Apunte de Ingeniería del Software II - Primera Parte

Tomás C.

Segundo cuatrimestre, 2022

Acerca de este apunte

Este es un compilado con los conceptos teóricos de la materia "Ingeniería de Software II", escrito en base a las clases teóricas (tanto clases presenciales, como videos, como apuntes propios sobre las clases) del segundo cuatrimestre de 2022.

La materia se dividió en dos partes: testing automatizado (a cargo de Juan Pablo Galeotti) y verificación de sistemas concurrentes (a cargo de Sebastián Uchitel), y de esa manera también se divide este apunte. En cuanto a la estructura, cada sección corresponde aproximadamente a una clase teórica de la materia. Esta es la **primera parte** del apunte, correspondiente a la primera mitad de la materia (testing automatizado).

Esto no pretende ser un *resumen* (en el sentido de acortar las cosas, como se puede ver por la cantidad de páginas) de los contenidos de la materia, sino que la idea de este apunte es ser una recopilación de lo estudiado en clase, que sea útil para preparar el examen final. Dicha utilidad podría no existir más pasadas las fechas de examen del verano 2022/2023, dado que la materia suele cambiar su contenido.

En el caso de que este apunte sea utilizado por otras personas, advierto que puede contener errores conceptuales, errores de redacción, errores de *tipeo*, errores de ortografía (i.e. "herrorez") o gramática, errores adentro de errores, etcétera.

Título completo: *Apunte para el examen final de Ingeniería del Software II.*

Autor: Tomás C.

Fecha de finalización: 7 de diciembre de 2022.

Índice

I Testing Automatizado de Software 1

1. Introducción al Testing 1
1.1. Definición . . . 1
1.2. Automatización del testing . . . 1
1.3. Generación automática de casos de test . . . 1
1.4. Niveles de testing . . . 2
1.5. Etapas de un test . . . 2
1.6. Defectos, infecciones y fallas . . . 3
1.7. Criterios de adecuación . . . 3
1.7.1. Testing estructural . . . 4
1.7.2. Statement testing . . . 4
1.7.3. Edge testing . . . 4
1.7.4. Branch testing . . . 4
1.7.5. Límites de la cobertura . . . 5

2. Mutation Testing 6
2.1. Mutantes . . . 6
2.1.1. Mutación orientada a objetos . . . 7
2.2. Mutation score . . . 7
2.3. Mutantes equivalentes . . . 8
2.4. Performance . . . 8
2.4.1. Mutación fuerte y débil . . . 8

3. Random Testing 10
3.1. Presupuesto . . . 10
3.2. Uso en programación orientada a objetos . . . 11
3.2.1. Random testing guiado por feedback . . . 12
3.2.2. Secuencias equivalentes . . . 12
3.2.3. Oráculos . . . 13
3.3. Limitaciones . . . 14
3.4. Resumen . . . 14

4. Ejecución Simbólica 15
4.1. Constraint solving . . . 15

4.1.1. DPLL . . . 15
4.1.2. CDCL . . . 16
4.1.3. Limitaciones de los SAT-solvers . . . 16
4.2. Aplicación para testing . . . 17
4.3. Parametrized Unit Tests . . . 17
4.4. Heap constraints . . . 18
4.5. Limitaciones de la ejecución simbólica (estática) . . . 18
4.6. Ejecución simbólica dinámica . . . 19
4.6.1. Resumen . . . 20

5. Search-based testing **21**
5.1. Hill climbing . . . 21
5.2. Algoritmos genéticos . . . 22
5.2.1. Población inicial . . . 23
5.2.2. Función de fitness . . . 23
5.2.3. Condición de parada . . . 23
5.2.4. Roulette wheel selection . . . 23
5.2.5. Rank selection . . . 24
5.2.6. Tournament selection . . . 25
5.2.7. Crossover . . . 25
5.2.8. Mutación . . . 25
5.3. Branch distance . . . 26
5.4. Limitaciones de la branch distance . . . 26
5.4.1. Control-dependencia . . . 26
5.4.2. Approach level . . . 29
5.4.3. Combinando branch distance y approach level . . . 29
5.5. Testability transformation . . . 30
5.5.1. Flag problem . . . 30
5.5.2. Predicados anidados . . . 31
5.5.3. Otras transformaciones . . . 31
5.6. Búsqueda con único objetivo vs. múltiples objetivos . . . 32
5.6.1. Whole-test suite generation . . . 32
5.6.2. *Dominancia* . . . 33
5.6.3. Criterios de cobertura y EvoSuite . . . 33
5.6.4. Combinación de criterios de cobertura . . . 35
5.6.5. Post-procesamiento del test suite . . . 35

5.7. Variantes de algoritmos genéticos . . . 36
5.7.1. Algoritmo genético estándar . . . 36
5.7.2. Algoritmo genético monotónico . . . 37
5.7.3. Algoritmo genético steady-state . . . 38
5.7.4. Algoritmo genético breeder . . . 39
5.7.5. Algoritmo genético celular . . . 40
5.7.6. Algoritmo genético 1 + (λ, λ) . . . 41
5.7.7. Algoritmo evolutivo $\mu + \lambda$ . . . 42
5.7.8. Algoritmo evolutivo (μ, λ) . . . 43
5.8. Herramientas basadas en algoritmos genéticos . . . 44

6. Fuzzing **47**
6.1. Fuzzing en lenguajes con intérpretes . . . 47
6.2. Black box fuzzing . . . 48
6.3. Grey box fuzzing . . . 49
6.3.1. Boosted grey box fuzzing . . . 49
6.4. Sanitizers . . . 51
6.5. White box fuzzing . . . 51

Parte I

Testing Automatizado de Software

1. Introducción al Testing

1.1. Definición

Definición según ANSI/IEEE 1059[1]: "[el testing] es un proceso de analizar un objeto de software para detectar diferencias entre las condiciones existentes y las requeridas (i.e. detectar defectos), y evaluar las características del objeto de software".

Límites del testing: usando testing, se puede mostrar la presencia de *bugs*, pero no se puede mostrar su ausencia. Es decir, se puede probar que un programa falla, pero no que no falla (a no ser que sea posible probar todas las posibles entradas para el programa).

1.2. Automatización del testing

- La ejecución de tests es algo ya estándar en la actualidad; un proceso por el cual se ejecuta un software bajo test usando un test suite, y una herramienta decide automáticamente si el test pasa o falla.
- Otra cosa distinta es la *generación automática* de tests (generar esas test suites de manera automática), que no está tan estandarizada como la automatización de la ejecución y decisión sobre tests.
- La tarea de *escribir tests* sigue siendo, en general, manual. Este suele ser un proceso tedioso, costoso e incompleto.
- La idea sería entonces aprovechar el poder de cómputo disponible actualmente para generar casos de test automáticamente, mejorando la eficiencia y la calidad en el proceso de creación de software.

1.3. Generación automática de casos de test

Esto consiste en que, a partir de un programa, exista un **test generator** (*generador de tests*), que genera automáticamente casos de test. Es decir, derivar casos de test a partir del código.

Se puede usar para distintos escenarios, por ejemplo:

1. **Detección de fallas**: cómo detectar y clasificar una falla en un programa. El test generator genera casos de test, se ejecuta el programa bajo test con esas entradas, y se utiliza un **oráculo** para clasificar una falla. Un oráculo es una entidad que indica, en tiempo de ejecución, si el resultado de la ejecución del programa es correcto o no; funciona como una especie de post-condición. El problema es que no siempre hay un oráculo disponible para un programa cualquiera. Pueden ser oráculos:

[1] `https://ieeexplore.ieee.org/document/838043`

- Especificaciones formales.
- Aserciones en el programa.
- Aserciones en el test.
- Contratos en el código.
- Oráculos manuales.

2. **Testing de robustez**: esto es usar los casos de test para verificar si el programa bajo test *crashea* o no. Esto puede ser útil cuando no hay un oráculo para detectar fallas; el hecho de que el programa *crashee* funciona como un *oráculo implícito*.

3. **Testing de regresión**: consiste en re-ejecutar tests para asegurarse de que el software previamente desarrollado y testeado se siga comportando correctamente después de hacerle cambios. Si esto no ocurre, se dice que ocurrió una *regresión*. La idea entonces es usar resultados de una versión anterior del programa para testear una nueva versión.

1.4. Niveles de testing

Un **test unit** (*unidad de test*) consiste de:

1. **Test de sistema** (o *end-to-end testing*): es probar el resultado de usar un sistema. Suele involucrar el trabajo de todo el equipo de desarrollo.

2. **Test de integración**: probar funcionamiento de la interacción entre unidades/módulos desarrollados por diferentes personas.

3. **Test de unidad**: es verificar el comportamiento de una unidad, normalmente desarrollada por una persona o un grupo reducido de ellas.

Como se puede ver, la división entre niveles de testing no es únicamente por el artefacto de software que se analiza, sino también por el tamaño del grupo humano responsable de ese artefacto.

1.5. Etapas de un test

Un test consiste de tres etapas:

1. **Setup**: preparación requerida antes de ejercitar el test.

2. **Ejercitación**: es el código que ejercita la funcionalidad bajo test.

3. **Chequeo**: código que verifica la respuesta obtenida contra el resultado esperado (suele ser en forma de `asserts` de algún tipo). Esta es la noción del oráculo.

A su vez, hay una diferencia entre test case y test suite:

- **Test case**: compuesto por código para ejecutar las tres etapas del test.
- **Test suite**: conjunto de test cases.

1.6. Defectos, infecciones y fallas

El proceso por el cual se genera una falla en un programa se puede esquematizar de la siguiente manera:

1. El programador introduce un **defecto** en el código fuente del programa.
2. El código se ejecuta, y el defecto introducido crea una **infección**.
3. La infección se *propaga* en la ejecución de otros componentes del programa (que podrían no tener defectos).
4. La infección genera una **falla**, es decir un evento que puede ser observado por un observador externo, que no se corresponde con el comportamiento esperado para el programa.

Así, existen diferentes tipos de *errores*:

- **Error**: desviación no intencional de lo que se espera del programa.
- **Defecto**: error en el código del programa, específicamente uno que puede crear una infección, que podría conducir a una falla.
- **Infección**: un error en el estado del programa, específicamente uno que puede llevar a una falla.
- **Falla**: un error visible externamente en el comportamiento del programa.

Notar que la infección puede no llegar a producirse a pesar de que exista un defecto (por ejemplo, si no se llega al estado que genera que ese defecto se *dispare*). Por lo tanto, existen ciertos desafíos que enfrentar en el testing:

1. Hay que *disparar* el error en cuestión: ejecutar el defecto, hacer que la infección se propague, y resulte en una falla (cubrimiento máximo del comportamiento).
2. Hay que *reconocer* el error como tal (oráculo).
3. Hay que identificar *funcionalidad faltante* (especificación).

Y todo esto debería poder hacerse **eficientemente**.

1.7. Criterios de adecuación

Un criterio de adecuación de test es un predicado que es verdadero o falso para un par $\langle programa, test\ suite \rangle$. Normalmente se expresan en forma de una regla.

Estos criterios sirven para determinar *qué tan buenos* son los tests, para poder saber cuándo una test suite es “lo suficientemente buena”.

1.7.1. Testing estructural

El **control flow graph** (CFG, o *diagrama de control de flujo*) puede servir como herramienta a partir de la cual formar un criterio de adecuación. Cuantos más nodos (o ejes, ramas...) sean cubiertos, más chances hay de descubrir una falla.

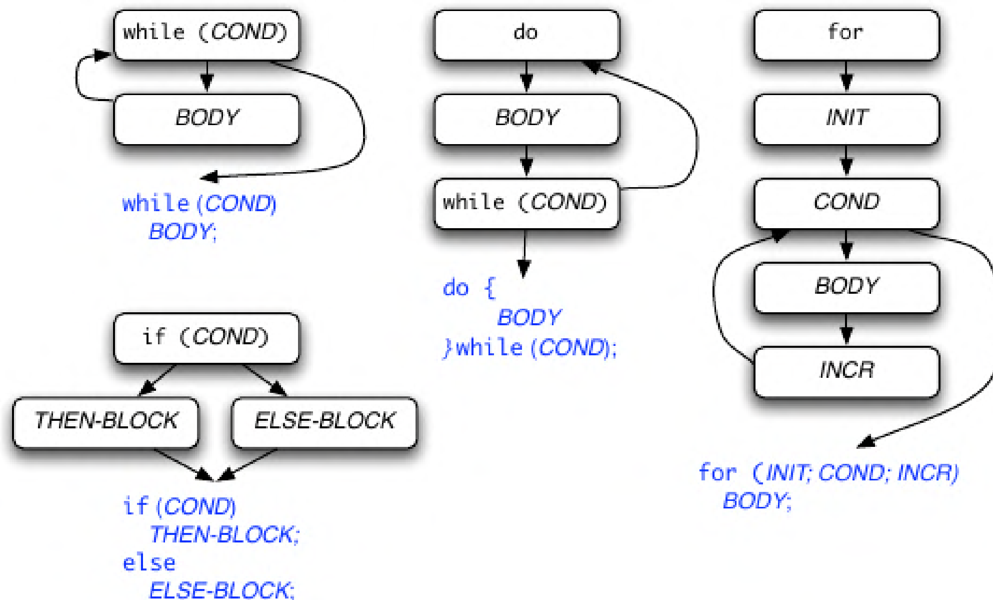


Figura 1: Patrones del control de flujo para distintas estructuras que pueden aparecer en el código (control flow patterns).

1.7.2. Statement testing

- Criterio de adecuación: cada statement (cada nodo en el CFG) debe ser *ejecutado al menos una vez*.
- Idea: un defecto en un statement sólo puede ser revelado ejecutando el defecto.
- Cobertura: $\frac{\#\text{ statements ejecutados}}{\#\text{ statements}}$

1.7.3. Edge testing

- Criterio de adecuación: cada arco en el CFG debe ser *ejecutado al menos una vez*.
- Esto incluye a statement testing, pues al recorrer todos los arco se recorren todos los nodos.
- Cobertura: $\frac{\#\text{ arcos ejecutados}}{\#\text{ arcos}}$

1.7.4. Branch testing

- Criterio de adecuación: cada branch en el CFG debe ser *ejecutado al menos una vez*. Los branches son las decisiones (guardas de `if`, `while`...).
- No incluye (o *subsume*) ni statement testing ni edge testing, pues se puede ejercitar todas las decisiones del programa pero no cubrir todos los arcos o nodos del CFG.

