



## Proyecto final

### Los algoritmos genéticos y su relación con la evolución biológica

Francisco Contreras Ibarra , José Ethan Ortega González  y Alvaro Ramírez López 

#### PROYECTO FINAL

Entrega 13/06/2024

#### Números de cuenta

Francisco Contreras Ibarra  
316083786  
José Ethan Ortega González  
316088327  
Alvaro Ramírez López  
316276355

#### Correos

Francisco Contreras Ibarra  
franciscoc.ibarra@ciencias.unam.mx  
José Ethan Ortega González  
ethan@ciencias.unam.mx  
Alvaro Ramírez López  
alvaro@ciencias.unam.mx

## Introducción

Los algoritmos genéticos pueden aplicarse en áreas que van desde el diseño de circuitos integrados hasta redes de comunicación [1, p. X]. Al estar basados en el proceso evolutivo de las especies, vale la pena determinar si es una buena herramienta para representar la evolución de los caracteres de una especie. En este proyecto se utilizarán los algoritmos para realizar esta tarea para una población de pingüinos de algunas islas de la Antártida.

Finalmente, el objetivo de este proyecto es determinar si es posible utilizar los algoritmos genéticos para simular como la población de un organismo evoluciona a través de distintas generaciones.

**Palabras clave** Genética, algoritmos genéticos, Mutación Genética, Recombinación Genética, Árboles Genealógicos, Cromosoma

## 1. Elaborando el framework

Se desarrolló un framework en el lenguaje de programación **elixir** que nos permite resolver varios problemas de optimización con la ayuda de los algoritmos genéticos. Nos basamos principalmente en el libro **Genetic Algorithms in Elixir**, donde nos enseña paso a paso como crear el framework e incluso presenta un ejemplo en los capítulos 9 y 10 de lo que se quiere lograr con el proyecto. En el **Listado 1** presentamos la estructura de los archivos del framework.

```

├── proyecto_final
│   ├── lib
│   │   ├── genetic.ex
│   │   ├── problem.ex
│   │   ├── toolbox
│   │   │   ├── crossover.ex
│   │   │   ├── mutation.ex
│   │   │   ├── reinsertion.ex
│   │   │   └── selection.ex
│   │   ├── types
│   │   │   └── chromosome.ex
│   │   ├── utilities
│   │   │   ├── application.ex
│   │   │   └── statistics.ex
│   ├── mix.exs
│   ├── scripts
│   │   ├── penguin_simulation.ex
│   │   └── tiger_simulation.exs

```

**Listado 1 Estructura de archivos:** La estructura básica de un proyecto de **Mix**, una herramienta para administrar proyectos de Elixir, entre otras cosas más.

El framework tiene dos carpetas de interés: `lib` y `scripts`. La carpeta `lib` contiene toda la lógica necesaria para implementar las distintas etapas de los algoritmos, junto con algunos mecanismos que nos permiten rastrear valores estadísticos y la genealogía del problema.

Por otro lado, la carpeta `scripts` contiene la definición de todos los problemas que nos interesa resolver. Para cada uno de estos problemas es necesario crear un script de Elixir donde se defina un módulo con las tres funciones que todo problema que queramos resolver con algoritmos genéticos deba de tener, además del código necesario para la ejecución del programa.

En esta sección describiremos el archivo `lib/proyecto_final.ex`, donde se encuentra la implementación general del algoritmo, mientras que las etapas más específicas del algoritmo se explicarán más adelante, en sus respectivas secciones. En el [Listado 2](#) se muestra la estructura general del código que se encuentra en ese archivo.

```
defmodule Genetics do
  alias Toolbox.{Crossover, Mutation, Select, Evaluate, Reinsertion}

  require Logger

  def run(problem, opts \\ []), do: ...
  def initialize(genotype, opts \\ []), do: ...
  def evolve(population, problem, generation, opts \\ []), do: ...
  def evaluate(population, fitness_function, opts \\ []), do: ...
  def select(population, opts \\ []), do: ...
  def crossover(population, opts \\ []), do: ...
  def mutation(population, opts \\ []), do: ...
  def reinsertion(parents, offspring, leftover, opts \\ []), do: ...
  defp init_statistics(), do: ...
  defp statistics(population, generation, opts), do: ...
end
```

**Listado 2 Estructura principal del código:** Las funciones que componen al archivo `lib/proyecto_final.ex` y por lo tanto, a nuestro algoritmo.

A continuación se describirá el funcionamiento de `run`, `initialize`, `evolve` y `evaluate`. Las funciones `select`, `crossover`, `statistics` se explicarán mas adelante:

- `run(problem, opts \\ [])`: Esta función es la principal del algoritmo genético. Recibe un problema y un conjunto opcional de hiperparámetros. Inicia el algoritmo genético llamando a la función `initialize` para crear la población inicial y luego llama a la función `evolve` para evolucionar la población a través de las generaciones.
- `initialize(genotype, opts \\ [])`: Esta función se encarga de inicializar la población. Recibe una función `genotype` y un conjunto opcional de hiperparámetros. La función `genotype` se utiliza para generar cada individuo de la población. El tamaño de la población se obtiene de los hiperparámetros con un valor por defecto de 100. Finalmente, la población inicial se añade a la genealogía.
- `evolve(population, problem, generation, opts \\ [])`: Esta función es la encargada de evolucionar la población a través de las generaciones. Recibe la población actual, el problema, la generación actual y un conjunto opcional de hiperparámetros. Primero, evalúa la población utilizando la función `evaluate`. Luego, verifica si se ha alcanzado la condición de terminación del problema. Si no se ha alcanzado, selecciona a los padres, realiza el cruce y la mutación, y reinserta a los hijos en la población. Finalmente, llama a sí misma para la siguiente generación.
- `evaluate(population, fitness_function, opts \\ [])`: Esta función se encarga de evaluar la aptitud de cada individuo en la población. Recibe la población, la función de aptitud y un conjunto opcional de hiperparámetros. Utiliza el operador de evaluación especificado en los

hiperparámetros (o `Evaluate.heuristic_evaluation` por defecto) para evaluar a cada individuo. Devuelve la población con los valores de aptitud actualizados.

## 2. Recolección de datos

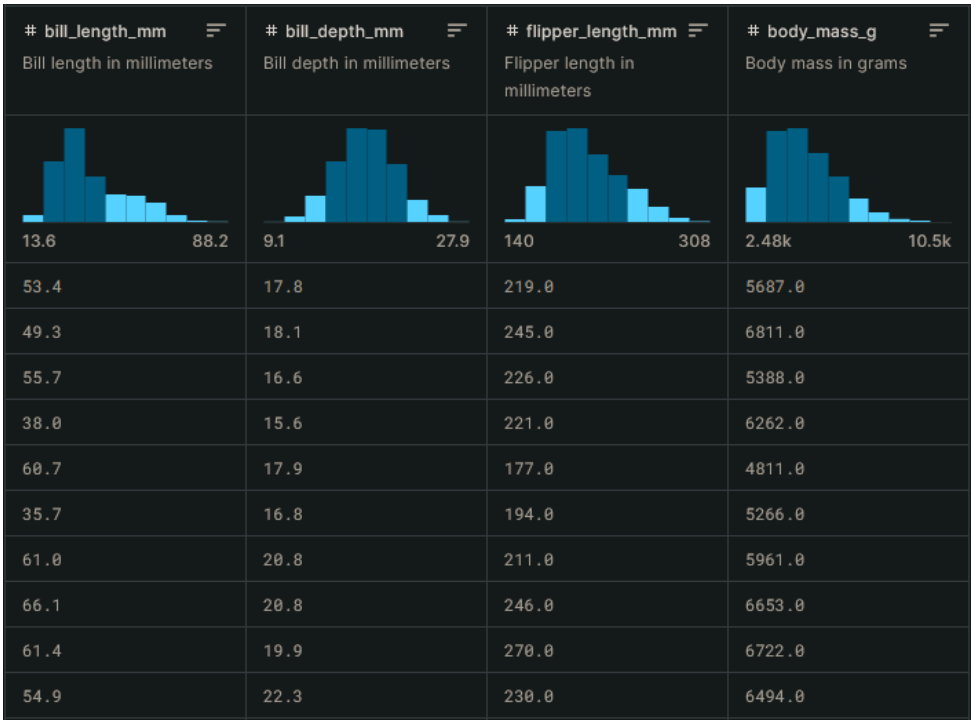
Los datos que se utilizaron para la ejecución del algoritmo se pueden encontrar en el dataset [Palmer Penguins Dataset Extended](#)[2]. Este dataset contiene descripciones de muestras hipotéticas correspondientes a 3 especies de Pingüinos pertenecientes al archipiélago de Palmer, cerca de la Antártida, así mismo dicho dataset proporciona métricas sobre puntos importantes como los son los indicadores de salud, etapas de vida, dieta, etc.

