Evolution of Life Cycle Differentiation using Genetic Programming

Justin C. Haugh
CS426 Final Project
Department of Computer Science
Stanford University
Stanford, California 94305
jhaugh@cs.stanford.edu
http://www.stanford.edu/people/jhaugh

Abstract: This paper describes the emergence of age and sexual differentiation among computer programs in a digital ecosystem. Programs to control the behavior of simulated mice are randomly generated, and are evolved over time using a steady-state genetic programming system with tournament selection. Problems and early failures are described, and solutions are discussed. Evolved programs demonstrating life stage differentiation are examined with a comparison of relative fitness.

1 Introduction and Overview

Virtually every form of complex life in the natural world undergoes change over its life cycle. Advanced life forms such as primates develop from prenatal forms into infancy, childhood, adolescence, adulthood, and old age. Other mammals pass through most of the same stages. Even simple organisms such as insects develop from pupae to larvae to adults, with each state boasting a unique appearance and characteristic set of behaviors.

The rich diversity of natural organisms is not currently shared with their computer-based counterparts. Many computer models of life are restricted to organisms with a single life stage, or program; these digital creatures follow the same series of instructions every time step until the program is terminated. Hence, the behavior exhibited by many digital organisms lacks the life cycle diversity of natural organisms. Moreover, self-replicating programs often reproduce asexually via copying with mutation and do not enjoy the well-known genetic and task-specialization benefits of sexuality.

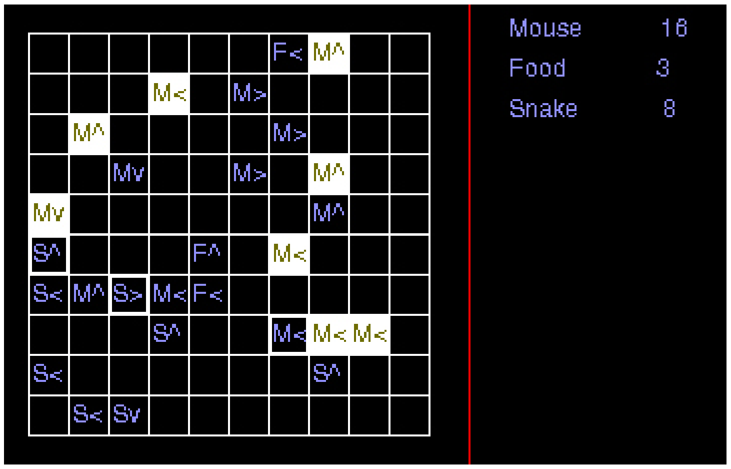
This paper investigates the emergence of life cycle differentiation among digital organisms. A population of random mouse control programs is generated at the start, and behavior diverges between adult male, adult female, and child programs. The hypothesis is that life stage differentiation will enable the simulate creatures to compete against other creatures with greater success than a single-stage creature.

2 Methods

The overall goal is to evolve a successful control program for a multi-stage digital creature by means of genetic programming (GP). I leveraged the Predator world infrastructure that I was given for a course project in CS193D: C++ and Object-Oriented Programming. The Predator world was used to simulate each creature control program to get a fitness measurement. For convenience, I will refer to the simulated creatures as Mice. Fitness is defined to be the sum, over 100 time steps, of the number of Mice alive at each time step, averaged over two 100-step simulations.

2a The Predator World

Digital creatures exist in a 10x10 square world of grid locations. Each location can hold one creature at a time. Creatures can move, turn, eat, mate, and attack one another. There are three kinds of creatures: Mice, Snakes, and Food. Mice are controlled by genetically evolved programs, while Snakes execute a fixed program. During a simulation, the world is seeded by 15 creatures of each type, each placed in a random location. For each turn, each creature is given a chance to act. Every time step, an additional Food creature is added to the world.



```
Mouse     16
Food       3
Snake      8
```

Figure 1: A typical run of the Predator simulation.