- Es el criterio más comúnmente usado en la industria.
- Cobertura: $\frac{\#\text{ branches ejecutados}}{\#\text{ branches}}$

1.7.5. Límites de la cobertura

La información de cobertura es una dimensión para medir la *potencia* de los tests, pero una cobertura de 100 % no garantiza que no haya fallas sin detectar.

Ejecutar todo el programa no suele ser suficiente: de alguna manera habría que chequear el comportamiento funcional del programa.

Al no saber donde se encuentran los defectos en el código, es difícil saber cuánto del código es chequeado contra comportamiento esperado. Lo que sí se puede saber es qué errores aparecieron previamente...

2. Mutation Testing

La idea será entonces aprender de los errores previos para prevenir que ocurran nuevamente. A partir de la *simulación* de errores anteriores, se puede comprobar si los defectos simulados pueden ser detectados por la test suite actual. Esta técnica se conoce como *fault-based testing* o *mutation testing.*

El procedimiento consiste en generar versiones alternativas del programa (llamadas **mutantes**), y ejecutar los tests sobre esos mutantes. Si el test suite falla cuando antes de mutar era exitoso, entonces fue capaz de detectar la desviación de comportamiento introducida por la mutación. En caso contrario, harán falta más/mejores tests, dado que esa desviación no fue detectada (pasaron los tests).

Se juzga la efectividad de un test suite para encontrar errores midiendo cuán bien puede encontrar defectos "artificiales". Esto será válido sólo si los *bugs plantados* son **representativos** de los *bugs reales*: no necesariamente iguales, pero las diferencias no deberían afectar la selección.

2.1. Mutantes

Un mutante es una versión **levemente modificada** del programa original. Se generan a partir de **cambios sintácticos válidos** (válidos porque el código debe ser *compilable*), y suelen ser cambios simples, que pueden estar simulando, por ejemplo, un *typo* en el código.

Para generar mutantes, se pueden usar **operadores de mutación**: reglas para derivar mutantes a partir de un programa. Para algunos lenguajes de programación, se han definido diferentes operadores de mutación genéricos; en general, suelen estar basados en fallas reales, representando errores típicos.

Ejemplos de operadores de mutación:

- ABS - Absolute Value Insertion: cambiar un valor numérico por su valor absoluto (positivo o negativo).
- AOR - Arithmetic Operator Replacement: reemplazar un operador aritmético por otro.
- ROR - Relational Operator Replacement: reemplazar un operador de comparación por otro.

id	operator	description	constraint
Operand Modifications			
crp	constant for constant replacement	replace constant $C1$ with constant $C2$	$C1 \neq C2$
scr	scalar for constant replacement	replace constant C with scalar variable X	$C \neq X$
acr	array for constant replacement	replace constant C with array reference $A[I]$	$C \neq A[I]$
scr	struct for constant replacement	replace constant C with struct field S	$C \neq S$
svr	scalar variable replacement	replace scalar variable X with a scalar variable Y	$X \neq Y$
csr	constant for scalar variable replacement	replace scalar variable X with a constant C	$X \neq C$
asr	array for scalar variable replacement	replace scalar variable X with an array reference $A[I]$	$X \neq A[I]$
ssr	struct for scalar replacement	replace scalar variable X with struct field S	$X \neq S$
vie	scalar variable initialization elimination	remove initialization of a scalar variable	
car	constant for array replacement	replace array reference $A[I]$ with constant C	$A[I] \neq C$
sar	scalar for array replacement	replace array reference $A[I]$ with scalar variable X	$A[I] \neq X$
cnr	comparable array replacement	replace array reference with a comparable array reference	
sar	struct for array reference replacement	replace array reference $A[I]$ with a struct field S	$A[I] \neq S$
Expression Modifications			
abs	absolute value insertion	replace e by abs (e)	$e < 0$
aor	arithmetic operator replacement	replace arithmetic operator ψ with arithmetic operator ϕ	$e_1 \psi e_2 \neq e_1 \phi e_2$
lcr	logical connector replacement	replace logical connector ψ with logical connector ϕ	$e_1 \psi e_2 \neq e_1 \phi e_2$
ror	relational operator replacement	replace relational operator ψ with relational operator ϕ	$e_1 \psi e_2 \neq e_1 \phi e_2$
uoi	unary operator insertion	insert unary operator	
cpr	constant for predicate replacement	replace predicate with a constant value	
Statement Modifications			
sdl	statement deletion	delete a statement	
sca	switch case replacement	replace the label of one case with another	
ses	end block shift	move } one statement earlier and later	

Figura 2: Otros operadores de mutación típicos.

2.1.1. Mutación orientada a objetos

También hay operadores de mutación para programas orientados a objetos:

- AMC - Access Modifier Change
- HVD - Hiding Variable Deletion
- HVI - Hiding Variable Insertion
- OMD - Overriding Method Deletion
- OMM - Overridden Method Moving
- OMR - Overridden Method Rename
- SKR - Super Keyword Deletion
- PCD - Parent Constructor Deletion
- ATC - Actual Type Change
- DTC - Declared Type Change
- PTC - Parameter Type Change
- RTC - Reference Type Change
- OMC - Overloading Method Change
- OMD - Overloading Method Deletion
- AOC - Argument Order Change
- ANC - Argument Number Change
- TKD - this Keyword Deletion
- SMV - Static Modifier Change
- VID - Variable Initialization Deletion
- DCD - Default Constructor 2

Figura 3: Mutación orientada a objetos

2.2. Mutation score

La idea es analizar para cuántos mutantes el test suite falla si antes no fallaba en el programa original. Los mutantes para los cuales esto ocurre se llaman **mutantes muertos**, y los que no

son detectados son los **mutantes vivos**. Si hay mutantes vivos, significa que el test suite no es suficiente, hay que agregar más test cases.

Es decir, a más mutantes muertos, mejor es el test suite. Esto se cuantifica con el **mutation score**:

$$\text{Mutation Score} = \frac{\#\ \text{Mutantes muertos}}{\#\ \text{Mutantes}}$$

2.3. Mutantes equivalentes

Un mutante podría ser semánticamente equivalente al programa original, ya que una mutación es un cambio sintáctico, y esto podría dejar la semántica inalterada.

Se dice que un programa es un mutante no equivalente cuando existe una entrada para el programa (dentro del universo de posibles entradas) tal que el resultado para el programa original es distinto que para el mutante.

Los mutantes equivalentes son difíciles de detectar (es un problema indecidible), dado que pueden ser alcanzados por la ejecución, pero puede que no se genere una infección, o esta no se propague.

2.4. Performance

Hay muchos posibles mutantes para un programa, dado que para cada línea del código se pueden generar muchos mutantes (por la gran cantidad de operadores de mutación posibles). Cada operador de mutación resulta en muchos mutantes, y cada mutante es un programa, que debe ser compilado. Además, cada test individual debe ser ejecutado para cada mutante. Todo esto puede resultar computacionalmente muy costoso.

Una alternativa es no ejecutar los mutantes que mutan líneas de código no cubiertas por el test suite (pero esto no siempre es sencillo).

Para evitar tener que compilar todos los mutantes, aparece la noción de **meta-mutante**: codificar todos los mutantes dentro de un único programa (como un `switch`, donde se selecciona qué variante de mutante usar).

2.4.1. Mutación fuerte y débil

El tipo de mutación que se vio hasta ahora corresponde a la **mutación fuerte**: se considera un mutante detectado si la mutación se propaga hasta algún comportamiento observable (el resultado de la ejecución, e.g. un resultado erróneo, un *crash*, etc.); es decir, una falla.

En contraposición con esto, existe la **mutación débil**, que considera las mutaciones que afectan el estado del programa (infecciones) de manera inesperada. Para eso, se debe observar el estado interno del programa, comparando los estados antes y después de la mutación; si el estado es distinto después de mutar, entonces se considera muerto al mutante.

Las consecuencias de considerar este tipo de mutación son:

- Hay menos mutantes equivalentes.
- Cuando se llega a una infección, se puede detener la ejecución.
- No hay garantías de que la infección se fuera a propagar.

El resultado de esto es una mejora considerable en eficiencia (hasta el 50 % del tiempo de ejecución).

3. Random Testing

El random testing consiste en generar casos de test de manera aleatoria. Esto se hace con un **random driver**, que ejecuta el programa bajo test usando valores aleatorios como entrada.

```
Let M(p0:T0,…,pk:Tk) be a program
while budget is not empty
 For each input parameter pi:Ti
  If Ti is primitive vi:=get random Ti
  Else vi:=null
 Add M(v0,…,vk) to tests
Return tests
for each M(v0,…,vk) in tests
 run x=M(v0,…,vk) and collect value x
 add "x=M(v0,…,vk);assert x==val(x)"
 to assert_tests
return assert_tests
```

Figura 4: Pseudocódigo para generar $k+1$ entradas aleatorias de tipos de datos primitivos, y ejecutar el programa con dichas entradas.

Algunos ejemplos de herramientas que implementan esta técnica son:

- Monkey: genera eventos de interfaz de usuario al azar para aplicaciones Android.
- Randoop: estándar *de facto* del random testing para programas escritos en Java.

3.1. Presupuesto

El **presupuesto de testing** (budget) es la cantidad de recursos que se le asigna a la generación de casos de test. Esto puede estar expresado en tiempo, cantidad de operaciones, cantidad de tests, etc.

Se suele establecer un presupuesto para esta tarea ya que normalmente no tiene una cota superior en cuanto a recursos a destinar, y ciertos objetivos más *naturales* pueden ser inviables; por ejemplo, si el criterio de parada es alcanzar un 100 % de cubrimiento del código, podría pasar que la generación de tests nunca termine, pues puede haber código inalcanzable.

La idea entonces es generar entradas posibles para el programa hasta que se agote el presupuesto de testing, y luego ejecutar el programa con cada entrada, para clasificar el resultado de acuerdo al comportamiento esperado. Si se dispone de un oráculo, esto se puede hacer directamente, y en caso contrario se puede realizar testing de regresión.

Observación. Usar el tiempo de generación como presupuesto no suele ser buena idea, ya que este tiempo dependerá de las características de hardware y software de la computadora que ejecute el código.

3.2. Uso en programación orientada a objetos

En su forma más básica, random testing genera entradas para tipos de datos primitivos. Es decir, si un parámetro tiene un tipo básico (números, booleanos, caracteres, etc.), entonces basta con elegir un valor al azar para usarlo como entrada.

Un problema que se puede presentar es el manejo de excepciones: en un lenguaje orientado a objetos, que ocurra una excepción no siempre es sinónimo de una falla (e.g. acceder a una posición inválida de una lista debería resultar en una excepción). Por lo tanto, no es tan fácil como decidir "toda excepción significa que hay algo mal".

En caso de que un parámetro tenga un tipo T genérico, se puede elegir aleatoriamente, por ejemplo, entre el valor `null` y una invocación al constructor por defecto, de existir. Pero con eso sólo se puede conseguir una instancia *básica* de un objeto; de alguna manera hay que realizar modificaciones al resultado de la construcción, para poder testear el comportamiento del objeto. Es necesario crear **secuencias de invocaciones a métodos** para construir instancias de mayor interés.

Para esto último es que se construye un **catálogo** de métodos (posibles métodos que se pueden usar para crear instancias de los objetos a testear). A partir de este catálogo, se puede usar un generador de secuencias de invocaciones a los métodos catalogados. Estas secuencias tendrán como casos base una serie de secuencias para construir objetos válidos, que será una lista inicial para construir nuevas secuencias de métodos.

```
while budget is not empty
   choose M(p1:T1,…,pk:Tk):Tr from catalog C
   for each input parameter of type pi:Ti
     choose randomly Si from tests s.t. returns type Ti

   build new sequence Snew=S1;…;Sk;Tr vnew=M(v1,…,vk)
   Add Snew to tests

assert_tests := add_assertions(tests)
Return assert_tests
```

Figura 5: Pseudocódigo para generar secuencias de invocaciones a métodos.

A medida que se generan más secuencias de invocaciones, se construye *recursivamente* sobre la lista existente de invocaciones. Notar que es posible que algunas secuencias de invocaciones generen una excepción; estas secuencias pueden ser útiles (para verificar que las excepciones ocurren cuando corresponde), pero no deberían usarse para construir nuevas secuencias, dado que sería un desperdicio de tiempo de cómputo generar cada vez más entradas inválidas. Es por eso que como refinamiento del random testing aparece el **random testing guiado por feedback**.

3.2.1. Random testing guiado por feedback

La idea es ir ejecutando las secuencias de métodos generadas, examinar los resultados y modificar el mecanismo de generación según corresponda (e.g. si una secuencia genera una excepción, no usarla para construir nuevas secuencias).

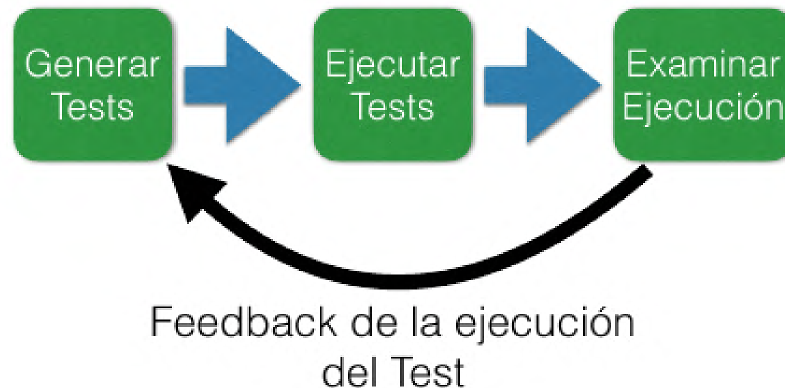


Figura 6: Esquema que muestra el uso de feedback para la generación de tests aleatorios.

Es un proceso *cíclico*, en el sentido de que usa el resultado de una secuencia de invocaciones a métodos para decidir si usar o no esas secuencias para construir nuevas.

