Técnicas de pruebas automáticas para algoritmos evolutivos

Daniel Molina Juan Manuel Dodero

Resumen— **El uso de pruebas automáticas es una herramienta muy útil para detectar errores en el código, pero el comportamiento no-determinista de ciertos operadores de los algoritmos evolutivos impide que los resultados sean repetibles, con lo que se suele considerar que no son aplicables. En este trabajo se proponen nuevas estrategias de pruebas específicas para estos casos. Las estrategias propuestas permiten comprobar las características de los operadores, y hacer pruebas de bajo nivel sobre los operadores a pesar del uso de números seudoaleatorios. Para demostar el uso estas estrategias, se aplican a un algoritmo genético estacionario, comprobando cómo se pueden realizar pruebas automáticas sobre los algoritmos evolutivos.**

Palabras clave— **algoritmos evolutivos, tests, pruebas automáticas, no determinismo, behavior driven development**

I. Introducción

Los algoritmos evolutivos [1], AEs, son algoritmos no deterministas que han demostrado obtener muy buenos resultados en todo tipo de problemas complejos [2].

Como todo programa, las implementaciones de los AEs son susceptibles de tener errores en su programación, errores que pueden reducirse por medio de técnicas como el uso de pruebas automáticas [3].

Especialmente interesante es el uso de *Test-Driven Development* que planea desarrollar el software partiendo de la especificación de los tests [4], [5]. Este enfoque parece permitir obtener una mejora visible en el ratio de errores [6].

Lamentablemente, el carácter no determinista (por el uso de operadores seudoaleatorios) de los AEs impiden que los resultados sean repetibles, dificultando las pruebas [7], [8], lo que conduce a una falta de pruebas en partes del algoritmo que pueden ser críticas.

Sin embargo, en los últimos años han surgido nuevos enfoques, técnicas y herramientas que abren nuevas posibilidades. Uno de estos nuevos enfoques es la metodología *Behavior Driven Development*, BDD, que predica especificar las pruebas desde el punto de vista del cliente, describiendo las funcionalidades requeridas en vez de detalles de implementación [9]. A la hora de probar una clase, hay que valorar las dependencias con otras, ya que la parte que se desea probar puede llamar a su vez a clases auxiliares, lo cual dificulta las pruebas. Para resolver este problema, ha surgido la técnica de definir objetos *mocks* que mantienen el mismo interfaz de las clases auxiliares, y que permiten probar la clase cuando aún no se han terminado de implementar las auxiliares, así como comprobar sus interacciones [10].

En este trabajo planteamos cómo dichas nuevas técnicas pueden ser utilizadas de forma conjunta para realizar un modelo de pruebas automáticas sobre un algoritmo evolutivo. En concreto, se aplican sobre un algoritmo genético con codificación real [11], aunque el enfoque planteado puede extenderse a otros AEs.

En nuestro trabajo abordamos la problemática del no-determinismo por medio de dos vías. Por un lado, aplicamos el enfoque BDD para poder definir comprobaciones sobre el comportamiento deseado en vez de sobre la implementación; y, por otro lado, definimos tests de caja blanca mediante el uso de objetos *mocks* para simular la generación de números seudoalealeatorios.

II. Entorno BDD

El enfoque BDD plantea la idea de realizar tests que no dependen de la estructura interna del código a probar, sino de la funcionalidad que debe ofrecer. De esta manera, se pueden plantear tests que pueden ser comprendidos por el cliente. En nuestro caso, pueden ser entendibles por otro investigador que conozca la descripción del algoritmo pero no la implementación.

Para hacer esa separación aún más clara las librerías más modernas que siguen el enfoque BDD separan los tests en dos partes bien diferenciadas. Por un lado, una descripción que permite indicar por medio de frases en lenguaje natural tanto las condiciones como los resultados esperados; y por otro, ficheros que asocian las expresiones anteriores con código ejecutable que realiza las tareas a probar y la comprobación de la condición.

