

genetics.js

Framework web de computación evolutiva

Cristian Manuel Abrante Dorta

Universidad de La Laguna

21 de junio de 2019

Escuela Superior de Ingeniería y Tecnología
Universidad de La Laguna

1 Introducción

- \mathcal{P} vs \mathcal{NP}
- Optimización combinatoria
- Metaheurísticas

2 Computación evolutiva

- Definición
- Historia
- Estructura

3 Objetivos

- Bibliotecas de algoritmos evolutivos
- Objetivos
- Roadmap

4 Desarrollo

- Tecnologías utilizadas
 - Tecnologías utilizadas en el desarrollo
 - Tecnologías utilizadas en producción
- Estructura del software

5 Ejemplo de uso

6 Conclusions and future lines

- 1 **Introducción**
- 2 Computación evolutiva
- 3 Objetivos
- 4 Desarrollo
- 5 Ejemplo de uso
- 6 Conclusions and future lines

En primer lugar, veremos una clasificación de los problemas en función de su complejidad computacional.

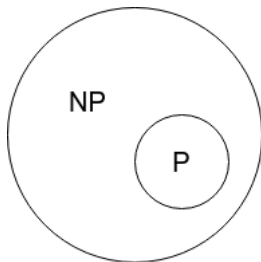


Figura: Representación de las clases de problemas \mathcal{P} y \mathcal{NP}

- **Clase \mathcal{P} :** Se puede **encontrar** una solución en tiempo polinomial.
- **Clase \mathcal{NP} :** Se puede **verificar** una solución en tiempo polinomial.

Optimización combinatoria

Dentro de la clase \mathcal{NP} , existen una gran cantidad de problemas de **optimización combinatoria**.

Optimización combinatoria

En este tipo de problemas, se trata de optimizar una función objetivo, que depende de una serie de variables **discretas**, sujetas a un conjunto de restricciones.

Algunos ejemplos de este tipo de problemas:

- Problema del viajante de comercio (TSP).
- Problema de rutas de vehículos (VRP).
- Problema del vertex cover.

Formalmente, un problema (P) de optimización combinatoria se puede definir como una tupla:

$$P = (S, f, \Omega)$$

Donde:

- **Espacio de soluciones** (S): Es el conjunto finito y numerable de todas las posibles soluciones al problema.
- **Función objetivo** (f): Para cada solución de S , devuelve su puntuación.

$$f : S \rightarrow \mathbb{R}$$

- **Conjunto de restricciones** (Ω): Conjunto de restricciones que debe satisfacer una solución de s ($s \in S$), para ser válida.

Optimización combinatoria

El concepto de **óptimo global** (s^*) es la solución para la cual la función objetivo tiene una evaluación más alta, en el caso de un problema de **maximización**:

$$\forall s \in S, \quad f(s^*) \geq f(s)$$

O mas baja en el caso de un problema de **minimización**:

$$\forall s \in S, \quad f(s^*) \leq f(s)$$

Para encontrar esta solución, existen varias opciones:

- Enumerar todas las soluciones de S . Esto suele ser **inviable** debido al gran tamaño de este conjunto.
- Utilizar una **exploración inteligente** del espacio de soluciones. Utilizando por ejemplo, **metaheurísticas**.

Metaheurísticas

Las metaheurísticas se centran en encontrar soluciones a problemas de optimización aplicando una búsqueda en el espacio de soluciones (S), cuando no se tiene información específica del problema.

Existen muchas técnicas metaheurísticas, que se pueden clasificar según diferentes criterios, sin embargo en este trabajo nos centraremos en la **computación evolutiva**.

- 1 Introducción
- 2 Computación evolutiva
- 3 Objetivos
- 4 Desarrollo
- 5 Ejemplo de uso
- 6 Conclusions and future lines

“La gran ventaja de la evolución es la gran cantidad de especies diferentes que ha creado, cada una adaptada a su medio”

— A.E. Eiben y J.E. Smith

Computación evolutiva

La **computación evolutiva** es una técnica metaheurística, bio-inpirada y basada en población. Utilizada para resolver diversos problemas complejos, y especialmente útil a la hora de resolver problemas de **optimización combinatoria**.

La inspiración principal de la computación evolutiva es la **Teoría de la evolución de Darwin**. En la cual se exponen conceptos clave como la **selección natural** o la **supervivencia de los individuos más adaptados**.