Al final del proceso, se generan aserciones tanto para aquellas secuencias que generaron excepciones, como para las que construyeron instancias válidas.

```
while  budget is not empty

   choose M(p1:T1,…,pk:Tk):Tr from catalog C
   for each input parameter of type pi:Ti
      choose randomly Si from normal_tests s.t. returns Ti

   build new sequence "Snew=S1;…;Sk;Tr vnew=M(v1,…,vk)"
   Run Snew
   If execution signaled an exception
      Add Snew to exception_tests
   Else
      Add Snew to normal_tests

assert_tests := add_assertions(normal_tests+exception_tests)
Return assert_tests
```

Figura 7: Pseudocódigo para generar secuencias de invocaciones a métodos teniendo en cuenta el feedback.

3.2.2. Secuencias equivalentes

Otro problema que puede surgir con esta estrategia es que se generen secuencias distintas de métodos, que produzcan el mismo objeto (e.g. crear una lista vacía resulta en lo mismo que crearla, agregarle un elemento y quitárselo).

Evitar que se generen secuencias de invocaciones equivalentes no es fácil, pero sí se puede evitar que se agreguen ambas al conjunto de tests:

```
while  budget is not empty

  choose M(p1:T₁,…,pk:Tₖ):Tᵣ from catalog C
  for each input parameter of type pᵢ:Tᵢ
     choose randomly Sᵢ from normal_tests s.t. returns Tᵢ

  build new sequence "Snew=S₁;…;Sₖ;Tᵣ vnew=M(v₁,…,vₖ)"
  Run Snew
  If execution signaled an exception
    Add Snew to exception_tests
  Else If execution created a new instance
    Add Snew to normal_tests
  Else
    Discard sequence Snew

assert_tests := add_assertions(normal_tests+exception_tests)
Return assert_tests
```

3.2.3. Oráculos

Si se dispone de un oráculo, se puede clasificar los tests entre los que pasan y los que fallan. Se pueden usar los `asserts` dentro del código, pero esto asume que las condiciones verificadas valen independientemente del valor (y estas suposiciones muchas veces deberían ser chequeadas por el test).

Por eso es que a veces no alcanza con esto, con lo cual aparece como alternativa utilizar **oráculos implícitos**; por ejemplo, propiedades generalmente válidas (e.g. comprobar que la relación `equals()` sea de equivalencia).

Si se puede obtener este tipo de propiedades, es posible agregar una secuencia de métodos a un conjunto de tests fallidos cuando esta viola un *oráculo*:

```
while  budget is not empty

  choose M(p1:T₁,…,pk:Tₖ):Tᵣ from catalog C
  for each input parameter of type pᵢ:Tᵢ
     choose randomly Sᵢ from normal_tests s.t. returns Tᵢ

  build new sequence "Snew=S₁;…;Sₖ;Tᵣ vnew=M(v₁,…,vₖ)"
  Run Snew
  If execution violated a Oracle
    Add Snew to failing_tests
  Else If execution signaled an exception
    Add Snew to exception_tests
  Else If execution created a new instance
    Add Snew to normal_tests
  Else
    Discard sequence Snew

passing_tests := add_assertions(normal_tests+exception_tests)
Return passing_tests,failing_tests
```

La herramienta Randoop implementa varios oráculos implícitos que se usan de manera

similar a lo mostrado en el pseudocódigo.

3.3. Limitaciones

La principal limitación del random testing es que, en ciertos programas, puede haber líneas de código que tengan una probabilidad muy baja de ejecutarse con una entrada aleatoria. Por ejemplo, en el siguiente fragmento de código, la línea 7 sólo sería cubierta si $y \geq 0$, y $x = 10000$.

```
1. def testme(x, y):
2.  if (y<0):
3.     return -x
4.  else:
5.    z = x - y
6.    if (y+z=10000):
7.      raise Exception(“error”)
8.    else:
9.      return z
```

Figura 8

Asumiendo que la asignación de valores aleatorios para x e y es equiprobable, y que el rango de representación de los enteros es de 2^{32} valores posibles, entonces y tiene una probabilidad de $\frac{2^{31}}{2^{32}} = 50\,\%$ de cumplir la condición, pero x tiene una probabilidad de $\frac{1}{2^{32}} \sim 0\,\%$ de hacerlo. Con lo cual, la probabilidad de cubrir la línea 7 con una entrada aleatoria es extremadamente baja.

3.4. Resumen

- Es una búsqueda (casi) completamente no guiada.
- Si una técnica sistemática no es mejor que random testing, entonces no es valiosa.
- Es barata y fácil de implementar.
- Funciona bastante bien en muchos casos.
- Tiene problemas para alcanzar líneas de código con condiciones muy específicas.

4. Ejecución Simbólica

Como su nombre lo indica, la ejecución simbólica consiste en ejecutar un programa *simbólicamente*: analizar un programa para determinar qué entradas producen que cada parte de su código sea ejecutado.

Se suele esquematizar usando el árbol de ejecución para el programa, y analizando qué caminos del árbol son cubiertos para una entrada determinada.

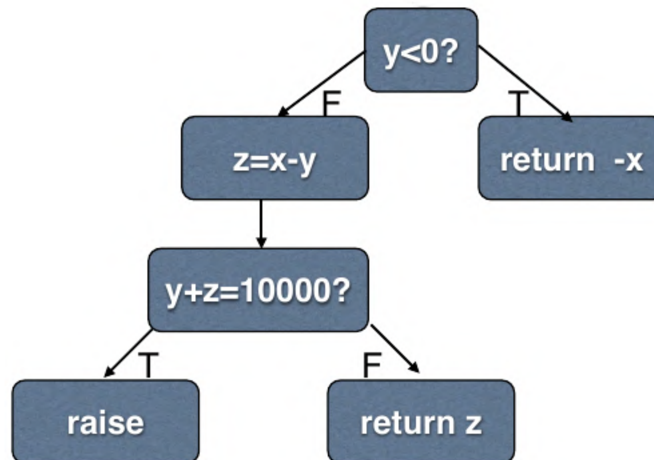


Figura 9: CFG del programa de la Figura 8

En una ejecución simbólica, cada variable del programa tiene un valor en un "estado simbólico", y cada camino de ejecución supone una serie de condiciones sobre esos estados. Estas condiciones forman la llamada **condición de camino** (*path condition*).

Las condiciones de camino no son entradas del programa, sino restricciones para posibles entradas. Estas restricciones se pueden obtener analizando el árbol de ejecución del programa, y obtener entradas que las satisfagan puede ser sencillo como tarea manual en programas pequeños; pero para automatizar este proceso y proveer más escalabilidad, es necesario contar con una forma de generar automáticamente entradas que verifiquen las condiciones de camino.

4.1. Constraint solving

Un constraint solver (SAT-solver, *solvente de teoremas*) es un programa que resuelve fórmulas lógicas descritas en un lenguaje formal (normalmente en CNF: Forma Normal Conjuntiva). Lo que decide el programa es si una fórmula dada es *satisfactible* (SAT), *insatisfactible* (UNSAT), o, en caso de no poder llegar a ninguna conclusión, *desconocido* (UNKNOWN / TIMEOUT). En caso de que la fórumla sea SAT, el constraint solver dará un valor para cada variable.

4.1.1. DPLL

Los SAT-solvers suelen estar basados en el algoritmo DPLL (Davis-Putnam-Logemann-Loveland), que tiene una complejidad temporal en peor caso de $O(2^n)$ (recordar que SAT es un problema NP-completo).

```
Algorithm DPLL
  Input: A set of clauses Φ.
  Output: A Truth Value.

function DPLL(Φ)
   if Φ is a consistent set of literals
       then return true;
   if Φ contains an empty clause
       then return false;
   for every unit clause {l} in Φ
      Φ ← unit-propagate(l, Φ);
   for every literal l that occurs pure in Φ
      Φ ← pure-literal-assign(l, Φ);
   l ← choose-literal(Φ);
   return DPLL(Φ ∧ {l}) or DPLL(Φ ∧ {not(l)});
```

Figura 10: Pseudocódigo del algoritmo DPLL

La idea general del algoritmo es la siguiente:

- Elegir un literal cuyo valor no haya sido fijado aún, y asignarle un valor.
- Propagar el valor del literal, hasta no poder eliminar más literales.
- Si hay un conflicto (una asignación de valores que necesariamente hace que la fórmula sea insatisfactible), hacer backtracking hasta la decisión del primer paso.
- Si no hay conflicto, elegir una nueva variable para asignar.

4.1.2. CDCL

CDCL, o Conflict-Driven Clause Learning, es una mejora del algoritmo DPLL basada en *aprendizaje de cláusulas*. La idea es que cuando se halla un conflicto, se lo *aprende* como una fórmula nueva, que se agrega a las condiciones. De esta manera, se puede encontrar las contradicciones más rápido dentro de la fórmula (ahorrando cómputo al descartar posibilidades más rápidamente).

Esto es, cuando se llega a un conflicto, el algoritmo tiene en cuenta el conjunto minimal de decisiones que llevaron a dicho conflicto, buscando los antecesores inmediatos de los conflictos en un *grafo de dependencias* entre los valores de las variables proposicionales, que el algoritmo debe mantener en cada paso.

Con esto, se agrega una nueva cláusula al conjunto de cláusulas a satisfacer cada vez que aparece un conflicto. Luego, se hace backtracking hasta el punto más *bajo* en el que se decida sobre el valor de las variables involucradas en el conflicto.

4.1.3. Limitaciones de los SAT-solvers

Los SAT-solvers sólo resuelven fórmulas de la lógica proposicional, pero los programas pueden generar condiciones de un nivel expresivo más alto (e.g. con cuantificadores) a partir de la ejecución simbólica.

Para poder decidir sobre fórmulas en una lógica más expresiva (como puede ser la lógica de primer orden), se debe sacrificar la decidibilidad del problema. Las herramientas que hacen esto se llaman **SMT-solvers**, que permiten resolver fórmulas con cuantificadores, y con teorías (básicamente agregando tipos de datos). Estos solvers también suelen estar basados en el

algoritmo DPLL, con el añadido de solventes específicos para las distintas teorías. Existe un lenguaje universal para interactuar con los SMT-solvers, llamado SMT-Lib.

Son ejemplos de SMT-solvers Z3, CVC4, Yices...

4.2. Aplicación para testing

Para aquellas fórmulas que sean satisfactibles, el SAT-solver aporta valores para las variables del programa que satisfagan dichas fórmulas; esto es: estados que cumplen la condición de camino de la ejecución simbólica.

A partir de esto, se puede ir armando cada caso de test, de manera tal que se cubra el 100 % del código (el árbol de ejecución).

En caso de que una fórmula no sea satisfactible (también puede pasar que ocurra un TIMEOUT o UNKNOWN), no existe una entrada que cumpla la condición de camino correspondiente.

```
execution_paths = compute_paths(cfg, limit).iterator()
For each cfg_path in execution_paths
  If budget is not empty
    path_condition = exec_symbolic(cfg_path)

    solution = solve(path_condition)

    if solution is SAT:
      new_test = create_test(entry_f, solution)
      Add new_test to testSuite

return testSuite
```

Figura 11: Pseudocódigo para la generación de casos de test usando ejecución simbólica.

4.3. Parametrized Unit Tests

En el Unit Testing Convencional (CUT), los tests son pequeños programas con test inputs y aserciones. Recientemente, se introdujo el concepto de Unit Testing Parametrizado (PUT), que consiste en parametrizar el test unitario, separando dos dimensiones:

- La especificación del comportamiento esperado (assertions).
- La selección de datos de test relevantes (coverage).

La idea es que un único PUT puede reemplazar múltiples CUTs (reduciendo el tamaño del código), y los datos de test (que son parámetros del PUT) se pueden generar automáticamente: una herramienta puede generar casos de test variando los parámetros del test, que luego es alimentado con estos valores.

Los PUTs apuntan a atenuar dos problemas con los CUTs:

- La falta de datos de test necesarios para ejercitar funcionalidad importante. Los PUTs intentan **aumentar la detección de fallas**.

- La identificación de datos de test que ejercitan el mismo escenario/comportamiento. Los PUTs intentan **eliminar redundancia**.

Varios de los frameworks de testing soportan PUTs (e.g. MBUNIT y NUnit para .NET, JUnit para Java).

Los PUTs tienen **relación con la ejecución simbólica**: se los puede pensar como entry points para la generación de test con ejecución simbólica. Notar que todos los argumentos del PUT deben ser soportados por el constraint solver (normalmente los tipos soportados son los básicos, como números enteros, reales, strings, etc.); la generación de casos de test con esta técnica **no crea secuencias de invocaciones a métodos** como sí ocurría con el random testing orientado a objetos.

4.4. Heap constraints

Cuando un PUT recibe un parámetro de un tipo no soportado por el constraint solver (por ejemplo, algún objeto definido en el programa), hay que extender el lenguaje del constraint solver si se desea utilizar ejecución simbólica. La manera más genérica de extenderlo es agregarle la posibilidad de hacer **referencia a posiciones de memoria**.

Es decir, serán constraints sobre el heap (la memoria del programa). Esto es: al constraint solver se le pide un heap que cumpla con ciertas condiciones; que permita llegar a un cierto estado (snapshot) de la memoria del programa.

El problema es que, al ser esto una solución genérica que habla directamente de la memoria, nada asegura que la lógica de los métodos implementados (públicos) de una clase sea respetada. Podría pasar, por ejemplo, que la manera en la que el constraint solver llegó al estado del heap que se le pidió, no sea una forma válida de construir el objeto en cuestión. Sobre todo puede haber problemas cuando los campos que hay que modificar para llegar al heap objetivo eran campos privados de un objeto: el constraint solver no puede saber cómo crear esa configuración de memoria. Si hay métodos privados, no se sabe cuál es la secuencia de métodos que lleva a generar un cierto grafo de memoria.

Este problema se puede paliar si el lenguaje de programación es **reflexivo**; es decir, cuando el lenguaje permite analizar el mismo programa de manera introspectiva.

Otra manera de asegurar que los objetos creados por el constraint solver para satisfacer las heap constraints cumplan con el invariante interno del objeto es hacer explícito ese invariante de representación en el código, y descartar la instancia creada si no cumple con el mismo.