Para la implementación del algoritmo y su ejecución se deben de considerar 2 cuestiones: la recolección de datos sobre los individuos de la población para la simulación y las condiciones del habitat o ambiente que se usaran para la simulación.

Para los individuos se tomaron en consideración solamente 4 columnas de todo el dataset, se tomo esta decisión dado que estas características nos proporcionan mas información sobre las 3 especies de Pingüinos, por el momento se comenzara a describir estas 4 columnas:

- Body mass (Masa corporal): de 2.48 a 10.5 Kg
- Bill Length (Longitud del Pico): de 13.6 a 88.2 mm
- Bill Depth (Profundidad del Pico): de 9.1 a 27.9 mm
- Flipper Length (Longitud de la Aleta): de 140 a 308 mm

También se podrá observar una previsualización de las 4 columnas mencionadas del dataset en la [Figura 1](#).



**Figura 1 Vista previa del dataset para los individuos:** Se muestran las 4 columnas del dataset seleccionadas junto con algunas filas rellenas con los posibles valores para las columnas.

Para el caso del hábitat o ambiente, se tomara la columna del dataset llamada `Island` la cual proporciona mas información sobre la interacción de la población con su entorno, en la [Figura 2](#) podemos ver una muestra de los 2 principales entornos en los Pingüinos

<b>A island</b>	
Island where the penguin was found	
Biscoe	52%
Dream	33%
Other (512)	15%

**Figura 2 Vista previa del dataset para factores ambientales:** Factores como el clima en la Isla Biscoe y la fuente de alimentos que proporcione la Isla Dream pueden ser importantes para la selección del mejor individuo.

En el [Listado 3](#) se muestra el pequeño script (escrito en Elixir) que usamos para limpiar el dataset y solo tomar en cuenta los datos que nos interesan.

```
defmodule CsvManipulator do
  alias NimbleCSV.RFC4180, as: CSV

  def run(input_file, output_file) do
    input_file
    |> File.stream!()
    |> CSV.parse_stream()
    |> Stream.map(&Enum.slice(&1, 2..5)) # solo quedarse con las columnas 3 a 6,
    |> CSV.dump_to_iodata()
    |> (&File.write!(output_file, &1)).()
  end
end

input_file = "datasets/palmerpenguins_extended.csv"
output_file = "datasets/cromosomas.csv"
CsvManipulator.run(input_file, output_file)
```

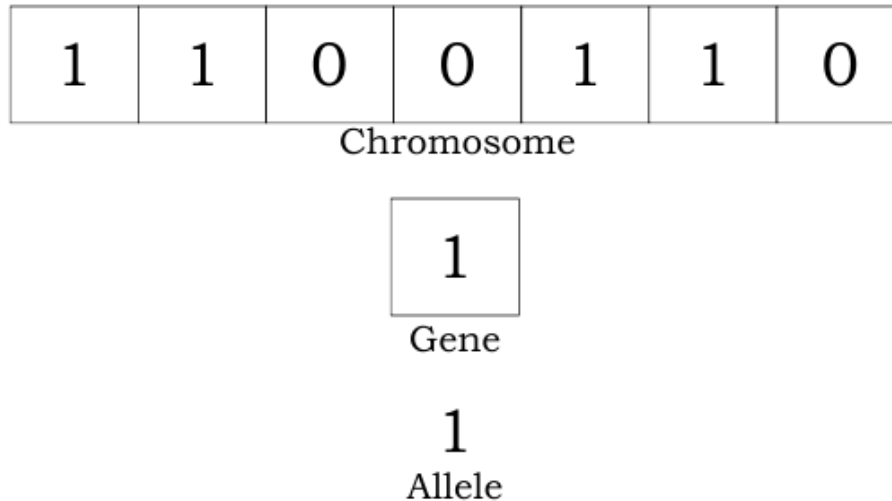
**Listado 3 Script para limpiar el dataset:** Nos quedamos solo con las columnas 3 a 6 del dataset. Como los índices empiezan en 0, se escribe que las columnas que nos interesan son la 2 a la 5.

### 3. Codificando los datos obtenidos

Una vez que ya tenemos los datos ahora tenemos que representarlos de una forma que nuestro framework pueda entender. Para esto se debe de definir un modulo que implemente al protocolo `Problem`, es decir, definir tres funciones que todo problema debe de tener: `genotype`, `fitness_function` y `terminate?`.

#### 3.1. La función `genotype`

La función `genotype` es la encargada de crear un *cromosoma*: una posible solución para el problema que se esté tratando de resolver. Un *cromosoma* puede verse como una serie de *genes* que consisten de valores llamados *alelos*. Además, los genes pueden usarse para representar varias cosas, desde el nombre de ciudades, hasta un carácter de algún organismo. En la [Figura 3](#) se representa de una manera más visual estos conceptos.



**Figura 3 Representación de un cromosoma:** Se muestra la diferencia entre tres conceptos biológicos reales, redefinidos para los algoritmos genéticos: cromosomas, genes y alelos.

Particularmente en el código de nuestro framework, un cromosoma está representado con una struct (un). En el [Listado 4](#) se muestra el código de elixir que define al tipo `Chromosome`.

```
defmodule Types.Chromosome do
  @type t :: %__MODULE__{
    genes: Enum.t(),
    size: integer(),
    fitness: number(),
    age: integer()
  }

  @enforce_keys :genes
  defstruct [:genes, size: 0, fitness: 0, age: 0]
end
```

**Listado 4 Definición de los cromosomas:** Un cromosoma se representa con un struct. Un cromosoma tiene un enumerable llamado `genes` (normalmente una lista), un entero llamado `size` que representa la cantidad de genes, un número llamado `fitness` que representa que tan bueno o malo es el cromosoma y otro entero llamado `age` que representa la edad del cromosoma.

Existen varios tipos de genotipos para representar a nuestros problemas, es decir, diferentes codificaciones para que el framework pueda procesar la información del problema. Dado que en el dataset se están usando letras y símbolos para representar los valores de las columnas, usaremos un *genotipo de valores reales*. Esto no es más que un cromosoma donde sus alelos pueden tener números reales, cadenas, entre otras cosas como valores. [\[3, p. 45\]](#)

En el [Listado 5](#) se presenta la implementación de la función `genotype` que ya usa los datos de los pingüinos.

```
@impl true
def genotype(opts \\ []) do
  index = Keyword.get(opts, :index)
  initial_data = Map.fetch!(Keyword.get(opts, :initial_data), :data)
  gene = Enum.at(initial_data, index)

  %Chromosome{genes: gene, size: Enum.count(gene)}
end
```

**Listado 5 Implementación de la función `genotype`** : La implementación es bastante simple, se crea un cromosoma para cada una de las filas de los datos reunidos.

Podemos notar algunas similitudes y diferencias entre los conceptos desde la perspectiva de los algoritmos con la de la biología:

- En los algoritmos un genotipo contiene un cromosoma que a su vez es una colección de alelos, mientras que en la biología un genotipo es un conjunto de alelos. [4, p. G-15]
- Si bien no tenemos la definición de ADN en los algoritmos genéticos, podemos considerar al ADN de una solución como un cromosoma. Eso no es completamente correcto, ya que un organismo puede tener más de un cromosoma.
- Biológicamente hablando, las mutaciones son el origen principal de los alelos [4, p. 265]. Esto es parcialmente cierto en los algoritmos genéticos, si bien más adelante cuando se empiecen a mutar a los miembros de nuestra población obtendremos nuevos alelos, los valores posibles de los alelos, en un principio, se definen por el tipo de genotipo que se haya elegido.
- En la definición de nuestro problema tratamos cada uno de los caracteres de los hongos como un gen, esto se ajusta bastante bien a la definición biológica del gen, donde un gen puede llegar a definir alguna característica del organismo.