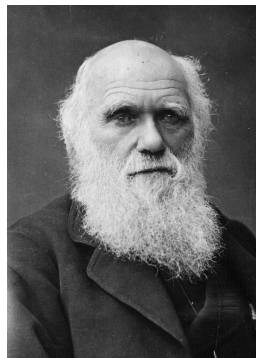


Figura: Charles Darwin

El desarrollo de la computación evolutiva se produce a partir de los **años sesenta**:

- 1962 Bremermann ejecuta el primer experimento sobre **Optimización mediante evolución y recombinación**.
- 1965 Fogel, Owen y Walsh introducen el término de **Programación evolutiva**.
- 1973 Holland desarrolla los **algoritmos genéticos**.
- 1973 Rechenberg y Schwefel crean las **estrategias evolutivas**.
- 1990 Se unifican todos los conceptos, definiendo el término **computación evolutiva**

La estructura de un algoritmo evolutivo es la siguiente:

Algorithm 1 Esquema básico de un algoritmo evolutivo

INICIALIZAR la población con n individuos aleatorios;

EVALUACIÓN de la población mediante la función de puntuación (*fitness*);

while *CONDICIÓN DE PARADA no sea satisfecha* **do**

 SELECCIÓN de padres;

 RECOMBINACIÓN de pares de padres;

 MUTACIÓN de la descendencia;

 EVALUACIÓN de la descendencia;

 SELECCIÓN de supervivientes para la siguiente generación;

end

Los algoritmos evolutivos permiten realizar una exploración inteligente del espacio de soluciones (S) del problema gracias a:

- **Variación:** Las fases de mutación y recombinación garantizan la diversidad en las soluciones.
- **Intensificación:** Las fases de selección de padres y descendencia, garantizan que se evalúen las mejores soluciones.

- 1 Introducción
- 2 Computación evolutiva
- 3 **Objetivos**
- 4 Desarrollo
- 5 Ejemplo de uso
- 6 Conclusions and future lines

Existen numerosas bibliotecas de algoritmos evolutivos, que contienen una implementación de los métodos más comunes existentes en cada fase.

- **Desarrolladas en Java:** Opt4J, Optimization Algorithm Toolkit, JMetal, ...
- **Desarrolladas en C++:** ParadisEO, METCO,...

Se ha identificado la carencia de un *framework* de este propósito, pero **orientado a aplicaciones web**.

Desarrollo

El objetivo de este proyecto es el desarrollo de **genetics.js**, un *framework* que contenga los principales métodos que cubran todas las fases de un algoritmo evolutivo.

Además debe cumplir los siguientes criterios:

- Garantizar la compatibilidad con aplicaciones web.
- Estar implementado en un lenguaje de programación moderno y soportado por la comunidad.
- Utilizar herramientas que garanticen la continuidad del proyecto.
- Contar con una buena documentación.
- Garantizar que pueda ser extensible.

Para cumplir estos criterios y objetivos se ha elaborado un **Roadmap** basado en **semantic versioning**:

- 0.1.0: Implementación de la codificación de soluciones mediante individuos.
- 0.2.0: Implementación de los operadores de mutación.
- 0.3.0: Implementación de los operadores de cruce.
- 0.4.0: Implementación de los operadores de selección de padres.
- 0.5.0: Implementación de los operadores de selección de supervivientes.
- 0.6.0: Implementación de las clases gestoras de la población de individuos.

- 1 Introducción
- 2 Computación evolutiva
- 3 Objetivos
- 4 Desarrollo
- 5 Ejemplo de uso
- 6 Conclusions and future lines

Tecnologías utilizadas

Dado que la biblioteca debe ser completamente compatible con aplicaciones web, lo más lógico es desarrollarla con el *stack* del lenguaje **JavaScript**.



Figura: Logos de JavaScript, Node y NPM

Distinguiremos entre tecnologías utilizadas **durante el desarrollo** y dependencias en **producción**.

Tecnologías utilizadas en el desarrollo

El lenguaje de programación utilizado para el desarrollo ha sido **TypeScript**:



Figura: Logo de TypeScript

La utilización de este lenguaje aporta ciertas ventajas frente a **JavaScript**, como seguridad de tipos, y escalabilidad.

Tecnologías utilizadas en el desarrollo

Para garantizar la continuidad, estabilidad y calidad del software se han utilizado las siguientes herramientas:



Figura: Logos de TypeDoc, Jest, CircleCI y Coveralls

- **Documentación:** Se ha usado **TypeDoc**.
- **Tests:** Se ha utilizado **Jest**.
- **Integración continua:** Se ha usado **CircleCI**.
- **Coverage:** Se ha elegido **Coveralls**.