Mice can either be male or female, randomly chosen at birth with 0.5 probability for each. Adult mice of both genders have a strong attack that is capable of killing a Snake in one blow, unless the Snake has recently feasted as has high energy. Male mice have the same capabilities as female mice, except that female mice can become pregnant after mating. In order to mate, a male and a female have to be in adjacent squares, the female must not be pregnant, and one of them has to execute the Mate command. Pregnancy lasts for five time steps, during which the female cannot change location and has limited attack ability.

When pregnancy ends, the female attempts to give birth to a new child mouse. If there is an open square next to the female, that square is filled by the new child, otherwise the female remains pregnant until such a square becomes available or the female dies. Child mice cannot move and have low energy, making them vulnerable to Snakes. After five turns of childhood, mice become adults and have full abilities befitting their gender.

Snakes are powerful and fast. They can move two squares per turn and wander about randomly looking for enemy creatures. When adjacent to a mouse or food, Snakes will attack; they are powerful enough to kill a mouse child in one blow, but can be beaten by a single non-pregnant adult mouse. When a Snake has recently feasted, it gains the ability to reproduce and generates an egg inside itself. After four turns, a Snake will hatch its egg in an adjacent square if one is available.

All creatures expend a point of energy every turn, and age by one turn. After 40 turns, a creature is considered old, and its energy rapidly declines each turn thereafter. A creature with no energy left is removed from the world.

2b Mouse Control Programs

Each turn, a mouse executes its control program, taking up to two actions if it is an adult, or one action if it is a pregnant female or child. Programs are represented as binary trees, with internal nodes chosen from the function set and the leaves chosen from the terminal set. Functions may execute one or both sub-branches depending on the specific function and state of the simulation.

The genetic programs undergoing adaptation in the Mouse Life Cycle Problem have complex structure. Rather than a single result-producing branch, each genetic program contains three main branches: one to be executed by adult male mice, one for adult female mice, and one for children. Additionally, three automatically defined functions (ADFs) are available, one for the males (ADF0), one for females (ADF1), and one for all adult mice (ADF2).

The GP infrastructure operates on the code for an entire species; hence, the male, female, and child branches undergo co-evolution. All three branches contribute to the overall species fitness, since the simulated mice begin life as children and are evenly divided between males and females. Different branches have access to different, overlapping subsets of the set of functions and terminals. Table 1 presents the tableau for the Mouse Life Cycle Problem, and lists the function and terminal sets for each result-producing branch. Table 2 describes the terminals, and Table 3 explains the set of functions.

Table 1: Tableau for the Mouse Life Cycle Problem

Objective:	To populate the 10x10 world with as many mice as fast as possible.
Terminal set: (Male)	`ADF0, ADF2, Attack, Eat, Mate, Donate, TurnRight, TurnLeft, Move, MoveRandom, MoveNorth, MoveEast, MoveSouth, MoveWest, MoveTowardMate, MoveTowardEnemy, MoveAwayEnemy, MoveTowardFood`
Terminal set: (Female)	`ADF1, ADF2, Attack, Eat, Mate, Donate, TurnRight, TurnLeft, Move, MoveRandom, MoveNorth, MoveEast, MoveSouth, MoveWest, MoveTowardMate, MoveTowardEnemy, MoveAwayEnemy, MoveTowardFood`
Terminal set: (Child)	`Attack, Eat, Donate, TurnRight, TurnLeft`
Function set: (Male)	`TwoCommands, IfHurt, IfOld, IfMateNearby, IfChildNearby, IfEnemyNearby, IfFoodNearby, IfEnemyAhead, IfMateAhead, IfFoodAhead, IfPregnantNearby`
Function set: (Female)	`TwoCommands, IfHurt, IfOld, IfMateNearby, IfChildNearby, IfEnemyNearby, IfFoodNearby, IfEnemyAhead, IfMateAhead, IfFoodAhead, IfPregnant`
Function set: (Child)	`TwoCommands, IfHurt, IfMateNearby, IfChildNearby, IfEnemyNearby, IfFoodNearby, IfEnemyAhead, IfMateAhead, IfFoodAhead, IfPregnantNearby`
Fitness cases:	Two independent, identically distributed random population simulations. Each simulation runs for 100 time steps.
Raw fitness:	The average over all fitness simulations of the sum, over all time steps, of the number of mice alive at each time step.

