Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Una Evaluación Empírica del Enfoque de Sapienz para la Generación Automática de Casos de Test para Aplicaciones Android

Tesis de Licenciatura en Ciencias de la Computación

Iván Arcuschin Moreno

Director: Juan Pablo Galeotti

Buenos Aires, 2018

UNA EVALUACIÓN EMPÍRICA DEL ENFOQUE DE SAPIENZ PARA LA GENERACIÓN AUTOMÁTICA DE CASOS DE TEST PARA APLICACIONES ANDROID

Las aplicaciones Android, como cualquier otro programa, deben ser testeadas para garantizar un umbral mínimo de calidad. Es responsabilidad de los desarrolladores escribir casos de test, pero esta tarea suele llevar mucho tiempo y es propensa a errores, por lo que los desarrolladores no siempre la llevan a cabo correctamente. Las herramientas de generación automática de casos de test buscan aliviar dicha situación. Sapienz es una técnica para generación automática de casos de test para aplicaciones Android. Esta técnica utiliza un algoritmo evolutivo multi-objetivo que apunta a maximizar la cobertura y cantidad de crashes encontrados, a la vez que minimizar la longitud de los casos de test que encuentran fallas. En un estudio empírico realizado se mostró que Sapienz supera a Monkey, considerada como la más popular entre los desarrolladores Android, y Dynodroid, considerada como el estado del arte en su momento, con un larga significancia estadística en los 3 objetivos mencionados. Sin embargo, dicho estudio no contiene un análisis pormenorizado que muestre en detalle que partes de la técnica son las que le permiten superar a sus competidores. Luego, en esta tesis nos proponemos comparar Sapienz con dos variantes de si misma que deshabilitan diferentes componentes.

Palabras claves: Android, Testing, Automático, Evolutivo, Multi-Objetivo, Sapienz.

AN EMPIRICAL EVALUATION OF SAPIENZ APPROACH FOR AUTOMATICALLY GENERATING TEST CASES FOR ANDROID APPLICATIONS

ANDROID applications, like any other program, must be tested to ensure a minimum quality threshold. It is up to the developers to write test cases. However, manually writing test cases is a highly time-consuming and prone to errors task, so developers do not always fulfill it properly. Automatic test case generation tools seek to alleviate this situation. SAPIENZ is an automatic test generation technique for ANDROID applications. This technique uses a multi-objective evolutionary algorithm that aims to maximize coverage and fault detection, while also minimizing average length of failing test cases. It was shown in an empirical study that SAPIENZ outperforms MONKEY, regarded as the most popular among ANDROID developers, and DYNODROID, regarded as the state of the art at that moment. However, this study does not contain a detailed analysis that shows which parts of the technique are those that allow it to outperform its competitors. Then, in this thesis we propose to compare SAPIENZ with two variants of itself that disable different components.

Keywords: ANDROID, Automatic, Testing, Evolutionary, Multi-Objective, SAPIENZ.

AGRADECIMIENTOS

A los jurados, Carlos Lopez Pombo y Gervasio Perez, por tomarse el tiempo para leer la tesis.

A todos mis compañeros del LaFHIS, por haberme dado un lugar dónde trabajar, por los almuerzos juntos, por las tardes de música y por bancarme en todo este tiempo.

A mi director, Juan Pablo Galeotti, por tenerme paciencia y darme ánimos.

A mis amigos y compañeros de cursada, por todos los momentos compartidos durante la carrera.

A Carolina, por darme fuerza y acompañarme en incontables tardes de estudio.

A mi hermano, Nicolas, quien hace que todo sea un poco más divertido.

A mis padres, Gabriela y Oscar, por que sin su apoyo, cariño y amor no hubiera llegado hasta acá.

A mis abuelas, Berta y Estela.

Índice general

1. MOTIVACIÓN

But it is one thing to read about dragons and another to meet them.
– Ursula K. Le Guin, A Wizard of Earthsea

En los últimos años hemos presenciado un incremento sorprendente en la utilización de dispositivos móviles. En particular, el Sistema Operativo ANDROID ha sobrepasado los 2 billones de usuarios activos mensuales [1], lo cual es notable teniendo en cuenta que su primer versión fue lanzada en el año 2008, apenas 10 años atrás.

Dispositivos Móviles. Concretamente, llamamos dispositivos móviles al rango de dispositivos de cómputo portátiles tales como *smartphones* y tabletas. Dichos dispositivos suelen ser suficientemente pequeños para ser sostenidos y operados en las manos.

Típicamente, un dispositivo móvil cuenta con una interfaz de pantalla plana LCD, que provee una superficie táctil con botones digitales y un teclado, digital o físico. La mayoría de estos dispositivos son capaces de conectarse a internet y de comunicarse con otros dispositivos mediante Wi-Fi, Bluetooth, redes celulares o NFC (Near Field Communication). Otras capacidades comunes suelen ser: cámaras integradas, reproductor de música, recibir y hacer llamadas, video juegos, y sistemas de ubicación (ej. GPS).

Sistemas Operativos Móviles. Los dispositivos móviles funcionan con lo que se denomina como Sistemas Operativos Móviles, que permiten instalar y utilizar variadas aplicaciones o programas. Entre los más populares, encontramos a ANDROID y iOS.

ANDROID, desarrollado por Google, es por lejos el más utilizado a nivel mundial[2]. Si bien es un sistema operativo gratuito y de *código abierto*, en los dispositivos vendidos una buena parte del software que viene incorporado (incluyendo aplicaciones de Google, así como aplicaciones específicas del fabricante) es software propietario y de *código cerrado*.

iOS (anteriormente llamado *iPhone OS*), desarrollado por Apple, es el segundo más utilizado a nivel mundial. Es de *código cerrado* y propietario.

Aplicaciones Móviles. Llamamos aplicaciones móviles (*apps*) a los distintos programas o software que uno puede instalar en un Sistema Operativo Móvil.

El mencionado incremento en la cantidad de usuarios ANDROID ha tenido asociado también un gran incremento en la cantidad de aplicaciones existentes.

Los usuarios ANDROID suelen buscar y descargar aplicaciones desde la tienda virtual *Google Play Store*, la cuál funciona como la tienda oficial de aplicaciones para ANDROID. Se estima que la tienda *Google Play Store* ha procesado durante el año 2016 más de 82 billones de descargas y a Junio de 2018 contiene más de 3.7 millones de aplicaciones publicadas[3].

