# 4-ga

December 9, 2020

## 1 Algoritmos Genéticos

### 1.1 Un poco de biología

Podemos concebir que cada organismo, tiene, un conjunto de reglas que indican como formarlo a partir de componentes básicos. Estas reglas están codificadas en los **genes** del organismo, los cuales están conectados en **cromosomas**. Cada **gen** representa un rasgo del organismo (su color de pelo, por ejemplo) (Esto es una sobresimplificación) y tiene diferentes posibles valores (**alelos**). Los genes y sus valores se conocen como **genotipo** y la expresión del genotipo (el organismo) se llama **fenotipo**.

Al conjunto completo de material genético se le conoce como **genoma**.

Cuando los organismos se aparean comparten sus genes. El nuevo organismo que surge de esta cruza tendra la mitad de genes de cada organismo (**heredó** de sus padres), a este proceso se le denomina **recombinación** (también llamado **crossover**). Muy raras veces un gen **muta**.

A partir de aquí el organismo (**fenotipo**) interactúa con el mundo y si es *apto* (**selección natural**), se reproducirá, pasando sus genes (la mitad) a su descendencia.

Este proceso tan simple puede generar una gran variabilidad en las **poblaciones** e irlas *optimizando* respecto a su medio ambiente.

### 1.2 Un poco de computación

Viéndolo de cierta manera, la naturaleza está *resolviendo* problemas de optimización: ¿Cuál es el fenotipo que mejor posibilidades de sobrevivir y reproducirse? y esto se puede abstraer a ¿Cuál es la solución a este problema en particular?

Podemos afinar lo dicho y decir que la naturaleza *busca* soluciones que *optimicen* o resuelvan el problema. Si implementamos (o simulamos) este "algoritmo" tenemos lo que se conoce como **algoritmo genético**.

En el **algoritmo genético**, cada fenotipo es una posible realización de una solución. El espacio de todas las posibles soluciones es el **espacio de búsqueda** y cada punto es una posible solución. A esta solución usando una función de *fitness* se le puede asignar un valor.

Buscar la solución en este espacio es equivalente a buscar un valor extremal en el espacio de búsqueda.

El **algoritmo genético** queda definido como una búsqueda heurística que intenta inspirarse en los procesos de la selección natural:

1. Herencia
2. Mutación
3. Crossover
4. Selección

Los **algoritmos genéticos** son parte del campo de estudio de la **computación evolutiva** y fueron desarrollados por **John Holland**, en 1975.

## 1.3 Algoritmo básico

1. Generar una población de `n` cromosomas (posibles soluciones al problema)
2. Evaluar con la función `fitness` a cada cromosoma `x` de la población.
3. Crear una nueva población repitiendo los pasos siguientes hasta que haya una nueva población:
    a. Seleccionar dos cromosomas de la población de acuerdo a su valor asignado con `fitness`.
    b. Aplicar un `crossover` al par de cromosomas para formar un nuevo cromosoma.
    c. Ver si se aplica una mutación a un gen.
    d. Agregar el nuevo cromosoma a la nueva población.
4. Reemplazar la población con la nueva población generada.
5. Si se se satisface una condición de alto, terminar y regresar la mejor solución de la población actual.
6. Si no, regresar al paso 2.

### 1.3.1 Lo difícil...

Básicamente hay dos cosas difíciles ¿Cómo hago un cromosoma?¿Cómo hago mi función de `fitness`? Ambas dependerán del problema que se quiera resolver.

## 1.4 Operadores Genéticos

### 1.4.1 Codificando el cromosoma

La opción más popular es usar una cadena de texto binaria:

[1]:
```
cromosoma1 = '11011001001101101'
```

[2]:
```
cromosoma2 = '11110000000000001'
```

[4]:
```
print(cromosoma1)
print(cromosoma2)
```

```
11011001001101101
11110000000000001
```

Cada `bit` representa una característica de la solución, o puede representar un número, o un conjunto de bits contíguos representa un alelo, etc.