Parameters:	Variable. Most successful: M = 2000, G = 100. Others: M = 1000, G = 100, M = 500, G = 100, M = 10000, G = 10.
Success predicate:	None.
Result designation:	The individual with the highest fitness over the entire run is designated the result.

Table 2: Terminals

Terminal	**Description**
`Attack`	If an enemy is adjacent, injures the enemy and adds strength to attacker. Attack power is based on age and pregnancy status.
`Eat`	Eats one piece of food if one is present in an adjacent square.
`Mate`	Mates with a mouse of the opposite sex if one is adjacent.
`Donate`	Gives energy to an adjacent creature of the same species.
`TurnRight`	Turns the creature 90 degrees clockwise.
`TurnLeft`	Turns the creature 90 degrees counter-clockwise.
`Move`	Advances the creature one square forward if it is in bounds.
`MoveRandom`	Picks a random direction and moves one square in that direction.
`Move<Direction>`	Moves one square in a given direction: North, South, East, or West.
`MoveTowardMate`	Finds the closest mate within range, and moves one square in their direction. Range is defined to be any square less than four steps away.
`MoveTowardEnemy`	Finds the closest enemy in range and moves one square towards it.
`MoveAwayEnemy`	Finds the closest enemy in range and moves one square away from it.
`MoveTowardFood`	Finds the closest food in range and moves one square towards it.

Table 3: Functions

Function	**Description**
`TwoCommands(x,y)`	Executes the command represented by `x`, then the command `y`.
`IfHurt(x,y)`	If the mouse is low on energy, executes `x`, otherwise executes `y`.
`IfOld(x,y)`	If the mouse is very old, executes `x`, otherwise executes `y`.
`IfPregnant(x,y)`	If the mouse is pregnant, executes `x`, otherwise executes `y`.
`IfEnemyNearby(x,y)`	If there is an enemy in an adjacent square, executes `x`, otherwise executes `y`.
`IfMateNearby(x,y)`	If there is a viable mate in an adjacent square, executes `x`, otherwise executes `y`.
`IfChildNearby(x,y)`	If there is a child mouse in an adjacent square, executes `x`, otherwise executes `y`.
`IfFoodNearby(x,y)`	If there is food in an adjacent square, executes `x`, otherwise executes `y`.
`IfPregnantNearby(x,y)`	If there is a pregnant female mouse in an adjacent square, executes `x`, otherwise executes `y`.
`IfEnemyAhead(x,y)`	Searches for enemies in the direction the mouse is facing, up to four

	squares away, and if one exists, executes `x`, otherwise executes `y`.
`IfMateAhead(x,y)`	Searches for an mate in the direction the mouse is facing, up to four squares away, and if one exists, executes `x`, otherwise executes `y`.
`IfFoodAhead(x,y)`	Searches for food in the direction the mouse is facing, up to four squares away, and if one exists, executes `x`, otherwise executes `y`.

2c System Architecture

Simulated digital ecosystems with reproducing agents present an interesting problem to a genetic programming system designer. Since the objects that are being evolved are themselves potentially viable self-reproducing entities, the system designer can elect to simply utilize the simulated world itself as the genetic programming infrastructure. Creatures that are more fit will survive to reproduce in the world, and creatures that are less fit will die out and be replaced. If the genetic operations of crossover and mutation are performed when two digital creatures mate, then the simulated world will essentially perform the identical operations as the standard genetic programming infrastructure: fitness-proportionate selection, reproduction, crossover and mutation. Indeed, GP software is intended to model the very same real-world population processes that lead to evolution in natural organisms.

The alternative is to use a standard GP package to simulate the evolution of species program code. The simulated world, then, is simply used to generate a fitness measurement for each program in the population. The world is seeded with multiple individuals, each of which posses the same program from the high-level GP population. The program code, when executed by the individuals, causes them to behave and reproduce for a given period of time.

