

Nature of Code

Patrick Dwyer

Fall 2005

Week 9 - November 1st

Genetic Algorithms

In many of our examples, we've made objects with various behaviors and characteristics; virtual plants, autonomous steering agents, attractive bodies, etc. In a lot of cases we've picked pseudo-random numbers or created custom random numbers in order to initialize these properties as well as assigned simple rules to determine motion behaviors. What if we could encode the properties and behaviors of our objects into DNA-like structures and simulate the elements of biological adaptation - *variation*, *heredity*, and *selection* - in order to evolve the properties and behaviors of our objects? In doing so we would add a new dimension of simulation to any of the topics we've covered so far; Evolution.

A genetic algorithm refers to the process of evolving populations of DNA-like data (encoded as a series of data), and the methods and techniques for pairing, mating, mutating and testing the populations of data. Much like life on Earth evolves and grows from instructions coded in our DNA, we can create a system that grows and evolves according to rules and expressions that we choose. To do so we need to determine what we want our population to do, and how we want our pseudo-DNA to control it.

Setup

Step 1: The Population

- Create an empty population (either an array or an ArrayList)
- Fill the population with DNA encoded objects (pick random values to start)

Loop (repeat loop to continually evolve population)

Step 1: Selection

- Create an empty mating pool (an ArrayList)
- For every member of the population, evaluate the members fitness based upon some criteria or function, and add the member to the mating pool in a manner proportional to its fitness. i.e. The more fit the member, the more times it appears in the mating pool, and the more likely it is to later be picked for reproduction

Step 2: Reproduction

- Create a new empty population
- Fill in the new population by using the following reproduction steps:
 1. Pick two "parent" objects from the mating pool
 2. Crossover (mating): create a "child" object by mating the two parent objects
 3. Mutation: mutate the child's DNA based upon a given probability
 4. Add the child to the new population

- Replace the old population with the new population and continue looping...

Strictly speaking, a genetic algorithm involves the use of virtual DNA encoded as bits and follows a series of precise steps. In this week's first example, we will attempt to adhere as closely as possible to the strict definition of a genetic algorithm. However, we will employ some minor shortcuts. As we continue with further examples, we may also find it more convenient to allow ourselves some latitude. For example, we might encode our DNA as an array of floating point values (instead of bits), and use custom random number distributions to mutate (as opposed to flipping bits). In addition, we might expand our concept of selection, creating systems where objects have built-in life-spans, mate based on an intersection test, etc.

Evolving Text

Our initial exploration of Genetic Algorithms will involve objects trying to mutate themselves to match a string of text. Visually this example is fairly boring, but it explicitly outlines the techniques of Genetic Algorithms, and gives us a visual means to understand what really goes on inside our evolving system. We will create a population of random phrases, as well as a target phrase. Each random phrase's fitness will be evaluated according to how close it resembles the target phrase. As the phrases continue to "evolve", they will eventually reach the target sequence of characters.

What's interesting about this example is it proves the power of genetic algorithms over brute force algorithms (sequentially trying every possibility). To randomly arrive at a the phrase "More is different" would take unfathomable amounts of time¹, yet genetics allows us to get there potentially within seconds.

Nevertheless, this case is not terribly interesting simply because the "answer" is known. We could easily arrive at the target phrase because, well, we already know the target phrase! More interesting examples (several of which are described in Chapter 20 of *Computation Beauty of Nature*) involve using the evolutionary process to find solutions in seemingly unknowable situations (where the variables of the system are too numerous to use conventional methods).

Coding Our Genetic Algorithm

The first thing we need to do is encode our virtual DNA. For this we create a DNA class - it contains an array of characters (the "DNA" or "genotype"). We have two constructors; one that fills that new DNA sequence with random characters and one that fills it based upon an existing array.

```
class DNA {

    // Our genetic sequence
    char[] dna;

    // Create a new random series of DNA with a given length
    public DNA(int chars) {

        // create the dna array
        dna = new char[chars];
    }
}
```

¹ Actually, to brute-force this phrase would potentially require checking 87112285931760246646623899502532662132736 possible letter combinations. If the computer could check one every nanosecond (one billionth of a second) it would take 2762312466126339632376455 years to try all the combinations.

```

        // fill the dna with random values
        for (int i = 0; i < chars; i++) {
            dna[i] = (char)random(0, 255);
        }
    }

    // Create a new DNA sequence from an existing array of characters
    public DNA(char[] basedna) {

        // just copy the dna sequence
        dna = (char[])basedna.clone();

    }

