Ingeniería del Software II

Taller #4 – Algoritmos Genéticos

LEER EL ENUNCIADO COMPLETO ANTES DE ARRANCAR.

Fecha de entrega: 25 de Abril de 2024
Fecha de re-entrega: 16 de Mayo de 2024 (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 un programa. Trabajaremos sobre la función `cgi_decode`, que decodifica un string codificado con el protocolo CGI. El proyecto para este taller contiene los siguientes archivos:

- `src/cgi_decode.py`: implementación de la función `cgi_decode`.
- `src/evaluate_condition.py`: implementación (a completar) de una función para evaluar una condición de un *branch* y actualizar los mappings de distancias a *branches*.
- `src/cgi_decode_instrumented.py`: implementación (a completar) de una función que reemplaza cada condición del programa original por la llamada correspondiente a `evaluate_condition`.
- `src/get_fitness_cgi_decode.py`: implementación (a completar) de una función que computa el valor de fitness para una lista de casos de tests.
- `src/create_population.py`: implementación (a completar) de una función que crea una lista de `size` individuos.
- `src/evaluate_population.py`: implementación (a completar) de una función que dado una lista de individuos, retorna un mapping de cada individuo a su valor de fitness.
- `src/selection.py`: implementación (a completar) de una función que dado una mapping de individuos a su valor de fitness devuelve el mejor individuo.
- `src/crossover.py`: implementación (a completar) de una función que dado dos padres retorna el single-point cross-over.
- `src/mutate.py`: implementación (a completar) de una función que dado un individuo aplica una mutación de acuerdo a una probabilidad.
- `src/genetic_algorithm.py`: implementación (a completar) de un algoritmo genético.

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, ejecute los siguientes comandos en la carpeta del taller:

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

Alternativamente, si el comando `python3 -m venv venv` no funciona, puede utilizar el comando `virtualenv venv`.

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 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 mappings de *fitness* invocando a `update_maps`. Por ejemplo,

- `evaluate_condition(1, "Eq", 10, 20)` retorna False y actualiza el distances_true, distances_false para la condición 1
- `evaluate_condition(2, "Eq", 20, 20)` retorna True y actualiza el distances_true, distances_false para la condición 2
- `evaluate_condition(4, "In", 'a', ['b','c','d'])` retorna False y actualiza el distances_true, distances_false 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 mappings de `distances_true` y `distances_false` en este punto.

Ejercicio 3

En este taller representaremos cada caso de test como un string, mientras que un test suite será representado con una lista de casos de test.

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.

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 mappings de distancias a *branches* estén vacíos.

   Por ejemplo:

   - Ejecutando `cgi_decode_instrumented(“Hello+World”)` retorna “Hello World”
   - El mapping `distances_true` queda `{1: 0, 2: 0, 3: 35}`
   - El mapping `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 una lista de casos de tests 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. Puede utilizar la función `clear_maps` para limpiar los diccionarios de distancias a *branches*.

Tener en cuenta que la función `cgi_decode_instrumented` puede lanzar una excepción si el input no es válido. En ese caso, alcanza con medir el valor de fitness hast el momento de la excepción.

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''])` debe retornar ...
- `get_fitness_cgi_decode([''%AU''])` debe retornar ...
- `get_fitness_cgi_decode([''%UU''])` debe retornar ...
- `get_fitness_cgi_decode([''Hello+Reader''])` debe retornar ...
- `get_fitness_cgi_decode([''''])` debe retornar ...
- `get_fitness_cgi_decode([''%''])` debe retornar ...
- `get_fitness_cgi_decode([''%1''])` debe retornar ...
- `get_fitness_cgi_decode([''+''])` debe retornar ...
- `get_fitness_cgi_decode([''+%1''])` debe retornar ...
- `get_fitness_cgi_decode([''%1+''])` debe retornar ...

Ejercicio 5

Completar la implementación de la función `create_population(population_size)` que crea una lista de `size` individuos, y cada individuo es una lista de entre 1 y 15 casos de tests, donde cada caso de test es un string de entre 0 y 10 caracteres. 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, retorna un mapping de cada individuo a su valor de fitness usando la función `get_fitness_cgi_decode`.

Ejercicio 7

Completar la implementación de la función `selection(fitness_by_individual, tournament_size)` que dado una mapping de individuos a su valor de fitness y un tamaño de torneo como un entero positivo, realiza *Tournament Selection* y retorna el ganador del torneo.

Ejercicio 8

Completar la implementación de la función `crossover(parent1, parent2)` que dado dos padres retorna el *single-point cross-over* (*offspring1* y *offspring2*).

Ejercicio 9

Completar la implementación de la función `mutate(individual)` que dado un individuo aplica una mutación de acuerdo a una probabilidad. La mutación puede (con igual probabilidad) agregar un nuevo caso de test aleatorio de hasta 10 caracteres, eliminar un caso de test, o 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.

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

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

a. Usando todas las funciones definidas anteriormente, completar el archivo `genetic_algorithm.py` con la implementación de un algoritmo genético donde los individuos son listas de casos de tests, y cada caso de test es un string. Puede implementar, por ejemplo, el algoritmo genético standard sin elitismo.

b. Crear 3 tests de regresión distintos en el archivo `test_genetic_algorithm.py` para el algoritmo implementado. Utilice el método de Python `random.seed(semilla)` para que los tests sean determinísticos. Estos tests deben informar (utilizando *asserts* en el test):

   - La cantidad de generaciones que realiza el algoritmo.
   - El *branch coverage* logrado al final del algoritmo por el mejor individuo.

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.