Obviamente otras representaciones son posibles como usar números enteros en lugar de bits (para el problema del viajero o **TSP**, cada numero representa una ciudad y el algoritmo permuta.

**Ejercicio** Crea una clase llamada `GA`, en este momento vacía. Y también genera una clase abstracta llamada `Cromosoma`.

**Ejercicio** Crea una clase que herede de `Cromosoma`, llamada `BitCromosoma`, la cual representa un cromosoma codificado de forma binaria. El constructor de esta clase recibe una longitud y opcionalmente una cadena, si no está presente genera una cadena al azar de la longitud establecida.

[1]:
```
import numpy as np
```

[2]:
```
# Escribe aquí la clase GA
class GA():
    """
    tamaño_poblacion --> cuantos tenemos
    tasa_mutacion --> cuantos mutan
    tasa_crossover --> cuantos pasan la prueba
    max_generaciones --> cuantas veces lo dejamos que corra
    fitness --> nos dice que tan bueno es

    poblacion -->
    generacion -->
    mejor_solucion_historica --> para llevar el track de lo que hacemos
    mejor_solucion_actual --> para llevar el track de lo que hacemos
    """
    def __init__(self, tamaño_poblacion, tasa_mutacion, tasa_crossover,␣
↪max_generaciones, fitness):
        self.tamaño_poblacion = tamaño_poblacion
        self.tasa_mutacion = tasa_mutacion
        self.tasa_crossover = tasa_crossover
        self.max_generaciones = max_generaciones
        self.fitness = fitness
        self.poblacion = []
        self.generacion = 0
        self.mejor_solucion_historica = ''
        self.mejor_solucion_actual = ''

    def poblar(self):
        for i in range(self.tamaño_poblacion):
            self.poblacion.append(BitCromosoma())

    def __str__(self):
        #Para poner que es lo que imprime
        return("Poblacion actual: ", self.poblacion,
               "\nMejor solucion historica: ", str(self.
↪mejor_solucion_historica),
```

```
            "\nMejor solucion de la generacion: ", str(self.
↪mejor_solucion_actual),
            "\nNumero de generacion: ", str(self.generacion))

    def run(self):
        #para calcular el fitness de toda la poblacion
        fitness_arr = []
        for i in self.poblacion:
            fitness_arr.append(self.fitness(i))

        self.mejor_solucion_actual = self.poblacion(fitness_arr.index(max(self.
↪fitness_arr)))

        if self.fitness(self.mejor_solucion_historica) < max(self.fitness_arr):
            self.mejor_solucion_historica = self.mejor_solucion_actual

        self.poblacion = self.crear_poblacion(self.fitness_arr)

        if self.condicion():
            return self.mejor_solucion_actual
        else:
            return False

    def seleccionar(self):
        self.poblacion.sort(reverse=True, key=self.fitness)
        fitness_poblacion = sum(self.fitness_arr)
        num_azar = np.random.uniform(low=0, high=fitness_poblacion)
        suma_fitness = 0
        for i in self.poblacion:
            suma_fitness += i
            if suma_fitness > num_azar:
                return i

    def fitness():
        pass
```

[2]:
```
# Escribe aquí la clase abstracta Cromosoma
class Cromosoma():
    def __init__(self):
        pass
```

[3]:
```
# Escribe aquí la clase BitCromosoma
class BitCromosoma(Cromosoma):
    def __init__(self):
        pass
```