Tecnologías utilizadas en el desarrollo

Para formatear el código y depurar los posibles errores he utilizado:



Figura: Logos de Prettier, ts-lint y WebStorm

- **Prettier**: herramienta utilizada para formatear el código de manera común.
- **ts-lint**: Utilizado para comprobar errores de formato y de código antes de realizar la compilación.
- **WebStorm**: IDE utilizado para el desarrollo.

La intención es construir un **framework minimalista**, de tal forma que se tengan las mínimas dependencias posibles en producción.

random.js

Esta biblioteca se ha utilizado para llevar a cabo la **generación de números aleatorios**, pues nos permite entre otras cosas establecer la semilla con la que se generarán estos valores.

El desarrollo de **genetics.js** se ha hecho como un **paquete npm**.



Figura: Logo de genetics.js

Los diferentes módulos que compondrán el software desarrollado se corresponden con los métodos más comunes para implementar las diferentes fases de los algoritmos evolutivos.

Individuos

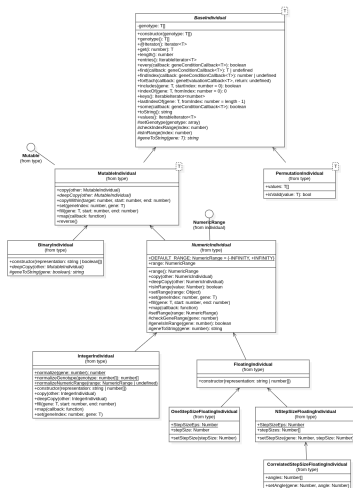
En los algoritmos evolutivos, los individuos de la población se corresponden con las **diferentes soluciones que puede tener el problema** que estemos resolviendo.



Figura: Ejemplo de conversión entre genotipo y fenotipo

- **genotipo**: codificación de la solución que posee el individuo.
- **fenotipo**: valor que se le asocia a dicha codificación.

Individuos



Los diferentes individuos implementados han sido:

- BaseIndividual
- MutableIndividual
- BinaryIndividual
- NumericIndividual
- IntegerIndividual
- FloatingIndividual

Figura: Diagrama UML de los individuos

```
import {
  BinaryIndividual,
  IntegerIndividual,
  FloatingIndividual } from "genetics-js"

// genotipo: [true, false, false, false, true]
const ind = new BinaryIndividual("10001");

// genotipo: [3, 4, -2, 1]
const ind1 = new IntegerIndividual("3 4 -2 1");

// genotipo: [0.03, -1.0, 0.25]
const ind2 = new FloatingIndividual("3E-2 -1 0.25");
```

Figura: Ejemplo de instanciación de diferentes individuos

Generador de individuos

El generador de individuos se utiliza para inicializar aleatoriamente la población.

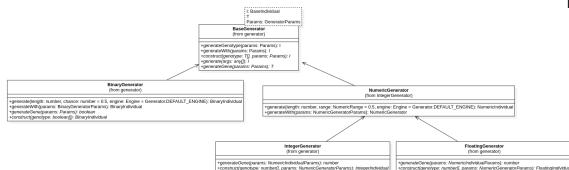


Figura: Diagrama UML del generador aleatorio

Los diferentes generadores implementados han sido:

- BaseGenerator
- BinaryGenerator
- NumericGenerator
- IntegerGenerator
- FloatingGenerator

Generador de individuos

```
import {  
  BinaryGenerator,  
  IntegerGenerator,  
  FloatingGenerator } from "genetics-js"  
  
const binaryGen = new BinaryGenerator();  
const binaryInd = binaryGen.generateWith({ length: 4, chance: 0.3 });  
  
const integerGen = new IntegerGenerator();  
const integerInd = integerGen.generateWith({ length: 4, range: [1, 5] });  
  
const floatingGen = new FloatingGenerator();  
const floatingInd = floatingGen.generateWith({ length: 4, range: [-0.5, 2] });
```

Figura: Ejemplo de uso de diferentes generadores de individuos

Gestión de la población

La clase `Population` se utiliza para gestionar la población de individuos que compone nuestro problema.

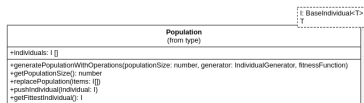


Figura: Diagrama UML de la clase `Population`

Los diferentes parámetros a los que se puede acceder son:

- `averageAge`
- `averageFitness`
- `fitnessSum`
- `fittestIndividualIndex`

Selección de padres

La fase de selección de padres consiste en seleccionar que individuos serán los que se **reproducirán para formar la descendencia**.