```

Next we need a method that calculates the fitness of this DNA sequence. Here, the fitness is a value between 0 and 1.0, the percentage of "correct" characters:

```

    public float getFitness(String target) {

        int score = 0;

        // Check each character, and record how many match the target string
        for (int i = 0; i < dna.length; i++) {
            if (dna[i] == target.charAt(i)) {
                score++;
            }
        }

        // our fitness is simply the ratio of matched characters to the total
        // number of characters in the string
        float fitness = (float)score / (float)target.length();
        return fitness;
    }

```

Finally, we need two methods, one for "crossover" (a method that takes a given object's current DNA sequence and combines it with another's), and one for "mutation" (a method that mutates given characters in the sequence randomly, according to some probability).

```

    public DNA mate(DNA partner) {
        // create the array for the new child dna
        char[] childDNA = new char[dna.length];

        // choose a crossover point
        int crossover = int(random(dna.length));

```

```

// now copy the dna from the appropriate parent
for (int i = 0; i < childDNA.length; i++) {
    if (i > crossover) {
        childDNA[i] = dna[i];
    } else {
        childDNA[i] = partner.getDNAat(i);
    }
}
// create the new DNA object
DNA newDNA = new DNA(childDNA);
return newDNA;
}

// Mutate our DNA based upon the probability of any given character in
// the DNA changing.
public void mutate(float chance) {
    for (int i = 0; i < dna.length; i++) {
        if (random(1) < chance) {
            dna[i] = (char)random(0, 255);
        }
    }
}
}

```

Once our DNA class is complete, we can create a population class and implement the steps described above for simulating evolution. Our “main” program is quite simple, calling methods on the population class to control the evolution. The actual code of our main program is a bit more complex, adding a visual element so we can keep track of the progress of our evolutionary population.

```

Population populus;
void setup() {
    targetPhrase = "More is different";
    populationSize = 30;
    mutationRate = 0.01f;
    populus = new Population(targetPhrase, mutationRate, populationSize);
}

void draw() {
    populus.simulate();
}

```

The full code for the population class is available in the example. Let's examine a couple pieces to get some insight into how our evolutionary system is working. Each population object has the following instance variables:

```

// The maximum number of organisms in any generation
int maxOrganisms;

```

```

// The rate at which the organisms mutate
float mutationRate;

// The array holding our current population of organisms
DNA[] population;

// The fitness score of each organism in the population array
float[] fitness;

// The mating pool we use when creating a new generation
ArrayList geneticPool;

// The phrase the organisms are trying to evolve to match
String targetPhrase;

// Total number of generations so far
int generations;

// Has the population reached 100% fitness (matched the targetPhrase)
boolean finished;

```

The crucial two steps in our population class are creating and filling the mating pool, as well as creating the next generation based on that mating pool. Each member of the population has a fitness stored in an array of identical size (filled in the `calcFitness` method.) In order to create our weighted mating pool (with high fitness DNA more likely to be picked for mating), we must calculate the fitness normal for each DNA sequence, using that value to calculate the number of times each object should appear in the mating pool (we use an `ArrayList` data structure to manage the pool.) To accomplish this we multiply the fitness normal by 10000. For example, a fitness normal of 0.003 would yield 30 entries into the mating pool, 0.0001 would yield 1 entry.

```

public void naturalSelection() {

    // clear the gene pool
    geneticPool.clear();

    // get the total fitness of the whole population
    float totalFitness = getTotalFitness();

    // For each DNA object calculate it's relative fitness to the whole
    // group.
    for (int i = 0; i < population.length; i++) {

        // get the relative fitness
        float fitnessNormal = fitness[i] / totalFitness;
    }
}

```

```

// figure out how many entries to the gene pool this object gets
int poolEntries = (int)(fitnessNormal * 10000.0f);

// add the object to the gene pool the appropriate number of
times
for (int j = 0; j < poolEntries; j++) {
    geneticPool.add(population[i]);
}
}
}

```

Once the mating pool is complete, the next step is easy. We refill our main population array by picking two parents and using the crossover and mutation methods written into our DNA class.

```

public void generate() {

    for (int i = 0; i < population.length; i++) {

        // pick two random candidates from the gene pool
        int m = (int)random(geneticPool.size());
        int d = (int)random(geneticPool.size());

        // get the parent DNA object
        DNA mom = (DNA)geneticPool.get(m);
        DNA dad = (DNA)geneticPool.get(d);

        // create the new DNA object
        DNA child = mom.mate(dad);

        // mutation
        child.mutate(mutationRate);

        // add to the current population
        population[i] = child;
    }

    generations++;
}
}

```

Resources

- [The Computational Beauty of Nature, Chapter 20 - Genetics and Evolution](#)
- Craig Reynolds; <http://www.red3d.com/cwr/evolve.html>