Hay múltiples herramientas BDD, y para los distintos lenguajes: En Ruby, *Cucumber*[1] es la más popular, en Java *JBehave*[2], y en Python, hay otros como *Lettuce*[3]. En este trabajo hemos usado *Lettuce* bajo Python. En todas ellas se trabaja de igual forma, y admiten el mismo formato de definición de tests.

En el directorio *tests/features* se crean distintos ficheros tests con extensión *.feature* en el que cada uno contiene los tests asociados a una característica del algoritmo.

En cada uno de estos ficheros se define una serie de escenarios, con el formato indicado en la Figu-

Universidad de Cádiz. E-mail: daniel.molina@uca.es
Universidad de Cádiz. E-mail: juanma.dodero@uca.es

[1] http://cukes.info/
[2] http://jbehave.org/
[3] http://lettuce.it/

```
Scenario: <Descripción>
    Given <sentence_init>
    When <sentence_condition>
    Then <sentence_expected>
```

Fig. 1. Formato de un escenario

ra 1, en donde la sección *Given* describe el estado del sistema antes de un evento, *When* describe las condiciones del evento que se quiere probar, y *Then* indica el resultado esperado tras el evento. La Figura 2 muestra, como ejemplo, una definición de pruebas sobre el operador factorial.

```
Feature: Compute factorial
  In order to play with BDD
  As beginners
  We'll implement factorial

  Scenario: Factorial of 0
    Given I have the number 0
    When I compute its factorial
    Then I see the number 1

  Scenario: Factorial of 1
    Given I have the number 1
    When I compute its factorial
    Then I see the number 1

  Scenario: Factorial of 2
    Given I have the number 2
    When I compute its factorial
    Then I see the number 2

  Scenario: Factorial of 5
    Given I have the number 5
    When I compute its factorial
    Then I see the number 120
```

Fig. 2. Definición de los tests sobre el operador factorial

También se puede indicar de forma más concisa, tal y como se ve en la Figura 3.

```
Feature: Compute factorial
  In order to play with BDD
  As beginners
  We'll implement factorial

  Scenario: Factorial of several numbers
    Given I have the number <number>
    When I compute its factorial
    Then I see the number <expected_result>

   Examples:
    | number | expected_result |
    | 0      | 1               |
    | 1      | 1               |
    | 2      | 2               |
    | 5      | 120             |
```

Fig. 3. Definición concisa de tests sobre factorial

Una vez definidos los tests, es necesario un fichero que asocia cada expresión en lenguaje natural de las secciones *Given*, *When* y *Then* con el código asociado de test. Dicha asociación se realiza por medio del fichero *steps* situado en el mismo directorio.

Por ejemplo, la Figura 4 muestra el contenido del fichero *tests/features/steps.py*, que contiene el código de las pruebas asociado a los casos de tests de las Figuras 2 ó 3. Se puede observar que se definen tres funciones, y por medio del decorador *step* se asocian a la frase correspondiente. En dos funciones se utiliza en *step* una expresión regular, que permite al sistema llamar a la función utilizando el valor numérico indicado en la frase. Una función recupera el valor numérico y lo guarda en la variable *world.number* (world es un almacén de datos para compartir variables entre funciones); otra realiza la operación a probar; y la última comprueba, mediante una sentencia *assert*, el resultado obtenido con el esperado.

```
from lettuce import *

@step('I have the number (\d+)')
def have_the_number(step, number):
    world.number = int(number)

@step('I compute its factorial')
def compute_its_fatorial(step):
    world.number = factorial(world.number)

@step('I see the number (\d+)')
def check_number(step, expected):
    expected = int(expected)
    assert world.number == expected
```

Fig. 4. Fichero steps.py

Para hacer la prueba se usa el programa BDD asociado (*lettuce* en este caso). En la Figura 5 se observa la salida cuando no hay error, y en la Figura 6 la salida cuando el test detecta un error. Se puede observar que indica la comprobación concreta que no se cumple.

```
$ lettuce -v 2 .
Factorial of several numbers ...
1 feature (1 passed)
4 scenarios (4 passed)
12 steps (12 passed)
```

