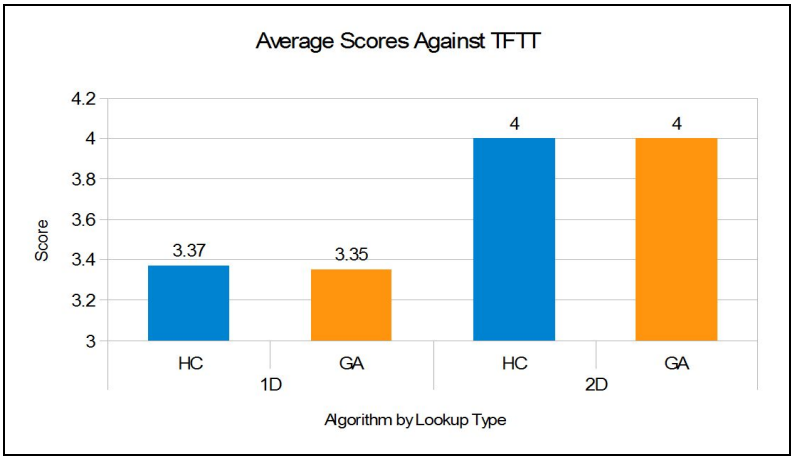
Compare and Contrast

Both algorithms seemed to prefer similar history lengths with around 5-6 moves for a 1D lookup. The graphs below show their average scores compared to their memory depth, for a 1D lookup and a variety of opponents in the training set:





Both algorithms performed exceptionally better when using 2D lookups as opposed to 1D lookups. The graph below shows the average scores for both algorithms when trained against TFTT, using both lookup versions:



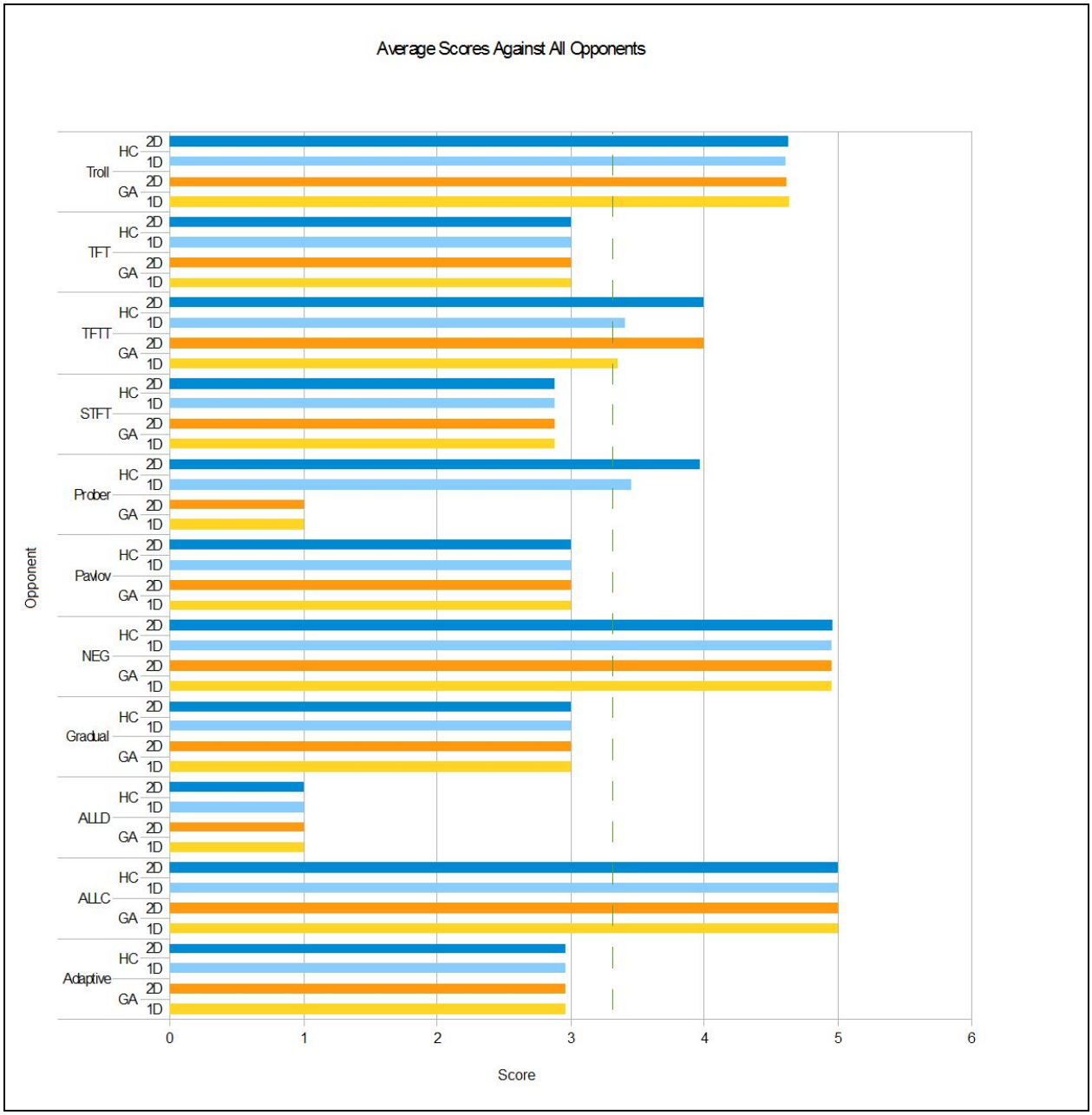
The approach of the two algorithms is unique in their own way specific to both the data collection and findings. The findings show that the genetic algorithm is stronger in terms of consistently producing strong tables. This algorithm is also using an unconstrained domain compared to the local domain of a hill-climbing. Comparably, the hill climbing algorithm also produces optimal lookup tables, but it sometimes returns a local max before it gets there. Fixing this problem requires random restarts and sideways moves. The genetic algorithm produces the optimal lookups almost every time, but it typically takes longer than hill climbing (depending on the population size and number of generation of course).