I chose to use the standard GP infrastructure to simulate genetic evolution. While it is simple to implement either model, the standard approach has the advantage of side-stepping potentially complicated problems of population dynamics. If the simulation itself were to handle the genetic evolution process, the success of the program would depend on factors such as population size and diversity over time, dominance of single individuals, and extinction. Using an explicit GP infrastructure ensures constant population size and gives the designer the ability to control diversity.

2d Genetic Programming System

My initial plan was to use the **lilgp** software package for the GP infrastructure. However, the Predator infrastructure was written in C++, and my attempts to compile and link lilgp with Predator failed. Eventually I decided on the **gpc++ 0.40** software package; the 0.40 indicates it is not commercial software, but the GP infrastructure is fully functional.

However, **gpc++** does not use the standard two-population model for genetic programming used by **lilgp** and preferred by Koza. It was written for use in settings with limited memory, and so opts for a steady-state GP system with tournament selection. In essence, a steady-state GP maintains a single population of programs; tournaments between randomly-selected programs are set up to select two winners and a loser; the two winners produce a child via crossover, and the loser's place in the population is taken by the new child. One "generation" is simply a number of selection-replacement operations performed in a sequence; reproduction is therefore implicit. Mutation occurs once per generation with variable frequency; I used 0.001 for the mutation rate.

According to Koza, the two versions of GP have comparable performance, so I did not attempt to re-implement a two-population GP system for this paper. To summarize, then, I used the standard genetic operators of fitness-proportionate selection, crossover, and mutation. There was no need to define a percentage for crossover vs. reproduction, due to the nature of the steady-state GP system.

2e Platform

Each run of the GP system was performed on one of the **fable** machines on the Stanford University Network. Each is a Sun Ultra60 Creator3D machine with 384MB RAM, running the Solaris 8 operating system. A single run with M=2000, G=100 took an average of six hours.

3 Results

The accumulation of results for this project was more of a process than a single event. Initially, there were a number of problems with the simulation structure and parameter selection, resulting in changes which led to further problems. In the end I was successfully able to evolve programs for the Mice that exhibited interesting behavior and demonstrated life cycle differentiation. First I will discuss selected problems, and then I will describe the final successful results.

3a Parameter Selection

Initially, I tried very small population sizes, deep program trees, and many generations. It became clear over several runs that the three-life-stage co-evolution problem has a complex and subtle fitness landscape that could not be successfully navigated by small population sizes such as 50-200. Indeed, the larger the population I tried, the better the results. Population sizes of 500, 1000, then 2000 became typical. In the end, runs with population size 2000 seemed to produce the optimal balance of time and fitness. Figure 2 demonstrates the inherent tradeoffs between population size and number of generations, and the resulting fitness gains from larger populations and tweaked population dynamics.

Maximum program depth, on the other hand, benefited from a heavy hand. I started with a maximum initial program depth of 6 and maximum crossover depth of 17. However, this produced bloated and extremely redundant programs, large portions of which could never be reached due to contradictory conditionals. Eventually, maximum initial and crossover depths of five seemed to result in successful yet concise programs with only a small degree of redundancy. It is probably the case that this limited the diversity of the resulting programs, but the programs that did evolve were quite successful despite the limitation.

My initial runs sometimes took 1000 or more generations, but were able to finish with several hours due to the small population sizes. Later, more successful runs took much longer to process since the creatures were surviving longer, and because the population sizes were larger. Hence a run of M=2000, G=50 was taking some eight hours by the end of the research effort.

3b Population Dynamics

Equal to the task of tuning the GP parameters was the tuning of the population simulation itself. Getting the right balance between Mice, Snakes, and Food was a difficult challenge that required extensive testing. If either Mice or Snakes became too powerful initially, it would wipe out the other species and dominate the world. This was an especially difficult problem because the Snakes did not evolve in concert with the Mice, so they needed to be competitive enough to present a threat to highly evolved Mice, while remaining docile enough to allow random Mice programs a chance to grow without being prematurely extinguished. Eventually, however, the Mice become too intelligent about the balance between mating, eating, and defending, and a good Mouse program can wipe out the Snakes quickly.

