# Tarea 3 - Programación genética

Profesor: Alexandre Bergel
Auxiliar: Juan-Pablo Silva
Ayudantes: Marco Caballero
María José Berger

Para esta tarea deberán implementar lo necesario para aplicar técnicas de programación genética para encontrar ecuaciones que se acerquen a un conjunto de puntos, o a una expresión particular.

## 1 Contenidos

En esta sección explicaremos lo que deberán hacer en la tarea y lo que se espera que tengan implementado.

### 1.1 Código del algoritmo

Un algoritmo de programación genética se compone de 3 cosas.

- **Librería de arboles**: se debe tener una librería que represente arboles que tenga las siguientes características:
  - **Evaluar**: un método que permita evaluar el árbol.
  - **Copiar**: un método que permita copiar el árbol y generar uno nuevo sin referencias al anterior. Esto es importante, ya que al momento de hacer *crossover* tendremos una referencia al árbol original, que al cambiarlo modificara todos los que lo referencian.
  - **Serializar**: no serializar en el sentido de binario en disco, sino más bien crear una lista con todos los nodos del árbol. Esto es para poder seleccionar un nodo al azar del árbol.

  Son libres de usar, o basarse en la librería de arboles que subimos a U-Cursos. El archivo `arboles.py` en material docente.
- **Generador aleatorio de arboles**: el generador de arboles es necesario ya que los genes en la programación genética son efectivamente arboles. El generador debe tener las siguientes entradas como mínimo:
  - **Conjunto de nodos internos**: estos representan a las funciones u operaciones con las que trabajaremos. Por ejemplo podría representar *suma*, *resta*, *maximo*, etc.
  - **Conjunto de nodos terminales**: estos serian los valores. Por ejemplo podrían ser números (1, 2, 3, etc) o ser variables (`x`, `y`).

  - **Profundidad**: indica la profundidad del árbol. Es decir, cuando el árbol debe dejar de expandirse. Pueden pensarlo como cuando deben seleccionar un nodo terminal en vez de uno interno.

  Adicionalmente también puede agregar una probabilidad de que el árbol elija no seguir expandiéndose y seleccione un terminal en vez de un interno, para no tener arboles balanceados.

  Al igual que la librería de arboles, en material docente pueden encontrar el archivo `ast.py` que genera arboles aleatorios, para un conjunto de entrada de nodos internos y terminales.

- **Algoritmo genético adaptado**: el algoritmo genético es lo mismo que antes, en la clase de *algoritmos genéticos*, ya que en un principio es el mismo proceso. La diferencia entra al momento del *crossover* y la mutación, ya que ahora como tratamos con arboles hay que tener un poco más de cuidado. Considere los siguientes pasos para esta parte:

  1. Hacer una copia del primer padre, llamemosle `new_element`.
  2. Elegir un punto al azar de `new_element`, es decir, elegir un nodo al azar (no hay distinción entre internos o terminales). Llamemos a este punto `p1`.
  3. Elegimos un punto al azar del segundo padre, luego debemos hacer una copia de ese nodo elegido. A la copia le llamaremos `p2`.
  4. Reemplazamos `p1` con `p2`.

  Básicamente usamos al padre 1 como base para el nuevo individuo. Lo copiamos para no tener referencias colgantes. Elegimos un punto a reemplazar. Luego del padre 2 extraemos un nodo al azar y lo copiamos (no necesitamos copiar todo el padre 2, solo el sub-árbol que usaremos). Finalmente reemplazamos el nodo elegido del padre 1 por el del padre 2. Para la mutación es seleccionar un nodo al azar y reemplazarlo con un árbol generado al azar (con el generador) con alguna profundidad que ustedes pueden decidir, puede ser de una distribuci+on uniforme entre 0 y un número máximo que ustedes decidan.

Nuevamente, la programación genética hace uso del los algoritmos genéticos para evolucionar su población, por lo que puede usar el código de su tarea 2 y adaptar el *crossover* y mutación solamente. El fitness y generador de genes se pasan como argumentos al algoritmo, por lo que no necesita cambiar eso.

