Project 1: Optimization of Ant Foraging and Reproduction Using an Evolutionary Genetic Algorithm

Abstract
A genetic algorithm was implemented that focused on a specific ant population based on their ability to detect sugar over salt, the distance they can travel, and effectively reproduce. The experiment focused on a population of 20 ants with random genomes consisting of 100 characters representing 6 different genes and the change of their fitness over time. Functions that mimic genetic mutations and sexual reproduction were applied for 20 generations. Results indicated that beneficial traits, such as sensitivity to sugar, became more dominant while less beneficial traits, such as sensitivity to salt, became less frequent. This study demonstrates the effectiveness of evolutionary algorithms in optimizing traits for simulated biological evolution.

Experiment Description
The simulation used in this experiment focuses on an ant colony that begins with 20 individual ants. To create the next generation, functions that implement genetic mutation and sexual reproduction are used on those individuals to create new genomes. Each generation consists of 20 ants, with each ant essentially creating its replacement through sexual reproduction (2 parents always create 2 children).
In this experiment, 6 traits were selected. Utilizing the characters of *S* for sensitivity to sugar, *T* for sensitivity to salt, *D* for ability to travel 50 yards from the colony, *H* for ability to travel 20 yards from the colony, *F* for increased fertility, and *A* for average fertility. The genomes are initially randomly assigned from these 6 base genes, and are 100 characters long. In order to calculate the fitness of an individual, there is a function containing a for loop that iterates through the entire genome and if statements that affect the fitness value based on the character at *genome[i]*. For the characters *S, D,* and *F* the fitness increases by 1. For the characters *T* and *H* the fitness decreases by 1. For the character *A,* the fitness is not changed.
The evolutionary process utilizes a tournament style selection, uniform crossover, and mutation. Tournament style selection selects ants with the highest fitness to reproduce out of a set of 6 random candidates. Uniform crossover is created in a function where given the genome of 2 parents, both are iterated through. Depending on a random choice for the value 1 or 0, the genome of the child either takes a gene at a given index from parent 1 or parent 2. The resulting genome is a 100 character array that contains genes from both parents, with a random change of which gene has been passed on. Mutation is performed by taking a random index from within the genome of the given ant, and randomly swapping that gene with another gene. This is done for every ant, performed after the tournament selection but before the uniform crossover.
The hypothesis for this experiment is that over multiple generations, the fitness of the fittest individual will increase by having the genes that increase fitness increase in frequency and genes that decrease fitness will decrease in frequency.

Algorithm Descriptions
Genetic Representation

The genetic representation of the ants is an array of characters that consists of 100 characters that are meant to resemble genes within a genome. Each ant only has 1 genome. To create a genome, an empty array of length 100 is created. The array *bases* is defined with the six traits (*S, T, D, H, F, A*). From *bases*, for each index of the empty array *genome* the index is assigned a random value from *bases*. This results in an array 100 characters long that contains a random assortment of the characters *S, T, D, H, F, A*. This genome and an ant's base fitness (0), are defined at the beginning of the *Ant* class and within the function *__init__* which defines the attributes of the object *self*.

Calculate Fitness Function

The fitness of the ants is what is being measured in the experiment. In order to calculate that fitness, the function *calcFitness* is used. Within this function, the fitness of any given ant is assigned to 0. Then, the function goes through the array of the genome and calculates the fitness based on the traits within the genome. As defined above, the characters *S, D,* and *F* increase the fitness, characters *T* and *H* decrease the fitness, and character *A* does not affect the fitness. Utilizing if statements, the fitness is changed as the genome array is iterated through using a for loop depending on the character of a given index *i*.

Initial Generation of the Population

The initial generation is completely random. These ants are the only ones to have 100 character genomes that are randomly selected from the *bases* array.

Tournament Selection

Tournament selection is done by taking 6 random ants from the generation and returning the ant that has the highest fitness. The function that performs this is *parent_select* and is only called within the function that creates a new generation, *generation*.

