Evolving Fighting Creatures: A Look into Fitness and Competitive Coevolution*

Alex Cochrane
University of Idaho
coch9894@vandals.uidaho.edu

Jordan Leithart
University of Idaho
leit7193@vandals.uidaho.edu

ABSTRACT
An essential part of any genetic program is the use of a well defined fitness function that produces the desired outputs. For competitive coevolution this does not change, however, the ability to view the affects of different fitness functions on two simultaneously evolving populations can be seen through competition. Through competition, the value of a good fitness function will become apparent from the winner of the competition. We propose that it is possible to see the affects of different fitness functions through control of an individuals fitness which then can be normalized to compare to other individuals fitnesses in the population.

HOW WE DID IT

RESULTS

Keywords
Genetic Programming, Coevolution, Competitive Coevolution, Evolutionary Computation, Red Queen Effect, Fitness Function

1. INTRODUCTION
Fitness is the driving force of evolutionary computation.[2]

A big part of our project deals with the Red Queen Effect. The Red Queen Effect deals with the idea that a population may be improving some trait, even though their fitness might remain constant[1, p. 103]. Previous work done by Marc Ebner[1] states that before his work on evolving artificial plants in a single population, the usual setting for the Red Queen Effect was a predator population versus a prey population[1, p. 104]. Ebner states that ecological interactions are an important part evolution. In Ebner's plant ecosystem, it was common to see a fluctuating fitness around a constant level that sometimes even decreased.

*The full project lives at https://github.com/coch9894/Evolving-Fighting-Creatures

For our experiment, we would like to drop the distinction of predator and prey from the individuals. Instead we would like to look at the strategies of an individual and whether those strategies are effected by the Red Queen Effect. Our creatures who are fighting will only be distinguished by their strategy. This strategy will be developed by the differences in their fitness function. Therefore during evaluation we will compare two individuals at a time. Our engine will allow us to compare any two individuals we want with no bias going to either side.

What we would like to see is the Red Queen Effect disappear from our project because of our method of training and testing. Our project will go through a set number of rounds. A normal round will have a single population evaluating against other members of that same population. Every 10 rounds however we will evaluate a population against another. This will hopefully change the strategy of both populations and help them improve their fitnesses by keeping their opponents inconsistent.

While Ebner saw a fluctuating fitness around a constant value in his results[1, p. 120], we hypothesize that by keeping opponents inconsistent and varying we can see a small yet steady improvement in one populations best fitness. This best fitness is representative of an individual that has been trained to handle similar strategies to his own, but can still beat differing strategies. A final test will to test this individual against a saved set of best individuals from each round. If the best individual can pass a large majority of the tests (greater than 90%), our hypothesis will be justified. If it cannot pass our hypothesis will have failed.

2. THE EXPERIMENT
The simplest way to test our hypothesis was to build a genetic program that could be used in conjuncture with a graphics engine so our end results could be shown and quantified. To do so we built a genetic program that would evolve the instructions for an individual. To evaluate these individuals with different fitness functions it seemed best to use the idea of training two populations on separate fitness functions and then testing them against each other at regular intervals to get a grasp on the validity of each fitness function. After each generation, the best individual from *either* population was saved into a separate population as a control group. When evolution ended, the best individual was tested against this control group to check for the Red Queen Effect. This allowed us to observe if we were creating

a overall best solution or if we were cycling through a few strategies.

2.1 Individuals
Our individuals were made up of common and new behaviours that we needed to model our creatures. The following list of Non-Terminals and Terminals is all we used to make up our individual. Evolution was the key to setting them in the correct order and will be covered in the next section.

- Non-Terminals
  - Prog2 - When called during evaluation, pushes its left child, then its right child onto the correct player stack.
  - Prog3 - When called during evaluation, pushes its left child, middle child, then its right child onto the correct player stack.
- Terminals
  - Move - When called during evaluation, moves the correct players x position by cos(this->direction)*speed + 0.5 where the direction is where the player is facing. Also moves the correct players y position by sin(this->direction)*speed + 0.5.
  - Turn Left - When called during evaluation, computes this->direction += BASE_ANGLE which changes the direction the player is facing.
  - Turn Right - When called during evaluation, computes this->direction -= BASE_ANGLE which changes the direction the player is facing.
  - Aim - When called during evaluation, computes the angle to turn by using the current position of both players and the arc-tangent. In other words, we set the angle to turn to be atan((y2-y1)/(x2-x1)).
  - Shoot - When called during evaluation, adds a new bullet to the environment list containing environment variables.