The small world and generally small populations presented an additional problem for the Mice. Initially, the gender of each mice was selected completely at random. However, fluctuations in the ratio of males to females in populations with typical sizes of 10-20 can be disastrous. A run of five consecutive male births is usually enough to wipe out a population of mice, as the older females become too weak to breed and the males are left with no one to mate with. The solution was to enforce a perfect male-female birth ratio, which ensures future mating compatibility for a population of a given size, and makes other factors more important in adaptive success.

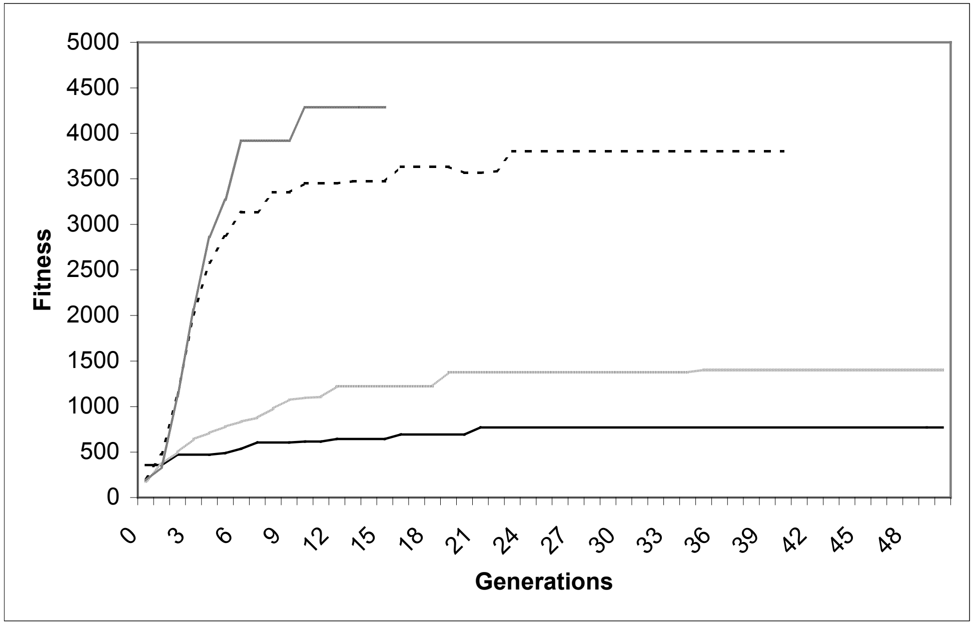


Figure 2: Fitness of several combinations of simulation settings and GP parameters. Top: M=2000, final simulation settings. Second: M=2000, partially refined simulation settings. Third: M=2000, initial simulation settings. Bottom: M=500, initial sim. settings.

3c Successful Programs

The most successful programs were those that exhibited the greatest differentiation between life stages. In general, the successful programs had Males that searched for food, chased after and attacked enemies, and then made attempts to mate if the other two conditions did not succeed. The most successful programs were able to quickly find and eliminate the Snake population before it could reproduce and become a threat to weak Mice. However, all the successful programs had a high degree of complexity, so it is difficult to provide an analytical summary of successful behavior. Moreover, all programs to succeed are by definition being simultaneously executed by tens of creatures at once, so it is hard to track a single creature and follow its behavior.