Over all, the generic algorithm scores higher based on some predefined evaluation criteria:

Rank	Criteria	Genetic	Hill Climb
1	Consistency	(+) Produces similar tables every time	(-) Produces a range of results if not optimized with restarts/sideways moves
2	Domains	(+) Unconstrained domain	(-) Local domain
3	Speed	(-) Longer run time, depending on size of parameters	(+) Finds the local max within a few iterations

Objective

The objective of this assignment was to investigate the use of hill climbing and genetic algorithms on the prisoner's dilemma problem. We chose to also look at the effects of keeping the player's and opponent's history with a 2D array, as opposed to just the opponent's history. Ultimately, the objective of the prisoner's dilemma is to find the stronger prisoner, and we were able to produce optimal prisoners with both algorithms. Although, some opponents force the score to be low (by defecting continuously regardless of strategy), and no optimization algorithm is going to do better. See the below chart for lists of the average scores against different strategies :


Average Scores Against All Opponents
Opponent
Troll
TFT
TFTT
STFT
Prober
Pavlov
NEG
Gradual
ALLD
ALLC
Adaptive
HC
GA
2D
1D
0
1
2
3
4
5
6
Score

Appendices

Appendix A

Well Known Strategies

ALL-C:

Always cooperate

ALL-D:

Always defect

TFT:

Tit for Tat. Repeat opponent's last move.

TFTT:

Tit for Two Tat. Cooperates, but will defect if the opponent defects twice in a row.

STFT:

Suspicious Tit for Tat. Like Tit for Tat but begins by defecting.

Pavlov:

Cooperates on the first move. If a reward or temptation payoff is received in the last round then repeats last choice, otherwise chooses the opposite choice

NEG:

Negation. Plays the opposite move.

Our Human-Designed Strategies

Troll:

10% of the time it plays a random move and 90% of the time it plays the opponent's last move.

Prober:

Starts with D,C,C then proceeds to continually defects if the opponent Cooperated in either of the second or third moves, otherwise plays TFT.