### 3.2. `fitness_function`

La `fitness_function` es considerablemente más complicado de definir. Para poder definirla primero tenemos que asignarle una puntuación a cada uno de los caracteres, en base a que tan beneficio o perjudicial es para el entorno en el que viven los pingüinos.

Idealmente, se tendría que realizar una investigación amplia y detallada, acompañada de observación de las poblaciones de pingüinos en las distintas islas de las que se obtuvieron los datos. Sin embargo, realizar un trabajo como este no es tarea fácil y está mas allá del alcance de este proyecto.

Debido a esto nos vemos forzados a definir valores en base a nuestra propia intuición. En la **Tabla 1** se muestran cada uno de los caracteres con su puntuación para cada una de las islas.

Carácter	Biscoe	Dream	Torgensen
Longitud del Pico	0.0	-2.1	1.7
Profundidad del Pico	-0.8	0.0	2.3
Longitud de la Aleta	0.2	0.1	0.5
Masa corporal	2.5	1.3	-1.4

**Tabla 1 Tabla de puntuaciones:** Las puntuaciones para cada uno de los caracteres en las distintas islas. Si la puntuación es cero entonces el carácter no tiene efecto alguno en la supervivencia del individuo. Si la puntuación es positiva entonces tiene un efecto positivo en la supervivencia del individuo. Por último, si la puntuación es negativa entonces tiene un efecto negativo en la supervivencia del individuo.

Consideramos que esta era una buena tarea para desarrollarla con la ayuda de **ChatGPT**, se le pidió que creara una tabla como la de arriba, y que le asignara una puntuación a cada uno de

los caracteres. En [Apéndice A](#) se encuentra la conversación que se usó para generar los datos, junto con la justificación que se le pidió para los valores de las puntuaciones.

Una vez que se tiene la tabla con cada uno de las puntuaciones, es necesario integrar los valores en el código. Sin embargo, en esta parte nos enfrentamos a algunos problemas al integrarlo al framework.

Debido a las estrategias de recombinación y de mutación que se estaban utilizando al terminas la ejecución del algoritmo se obtenían valores bastante elevados (por ejemplo, pingüinos con picos de metros). Para resolver esto se decidió implementar a `fitness_function` como una función de penalización.

```
@impl true
def fitness_function(chromosome, opts \\ []) do
  initial_data = Keyword.get(opts, :initial_data)
  {scores, limit} = Map.fetch!(initial_data, Keyword.get(opts, :island, :biscoe))

  potential_individual =
    chromosome.genes
    |> Enum.zip(scores)
    |> Enum.map(fn {t, s} -> t * s end)
    |> Enum.sum()

  if potential_individual > limit, do: 0, else: potential_individual
end
```

**Listado 6 Implementación de la función `fitness_function`** : Para implementar la función se hace uso de dos componentes principales: los `scores` y el `limit` . Por la forma en la que está implementada, se puede decir que es una función de penalización.

La idea general de una función de penalización es favorecer valores que están por debajo de un limite previamente establecido, de esta manera aquellos valores con un gran valor de aptitud (y con valores para los caracteres que no tienen sentido) no son tomados en cuenta.

La forma en la que calculamos el limite fue sacando el promedio de el valor de aptitud de todos los cromosomas, tomando en cuenta las puntuaciones de cada isla. Lamentablemente esto junto con la falta de tiempo, nos forzó a calcular el valor de aptitud de un cromosoma dos veces. En [Listado 7](#) se muestra el código para calcular el limite. Además se muestra el resultado: un `Map` con la información de cada isla.

```
island_data =
  [biscoe: [2.5, 0.0, -0.8, 0.2], dream: [1.3, -2.1, 0.0, 0.1], torgensen: [-1.4, 1.7,
2.3, 0.5]]
  |> Enum.map(fn {k, score} ->
    lst =
      for row <- initial_data.data do
        row
        |> Enum.zip(score)
        |> Enum.map(fn {t, s} -> t * s end)
        |> Enum.sum()
      end
      Map.new()
      |> Map.put(k, {score, Enum.sum(lst) / length(lst)})
    end)
  |> Enum.reduce(&Map.merge(&1, &2))

# Resultado:
%{
  biscoe: {[2.5, 0.0, -0.8, 0.2], 897.5858121901398},
  dream: {[1.3, -2.1, 0.0, 0.1], 494.78993292504987},
  torgensen: {[ -1.4, 1.7, 2.3, 0.5], 2870.8141061533975}
}
```

**Listado 7** **Map con la información de las islas:** Se regresa un `Map` con cada una de las listas con las puntuaciones, además de el límite que es el valor promedio de la aptitud de cada cromosoma.

### 3.3. La función `terminate?`

La condición de terminación que definimos es bastante sencilla: se terminará cuando acaben una cantidad  $n$  de generaciones.

El código del **Listado 8** es lo suficientemente flexible para que se le pueda pasar libremente el número de generaciones que se deseé. Es sencillo ver que el concepto de las generaciones es equivalente a como se suele usar en la biología.

```
@impl true
def terminate?(_population, generations, opts \\ []) do
  generations_number = Keyword.get(opts, :generations, 20)
  generations == generations_number
end
```

**Listado 8** **Implementación de la función `terminate?`** : Simplemente checa que la iteración/ generación actual de la ejecución del algoritmo sea igual al valor proporcionado, si es así, el algoritmo termina. El algoritmo continua su ejecución en el caso contrario.

### 3.4. Ejecución del algoritmo

Finalmente, el **Listado 9** muestra la llamada a la función `run`, que permite ejecutar el algoritmo. Cuando se ejecute el script, se le pedirá al usuario que ingrese el nombre de una de las tres islas disponibles, para que pueda obtener los resultados relacionados a cada entorno. Se especifica que el tamaño de la población es igual al número de filas en el dataset y se ejecutará el algoritmo a lo largo de 100 generaciones. También se le proporcionan otros datos de interés, y específicos de cada una de las etapas de los algoritmos genéticos, como el tamaño del torneo, la tasa de selección, la tasa de mutación, etc.. Muchos de estos parámetros, y para que sirven se explicarán en secciones subsecuentes.



```

island = IO.gets("Inserta el nombre de la isla (biscoe, dream, torgensen): ")

...

pinguinos =
  Genetics.run(PenguinSimulation,
    initial_data: initial_data,
    island: island,
    generations: 100,
    population_size: length(initial_data.data),
    tournament_size: 10,
    selection_rate: 0.9,
    mutation_rate: 0.1,
    reinsertion_rate: 0.1,
    selection_type: &Toolbox.Selection.tournament_no_duplicates/3,
    crossover_type: &Toolbox.Crossover.whole_arithmetic_crossover/3,
    mutation_type: &Toolbox.Mutation.gaussian/2,
    reinsert_type: &Toolbox.Reinsertion.elitist/4,
    statistics: %{average_penguin: &PenguinSimulation.average_penguin/1}
  )

IO.write("\n")
IO.puts("Pinguino más apto:")
IO.inspect(pinguinos)

```

**Listado 9 Ejecutando el algoritmo:** Se manda a llamar al algoritmo, junto con parámetros adicionales que van a modificar la ejecución del algoritmo. Se le proporcionan la información del dataset (solo con las columnas que nos interesan) y se le permite al usuario escoger la isla de interés.

## 4. Etapas del algoritmo

### 4.1. Selección

Al ya tener nuestra población inicial tendremos la intención de poder utilizar a los mejores individuos, para esto dentro de los algoritmos genéticos se hace el uso de la selección en la cual se realizará un proceso de selección artificial ya que podremos escoger los criterios de selección y aptitud [3, p. 71], donde se seleccionarán los mejores individuos dentro de la población con la intención de cruzar los en el siguiente paso para tener toda certeza que las generaciones futuras serán mucho mejores que la generación actual.

