CS369 Computational Science
Evolutionary Algorithms
Matthew Egbert 2017

Introduction / Evolution Refresher

Evolution occurs when…

- there is a **population** of individuals
- each individual **dies**
- each individual can **reproduce**
- there is **heritable variation** in the population, meaning:
    - some individuals are **more fit** i.e. more likely to reproduce than other (variety)
    - a significant portion of the "traits" that contribute to fitness are passed from one generation to the next (heritability)

Over successive generations, the make-up of the population changes. Without the need for any individual to change, the population **adapts** to its environment. A shorthand way of thinking about this is that "fitness increases," but this is not strictly true and it glosses over some details.

We can distinguish **Natural Evolution** from **Artificial Evolution**.

- In Natural Evolution, **fitness** is an abstraction of a wide variety of factors including phenotypic traits, environment, luck, other organisms (conspecifics or predators or prey or competitors), etc.
- In Artificial Evolution, **fitness** is determined artificially (i.e. by a person).
    - breeding is an example of artificial evolution, where people select certain organisms for their desirable (sweeter fruit, greater yield grain, stronger beasts of burden etc.), and improve their relative reproduction rates
    - A more recent addition is **evolutionary algorithms (a.k.a. genetic algorithms)**, where the key evolutionary dynamics (mutation, selection, reproduction, etc.) are simulated so as to evolve a desired system...
        - a class schedule that minimizes conflicts
        - the shape of an airplane wing
        - reaction rates to make a model of metabolic dynamics match empirical data
        - just about anything you can think of! (although in practice it is often not so easy)

We will now look at the basics of genetic algorithms. Including a variety of ways to construct them. Along the way, we will briefly consider (i) why it can be difficult for artificial evolution to find a solution, and which situations lead to better or worse chances of success; and (ii) what methods can be used to improve genetic algorithms so that they (a) are more likely to find high quality "solutions" and (b) find those solution more quickly?

A First Evolutionary Algorithm

To start, let's look at a first evolutionary algorithm... one for designing paper airplanes.

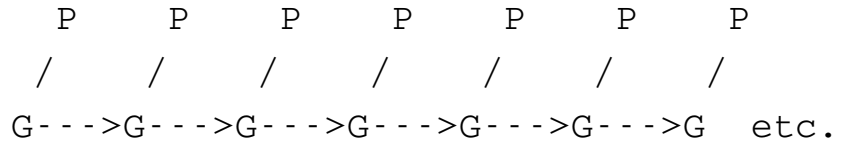
1. Generate 20 random sequences of folding instructions
2. Fold each piece of paper according to instructions written on them
3. Throw them all out of the window
4. Pick up the ones that went furthest, look at the instructions
5. Produce 20 new pieces of paper, writing on each bits of sequences from parent pieces of paper (occasionally make a mistake while copying the instructions)
6. Repeat from (2) on.

Q: What is **fitness** in this artificial evolution?

Q: What process(es) maintain the presence of **variety**, and what would happen if these process(es) were not included? **A:** the variation in a population can be lost if there is no process maintaining it – mutation is the most frequently considered process through which variety is maintained.

Some terminology:

- The **genotype** is what is copied from one generation to the next. Genotypes can be copied, mutated, and recombined. **Q:** What is the genotype in the above EA?
- **Recombination** is the mixing of parent genotypes to produce an offspring's genotype
- The **phenotype** of an individual is what undergoes selection (can be evaluated for fitness). **Q:** What is the phenotype in the above EA?



Instead of paper airplanes, you could be evolving for optimal timetables. The genotype might be an encoding of allocations; the phenotype the schedule itself, and fitness the negative number of clashes. Or if you were evolving optimal aircraft wing design, the genotype would specify various wing dimensions, the phenotype the actual wing (or a simulation thereof..) and the fitness would be (say) some function of the lift, drag, and manufacturing cost of the wing.

Let's look now at some pseudocode for a simple GA.

```
POP_SIZE = 30
N_GENES  = 10
N_GENERATIONS = 100

parents[POP_SIZE,N_GENES];
offspring[POP_SIZE,N_GENES];

## INITIALIZE THE POPULATION
for i in range(0,POP_SIZE) :
    for j in range(0,N_GENES) :
        parents[i,j] = appropriate_random_value()

## EVOLVE. Each time around the outer loop is one "generation"
for generation_i in range(0,N_GENERATIONS) :

    ## evaluate the fitness of every individual
    fitnesses = [eval_fitness(indiv) for indiv in parents]

    for offspring_i in range(0,N_GENES) :
        ## randomly pick 2 individuals from parents
        ## selecting preferentially for the fitter ones
        parent_a, parent_b = biased_random_selection_of_two_parents()

        ## recombine to make 1 offspring, and mutate the offspring
        offspring[offspring_i] = mutate( recombine(parent_a,parent_b) )

    ## overwrite the parents with the offspring, (throwing
    ## away the previous generation in the process)
    parents[:] = offspring[:]
```

Genotypes in genetic algorithms

The most common genotype formats in genetic algorithms are

- A **sequence of symbols from a finite alphabet.** This kind of genotype is most directly comparable to the nucleotides (the G's C's A's and T's) of real DNA, but often is EA is a binary string (a sequence of 0s and 1s).
- A **sequence of real values** e.g.: [0.0151, 0.4, 0.942…]. These values are often constrained to lie in a certain range, e.g. between 0 and 1, but they don't always have to be.
- A **tree.** These are the non-linear data structures, I am sure you are all familiar with. A value of some kind (discrete or continuous) is generally associated with each node in the tree, and each of these nodes would correspond to a single "gene."