Fig. 5. Salida por pantalla si no hay error

```
$ lettuce -v 3 .
Feature: Compute factorial
# tests/features/factorial.feature:1
...

Traceback (most recent call last):
  File ".../lettuce/core.py", line 113, in \
                                        __call__
    ret = self.function(self.step, *args, **kw)
  File ".../tests/features/steps.py", line 15 \
                in check_number
    assert world.number == expected
AssertionError

1 feature (0 passed)
4 scenarios (3 passed)
12 steps (1 failed, 11 passed)
```

Fig. 6. Salida por pantalla en caso de error

III. Algoritmo de referencia

Vamos a describir el algoritmo sobre el que se van a definir las comprobaciones [4].

[4]Tanto el código del algoritmo como el de los tests está libremente disponible en la url

```
Feature: SSGA Init
    In order to test SSGA we test that the init
    population has the right popsize,
    dimension, and that initial fitness between
    the individuals are different.

    Scenario: Init Population
        Given I have an SSGA algorithm \
            for dimension <dimension>
        When I init the population with \
                <popsize> individuals
        Then the population size is <popsize>
        Then dimension for each individual \
                is <dimension>
        Then fitness is initialized
```

Fig. 7. Fichero *init.feature*

El algoritmo elegido es un algoritmo genético estacionario, SSGA, que hace uso del operador de cruce $BLX - \alpha$ [12]. Como mecanismo de selección de padres, se aplica el criterio *Negative Assortative Mating, NAM*. El primer padre es elegido de forma aleatoria de la población, y para calcular el segundo padre se seleccionan aleatoriamente N_{num} soluciones y se elige la solución más alejada del primer padre. Como mecanismo de reemplazo se elige el *Replacement Worst*, en el que el nuevo individuo reemplaza al peor individuo de la población, si lo mejora. Se ha demostrado que esta combinación de selección y reemplazo permite un adecuado equilibrio entre exploración y explotación [13].

Se puede observar que es un algoritmo que aunque simple posee características típicas de un algoritmo evolutivo típico: Es un algoritmo poblacional, elitista, y con distintas operaciones con componente aleatorio: cruce, y selección.

IV. Pruebas de comportamiento

El uso de pruebas BDD permite establecer pruebas de los distintos elementos del algoritmo como la inicialización, selección, y cruce, en un formato legibles por alguien experto en el algoritmo. En este apartado se mostrarán las definiciones de los tests, el fichero *steps.py* asociado se incluye en el apéndice, en las Figuras 13, 14, y 15.

A. Inicialización

Para comprobar la inicialización hemos definido unos tests que comprueben el estado de la población inicial. La Figura 7 muestra la especificación de los tests. Este test es relativamente sencillo, pero permite comprobar la coherencia de la población inicial.

B. Selección

Para probar la selección *NAM*, un primer intento sería crear un test a bajo nivel que probase que cada elección de padre siguiese el criterio indicado. Este enfoque tiene dos graves inconvenientes. Por un lado, el comportamiento aleatorio del operador impide que los resultados sean repetibles. Además, en este caso el código del test sería una repetición del original, por lo que si hubiese un error en su implementación, dicho error se trasladaría al test, no detectándose. Por tanto, no es un modelo recomendable.

El enfoque que proponemos es el siguiente: aprovechar que el resultado del operador define una serie de características que se deben cumplir si está implementado de forma correcta. Por tanto, nos centramos en comprobar el comportamiento esperado en vez del código. Evidentemente, no es una prueba completa, pero sí nos puede ser de utilidad para detectar errores.

```
Feature: SSGA NNAM
    In order to test the NAM selection we check
    that the two parents arre not the same one,
    and than criterion between the the first
    parent and the second parent is rgiht.

    Scenario: NAM selects different parents
        Given I have an SSGA algorithm for \
                dimension 10
        Given I init the population with \
                <popsize> individuals
        When I select parents with tournament \
                size <tournament_size>
        Then the parents are different \
                after <numruns> tests

    Examples:
| popsize | tournament_size | numruns |
|    50   |  3              | 300     |
...

    Scenario: NAM Select the far away parent
        Given I have an SSGA algorithm for \
                dimension 10
        Given I init the population with \
                50 individuals
        When I select parents with tournament \
                size 49
        Then the distance between parents is \
                the longest
```