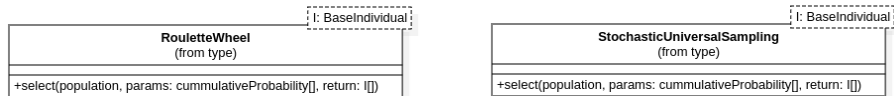
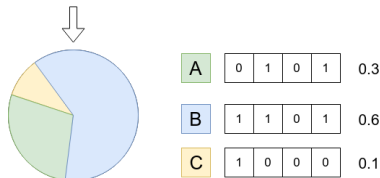


Figura: Diagramas de RouletteWheel y StochasticUniversalSampling

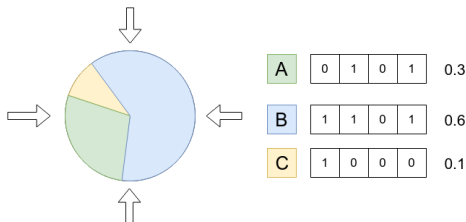
Los métodos de selección implementados han sido:

- RouletteWheel
- StochasticUniversalSampling

Selección de padres



(a) Esquema del procedimiento Roulette Wheel



(b) Esquema del procedimiento Stochastic Universal Sampling

Operaciones de cruce

Las operaciones de cruce consisten en generar una descendencia a partir de **dos individuos padres**.

Los métodos que se han implementado han sido:

- BaseCrossover e interfaz Crossover
- NPointsCrossover
- OnePointCrossover
- UniformCrossover
- SimpleArithmeticRecombination
- SingleArithmeticRecombination
- WholeArithmeticRecombination

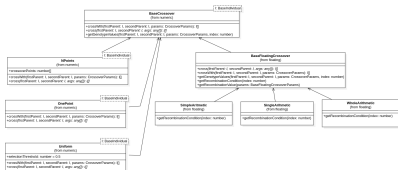


Figura: Diagrama UML de las operaciones de cruce

```
import { OnePointCrossover, BinaryIndividual } from "genetics-js";

const cross = new OnePointCrossover();
const ind1 = new BinaryIndividual("0110010");
const ind2 = new BinaryIndividual("0100100");

/**
 * Una posible descendencia que se generaría:
 * 011 | 0010 -> 0110100
 * 010 | 0100 -> 0100010
 */
cross.crossWith(ind1, ind2);
```

Figura: Ejemplo de una operación de cruce

Operaciones de mutación

Las operaciones de mutación tratan de **producir una cierta variación** en la descendencia generada mediante un operador de cruce.

Los métodos que se han implementado han sido:

- MutationBase e interfaz Mutation
- BitwiseMutation
- CreepMutation
- RandomResetting
- FloatingNonUniformMutation
- FloatingUniformMutation

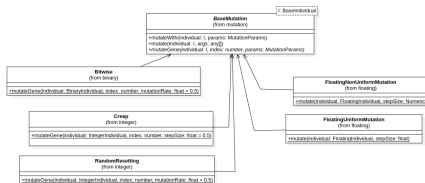


Figura: Diagrama UML de las operaciones de mutación

Operaciones de mutación

```
import { BitwiseMutation, BinaryIndividual } from "genetics-js";

const mutation = new BitwiseMutation();
const ind = new BinaryIndividual(" 0010010 ");

/**
 * Posible resultado tras elegir los bits 1 y 5 para ser mutados:
 * 0010010 -> 0110000
 */
mutation.mutateWith(ind, { mutationRate: 0.1 });
```

Figura: Ejemplo de una operación de mutación

Reemplazo

La fase de reemplazo consiste en determinar que individuos compondrán la siguiente generación teniendo a la población inicial y a la descendencia generada.

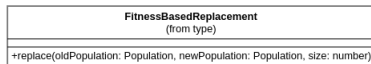
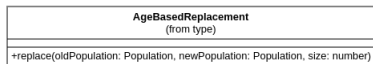


Figura: Diagramas de AgeBasedReplacement y FitnessBasedReplacement

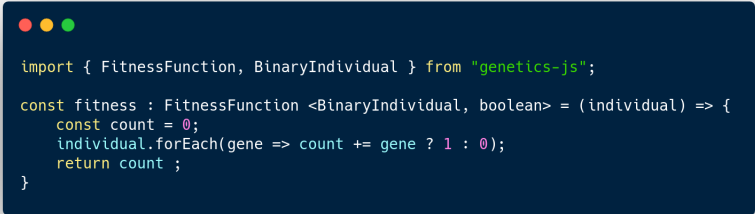
Los métodos de selección implementados han sido:

- AgeBasedReplacement
- FitnessBasedReplacement

Puntuación (*fitness*)

La función de puntuación (*fitness*) se utiliza para determinar **cuanto de adaptado al medio** está el individuo que se pretende evaluar. En el caso de un problema de optimización, esta función se corresponde con la que se pretende optimizar.

