Evolving Fighting Creatures: A Look into Fitness and Competitive Co-Evolution

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 co-evolution 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, Co-Evolution, Competitive Co-Evolution, Evolutionary Computation, Red Queen Effect, Fitness Function

1. INTRODUCTION

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 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.

2.4 Tables

Because tables cannot be split across pages, the best placement for them is typically the top of the page nearest their initial cite. To ensure this proper "floating" placement of tables, use the environment **table** to enclose the table's contents and the table caption. The contents of the table itself must go in the **tabular** environment, to be aligned properly in rows and columns, with the desired horizontal and vertical rules. Again, detailed instructions on **tabular** material is found in the *LaTeX User's Guide*.

Table 1: Frequency of Special Characters

Non-English or Math	Frequency	Comments
Ø	1 in 1,000	For Swedish names
π	1 in 5	Common in math
\$	4 in 5	Used in business
Ψ_1^2	1 in 40,000	Unexplained usage

Figure 1: A sample black and white graphic (.eps format).

Immediately following this sentence is the point at which Table 1 is included in the input file; compare the placement of the table here with the table in the printed dvi output of this document.

To set a wider table, which takes up the whole width of the page's live area, use the environment **table*** to enclose the table's contents and the table caption. As with a single-column table, this wide table will "float" to a location deemed more desirable. Immediately following this sentence is the point at which Table 2 is included in the input file; again, it is instructive to compare the placement of the table here with the table in the printed dvi output of this document.

2.5 Figures

Like tables, figures cannot be split across pages; the best placement for them is typically the top or the bottom of the page nearest their initial cite. To ensure this proper "floating" placement of figures, use the environment **figure** to enclose the figure and its caption.

This sample document contains examples of **.eps** and **.ps** files to be displayable with LaTeX. More details on each of these is found in the *Author's Guide*.

As was the case with tables, you may want a figure that spans two columns. To do this, and still to ensure proper "floating" placement of tables, use the environment **figure*** to enclose the figure and its caption.

Note that either **.ps** or **.eps** formats are used; use the `\epsfig` or `\psfig` commands as appropriate for the different file types.

2.6 Theorem-like Constructs

Figure 2: A sample black and white graphic (.eps format) that has been resized with the `epsfig` command.

Table 2: Some Typical Commands

Command	A Number	Comments
`\alignauthor`	100	Author alignment
`\numberofauthors`	200	Author enumeration
`\table`	300	For tables
`\table*`	400	For wider tables

Figure 3: A sample black and white graphic (.ps format) that has been resized with the `psfig` command.

Other common constructs that may occur in your article are the forms for logical constructs like theorems, axioms, corollaries and proofs. There are two forms, one produced by the command `\newtheorem` and the other by the command `\newdef`; perhaps the clearest and easiest way to distinguish them is to compare the two in the output of this sample document:

This uses the **theorem** environment, created by the `\newtheorem` command:

THEOREM 1. *Let f be continuous on $[a,b]$. If G is an antiderivative for f on $[a,b]$, then*

$$\int_a^b f(t)dt = G(b) - G(a).$$

The other uses the **definition** environment, created by the `\newdef` command:

Definition 1. If z is irrational, then by e^z we mean the unique number which has logarithm z:

$$\log e^z = z$$

Two lists of constructs that use one of these forms is given in the *Author's Guidelines.*

and don't forget to end the environment with figure*, not figure!

There is one other similar construct environment, which is already set up for you; i.e. you must *not* use a `\newdef` command to create it: the **proof** environment. Here is a example of its use:

PROOF. Suppose on the contrary there exists a real number L such that

$$\lim_{x\to\infty} \frac{f(x)}{g(x)} = L.$$

Then

$$l = \lim_{x\to c} f(x) = \lim_{x\to c} \left[gx \cdot \frac{f(x)}{g(x)} \right] = \lim_{x\to c} g(x) \cdot \lim_{x\to c} \frac{f(x)}{g(x)} = 0 \cdot L = 0,$$

which contradicts our assumption that $l \neq 0$. $\square$

Complete rules about using these environments and using the two different creation commands are in the *Author's Guide*; please consult it for more detailed instructions. If you need to use another construct, not listed therein, which you want to have the same formatting as the Theorem or the Definition[?] shown above, use the `\newtheorem` or the `\newdef` command, respectively, to create it.

3. RESULTS

APPENDIX

A. HEADINGS IN APPENDICES

The rules about hierarchical headings discussed above for the body of the article are different in the appendices. In the **appendix** environment, the command **section** is used to indicate the start of each Appendix, with alphabetic order designation (i.e. the first is A, the second B, etc.) and a title (if you include one). So, if you need hierarchical structure *within* an Appendix, start with **subsection** as the highest level. Here is an outline of the body of this document in Appendix-appropriate form:

A.1 Introduction

A.2 The Body of the Paper

A.2.1 Type Changes and Special Characters

A.2.2 Math Equations

Inline (In-text) Equations

Display Equations

A.2.3 Citations

A.2.4 Tables

A.2.5 Figures

A.2.6 Theorem-like Constructs

A Caveat for the TEX Expert

A.3 Conclusions

A.4 Acknowledgments

A.5 Additional Authors

This section is inserted by LATEX; you do not insert it. You just add the names and information in the `\additionalauthors` command at the start of the document.

A.6 References

Generated by bibtex from your .bib file. Run latex, then bibtex, then latex twice (to resolve references) to create the .bbl file. Insert that .bbl file into the .tex source file and comment out the command `\thebibliography`.

Figure 4: A sample black and white graphic (.eps format) that needs to span two columns of text.

B. MORE HELP FOR THE HARDY

The acm_proc_article-sp document class file itself is chock-full of succinct and helpful comments. If you consider yourself a moderately experienced to expert user of LaTeX, you may find reading it useful but please remember not to change it.