Evolutionary Adaptive Methods Applied to a Darwinian Predator-Prey Model

Paul Baird-Smith and Ross Flieger-Allison

May 17, 2017

1 Introduction

Biology presents us with a number of ways of changing the overall genetics of a gene pool. In a gene pool of organisms that reproduce via sexual reproduction, there exist many methods of changing DNA so that sufficient genetic variety exists among the offspring. Some of these methods include crossover and gene transfer between chromosomes, for example. Interestingly, the same applies to gene pools of organisms that undergo asexual reproduction; here, the genetics of the organisms are often changed via mutation, conjugation, and translocation, to name a few.

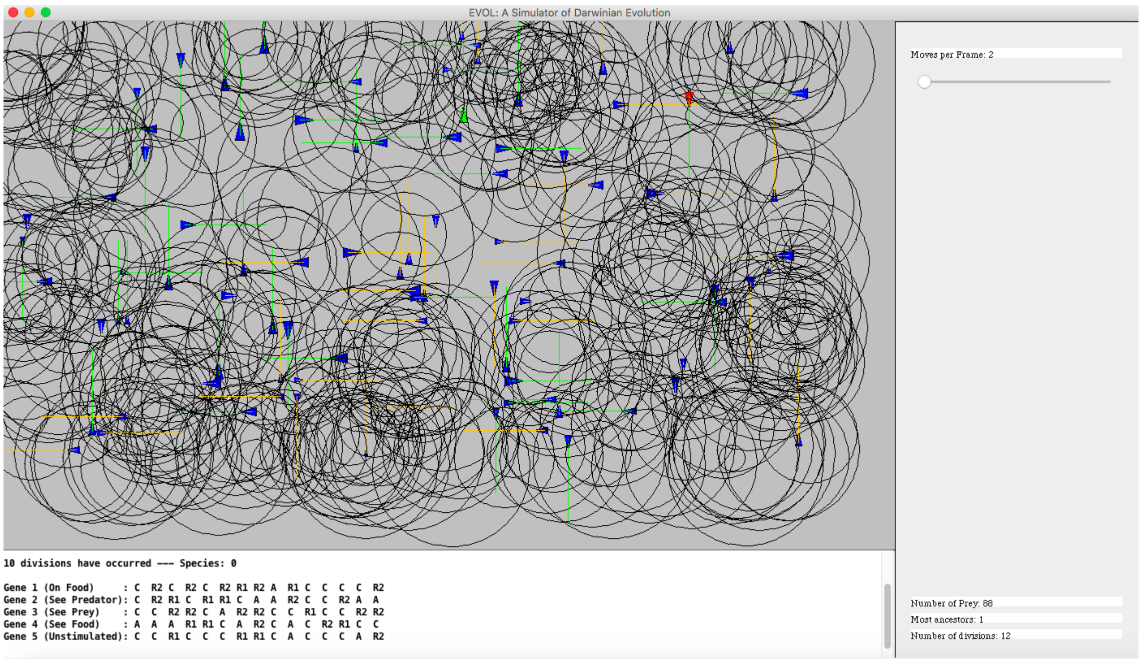
In the 1990's, these methods of mutating the genomes of living organisms were heavily studied by computer scientists interested in artificial intelligence (AI), in an attempt to develop algorithms that worked just like evolution. By developing analogues to genes, chromosomes, and genomes, it became possible to "evolve" a host of intelligences and creatures. Much like natural selection, it is possible to introduce random mutations into an algorithm, providing the same sort of variety in artificial intelligences that already existed in nature.

Our goal in this project is to demonstrate the power of these algorithms on Darwinian creatures. To do this, we are developing a Java applet that simulates an environment containing predators, prey, and immobile food sources for the prey to feed on. If the prey run into a food source, they can choose to feed off of it and grow larger. If a predator runs into a prey, it will immediately consume the prey and grow larger. Since the predators and prey are meant to represent carnivores and herbavores, respectively, the predators cannot eat from the same food sources that the prey eat from.