The most computationally intensive result was the following program, the best output of a nine-hour run of M=10000, G=10 culminated with the following program with fitness 3885:

```
Male: ( ( TwoCommands ( IfFoodNearby Eat ( TwoCommands Attack ADF2 ) ) (
IfFoodNearby Eat ( TwoCommands ( TwoCommands Attack ADF2 ) ( TwoCommands (
TwoCommands ( IfEnemyNearby Attack MoveTowardMate ) ( TwoCommands Attack ADF2
) ) ( IfFoodNearby Eat ( TwoCommands ( IfMateAhead TurnRight MoveTowardMate )
( TwoCommands Attack ADF2 ) ) ) ) ) ) ) ) )
ADF0: ( ( IfEnemyNearby MoveTowardEnemy MoveAwayEnemy ) )
Female: ( ( TwoCommands ( TwoCommands Attack ( TwoCommands ( TwoCommands
MoveTowardMate Eat ) ( TwoCommands Attack ( IfEnemyNearby Attack Mate ) ) ) )
( TwoCommands Attack ( TwoCommands ( TwoCommands MoveTowardMate Eat ) (
TwoCommands ( TwoCommands MoveTowardMate Eat ) ( TwoCommands ( TwoCommands
MoveTowardMate Eat ) ( TwoCommands Attack ( IfEnemyNearby Attack Mate ) ) ) )
) ) ) )
ADF1: ( ( IfOld MoveSouth MoveAwayEnemy ) )
Child: ( ( IfEnemyNearby ( IfFoodAhead ( IfFoodAhead Attack TurnLeft ) (
IfMateNearby Eat Donate ) ) ( IfHurt TurnRight TurnLeft ) ) )
ADF2: ( ( IfChildNearby ( IfMateAhead Mate MoveTowardMate ) ( IfEnemyNearby
Donate MoveRandom ) ) )
```

The Male in this program first checks if there is food to be eaten; it answers the age old question of food vs. reproduction in favor of the former. It will also use its second move to eat if there is another piece of food. Otherwise, it tries to attack an enemy; if there is no enemy to attack or food remaining, it moves towards a mate. The females are fairly straightforward: attack any enemies, then attempt to move towards a mate. One second move, it tries to attack any nearby enemies, and then tries to mate. The repetition of this same pattern is the basic structure. Hence, both males and females move toward each other, but males focus on food first. The children try to attack any enemies, and eat, but only if a mate is nearby. Clearly nonsensical behavior, but since the overall species fitness was selected for, it was not deleterious to the species survival. ADFs do not play a major role in this species.