Fig. 8. Fichero *selection.feature*

En este caso, nos podemos basar en el requisito que ambos padres sean siempre diferentes y; por otro lado, si el grupo de torneo incluyese al resto de la población, el segundo padre debe ser el individuo más distante al primero. La Figura 8 muestra la especificación de dichos tests.

C. Cruce

Para probar el algoritmo $BLX - \alpha$, aplicamos el mismo enfoque que en el anterior. En este caso, nos centramos en el comportamiento con $\alpha = 0$. Los comportamientos que queremos probar son que el cruce de un individuo consigo mismo devuelve ese mismo individuo, y que el cruce de dos padres genera un hijo que está entre dichos padres. La Figura 9 muestra la especificación.

D. Ejecutando el Algoritmo

Para que el algoritmo sea correcto, existen dos comportamientos que debe de cumplir: que el algoritmo sea elitista; y que el número de evaluaciones que realmente realice sea el que se le indica. La Fi-

https://github.com/dmolina/ssgatests

```
Feature: SSGA Crossover
    Check for BLX-0 that crossing a solution
    with itself gives the same solution,
    and that offspring are always between
    their parents

    Scenario: Crossover with itself
        Given I have an SSGA algorithm
        When I init the population with \
            50 individuals
        When I set the same parent as mother
        When I cross with alpha 0
        Then the children is the same

    Scenario: Offspring is between their parents
        Given I have an SSGA algorithm
        When I init the population with \
            50 individuals
        When I set randomly two parents
        When I cross with alpha 0
        Then the children is between them
```

Fig. 9. Fichero *crossover.feature*

```
Feature: SSGA run
    Check two features of the algorithm. First,
    the real evaluations number is right, and
    the elitism of the algorithm.

    Scenario: SSGA actually use the \
              maximum evaluation number
        Given I have a SSGA algorithm
        When I init the population \
            with <population_size>
        When I run the algorithm during \
            <num_itera> iterations
        Then they were evaluated \
            <num_eval> solutions

    Examples:
| population_size | num_itera | num_eval |
|    50           |   1000    |   1000   |
...

    Scenario: SSGA is elitist
        Given I have a SSGA algorithm
        When I init the population with \
            50 individuals
        When I study the evolution of the \
            <individual> individual
        When I run the algorithm during \
            <num_itera> iterations
        Then its fitness is always better

    Examples:
        | individual | num_itera |
        |    best    |  1000     |
        |    worst   |  1000     |
        |    mean    |  1000     |
```

Fig. 10. Fichero *run.feature*

gura 10 muestra la especificación de dichas comprobaciones.

V. Pruebas con números seudoaleatorios

Hasta ahora hemos abordado el problema del no-determinismo por medio de pruebas sobre características que debe cumplir un algoritmo, en vez de sobre los resultados concretos, es decir, tests de caja negra. Sin embargo, sería deseable poder complementarlo con tests de caja blanca.

Para conseguir hacer tests de caja blanca más cercanos al código sería necesario que el resultado fuese reproducible, pero el uso de número seudoaleatorios lo impide. En la práctica, sería posible hacerlo modificando el código para que en vez generar números aleatorios utilizase un conjunto de números generados inicialmente. Sin embargo, este enfoque requeriría una modificación del código para las pruebas, lo que no es deseable.

Una alternativa es utilizar herramientas que permitan sustituir de forma dinámica las clases que realizan una determinada tarea por objetos *mocks* que simulan dicho comportamiento. Un ejemplo, es la librería mocks[5] de Python.