4.5. Limitaciones de la ejecución simbólica (estática)

La ejecución simbólica que se vio hasta ahora es la versión **estática**; todo el procedimiento no involucra la ejecución concreta del programa, sino que es todo simbólico.

Esto tiene ciertas limitaciones:

- La cantidad total de caminos puede crecer exponencialmente, perjudicando la eficiencia.
- Si hay ciclos en el árbol de ejecución, la cantidad de caminos sería infinita.
- Los dos puntos anteriores son un problema porque, a priori, no se sabe qué caminos son posible y cuáles no, por lo que se hará exploración innecesaria (que podría no terminar si hay ciclos).

- Ciertas operaciones son complicadas de resolver para el constraint solver (e.g. aritmética no lineal, hashing, problemas NP-completos). Las expresiones pueden ser arbitrariamente complejas, y el constraint solver puede tardar mucho en resolver la fórmula (o incluso no ser capaz de hacerlo).
- No puede tener en cuenta dependencias de información en tiempo de ejecución (e.g. archivos, sockets). Por lo tanto, no se puede obtener condiciones de camino que dependan del tiempo de ejecución: toda la información debe ser estática.

El problema de la explosión combinatoria (o infinita) de caminos en el CFG, donde muchos de esos caminos pueden ser irrealizables, se debería poder solucionar *guiando* la selección de caminos para la ejecución simbólica. Eso será la idea en la **ejecución simbólica dinámica**.

4.6. Ejecución simbólica dinámica

La ejecución simbólica dinámica aprovecha la ejecución **concreta** del programa para mejorar el mecanismo de generación de la ejecución simbólica.

El procedimiento consistirá en empezar ejecutando el programa con una entrada cualquiera, y analizando qué condición de camino induce esa entrada. Luego, hacer backtracking en el árbol de ejecución hasta llegar a la decisión más abajo que tenga alguna rama sin cubrir.

Tomar entonces la negación de la decisión que se tomó en esa rama, y entregársela como entrada al constraint solver. Si resulta SAT, éste dará posibles valores de entrada que la satisfagan, lo cual representará una nueva condición de camino (y un camino más cubierto en el árbol). En caso de no ser SAT, ese camino se asume irrealizable (no podrá ser cubierto por la ejecución simbólica). Este proceso sigue hasta que se explore todo el CFG.

```
t = create_test(M)

while budget is not empty:
 path_condition = exec_concolic(t)
 for i from path_condition.size()-1 downto 0
   branch = path_condition[i]
   if neg(branch) is not covered:
     query = path_condition[0..i-1] && neg(branch)
     solution = solve(query)
     if solution is SAT:
       t = create_test(M, solution)
       Add t to testSuite
       break
 return testSuite

return testSuite
```

Figura 12: Pseudocódigo para la generación de casos de test usando ejecución simbólica dinámica.

La clave de la ejecución simbólica dinámica es que la exploración (la búsqueda de nuevas condiciones de camino a resolver) es **guiada por ejecuciones concretas**. De aquí viene el concepto de *ejecución concólica* (**concolic execution**: **conc**rete + symb**olic**).

Notar que siempre que se le entrega algo al constraint solver, hay más chances de que sea satisfactible, en comparación con la ejecución simbólica clásica, ya que proviene de una ejecución concreta. En cambio, en la variante clásica (estática), si ya se empezó con una condición insatisfactible, entonces al ir variando y explorando sobre ese camino, se estará perdiendo el tiempo (todo lo que siga será insatisfactible).

A su vez, algunas limitaciones de la ejecución simbólica estática pueden ser superadas reemplazando valores simbólicos por aproximaciones con valores concretos; esto puede ser particularmente útil cuando el lenguaje del constraint solver no permite representar alguna expresión especialmente compleja.

Observación. Al aproximar un valor simbólico usando un valor concreto, existe el riesgo de introducir **imprecisión**:

- La solución retornada no recorre el camino esperado.
- Es necesario tener en cuenta esta situación y adaptar el algoritmo de ejecución simbólica en caso de introducir imprecisiones. La idea sería ir comparando la solución obtenida con la solución esperada, y en caso de no haber coincidencia, se dice que hay **divergencia**.

Cambios en la generación si hay divergencia:

- Mantener una lista de las condiciones de camino que se generan con ejecución simbólica.
- Cada vez que se crea una nueva condición de camino para resolver, verificar que sea un prefijo de la condición de camino obtenida de ejecutar el nuevo test case (es decir, que el camino se vaya extendiendo, y no que *diverja* hacia otro lugar).
- Si no ocurre esto, entonces comparar si ya se había obtenido esa condición de camino. Si ya se había explorado, se decide si vale la pena continuar con el algoritmo de generación.

4.6.1. Resumen

- Ejecuta concretamente el test, pero guarda la condición de camino.
- Utiliza un constraint solver para crear nuevas entradas.
- Tiene limitaciones inherentes al constraint solver (aritmética no lineal, hashing, etc.).
- Cuanto más complejos son los programas a testear, más grandes serán las condiciones de camino, y más costo computacional tendrá resolverlas (complejidad exponencial).

5. Search-based testing

El testing basado en búsqueda (search-based testing) es parte de la disciplina conocida como Search-based Software Engineering, que consiste en transformar los problemas de la ingeniería del software en problemas de **optimización**. Como los espacios de búsqueda en este tipo de problemas suelen ser muy grandes, es usual utilizar algoritmos de búsqueda meta-heurísticos para resolver los problemas.

Recordar: Las heurísticas

- pueden no siempre encontrar la mejor solución.
- encuentran una solución *buena* en una cantidad *razonable* de tiempo.
- sacrifican completitud en pos de *eficiencia*.
- son útiles para resolver problemas *difíciles*.

Este tipo de técnicas pueden ser útiles en escenarios en los que random testing tiene problemas. Por ejemplo, cuando satisfacer una condición en el código es muy poco probable usando entradas aleatorias no guiadas.

La idea entonces será:

1. Definir una noción de **cercanía** del input con respecto al **goal** (meta, objetivo) que se quiere alcanzar.
2. Por cada nueva solución, explorar el **vecindario** de soluciones.
3. Elegir la solución del vecindario que sea **mejor** que la actual.
4. Repetir hasta no poder encontrar mejoras.

Normalmente la noción de *bueno* en relación a un valor de solución se asocia a una dimensión numérica dada por la **función de fitness**. Esta función cuantifica qué tan bueno es un elemento del espacio de soluciones, con respecto al objetivo. En el caso de testing, el objetivo podría ser cubrir una cierta línea de código, por ejemplo.

5.1. Hill climbing

El hill climbing es exactamente la idea descrita antes:

1. Elegir un valor aleatorio.
2. Explorar el vecindario (cómo definirlo es esencial para la implementación).
3. Elegir el mejor vecino (cómo elegirlo también es importante).
4. Repetir hasta el máximo valor (llegar a la *cima de la colina*).

El principal problema de este acercamiento es que es posible que el proceso se *estanque* en un máximo local, y no sea capaz de descubrir un máximo global para lo que sea que esté buscando optimizar.

Una meta-heurística que se puede usar para enfrentar este problema es **taboo search**. El objetivo de esta técnica es evitar que la búsqueda entre en ciclos de no-mejora, prohibiendo o penalizando ciertos movimientos que llevan al algoritmo a un espacio de soluciones ya explorado.

Para eso, se almacenan los últimos k movimientos (*lista tabú*). Al elegir un nuevo input, no se puede tomar ninguna solución que esté en esa lista. De esta manera, se evitan ciclos de longitud k.

Otro problema más importante con hill climbing viene cuando los vecindarios son demasiado grandes, y la heurística de búsqueda se torna demasiado lenta. Otras alternativas más sofisticadas surgen entonces.

5.2. Algoritmos genéticos

Los algoritmos genéticos y evolutivos están inspirados en la evolución biológica: genes como información que se pasa de una a otra generación, selección natural, supervivencia del más apto, diversidad genética, etc.

Esquemáticamente, un algoritmo genético consiste en:

1. Inicializar una población inicial.
2. Evaluar la población actual.
3. Seleccionar padres.
4. Crear hijos.
5. Si terminó, retornar el mejor individuo. Si no, volver a 2.

Sin embargo, cada uno de estos pasos plantea varias decisiones a tomar para dar con una implementación concreta de uno de estos algoritmos:

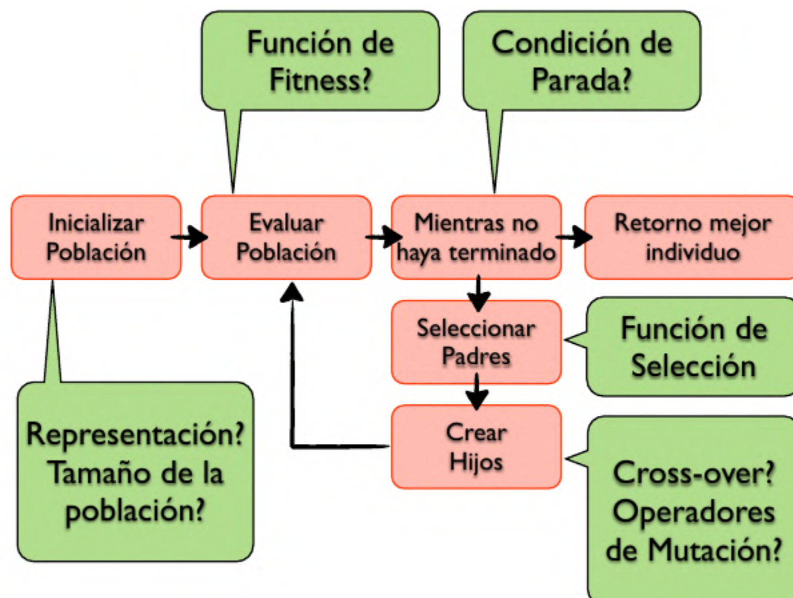


Figura 13: Los componentes que hacen un algoritmo genético.

5.2.1. Población inicial

Hay diferentes maneras de generar la población inicial:

- Aleatoriamente.
- Usando soluciones existentes.
- Usando un test suite con cobertura de un criterio de menor potencia.
- Manualmente.

5.2.2. Función de fitness

Como se mencionó antes, la manera clásica de evaluar un individuo es la **función de fitness**, que mide *cuán bueno* es un individuo como solución.

Observación. La función de fitness es específica para el problema.

Se dice que la función de fitness *determina el paisaje de búsqueda* (fitness landscape). Algunos paisajes de búsqueda facilitan la tarea de búsqueda: en general los que son más *suaves* (smooth) permiten guiar la búsqueda de una mejor solución. En cambio, los que tienen más *quiebres* (rugged) suelen dificultar la exploración.

5.2.3. Condición de parada

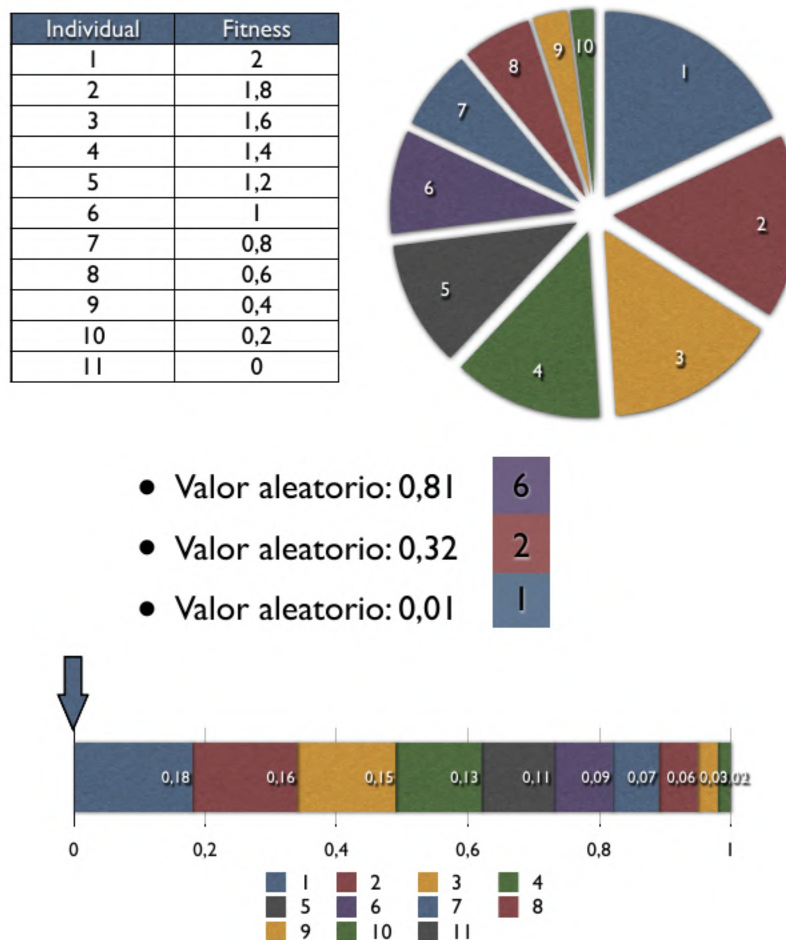
Algunas posibles opciones son:

- Tiempo de ejecución.
- Cantidad de iteraciones.
- Cantidad de evaluaciones de fitness de individuos.
- Una cota para la función de fitness.

5.2.4. Roulette wheel selection

El método de la ruleta es el más básico para seleccionar los *padres* (los individuos que se van a reproducir) en una población determinada. Consiste en tomar la probabilidad de elegir un individuo **proporcionalmente a su valor de fitness**.

Esto es, asignar rangos con amplitud (probabilidad) determinada por el fitness. El problema de este método es que cuanta más disparidad haya entre los valores de fitness de la población, más *descompensadas* quedarán las probabilidades; es decir, será cada vez menos probable elegir a los individuos con menor fitness, hasta llegar posiblemente al extremo en el que el individuo con mayor fitness absorbe a todos los demás. Esto no es deseable pues se **pierde diversidad genética**, lo cual puede resultar en un estancamiento en la calidad de los individuos.