The most successful program of all was the following, discovered in generation 15 of a M=2000 run, with fitness 4287:

```
Male: ( ( TwoCommands ( TwoCommands ( IfFoodNearby ( IfHurt MoveWest Eat ) (
TwoCommands Attack Mate ) ) ( TwoCommands ( TwoCommands Attack Mate ) (
TwoCommands ( TwoCommands ( TwoCommands Attack Mate ) ( TwoCommands (
TwoCommands ( IfFoodNearby ( IfHurt MoveWest Eat ) ( IfMateNearby ADF0
MoveTowardEnemy ) ) ( TwoCommands ( TwoCommands Attack Mate ) ( TwoCommands (
IfChildNearby MoveNorth ( TwoCommands ( TwoCommands ( TwoCommands Attack Mate
) ( TwoCommands ( TwoCommands Attack Mate ) ( IfFoodNearby ( IfHurt MoveWest
Eat ) ( TwoCommands MoveTowardMate ADF0 ) ) ) ) ( TwoCommands Attack Mate ) )
) ( TwoCommands ( IfFoodNearby ( IfHurt MoveWest Eat ) ( TwoCommands (
TwoCommands Attack Mate ) ( TwoCommands ( TwoCommands Attack Mate ) (
IfFoodNearby ( IfHurt MoveWest Eat ) ( TwoCommands MoveTowardMate ADF0 ) ) )
) ) ( TwoCommands ( TwoCommands ( TwoCommands ( IfFoodNearby ( IfHurt
MoveWest Eat ) ( TwoCommands Attack Mate ) ) ( TwoCommands ( TwoCommands
Attack Mate ) ( TwoCommands ( TwoCommands ( TwoCommands Attack Mate ) (
TwoCommands ( TwoCommands ( IfFoodNearby ( IfHurt MoveWest Eat ) (
IfMateNearby ADF0 MoveTowardEnemy ) ) ( TwoCommands ( TwoCommands Attack Mate
) ( TwoCommands ( IfChildNearby MoveNorth ( TwoCommands ( TwoCommands (
TwoCommands Attack Mate ) ( TwoCommands ( TwoCommands Attack Mate ) (
IfFoodNearby ( IfHurt MoveWest Eat ) ( TwoCommands MoveTowardMate ADF0 ) ) )
) ( TwoCommands Attack Mate ) ) ) ( TwoCommands ( IfFoodNearby ( IfHurt
MoveWest Eat ) ( IfMateNearby ADF0 MoveTowardEnemy ) ) ( TwoCommands (
TwoCommands Attack Mate ) ( TwoCommands ( TwoCommands ( TwoCommands Attack
Mate ) ( TwoCommands ( TwoCommands Attack Mate ) ( IfFoodNearby ( IfHurt
MoveWest Eat ) ( TwoCommands MoveTowardMate ADF0 ) ) ) ) ( TwoCommands Attack
Mate ) ) ) ) ) ) ) MoveEast ) ) ( TwoCommands Attack Mate ) ) ) ) MoveEast )
( TwoCommands ( TwoCommands ( TwoCommands Attack Mate ) ( TwoCommands (
TwoCommands Attack Mate ) ( IfFoodNearby ( IfHurt MoveWest Eat ) (
TwoCommands MoveTowardMate ADF0 ) ) ) ) ( TwoCommands Attack Mate ) ) ) ) ) )
) MoveEast ) ) ( TwoCommands Attack Mate ) ) ) ) MoveEast ) )
ADF0: ( ( IfOld Donate ( TwoCommands Mate MoveWest ) ) )
Female: ( ( IfEnemyNearby Attack ( IfEnemyNearby Attack ( IfFoodNearby Eat
MoveTowardFood ) ) ) )
ADF1: ( ( IfHurt MoveRandom TurnLeft ) )
Child: ( ( IfEnemyAhead Attack TurnLeft ) )
ADF2: ( ( IfMateNearby ( IfFoodAhead ( IfChildNearby ( IfMateAhead MoveNorth
TurnLeft ) ( IfEnemyNearby MoveRandom MoveNorth ) ) ( IfMateAhead (
IfMateAhead MoveNorth TurnLeft ) ( IfEnemyAhead Donate Mate ) ) ) ( IfHurt (
IfPregnant MoveTowardMate TurnLeft ) ( IfChildNearby ( IfMateAhead MoveNorth
TurnLeft ) ( IfEnemyNearby MoveRandom MoveNorth ) ) ) ) )
```

I cannot pretend to offer an analysis of the Male code. However, the Females are effortlessly clear: attack enemies if nearby, eat food if nearby, and if not, move toward food. Interestingly, the females make no attempt to mate, relying on the males to find them and try to mate. The males make use of ADF0 unlike in the previous example.

4 Conclusion

This paper has demonstrated that life cycle diversity, and the advantages of specialization, can be achieved using genetic programming. Functions for controlling each life stage can be co-evolved and will automatically specialize based on the fitness landscape. Perfecting population dynamics can be a difficult challenge that may be best solved using co-evolution of predator and prey creatures.

5 Future Work

A future study into life cycle differentiation should seek to co-evolve the predators and prey in an ecosystem, and give each equal weight in terms of GP algorithm time. The natural world does not remain fixed while a particular species adapts, and it was problematic to successfully evolve a single side of the predator-prey relationship without the other. Too weak prey and the predator does not evolve; to strong prey and the predator cannot get a foothold on evolution. However, the computer time necessary for such a simulation may be prohibitive, since the single-species case analyzed in this paper was highly intensive.

6 References

Fraser, Adam. “Genetic Programming: gpc++ 0.40.” http://www.cs.ucl.ac.uk/research/genprog/.

Koza, John R. 1992. *Genetic Programming I.* Cambridge, Massachusetts: The MIT Press.

“lil-gp 1.1Beta.” http://garage.cps.msu.edu/software/lil-gp/lilgp-index.html. Genetic Algorithms Research and Applications Group, Michigan State University.