

NOC – Evolution of code

Keivan Ipchi Hagh

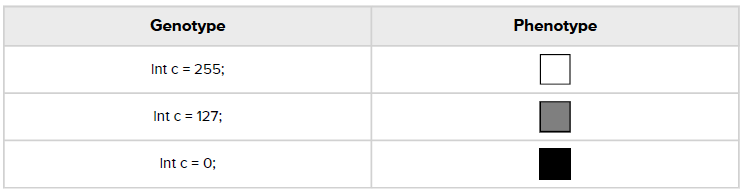


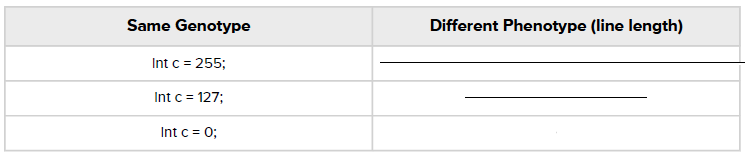
March 21, 2020

# The Generic Algorithm, Part I: Creating a Population

Create a population of randomly generated elements called “DNA”. In field of generic algorithms, we have two important concepts:

* Genotype: This is what gets passed down generation to generation (Heritage).
* Phenotype: This is the expression of data. How we choose to express the module.





More specific description would be:

*Create a population of N elements, each with randomly generated DNA*

# The Generic Algorithms, Part II: Selection

Here, Darwinian principle of selection is applied as followed:

1. *Evaluate fitness*

Fitness, is an overall score given to each element, describing its similarity level with the desired target.

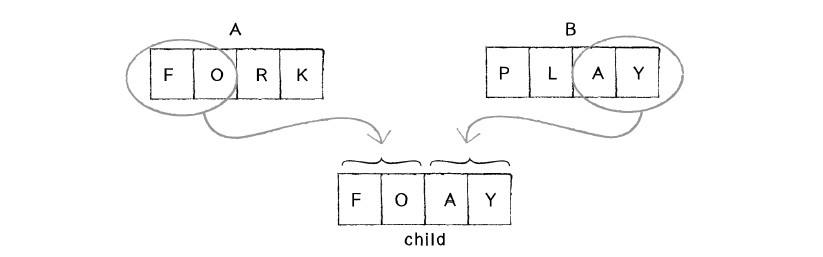
1. *Create a mating pool*

Once fitness is calculated, we must make a (as we call) *mating pool* using a probabilistic method. Basically, a weighed wheel of fortune.

# The Generic Algorithms, Part III: Reproduction

Now that we have chosen two or more parents using our *mating pool*, we must pass their DNA to a newly made element. How we do this:

1. Crossover

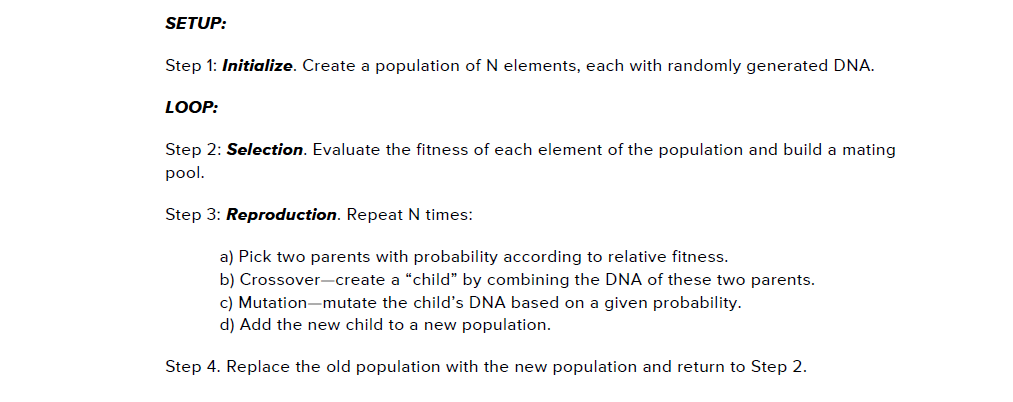
An algorithm that determines how the DNA genes are passed from parents to child.

There are various algorithms to implement *Crossover*. Implementations have not been mentioned here.

1. Mutation

Mutation maintains some variation when DNA is inherited. This ensures the population variation in future generations.

# Setup Sudo-Code:



# Crossover Implementation:

// Crossover, creates a new genes array using the two parents

    DNA crossover(DNA partner) {

      int midPoint = (int)random(genes.length);  // Random location in genes array

      DNA child = new DNA(genes.length);  // Create child DNA

      // Inherit from patner and self

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

        if (i < midPoint)

          child.genes[i] = this.genes[i];

        else

          child.genes[i] = partner.genes[i];

      return child;

    }

# Mutation Implementation:

// Mutate genes based on the given probability

    void mutate(float mutationRate) {

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

        if (random(1) < mutationRate) {

          this.genes[i] = PVector.random2D();

          this.genes[i].mult(random(0, maxForce));

        }

    }

# Accept Reject Implementation (Weighted Probability):

// Accept Reject algorithm for calculating weighted probability

    Rocket acceptReject(float maxFitness) {

      int safe = 0;  // In we couldn't find the appropriate parent, then giveup (Throw error)

      while (safe < 1000) {

        Rocket partner = population[(int)random(population.length)];  // Choose a random Rocket

        float prob = random(maxFitness);

        if (prob < partner.getFitness())

          return partner;

        safe++;

      }

      return null;

    }

# Natural Selection Implementation:

// selection

    void generate() {

      float maxFitness = getMaxFitness();

      Rocket[] newPopulation = new Rocket[population.length];

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

        // Choose two fittest parents' DNAs

        DNA parent1\_DNA = acceptReject(maxFitness).dna;

        DNA parent2\_DNA = acceptReject(maxFitness).dna;

        DNA child\_DNA = parent1\_DNA.crossover(parent2\_DNA);  // Inherit genes

        child\_DNA.mutate(mutationRate);  // Mutate genes

        newPopulation[i] = new Rocket(new PVector(width / 2, height + 20), child\_DNA);

      }

      population = newPopulation;  // Replace the old generation with the new one

      generations++;

    }

# Mating Pool Implementation:

// Generate a mating pool

  void selection() {

    // Clear the ArrayList

    matingPool.clear();

    // Calculate total fitness of whole population

    float maxFitness = getMaxFitness();

    // Calculate fitness for each member of the population (scaled to value between 0 and 1)

    // Based on fitness, each member will get added to the mating pool a certain number of times

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

      float fitnessNormal = map(population[i].getFitness(), 0, maxFitness, 0, 1);

      int n = (int) (fitnessNormal \* 100);  // Arbitrary multiplier

      for (int j = 0; j < n; j++) {

        matingPool.add(population[i]);

      }

    }

  }

\*Credits: These implementations are used in “Shakespeare Monkey Typer” and “AutonomousRockets” inspired from NOC by Daniel Shiffman.