[ ]:
```

```

[6]:
```
%cat ga.py
```

```
# Escribe aquí la clase GA
import numpy as np
from Cromosoma import BitCromosoma
import random


class GA:
    def __init__(
            self, tamaño_poblacion, tasa_mutacion,
            tasa_crossover, max_generaciones, fitness,
            num_elites
    ):
        self.tamaño_poblacion = tamaño_poblacion
        self.tasa_mutacion = tasa_mutacion
        self.tasa_crossover = tasa_crossover
        self.max_generaciones = max_generaciones
        self.fitness = fitness
        self.poblacion = []
        self.generacion = 0
        self.mejor_solucion_historica = ''
        self.mejor_solucion_actual = ''
        self.num_elites = num_elites

    def run(self):
        fitness_arr = []
        self.poblacion.sort(reverse=True, key=self.fitness)
        # Evaluación del fitness de toda la poblacion
        for i in self.poblacion:
            fitness_arr.append(self.fitness(i))
        # Encontramos la mejor solución de la generación.
        self.mejor_solucion_actual = \
            self.poblacion[fitness_arr.index(max(fitness_arr))]
        # Comparamos con la mejor solución que hemos encontrado y actualizamos.
        if self.fitness(self.mejor_solucion_historica) < max(fitness_arr):
            self.mejor_solucion_historica = self.mejor_solucion_actual
        # Hacemos la siguiente generación de los genes
        self.poblacion = self.crear_poblacion(fitness_arr)
        # Evaluamos condición de paro
        if self.condicion():
            return self.mejor_solucion_actual
        else:
            return False

    def crear_poblacion(self, fitness_arr):
```

```
    pass

def crossover(self, i, j):
    crossover_point = random.randint(0, i.length)
    pass

def seleccionar(self, fitness_arr):
    new_gen = self.poblacion[1:self.num_elites]
    fitness_poblacion = sum(fitness_arr)
    for i in range(len(self.poblacion) - self.num_elites):
        num_azar = np.random.uniform(low=0, high=fitness_poblacion)
        suma_fitness = 0
        for j in self.poblacion:
            suma_fitness += self.fitness(j)
            if suma_fitness > num_azar:
                new_gen.append(j)
    return new_gen

def plot():
    pass

def fitness():
    pass

def poblar(self):
    for i in range(0, self.tamaño_poblacion):
        self.poblacion.append(BitCromosoma())

def __str__(self):
    return ("Población Actual" + self.poblacion +
            "\nMejor Solución Histórica" +
            str(self.mejor_solucion_ghistorica) +
            "\nMejor Solución de la Generación" +
            str(self.mejor_solucion_actual) +
            "\nNúmero de generación" + str(self.generacion))


# def seleccionar(self):
# #self.poblacion.sort(reverse=True, key=self.fitness)
# fitness_poblacion = sum(self.fitness_arr)
# num_azar = np.random.uniform(low=0, high=fitness_poblacion)
# suma_fitness = 0
# for i in self.poblacion:
#     suma_fitness += self.fitness(i)
#     if suma_fitness > num_azar:
#         return i
```

[7]:
```
%cat Cromosoma.py
```

```
import random


class Cromosoma:
    def __init__(self):
        pass


class BitCromosoma(Cromosoma):
    def __init__(self, length):
        self.length = length
        self.gen_rand()

    def gen_rand(self):
        self.gen = [random.randrange(2) for i in range(self.length)]

    def __str__(self):
        return ''.join([str(i) for i in self.gen])


if __name__ == '__main__':
    b = BitCromosoma(length=8)
    print(b)
```

[11]:
```
import ga
import Cromosoma
```

### 1.4.2 El algoritmo

**Ejercicio** Regresa a la clase `GA`, crea un constructor que reciba como parámetros: `tamaño_poblacion`, `tasa_mutacion` `tasa_crossover`, `max_generaciones` y `fitness`. En el constructor inicializa además la variable `poblacion`, `generacion`, `mejor_solucion_historica`, `mejor_solucion_actual`.

**Ejercicio** Agrega un método `poblar` que cree una población de `BitCromosomas` y las guarde en la variable `poblacion`.

Sé que te preguntarás por qué se ve tan feo esto último, una posible solución está en usar **patrones de diseño** y en particular el patrón de diseño `Factory`. Más info aquí.

**Ejercicio** Agrega un método `__str__` que imprima la población actual, la mejor solución histórica, la mejor solución de la generación actual y el número de generación.

**Ejercicio** Agrega un método `run` y codifica los pasos de la sección **Algoritmo básico**. Este método debe de guardar separadas por comas el número de la generación, el fitness promedio de la población, la mejor solución actual y la mejor solución historica.

