Nature Inspired Computing

K.Dimopoulos

About Evolutionary computing

EC is the field of computer science that borrows ideas from Darwin's **theory of evolution** (as expressed originally in his book “The origin of species”), and expresses these ideas as **optimization algorithms** in computer science.

The “infinite monkey theorem”

A monkey hitting keys randomly on a typewriter will eventually type the complete works of Shakespeare (given an infinite amount of time).

- Let’s consider a monkey named George. George types on a reduced typewriter containing only **27** characters: twenty-six letters and one space bar. So the probability of George hitting any given key is one in twenty-seven.
- Therefore, the probability that George will type the full phrase “to be or not to be that is the question” (39 chars long) is: (1/27) multiplied by itself 39 times, i.e. $(1/27)^{39}$.
- Even if George is a computer simulation and can type one million random phrases per second, for George to have a 99% probability of eventually getting it right, is more than the age of the universe!

The 3 elements of Natural Selection (NS)

1. **Heredity.** There must be a process in place by which children receive the properties of their parents. In biology this is achieved with the genes of the parent organisms (**genotype**) giving birth to the child (**phenotype**)
2. **Variation.** There must be a variety of traits present in the **population** or a means with which to introduce variation.
3. **Selection.** There must be a mechanism by which some members of a population have the opportunity to be parents and pass down their genetic information and some do not. This is typically referred to as “survival of the fittest.”

Genetic Algorithms

A Genetic Algorithm is a structured form of Natural Selection

Characteristics of a Genetic Algorithm

To solve a given problem with a GS we need to have:

- A way of uniquely coding a possible solution of the problem as a vector (array) of things (or objects).
  - These can be numbers, characters, objects, other arrays, whatever. However there should be a one to one encoding of the vector (genotype) to the solution (phenotype)
- A population of such candidate solutions, that each solve the problem with varying success. These are initially randomly created.
- A way of evaluating the fitness (performance) of each member of the population of solutions.
- A way of creating offspring of the solutions with the best fitness by using their genotypes. Typically this is done with two operations: **crossover** and **mutation**

The Genetic Algorithm steps

1. **Initialisation.** Initialise the population with random possible solutions.
2. **Evaluate fitness.** For each member in the population evaluate its fitness (how well it solves the given problem.
3. **Selection.** Taking the fitness into consideration select 2 parents. Best fitted members should have a higher chance of being selected.
4. **Procreation.** Create offspring from the 2 parents. The offspring will substituted the members of the population with the worst performance. This is repeated fo at least until 60% of the initial population is substituted. However, it is possible to substitute all of the initial population.
5. Repeat from step 2, until satisfied.

Population initialisation

Not much to say here. Create random solutions. We do not care how well they perform. We care that there is as much variation as possible.

The population size is usually fixed.

The population size has an impact on how fast we find an optimal solution. The bigger the population the smallest the generations (iterations) needed to find a solution

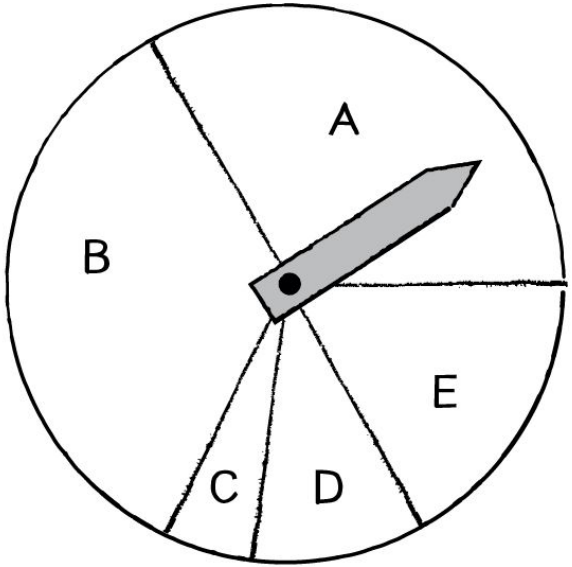
The population size has an impact on the algorithm speed of each generation. The bigger the population the longer time needed in each generation

Fitness evaluation

For our genetic algorithm to function properly, we will need to design what is referred to as a fitness function. The function will produce a numeric score to describe the fitness of a given member of the population.

This, of course, is not how the real world works at all. Creatures are not given a score; they simply survive or not. But in the case of the traditional genetic algorithm, where we are trying to evolve an optimal solution to a problem, we need to be able to numerically evaluate any given possible solution.