Luego, dada la mencionada popularidad de la plataforma, el foco de la tesis está en cómo mejorar la calidad de las aplicaciones ANDROID. En particular, trabajaremos sobre una herramienta para *testing* automático en ANDROID.

2. PRELIMINARES

2.1. Testing de software

El testing de software consiste en observar la ejecución de un programa para validar si se comporta como es esperado y para identificar posibles errores.

Dado que el proceso de testing apunta a mejorar la calidad del software, suele ser una etapa muy importante y costosa de la Ingeniería de Software. Se estima que estos costos pueden llegar al cincuenta por ciento del costo total de desarrollo de un sistema[4].

Una vez finalizado el proceso de testing, el resultado es un conjunto de casos de test, también llamado ***test suite***.

Definición 2.1.1. Definimos un **caso de test** como una entidad compuesta por:

- Una entrada (o *input*) para el programa a probar.
- Un procedimiento que verifica que el programa se comporta como es esperado frente a la entrada provista. Dicho procedimiento se suele llamar *oráculo*.

A su vez, se suelen dividir los casos de test en dos categorías:

- **Casos de test unitarios**: aquellos que ejercitan partes individuales de un programa (ej: un método, una clase, etc.).
- **Casos de test de sistema**: aquellos que ejercitan todo el sistema en su conjunto. Apuntan a detectar errores en la interacción entre las distintas partes o módulos de un programa.

Luego, un conjunto de casos de test nos da la garantía de que las partes del programa que fueron ejercitadas, y que satisfacen correctamente el *oráculo* asociado, se comportan de acuerdo a lo esperado. Es importante hacer notar que el testing sólo muestra la presencia de errores, y no su ausencia [5]. Es decir, podría haber errores en nuestro programa que no fueron detectados por el conjunto de casos de test, ya sea porque algunas funcionalidades no fueron ejercitadas, o porque el *oráculo* asociado no era correcto.

Como se puede ver, un conjunto de casos de test es efectivo solo si los test son lo suficientemente buenos y extensos para cubrir toda la funcionalidad deseada. Sin embargo, escribir manualmente casos de test es una tarea tediosa y propensa a errores que suele requerir mucho tiempo, y por lo tanto dinero, y los desarrolladores no siempre la realizan de manera adecuada.

Es por esta razón que a lo largo de los años han surgido distintas formas de automatizar los procesos involucrados[6]. Uno de estos procesos, y en el cual nos centraremos en esta tesis, es el de generación automática de casos de test.

2.2. Generación automática de casos de test

La generación automática de casos de test, como su nombre lo indica, consiste en generar casos de test para un sistema o programa en particular (que llamaremos sujeto a

```
def(x):
  if (x > 0)
    return x;
  else
    return -x;
```

Figura 2.1: En este ejemplo se define una función que devuelve el valor absoluto de un número. Debido a que hay una estructura de control *if-else*, aparecen dos ramas: la rama del cuerpo del *if* (cuando x es positivo) y la rama del cuerpo del *else* (cuando x es menor o igual a cero).

testear). También es posible que el sujeto sea una parte o módulo de un programa (por ejemplo, una clase).

Notar que la naturaleza del testing es encontrar contra-ejemplos (casos de test) que muestren que el sujeto se comporta de manera incorrecta bajo ciertas condiciones. Por esta razón, no hay un tiempo límite a cuanto puede durar este proceso: puede ser terminado en cualquier momento y el resultado final es el conjunto de casos de test descubiertos hasta dicho momento. En la práctica, el proceso de generación de casos de test se pone a correr con un tiempo límite prefijado, que llamaremos **presupuesto de búsqueda**.

A la hora de guiar la generación de casos de test, no es extraño utilizar algún criterio de cobertura. Un **criterio de cobertura** consiste en un conjunto finito de objetivos de cubrimiento, y un enfoque común es atacar cada uno de esos objetivos a la vez, generando *test inputs* ya sea simbólicamente [7] o utilizando alguna técnica basada en búsqueda [8].

Llamamos un **objetivo de cubrimiento** a una parte del código del programa que se quiere ejercitar mediante un caso de test. Usualmente esas partes suelen ser, o bien lineas del código, o bien ramas. Se entiende por rama a cada uno de los ejes en el *control-flow graph* de un programa. Es decir, para todo "if" en el programa, hay dos ramas: una para el caso "then" (cuando la guarda del "if" evalúa a verdadero) y otra para el "else" (cuando la guarda evalúa a falso). Un ejemplo de esto puede verse ilustrado en la Figura 2.1.

Luego, hay distintas métricas populares en la literatura [9] que utilizan esta intuición para comparar *test suites*:

Definición 2.2.1. Definimos la métrica **cobertura de lineas** de un *test suite*, con respecto a un sujeto a testear, como el porcentaje del total de las lineas de código del sujeto que fueron ejercitadas por los casos de test. Que una linea, o cualquier otra porción de código, sea ejercitada por un caso de test quiere decir que en algún punto de la ejecución del caso de test, el programa pasó por la linea en cuestión.

Definición 2.2.2. Definimos la métrica **cobertura de ramas** de un *test suite*, con respecto a un sujeto a testear, como el porcentaje del total de las ramas en el código del sujeto, que fueron ejercitadas al ejecutar los casos de test.

Definición 2.2.3. Definimos la métrica **crashes alcanzados** por un *test suite*, con respecto a un sujeto a testear, como la cantidad de *crashes*, que dicho *test suite* logra provocar en el sujeto. Decimos que un programa incurre en un **crash** cuando este deja de funcionar correctamente y el sistema operativo fuerza el cierre del mismo.

Definición 2.2.4. Definimos la métrica **largo promedio** de los casos de test de un *test suite*. Es deseable que esta métrica sea lo más chica posible, ya que esto ayuda a que los

desarrolladores puedan comprender de manera más sencilla cual es el comportamiento que se está ejercitando sobre el sujeto. Sin embargo, es importante notar que el valor mínimo para este métrica es trivial: el caso de test vacío tiene longitud cero. Por esta razón, sólo vamos a calcular el largo promedio para aquellos *test suite* que logren provocar un *crash* en el sujeto. Si ningún caso de test del conjunto logra provocar un *crash*, decimos que esta métrica se indefine.

