Ingeniería del Software II

Taller #4 – Algoritmos Genéticos

LEER EL ENUNCIADO COMPLETO ANTES DE ARRANCAR.

Fecha de entrega: 14 de Mayo de 2025 **(16:59hs)**
Fecha de re-entrega: 28 de Mayo de 2025 **(16:59hs) (no hay extensiones)**

Contenido del taller

El objetivo de este taller es desarrollar en `Python` un algoritmo genético para generar casos de test que maximicen la cobertura de branches de un programa. En particular trabajaremos sobre la función `cgi_decode`, que decodifica un string codificado con el protocolo CGI.

Nuestros individuos serán test suites. Representaremos cada caso de test como un string, luego un test suite será representado con una lista de casos de test.

El proyecto para este taller contiene los siguientes archivos:

- `src/cgi_decode.py`: implementación completa de la función `cgi_decode`.
- `src/individual.py`: clase que representa a un individuo del algoritmo genético.

Archivos con funciones a completar:

- `src/evaluate_condition.py`
- `src/cgi_decode_instrumented.py`
- `src/get_fitness_cgi_decode.py`
- `src/create_population.py`
- `src/evaluate_population.py`
- `src/selection.py`
- `src/crossover.py`
- `src/mutate.py`
- `src/genetic_algorithm.py`

Algunos de los archivos mencionados tienen otro correspondiente en la carpeta `test` (e.g., `test/cgi_decode.py`). El taller provee un archivo `requirements.txt` para instalar todas las dependencias necesarias en un ambiente virtual de Python. Para instalar dichas dependencias, puede ejecutar los siguientes comandos en la carpeta del taller:

```
python3 -m venv venv
source venv/bin/activate
pip install -r requirements.txt
```

Si el comando `python3 -m venv venv` no funciona, puede utilizar el comando `virtualenv venv`. Alternativamente, se puede usar una IDE como *PyCharm* y configurar el entorno virtual desde la misma.

Puede ejecutar el test suite del proyecto y obtener su cobertura ejecutando el comando `./run_tests.sh`. Se recomienda la utilización de dicho comando para correr los tests ya que configura el entorno de `Python` para que los tests sean menos aleatorios. Para correr los casos de test en un archivo en particular, se puede pasar como primer argumento el módulo correspondiente. Por ejemplo, para ejecutar los tests en el archivo `test_cgi_decode.py`, se puede ejecutar el comando `./run_tests.sh test.test_cgi_decode`. Igualmente, también se pueden correr los tests en una IDE, como *PyCharm*, configurando la variable de entorno `PYTHONHASHSEED=0`.

Ejercicio 1

Completar el archivo `test/test_cgi_decode.py` con un test suite para el programa `cgi_decode` que tenga 100 % de cubrimiento de líneas y de branches. Para escribir los tests utilice las *aserciones* provistas por la librería de Python `unittest`. Puede ver los distintos métodos disponibles para escribir aserciones en `https://docs.python.org/3/library/unittest.html#unittest.TestCase.assertEqual`.

Ejercicio 2

a. Completar la implementación de la función `evaluate_condition`, que se encarga de evaluar una condición de un *branch* y actualizar los diccionarios de distancias a *branches*. Recibe los siguientes argumentos:

- condition_num: un entero que representa el identificador de la condición.
- op: La operación de comparación. Las comparaciones puede ser "Eq" (==), "Ne" (!=), "Lt" (<), "Gt" (>), "Le" (<=), "Ge" (>=), "In" (pertenencia a una colección, e.g., $x \in C$).
- lhs: el valor de la expresión izquierda de la comparación.
- rhs: el valor de la expresión derecha de la comparación

La función `evaluate_condition` debe comparar los valores `lhs` y `rhs` usando la operación declarada en el argumento `op` y retorna `True` o `False` de acuerdo al resultado de la comparación. Además, debe actualizar los diccionarios de *fitness* invocando a `update_maps`. Por ejemplo,

- `evaluate_condition(1, "Eq", 10, 20)` retorna False y actualiza el distances_true (valor 10), distances_false (valor 0) para la condición 1
- `evaluate_condition(2, "Eq", 20, 20)` retorna True y actualiza el distances_true (valor 0), distances_false (valor 1) para la condición 2
- `evaluate_condition('4', "In", 'Z', {'1': 1, 'F': 15})` retorna False y actualiza el distances_true (valor 3), distances_false (valor 0) para la condición 4.

Considere los siguientes casos para los tipos que pueden tomar los argumentos `lhs` y `rhs`:

- Ambos son enteros.
- Ambos son caracteres (i.e., strings de largo 1). En este caso se debe usar la función *ord*() para comparar dichos valores numéricamente.
- `lhs` es un caracter y `rhs` es un diccionario. En este caso se debe comparar numéricamente `lhs` contra la colección de claves del diccionario, también usando la función *ord*().