Individual	Fitness
1	2
2	1,8
3	1,6
4	1,4
5	1,2
6	1
7	0,8
8	0,6
9	0,4
10	0,2
11	0

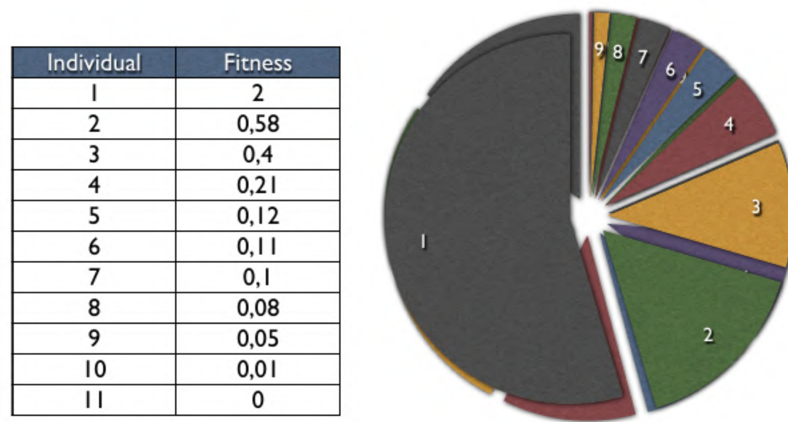
Figura 14: Ejemplo de una selección con el método de la ruleta.

5.2.5. Rank selection

Este método de selección consiste en **rankear individuos de acuerdo a su fitness**, usando ese valor para definir rangos de selección, no necesariamente con un orden que defina el individuo elegido.

No hay diferencia si el individuo con mejor fitness es diez veces mejor que el segundo, o solo un poquito mejor. Por lo tanto, la probabilidad de elegir a un individuo con fitness nula es mayor que cero, lo cual no ocurre en el método de la ruleta. Esto es bueno dado que siempre se deja abierta la chance a que sea elegido cualquier individuo (aunque sea poco probable), lo cual preserva la diversidad genética.

Es por esto que rank selection suele ser preferible en la práctica.



Individual	Fitness
1	2
2	0,58
3	0,4
4	0,21
5	0,12
6	0,11
7	0,1
8	0,08
9	0,05
10	0,01
11	0

Figura 15: Ejemplo de una selección con rank selection.

5.2.6. Tournament selection

Este método de selección se basa en construir un *torneo* con n individuos elegidos aleatoriamente entre la población, y elegir el mejor de los n individuos como el seleccionado. Hay distintas maneras de hacer un torneo (enfrentamientos directos, todos contra todos, por rondas, etc.), y el tamaño del torneo define la *presión selectiva*: a más chico el torneo, menor ingerencia tiene el fitness en la decisión (en el caso extremo en que $n = 1$, la función de fitness no se usa para decidir). En cambio, si n es el tamaño de la población, siempre va a ganar el mejor individuo (siempre y cuando la elección sea sin repetidos, lo cual puede pasar o no). En cualquier otro caso, siempre hay alguna probabilidad de elegir al individuo de menor fitness.

5.2.7. Crossover

El método de reproducción de individuos está ligado a la forma de representación de los mismos. Una forma clásica de creación de *hijos* es el **crossover**, que consiste en cruzar individuos de manera tal que cada padre transmita una parte de su información genética a cada hijo.

Sería deseable que el crossover siempre resulte en un individuo válido, dado que la idea es generar una población con el mismo tamaño que la anterior.

Existen distintas formas de hacer un crossover:

- Single-point crossover: elegir un único punto en los dos padres y dividir/unir en ese punto.
- Two-point crossover: elegir dos puntos en los padres e intercambiar la parte interior.
- Fixed/variable length crossover: igual punto de crossover en ambos padres, descendencia con tamaño constante (vs. no constante).
- Uniform crossover: los genes son aleatoriamente seleccionados de cada padre.

5.2.8. Mutación

La mutación es un **cambio en los genes** de un individuo. El objetivo de mutar es introducir variaciones en la genética de la población para evitar estancarse en máximos locales en el espacio

de búsqueda.

Normalmente, hay una probabilidad de que cada individuo mute, y el tipo de mutación también se puede elegir aleatoriamente (e.g. en strings, agregar, quitar o cambiar un carácter). Además, suelen ser cambios pequeños, ligados a la estructura de representación de los individuos.

5.3. Branch distance

La noción de branch distance aplica específicamente al escenario de testing con algoritmos genéticos, ya que se define como **la distancia del predicado a ejercitar el branch** (tanto por verdadero como por falso).

El cálculo de la branch distance se hace de la siguiente manera (k es alguna constante positiva):

Condición	Branch distance
$a = b$	$\lvert a-b\rvert = 0 \, ? \, 0 \ : \ \lvert a-b\rvert$
$a \neq b$	$a \neq b \, ? \, 0 \ : \ k$
$a < b$	$a < b \, ? \, 0 \ : \ (a-b)+k$
$a \leq b$	$a \leq b \, ? \, 0 \ : \ (a-b)$
$a > b$	$a > b \, ? \, 0 \ : \ (b-a)+k$
$a \geq b$	$a \geq b \, ? \, 0 \ : \ (b-a)$
$o.m(v_1, ..., v_n)$	$true \, ? \, 0 \ : \ k$
$A \wedge B$	$distance(A) + distance(B)$
$A \vee B$	$\text{mín}(distance(A), distance(B))$
$\neg A$	Aplicar De Morgan

5.4. Limitaciones de la branch distance

Si en un programa hay decisiones que dependen de otras decisiones anteriores, la branch distance no es una métrica suficiente para medir cuán cerca estuvo de ejecutarse una decisión que depende de otra. Esto es porque la branch distance mide, para una branch que fue ejecutada, cuán cerca se estuvo de hacerla verdadera/falsa, pero no considera decisiones que no hayan llegado a ser cubiertas.

5.4.1. Control-dependencia

A partir de esto, se introduce el concepto de **nodos dominadores** en un CFG. Se dice que el nodo A domina al nodo B si todo camino hacia B debe pasar por A (A es una *parada necesaria* para llegar a B desde la raíz).

A su vez, se dice que un dominador es inmediato si es el dominador más cercano en todo el camino desde la raíz del árbol. Por lo tanto, la raíz no tiene dominador inmediato.

Notar que esta propiedad se puede calcular estáticamente a partir de un análisis del código.

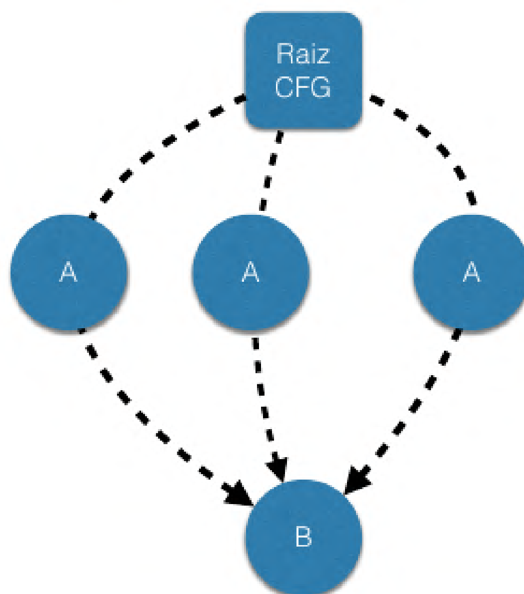


Figura 16: Esquema de un CFG en el que A domina a B. Las líneas punteadas representan caminos en el CFG.

Adicionalmente, se define el concepto de **post-dominador**: el nodo B post-domina al nodo A si todos los caminos desde A hasta la salida del programa deben pasar por B (como el dominador pero viendo el árbol al revés).

Análogamente, el post-dominador inmediato es el dominador más cercano en todo el camino hacia el nodo de salida.

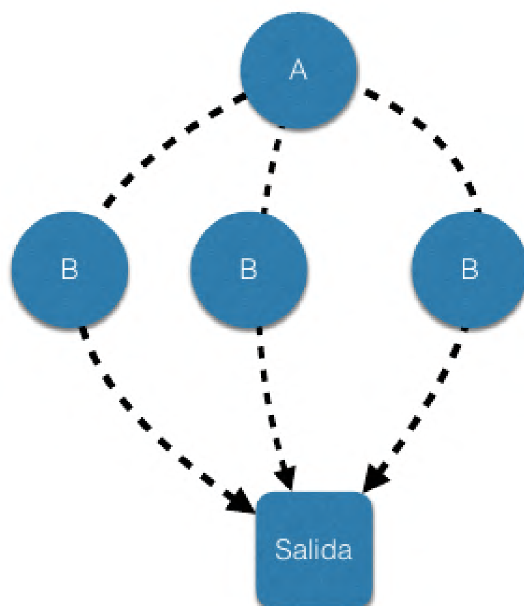


Figura 17: Esquema de un CFG en el que B post-domina a A. Las líneas punteadas representan caminos en el CFG.

Considerar el siguiente programa que calcula el máximo común divisor entre dos números:

```
def gcd(x, y):
    tmp = 0
    while y != 0:
```

```
        tmp = x % y
        x = y
        y = tmp
    return x
```

En la siguiente figura se puede ver las relaciones de dominadores y post-dominadores para el ejemplo:

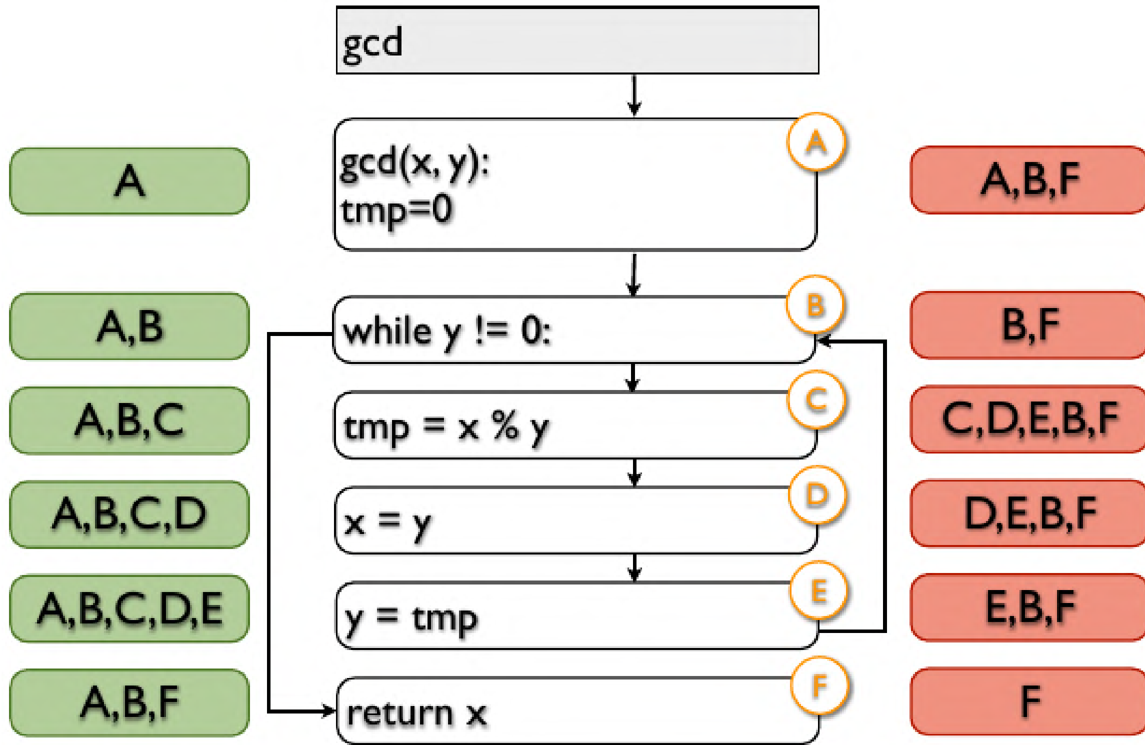


Figura 18: CFG para el programa del máximo común divisor, y relaciones de dominación y post-dominación. En verde, los nodos que dominan a cada nodo. En rojo, los nodos que post-dominan a cada nodo.

A partir de estos conceptos de dominación y post-dominación, se define la **control-dependencia** de la siguiente manera: B es control-dependiente de A si:

- A domina a B.
- B no post-domina a A.
- A tiene al menos dos sucesores.
- B post-domina a un sucesor de A.

Observación. Intuitivamente, B es control-dependiente de A si necesariamente hay que cubrir A para poder ejecutar B, pero no al revés.

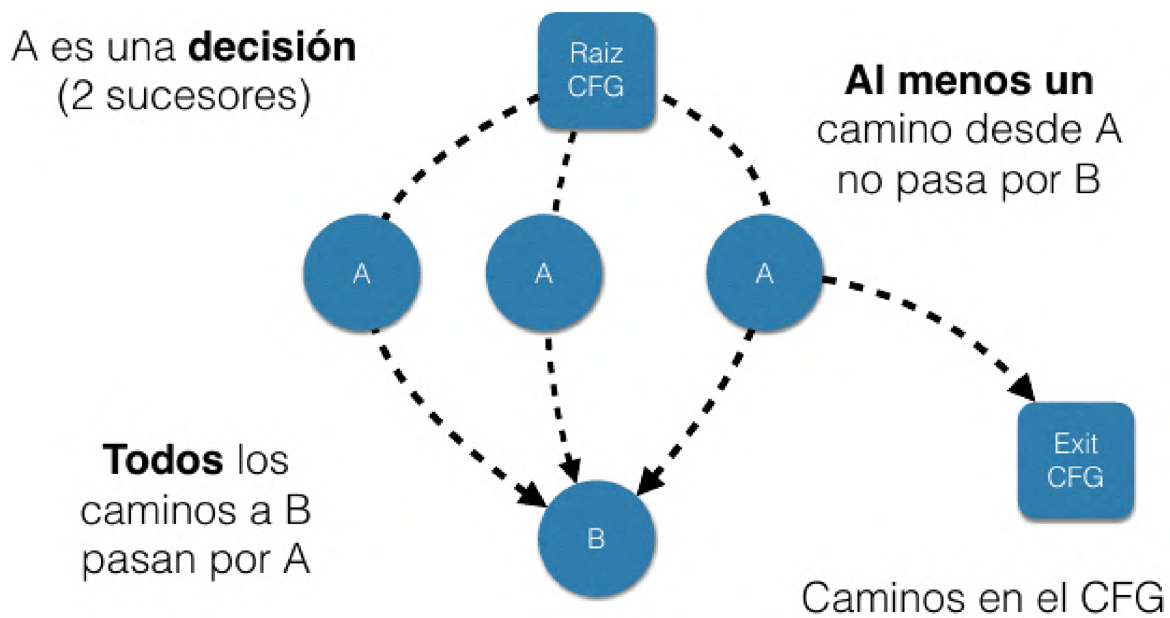


Figura 19: En este CFG, B es control-dependiente de A.