Con una herramienta como ésta, se puede modificar el comportamiento de un método o función para que, durante el test, cuando se vaya a llamar a una determinada función, su ejecución se sustituya por otra (usando *mocks*) con el mismo interfaz y que simula su comportamiento.

En nuestro caso, podemos reemplazar durante el test el código que realiza la generación de números aleatorios por código que devuelve números previamente fijados en el test. De esta manera, se puede simular el comportamiento del código ante distintas secuencias de número seudoaleatorios sin tener que cambiar el código original.

```
    Scenario: Checking crossover with \
              seudorandoms and alpha=0
        Given I have a SSGA algorithm
        When I init the population \
            with 50 individuals
        When I set randomly two parents
        When I use pseudorandoms=<seudo_random>
        When I cross with alpha 0
        Then the children is equals to \
            the <criterion> of its parents

    Examples:
        | seudo_random | criterion |
        |    0         |  minimum  |
        |    1         |  maximum  |
        |    0.5       |  mean     |
```

Fig. 11. Escenario de cruce usando seudoaleatorios

```
from mock import patch, Mock

@step('When I use pseudorandoms=([\d.]+)')
def set_pseudorandoms(step, expected_random):
    # Change the random generation into the test
    world.newRandom = patch('earandom.randreal',
                                spec=True)
    # Set the returns when it is called
    world.random = world.newRandom.start()
    expected_random = float(expected_random)
    returns = expected_random*np.ones(dim)
    world.random.return_value = returns
```

Fig. 12. Código que modifica los números seudoaleatorios

Como ejemplo, podemos complementar los tests anteriores sobre el operador de cruce con el test indicado en la Figura 11, en donde la asignación de generación de números aleatorios se realiza de forma sencilla al interpretar la sentencia 'I use pseudorandoms...' con el código asociado indicado en la Figura 12 del fichero *steps.py*. En dicho código se indica,

[5]http://www.voidspace.org.uk/python/mock/

por medio de la función *patch* que cuando durante el test se desee llamar a la función *erandom.randreal*, se llame, en cambio, al objeto *mock* newRandom. Y luego se configura la salida que devuelve cuando se realicen dichas llamadas. Puede consultarse una explicación mucho más detallada en la documentación oficial de la librería *mocks*.

El uso de pruebas automáticas es una herramienta muy útil para detectar errores en el código, pero el comportamiento no-determinista de los AEs se suele ver como un impedimento para poder realizarlas.

En este trabajo se plantean dos estrategias para aplicar pruebas automáticas sobre AEs, a pesar del uso de números seudoaleatorios, que hace que los resultados no sean repetibles.

En la primera, en vez de aplicar tests sobre resultados concretos de un operador, se propone realizar comprobaciones sobre el comportamiento que dichos operadores deben ofrecer. Se muestra el uso de esta estrategia aplicándola a los operadores de inicialización, cruce y ejecución de un SSGA.

La otra estrategia propuesta es utilizar librerías que permiten reemplazar durante los tests el código de generación de números seudoaleatorios por código que devuelve secuencias de números definidos de antemano, para hacer comprobaciones de más bajo nivel, o de caja blanca. Se ha mostrado el uso de esta estrategia para tests adicionales sobre el operador de cruce $BLX - \alpha$.

En conclusión, en este trabajo se proponen estrategias que permiten aplicar pruebas automáticas a las implementaciones de AEs, con lo que se puede detectar más fácilmente errores de implementación, al mismo tiempo que son entendibles por cualquier investigador en AEs sin necesidad de conocer los detalles concretos de su implementación. Por tanto, se prueba que los AEs pueden (y deberían) incorporar pruebas automáticas para mejorar la confianza en la implementación de los algoritmos.

VI. Agradecimientos

Este trabajo fue financiado por los proyectos de investigación TIN2008-05854 y P08-TIC-4173.

Apéndice

Las Figuras 13, 14 y 15 muestran el código de test asociado a las definición de las pruebas del artículo. El código está libremente disponible en la url https://github.com/dmolina/ssgatests.