A continuación se muestran ejemplos de actualización para distances_true, distances_false, utilizando $K = 1$ para calcular la distancia de branch:

Operación	distance_true	distance_false
20 == 10	10	0
20 == 20	0	1
20 != 10	0	10
20 != 20	1	0
$10 \leq 20$	0	11
$20 \leq 10$	10	0
$20 \leq 20$	0	1
$10 < 20$	0	10
$20 < 10$	11	0
$20 < 20$	1	0
10 In []	sys.maxsize	0
10 In [1,2,3]	7	0
10 In [10]	0	1
10 In [10,10]	0	1
13 in [11,12,18]	1	0

b. Completar el test suite en el archivo `test/test_evaluate_condition.py` para que tenga 100 % de cubrimiento de líneas y de branches para la función `evaluate_condition`. No hace falta testear los valores de los diccionarios de `distances_true` y `distances_false` en este punto.

Ejercicio 3

a. Completar la implementación de la función `cgi_decode_instrumented(test_case)`. Para esto, deberán copiar la implementación original del programa `cgi_decode` pero reemplazando todas las condiciones por llamadas a la función `evaluate_condition`, indicando el identificador de la condición.

Por ejemplo, si la primer condición que aparece es `i <10`, debe ser reemplazada por `evaluate_condition(1, ‘‘Lt’’, i, 10)`.

b. Usando los casos de test del ejercicio anterior como inspiración, escribir casos de test nuevos en el archivo `test_evaluate_condition_for_cgi_decode_instrumented.py` para comprobar que `distances_true` y `distances_false` son actualizados correctamente al ejecutar nuestro programa instrumentado. Además, verificar que el programa instrumentado retorna el mismo resultado que el programa original. Tenga en cuenta que debe llamar a la función `clear_maps()` antes de ejecutar cada caso de test para que los diccionarios *globales* de distancias a *branches* estén vacíos.

Por ejemplo:

- Ejecutando `cgi_decode_instrumented(‘‘Hello+World’’)` retorna “Hello World”
- El diccionario `distances_true` queda `{1: 0, 2: 0, 3: 35}`
- El diccionario `distances_false` queda `{1: 0, 2: 0, 3: 0}`

Ejercicio 4

Se desea crear una función de fitness para guiar inputs que ejerciten todo el código del programa `cgi_decode`.

a. Completar la implementación de la función `get_fitness_cgi_decode(test_suite)` que computa el valor de fitness para un individuo, corriendo su test-suite usando la función `cgi_decode_instrumented(test_case)`.

Dado que estamos usando *branch coverage*, el fitness va a estar dado por la suma de un valor determinado para cada objetivo (una rama verdadera o falsa en un *branch*) en el programa que

estamos testeando. Para un objetivo en particular, si el test suite logra ejecutar el *branch*, entonces usamos como valor la distancia normalizada[1]. Sino, el valor que usamos es 1.

Recuerde utilizar la función `clear_maps` para limpiar los diccionarios de distancias a *branches* antes de correr el `test_suite`.

Tener en cuenta que la función `cgi_decode_instrumented` puede lanzar una excepción si el input no es válido. **Esa excepción no debe cortar la ejecución del test-suite.**

b. Tome como guía los siguientes ejemplos y escriba casos de tests para la nueva función implementada en el archivo `test_get_fitness_cgi_decode.py`:

```
get_fitness_cgi_decode(["%AA"]) = 2.357142857142857
get_fitness_cgi_decode(["\%AU"]) = 3.03021978021978
get_fitness_cgi_decode(["\%UU"]) = 4.53021978021978
get_fitness_cgi_decode(["+"]) = 6.5
get_fitness_cgi_decode(["Hello+Reader"]) = 4.972222222222222
get_fitness_cgi_decode([""]) = 8.5
get_fitness_cgi_decode(["%A"]) = 6.023809523809524
get_fitness_cgi_decode(["%A", "%", "\%1+", "%+1", "a+%AA"]) = 0
get_fitness_cgi_decode(["%"]) = 5.857142857142858
```

Ejercicio 5

Completar la implementación de la función `create_population(population_size)` que crea una lista de #`population_size` individuos. Donde cada individuo contiene un test-suite de entre 1 y 15 casos de tests, y cada caso de test es un string de entre 0 y 10 caracteres.
Se recomienda completar y utilizar las funciones auxiliares definidas. Para la creación de los strings, solo está permitido utilizar los caracteres disponibles en `string.printable` de Python.