Uniform Crossover

Uniform crossover is performed in the method *uniCross*. This method takes 2 parent ants, and for the length of the first parent's genome (*self*), a for loop iterates through all indexes. While iterating through this loop, there is a 50% chance for every index that it will be replaced by the value at the same index in the genome of the second parent (*other*). A new array is not created, instead the same ant object is being utilized and changed. This function is only called within the function *generation*.

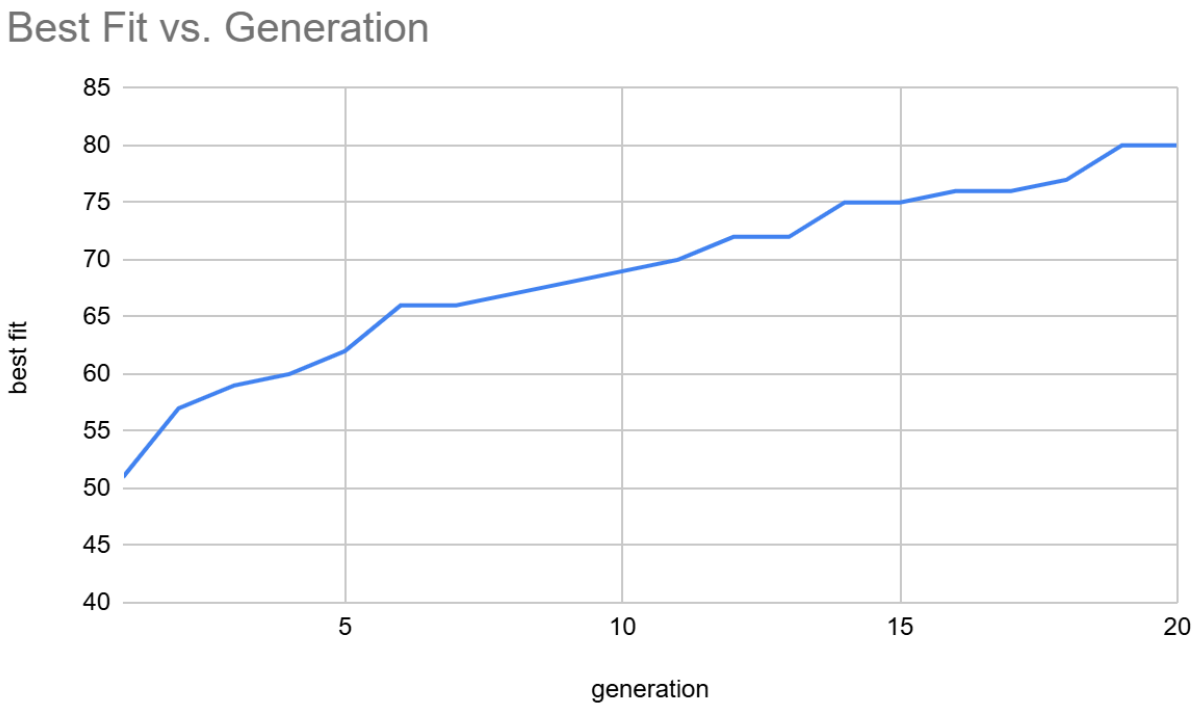
Mutation Method and Rate

The mutation method and rate for this experiment are consistent through all generations. The rate of mutation is 1 mutated gene per every individual per generation. To achieve this, the method *mutate* takes an ant and chooses a random number between 0 and 99 that represents an index in the ant’s genome. The function then assigns the value at that index to a random value from the array *bases*, therefore most likely changing the gene (an 83.3% chance that the gene will be changed).