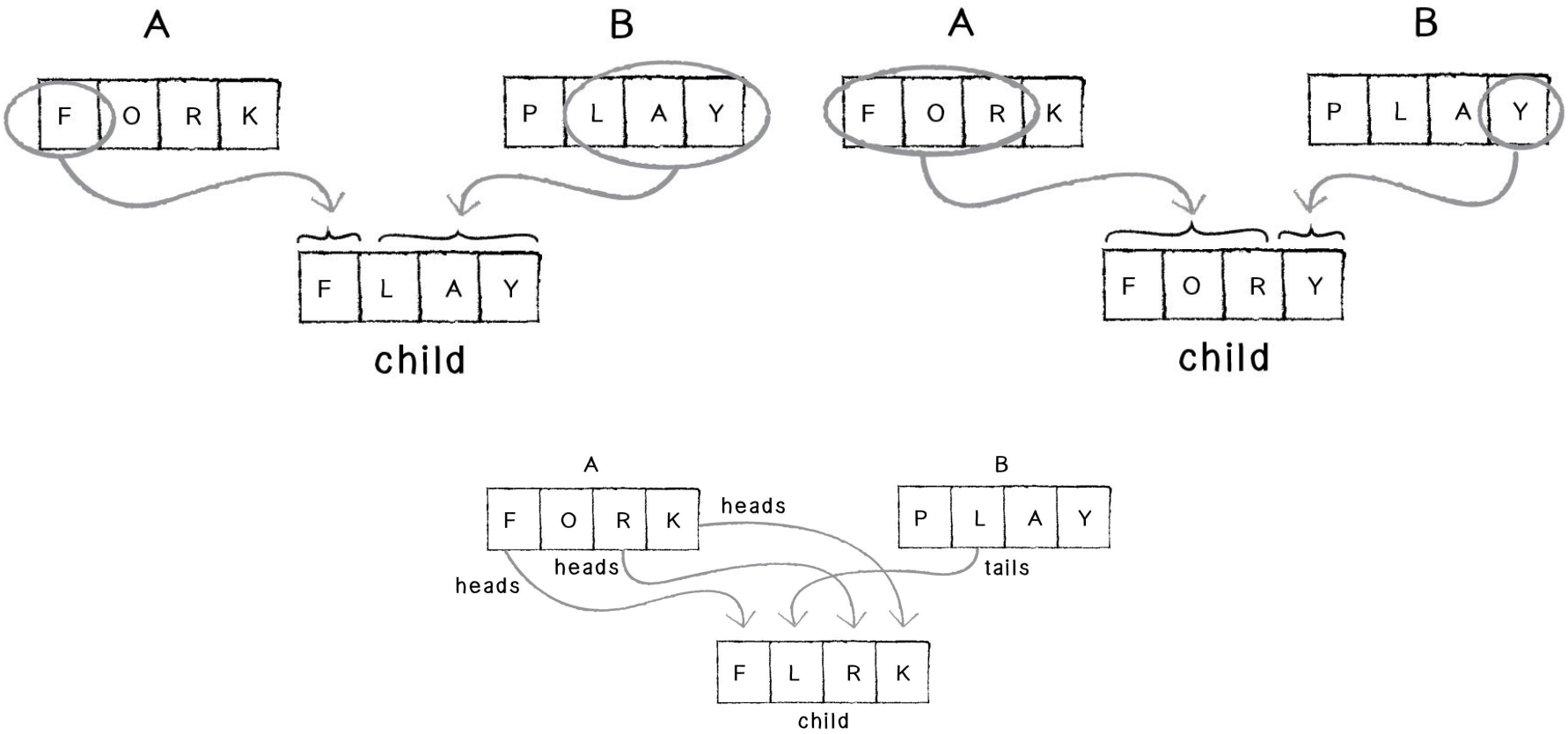
Selection



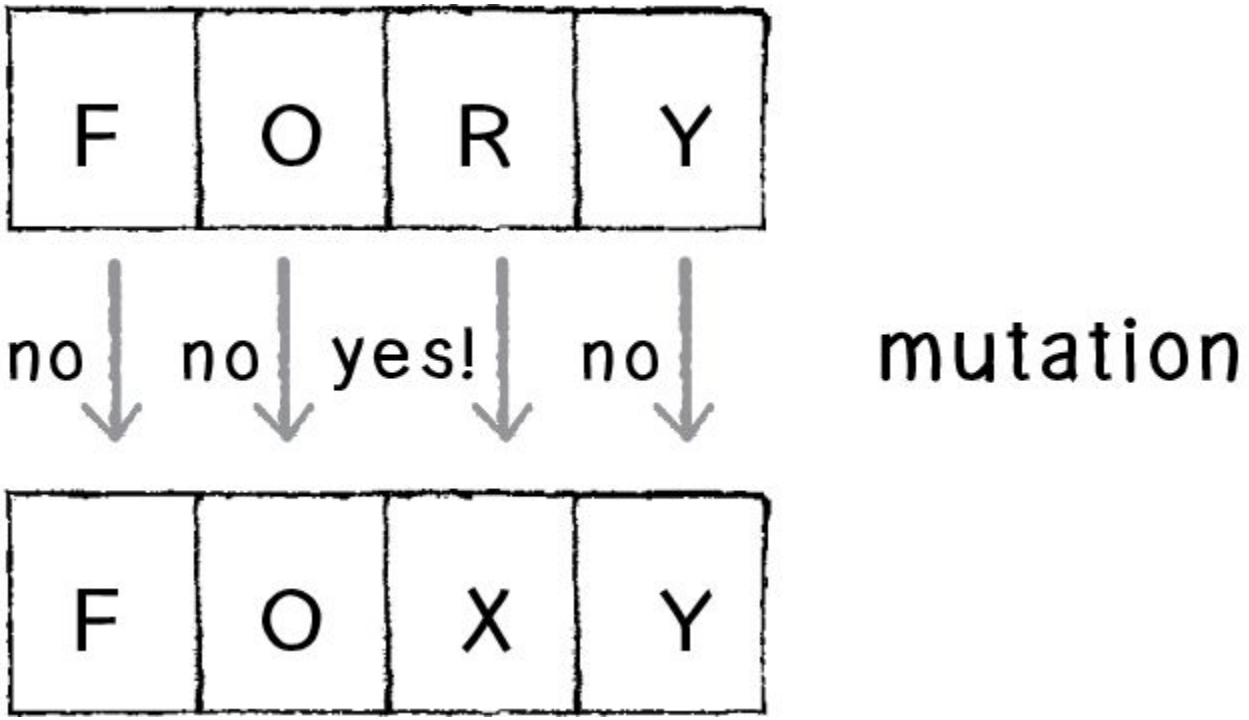
Spin the wheel!

Parent	Probability
A	30%
B	40%
C	5%
D	10%
E	15%

Procreation - Crossover



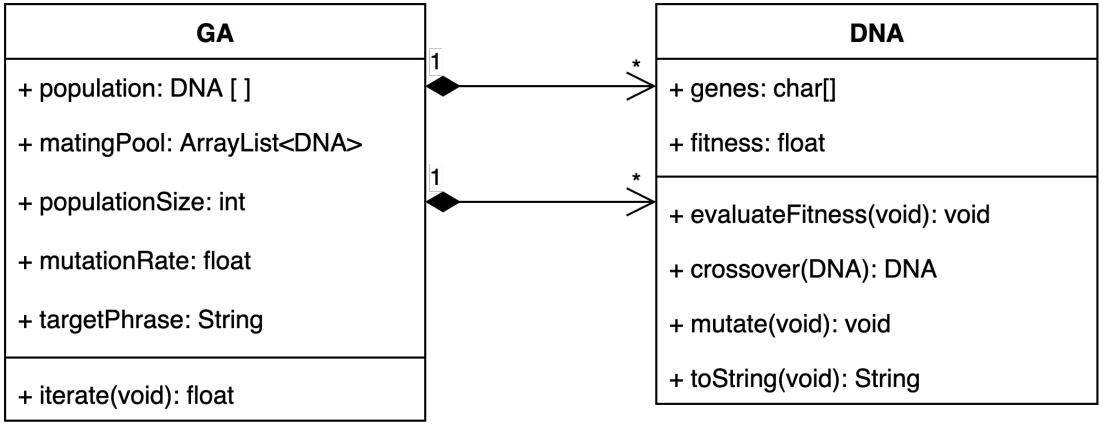
Procreation - mutation



A GA example

We will begin with creating a class that encodes a possible solution to the typing-monkey example.

- **Problem**: find the target phrase
- **Fitness**: % of correct letters in correct places
- **Selection**: add to matingPool copies of each solution based on fitness
- **Procreation**: 1-point crossover + mutation



SMART ROCKETS

Evolving Forces: Smart Rockets

A population of rockets launches from the bottom of the screen with the goal of hitting a target at the top of the screen (with obstacles blocking a straight line path).

Our rockets will have only one thruster, and this thruster will be able to fire in any direction with any strength for every frame of animation.

Let's start by taking our basic `Mover` class and renaming it `Rocket`.

Using the above framework, we can implement our smart rocket by saying that for every frame of animation, we call `applyForce()` with a new force. The "thruster" applies a single force to the rocket each time through `draw()`.