2.2.1. Problemas de la generación de casos de test

A pesar de existir muchos enfoques diferentes para generar casos de test, uno de los problemas comunes que sigue presente al día de hoy es el **problema del oráculo**. Si bien algunos programas poseen propiedades esenciales, que o bien están formalmente especificadas o bien deben cumplirse universalmente (por ejemplo, los programas normalmente no *crashean*) de forma que no hace falta definir un oráculo *explícito*, en el caso general uno no puede asumir que existe tal oráculo. Esto significa que por más que logremos generar automáticamente entradas para el programa, una persona deberá encargarse de definir manualmente un oráculo.

Para hacer que esta tarea sea lo más sencilla posible, la generación de tests debe apuntar no sólo a obtener un buen grado de cobertura de código, sino también a generar *test suites* pequeños, de forma que los desarrolladores que utilicen dichos tests puedan entender cual es el comportamiento incorrecto que está sucediendo.

2.2.2. Enfoque *random* para generar casos de test

Un primer acercamiento sería el enfoque *random*: generar aleatoriamente casos de test hasta encontrar uno que cubra el objetivo planteado (por ejemplo, una rama o linea del código). El problema con este enfoque es que muchas veces el *espacio de búsqueda* asociado a un objetivo puede ser muy grande (e incluso insatisfacible, como veremos más adelante). Un ejemplo trivial sería si tenemos un programa con una condición $x == 10$; el enfoque *random* podría generar infinitos casos de test: uno por cada valor de $x \in \mathbb{N}$.

Es por esta razón que suelen ser necesarias otro tipo de estrategias, como las meta-heurísticas de búsqueda, que utilizan funciones auxiliares que ayudan a guiar el proceso de búsqueda.

2.3. Algoritmos genéticos para generar casos de test

Los algoritmos genéticos son un tipo de estrategia que utiliza meta-heurísticas de búsqueda e intentan imitar los mecanismos de selección natural. Una población de individuos es evolucionada utilizando operadores genéticamente inspirados, dónde cada individuo representa una posible solución al problema que se quiere resolver.

En años recientes, las estrategias de algoritmos genéticos han sido aplicadas exitosamente al problema de generar casos de test unitarios para programas Java [10]. Intuitivamente, el proceso típico de un algoritmo genético es:

- En primer lugar, se comienza generando aleatoriamente una población.
- Luego, varias iteraciones (también llamadas, generaciones) evolucionan la población hacía el objetivo deseado.

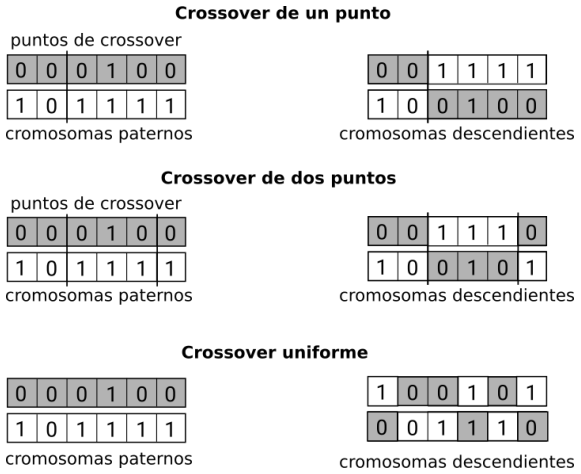

Figura 2.2: Variantes del operador de *crossover* ilustradas de manera esquemática. En la práctica, un cromosoma podría ser un caso de test, representado por ejemplo como una secuencia de líneas de código. Luego, realizar *crossover* entre dos casos de test, implica decidir que líneas de qué padre terminan en cada uno de los descendientes.

- Para producir una nueva generación, los individuos más *aptos* son seleccionados de acuerdo a algún mecanismo. Por ejemplo, *rank selection* [11] ordena la población de acuerdo a su aptitud y arma un *ranking*. Luego, la probabilidad de que cada individuo sea seleccionado depende de la posición en la cual quedó en el *ranking*: el primero tendrá probabilidad 1/2, el segundo 1/3, el tercero 1/4, etc.

- Después de esto, se genera una nueva descendencia aplicando operadores genéticos como *crossover* y mutación con ciertas probabilidades paramétricas.

El operador de ***crossover*** (también llamado operador de *recombinación*) se utiliza para combinar la información genética de dos padres y obtener una nueva descendencia. En otras palabras, es una forma de generar estocásticamente nuevas soluciones en base a una población existente, análogamente a la reproducción sexual que sucede en biología. Otra forma de generar soluciones podría ser la de clonar una solución existente, que sería análogo a la reproducción asexual. Hay diferentes variantes populares del operador de *crossover* [12], ilustradas en la Figura 2.2.

- *Crossover* de un punto: Se elige un punto aleatorio en los cromosomas de ambos padres, y se designa como "punto de *crossover*". La información genética a la derecha de ese punto se intercambia entre dichos cromosomas. El resultado es dos hijos, cada uno llevando genes de ambos padres.

- *Crossover* de dos puntos: En esta variante, se elijen dos puntos aleatorios en los cromosomas de los padres. La información genética a intercambiar es la que se encuentra entre los puntos. Esta variante es equivalente a realizar *crossover* de un punto dos veces, utilizando puntos distintos. Notar que se puede generalizar a *crossover* de k puntos (con $k \in \mathbb{N}$).

- *Crossover* uniforme: Cada gen en los cromosomas de la descendencia es elegido independientemente de los dos padres de acuerdo a una distribución elegida. Típicamente,

cada gen se elige de algún padre con igual probabilidad. Es decir, con un 50 % de probabilidad de mezcla y dados dos padres x_1 y x_2, los hijos x'_1 y x'_2 tendrán cada uno la mitad de los genes de un padre y la mitad del otro.

Las soluciones generadas se suelen mutar antes de ser agregadas a la nueva población. Esto permite introducir material genético no presente en ninguno de los dos padres. Ejemplos de operadores de mutación pueden ser alterar una linea o evento dentro de un caso de test.