The predators and the prey both have their own "genomes", governing their reactions to certain stimuli. Figure 2 shows an example genome of one of these creatures. Each creature, once it becomes large enough, splits into four new creatures if it is a prey, and two new creatures if it is a predator. These new creatures will have a slightly different genome from the original parent creature, as a random number of mutations (that depend on a selected mutation rate) can occur in the genome of each child creature.

There exist many methods by which DNA can be changed in a gene pool, some of which do not have biological analogues, but have proved very useful in the study of evolutionary algorithms. At the moment of writing, we have implemented both mutation and transposition as genetic operators on our creatures' genomes. We would therefore be interested in understanding how these genetic operators affect the performance of our prey in the environment.

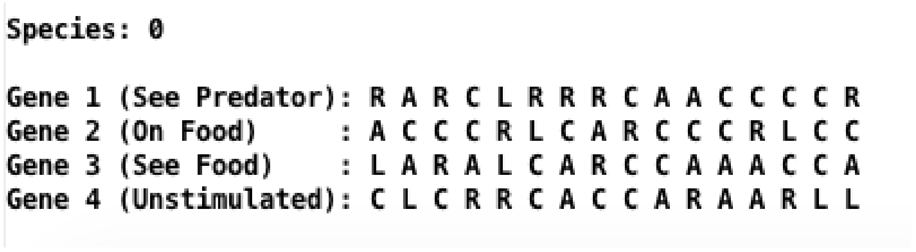
The reason for choosing to follow the development of the prey is that the they are the "middlemen" of our system: for a prey to survive, its genome must be complicated enough to run away from predators, and to run towards and eat food. The performance of these creatures then make a great test for our genetic algorithms.



```
10 divisions have occurred --- Species: 0

Gene 1 (On Food)      : C  R2 C  R2 C  R2 R1 R2 A  R1 C  C  C  C  R2
Gene 2 (See Predator): C  R2 R1 C  R1 R1 C  A  A  R2 C  C  R2 A  A
Gene 3 (See Prey)     : C  C  R2 R2 C  A  R2 R2 C  C  R1 C  C  R2 R2
Gene 4 (See Food)     : A  A  A  R1 R1 C  A  R2 C  A  C  R2 R1 C  C
Gene 5 (Unstimulated): C  C  R1 C  C  C  R1 R1 C  A  C  C  C  A  R2
```

Figure 1: A snapshot of the Java applet's GUI and simulation "world".



```
Species: 0

Gene 1 (See Predator): R A R C L R R R C A A C C C C R
Gene 2 (On Food)     : A C C C R L C A R C C C R L C C
Gene 3 (See Food)    : L A R A L C A R C C A A A C C A
Gene 4 (Unstimulated): C L C R R C A C C A R A A R L L
```

Figure 2: The four-gene genome of a prey creature (species 0).

We are mainly interested in testing the result from literature [3], which states that adding transposition as a genetic operator does not increase the effectiveness of a genetic algorithm. We do this first by determining our creature fitness function as total amount of food eaten in lifetime. We then use a genetic algorithm with transposition and compare its performance in certain key statistics to that of an algorithm without transposition.

To this end, we will begin by providing a brief summary of the genetic algorithm and the architecture of the Java applet we created. This will include detailing important classes and data structures we implemented in developing the applet. We will then discuss the results of our experiments, using several key metrics and graphs to compare our genetic algorithm with and without transposition as a genetic operator. Finally, we will discuss and analyze these results, and ultimately show that our experiments do not conclusively demonstrate that transposition has an effect on our genetic algorithm.

2 Our Genetic Algorithm

Our genetic algorithm is an implementation of "elitist" selection, which is an evolutionary selection algorithm that that works as follows: we create a large initial population of creatures (60 in our case) with random genomes. This is so that we have a lot of randomly-selected starting points in our state space to explore from. We then evaluate each of the creatures and rank them according to their scores, selecting out the top $x\%$ (in our case

$x = 50$) as our most-fit candidates.