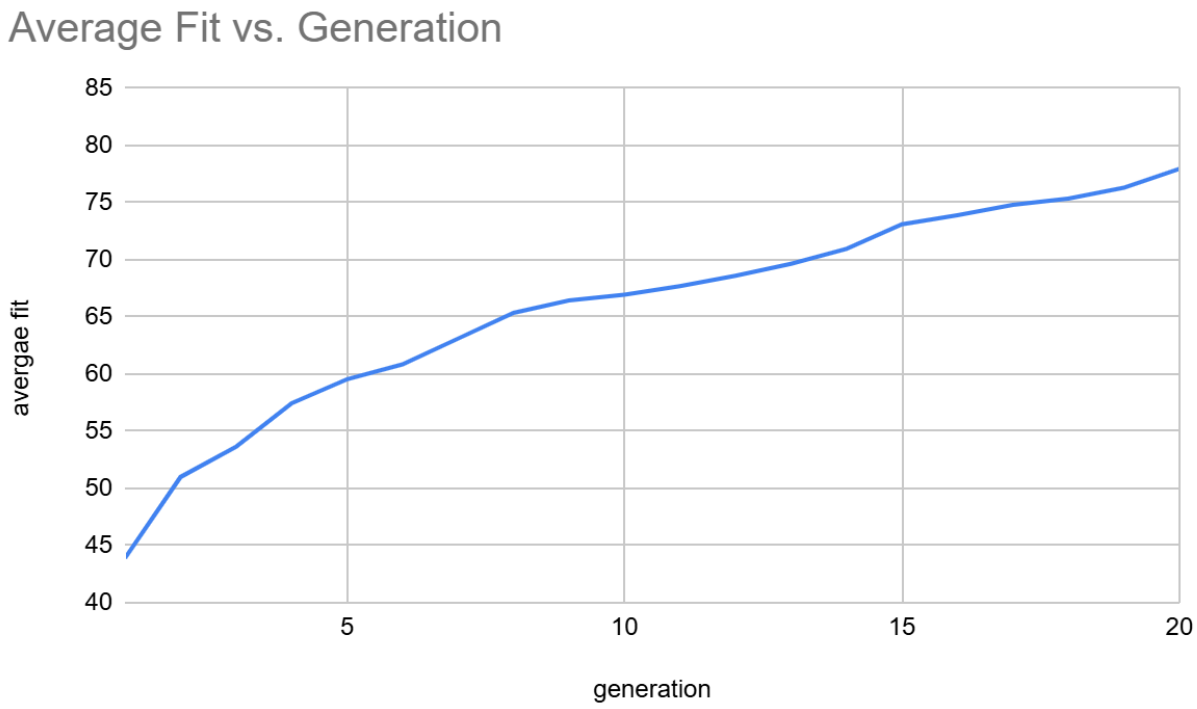
Evolution Process

This evolution occurs over generations, in this experiment over 20 generations. In order to achieve this, there is a function called *generation*. Within this function, a new population is created and initialized as an empty array. Since the population size has already been declared as 20 individuals, a for loop is used to create 20 new individuals but has a range of 10 since in each iteration of the loop, 2 parents create 2 offspring. These 2 parents undergo tournament selection with the *parent_select* function. These parents are then mutated, and perform uniform crossover. The results of the uniform crossover are then added to the new population. Once the population has been established, 20 individuals have been defined, the new population is assigned to the same variable as the old population and therefore replacing it. From this new population, the function *calcStats* is called to find the value of highest fitness and the average fitness of the population.

Results



This graph represents the highest fitness value for each generation. The lowest fitness is 51 in generation 1, and the highest fitness is 80 in generations 19 and 20. With this graph, it is shown that the highest fitness value is increasing over time meaning that the frequencies of the genes that increase fitness are increasing as well.



This graph represents the average fitness values for each generation. This graph has less steep increases, and shows a more gradual increase across generations. The average fitness also shows the increased frequency of the genes that positively impact fitness.

Based on these results, we can see that as the generations progressed both the highest fitness of an individual in the generation increased as well as the average fitness of a generation.

Conclusion

From the results shown above, the genetic algorithm clearly selected traits that are beneficial to the ant population over 20 generations. Genomes contain the characters of *S, D,* and *F* rather than the characters *T, H,* and *A* that do not positively affect the fitness of an individual. Genetic diversity was maintained in this simulation due to the functions that implements mutation and uniform crossover.