Referencias

[1] T Bäck, D B Fogel, and Z Michalewicz, Eds., *Handbook of Evolutionary Computation*, IOP Publishing Ltd., Bristol, UK, 1997.

[2] Patrick Siarry and Zbigniew Michalewicz, *Advances in metaheuristics for hard optimization*, Springer-Verlag New York Inc, springer edition, 2007.

[3] G J Myers, C Sandler, T Badgett, and T M Thomas, "The art of software testing," 2004.

[4] K Beck, *Test-driven development: by example*, Addison-Wesley Professional, 2003.

[5] Forrest Shull, Grigori Melnik, Burak Turhan, Lucas Layman, Madeline Diep, and Hakan Erdogmus, "What Do We Know about Test-Driven Development?," *IEEE Software*, vol. 27, no. 6, pp. 16–19, 2010.

[6] Nachiappan Nagappan, E. Michael Maximilien, Thirumalesh Bhat, and Laurie Williams, "Realizing quality improvement through test driven development: results and experiences of four industrial teams," *Empirical Software Engineering*, vol. 13, no. 3, pp. 289–302, Feb. 2008.

[7] R.M. Hierons, "Testing from a nondeterministic finite state machine using adaptive state counting," *IEEE Transactions on Computers*, vol. 53, no. 10, pp. 1330–1342, Oct. 2004.

[8] Gordon Fraser and Franz Wotawa, "Test-Case Generation and Coverage Analysis for Nondeterministic Systems Using Model-Checkers," *International Conference on Software Engineering Advances (ICSEA 2007)*, pp. 45–45, 2007.

[9] D Chelimsky, D Astels, Z Dennis, A Hellesoy, B Helmkamp, and D North, "The RSpec Book: Behaviour Driven Development with RSpec, Cucumber, and Friends," *Pragmatic Bookshelf*, 2010.

[10] Madhuri R. Marri, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte, "An empirical study of testing file-system-dependent software with mock objects," *2009 ICSE Workshop on Automation of Software Test*, pp. 149–153, May 2009.

[11] F Herrera, M Lozano, and A.M. Sanchez, "A Taxonomy for the Crossover Operator for Real-coded Genetic Algorithms," *International Journal of Intelligent Systems*, vol. 18, no. 3, pp. 204–217, 2003.

[12] L.J. Eshelman and J.D. Schaffer, "Real-coded Genetic Algorithms in Genetic Algorithms by Preventing Incest," *Foundation of Genetic Algorithms 2*, pp. 187–202, 1993.

[13] D Whitley, "The GENITOR Algorithm and Selection Pressure: Why Rank-Based Allocation of Reproductive Trials is Best," *Proc. of the Third Int. Conf. on Genetic Algorithms*, pp. 116–121, 1989.

Inicialización

```
@step('I have a SSGA algorithm for dimension (\d+)')
def have_configuration(step,dim):
    dim = int(dim)
    world.ssga = SSGA(fitness,domain,dim,size=popsize)

@step('I init the population with (\d+) individuals')
def init_population(step,popsize):
    world.popsize = int(popsize)
    world.ssga.initPopulation(int(popsize))

@step('population size is (\d+)')
def popsize_right(step,popsize_expected):
    popsize_expected = int(popsize_expected)
    (values_size,values_dim)=world.ssga.population().shape
    assert values_size == popsize_expected

@step('fitness is initialized')
def fitness_right(step):
    fits = world.ssga.population_fitness()
    assert fits.size == world.popsize

@step('all fitness values are different')
def fitness_not_same(step):
    fits = world.ssga.population_fitness()
    # Check the size of removing equals fitness
    assert np.unique(fits).size == world.popsize
```