Notar que para saber cuan *apto* es un individuo necesitamos una **función de *fitness***, que va a estar relacionada con el objetivo que querramos cubrir. Por ejemplo, si estamos hablando de cubrimiento de ramas, lo más usual es utilizar una combinación de *branch distance* y *approach level* [13]. La métrica *approach level* nos indica cuan cerca estuvo el programa de ejecutar el nodo en el *control flow graph* del cual depende la rama que queremos ejecutar. La métrica *branch distance* nos indica con alguna heurística cuan lejos está un predicado de obtener su valor opuesto. Es decir, si el predicado es verdadero, entonces *branch distance* nos dirá cuan lejos está el input actual de otro que haga valer falso el predicado.

2.3.1. Enfoque *Single Target*

Ahora, es posible que dado un sujeto a testear querramos generar casos de test para cubrir más de una rama o linea de código. Es decir, queremos por ejemplo un conjunto de casos de tests S que logren cubrir todas las ramas B del programa en cuestión. En este caso estamos ante una situación en la que tenemos varios *goals* para atacar.

El enfoque *Single Target* consiste en dividir el presupuesto total de búsqueda entre todos los *goals* disponibles, y utilizar cada uno de estos *slots* para trabajar en un *goal* a la vez. Al finalizar, se combinan en un *test suite* todos los casos de test que llegaron a cubrir un *goal*.

Uno de los problemas con este enfoque es que es difícil de predecir el tamaño final del *test suite* ya que un caso de test generado para un *goal* podría llegar a cubrir implícitamente otros *goals* distintos. Esto se suele llamar *cubrimiento colateral*.

Otro problema que surge cuando se intenta atacar un *goal* a la vez es que no todos los *goals* se logran cubrir con la misma dificultad. Un ejemplo típico de esto son los ***goals* insatisfacibles**, es decir, aquellos para los no existe ningún input que los cubra. Este problema en general es indecidible [14]. Luego, por definición, intentar cubrir un *goal* insatisfacible va a ser tiempo gastado en vano.

Esto lleva a la pregunta de cómo asignar adecuadamente el tiempo total disponible a los distintos objetivos, y cómo redistribuir este tiempo a medida que los objetivos se van cubriendo.

2.3.2. Enfoque *Whole test suite generation*

La técnica de *Whole test suite generation* [10] (generación de conjuntos de casos de test de forma completa) propone utilizar una técnica evolutiva en la que, en lugar de evolucionar cada caso de test por separado, se evolucionan todos los casos de test de un *test suite* al mismo tiempo, y la función de *fitness* considera todos los *goals* a cumplir simultáneamente. Para lograr esto, en la técnica mencionada se define la función de *fitness* para un *test suite* como la sumatoria de la *fitness* de cada *goal* a cubrir:

$$fitness(T) = |M| - |M_T| + \sum_{b_k \in B} d(b_k, T)$$

Dónde $|M|$ es la cantidad de métodos totales, $|M_T|$ es la cantidad de métodos ejecutados por el test suite, b_k es una rama a cubrir del programa y $d(b_k, T)$ es el *branch distance* normalizado para la rama b_k y el *test suite* T. Concretamente,

$$d(b,T) = \begin{cases} 0, & \text{si se cubrió la rama,} \\ \nu(d_{min}(b,T)), & \text{si la rama se ejecutó al menos una vez,} \\ 1, & \text{si no.} \end{cases}$$

De esta forma, minimizar esta función implica: ejecutar todos lo métodos disponibles del programa y ejecutar todas las ramas dentro de cada uno de los métodos.

La técnica comienza con una población inicial de *test suites* (cada cual contiene varios casos de test) generados aleatoriamente, y luego utiliza un algoritmo genético para guiar la optimización usando la función de *fitness* presentada, mientras se usa el largo del *test suite* como objetivo secundario. Es decir, se utiliza alguna técnica para elegir los individuos con mejor *fitness* (por ejemplo, *rank selection*), y en caso de empates, se eligen aquellos *test suites* que tengan menor largo.

Notar que incluir el largo del *test suite* directamente en la función de *fitness* podría llegar a tener efectos no deseados porque necesitaríamos combinar linealmente dos valores para diferentes unidades de medición. Sin embargo, es importante mencionar que este mismo "truco" de combinar linealmente diferentes objetivos podría utilizarse de manera análoga para otros objetivos de cubrimiento (por ejemplo, lineas de código).

Al finalizar, el *test suite* que tenga mejor *fitness* es minimizado. Existe evidencia de que esta técnica permite conseguir una mayor cobertura general que la técnica tradicional que se enfoca en los objetivos de forma individual, y resultados que apoyan la validez y efectividad de esta técnica en la generación automática de casos de test [10].

2.3.3. Variantes de algoritmos genéticos

Dentro de los algoritmos genéticos hay algunas variantes clásicas [15]:

Algoritmo genético "Standard": Comienza creando una población inicial de tamaño p_n. Luego, se seleccionan un par de individuos de la población utilizando alguna estrategia de selección (por ejemplo, *rank selection*). A continuación, ambos individuos seleccionados se recombinan utilizando *crossover* con una probabilidad c_p para producir dos nuevos individuos o_1 y o_2. Después, se aplica algún operador de mutación en la descendencia generada, cambiando los genes independientemente con probabilidad m_p, usualmente igual a $\frac{1}{n}$, dónde n es la cantidad de genes en el cromosoma. Los dos individuos mutados se agregan a la nueva población. Al final de cada iteración se calcula el *fitness* de todos los individuos en la nueva población.

Algoritmo genético "Monotonic": Es una variación de *Standard* que, después de mutar y evaluar cada descendencia, sólo incluye o bien la mejor descendencia o bien el mejor padre en la nueva población (mientras que *Standard* siempre incluye la descendencia nueva, sin importar su *fitness*).

Algoritmo genético "Steady state": Es una variación de *Standard* que usa la misma técnica de reemplazo que *Monotonic*, pero en vez de crear una nueva población

utilizando la descendencia, los nuevos individuos van reemplazando a los padres en la población actual inmediatamente después de la fase de mutación.