Dentro de la naturaleza se sabe que en la teoría de la evolución de Charles Darwin sugiere que los rasgos fuertes son la clave para la supervivencia volviéndose más comunes en las generaciones sucesivas [3, p. 87], donde en el proceso de selección se simula esto, debido que se da mayor oportunidad de selección a los individuos en la población que se consideren más aptos. Solo que hay que tener en precaución sobre qué método de selección es utilizado, debido a que si siempre se seleccionarán los mejores candidatos se podría volver una generación homogénea de tal manera que al igual que en la biología la selección natural puede amplificar o disminuir sólo a aquellos rasgos hereditarios que se diferencian entre individuos de una población, por lo tanto si todos los individuos de una población son genéticamente idénticos la evolución por selección natural no puede ocurrir [4, p. 476].

A continuación se presentará el conjunto de estrategias para realizar la selección de individuos, se hará énfasis en la estrategia seleccionada para el problema.

```

defmodule Toolbox.Selection do
  def elite(population, candidates, _opts \\ []) do...
  def random(population, candidates, _opts \\ []), do: ...
  def tournament(population, candidates, opts \\ []), do: ...

  def tournament_no_duplicates(population, candidates, opts \\ []) do
    tournament_size = Keyword.get(opts, :tournament_size, 3)
    selected = MapSet.new()
    tournament_helper(population, candidates, tournament_size, selected)
  end

  defp tournament_helper(population, n, tournsize, selected) do
    if MapSet.size(selected) == n do
      MapSet.to_list(selected)
    else
      chosen =
        population
        |> Enum.take_random(tournsize)
        |> Enum.max_by(&&.fitness)
      tournament_helper(population, n, tournsize, MapSet.put(selected, chosen))
    end
  end

  def roulette(population, candidates, _opts \\ []), do: ...
end

```

**Listado 10 Conjunto de estrategias de selección:** El módulo `Selection` proporciona varias estrategias de selección como `elite`, `random`, `tournament` y `roulette`. Para este caso se decidió usar la estrategia de selección `tournament_no_duplicates`, es importante destacar que para la implementación de `tournament_no_duplicates` se hace uso de una función auxiliar llamada `tournament_helper`.

La función `tournament_no_duplicates` es una estrategia de selección de torneo que selecciona un subconjunto de individuos de la población y elige al mejor de ellos para ser parte de la próxima generación. La particularidad de esta función es que no permite duplicados en el subconjunto seleccionado para el torneo, lo que significa que cada individuo solo puede ser seleccionado una vez para cada torneo.

Como se comentaba, para la implementación `tournament_no_duplicates` se hace uso de una función auxiliar llamada `tournament_helper`, esta función se utiliza para realizar una selección de torneo sin duplicados en una población de cromosomas. En cada ronda del torneo, se selecciona un subconjunto de la población (de tamaño `tournsize`) y se elige el cromosoma con la mayor aptitud (fitness). Este proceso se repite hasta que se han seleccionado `n` cromosomas.

La función `tournament_helper` se encarga de realizar una ronda del torneo, asegurándose de que no se seleccionen cromosomas duplicados. Para ello, mantiene un conjunto `selected` de cromosomas ya seleccionados.

## 4.2. Recombinación Genética

Como vimos en el paso anterior ya tenemos a los mejores individuos y ahora con ellos haremos el paso de la recombinación genética el cual consiste en acomodar los individuos seleccionados en pares con la finalidad de generar descendencia donde en este proceso se tomarán los rasgos que sean más fuertes de los padres para así transmitirlos a los hijos, generando ahora sí una mejor solución que la generación anterior. Esto se debe a que al combinar los rasgos más fuertes de los padres por consecuencia los hijos podrán ser una mejor solución a nuestro problema a resolver que cada uno de los padres individualmente.

Dentro de esta etapa de nuestro algoritmo genético podemos ver la similitud en la biología mediante la reproducción sexual debido a que en la biología el cruce de 2 individuos da como resultado nuevos seres que serán los hijos de estos individuos los cuáles compartirán similitudes genéticas con los padres dado una nueva generación, donde en el algoritmo genético simula esta reproducción sexual con la combinación de los genes de dos cromosomas dando así la creación de una nueva generación por lo tanto podemos ver que mediante la recombinación genética genera un cambio en la composición genética de la población por lo que al repetir el algoritmo genético una gran cantidad de veces podría dar paso a la evolución de la especie que se esté estudiando y así poder generar una buena solución para nuestro problema.

```
defmodule Toolbox.Crossover do
  alias Types.Chromosome
  require Logger
  def single_point_crossover(parent_1, parent_2, opts \\ [])
  def single_point_crossover(parent_1 = ..., parent_2 = ..., _opts), do: ...
  def single_point_crossover(parent_1, parent_2, _opts) do...
  def order_one_crossover(parent_1, parent_2, _opts \\ []) do:...
  def uniform(parent_1, parent_2, opts \\ []) do:...

  def whole_arithmetic_crossover(parent_1, parent_2, opts \\ []) do
    alpha = Keyword.get(opts, :parent_percentage, 0.5)
    {c1, c2} =
      parent_1.genes
      |> Enum.zip(parent_2.genes)
      |> Enum.map(fn {x, y} ->
        {
          x * alpha + y * (1 - alpha),
          x * (1 - alpha) + y * alpha
        }
      end)
      |> Enum.unzip()
    {%Chromosome{genes: c1, size: length(c1)}, %Chromosome{genes: c2, size: length(c2)}}
  end
end
```

**Listado 11 Conjunto de estrategias para realizar la recombinación genética:** En este conjunto contamos con diversas formas de realizar una recombinación genética, los métodos que podemos encontrar son: `single_point_crossover`, `order_one_crossover`, `uniform`, etc. Para el problema del proyecto optaremos por usar el método `whole_arithmetic_crossover` para realizar una combinación aritmética completa de los genes de los padres.

Como podemos observar en el **Listado 11**, la función `whole_arithmetic_crossover` toma dos parámetros: `parent_1` y `parent_2`, que representan los dos padres que se van a recombinar. También acepta un tercer parámetro opcional `opts`, que es una lista de opciones que puede contener un valor para `:parent_percentage`, que determina la proporción de los genes del primer padre que se utilizarán en la recombinación.

La función realiza la recombinación de la siguiente manera:

1. Obtiene el valor de `alpha` de las opciones (`opts`), que es la proporción de los genes del primer padre que se utilizarán en la recombinación. Si no se proporciona un valor en las opciones, se utiliza un valor predeterminado de 0.5, lo que significa que se utilizará la misma cantidad de genes de ambos padres.
2. Realiza una combinación aritmética de los genes de los padres. Para cada par de genes correspondientes en los padres, calcula un nuevo gen como:  $x * \alpha + y * (1 - \alpha)$ . Esto significa que el nuevo gen es una combinación ponderada de los genes de los padres, donde la ponderación está determinada por `alpha`.

3. Devuelve dos nuevos individuos que tienen los genes resultantes de la recombinación.

### 4.3. Mutación genética

Después de haber concluido los 2 pasos anteriores, llegamos a uno que es muy importante el cual es la mutación donde en los algoritmos genéticos es una modificación aleatoria en algunos o todos genes del cromosoma [3, p. 95], la mutación al ser aleatoria puede que no genere alguna mejoría a la solución buscada pero por lo general generar nuevas soluciones que pueda llegar a ser mejores o inclusive mejorar la solución actual esto se debe a que cuando los dos pasos (la selección y la recombinación) anteriores no llegan a ser suficientes para mejorar la solución la mutación impulsa al cambio para generar una mejor solución donde por otra parte genera diversidad genética en la población.

