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. We would like to see if logical fitness functions are always the best at producing good results and if training and testing have any affect on the appearance of the red queen effect.

RESULTS

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

1. INTRODUCTION
Fitness is the driving force of evolutionary computation. Unfortunately, they are also the most computationally expensive process within evolutionary computation. To solve this problem approximate models were created that were computationally efficient[3]. What we want to look at for our experiment is the effectiveness of fitness functions and their ability to perform against each other. Relating this to approximate models of fitness, which are usually assumed to be correct[3], we can see that the method for an approximate model will only be correct when the approximate model itself is globally correct. For our experiment we want to see if their is a correlation between logically correct strategies and winning competitions. Our fitness functions will differ in the way they value different aspects of an individual. To compare them however we had a need to create a normalizing function. More on these methods will be covered later.

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

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]. 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][2]. 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.

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. This method is similar to that of *evolution control*[3].

Evolution control deals with evaluating with both an original fitness function and an approximate function. One of its methods is to have control individuals who use the original fitness function. While evaluating, the approximate function is used except for at a set interval where the control or original function is used to produce control individuals. In our case our original fitness functions is to compare fitness functions and have individuals from two differing populations compete.

One thing that we want to do is be able to know what the best is at any given time. This has been looked at before and is called a master tournament[2]. The master tournament takes the best individuals and saves them at each generation so they can be tested against at the very end. For us we would like to use this idea for a similar purpose. We would like to see optimization happen. That is what our different fitness functions are really for.

While Ebner saw a fluctuating fitness around a constant value in his results[1], 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 Elitism

To maintain the best players across generations, we decided to create an elite for each generation. This elite was pitted against every other player in the population to ensure that it was indeed the best player. Each elite was chosen based on the training fitness assigned to that specific search, see Table 1. This particular area of our program was where most of the time was spent. We had to check each player against every other player which added up computational power quickly. Once we had gathered all the fitness, we averaged it out and returned the better player The following is pseudo code for our get elite function:

```
Player *one;
Player *two;
for(int i = 0; i < POP_SIZE - 1; i++){
  one = pop->GetIndividual(i);
  for(int j = i+1; j < POP_SIZE; j++){
    two = pop->GetIndividual(j);
    Evaluate(one, two, false);
    pop->fitnessPopulation[i] += one->
        GetTrainingFitness();
    pop->fitnessPopulation[j] += two->
        GetTrainingFitness();
  }
  pop->fitnessPopulation[i] = pop->
      fitnessPopulation[i]/POP_SIZE;
}
pop->fitnessPopulation[POP_SIZE] = pop->
    fitnessPopulation[POP_SIZE]/POP_SIZE;
int winningIndex = 0;
for(int i = 0; i < POP_SIZE; i++){
  if(pop->fitnessPopulation[winningIndex] <
      pop->fitnessPopulation[i]){
    winningIndex = i;
  }
}
return pop->GetIndividual(winningIndex);
```

2.3 Tournament Select

The tournament select function grabbed two random players. Then, it evaluated the two players against each other. We specifically made it so that the players could not be the same player. The tournament select then compared their fitness (based on the training fitness used for that specific genetic program), and selected whichever one was better. The following is pseudo code for our tournament select function:

```
int oneIndex = rand()%POP_SIZE;
int twoIndex = rand()%POP_SIZE;

```

Table 1: Evolutionary Characteristics

Algorithm	Generational
Population size	15 Players
Selection method	Tournament Select (tournament size was 2)
Elitism	Yes, the best individual per generation was copied over
Crossover method	Subtree crossover
Crossover rate	We took a random integer whose range was the size of each parent, then traversed the tree in a depth-first method. When I had visited the number of nodes equal to that random integer We crossed over on that node.
Mutation method	Node mutation; 10% chance that each node in each individual would be mutated
Operator/non-terminal set	{prog3, prog2}
Terminal set	{rotate left, rotate right, aim, move forward, shoot}
Fitness function 1	MaxMax; The goal of this training fitness was to maximize the number of times the opponent was hit by the player's bullets, and to maximize the number of times the player was hit.
Fitness function 2	MinMax; The goal of this training fitness was to minimize the number of times the opponent was hit by the player's bullets, and to maximize the number of times the player was hit.
Fitness function 3	MinMin; The goal of this training fitness was to minimize the number of times the opponent was hit by the player's bullets, and to minimize the number of times the player was hit.
Fitness function 4	MaxMin; The goal of this training fitness was to maximize the number of times the opponent was hit by the player's bullets, and to minimize the number of times the player was hit.

```
  while(oneIndex == twoIndex){
    oneIndex = rand()%POP_SIZE;
  }

  Player *one = pop->GetIndividual(oneIndex);
  Player *two = pop->GetIndividual(twoIndex);
  Evaluate(one, two, false);
  if(two->GetTrainingFitness() > one->
      GetTrainingFitness()){
    return twoIndex;
  }

  return oneIndex;
```

2.4 Crossover

The crossover function took the two winners of the above tournament select and crossed over at a random number, see Table 1. This algorithm performed a depth-first traversal through the player tree. Once it arrived at the random node, it crossed over with the second player. The actual crossover algorithm required a deep copy of the subtree to be made and then attached. We used a temporary tree to hold that subtree and then attached it by setting the parent node equal to the tmpParent's node. Here is some pseudo code to explain this:

```
  node *tmpNode = firstNode->copyTree(firstNode
      , firstNode->parent);
  firstNode = firstNode->copyTree(secondNode,
      firstNode->parent);
  secondNode = secondNode->copyTree(tmpNode,
      secondNode->parent);
```

2.5 Mutation

The mutation function traversed the player tree in a depth-first traversal. At every node, a random number between 1 and 100 was generated. If the number was less than 10, the current node was mutated. To make sure that the node didn't change types (terminal to non-terminal and visa versa), if the node was a terminal a random terminal node was generated. Because we only had two non-terminals in our set, see Table 1, we just switched prog3 over to prog2 and visa versa.

2.6 Evaluate

The evaluate function was the meat of our genetic program. It took in two separate players and pitted them against each other, counting up the number of times each player was hit. Because our player tree (which had all of our actions in it), was in a separate class from the player class, we had to create a depth-first traversal without using recursion. To do this effectively, we had to use a stack data structure. By pushing right to left, and then popping the top off we were able to simulate a recursive depth-first traversal. In the interest of time, we gave each player only 300 actions. This number would be decreased every time an action was performed to ensure that only 300 actions were taken.

2.7 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.

2.7.1 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.

2.7.2 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

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

WHAT WE FOUND OUT

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] D. Floreano and S. Nolfi. God save the red queen! competition in co-evolutionary robotics.

[3] 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?