The way we score these creatures is using a pre-determined fitness metric. Over the course of development, we tested a few different metrics, including total time to division. However, as our model currently has only one generation of creatures alive at any one time, we decided to remove the division dynamic, and thus this metric does not work. Instead, we replaced the *time to divide* metric with a *total food eaten* metric, which is calculated when a creature dies. Whenever we make a reference to our fitness metric in the following sections, this is the metric we are using.

Once we have scored, ranked, and selected the creatures, we copy the selected creatures directly into the next generation of creatures (effectively letting the "parents" live-on with their children in the next generation). The remaining $(100 - x)\%$ of creatures are created by operating on the genomes of the most-fit parents, to create mutated children. This is where there is some flexibility in the algorithm. As there are a number of genetic operators, we can experiment with different operators to test their effects on the performance of the algorithm and convergence of optimal creatures. The two operators we have implemented in this case are **mutation** and **transposition**.

Considering that our genome is simply a set of instructions (encoded as a bit string), mutation consists of toggling one of these instructions and changing it to another instruction by flipping random bits of the underlying encoding. Mutation can take several parameters, determining the severity and probability of mutations. In our model, we implemented mutation to work as follows: set a fixed constant and call it `MUTATION_RATE`. The creature's genome will be mutated at `MUTATION_RATE` different positions, with a 25% chance that this mutation actually changes the instruction at that position.

Transposition, on the other hand, consists of finding a sequence of instructions within a gene and changing/shifting the location of that sequence within the gene. This is also a common genetic operator in biological evolution. We discuss the implementation of these operators further in our discussion of the `Creature` class.

We can then use these operators to explore our state space of possible genomes. Once we have created our new generation of creatures, we repeat this generational process over and over, preserving certain high-scoring creatures and creating mutants in each new generation.

3 Applet Architecture

To test the performance of our genetic algorithm, we use a Java applet, a snapshot of which is shown in Figure 1. The most important API we use here is the Java Graphics API, which allows us to easily draw geometric shapes and lines onto the applet's canvas. The infrastructure of our system is as follows:

3.1 Java Classes:

- `Evol` - This is the main program driver for our system. It extends the Java `JApplet` class and is where the simulated environment runs from. It handles all the drawing onto the applet's canvas and updates the applet's state at every call to `run()`.

- `GameObject` - This class serves as a general structure for any `Creature` or `Food` object that will be drawn onto the applet and exists in the applet's simulation environment. This holds information such as x- and y-coordinates, and has an abstract `draw()`

function to be overridden by specific `GameObject` subclasses to specify what graphics to paint on the canvas.

- `Controller` - The `Controller` is a vector of relevant `GameObjects`. These will all be drawn onto the canvas of the applet and updated at every call to `run()`. This simplifies our code by having just one call to `draw()` in the applet's `paint()` method.

- `Food` - This represents a food source edible by the prey creatures. It extends `GameObject` and holds a specific food amount, as well as the corresponding diameter of the food source when it's drawn as an oval onto the canvas.

- `Creature` - This class maintains the data structures and methods necessary to hold a `Creature`'s genome, as well as other important predetermined constants that help balance the environment (e.g. food consumption rate, movement speed, size, food growth rate, etc.).

- `MersenneTwister` - This class encapsultes the random number generator (RNG) used for our simulation. We chose to use the Mersenne Twister random number generator as opposed to the default Java random number generator, as the Java RNG implements a linear congruential generator, which has the unfortunate property that it creates hyperplanes when generating random numbers in higher dimensions (more than two or three). To avoid these hyperplanes, we used an open source implementation of the Mersenne Twister for Java, which is equidistributed in up to 623 dimensions, more than enough for our simulator.