Selección

```
@step('I select parents with tournament size (\d+)')
def set_parents(step,tsize):
    world.nam_tsize=int(tsize)

@step('the parents are different after (\d+) tests')
def mother_parent_different(step,tests):
    tests = int(tests)

    for i in range(tests):
        [mother,parent]=world.ssga.getParents(world.nam_tsize)
        assert mother != parent

@step('the distance between parents is the longest')
def best_parent(step):
    ssga = world.ssga
    [motherId,parentId]=ssga.getParents(world.nam_tsize)
    population = ssga.population()
    mother = population[motherId]
    parent = population[parentId]
    distances = [utils.distance(population[i],mother) for i in range(world.popsize)]
    max_distances = np.array(distances).max()
    distance = utils.distance(parent, mother)
    assert distance == max_distances
```

Fig. 13. Fichero *steps.py* (Parte 1)

Cruce

```
@step('I set the same parent as mother')
def set_parents_same(self):
    population = world.ssga.population()
    motherId = random.randint(0, world.ssga.popsize)
    world.mother = population[motherId]
    world.parent = world.mother

@step('I set randomly two parents')
def cross_set_random(step):
    population = world.ssga.population()
    [motherId,parentId] = random.randint(0, world.ssga.popsize, 2)
    world.mother = population[motherId]
    world.parent= population[parentId]

@step('I cross with alpha ([\d.]+)')
def apply_cross(self,alpha):
    alpha = float(alpha)
    world.children = world.ssga.cross(world.mother,world.parent,alpha)

@step('The children is the same')
def has_same_children(self):
    assert utils.distance(world.children, world.parent)==0

@step('The children is between them')
def is_between_then(self):
    parents = np.array([world.mother,world.parent])
    min_parent = np.amin(parents,axis=0)
    max_parent = np.amax(parents,axis=0)
    assert (world.children >= min_parent).all(), "Brokes inferior limit"
    assert (world.children <= max_parent).all(), "Brokes superior limit"

@step('the parents are different after (\d+) tests')
def mother_parent_different(step,tests):
    tests = int(tests)

    for i in range(tests):
        [mother,parent]=world.ssga.getParents(world.nam_tsize)
        assert mother != parent

@step('the distance between parents is the longest')
def best!_!parent(step):
    ssga = world.ssga
    [motherId,parentId]=ssga.getParents(world.nam_tsize)
    population = ssga.population()
    mother = population[motherId]
    parent = population[parentId]
    distances = [utils.distance(population[i],mother) for i in range(world.popsize)]
    max_distances = np.array(distances).max()
    distance = utils.distance(parent, mother)
    assert distance == max_distances
```

Fig. 14. Fichero *steps.py* (Parte 2)

Ejecución

```
\begin{lstlisting}
def measureFitness(*args, **kwargs):
    if world.fitEval == []:
        world.fitEval.append(world.get_fitness(world.ssga.population_fitness()))

    yield aspects.proceed
    world.fitEval.append(world.get_fitness(world.ssga.population_fitness()))

@step('I study the evolution of the (\w+) individual')
def study_population(self, individual):

    if individual == 'best':
        world.get_fitness = np.min
    elif individual == 'worst':
        world.get_fitness = np.max
    elif individual == 'mean':
        world.get_fitness = np.mean
    else:
        assert False, "Error, individual '%s' is not known" %individual

    world.wrap_id = aspects.with_wrap(measureFitness, SSGA.updateWorst)


@step('I run the algorithm during (\d+) iterations')
def run_iterations(self,numevals):
    numevals = int(numevals)
    world.fitEval = []
    world.numevals = 0
    world.ssga.run(maxeval=numevals)

@step('they were evaluated (\d+) solutions')
def check_eval(self,solutions):
    assert world.numevals == int(solutions)

@step('its fitness is always better')
def check_fitness(self):
    size = len(world.fitEval)
    fitEval = world.fitEval

    for eval in xrange(size-1):
        before = float(fitEval[eval])
        after = float(fitEval[eval+1])
        assert after <= before

    aspects.without_wrap(measureFitness, world.ssga.updateWorst)
```

Fig. 15. Fichero *steps.py* (Parte 3)