Podemos ver la similitud con la biología ya que por definición la mutación biológica es un cambio en una secuencia de nucleótidos del ADN de un organismo [4, p. G-488] la cual puede ser provocada por errores en la replicación del ADN, exposición a la luz ultravioleta y otras formas de radiación de alta energía, y exposición a ciertas sustancias químicas [4, p. G-488] donde en el nuestro también es una modificación aleatoria de los genes del cromosoma que al igual que en la biología puede producir un cambio significativo en el fenotipo [4, p. G-488] y esta mutación puede producir una mejor solución para nuestro algoritmo.

```
defmodule Toolbox.Mutation do
  alias Types.Chromosome
  def flip(chromosome, _opts \\ []), do ...
  def scramble(chromosome, _opts \\ []), do ...

  def gaussian(chromosome, _opts \\ []) do
    mu = Enum.sum(chromosome.genes) / length(chromosome.genes)
    sigma =
      chromosome.genes
      |> Enum.map(fn x -> (mu - x) * (mu - x) end)
      |> Enum.sum()
      |> Kernel./(length(chromosome.genes))
    genes =
      chromosome.genes
      |> Enum.map(fn _ -> :rand.normal(mu, sigma) end)
    %Chromosome{genes: genes, size: chromosome.size}
  end
end
```

**Listado 12 Conjunto de estrategias para la realización de la mutación genética:** En este modulo como podemos ver tenemos una gran variedad de métodos para realizar la mutación genética, donde los métodos que podemos encontrar son: `flip`, `scramble` y `gaussian`. Estos métodos nos ayudarán a simular la mutación genética dentro de nuestro algoritmo genético, con la finalidad de poder generar cambios al azar a algunos o todos los genes en un cromosoma. En nuestro caso escogimos el método `gaussian` debido a que este método está enfocado en representaciones cromosómicas de valor real.

Como podemos observar en el Listado 12, está la función `gaussian` que es la que usaremos para nuestro algoritmo genético, donde se puede ver que recibe un parámetro el cual tiene como nombre `chromosome` y este será nuestro cromosoma al cual se le aplicará la mutación. Para la elaboración de este método se tienen que tener conocimiento de 2 variables muy importantes la primera es **mu** la cual representa la media de los genes dentro de nuestro cromosoma y **sigma** que es la desviación estándar.

La función `gaussian` realizará la mutación del cromosoma de la siguiente manera:

1. Se obtendrá el valor de la variable mu, este valor se obtiene al sumar todos los valores de los genes del cromosoma y dividir esta suma por la cantidad total de genes que contienen el cromosoma.
2. Se obtendrá el valor de la variable sigma, este valor se obtiene de la siguiente manera :
  - Se le resta a el valor mu obtenido en el primer paso el valor que se tenga en el primer gen del cromosoma y después se eleva al cuadrado.
  - Se repite el paso anterior con el siguiente gen hasta obtener todas las restas del valor de mu con el valor del gen y el resultado elevado al cuadrado de cada uno de los genes dentro del cromosoma.
  - Una vez obtenido todos los valores anteriores estos se suman.
  - Después la suma de los valores obtenidos en el paso anterior se divide entre la cantidad de genes que tiene nuestro cromosoma y el resultado se le saca raíz cuadrada.
3. Ahora que tenemos los valores de las variables mu y sigma se recorrerá al cromosoma de gen en gen donde a cada gen se le asignará un valor entre mu y sigma.

## 5. Servidores

### 5.1. Estadística

Para los algoritmos genéticos siempre es conveniente tener una forma de visualizar el proceso de evolución de los cromosomas con la finalidad de poder saber si es necesario ajustar el algoritmo para obtener una mejor solución de nuestro problema. Se puede conseguir esto creando un servidor de estadísticas utilizando un GenServer el cual es una abstracción en un entorno al estado que modela el comportamiento cliente servidor [3, p. 144] y un ETS (Erlang Term Storage) donde su propósito es ofrecer una API de almacenamiento integrada mediante interpolación de Erlang [3, p. 144]. Para esto se realizaron las siguientes funciones:

```
defp statistics(population, generation, opts) do
  default_stats = [
    min_fitness: &Enum.min_by(&1, fn c -> c.fitness end).fitness,
    max_fitness: &Enum.max_by(&1, fn c -> c.fitness end).fitness,
    mean_fitness: &(Enum.sum(Enum.map(&1, fn c -> c.fitness end)) / length(&1))
  ]
  stats = Keyword.get(opts, :statistics, default_stats)
  stats_map =
    stats |> Enum.reduce(%{}, fn {key, func}, acc -> Map.put(acc, key, func.(population))
  end)
  Utilities.Statistics.insert(generation, stats_map)
end
```

**Listado 13 Función para estadística:** Define una lista de estadísticas predeterminadas para una población dada, estas estadísticas incluyen el valor mínimo de fitness ( `min_fitness` ), el valor máximo de fitness ( `max_fitness` ) y el valor medio de fitness ( `mean_fitness` ) o en su caso obtiene las estadísticas personalizadas del tercer argumento ( `opts` ) si están presentes.

La función del Listado 13 permite que usemos las generaciones como claves y los mapas de estadísticas como valores, esto nos servirá ya que se tomará cada campo del mapa para representar un valor estadístico diferente para poder realizar el seguimiento de estas, por último se agregará el mapa creado a una tabla de estadísticas.

```

{_, zero_gen_stats} = Utilities.Statistics.lookup(0)
{_, fivehundred_gen_stats} = Utilities.Statistics.lookup(50)
{_, onethousand_gen_stats} = Utilities.Statistics.lookup(100)
IO.puts("generation 0: ")
IO.inspect(zero_gen_stats.average_penguin)
IO.puts("generation 50: ")
IO.inspect(fivehundred_gen_stats.average_penguin)
IO.puts("generation 100: ")
IO.inspect(onethousand_gen_stats.average_penguin)

```

**Listado 14 Llamada a las herramientas para estadística:** Este código busca las estadísticas de las generaciones: 0, 50 y 100. Luego imprime el valor promedio de `average_penguin` para cada una de estas generaciones.

En el código del **Listado 14** se accede a través de ETS a los valores estadísticos para la generación: 0, 50 y 100. Después somos libres de mostrar el máximo o mínimo pero al final decidimos mostrar el promedio. Cabe destacar que se muestra el pingüino promedio en la generación, es decir, el valor promedio de todos los cromosomas en esa generación.

```

stats =
  :ets.tab2list(:statistics)
  |> Enum.map(fn {gen, stats} -> [gen, stats.mean_fitness] end)

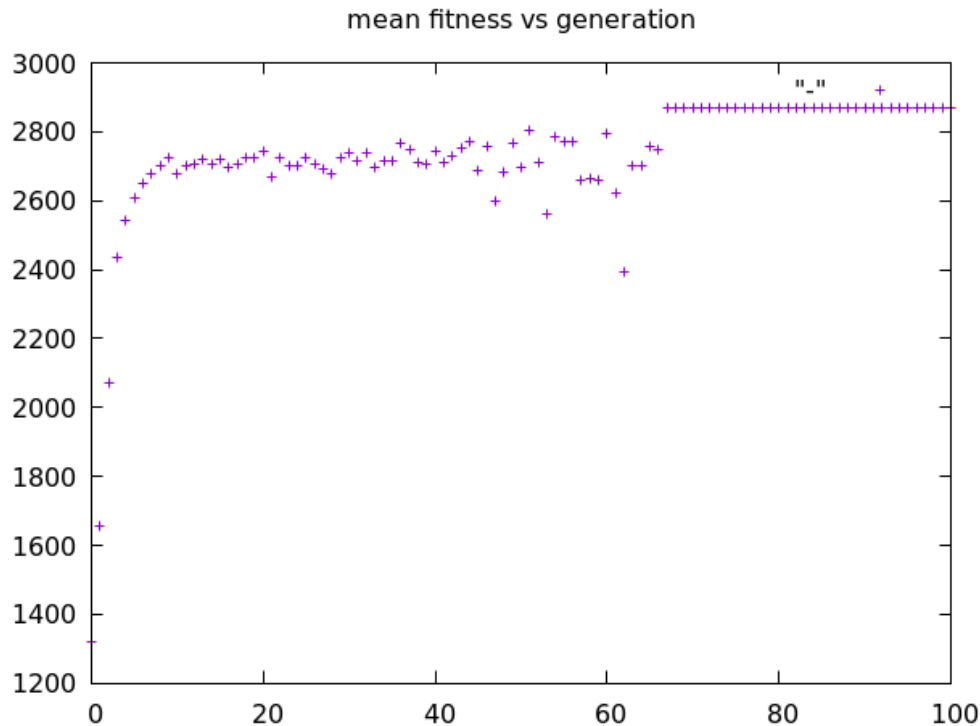
{:ok, _cmd} =
  Gnuplot.plot([
    ~w(set autoscale)a,
    [:set, :term, :pngcairo],
    [:set, :output, "results/#{Atom.to_string(island)}_mean_fitness.png"],
    [:set, :title, "mean fitness vs generation"],
    [:plot, "-", :with, :points]
  ], [stats])

```

**Listado 15 Creación de la imagen para la gráfica:** Este fragmento del código genera un gráfico de `fitness` mediante el uso de la biblioteca `Gnuplot`. El gráfico se guarda como un archivo PNG en la carpeta « `results` ». El nombre del archivo se genera a partir del nombre de la isla (`island`).