## 2 Ejercicios

Para que su tarea sea válida, debe incluir los siguientes ejercicios, reportando sus resultados en el `readme.md` de su Github

## 2.1 Encontrar Numero

La idea de este ejercicio es que el algoritmo intente encontrar una expresión que se acerque al número en cuestión, usando solo un conjunto de valores y funciones reducido, indicado por nosotros. Por ejemplo: si el número a encontrar es 10, dado el conjunto de funciones $\{+, *, -\}$, y los valores $\{2, 3, 5\}$, esperaríamos que el algoritmo encontrara algo como $2 * 5$, o quizás $5 + 5$.

### 2.1.1 Sin límite de repeticiones (0.6)

Con la idea anterior, el problema es: usando los conjuntos $\{+, -, *, max(\cdot)\}$ para las funciones y $\{25, 7, 8, 100, 4, 2\}$ para los terminales, encuentre el número 65346. Puede repetir los terminales cuantas veces necesite.

**Reporte sus resultados en una curva de evolución de fitness. Si su programa se cae, no genere el gráfico e indíquelo en el `readme`.**

### 2.1.2 Fitness (0.6)

Para el problema anterior de encontrar el número 65346, probablemente una vez avanzada la búsqueda el programa se volvía considerablemente más lento. Esto ocurre porque el tamaño del árbol se vuelve cada vez más grande, llegando a veces a superar el límite de recursión de Python para copiar referencias.

Entonces lo que debe hacer ahora es pensar en una función de fitness que castigue cuando los arboles crecen demasiado. Considere ahora que es un problema de optimización múltiple, donde quiere minimizar el error del árbol con el número a encontrar, y quiere también minimizar el tamaño del árbol. Puede considerar que ambas cosas son "igual de importantes".

**Haga un gráfico de evolución de fitness y agréguelo al `readme`.**

### 2.1.3 Sin repetición (0.6)

Hasta ahora hemos permitido la repetición infinita de los terminales, pero a veces podemos tener una restricción más fuerte sobre ellos. Ahora solo permitiremos que exista a lo más 1 de cada terminal en cada árbol. Necesitará ajustar su función de fitness para castigar a aquellos arboles que repiten los terminales. Por ejemplo, antes cuando queríamos encontrar 10, $2 * 5$ sería una solución válida, pero $5 + 5$ no porque repite el 5.

Entonces el problema es: usando los conjuntos $\{+, -, *\}$ para las funciones y $\{25, 7, 8, 100, 4, 2\}$ para los terminales, encuentre el número 65346. Solo puede usar los terminales 1 vez. Puede que no exista la combinación exacta que dé como resultado el número deseado, pero debemos encontrar el que más se acerque.

**Haga un gráfico de evolución de fitness y agréguelo al `readme`.**

## 2.2 Implementar variables (0.8)

Hasta ahora hemos solo trabajado con terminales absolutos, donde sabemos su valor en todo momento. Ahora implementaremos variables, como `x` e `y`, para las cuales no tenemos información sobre su valor antes de ejecutar el código. Esto nos permitirá encontrar ecuaciones y expresiones más complejas.

- Para implementar las variables sugerimos modificar el método de evaluación para que ahora reciba un diccionario de valores. Para esta tarea puede asumir que el diccionario no se modifica durante la ejecución, ya que no tenemos funciones de asignación de variables ni scope variable.
- Recuerde que todos los terminales tienen la misma probabilidad de parecer, por lo que si quiere que la `x` aparezca con la misma probabilidad de que aparezca un valor numérico, debe repetirla varias veces en el conjunto.

**En el `readme` indique las modificaciones que hizo para soportar las variables.**

## 2.3 Symbolic Regression (0.8)

Ahora que tenemos variables, podemos hacer búsqueda de ecuaciones que se acerquen a un conjunto de puntos, o encontrar ecuaciones especificas. Por simplicidad para esta tarea, intentaremos que el algoritmo encuentre la ecuación $x^2 + x - 6$. Por su puesto, no estamos buscando textualmente esa ecuación, nos interesa una ecuación que para un `x` dado, tenga el mismo resultado. Por lo mismo, $x * x - 6 + x$, o por ejemplo $x * x - 6 + 4 * x - 3 * x + 0$, también son soluciones validas. Como funciones utilice $\{+, -, *\}$, y como terminales use $\{-10, \ldots, 10, x\}$. Puede repetir los terminales cuantas veces necesite.