Algoritmo genético $1 + (\lambda, \lambda)$: Introducido por Doerr et al. [16], comienza generando una población inicial aleatoria de tamaño 1. Luego, se utiliza mutación para crear λ versiones distintas del individuo actual. La mutación se aplica con una probabilidad alta, definida como $m_p = \frac{k}{n}$, dónde k es típicamente más grande que uno, lo que permite que en promedio se mute más de un gen por cromosoma. Después, se aplica *crossover* uniforme entre el padre y el mejor mutante para generar λ individuos de descendencia. La intención al utilizar una alta probabilidad de mutación es explorar más rápidamente el espacio de búsqueda, mientras que el *crossover* uniforme entre el mejor mutante y el padre se utiliza para tratar de reparar los defectos causados por la mutación agresiva. A continuación, se evalúa toda la descendencia y se elije el que tenga mejor *fitness*. Si este individuo es mejor que el padre, entonces la población de tamaño 1 se reemplaza por dicho individuo. Este algoritmo puede ser bastante caro para valores grandes de λ, ya que la función de *fitness* tiene que evaluarse después de la mutación y después del *crossover*.

Algoritmo genético $\mu + \lambda$: Como su nombre sugiere, el número de padres y descendencia está restringido a μ y λ, respectivamente. Cada gen es mutado independientemente con probabilidad $\frac{1}{n}$. Luego de la mutación, la descendencia generada se compara contra cada padre, tratando de preservar los mejores individuos hasta el momento, incluyendo padres. Es decir, los padres son reemplazados una vez que se encuentra una mejor descendencia.

2.4. Testing multi-objetivo

En muchos casos es deseable optimizar los casos de test generados para más de una métrica. El ejemplo canónico es cuando uno quiere generar casos de test con la mayor cobertura posible (dado que ejecutar una porción de código es un requisito necesario para poder encontrar fallas en dichas lineas) que tengan la menor longitud posible. Sin embargo, estos dos objetivos son **contradictorios**: a mayor longitud de casos de test, más fácil es aumentar la cobertura, y por el contrario mientras más chico es el caso de test, más difícil es obtener una cobertura alta.

En este contexto, los algoritmos genéticos multi-objetivos se centran en atacar el problema de satisfacer múltiple objetivos al mismo tiempo (incluso contradictorios).

2.4.1. Optimalidad de Pareto

Intuitivamente, los problemas de optimización multi-objetivo buscan optimizar los componentes de una función vectorial. A diferencia de la optimización de un sólo objetivo, la solución a este problema no es un sólo punto en el espacio de búsqueda, sino una familia de puntos, conocidos cómo el **conjunto de Pareto óptimo**[17].

Cada punto en esta superficie es óptimo en el sentido de que ninguno de los otros puntos del conjunto puede mejorar una componente del vector sin que esto lleve a una degradación en alguna otra componente.

Formalmente,

Definición 2.4.1. Una solución $x = (x_1, \ldots, x_n)$ se dice que es **inferior** a otra $y = (y_1, \ldots, y_n)$ si y sólo si y es parcialmente menor que x:

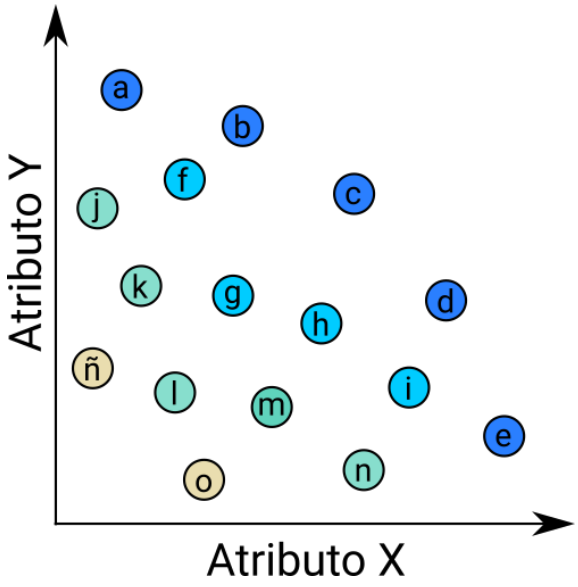

Figura 2.3: Ejemplo esquemático de frentes de Pareto sobre dos atributos X e Y. Los individuos a-e constituyen el primer frente de Pareto (el conjunto de Pareto óptimo) dado que cada uno de ellos no es inferior a ningún otro individuo de la población. Los individuos f-i constituyen el segundo frente de Pareto, ya que cada uno de estos sólo es inferior a algún individuo del primer frente. Los individuos j-n constituyen el tercer frente y los individuos ñ-o el cuarto.

$$\forall\ i = 1, \ldots, n,\ y_i \leq x_i \quad \wedge \quad \exists\ i = 1, \ldots, n : y_i < x_i$$

En dicho caso, podemos decir también que x es **superior** a y.

Notar además que puede pasar que x y y no sean ni inferiores ni superiores entre ellas.

Luego, una frontera de Pareto es un conjunto de soluciones tales que dentro de dicho conjunto ninguna solución es superior ni inferior a otra. El conjunto de Pareto óptimo es la frontera de Pareto tal que cada una de sus soluciones es no-inferior a cualquier otra solución en las demás fronteras. Un ejemplo de esto puede verse en la Figura 2.3.

Entonces, cada elemento en un conjunto de Pareto óptimo constituye una solución no-inferior al problema de optimización multi-objetivo.

Existen algoritmos (por ejemplo, NSGA-II [18]) que permiten, dado un conjunto de individuos con su correspondiente evaluación de función de *fitness* vectorial, obtener las distintas fronteras de Pareto existentes, y por lo tanto obtener el conjunto de Pareto óptimo. En el contexto de algoritmos evolutivos, esto se puede utilizar para generar una descendencia a partir de los individuos más aptos de la población actual.

2.5. Testing de Aplicaciones Móviles

Como todo software, las aplicaciones Android deben ser *testeadadas* correctamente por sus desarrolladores para evitar *crashes* y otros errores de comportamiento no deseados.

Uno de los problemas adicionales que tienen que enfrentar los desarrolladores de aplicaciones es la gran fragmentación de dispositivos Android [19] que hay en el mercado, lo cual aumenta el esfuerzo que deben realizar dada la cantidad de dispositivos que deben ser

considerados. Esto se suma a las diferentes versiones del Sistema Operativo ANDROID que coexisten al mismo tiempo en el mercado. De acuerdo a un estudio acerca del desarrollo de aplicaciones [20], el testing de aplicaciones móviles todavía utiliza fuertemente pruebas manuales, mientras que el uso de técnicas automáticas todavía es raro [21].