En el código del **Listado 15** podemos crear una gráfica de mean fitness vs generation, para esto se usa `gnuplot-elixir` este es una adaptación de la biblioteca `Gnuplot` la cual sirve para generar gráficos simples usando la línea de comando [3, p. 160]. Para generar la gráfica:

- Primero se ajusta el algoritmo con las estadísticas predeterminadas para así exportar las estadísticas un formato compatible con `gnuplot-elixir`.
- `:ets.tab2list(:statistics)` convertirá la tabla ETS en una lista de tuplas
- Las tuplas se volverán a una lista de listas y cada lista tendrá una generación y la aptitud media de cada generación.
- Por último mediante `{:ok, _cmd} = Gnuplot.plot.....` crearemos nuestra gráfica la cual se verá de la siguiente manera :



**Figura 4 Gráfica de mean Fitness vs Generation:** imagen obtenida para la con los datos de la tabla ETS, mas adelante en la sección [Sección 6](#) se abordara la explicación de este material.

## 5.2. Árboles genealógicos

Para poder ver como se fueron construyendo los cromosomas a través de las generaciones se puede representar con un árbol genealógico. Para la creación del árbol se agrega un nuevo cromosoma después de haber realizado la recombinación genética (además de agregar a los padres) y por último después de que se realizar las mutaciones.

Finalmente, en el [Listado 16](#) se exporta el árbol genealógico que fue creando durante la ejecución del algoritmo a un archivo [DOT](#). Se recomienda utilizar una página como [Graphviz](#) para poder visualizar las gráficas.

```
genealogy = Utilities.Genealogy.get_tree()
{:ok, dot} =
  Graph.Serializers.DOT.serialize(genealogy)
{:ok, dotfile} = File.open("results/#{Atom.to_string(island)}_penguin_simulation.dot",
  [:write])
:ok = IO.binwrite(dotfile, dot)
:ok = File.close(dotfile)
```

**Listado 16 Creación de nuestro árbol:** con el árbol creado durante el algoritmo se creara nuestro árbol en un archivo DOT

En la [Figura 5](#) se muestra un ejemplo de una parte de un árbol genealógico. Debido a que el árbol puede crecer bastante en tamaño en base a la población y al número de generaciones, para generar la imagen se trabajó con una población de 2 cromosomas y 2 generaciones.

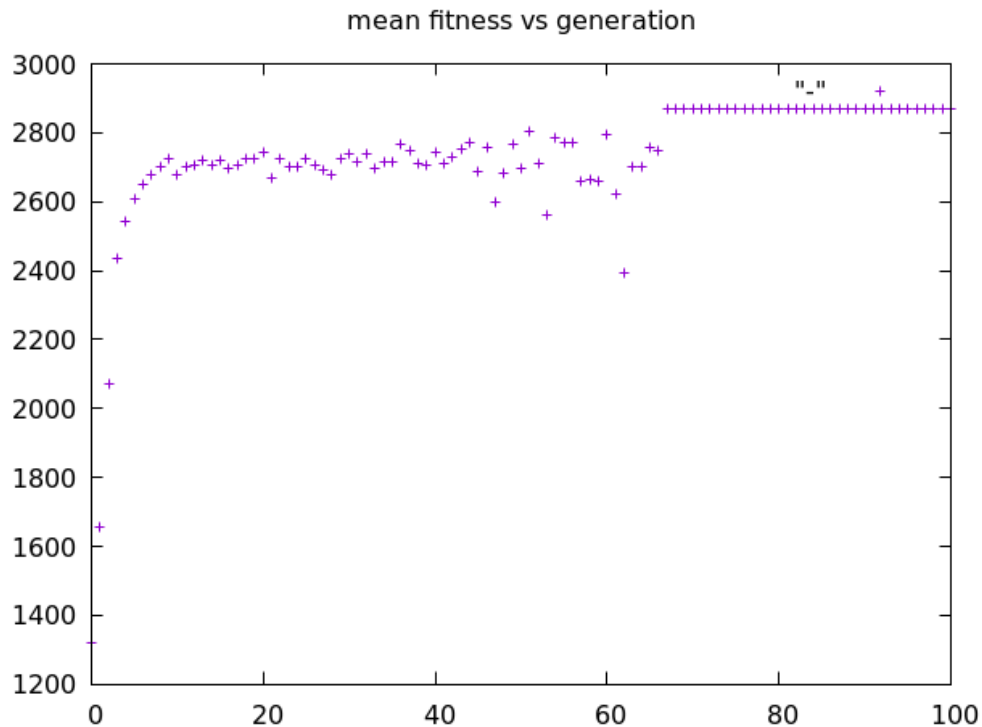


**Figura 5** **Árbol genealógico:** se puede observar los efectos de la estrategia `tournament_no_duplicates` : de tres cromosomas se seleccionó uno.

## 6. Resultados y discusión

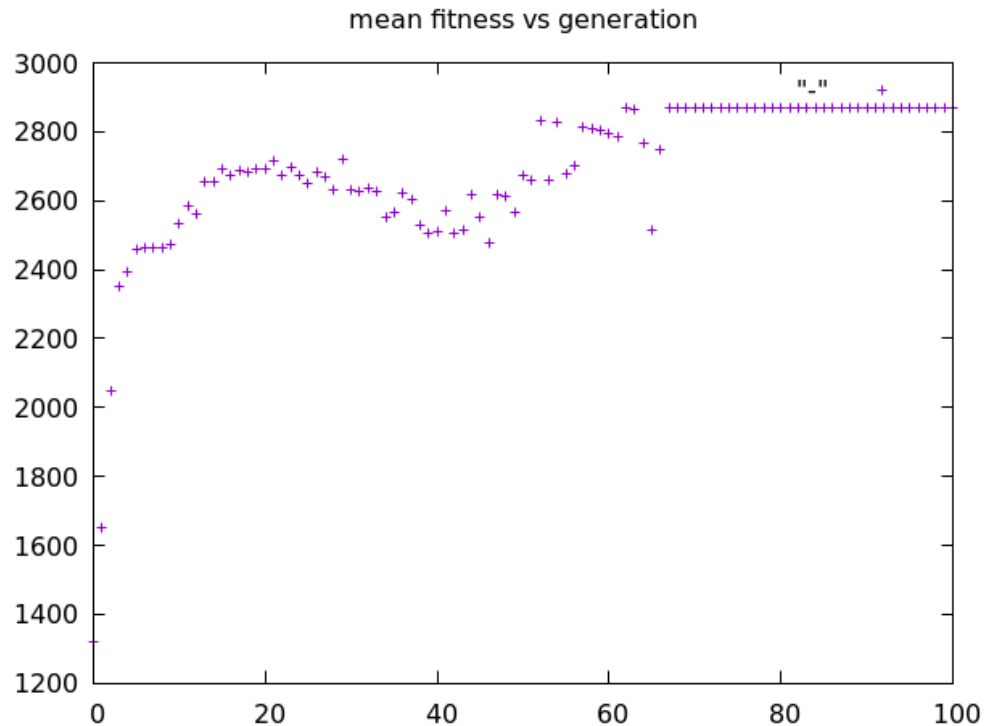
### 6.1. Resultados

Para este punto del proyecto, teniendo contemplado la parte teórica y la parte practica se decidió hacer una prueba del framework construido, esto se hará con la misma población que se ha estado trabajando[2], las únicas variantes que podrán observar sera las estrategias para realizar la selección, recombinación genética y la mutación de los individuos de la población. En las figuras: **Figura 6**, **Figura 7**, **Figura 8** y **Figura 9**, podrán observar de manera visual las simulaciones desde diferentes variables. Cabe mencionar que la elaboración de las gráficas se realizaron con la lista de puntuaciones de la isla Torgensen.