Steps of our algorithm

In the previous example, each frame calculated 1 generation. However, now we don’t want to do this every frame. Rather, our steps work as follows:

1. Create a population of rockets
2. Let the rockets live for N frames
3. Evolve the next generation
    - Selection
    - Reproduction
4. Return to Step #2

Continuous GAs

GA VS the real world

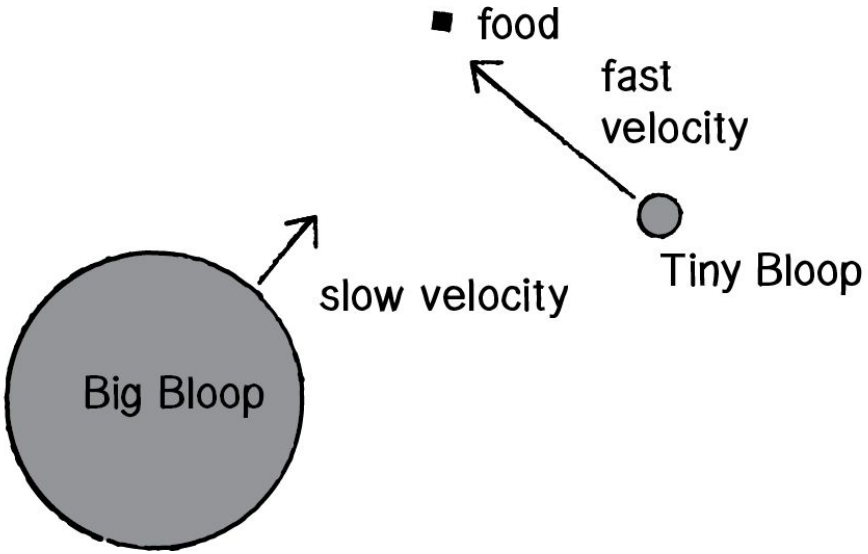
You may have noticed something a bit odd about every single evolutionary system we've built so far. After all, in the real world, a population of babies isn't born all at the same time. Those babies don't then grow up and all reproduce at exactly the same time, then instantly die to leave the population size perfectly stable. That would be ridiculous. Not to mention the fact that there is certainly no one running around the forest with a calculator crunching numbers and assigning fitness values to all the creatures.

In the real world, we don't really have "survival of the fittest"; we have "survival of the survivors." Things that happen to live longer, for whatever reason, have a greater chance of reproducing. Babies are born, they live for a while, maybe they themselves have babies, maybe they don't, and then they die.

Ecosystem Simulation

We'll create a creature called a "bloop," a circle that moves about the screen **according to Perlin noise**. The creature will have a **radius** and a **maximum speed**. The bigger it is, the slower it moves; the smaller, the faster.

We'll want to store the population of bloops in an ArrayList, as we expect the population to grow and shrink according to how often bloops die or are born. We can store this ArrayList in a class called World, which will manage all the elements of the bloops' world.



Our design

As the bloop has a physical form now (phenotype) we will separate it from the genotype (DNA).

In addition, the bloop will be eating food. We will model this in a separate class as an array of locations were the food exists (pasivelly). At random times new food will be added at our world.

So, in addition to the World we will have 4 classes:

Designing the Bloop phenotype

The Mover we did previously is the basis of our bloop, but with the following changes:

1. It uses **perlin noise to set a random velocity** and not forces However in the future we could use forces
2. It has a **size which affects the maxspeed** the bloop can move.
3. It has a **health that decreases with time**, and increases with food. **If the food gets to 0 the bloop dies!**
4. It can eat food, so at any time it needs to c**heck if any of the food is within its radius**
5. It has a **1% chance of spontaneously reproduce with mutation** only (as it is a single parent).

Mover
+ location: PVector + velocity: PVector + acceleration: PVector
+ step(void): void + display(void): void + bounceOnEdges(void): void + passEdges(void): void + applyForce(PVector): void

Updating the DNA class

The DNA now will now just hold a value that determines the size and maxspeed of the phenotype.

Although this is a single value, because in the future we might make our bloop more complex, we will model it as an array of a single float

We will also modify the crossover (for future use)

Finally we need a copy method that will (temporally) substitute the crossover function.

The Food and world class

The Food class is a wrapper class of an ArrayList of PVectors.

All we need are methods that:

- add a food at a location,
- returns the ArrayList, and
- displays the food

The World class is the class that everything comes together.

It will handle:

- the population (as an ArrayList now)
- The food collection
- Food being created

The complete class diagram