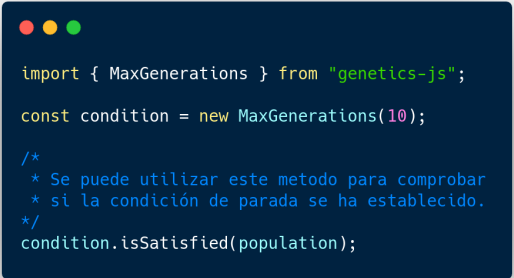
```
type FitnessFunction<I extends BaseIndividual<T>, T> =  
    (individual: I) => number
```



```
import { FitnessFunction, BinaryIndividual } from "genetics-js";  
  
const fitness : FitnessFunction <BinaryIndividual, boolean> = (individual) => {  
    const count = 0;  
    individual.forEach(gene => count += gene ? 1 : 0);  
    return count ;  
}
```


Condición de finalización

La condición de finalización es el criterio que determinará cuando se detendrá el algoritmo evolutivo. En este caso, **un número máximo de iteraciones sin mejora**.

A code editor window with a dark blue background and three colored window control buttons (red, yellow, green) in the top left corner. The code is written in a light-colored font and includes a JSDoc comment in blue.

```
import { MaxGenerations } from "genetics-js";  
  
const condition = new MaxGenerations(10);  
  
/*  
 * Se puede utilizar este metodo para comprobar  
 * si la condición de parada se ha establecido.  
 */  
condition.isSatisfied(population);
```

Figura: Utilización de la condición de parada

- 1 Introducción
- 2 Computación evolutiva
- 3 Objetivos
- 4 Desarrollo
- 5 Ejemplo de uso
- 6 Conclusions and future lines

Ejemplo de uso

Para el ejemplo de uso implementaremos una aplicación web con **React**, que permita resolver el problema de la mochila (**Knapsack Problem**).

Formulación del problema:

$$\begin{aligned} \max_x \quad & \sum_{i=1}^n v_i x_i \\ \text{sujeto a} \quad & \sum_{i=1}^n w_i x_i \leq W \\ & x_i \in \{0, 1\} \end{aligned}$$

Despliegue de la aplicación

<https://cristianabrante.github.io/GeneticsJsKnapsack/>

- 1 Introducción
- 2 Computación evolutiva
- 3 Objetivos
- 4 Desarrollo
- 5 Ejemplo de uso
- 6 Conclusions and future lines

Conclusions

The repercussion of **genetics.js** in the open source community has been significant:



50 downloads / weeks



18 stars
200 commits



81 followers

Conclusions



Concurso Universitario
De Software Libre

Special award to best project



Used for a scientific dissemination
campus for high school students

Some of the future lines that the project is going to follow:

- Changing the documentation technology to **docusaurus**, and manage the repository with **Lerna** and **Yarn**.
- Restructuring the most recently developed classes and some file organization.
- Improving the tests, mainly by creating **mockup data** with **jest**.
- Implementing asynchrony support.
- Allowing the execution of statistical tests.
- Adding new algorithms, individuals and methods for each phase.



A.E. Eiben and J.E. Smith

Introduction to evolutionary computing.

Springer, 2003.



John H Holland

Genetic algorithms and the optimal allocation of trials.

SIAM Journal on Computing, 1973.



Coromoto León, Gara Miranda, and Carlos Segura

Metco: a parallel plugin-based framework for multi-objective optimization.

International Journal on Artificial Intelligence Tools, 2009.



Sebastien Cahon, Nordine Melab, and E-G Talbi

Paradiseo: A framework for the reusable design of parallel and distributed metaheuristics.

Journal of heuristics, 2004.



Repositorio GitHub

GeneticsJS

<https://github.com/CristianAbrante/GeneticsJS>



Paquete en npm

genetics-js

<https://www.npmjs.com/package/genetics-js>



Cuenta de Twitter

@GeneticsJS

<https://twitter.com/GeneticsJs>



Blog del proyecto

genetics.js

<https://geneticsjs.wordpress.com/>

¿Preguntas?

Autor: Cristian Manuel Abrante Dorta (cristian@abrante.me)

Tutor: Eduardo Manuel Segredo González

Cotutora: Coromoto Antonia León Hernández