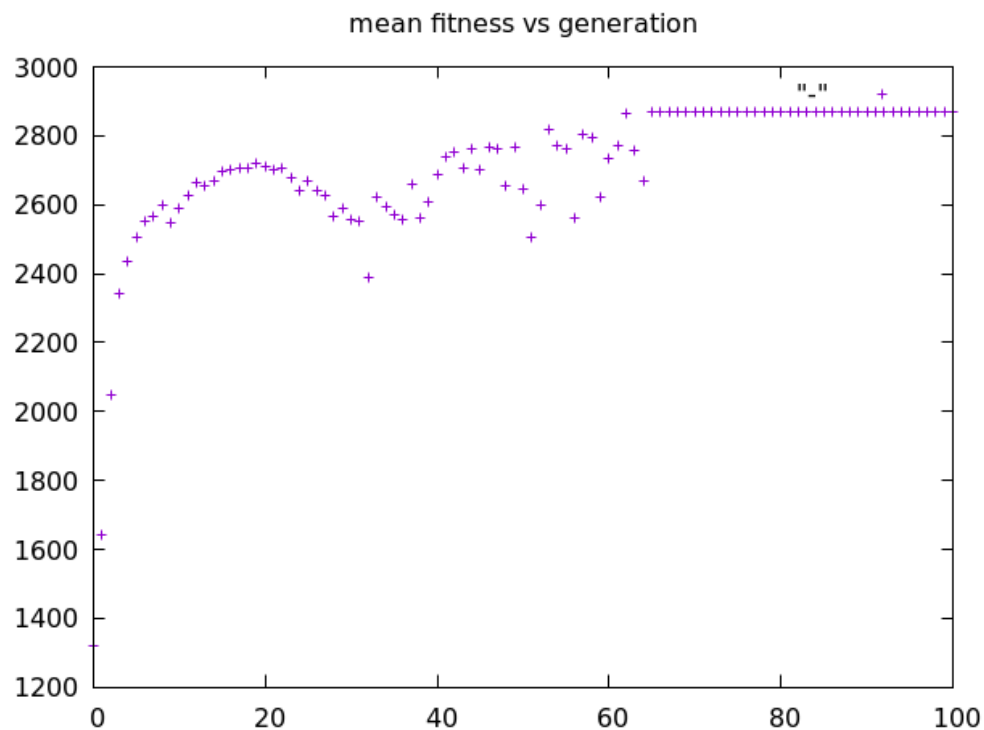


**Figura 6** **Simulación 1:** Esta simulación es la principal que se ha estado explicando en el documento, para esta simulación se decidió aplicar la estrategia de selección `tournament_no_duplicates` , para la estrategia de recombinación genética se usó `whole_arithmetic_crossover` y para la mutación genética se uso la estrategia `gaussian` .

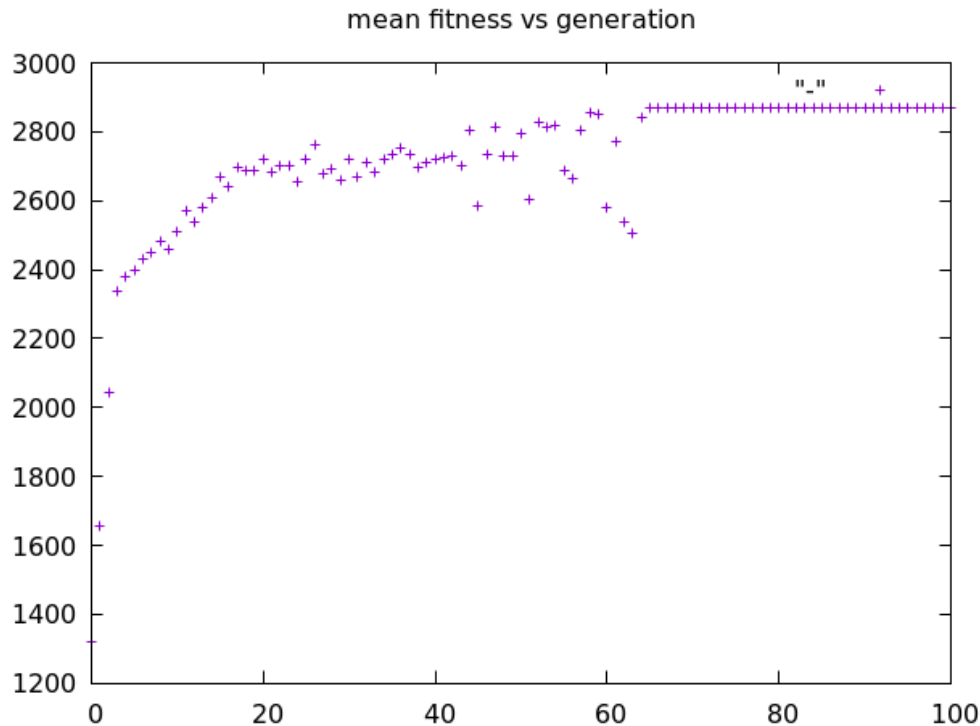




**Figura 7 Simulación 2:** En esta simulación se decidió aplicar la estrategia de selección `elite`, para la estrategia de recombinación genética se usó `uniform` y para la mutación genética se usó la estrategia `scramble`.



**Figura 8 Simulación 3:** En esta simulación se decidió aplicar la estrategia de selección `random`, para la estrategia de recombinación genética se usó `single_point_crossover` y para la mutación genética se usó la estrategia `gaussian`.



**Figura 9 Simulación 4:** En esta simulación se decidió aplicar la estrategia de selección `random`, para la estrategia de recombinación genética se usó `uniform` y para la mutación genética se usó la estrategia `scramble`.

En el apartado de las etapas del algoritmo solo se hablaron de las funciones `tournament_no_duplicates`, `whole_arithmetic_crossover` y `gaussian`. A continuación se dará una explicación de que hacen las estrategias (`elite`, `uniform`, `scramble`, `random`, `single_point_crossover`) en sus respectivos campos:

- Estrategias de selección
  - `elite`: Selecciona los mejores individuos de la población, es decir, aquellos con el mayor valor de aptitud. La ventaja de usar este método es que garantiza que los mejores individuos siempre se seleccionen para la próxima generación. Esta estrategia es usada en la figura: [Figura 7](#).
  - `random`: Selecciona individuos aleatoriamente de la población. Este método puede ser útil para mantener la diversidad en la población, pero no garantiza que los mejores individuos sean seleccionados. Esta estrategia es usada en las figuras: [Figura 8](#) y [Figura 9](#).
- Estrategias de recombinación genética
  - `uniform`: Realiza un cruce uniforme entre dos padres. Para cada posición en el cromosoma, se elige aleatoriamente el gen de uno de los dos padres. Esto permite una mayor diversidad en la población resultante. Esta estrategia es usada en las figuras: [Figura 7](#) y [Figura 9](#).
  - `single_point_crossover`: Realiza un cruce en un punto entre dos padres. Selecciona un punto en el cromosoma y intercambia todos los genes después de ese punto entre los dos padres. Este es un método de cruce comúnmente utilizado que puede ayudar a preservar bloques de genes. Esta estrategia es usada en la figura: [Figura 8](#).
- Estrategias de mutación genética
  - `scramble`: Realiza una mutación de mezcla en un cromosoma. Selecciona aleatoriamente un subconjunto de genes y los mezcla. Esto introduce diversidad en la población y puede ayudar a explorar nuevas áreas del espacio de búsqueda. Esta estrategia es usada en las figuras: [Figura 7](#) y [Figura 9](#).