5.4.2. Approach level

Con la noción de control-dependencia se puede construir un **grafo de control-dependencias**, y con eso se puede calcular a cuántos "saltos" se estuvo de ejecutar cierto nodo del CFG, usando la altura del árbol.

Esto se puede usar como función de fitness:

- Dependientes = cantidad de nodos control-dependientes para el nodo objetivo (i.e. ancestros del objetivo a cubrir)
- Ejecutados = cantidad de nodos control-dependientes (ancestros) ya ejecutados
- Función de fitness = (Dependientes - Ejecutados)

Esto es lo que se conoce como **approach level** (*nivel de aproximación*), una métrica que cuenta la cantidad de saltos de control-dependencia que hay entre el goal (objetivo) y el camino de ejecución tomado.

5.4.3. Combinando branch distance y approach level

El problema de usar sólo el approach level es que no tiene en cuenta lo que mide la branch distance (y viceversa). Entonces, se puede utilizar una combinación de las dos heurísticas en una función de fitness para guiar al algoritmo de búsqueda:

- El approach level indica cuán cerca se está del predicado.
- La branch distance indica cuán cerca se está de ejercitar el branch (positivo o negativo).

Esta heurística combinada se suele llamar Control+Branch Distance, y se calcula como la suma de las dos heurísticas combinadas. Sea t un camino acícliclo desde la raíz del CFG hasta el branch G:

1. Computar el approach level hasta G.
2. Sumar la branch distance del predicado más cercano alcanzado.

El objetivo entonces será minimizar la suma de approach level y branch distance. Esto puede ser un problema cuando la branch distance es un número muy grande (lo cual depende de los números involucrados en las condiciones). Si esto ocurre, se descompensa la función de fitness, que es *dominada* por la branch distance.

Para solucionar eso, se suele tomar una versión **normalizada** de la branch distance (e.g. $n(x) = \frac{x}{x+1}$ es una posible función de normalización, que deja el valor de la branch distance entre 0 y 1). Así, en vez de sumar el valor de branch distance directamente, se suma su valor normalizado, con lo cual de todos los caminos del CFG hasta el branch G (*goal*), se toma la mínima suma de approach level y branch distance normalizada.

Así, si por ejemplo la función de fitness da $5,75$, se puede interpretar que quedan 5 nodos sin alcanzar para llegar a G, y que hay una distancia de $0,75$ para cubrir el branch divergente.

5.5. Testability transformation

Esto consiste en transformar problemas para que sean más fáciles de testear usando generación basada en búsqueda. El programa transformado se usa para generar datos de test, y luego es descartado, por lo que no es necesario que la transformación respete el comportamiento o la signatura de todo el programa original.

5.5.1. Flag problem

El *problema de la bandera* (flag problem) ocurre cuando hay un *flag* (un valor booleano) que representa una condición sobre una variable con más valores posibles. Por ejemplo, escribir

```
bool flag = (x == 10);
if (flag) {
    ...
}
```

en vez de

```
if (x == 10) {
    ...
}
```

Esto es un problema para calcular la branch distance, puesto que al comparar sobre un valor booleano, no se puede tener la información de cuán cerca está la variable de alcanzar el valor para verificar la condición. Es decir, la branch distance no puede capturar el significado de ese flag.

Hay distintos niveles de este problema, con distintas soluciones que crecen en complejidad a medida que crece el nivel:

- Flag level 0: no se usa el flag, es el caso estándar (nada que transformar).

- Flag level 1: es como lo que pasa en el ejemplo de recién; es decir, ni el flag ni la variable son modificados antes del predicado. Para solucionar esta situación, hay que reemplazar el flag por su expresión, preservando la semántica. Esta sería la *transformación para testeabilidad*.
- Flag level 2: cuando las variables involucradas en el valor del flag sí cambian antes de ser usado en la condición, se puede introducir variables temporales que almacenan esos valores antes de ser modificados. Luego, la situación resultante se puede resolver como un flag level 1.
- Flag level 3: esto implica que no sólo se pueden modificar las variables del flag, sino también el propio flag antes de ser usado como condición. En este caso, hay que calcular cuál sería la expresión equivalente al flag al momento de llegar a la condición, hacer el reemplazo. Esto ya es más complicado de resolver.
- Flag level 4: la secuencia de flags contiene condicionales.
- Flag level 5: definición de flags en diferentes ciclos a donde se usa el flag.

5.5.2. Predicados anidados

Cuando hay dos condiciones que están anidadas, se puede mejorar la *testeabilidad* calculando y sumando ambas branch distances, y usando ese valor calculado como expresión para la condición. Por ejemplo, la siguiente función

```
void original(double a, double b) {
  if(a == b) {
    double c = b + 1;
    if(c == 0) {
      // target
    }
  }
}
```

se puede transformar de la siguiente manera:

```
void transformed(double a, double b) {
  double _dist = 0;
  _dist += branch_distance(a == b);
  double c = b + 1;
  _dist += branch_distance(c == 0);
  if(_dist == 0.0) {
      // target
  }
}
```

5.5.3. Otras transformaciones

También se puede mejorar la *testeabilidad* en los siguientes escenarios:

- Cambiar los chequeos de contenedores vacíos de `.isEmpty()` a `.size() == 0` para poder calcular un valor de branch distance más representativo.

- Cambiar las comparaciones entre objetos por las comparaciones de sus valores primitivos, cuando sea posible.
- Al chequear si un contenedor contiene cierto elemento, se puede cambiar la condición por el cálculo y comparación con 0 de la mínima distancia entre los elementos del contenedor y el elemento a consultar.
- Cambiar comparaciones por igualdad de strings por comparaciones de la distancia entre ellos con 0. Existen diferentes medidas para la distancia entre strings; por ejemplo, la distancia de Hamming o la distancia de Levenshtein.

5.6. Búsqueda con único objetivo vs. múltiples objetivos

La manera tradicional de encarar un problema de search-based test generation es optimizar cada caso de test para cada objetivo (goal) de manera aislada: la **estrategia de objetivo único**. El problema que puede tener esto es cómo decidir la manera en la que se distribuirá el presupuesto de testing entre los distintos casos de test. Sobre todo puede ser problemático cuando alguno de los objetivos es insatisfactible, o es mucho más complejo que otro.

Tomando un escenario de cobertura de branches, en una estrategia con único goal, se fija una branch como objetivo en cada momento, y los cromosomas de un individuo son los casos de test. Por lo tanto, para optimizar todo un test suite, hay que ejecutar el algoritmo genético múltiples veces.

Esto tiene algunos problemas:

- Puede haber branches que no sean realizables, resultando en un desperdicio de presupuesto.
- Algunas branches pueden requerir mayor presupuesto que otras.
- Distribuir adecuadamente el presupuesto entre los goals no es trivial.

5.6.1. Whole-test suite generation

Un enfoque alternativo es el **whole-test suite generation**, es decir una estrategia con múltiples objetivos; se busca optimizar el total del test suite. Esto hace que la distribución del presupuesto entre los goals no sea importante, y la insatisfactibilidad de goals individuales no afecta la búsqueda.

En este caso, para la cobertura de branches, el objetivo son todas las branches a la vez, y los cromosomas son todo el test suite. Por lo tanto, el algoritmo genético se ejecuta una sola vez. Además, mientras que en el enfoque de único objetivo había que sumar la branch distance y el approach level para calcular la función de fitness, en el caso del enfoque whole-suite, la noción de approach level no tiene sentido (porque la idea del approach level es cuantificar cuán cerca se está de llegar a la branch de interés, pero en este caso todas las branches son de interés), sino que solo se suma la branch distance para cada branch. Sumar el approach level sólo agrandaría los números de fitness, no tendría un efecto real.

5.6.2. *Dominancia*

El problema de este enfoque es que tiene muchos goals (branches a cubrir) pero un solo objetivo general (a nivel procedimiento, se busca minimizar la sumatoria de branch distances), por lo que hay una única dimensión a optimizar. Lo que genera esto es que se "pegotean" todas las goals (que se podrían pensar como un vector de dimensiones a optimizar) en un único escalar: la función de fitness. Una manera alternativa de plantearlo es tener n funciones de fitness, separando cada branch; esto haría que reaparezca el approach level para calcular las funciones de fitness.

Sin embargo, tener todo resumido en un único valor de fitness también tiene sus ventajas: hay un orden fácilmente definido en cuanto a cuán bueno es un test suite, y esto no es tan sencillo si hay n funciones de fitness que conviven: ¿cómo determinar qué individuo tiene mejor fitness si hay varias funciones que considerar?

Se dice que un individuo (un test case) x domina a un individuo y si es *mejor* en todos los componentes de la función de fitness. Es decir, si los valores del vector de funciones objetivo satisfacen que:

$$\forall i \in \{1, ..., k\}, f_i(x) \leq f_i(y)$$

y además

$$\exists\, j \in \{1, ..., k\} : f_j(x) < f_j(y)$$

Para solucionar los casos de *empate* (e.g. en un vector de fitness de dos dimensiones, un individuo tiene valor $(1, 5)$ y el otro $(5, 1)$) se introducen las dominancias de Pareto: los frentes de Pareto forman clases, dentro de las cuales todos los individuos son *igual de buenos*, pero cada clase es distinta de las demás en cuanto a *bondad*. Esto provee un orden para los individuos, incluso cuando la función de fitness se representa como un vector.

5.6.3. Criterios de cobertura y EvoSuite

La idea de aplicar un algoritmo genético para optimizar test suites causará que, por ejemplo, cambie el proceso de crossover (que será combinar dos test suites) y de mutación (que podría consistir en agregar, quitar o modificar líneas de código de los test cases que conforman un test suite; también se puede mutar agregando o quitando un test case completo). En caso de mutar un test case, hay que tener en cuenta que se puede romper algo: hay que chequear que el test siga siendo válido, y recomponerlo de no ser así.

En cada generación de test suites, el fitness del mejor individuo de la población será igual o mejor que el de la generación anterior.

Este tipo de enfoque es el que aplica la herramienta EvoSuite, que optimiza test suites con diferentes opciones de funciones de fitness para elegir (observar que las **funciones de fitness** son los **criterios de cobertura**). El algoritmo evolutivo genérico que usa EvoSuite es el siguiente:

Algorithm 1 The genetic algorithm applied in EvoSuite

1 $current_population \leftarrow$ generate random population
2 **repeat**
3 $Z \leftarrow$ elite of $current_population$
4 **while** $|Z| \neq |current_population|$ **do**
5   $P_1, P_2 \leftarrow$ select two parents with rank selection
6   **if** crossover probability **then**
7    $O_1, O_2 \leftarrow$ crossover P_1, P_2
8   **else**
9    $O_1, O_2 \leftarrow P_1, P_2$
10   mutate O_1 and O_2
11   $f_P = min(fitness(P_1), fitness(P_2))$
12   $f_O = min(fitness(O_1), fitness(O_2))$
13   $l_P = length(P_1) + length(P_2)$
14   $l_O = length(O_1) + length(O_2)$
15   $T_B =$ best individual of $current_population$
16   **if** $f_O < f_P \lor (f_O = f_P \land l_O \leq l_P)$ **then**
17   **for** O in $\{O_1, O_2\}$ **do**
18   **if** $length(O) \leq 2 \times length(T_B)$ **then**
19      $Z \leftarrow Z \cup \{O\}$
20   **else**
21      $Z \leftarrow Z \cup \{P_1$ or $P_2\}$
22   **else**
23    $Z \leftarrow Z \cup \{P_1, P_2\}$
24 $current_population \leftarrow Z$
25 **until** solution found or maximum resources spent

Figura 20: Algoritmo genético usado por EvoSuite.

Los posibles criterios de cobertura que se pueden usar son:

- Method coverage: se deben ejecutar todos los métodos de la clase bajo test.
- Top-level method coverage: como method coverage, pero cada método debe ser invocado directamente desde el test case (no llamado por otro método).
- Top-level method coverage no exception: igual que el anterior, pero sólo se consideran ejecuciones que terminan sin generar excepciones.
- Line coverage: se deben ejecutar todas las líneas del código.
- Branch coverage: todos los predicados de las decisiones evalúan a verdadero y a falso al menos una vez.
- Direct branch coverage: cada decisión en un método accesible es cubierta por una invocación directa desde un test de unidad.
- Weak mutation: se produce una infección del estado por cada mutante de la clase bajo test.
- Output coverage: los métodos deben cubrir particiones de valores de retorno (e.g. números positivos, negativos, dígitos, caracteres alfanuméricos...).
- Number of exceptions: cada método maximiza la cantidad de excepciones que emite.

Figura 21: Criterios de cobertura, que dan lugar a funciones de fitness.

5.6.4. Combinación de criterios de cobertura

A su vez, estos criterios se pueden combinar, resultando en una función de fitness que suma el resultado de la función de fitness de cada criterio combinado (además, se le puede asignar un peso diferente a cada uno). Es decir, si S es el test suite a optimizar combinando n criterios, y w_i es el peso asignado al i-ésimo criterio, y f_i es la función de fitness para el i-ésimo criterio, entonces el valor de la función de fitness será:

$$f(S) = \sum_{i=1}^{n} w_i \times f_i$$

Observación. No todos los criterios se pueden combinar con todos. Deben ser criterios **no contradictorios**; es decir, que un criterio no mejore cuando otro empeora. Por ejemplo, minimizar el tamaño del test suite, o minimizar el tiempo de ejecución de tests se contradice con el cubrimiento de líneas.

5.6.5. Post-procesamiento del test suite

Una vez que el algoritmo genético finaliza, EvoSuite realiza el siguiente post-procesamiento del test suite:

1. Minimización: remover los statements/tests que no contribuyen al cubrimiento de goals.
2. Generación de aserciones: agregar aserciones tales que al menos un mutante es matado por la aserción.
3. JUnit: comprobar que el JUnit resultante no tiene fallas.