- `KeyHandler` - The `KeyHandler` serves to interpret the input made by the user through the keyboard. This responds to different keystrokes and updates a set of booleans as needed when a key is pressed or released. The main functionality we implemented was a pause feature, as well as the ability to print the genome of the current "best" creature in the simulation, and the ability to print some general statistics about the simulation creatures at any given time.

The most complicated of these programs is `Creature`, so we would like to present some more detail on its implementation in the following section.

3.2 Creature Implementation:

Our implementation of the `Creature` class has the following important methods:

- `checkStimulus()` - Updates an integer representation of stimuli currently affecting the creature. The variable `stimuli` is stored as a 4-bit integer, where each of the bits can be toggled on or off according to whether the corresponding stimulus is acting on the creature. The bit-to-stimulus correspondence is as follows:

  Bit 1 toggled = creature sees food
  Bit 2 toggled = prey sees a predator
  Bit 3 toggled = predator sees a prey
  Bit 4 toggled = creature is on a food source

- `move(Rectangle bounds)` - Executes the next instruction in the creature's genome, according to the stimuli acting on the creature and moves the creature accordingly, within the given bounds. The execution of instructions is sequential and wrapped, meaning that if a creature performs the final instruction in a gene, it will restart execution at the start of the gene.

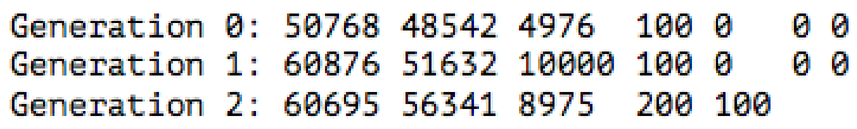
- `mutate()` - Mutates the creature's genome at a random index. We toggle an instruction by XOR-ing with a random number in the range [1, 4]. A mutation has an equal chance of changing any instruction into any new instruction.

- `tranpose(int genenum, String transposon)` - Searches for two instances of transposon within the gene at `geneNum` in the creature's genome. If these are found, `transpose()` moves the sequence between these two instances to another location in the gene, either preceding or following another instance of transposon. Note that the genome is not mutated if the gene does not have at least two instances of transposon. Here is an example of how transposition would occur: Say that we have the gene at `geneNum` is A C R R C A L L A R, and we call `transpose(geneNum, ''A")`. Then, within the method, we would select either A **C R R C** A L L A R or A C R R C A **L L** A R as our sequence to transpose. If we say that we pick the second sequence, our new gene might look like: A C R R C A **L L** A R $\rightarrow$ A C R R C **L L** A A R, transposing the sequence **L L** to before the second "A" instruction, rather than after.

3.3 Important Data Structures

The first and arguably most important data structure we used was the genome of the `Creature` objects. This was implemented as a fixed-length array of integers. Each integer in the array represents a gene, that can be understood as the binary representation of the integer. Because each instruction in a gene can be one of 4 values, (A, C, R, or L), we need 2 bits to encode each instruction, yielding a maximum of $\frac{32}{2} = 16$ instructions per gene and $16 \times 4 = 64$ possible instructions per genome. This is sufficiently large to encode reasonably complex behavior within our creatures, but not too large so as to make the genome overly specific in its instruction sequence and potentially too large a state space to explore.

Another important data structure we maintain is the `generationScores` vector. This is a vector of vectors of integers, which holds the score of every creature that has died while the applet has been running (which we calculate using our fitness function). This data structure gets mutated in the `testObjects()` method in the `Controller` class. In this method, we remove the creature from the controller once it has died, and we append its score to the vector in `generationScores` corresponding to its generation. As we mutate this structure, it might coneptually look like this:



```
Generation 0: 50768 48542 4976  100 0   0 0
Generation 1: 60876 51632 10000 100 0   0 0
Generation 2: 60695 56341 8975  200 100
```