In future research, there are multiple avenues to explore with this algorithm. A selection strategy other than tournament could be introduced which may or may not impact the highest individual fitness of a generation. The mutation rate could be increased or decreased and the impact on both the average fitness and highest individual fitness of generations could be measured. With this algorithm, it would be easiest to introduce more survival factors and genes that would increase or decrease the fitness of ants.

Appendix

```
from IPython.display import HTML
shell = get_ipython()
```

```
def adjust_font_size():
  display(HTML(''' <style>
    body {
      font_size: 32px;
    }
    '''
    ))
if adjust_font_size not in shell.events.callbacks['pre_execute']:
  shell.events.register('pre_execute', adjust_font_size)
import random
bases = ['S', 'T', 'D', 'H', 'F', 'A']
# S = sensitivity to smell of sugar
# T = sensitivity to smell of salt
# D = ant can travel 50yds from colony
# H = ant can travel 20yds from colony
# F = increased fertility
# A = average fertility
class Ant:
    def __init__(self):
        self.fitness = 0
        self.genome = []
        for i in range(100):  #creating genomes for all ants
          self.genome.append(random.choice(bases))
        #print("self.genome = ", self.genome)

    def mutate(self):
        index = random.randint(0, len(self.genome) - 1)
        self.genome[index] = random.choice(bases) #creating mutations
within genes

    def uniCross(self, other):
      #uniform crossover
        for i in range(len(self.genome)):
            if random.random() < 0.5:
                self.genome[i], other.genome[i] = other.genome[i],
self.genome[i]

    def calcFitness(self):
      #calculating the fitness
```

```
      self.fitness = 0
      for i in range(0, len(self.genome)):
        if self.genome[i] == 'S':
          self.fitness += 1
        elif self.genome[i] == 'D':
          self.fitness += 1
        elif self.genome[i] == 'F':
          self.fitness += 1
        elif self.genome[i] == 'T':
          self.fitness -= 1`


    def copy(self):
      #copies ant genome and fitness, then returns the copy
        clone = Ant()
        clone.genome = self.genome[:]
        clone.fitness = self.fitness
        return clone

    def __str__(self):
        return f"Fitness: {self.fitness}"


popSize = 20

class Population:
    def __init__(self):
        self.population = [Ant() for _ in range(popSize)] #creating ants
for the population
        self.calcStats()

    def calcStats(self):
        #update the population fitness
        for ant in self.population:
            ant.calcFitness()
        self.best = max(self.population, key=lambda a: a.fitness) #finding
best fitness from the set of self.population
        self.avgFit = sum(a.fitness for a in self.population) / popSize
#calculating average fitness all in one line
```

```
    def parent_select(self):
        #tournament style selection
        candidates = random.sample(self.population, 6) #select 6 random
candidates
        return max(candidates, key=lambda a: a.fitness) #return the
highest fitness of candidates, key=lambda a: a.fitness compares the
candidates' fitness and returns the highest fitness which then that ant is
used as the parent for unicross

    def generation(self):
        #next population being declared, crossed, mutated, and assigned to
self.population
        new_population = [] #empty array for the population
        for _ in range(popSize // 2): #dividing for whole number
            p1 = self.parent_select().copy() #selection of genes
            p2 = self.parent_select().copy() #selection of genes
            p1.mutate() #mutating genes of p1
            p2.mutate() #mutating genes of p2
            p1.uniCross(p2) #mixing those genes to new offspring
            new_population.extend([p1, p2]) #appending both at same time
        self.population = new_population #assigning the current population
the values of new_population
        self.calcStats() #calculating stats of the population

    def print_population(self):
        #for ant in self.population:
            #print(ant)
        print("Best Fitness: ", self.best.fitness)



# Running the Genetic Algorithm
pop = Population()
generations = 20

for gen in range(generations):
    print("Generation: ", (gen+1))
    pop.generation()
    pop.print_population()
```