5.7. Variantes de algoritmos genéticos

5.7.1. Algoritmo genético estándar

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

```
1: P ← GenerateRandomPopulation(p_s)
2: PerformFitnessEvaluation(δ, P)
3: while ¬C do
4:     N_P ← { } ∪ Elitism(P)
5:     while |N_P| < p_s do
6:         p_1, p_2 ← Selection(s_f, P)
7:         o_1, o_2 ← Crossover(c_f, c_p, p_1, p_2)
8:         Mutation(m_f, m_p, o_1)
9:         Mutation(m_f, m_p, o_2)
10:        N_P ← N_P ∪ {o_1, o_2}
11:    end while
12:    P ← N_P
13:    PerformFitnessEvaluation(δ, P)
14: end while
15: return P
```

Figura 22: Algoritmo genético estándar.

Este es el algoritmo genético genérico que se vio al principio. Siempre agrega el offspring (los hijos) a cada generación.

5.7.2. Algoritmo genético monotónico

El algoritmo genético monotónico (monotonic) hace un chequeo para quedarse con el mejor individuo entre los hijos y los padres de cada generación. Esta es la diferencia con el algoritmo estándar, que siempre agrega la nueva generación. Con este chequeo adicional, el algoritmo monotónico no permite que empeore el fitness promedio de la población.

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

1: $P \leftarrow$ GenerateRandomPopulation(p_s)
2: PerformFitnessEvaluation(δ, P)
3: **while** $\neg C$ **do**
4: $N_P \leftarrow \{ \} \cup$ Elitism(P)
5: **while** $|N_P| < p_s$ **do**
6: $p_1, p_2 \leftarrow$ Selection(s_f, P)
7: $o_1, o_2 \leftarrow$ Crossover(c_f, c_p, p_1, p_2)
8: Mutation(m_f, m_p, o_1)
9: Mutation(m_f, m_p, o_2)
10: PerformFitnessEvaluation(δ, o_1)
11: PerformFitnessEvaluation(δ, o_2)
12: **if** Best(o_1, o_2) is better than Best(p_1, p_2) **then**
13: $N_P \leftarrow N_P \cup \{o_1, o_2\}$
14: **else**
15: $N_P \leftarrow N_P \cup \{p_1, p_2\}$
16: **end if**
17: **end while**
18: $P \leftarrow N_P$
19: **end while**
20: **return** P

Figura 23: Algoritmo genético monotónico.

5.7.3. Algoritmo genético steady-state

El algoritmo genético steady-state (*estado estable*) reemplaza a los padres por los hijos cuando hay mejoras en el fitness. Notar que hay un solo ciclo en el algoritmo, mientras que en los anteriores había dos anidados. Esto tiene como consecuencia que no exista el concepto de *generación* en el algoritmo steady-state; en vez de generar una nueva población en cada paso, saca dos individuos y agrega dos si hay mejoras; el resto se mantiene igual.

La ventaja de no *pisar* la población anterior como sí hacen los algoritmos anteriores, es que las *generaciones* se van construyendo de manera más *directa*, y eso hace que cualquier mejora en el fitness se utilice inmediatamente después (no hay que esperar a la generación siguiente): los mejores individuos se usan justo después de ser creados.

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

```
1: P ← GenerateRandomPopulation(p_s)
2: PerformFitnessEvaluation(δ, P)
3: while ¬C do
4:     p_1, p_2 ← Selection(s_f, P)
5:     o_1, o_2 ← Crossover(c_f, c_p, p_1, p_2)
6:     Mutation(m_f, m_p, o_1)
7:     Mutation(m_f, m_p, o_2)
8:     PerformFitnessEvaluation(δ, o_1)
9:     PerformFitnessEvaluation(δ, o_2)
10:    if Best(o_1, o_2) is better than Best(p_1, p_2) then
11:        P ← P \ {p_1, p_2} ∪ {o_1, o_2}
12:    else
13:        P ← P \ {o_1, o_2} ∪ {p_1, p_2}
14:    end if
15: end while
16: return P
```

Figura 24: Algoritmo genético steady-state.

5.7.4. Algoritmo genético breeder

El algoritmo breeder (*criador*) imita el proceso de selección genética de un criadero. Selecciona un subconjunto de la población para reproducirse (truncate). Comparte con los dos primeros algoritmos que *pisa* la población completa en cada generación; pero es distinto en el sentido de que no usa una función de selección, sino que selecciona aleatoriamente a los individuos de la población truncada.

En reusmen, se reduce a: el truncamiento (quedarse con un cierto porcentaje de los mejores individuos), selección al azar entre ese porcentaje, y la elección al azar entre la descendencia.

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

1: $P \leftarrow$ GenerateRandomPopulation(p_s)
2: PerformFitnessEvaluation(δ, P)
3: **while** $\neg C$ **do**
4: $N_P \leftarrow \{\ \} \cup$ Elitism(P)
5: $P' \leftarrow$ Truncate(P)
6: **while** $|N_P| < p_s$ **do**
7: $p_1 \leftarrow$ SelectRandom(P')
8: $p_2 \leftarrow$ SelectRandom(P')
9: $o_1, o_2 \leftarrow$ Crossover(c_f, c_p, p_1, p_2)
10: Mutation(m_f, m_p, o_1)
11: Mutation(m_f, m_p, o_2)
12: $o \leftarrow$ SelectRandom(o_1, o_2)
13: $N_P \leftarrow N_P \cup \{o\}$
14: **end while**
15: $P \leftarrow N_P$
16: PerformFitnessEvaluation(δ, P)
17: **end while**
18: **return** P

Figura 25: Algoritmo genético breeder.

5.7.5. Algoritmo genético celular

La variante celular (cellular) del algoritmo genético introduce un *modelo de vecindario* sobre el cual hace la selección: se explora el vecindario para elegir los dos padres que se reproducen en cada generación. Para cada individuo de la población, se consigue este vecindario para la reproducción.

Luego del crossover, se queda con el mejor hijo, hace la mutación y agrega a la población al hijo o al individuo actual. Al final de todo esto, pisa la generación antigua con la nueva.

El resultado es que la fitness del mejor individuo nunca baja (no usa elitismo como las anteriores variantes, pero siempre se queda con el mejor individuo en cada iteración).

Notar que el vecindario es un subconjunto de la población, así que todo individuo del vecindario necesariamente está en la población.

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, Neighbourhood model n_m
Output: Population of optimised individuals P

1: $P \leftarrow$ GENERATERANDOMPOPULATION(p_s)
2: PERFORMFITNESSEVALUATION(δ, P)
3: **while** $\neg C$ **do**
4: $N_P \leftarrow \{ \}$
5: **for all** $p \in P$ **do**
6: $N_B \leftarrow$ GETNEIGHBOURHOOD(p, P, n_m)
7: $p_1, p_2 \leftarrow$ SELECTION(s_f, N_B)
8: $o_1, o_2 \leftarrow$ CROSSOVER(c_f, c_p, p_1, p_2)
9: PERFORMFITNESSEVALUATION(δ, o_1)
10: PERFORMFITNESSEVALUATION(δ, o_2)
11: $o \leftarrow$ BEST(o_1, o_2)
12: MUTATION(m_f, m_p, o)
13: PERFORMFITNESSEVALUATION(δ, o)
14: $N_P \leftarrow N_P \cup$ BEST(o, p)
15: **end for**
16: $P \leftarrow N_P$
17: **end while**
18: **return** P

Figura 26: Algoritmo genético celular.

5.7.6. Algoritmo genético 1 + (λ, λ)

Este algoritmo es distinto de todos los anteriores: comienza generando una población aleatoria de tamaño 1. Luego, se crean λ versiones mutadas del individuo, con alta probabilidad de mutación. El crossover se aplica al padre (el original) y al mejor de los mutantes, y el algoritmo se queda con el mejor descendiente.

El proceso de crossover se realiza $\lambda/2$ veces, y al final del ciclo, se tiene en M las λ mutaciones del individuo original, y en O hay λ cruzas entre el mejor individuo mutante p' y el original p. Toma luego las λ cruzas, y elige el mejor individuo, que puede no ser mejor que el original. Si $p' > p$, reemplaza p con p'.

Este algoritmo es costoso, pues realiza evaluaciones de fitness después de cada mutación/crossover (son 2λ evaluaciones de fitness por cada individuo). Cuanto mayor sea λ, más costoso será, pero a menor λ, mayor riesgo de no mejorar p' respecto de p.

Además, el algoritmo no devuelve una población optimizada como los otros, sino que retorna el mejor individuo.

Este algoritmo recuerda de cierta manera al hill climbing, ya que se mueve de a una posición, hacia una mejora. El vecindario serían las mutaciones y crossovers, pero no se evalúan exhaustivamente todos los mutantes, sino que son λ *vecinos* al azar.

Esta similitud con hill climbing hace que surja la inquietud de si es posible que este algoritmo se estanque en máximos locales; el grado en el que esto puede pasar depende de cuán "disruptivos" sean los cambios producidos por los crossovers (las mutaciones suelen producir cambios pequeños, los crossovers pueden tener más influencia).

Input: Stopping condition C, Fitness function δ, Offspring size λ, Crossover function c_f, Crossover probability c_p, Mutation function m_f, Mutation probability m_p
Output: Best individual p

```
1: p ← GenerateRandomIndividual()
2: PerformFitnessEvaluation(δ, p)
3: while ¬C do
4:     M ← { }
5:     for i ← 1, λ do
6:         o ← Mutation(m_f, m_p, p)
7:         PerformFitnessEvaluation(δ, o)
8:         M ← M ∪ {o}
9:     end for
10:    p' ← Best(M)
11:    O ← { }
12:    for i ← 1, λ/2 do
13:        o_1, o_2 ← Crossover(c_f, c_p, p, p')
14:        PerformFitnessEvaluation(δ, o_1)
15:        PerformFitnessEvaluation(δ, o_2)
16:        O ← O ∪ {o_1, o_2}
17:    end for
18:    p' ← Best(O)
19:    if p' is better than p then
20:        p ← p'
21:    end if
22: end while
23: return p
```

Figura 27: Algoritmo genético $1 + (\lambda, \lambda)$.

5.7.7. Algoritmo evolutivo $\mu + \lambda$

Este algoritmo crea λ nuevos individuos (tamaño de la descendencia O) por cada individuo de la población P. Luego, se queda con los μ mejores individuos (tamaño de la población P) sobre $P \cup O$ (esto sólo tiene sentido si $\mu \leq \lambda$).

La idea es repetir λ/μ veces una mutación p, para cada individuo $p \in P$. Con eso se genera la descendencia O, de tamaño λ, y se queda con los μ mejores.

Notar que, a diferencia de los algoritmos anteriores, en este no hay crossover (sólo hay mutaciones). Por eso no se suele llamar algoritmo *genético*, sino más generalmente *evolutivo*.

Input: Stopping condition C, Fitness function δ, Population size μ, Offspring size λ, Mutation function m_f, Mutation probability m_p
Output: Population of optimised individuals P

1: $P \leftarrow$ GENERATERANDOMPOPULATION(μ)
2: PERFORMFITNESSEVALUATION(δ, P)
3: **while** $\neg C$ **do**
4: $O \leftarrow \{\ \}$
5: **for all** $p \in P$ **do**
6: **for** $i \leftarrow 1, \frac{\lambda}{\mu}$ **do**
7: $o \leftarrow$ MUTATION(m_f, m_p, p)
8: $O \leftarrow O \cup \{o\}$
9: **end for**
10: **end for**
11: PERFORMFITNESSEVALUATION(δ, O)
12: $P \leftarrow$ select best μ individuals from $P \cup O$
13: **end while**
14: **return** P

Figura 28: Algoritmo evolutivo $\mu + \lambda$.

5.7.8. Algoritmo evolutivo (μ, λ)

Este algoritmo es igual al $\mu + \lambda$ (tampoco tiene crossover), pero en vez de seleccionar los μ mejores de la unión entre la población P y la descendencia O, selecciona los mejores sólo de la descendencia.

Como no evalúa el fitness de la generación pasada, y tampoco tiene una élite como otras variantes, no hay garantías de que el mejor individuo de una generación pase a la siguiente.

```
Input: Stopping condition C, Fitness function δ, Population size μ, Offspring
    size λ, Mutation function m_f, Mutation probability m_p
Output: Population of optimised individuals P
 1: P ← GenerateRandomPopulation(μ)
 2: PerformFitnessEvaluation(δ, P)
 3: while ¬C do
 4:     O ← { }
 5:     for all p ∈ P do
 6:         for i ← 1, λ/μ do
 7:             o ← Mutation(m_f, m_p, p)
 8:             O ← O ∪ {o}
 9:         end for
10:     end for
11:     PerformFitnessEvaluation(δ, O)
12:     P ← select best μ individuals from O
13: end while
14: return P
```

Figura 29: Algoritmo evolutivo (μ, λ).

5.8. Herramientas basadas en algoritmos genéticos

- El NSGA-II (Non-dominated Sorting Genetic Algorithm) es un algoritmo genético especializado en el problema de minimizar n dimensiones de fitness simultáneamente. Utiliza los conceptos de frentes de Pareto definidos en 5.6.2, y crowding distance para retornar un conjunto de tests ordenados.

Input:
$U = \{u_1, \ldots, u_m\}$ the set of coverage targets of a program.
Population size M
Result: A test suite T

1 **begin**
2 $t \longleftarrow 0$ // current generation
3 $P_t \longleftarrow$ RANDOM-POPULATION(M)
4 **while** *not (search_budget_consumed)* **do**
5 $Q_t \longleftarrow$ GENERATE-OFFSPRING(P_t)
6 $R_t \longleftarrow P_t \bigcup Q_t$
7 $\mathbb{F} \longleftarrow$ FAST-NONDOMINATED-SORT(R_t)
8 $P_{t+1} \longleftarrow \emptyset$
9 $d \longleftarrow 1$
10 **while** $\mid P_{t+1} \mid + \mid \mathbb{F}_d \mid \leqslant M$ **do**
11 CROWDING-DISTANCE-ASSIGNMENT($\mathbb{F}_d$)
12 $P_{t+1} \longleftarrow P_{t+1} \bigcup \mathbb{F}_d$
13 $d \longleftarrow d + 1$
14 Sort($\mathbb{F}_d$) //according to the crowding distance
15 $P_{t+1} \longleftarrow P_{t+1} \bigcup \mathbb{F}_d[1 : (M - \mid P_{t+1} \mid)]$
16 $t \longleftarrow t + 1$
17 $S \longleftarrow P_t$

Figura 30: Pseudocódigo de NSGA-II

- El algoritmo MIO (Many Independent Objectives) se utiliza en contextos donde hay muchas goals, y poco presupuesto (por ejemplo, en system level testing). Cada goal tiene asociada una población, y sólo se guardan poblaciones para targets alcanzados alguna vez durante la búsqueda. Retorna un archivo de tests optimizados.