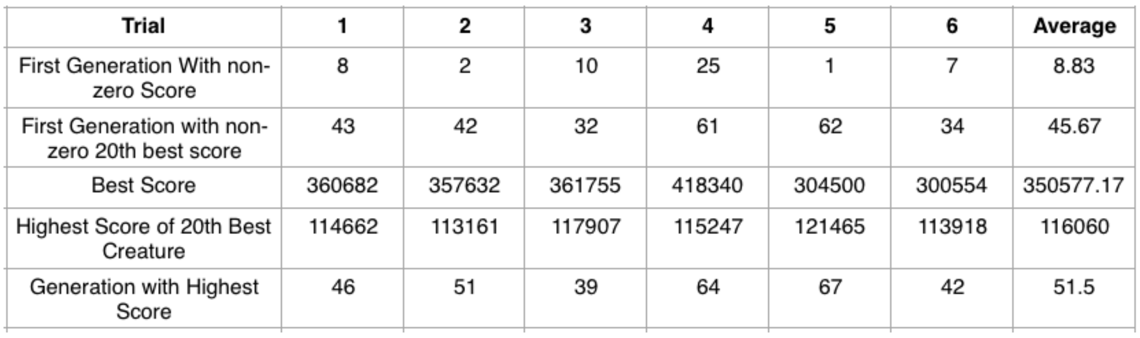
Figure 3: The `generationScores` data structure may look like this while two creatures in Generation 2 are still alive, and thus still unscored.

It is important to note that each vector in the data structure is sorted in descending order and that this invariant is never violated. When a creature dies, its score is appended in the correct position in the vector corresponding to its generation. Another important characteristic of this data structure is that all of its vectors should be of equal length if no generation of creatures is still running. As the size of our generations is fixed, the length of the vector holding the scores of creatures in a single generation should be fixed as well.

Now that we have a better understanding of the important data structures in our program, we can discuss our experiments and results in the following sections.

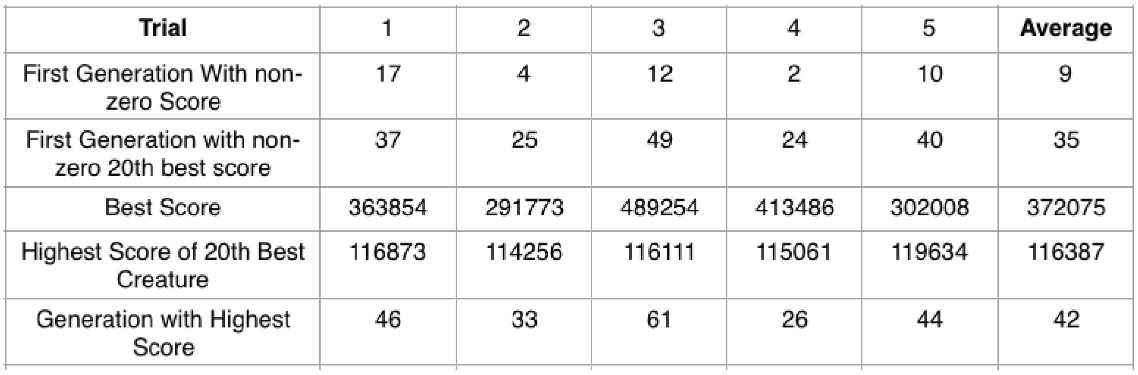
4 Results

To test whether or not transposition had an impact on our specific genetic algorithm, we ran our applet 10 times, 5 trials with transposition and mutation and 5 trials with only mutation. We compared the performances of our algorithms using key statistics, such as convergence rate, first creature with a non-zero score, and highest score achieved by a creature. The graphs and charts are displayed below:



Trial	**1**	**2**	**3**	**4**	**5**	**6**	**Average**
First Generation With non-zero Score	8	2	10	25	1	7	8.83
First Generation with non-zero 20th best score	43	42	32	61	62	34	45.67
Best Score	360682	357632	361755	418340	304500	300554	350577.17
Highest Score of 20th Best Creature	114662	113161	117907	115247	121465	113918	116060
Generation with Highest Score	46	51	39	64	67	42	51.5

Figure 4: Key statistics from the trials run without transposition as a genetic operator.