En los casos en los cuales se utiliza alguna técnica automática, esta suele ser la herramienta MONKEY [22], desarrollada por Google, e integrada actualmente en el sistema ANDROID. Dado que esta herramienta está ampliamente disponible y distribuida, se la considera como la más utilizada entre los desarrolladores ANDROID.

Si bien MONKEY realiza testing automático, lo hace de una manera bastante simple: generando secuencias de eventos aleatorios con la idea de explorar la aplicación bajo prueba y revelar fallas existentes. El *oráculo implícito* que utiliza es simple pero efectivo, considerando cualquier secuencia de inputs que llevan a un *crash* cómo una secuencia que muestra una falla.

El principal problema con este enfoque es que dichas secuencias de eventos pueden ser excepcionalmente largas, lo cual las hace imprácticas para que los desarrolladores investiguen la razón de la falla. Además, mientras más larga es una secuencia de eventos menos probable es que ocurra en la práctica. Por lo tanto, uno de los principales objetivos de cualquier herramienta para testing automático es encontrar fallas con la menor cantidad posible de eventos, haciendo que estas sean más útiles para los desarrolladores.

Otra herramienta disponible es DYNODROID [23]. Dicha herramienta también está basada en exploración aleatoria, pero tiene diversas características que hacen que su exploración sea más eficiente comparada con MONKEY. Primero que nada, es capaz de generar eventos de sistema, y lo hace chequeando cuales son los relevantes para la aplicación a testear. DYNODROID obtiene esta información monitoreando cuando una aplicación registra un receptor de eventos en el framework ANDROID, por lo que necesita instrumentar dicho framework. La estrategia de generación de eventos aleatorios de DYNODROID es más inteligente que la que utiliza MONKEY. Por un lado, puede seleccionar eventos que han sido usados la menor cantidad de veces (estrategia por *frecuencia*) y por otro puede llevar registro del contexto de la aplicación, seleccionando los eventos que sean relevantes en el momento. Una característica adicional de DYNODROID es que permite que el usuario provea cadenas de texto para usar cuando la exploración se bloquea (por ejemplo, para autenticación).

Es importante hacer notar que los casos de test generados por MONKEY y DYNODROID entran en la categoría de test de sistema, ya que el input de los casos de test son secuencias de eventos que interactúan directamente con el sistema ANDROID dónde está corriendo la aplicación bajo prueba.

2.6. Enfoque SAPIENZ

SAPIENZ [24] es una herramienta de generación automática multi-objetivo de casos de test para aplicaciones ANDROID, que apunta a maximizar la cobertura y cantidad de crashes encontrados, a la vez que minimizar la longitud de los casos de test que encuentran fallas.

SAPIENZ utiliza un algoritmo evolutivo $\mu + \lambda$ dónde los individuos son *test suites* y el valor de la función de *fitness* se registra cómo una 3-upla dónde cada componente representa uno de los objetivos: cobertura de lineas, largo promedio de los casos de test que encuentras crashes y cantidad de crashes encontrados.

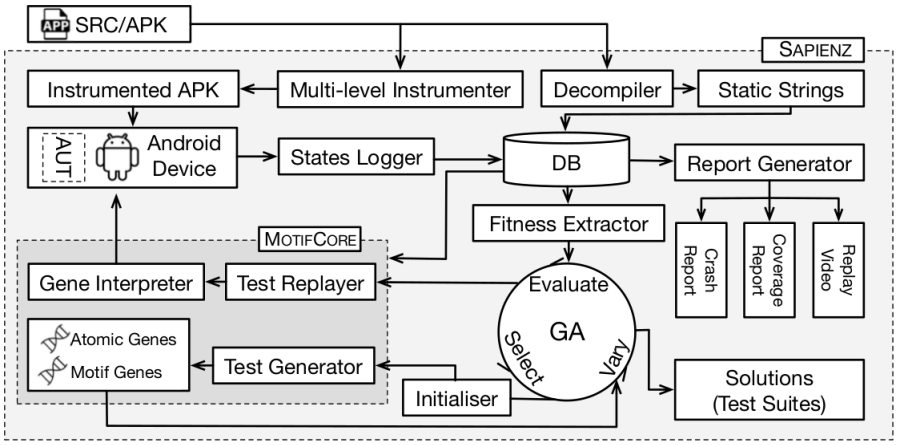

Figura 2.4: Flujo de trabajo de SAPIENZ.

Si bien la evaluación de casos de test puede ser una actividad muy costosa en términos de tiempos, también es altamente paralelizable. Por esta razón, SAPIENZ soporta la evaluación de casos de test en paralelo. Dado un conjunto de dispositivos físicos o emuladores, se les van asignando evaluaciones distintas al mismo tiempo. Si algún dispositivo termina y todavía quedan evaluaciones por realizar, se le asigna otra.

2.6.1. Algoritmo

El flujo de trabajo general de SAPIENZ se ilustra en la Figura 2.4. La herramienta comienza instrumentando la aplicación a testear. Esta instrumentación permite luego obtener los porcentajes de cobertura de la aplicación al ejecutar un caso de test. En este paso además se extraen las cadenas de texto definidas de manera estática en el código fuente. Dichas cadenas son luego usadas cómo entrada "realista" dentro la aplicación.

Las secuencias de eventos para los casos de test son generadas y ejecutadas por un componente especial llamado MOTIFCORE, el cual combina *fuzzing aleatorio* y exploración sistemática, que corresponden a dos tipos de genes: aquellos de bajo nivel, llamados *genes atómicos*, y los de alto nivel, llamados *genes motif*. Dicho componente se inyecta en el dispositivo antes de comenzar el algoritmo genético. Durante la evolución, el controlador se comunica con el MOTIFCORE mediante el comando `adb` (*Android Debugging Bridge*).

El Algoritmo 1 presenta el algoritmo completo de SAPIENZ. Cada *test suite* x es una solución para la aplicación bajo prueba, y una solución x_a **es dominada** por otra solución x_b ($x_a \prec x_b$) de acuerdo a una función de *fitness* f si y sólo si,

$$\forall\ i = 1, \ldots, n,\ f_i(x_a) \leq f_i(x_b) \quad \wedge \quad \exists\ i = 1, \ldots, n : f_i(x_a) < f_i(x_b)$$

Es decir, $f(x_a)$ es *inferior* a $f(x_b)$ con respecto a la noción de optimalidad de Pareto.