Gradual:

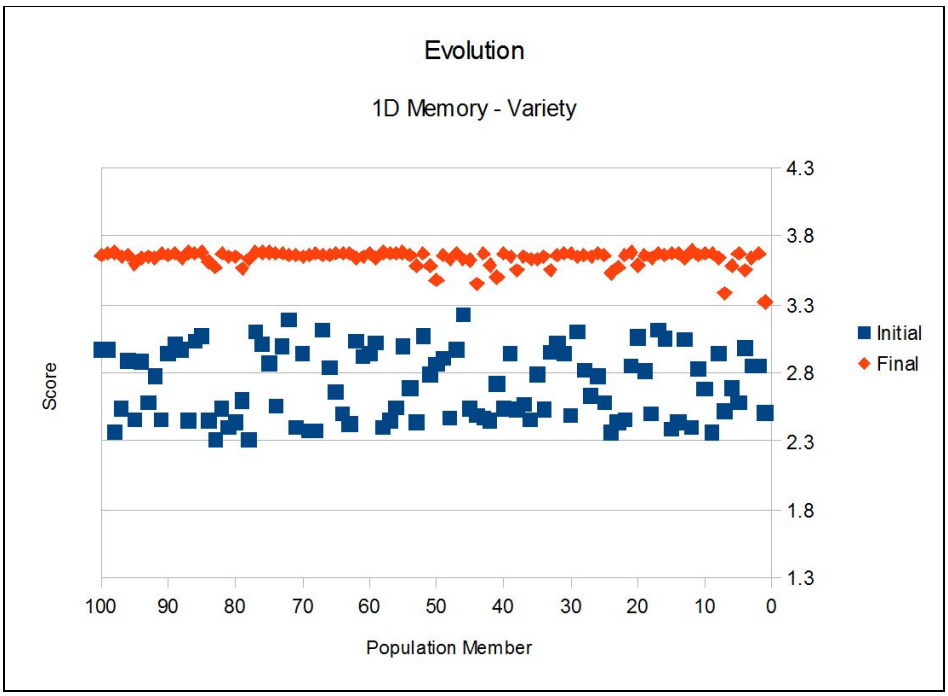
Cooperates on the first move and onwards but defects After the first defection of the other player, it defects one time and then proceeds to cooperate ... After the nth defection it reacts with n consecutive defections and then calms down its opponent by cooperating again.

Adaptive:

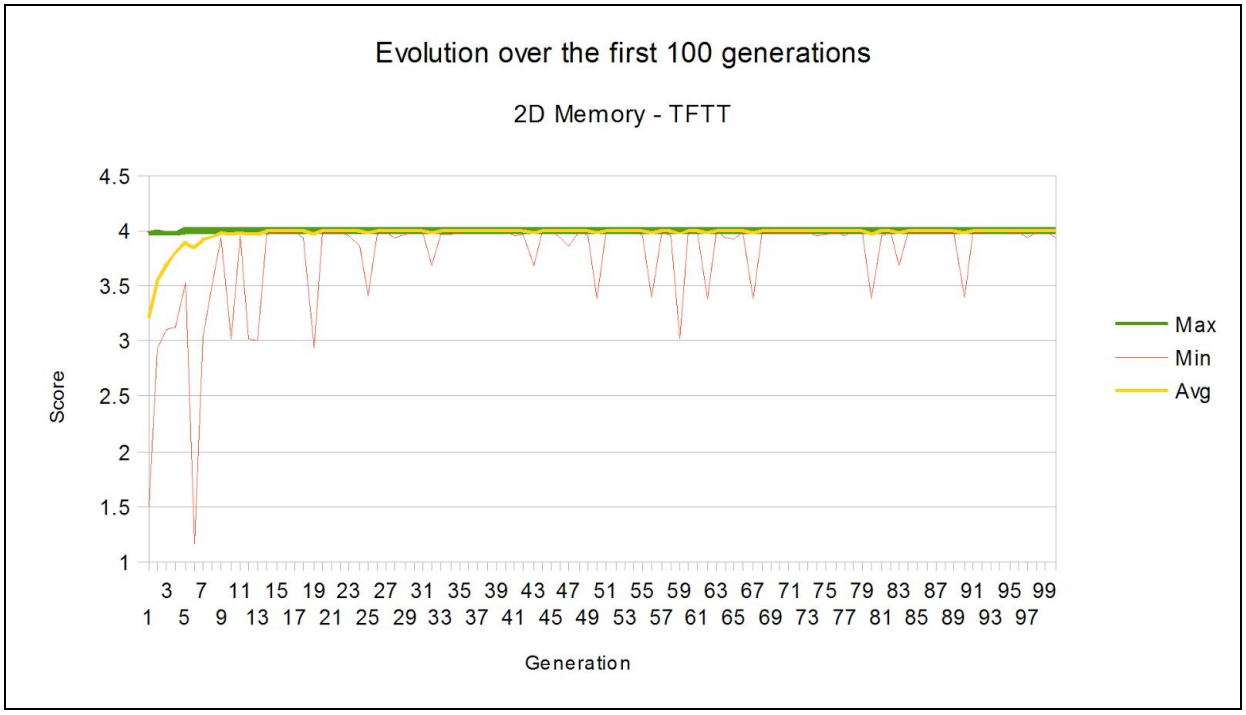
Starts with C,C,C,C,C,C,D,D,D,D,D and then takes choices which have given the best average score re-calculated after every move.