Trial	1	2	3	4	5	**Average**
First Generation With non-zero Score	17	4	12	2	10	9
First Generation with non-zero 20th best score	37	25	49	24	40	35
Best Score	363854	291773	489254	413486	302008	372075
Highest Score of 20th Best Creature	116873	114256	116111	115061	119634	116387
Generation with Highest Score	46	33	61	26	44	42

Figure 5: Key statistics from the trials run with transposition as a genetic operator.

The tables in Figures 4 and 5 show important metrics for the performance of the best and 33rd-percentile creature. We chose to follow the score of the 33rd-percentile creature, as opposed to the score of the median creature, because its performance is much more stable. We will discuss this in the next two sections.

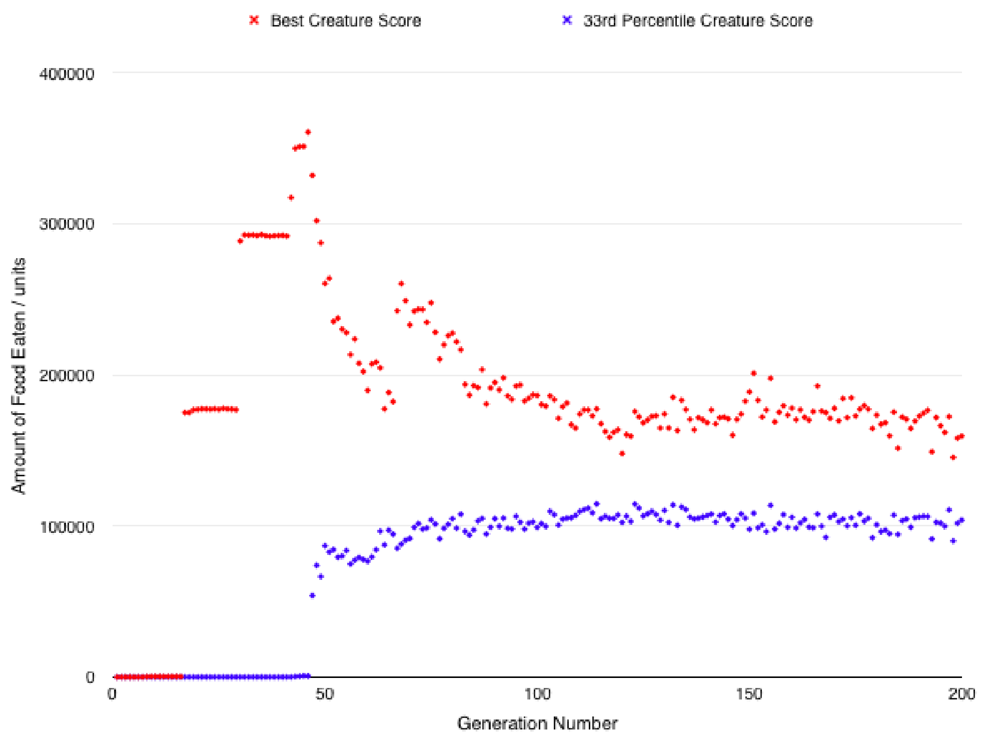


Figure 6: Graph displaying the score of the best creature for each generation (red) and the 20th best creature (blue). This plots the generations of the first trial of our genetic algorithm without transposition.