Entonces, el conjunto de Pareto óptimo consiste de todas las soluciones Pareto-óptimas (pertenecientes al conjunto de todas las soluciones, X_t):

$$P^* \triangleq \{x^* | \nexists\ x \in X_t,\ x^* \prec x\}$$

Dado que SAPIENZ realiza evolución completa de los *test suite*[10], cada individuo corresponde a uno de estos. La representación interna de un individuo se ilustra en la

Algoritmo 1: Algoritmo SAPIENZ

Entrada: Aplicación A, probabilidad de crossover p, probabilidad de mutación q, máxima cantidad de generaciones g_{max}, tiempo disponible t
Salida : Modelo de UI M, frente de Pareto PF, casos de test C

$M \longleftarrow K_0$; $PF \longleftarrow \emptyset$; $C \longleftarrow \emptyset$; `// inicialización`
generación $g \longleftarrow 0$;
iniciar dispositivos D;
instalar MOTIFCORE en D ; `// para exploración híbrida`
análisis estático de A ; `// para string seeding`
instrumentar e instalar A;
inicializar población P ; `// híbrido entre genes aleatorios y genes motif`
evaluar P con MOTIFCORE y actualizar (M, PF, C);
mientras $g < g_{max} \wedge \neg\, timeout(t)$ **hacer**
  $g \longleftarrow g + 1$;
  $Q \longleftarrow obtenerDescencencia(P, p, q)$; `// ver Algoritmo 3`
  evaluar Q con MOTIFCORE y actualizar (M, PF, C);
  $\mathcal{F} \longleftarrow \emptyset$; `// frentes no dominados`
  $\mathcal{F} \longleftarrow ordenarNoDominado(P \cup Q, |P|)$;
  $P' \longleftarrow \emptyset$; `// individuos no dominados`
  para cada *frente* F en $\mathcal{F}$ **hacer**
    si $|P'| \geq |P|$ **hacer**
      break;
    calcular métrica de cercanía para F;
    para cada *individuo* f en F **hacer**
      $P' \longleftarrow P' \cup \{f\}$;
  $P' \longleftarrow ordenar(P', \prec_c)$;
  $P \longleftarrow P'[0 : |P'|]$; `// nueva población`
devolver (M, PF, C)

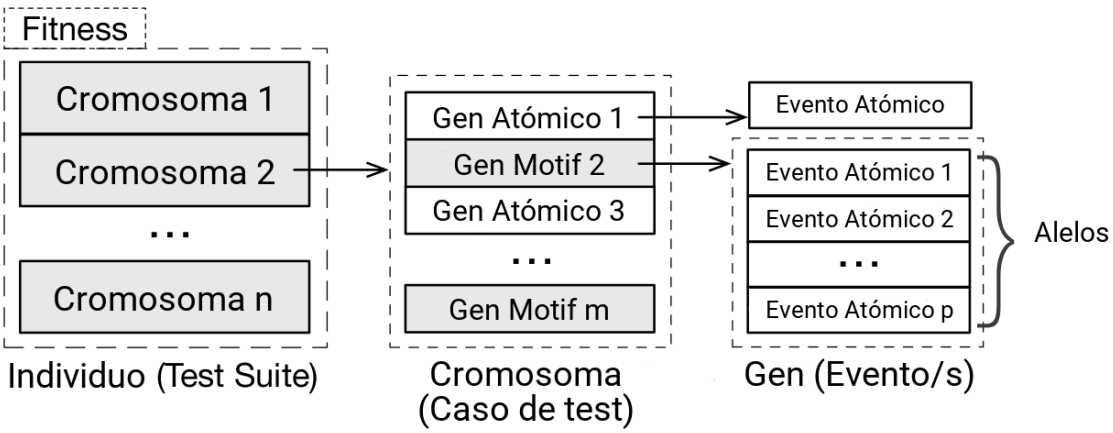

Figura 2.5: Representación genética de un individuo en SAPIENZ.

Algoritmo 2: La estrategia de exploración del componente MOTIFCORE

Entrada: Aplicación A, caso de test $T = \langle E_1, E_2, \ldots, E_n \rangle$, lista de eventos aleatorios $\mathcal{R}$, lista de eventos *motif* $\mathcal{O}$, cadenas de texto estáticas $\mathcal{S}$, modelo de UI M y reportes de test C
Salida : Se actualiza (M, C)

```
para cada evento E en T hacer
    si E ∈ R                                   // aplicar gen atómico hacer
        ejecutar el evento atómico E y actualizar M
    si E ∈ O                                   // aplicar gen motif hacer
        pantallaActual ⟵ obtenerPantallaActual(M);
        elementosDeUI ⟵ obtenerElementosDeUI(pantallaActual);
        para cada elemento w en elementosDeUI hacer
            si w es un campo de texto hacer
                insertar alguna cadena de texto s ∈ S en w;
            sino
                ejercitar w de acuerdo a los patrones motif en E;
            actualizar M;
    (a, m, s) ⟵ obtener actividades, métodos y lineas cubiertas;
    C ⟵ C ∪ (a, m, s) ;                        // actualizar reportes de cobertura
si se capturó un crash c hacer
    C ⟵ C ∪ c ;                                // actualizar reportes de crash
devolver (M, C)
```

Figura 2.5. Cada individuo consiste de varios cromosomas (secuencias de test $\langle T_1, \ldots, T_m \rangle$) y cada cromosoma contiene múltiples genes (secuencias de eventos $\langle E_1, \ldots, E_n \rangle$), que están compuestos de una combinación aleatoria de genes atómicos y genes *motif*.

Un **gen atómico** lleva a cabo un evento atómico que no puede ser descompuesto en más partes, por ejemplo: presionar una tecla. Por otro lado los **genes *motif*** son interpretados como una serie de alelos (eventos atómicos $\langle e_1, \ldots, e_p \rangle$).