Para lograr esto, debe tener una implementación de variables y adaptar la función de fitness. La función de fitness ya no estará buscando un valor especifico, sino que una función, potencialmente definida para todos los reales. Como no podemos probarlos todos, basta con que las 2 funciones sean los suficientemente cercanas para un rango de puntos, como podría ser $[-100, 100]$, yendo de 1 en 1.

**Haga un gráfico de evolución de fitness y agréguelo al `readme`.**

## 2.4 Implementar el nodo `Division` (0.6)

En esta sección dejaremos de evitar la división y veremos cómo solucionar sus problemas.

### 2.4.1 Nuevo Nodo

Para implementar la división, cree otro nodo sobre la librería y agréguelo al generador de arboles aleatorios.

### 2.4.2 Division por 0

El problema con la división es que cuando al denominador es 0, esta está indefinida. Para arreglar este problema usted deberá implementar una forma de castigar a los arboles que estén dividiendo por 0. Una sugerencia es atrapar la excepción que emitirá el programa, luego a los arboles que emitan esa excepción, castigarlos como arboles no validos.

### 2.4.3 Evaluacion

En el `readme`, indique que hizo para arreglar la división por 0. Y agregue un gráfico de evolución de fitness para *Symbolic Regression* usando la división. No se asuste si ahora el algoritmo tiene más problemas para encontrar la ecuación.

## 3 Análisis

Para uno de los ejercicios, al igual que la tarea 2, debe hacer un *Heatmap* para 2 hyperparámetros.

- **Heatmap de configuraciones:** la idea del *Heatmap* es explorar exhaustivamente muchas combinaciones de **población** y de **tasa de mutación**, para encontrar la que mejor se adecua al problema. Esto se refiere a graficar, por ejemplo, para todas las combinaciones de la **población** en el rango $[50, \ldots, 1000]$ yendo de 50 en 50, y la **tasa de mutación** entre $[0.0, \ldots, 1.0]$ de 0.1 en 0.1. El cómo saber cuándo una combinación es "mejor" que otra se deja a su elección, pero puede considerar que si el algoritmo **demora mucho** en encontrar algo entonces es peor que otro que **demora menos**.
  Basta con que haga este *Heatmap* para 1 de los ejercicios, **NO** necesita hacerlo para todos.

## 4 Evaluación

Para evaluar sus tareas se revisará su código, por lo que se pedirá que las partes relevantes estén comentadas para facilitar la corrección a los ayudantes. Además, para la parte de análisis, debe hacer un `readme` con esta información. A continuación se hace el desglose de evaluación:

- **Código del algoritmo** (1.0 punto): que este implementada la librería de arboles, generadora de arboles y la adaptación al algoritmo genético. Puede usar el código dado para los arboles y generadora.
- **Análisis** (1.0 punto): debe hacer el *heatmap* indicado en la sección 3.
- **Ejercicios** (4.0 puntos): corresponde a la implementación de los ejercicios mostrados en la sección 2. Cada ejercicio se evaluará mirando el `readme`, por lo que siga las instrucciones de cada ejercicio, agregando un gráfico donde se pide, y haciendo un comentario o acotación cuando se necesite.

Son 6 ejercicios, sus ponderaciones están en el titulo de la sección correspondiente. Todos valen 0.6, menos los de variables (2 ejercicios) que valen 0.8.

- **Nota**: Todo esto debe ser presentado en el `readme` de su repositorio.
  **NO** se extienda demasiado, es una descripción de lo que hizo más los resultados que obtuvo. Aquí puede encontrar una guía de como usar *markdown*[1].

## 5 Entrega

La fecha de entrega de la tarea es el **lunes 28 de octubre**. Como usaremos Github para bajar sus tareas, basta con que en U-cursos suban **cualquier** archivo, y en los comentarios de entrega agreguen el enlace a su repositorio en Github.

Los análisis e imágenes de sus gráficos deben estar en el `readme` de su repositorio, no es necesario hacer un informe ni un documento en *pdf* explicando las cosas. La extensión en corta, solo debe hacer un pequeño reporte en el `readme` con los puntos señalados anteriormente.

[1] `https://github.com/adam-p/markdown-here/wiki/Markdown-Cheatsheet`