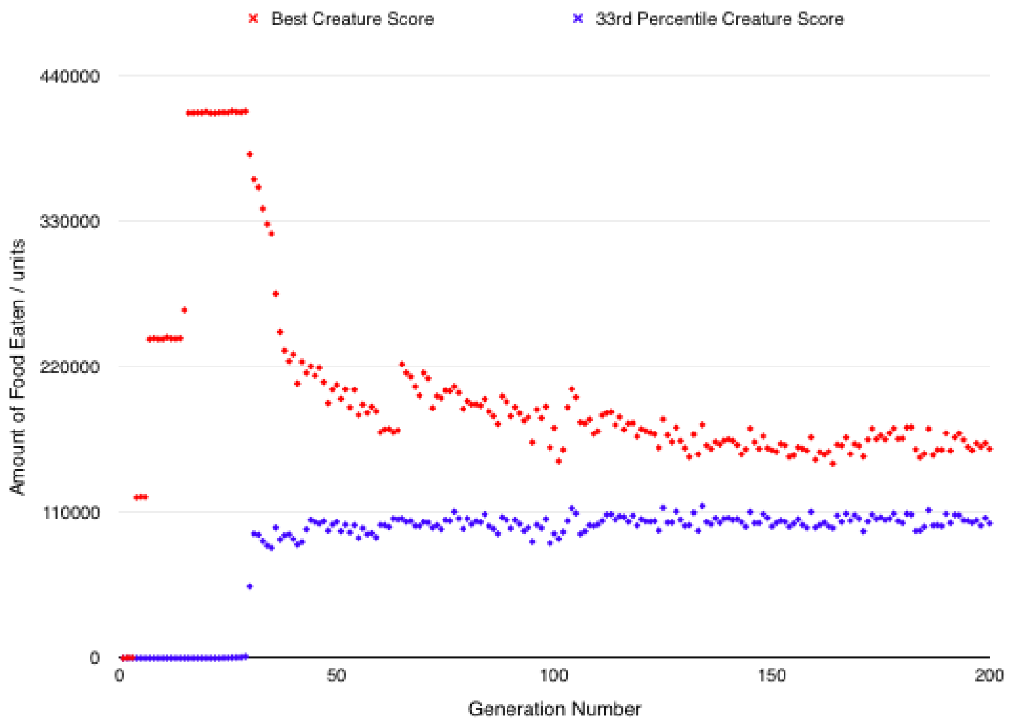


Figure 7: Graph displaying the score of the best creature for each generation (red) and the 20th best creature (blue). This plots the generations of the fourth trial of our genetic algorithm with mutation and transposition genetic operators.

The graphs in Figures 6 and 7 show the scores of the best and 20^{th} best creatures in each generation. The data used to create these graphs was taken from the trials that were closest to the average key statistics for each algorithm.

5 Analysis and Discussion

Before we begin our analysis, we may mention the choice to follow the metrics of the 20th best creature out of 60, rather than the median creature (the 30th). The reason for this choice is that we chose our elitist selection parameter to have a value of 0.5, meaning that we keep the top 50% of creatures from each generation and send them unmodified into our next generation. This leads to potentially erratic behvaior of the median creature, as occasionally, this creature will actually perform much worse in a following generation than it did previously.

Thus, though the performance curves of the median creature would likely look very similar to the performance curve of the 20th best creature in Figures 6, and 7, the outliers in the data would make our analysis of the two algorithms much more difficult and imprecise.

So, as we can see from the data we collected, the algorithm with transposition had an average highest score 21497.83 higher than the average without transposition, which amounts to a 5.7% difference. This is not large enough of a difference to conclude that transposition definitely impacted the overall mutation algorithm. We can look to the other statistics to see if we notice a significant difference in any one of them instead.

We see that the average generation number at which the algorithm with transposition reached its peak score was 9.5 generations earlier than the algorithm without transposition. Similarly, the algorithm with transposition produced a 20th best creature with a non-zero score 10 generations earlier than the algorithm without transposition. However, we also note that the first creature to achieve a non-zero score occurred on average in the 9th generation in both algorithms (if we round the generation number to the nearest whole number for the algorithm with transposition. We also notice that the algorithm with transposition only provided a 327 unit increase in the best score of the 20th creature, equivalent to a 0.2% increase in performance.