Mutation

Recall that it must be possible to copy, mutate, and recombine genotypes. The different genotype formats generally require different forms of mutation.

For **binary encodings** a good (very approximate) rule of thumb is to have 1 mutation per bit. So each time you generate an offspring, each bit has a 1/N chance of being flipped where N is the number of binary genes. This is better than picking one gene and flipping it – why?

For **real encodings** the approach is often quite different form of “creep mutation,” where you mutate every gene by a small amount. I use a value selected from a Gaussian distribution with a standard deviation of 2% of the total allowed range of the value.

This creep mutation can cause genes to leave their allowed range. Three possible solutions are the following (presuming an allowed gene value between 0 and 1):

- Clamping 1.06 → 1
- Bouncing 1.06 → 0.94
- Wrapping 1.06 → 0.06

Which of these is best to use can depend upon the problem at hand, but generally bouncing and wrapping are seen as better than clamping, as clamping increases the number of mutations that produce gene values at the limits, biasing the mutation. **Q:** Wrapping can also have negative effects, can you imagine what they might be?

Crossover

In the pseudocode, we included a "magic function" `recombine(parent_a,parent_b)`, which takes the genomes of the two parents and combines them to produce an offspring. In some organisms this recombination involves a process called **crossover,** where the gene sequences of the two parents are interspersed. This can be emulated in a genetic algorithm in a variety of ways. The most common (for linear genotypes) are:

- **one-point crossover:** randomly select a point along the genotype. Before this point the genes come from parent_a, after this point the genes come from parent_b.
- **n-point crossover:** same as above, but chose more than one point, having the gene source switch between parents at each point
- **uniform crossover:** at each loci (gene), select the parent with 50/50 odds

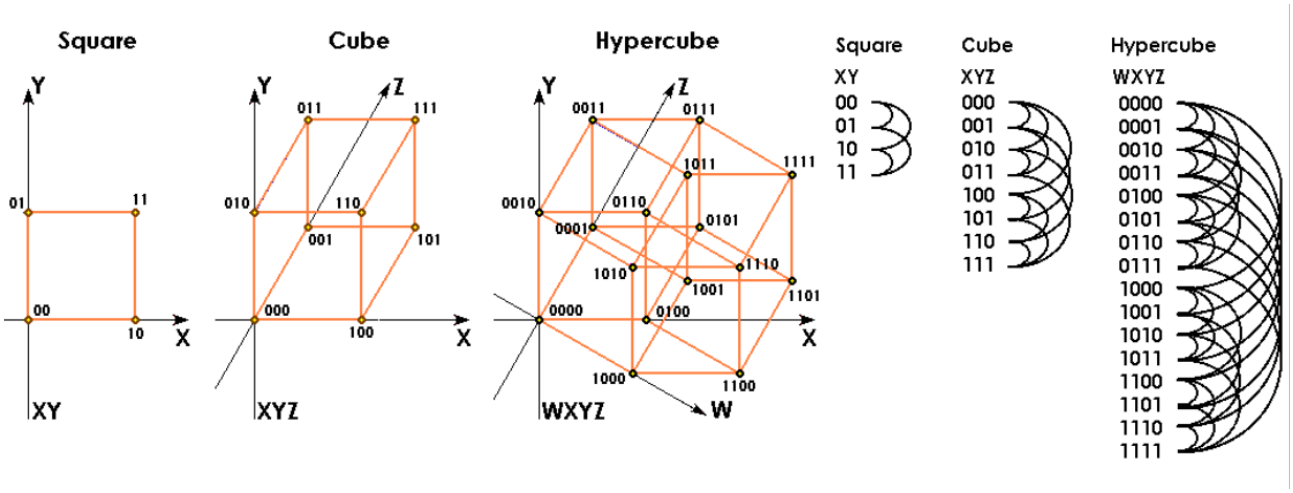
Fitness Landscapes

Q: What is the problem with having too low a mutation rate?

Q: What is the problem with having too high a mutation rate?

Q: When using creep mutation of real valued genes, why a mutation amount from a Gaussian distribution better than selecting from a flat distribution?

To answer these questions (or even to think about them), it can be useful to think about evolution as a **search** process, and to conceive of what is called a **fitness landscape**. In this way of thinking about genetic algorithms, evolution is searching for a set of parameters that maximise fitness. We can thus conceive of the **parameter space** as the set of all parameters. The mutation operator implies a measurement of distance in this space.



Each gene implies a different dimension of the parameter space. The cube above is the entire parameter space for a 3-binary-gene parameter space. When the parameter space (a.k.a. the search space) is this small, it is usually feasible to conduct an exhaustive search, so with GA's the search space is high-dimensional, which quickly becomes hard to visualize. When the genes vary continuously, it is hard/impossible to show anything greater then 3D.

The fitness landscape associates a fitness value with every point in the parameter space. Usually, we don't know the fitness landscape because if we did, we would know where the highest fitness value is, and we wouldn't need to bother with a genetic algorithm! Nevertheless fitness landscapes can be very useful in thinking about GAs. Let's take a look at one...

DEMO