  Observando la figura 31, por cada goal Z guarda una población de individuos que alcanzan ese goal (una especie de diccionario de goals a individuos). Cada población intenta optimizar un goal distinto, y los goals se van agregando a Z cuando haya al menos un test que los haya alcanzado. Esa es la primera fase del algoritmo.

  A partir de cierto momento empieza la segunda fase, al llegar a cierto valor indicado por parámetro. No se hace más generación de elementos al azar, y se empieza a hacer mutaciones sobre lo existente.

  Es decir, las dos fases son exploración y optimización local + cubrimiento de goals encontrados que siguen en Z.

```
Input: Stopping condition C, Fitness function δ, Population size N, Mutation
    function m_f, Mutation probability m_p, Probability of random sampling R,
    Start of focused search F
Output: Archive of optimised individuals A
 1: Z ← SetOfEmptyPopulations()
 2: A ← { }
 3: while ¬C do
 4:     if R > Random(0,1) then
 5:         p ← GenerateRandomIndividual()
 6:     else
 7:         p ← SampleIndividual(Z)
 8:         p ← Mutation(m_f, m_p, p)
 9:     end if
10:     for all t ∈ ReachedTargets(p) do
11:         if IsTargetCovered(t) then
12:             UpdateArchive(A, p)
13:             Z ← Z \ {Z_t}
14:         else
15:             Z_t ← Z_t ∪ {p}
16:             if |Z_t| > N then
17:                 RemoveWorstTest(Z_t, δ)
18:             end if
19:         end if
20:     end for
21:     UpdateParameters(F, R, N)
22: end while
23: return A
```

Figura 31: Pseudocódigo de MIO

- MOSA (Many-Objective Sorting Algorithm).
- DynaMOSA (Dynamic MOSA).
- EvoSuite: ya mencionado antes, ofrece diferentes algoritmos de los estudiados en esta clase para la optimización de test suites.
- MaJiCKe.
- EvoMaster: generación automática de casos de test para REST APIs. Es una herramienta de caja blanca, y utiliza el algoritmo evolutivo MIO.
- Sapienz: utiliza NSGA-II para whole-test suite generation. Lo interesante es que sus objetivos son:
    1. Maximizar cantidad de líneas cubiertas.
    2. Minimizar longitud de los test cases.
    3. Maximizar cantidad de unique crashes.

  , y son contradictorios.

```
Input: AUT A, crossover probability p, mutation probability q,
       max generation g_max, execution time t
Output: UI model M, Pareto front PF, test reports C
M ← K_0; PF ← ∅; C ← ∅;                              ▷ initialisation
generation g ← 0;
boot up devices D;                               ▷ prepare app exerciser
inject MotifCore into D;   ▷ for hybrid exploration (see §3.2)
static analysis on A;          ▷ for string seeding (see §3.3)
instrument and install A;
initialise population P;      ▷ hybrid of random and motif genes
evaluate P with MotifCore and update (M,PF,C);
while g < g_max and ¬ timeout(t) do
    g ← g+1;
    Q ← wholeTestSuiteVariation(P, p, q);   ▷ see Algorithm 2
    evaluate Q with MotifCore and update (M,PF,C);
    ℱ ← ∅;                                   ▷ non-dominated fronts
    ℱ ← sortNonDominated(P ∪ Q, |P|);
    P' ← ∅;                              ▷ non-dominated individuals
    for each front F in ℱ do
        if |P'| ≥ |P| then break;
        calculate crowding distance for F;
        for each individual f in F do
            P' ← P' ∪ f;
    P' ← sorted(P', ≺_c);     ▷ see equation 3 for operator ≺_c
    P ← P'[0 : |P|];                              ▷ new population
return (M,PF,C);
```

Figura 32: Pseudocódigo de Sapienz

6. Fuzzing

El **fuzz testing** consiste en estudiar cómo un programa soporta "ruido" en su input. Los **fuzzers** son herramientas que generan inputs para un programa con el objetivo de *crashearlo*, y encontrar fallas de seguridad.

Normalmente, se buscan aserciones violentadas, buffer overflows, pérdidas de memoria, etc.

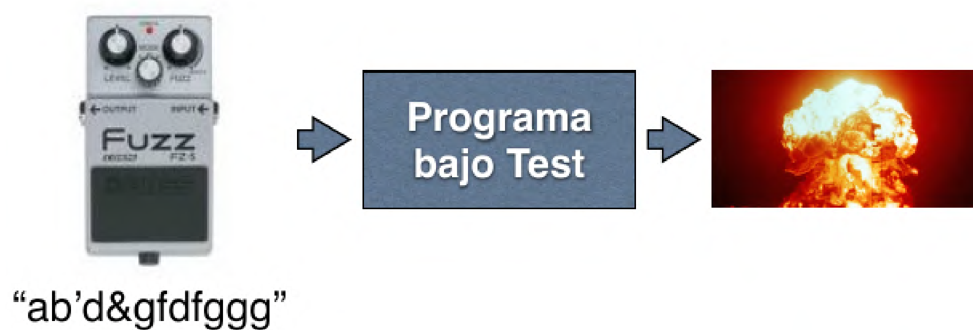


Figura 33: Fuzzing

Inicialmente, el fuzzing era hacer random testing a nivel de sistema (ir tirando entradas al azar para ver si el programa *explota*).

```
# Strings up to 1024 characters long
string_length = random choose 0..1024

 # Fill it with ASCII 32..128 characters
 out = ""
 Repeat string_length do
     out += random choose 32..128

 return out
```

Figura 34: Un fuzzer básico con strings.

Normalmente los *crashes* se dan por errores en acceso a memoria (uso de punteros, acceso a arreglos, falta de chequeo de códigos de retorno...). Este tipo de cosas son típicas de lenguajes con poca protección de memoria (sobre todo lenguajes menos modernos), que se siguen utilizando ampliamente.

El testing de caja negra (black-box) trata al programa bajo test como algo desconocido. Inicialmente, el fuzzing fue concebido como una técnica de caja negra.

6.1. Fuzzing en lenguajes con intérpretes

La aparición de intérpretes en lenguajes de programación complicó la tarea de fuzzing, ya que el intérprete rechaza las entradas que no sean válidas según el parser, y que eso ocurra es muy probable cuando se generan entradas aleatoriamente. Por eso, acaba siendo difícil ejecutar realmente el programa si el fuzzing es random.

Una manera de enfrentar esto es aprovechar que, en general, un parser viene con una gramática (descripción formal de las cadenas que acepta), con lo cual se puede usar la gramática para generar inputs sintácticamente válidos, de manera aleatoria. Esto es más efectivo que hacer

random testing clásico (aplicar aleatoriamente las reglas de la gramática se puede obtener una cadena válida para el fuzzer).

```
MAXSYMBOLS = 5

term = "$START"
while term has no terminal symbols
  rule = choose random grammar rule
  new_term = apply rule to term
  if number_of_non_terminals(new_term) < MAXSYMBOLS
   term = new_term

return term
```

Figura 35: Pseudocódigo para un generador de inputs basado en gramáticas.

Hay varias herramientas que generan inputs sobre gramáticas: LangFuzz, DomFuzz JSFuzz, CSmith, XMLMate...

6.2. Black box fuzzing

Se suele comenzar una *campaña* de fuzzing con una semilla (un conjunto inicial de inputs que serán "fuzzeados"). Estos inputs serán usados para generar nuevos inputs, de manera similar a la población inicial en un algoritmo genético.

La idea es primero ejecutar el programa con cada input de la semilla, y luego ir tomando inputs al azar de allí y mutarlos para continuar el proceso una cantidad aleatoria de veces. Algunas mutaciones posibles son:

- Insertar un carácter.
- Eliminar un carácter.
- Hacer un *flip* de un carácter o un bit.

```
Def BlackBoxFuzz(SEED):
 numberOfExecutedTests := 0
 While Budget is not empty:
  If numberOfExecutedTests < len(SEED)
     Test := SEED[numberOfExecutedTests]
  Else:
     Choose randomly input from SEED
     Test := mutate(input, K)
  Endif
  Run Test (Report if crashes/hangs/assertions)
  numberOfExecutedTests += 1
EndWhile
```

Figura 36: Pseudocódigo genérico para un fuzzing de caja negra.

6.3. Grey box fuzzing

La idea del grey box fuzzing (fuzzing de caja gris) es usar de cierta manera la información que provee la ejecución del programa (no llega a ser de caja blanca, pues no se conoce todo lo que ocurre dentro del programa).

La manera de hacer esto será medir si los inputs generados y ejecutados aportan en la cobertura de secciones no cubiertas del código. Si un input aportó cobertura, se lo agrega al cuerpo de inputs para futuras mutaciones.

Para esto, es necesario algún método de medición de cobertura (e.g. branches, basic blocks, etc.), y también habría que definir cómo representar ese aporte de información. Un enfoque es considerar la **energía** de cada input; cada elemento de la semilla tiene una energía, que se interpreta como la probabilidad de ser elegido.

Así, la semilla inicial va creciendo si un input contribuye a ampliar la cobertura, y habrá más probabilidad de elegir para mutaciones a un input que aportó cobertura.

```
Def GreyBoxFuzz(SEED):
  numberOfExecutedTests := 0
  init_len := len(SEED)
  While Budget is not empty:
    If numberOfExecutedTests < init_len
        Input := SEED[numberOfExecutedTests]
        Execute Input (Report if crashes/hangs/etc)
    Else:
        Choose randomly input from SEED
        NewInput := mutate(input, K)
        Execute NewInput (Report if crashes/hangs/etc)
        If NewInput adds new Coverage:
          SEED += NewInput
        Endif
    Endif
   numberOfExecutedTests+=1
  EndWhile
```

Figura 37: Pseudocódigo genérico para un fuzzing de caja gris.

6.3.1. Boosted grey box fuzzing

Un refinamiento de esta idea es aumentar la probabilidad de elegir un input de la semilla de acuerdo a las chances de descubrir otros caminos en el CFG (bajo la heurística de que hacer pequeños cambios sobre ese input puede cubrir más código).

La energía de un input s se define como $e(s)$, y se calcula de la siguiente manera:

$$e(s) = \frac{1}{f(p(s))^a}$$

, donde $p(s)$ es el camino en el CFG recorrido por la ejecución de s; $f(p(s))$ es la frecuencia de apariciones del camino $p(s)$ en el test suite; y a es un exponente positivo.

Observación. A mayor frecuencia de apariciones tenga un input, menor será su energía. Esto se puede interpretar desde el punto de vista de la teoría de la información: si un input apareció muchas veces, la información que provee es poca. Mientras que si un input es novedoso (aparece poco), cuando aparece representa más información.

En resumen, la heurística es que un input con más energía (que aportó cobertura menos frecuente) tiene más probabilidades de, a partir de mutaciones, descubrir nuevos caminos en el CFG.

```
Def BoostedGreyBoxFuzz(SEED):
  energy:= {}
  init_len = len(SEED)
  numberOfExecutedTests := 0
  While Budget is not empty:
    If numberOfExecutedTests < init_len
       Input := SEED[numberOfExecutedTests]
       Execute Input (Report if crashes/hangs/etc)
    Else:
       Choose input from SEED using energy(…)
       newInput := mutate(input, K)
       Execute NewInput (Report if crashes/hangs/etc)
       If NewInput adds new Coverage:
         SEED += NewInput
       Endif
    Endif
   numberOfExecutedTests += 1
   Update energy(s) for each s in SEED
  Endwhile
```

Figura 38: Pseudocódigo genérico para un fuzzing de caja gris *recargado*.

Un ejemplo conocido de fuzzing de caja gris es AFL (American Fuzzy Lop), que aplica una variante determinística del algoritmo de grey box fuzzing para fuzzing de sistemas:

1. Encolar inputs iniciales provistos por el usuario.
2. Tomar un input de la cola.
3. Reducir el input a su menor tamaño sin alterar su capacidad de cobertura.
4. Aplicar un conjunto de mutaciones al input usando distintas estrategias tradicionales de fuzzing (e.g. walking bit & byte flips, simple arithmetics, known integers, stacked tweaks).
5. Si alguno de los nuevos inputs resulta en un aumento de cobertura, agregarlo a la cola.
6. Volver a 2.

AFL se diferencia de otro fuzzer famoso, que es LibFuzzer; como su nombre lo indica, permite fuzzear una biblioteca. Para eso, necesita implementar una función que transforme un arreglo de bytes en un input válido para la biblioteca.

6.4. Sanitizers

Hay más problemas que pueden ocurrir durante la ejecución de un programa, a parte de *crashes*. Un programa *crashea* ante una falla catastrófica por un problema *serio*, pero hay otros problemas menos fáciles de detectar que también ocurren (pérdidas de memoria, mal manejo de punteros, comportamiento indefinido...).

Los sanitizers son frameworks que instrumentan un programa para realizar chequeos en tiempo de ejecución relacionados a estos problemas, con el costo de un overhead de tiempo de ejecución.

Algunos ejemplos son:

- AddressSanitizer (ASan): framework de instrumentación para detectar corrupción del manejo de memoria.
- Undefined Behavior Sanitizer (UBSan): detecta en tiempo de ejecución operaciones cuya semántica no esté bien descrita en C/C++.
- Memory Sanitizer (MSan): detecta lecturas no inicializadas en el heap.
- Leak Sanitizer (LSan): detecta pérdidas de memoria.

6.5. White box fuzzing

Este tipo de fuzzing (de caja blanca) supone que se conoce el detalle de lo que ocurre en el programa. Por eso, se lo suele asociar con la ejecución simbólica dinámica.