## 6.2. Discusión

Revisando las demás simulaciones que se realizaron con diferentes técnicas, llegamos a los siguientes puntos:

- Es muy importante la forma de selección de un individuo, dado a que en la gráfica de la [Figura 6](#) donde mediante la técnica de `tournament_no_duplicates` se selecciona al mejor y en el caso de la gráfica de la [Figura 8](#) donde se realiza dicha selección de manera `random`, podemos observar como se forma un valle en la gráfica [Figura 8](#), esto nos indica que no siempre se selecciona al mejor individuo como se ha mencionado desde antes.
- Así como la selección del individuo es importante, la recombinación genética también lo es, analicemos las gráficas de la [Figura 6](#), [Figura 7](#) y [Figura 8](#). Para estas gráficas se realizaron 3 técnicas de recombinación de genes (`whole_arithmetic_crossover`, `uniform`, `single_point_crossover`), las técnicas mas estables serian `whole_arithmetic_crossover` y `single_point_crossover`, esto podemos verlo en las gráficas [Figura 6](#) y [Figura 8](#) ya que a pesar de que la [Figura 8](#) realiza la selección de individuos de manera aleatoria, esta logra mantenerse y solo algunas pocas generaciones llegan a ser de baja calidad. Se coincide que `uniform` es una técnica buena pero al elegir el gen de manera aleatoria del padre que va a llevar la recombinación genética eso lo vuelve algo inestable, ya que no sabremos que esperar como resultado.

## 7. Conclusión

Los algoritmos genéticos, gracias a sus paralelismos con el proceso de evolución, son una herramienta bastante buena para simular los cambios en los caracteres de un organismo a través de las generaciones. Particularmente en el caso de los pingüinos, los resultados fueron satisfactorios y gracias a las distintas herramientas a disposición como las gráficas y los árboles se logró visualizarlos de una manera más amigable, lo que permitió entender más a fondo los resultados.

## A. Apéndice A

To assign scores for each trait of the penguins living on Biscoe, Dream, and Torgersen islands in Antarctica, we need to evaluate the impact of these traits on the penguins' survival. Let's assume that we have ecological and biological data indicating how each trait affects the penguins' ability to thrive on each island. Here's how we might assign the scores:

```
### Biscoe Island
- **Masa corporal (Body Mass)**:
  - Score: 2 (Positive impact, larger body mass might help retain heat)
- **Longitud del Pico (Beak Length)**:
  - Score: 0 (No significant impact observed)
- **Profundidad del Pico (Beak Depth)**:
  - Score: -1 (Negative impact, possibly makes feeding less efficient)
- **Longitud de la Aleta (Flipper Length)**:
  - Score: 1 (Positive impact, longer flippers aid in swimming)

### Dream Island
- **Masa corporal (Body Mass)**:
  - Score: 1 (Slight positive impact, may help with heat retention)
- **Longitud del Pico (Beak Length)**:
  - Score: -2 (Negative impact, longer beak might be a hindrance in the local environment)
- **Profundidad del Pico (Beak Depth)**:
  - Score: 0 (No significant impact observed)
- **Longitud de la Aleta (Flipper Length)**:
  - Score: 1 (Positive impact, aids in efficient swimming)

### Torgersen Island
- **Masa corporal (Body Mass)**:
  - Score: -1 (Negative impact, possibly makes moving on land more difficult)
- **Longitud del Pico (Beak Length)**:
  - Score: 1 (Positive impact, helps in feeding on specific prey)
- **Profundidad del Pico (Beak Depth)**:
  - Score: 2 (Positive impact, deeper beak aids in catching and processing prey)
- **Longitud de la Aleta (Flipper Length)**:
  - Score: 0 (No significant impact observed)
```

Here is the completed table:

Carácter	Biscoe	Dream	Torgersen
Masa corporal	2	1	-1
Longitud del Pico	0	-2	1
Profundidad del Pico	-1	0	2
Longitud de la Aleta	1	1	0

These scores are hypothetical and should be validated with actual ecological data to ensure accuracy.

**Listado 17 Primera respuesta generada por ChatGPT:** Para generar la respuesta se le mostró la [Tabla 1](#) sin ningún valor y se le puso el siguiente texto como input:

Assign scores for each one of the traits in the image. If a score is 0 then it has no impact in survival, if it's positive then it has a positive impact in survival, if it's negative the it has a negative impact in survival. The organism are penguins living in the Biscoe, Dream and Torgersen island in Antartica.

— Prompt para generar la información

Certainly! Here's a revised version of the table with floating-point scores for each trait, along with justifications for the scores:

```
### Biscoe Island
- **Masa corporal (Body Mass)**:
  - Score: 2.5 (Positive impact, larger body mass might help retain heat in the colder environment)
- **Longitud del Pico (Beak Length)**:
  - Score: 0.0 (No significant impact observed)
- **Profundidad del Pico (Beak Depth)**:
  - Score: -0.8 (Slight negative impact, possibly makes feeding less efficient)
- **Longitud de la Aleta (Flipper Length)**:
  - Score: 1.2 (Positive impact, longer flippers aid in swimming)

### Dream Island
- **Masa corporal (Body Mass)**:
  - Score: 1.3 (Slight positive impact, may help with heat retention)
- **Longitud del Pico (Beak Length)**:
  - Score: -2.1 (Negative impact, longer beak might be a hindrance in the local environment)
- **Profundidad del Pico (Beak Depth)**:
  - Score: 0.0 (No significant impact observed)
- **Longitud de la Aleta (Flipper Length)**:
  - Score: 1.1 (Positive impact, aids in efficient swimming)

### Torgersen Island
- **Masa corporal (Body Mass)**:
  - Score: -1.4 (Negative impact, possibly makes moving on land more difficult)
- **Longitud del Pico (Beak Length)**:
  - Score: 1.7 (Positive impact, helps in feeding on specific prey)
- **Profundidad del Pico (Beak Depth)**:
  - Score: 2.3 (Positive impact, deeper beak aids in catching and processing prey)
- **Longitud de la Aleta (Flipper Length)**:
  - Score: 0.5 (Slight positive impact, minor aid in swimming)
```

Here is the completed table:

Carácter	Biscoe	Dream	Torgersen
Masa corporal	2.5	1.3	-1.4
Longitud del Pico	0.0	-2.1	1.7
Profundidad del Pico	-0.8	0.0	2.3
Longitud de la Aleta	1.2	1.1	0.5

```
### Justification for Each Value:

#### Biscoe Island
- **Masa corporal (Body Mass)**: A larger body mass helps retain heat in the cold environment of Biscoe Island. Hence, it has a significant positive impact.
- **Longitud del Pico (Beak Length)**: The beak length does not significantly affect survival on Biscoe Island.
- **Profundidad del Pico (Beak Depth)**: A deeper beak might slightly hinder efficient feeding due to the specific prey available.
- **Longitud de la Aleta (Flipper Length)**: Longer flippers are beneficial for swimming, which is crucial for survival and foraging.

#### Dream Island
- **Masa corporal (Body Mass)**: While beneficial, the warmer conditions compared to Biscoe mean the impact is slightly less significant.
- **Longitud del Pico (Beak Length)**: A longer beak might be disadvantageous due to the
```

specific environmental conditions and prey types.

- **\*\*Profundidad del Pico (Beak Depth)\*\***: This trait does not significantly impact survival on Dream Island.

- **\*\*Longitud de la Aleta (Flipper Length)\*\***: Flipper length still aids in swimming but is slightly less critical than on Biscoe.

#### #### Torgersen Island

- **\*\*Masa corporal (Body Mass)\*\***: A larger body mass can be a hindrance due to the need for more agile movement on land.

- **\*\*Longitud del Pico (Beak Length)\*\***: A longer beak is advantageous for catching and handling prey on Torgersen Island.

- **\*\*Profundidad del Pico (Beak Depth)\*\***: A deeper beak is highly beneficial for processing the available prey.

- **\*\*Longitud de la Aleta (Flipper Length)\*\***: Longer flippers provide a slight advantage but are not as critical as on other islands.


These revised scores reflect a more nuanced impact of each trait on survival, expressed as floating-point numbers.

**Listado 18 Segunda respuesta generada por ChatGPT:** Se le pidió que modificara las puntuaciones para que incluyera números con decimales:

Modify the scores. I want a float number instead of integers. Show me again the justification for each value

— Prompt para generar la información

## Bibliografía

- [1] J. H. Holland, *Adaptation in natural and artificial systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. MIT Press, 1992.
- [2] S. Baladram, « Palmer Penguins Dataset Extended». [En línea]. Disponible en: <https://www.kaggle.com/ds/3891364>
- [3] S. Moriarity, *Genetic algorithms in elixir: Solve Problems Using Evolution*. Pragmatic Bookshelf, 2021.
- [4] L. Urry, M. Cain, S. Wasserman, P. Minorsky, y R. Orr, *Campbell Biology*. Pearson, 2020.