2.2 Genetic Program
SELECTION

CROSSOVER

MUTATION

2.3 Fitness & Other Algorithms
To keep the fitnesses simple we decided to keep track the number of times an individual hit the opponent and the number of times that an opponent hit the individual. These were the numbers that our fitness functions would try to control. For example we might want to try and maximize the number of successful hits of the opponent and maximize the number of times the opponent hits the individual. In theory this doesn't sound like that great of a strategy but maybe against some other fitness function it would be great.

When we would like to compare the different functions however we needed some way to standardize these fitnesses. We came up with what we viewed as a perfect fitness and created a normalizing function to use the two prior values to represent it. Therefore our fitness functions are really controlling methods and strategies for getting the highest normalized fitness. The following is pseudo code for our normalization function:

```
if(this->numSuccess == this->numFail)
{
    this->fitness = 1;
}
else
{
    if(this->numFail == 0){
        this->numFail += 0.01;
    }
    this->fitness = numSuccess/numFail;
}
```

Because we decided to use a division of Success/Failures, we had to account for 0 in the denominator. Therefore if the Successes=Failures we just set our "stalemate" value or "average" outcome at 1. Therefore anything over 1 is assumed to be above average during the division. If the denominator is 0 and the numerator is not, we take this as an exceptionally good individual and wish to divide by 0.01 which essentially multiplies our answer by 100.

2.3.1 *Fitness Functions*
As seen in Table 1, we will be testing our hypothesis using four different fitness functions. Since the fitness functions are how we will control our strategies we wanted to keep them basic in their functionality. Therefore, we settled on maximizing and minimizing the success and failures. By doing this, we can see if a logical solution like maximizing an individuals hits on an opponent and minimizing an opponents hit on itself, is as good in simulation as we logically would think it would be.

2.3.2 *Turning Logic*
When training or testing to obtain fitness, it is unknown which side of the board the individual will be on or if they will always be on the same side of the board. To compensate for this, when a player is designated to be Player1, a turn_left action will cause them to rotate their facing direction in a positive direction and a turn_right action will cause them to rotate their facing direction in a negative direction. However, when a player is designated as Player2, a turn_left action will cause them to rotate their facing direction in a negative direction and a turn_right action will cause them to rotate their facing direction in a positive direction. This removes any chance of an individual evolving a strategy for only one side of the board. This will help create a universal strategy.

3. RESULTS
PICTURES

3.1 Fitness Function 1 VS. Fitness Function 2
PICTURES

Table 1: Evolutionary Characteristics

Algorithm	
Population size	
Selection method	
Elitism	
Crossover method	
Crossover rate	
Mutation method	
Operator/non-terminal set	
Terminal set	
Fitness function 1	
Fitness function 2	
Fitness function 3	
Fitness function 4	

3.2 Fitness Function 2 VS. Fitness Function 3

PICTURES

3.3 Fitness Function 3 VS. Fitness Function 4

PICTURES

3.4 Fitness Function 1 VS. Fitness Function 3

PICTURES

3.5 Fitness Function 1 VS. Fitness Function 4

PICTURES

3.6 Fitness Function 2 VS. Fitness Function 4

PICTURES

4. CONCLUSIONS

WE DO GOOD

5. REFERENCES

[1] M. Ebner. Coevolution and the red queen effect shape virtual plants. *Genetic Programming and Evolvable Machines*, 7(1):103–123, 2006.

[2] Y. Jin, M. Olhofer, and B. Sendhoff. A framework for evolutionary optimization with approximate fitness functions. *Evolutionary Computation, IEEE Transactions on*, 6(5):481–494, Oct 2002.

APPENDIX

A. GRAPHICS

Lets show a run every ten rounds?