**Ejercicio** Agrega un método `plot` que despliegue como el fitness evoluciona con las generaciones.

### 1.4.3 Selección

Según la teoría de Darwin, los mejores sobreviven y se reproducen, pasando sus genes a la siguiente generación. Hay muchas maneras de seleccionar los mejores cromosomas para reproducirse. Un método muy usado es **selección de ruleta de la fortuna**.

Para explicarla, imagina una ruleta y divídela en `n` secciones donde cada sección tiene un área proporcional al `fitness` del cromosoma.

Si eliges al azar de la ruleta, los cromosomas con mejor `fitness` serán elegidos más veces.

El algoritmo es:

1. Calcula la suma de todos los `fitness` de todos los cromosomas de la población, llama a esa suma `fitness_poblacion`. Normaliza el `fitness`.
2. Ordena descendentemente.
3. Genera un número al azar en el intervalo $(0, $ `fitness_poblacion`).
4. Recorre la población sumando los `fitness`, cuando la suma sea mayor que el número generado al azar, regresas el cromosoma donde estás.
5. Regresa al punto 3 hasta tener todos los pares para producir la siguiente generación.

**Ejercicio** Agrega un método a `seleccionar` a `GA` que implemente el algoritmo recién discutido.

**Ejercicio** Una mejora sustancial al algoritmo es llamado `elistismo`. En esta modalidad, el algoritmo **copia** a la nueva generación al mejor (o mejores) cromosomas. ¿Por qué crees que sea bueno esto?

**Ejercicio** Agrega al constructor un parámetro `num_elites`, si es mayor que cero, antes de la selección tradicional, copias los `num_elites` mejores a la siguiente generación.

### 1.4.4 Crossover

Esta operación selecciona genes de los padres para generar un nuevo par de cromosomas. La manera más fácil de hacerlo es elegir al azar un punto donde hacer el cruce de la cadena, dividirla e intercambiarla.

```
crossover_point = random.choice(#Posibles posiciones del cromosoma)

hijo1 = cromosoma1[#usa slicing] + cromosoma2[...]

hijo2 = # Lo mismo, pero al revés
```

**Ejercicio** Crea un método `crossover` que reciba dos `cromosomas` y devuelva dos cromosomas. ¿Dónde debe de ir este método?

**Ejercicio** Crea un método abstracto `split` que reciba el `crossover_point` y devuelva partido el `cromosoma`. ¿Dónde debe de ir este método? Modifica `crossover` para que utilice el método `split`.

**Ejercicio** Implementa el método `split` en `BitCromosoma`.

Es posible pensar en otras manera de hacer el *crossover* , como por ejemplo, varios puntos de cruce.

### 1.4.5 Mutación

La mutación pretende sacar a la población de un extremal local. La mutación cambia al azar, en la codificación binaria, un ( varios) bit elegido(s) al azar. Si un cromosoma muta o no, depende de

una probabilidad que regularmente es pequeña.

**Ejercicio** Agrega un método abstracto `mutar` a `Cromosoma`.

**Ejercicio** Agrega al constructor de `BitCromosoma` un parámetro de `tasa_mutación`. Implementa el método `mutar` en `BitCromosoma` implementando lo discutido.

### 1.5 Tarea

**Fecha de entrega**: 2 de diciembre, 2020

**Ejercicio** Ejecuta un algoritmo genético, que trate de generar una cadena de longitud `8`, $solucion \equiv$ `11001100`. ¿Cuántas generaciones tardó? Muestra el avance cada `10` generaciones.

**Ejercicio** Ejecuta un algoritmo genético, que trate de generar una matriz de `12` $\times$ `12` que tenga una cruz enmedio (como la bandera de Suiza). ¿Cuántas generaciones tardó? Muestra el avance cada `20` generaciones.

**Ejercicio** Ejecuta un algoritmo genético, que encuentre el mínimo de $f(x) = x^2 + 2$.

```
[ ]:
```