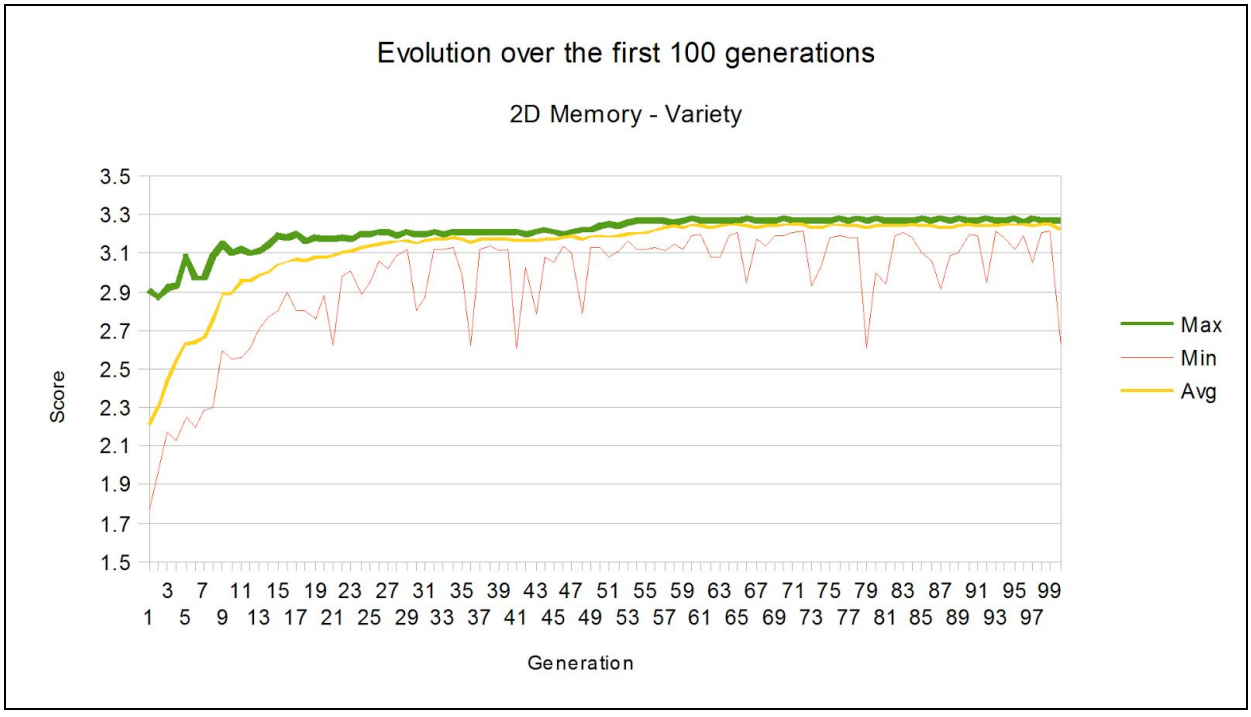
Appendix B

Scatterplots of population, before and after evolution, when training against a variety of opponents



Appendix C


Evolution over the first 100 generations
2D Memory - TFTT
Score
Generation
Max
Min
Avg


Evolution over the first 100 generations
2D Memory - Variety
Score
Generation
Max
Min
Avg