If we observe our data from the graph, we note that both algorithms behave extremely similarly. There are 3 clear "jumps" in performance before reaching the best creature reaches its highest score. This behavior is actually rather common of genetic algorithms, as it relates to the fitness gain associated with randomly acquiring a very beneficial behavior in the creature (e.g., the ability to avoid predators). This beneficial mutation reflect a major change in the creatures phenotype, as well as its genotype. Once the best creature reaches its highest peak in performance, the best creature decreases smoothly as the score of the 20th best creature quickly increases (a byproduct of competition amongst creatures able to efficiently find and consume food). We conjecture that the score of these two creatures will not converge to each other over time, but more-extensive tests would need to be run to determine whether or not this is truly the case.

6 Conclusion

The genetic algorithm with transposition seems to slightly outperform the genetic algorithm without transposition, if we judge our algorithms with the key metrics in Figures 6 and 7. This could imply that transposition actually added power to our algorithm, making it converge more quickly than without transposition.

However, with the data we collected, this is a very difficult position to maintain. The differences we found in these metrics could be due to random chance, which plays a very large role in genetic algorithms in general. The variation in "generation with first non-zero score", for example, is rather large in both tables. This large variation would also call for a much larger number of tests. We ran 5 and 6 tests for our algorithms with and without transposition respectively, but for such a large range in each metric, we would most likely need to run on the order of hundreds of tests to get a more accurate result. Unfortunately, time did not permit us to run such an extensive test suite, so we had to rely on the results we observed from the tests we were able to finish completely in the allotted time slot.

The most interesting conclusion can be drawn from the graphs in the Discussion section. These show that the shape of the graph remains the same with and without transposition as a genetic operator. The shape of the graph can be explained by competition for resources by the creatures. The score of the best creature first increases in jumps, corresponding to crucial mutations in its genome that give it great benefits over other creatures. Then, as the other creatures in the population begin to develop, the competition for food continues to grow, bringing down the performance of the best creature and bringing up the performance of the upper-quartile creatures. It is important to note, however, that although the score of the best creature decreases, this does *not* imply that the best creature actually has a worse genotype than it did previously. In fact, the best creature of generation 200 most likely has a more fit genome than the best creature of generation 60, because it is competing more fiercely for food and still doing better than all the other creatures in its generation.

Thus, the graphs in Figures 6 ad 7 actually show the very interesting effect of competition for resources on the performance of our genetic algorithm, an unexpected conclusion to our experiments. It would be exciting to explore this effect more in future work.

References

[1] Hodjat, Babak and Hormoz, Shahrzad and Miikkulainen, Risto. *Estimating the Advantage of Age-Layering in Evolutionary Algorithms*. Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2016, Denver, CO). 2016. "http://nn.cs.utexas.edu/?shahrzad:gecco16"

[2] Nawa, N., Hashiyama, T. , Furuhashi, T. and Uchikawa, Y.: A Study on Fuzzy Rules Using Pseudo-Bacterial Genetic Algorithm with Adaptive Operator. Proceedings of the IEEE International Conference on Evolutionary Computation, 589- 593. IEEE Press 1997.

[3] Simões, A. and Costa, E.: Transposition: A Biologically Inspired Mechanism to Use with Genetic Algorithms. Proceedings of the Fourth Int. Conference on Neural Networks and Genetic Algorithms, pp. 178-186. Springer- Verlag 1999

[4] Voss, M. and Foley, C.: Evolutionary Algorithm For Structural Optimization. Banzhaf, W., Daida, J., Eiben, A. E., Garzon, M. H., Honavar, V., Jakiela, M., and Smith, R. E. (eds.), Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'99), pp. 678-685, Orlando, Florida USA, CA: Morgan Kaufmann 1999.