Ejercicio 6

Completar la implementación de la función `evaluate_population(population)` que, dado una lista de individuos, obtiene para cada uno su valor de fitness usando la función `get_fitness_cgi_decode` y lo guarda usando la función `Individual::set_fitness`.

Ejercicio 7

Completar las implementaciones de las funciones `tournament_selection` y `selection`. Donde `tournament_selection` realiza una selección por torneo de tamaño tournament_size en toda la población y `selection` selecciona dos individuos de toda la población mediante el método `selection_function` pasado como primer parámetro.

Ejercicio 8

Completar las implementaciones de las funciones `single_point_crossover` y `crossover`. La última aplica la función `crossover_function` pasada por parámetro a los individuos, dada una probabilidad. Mientras que la primera realiza un cruce single point entre dos individuos como indica la figura 1.

[1] usando la función de normalización $x/x+1$

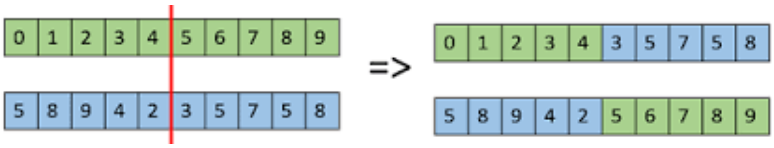

Figura 1: Visualización de single-point cross-over, recuerden que en nuestro caso un individuo es una lista de strings.

Ejercicio 9

Completar las implementaciones de las funciones `mutate` y `mutation`. La última en un estilo a los puntos anteriores es quien decide si la función `mutation_function` pasada por parámetro debe aplicarse o no al individuo, de acuerdo a una probabilidad.

Mientras tanto `mutate` debería aplicar alguna de las siguientes mutaciones, con igual probabilidad:

- agregar un nuevo caso de test aleatorio de hasta 10 caracteres
- eliminar un caso de test
- modificar un caso de test existente

En caso de modificar un caso de test existente, puede (con igual probabilidad) quitar, agregar o modificar un caracter del test.

La eliminicación de casos de tests o de caracteres solo debe considerarse si hay más de un caso de test o más de un caracter, respectivamente. La adición de casos de tests o caracteres solo debe considerarse si hay menos de 15 casos de tests o menos de 10 caracteres, respectivamente. La modificación de un caso de test o string solo debe considerarse si hay al menos un caso de test o al menos un caracter, respectivamente.

Ejercicio 10

Sea el siguiente algoritmo genético que llamaremos *algoritmo genético standard*:

Input: Stopping condition C, Fitness function δ, Population size p_s, Selection function s_f, Crossover function c_f, Crossover probability c_p, Mutation function m_f, Mutation probability m_p
Output: Population of optimised individuals P

$P \leftarrow$ GenerateRandomPopulation(p_s)
PerformFitnessEvaluation(δ, P)
while $\neg C$ **do**
  $N_P \leftarrow \{\ \} \cup$ Elitism(P)
  while $|N_P| < p_s$ **do**
    $p_1, p_2 \leftarrow$ Selection(s_f, P)
    $o_1, o_2 \leftarrow$ Crossover(c_f, c_p, p_1, p_2)
    Mutation(m_f, m_p, o_1)
    Mutation(m_f, m_p, o_2)
    $N_P \leftarrow N_P \cup \{o_1, o_2\}$
  end while
  $P \leftarrow N_P$
  PerformFitnessEvaluation(δ, P)
end while
return P

a. Usando todas las funciones definidas anteriormente, completar el archivo `genetic_algorithm.py` con la implementación de un algoritmo genético standard **sin elitismo**, donde los individuos contienen test_suites (listas de casos de tests), y cada caso de test es un string.

b. Una vez finalizado. Elija 3 semilllas distintas y configurelas utilizando el método de Python `random.seed(semilla)` para que las corridas sean determinísticas.

Para cada semilla debe agregar una serie de tests que informen (utilizando *asserts*):

- La cantidad de generaciones que realiza el algoritmo.
- El *fitness* y *branch coverage* logrado al final del algoritmo por el mejor individuo.
  Pista: Para obtener el branch coverage puede implementar una función parecida a *get_fitness_cgi_decode* de ser necesario.

Formato de Entrega

El taller debe ser entregado en el campus de la materia. La entrega debe incluir un archivo `entrega.zip` con el código implementado. Este debe estar detalladamente documentado. Además, debe incluir en la documentación una descripción de la resolución de cada ejercicio, incluyendo una breve discusión de las decisiones de diseño más importantes tomadas para resolver el taller.

El archivo `entrega.zip` debe contener también el reporte de coverage generado por `coverage.py` para todos los tests sobre la implementación final.