Cada gen *motif* actúa basándose en la información de la UI (*widgets* para las aplicaciones ANDROID) disponible en el momento de accionarse. Dichos genes se utilizan para ejecutar patrones de uso en la aplicación, por ejemplo: llenar todos los campos de texto en la pantalla actual y clickear el botón de "aceptar". Esto se logra pre-definiendo patrones que capturen la experiencia de un *tester* manual con respecto a las interacciones complejas en la aplicación.

En este sentido, los genes *motif* están inspirados en como funcionan las secuencias *motif* ADN: secuencias cortas que establecen un patrón que tiene una función biológica. Estos *motif* se combinan con secuencias atómicas de forma que juntos pueden expresar la funcionalidad completa del ADN.

En el caso de SAPIENZ, los genes *motif* buscan ejercitar patrones usuales y trabajar en conjunto con los genes atómicos para lograr un alto grado de cobertura. En este sentido, la exploración de SAPIENZ es híbrida: los genes atómicos y *motif* son complementarios, por lo que SAPIENZ los combina para formar secuencias de eventos híbridas. Es decir, la exploración aleatoria puede que logre cubrir estados de la UI utilizando combinaciones

Algoritmo 3: Algoritmo para generar descendencia en SAPIENZ

```
Entrada: Población P, probabilidad de crossover p, probabilidad de mutación q
Salida : Descendencia Q
Q ⟵ ∅;
para i en rango(0, |P|) hacer
    generar r ∼ U(0, 1);
    si r < p                                          // aplicar crossover hacer
        seleccionar aleatoriamente padres x1, x2;
        x'1, x'2 ⟵ crossoverUniforme(x1, x2);
        Q ⟵ Q ∪ {x'1, x'2};
    sino si r < p + q                                 // aplicar mutación hacer
        seleccionar aleatoriamente individuo x1;
        // variar casos de test dentro del test suite x1
        x ⟵ mezclarIndices(x1);
        para j en rango(1, |x|, step 2) hacer
            generar r ∼ U(0, 1);
            si r < q hacer
                x[i − 1], x[i] ⟵ crossoverDeUnPunto(x[i − 1], x[i]);
        // variar los eventos dentro del caso de test x[i]
        para j en rango(0, |x|) hacer
            generar r ∼ U(0, 1);
            si r < q hacer
                x[i] ⟵ mezclarIndices(x[i]);
        Q ⟵ Q ∪ {x};
    sino
        Q ⟵ Q ∪ {x1elegido aleatoriamente} ;          // aplicar reproducción
devolver Q
```

aleatorias de eventos atómicos, pero en general logra baja cobertura total. La exploración sistemática por otro lado puede lograr buena cobertura dentro de un conjunto de estados de la UI planificados, pero puede llegar a bloquearse en regiones no planificadas. La estrategia híbrida de SAPIENZ, mostrada en el Algoritmo 2, busca combinar estos dos mundos.

El procedimiento mediante el cual SAPIENZ genera la descendencia de una población se muestra en el Algoritmo 3. Este procedimiento realiza internamente operaciones de *crossover*, mutación y reproducción en cada individuo.

La variación entre individuos se logra utilizando un **operador de *crossover* uniforme** con 50 % de mezcla entre dos individuos al azar. Es decir, dados dos padres x_1 y x_2 los hijos x'_1 y x'_2 tendrán cada uno la mitad de los genes (casos de test) de un padre y la mitad del otro. Esto se puede ver ilustrado en la Figura 2.2.

La variación interna de cada individuo se obtiene con un operador de mutación más complejo. Dado que cada individuo es un *test suite* que contiene varios casos de test, el operador primero reordena aleatoriamente los casos de test y luego realiza, con probabilidad q, *crossover* de un punto entre dos casos de test vecinos. Luego un operador más fino reordena, con probabilidad q, los eventos en cada caso de test reemplazando la posición

de distintos eventos.

Notar que si bien los eventos atómicos poseen parámetros mutables, SAPIENZ decide solo mutar el orden de la ejecución de los eventos. La justificación dada es que los mutantes igual pueden llegar a ejercitar nuevas partes de la UI que no hayan sido alcanzadas por la población inicial, ya que se cambia el orden de los eventos.

El operador de reproducción simplemente elije aleatoriamente un individuo de la población y lo deja en la descendencia sin modificaciones.

SAPIENZ usa el operador de selección NSGA-II [18], que define un criterio de comparación $\prec_c$ basado en la distancia a las soluciones más cercanas. Para dos casos de test a y b, decimos que $a \prec_c b$ si y sólo si:

$$a_{rank} < b_{rank} \vee (a_{rank} = b_{rank} \wedge a_{dist} > b_{dist})$$

Dónde a_{rank} es el número de frente de Pareto en el que se encuentra el caso de test a (mientras menor sea, mejor). Esta selección favorece los casos de test con menor ranking de no-dominación y, cuando el ranking es igual, favorece aquella que es más distante a las soluciones vecinas (aquella que se encuentra en una zona de soluciones menos densa). Mediante el uso de este criterio, SAPIENZ progresivamente reemplaza las secuencias más largas con otras más cortas cuando estas son igualmente buenas. Al usar el criterio de optimalidad de Pareto, se evita sacrificar casos de test largos cuando son los únicos que encuentran una falla, o cuando son necesarios para obtener una mejor cobertura de código.

3. PREGUNTAS DE INVESTIGACIÓN

La presentación de SAPIENZ por Harman et al. [24] contiene un estudio estadístico que compara dicha herramienta contra MONKEY [22], considerada como la más popular entre los desarrolladores ANDROID, y DYNODROID [23], considerada como el estado del arte en su momento. En dicho estudio, se muestra que SAPIENZ supera a sus dos competidores con un larga significancia estadística en los 3 objetivos que ataca: cobertura de lineas, cantidad de crashes encontrados y largo promedio de los test cases que encuentran crashes.

Sin embargo, el estudio realizado no contiene un análisis pormenorizado que muestre en detalle que partes de la técnica son las que le permiten superar a sus competidores. Luego, en esta tesis nos proponemos comparar SAPIENZ, nuestro punto de referencia, con dos variantes de si misma que deshabilitan diferentes componentes.

RQ1 (Aporte del algoritmo evolutivo): Con respecto a la efectividad de SAPIENZ, ¿Cuánto aporta el hecho de utilizar un algoritmo evolutivo?

Algo importante para notar del algoritmo para generar la descendencia de una población en SAPIENZ (Algoritmo 3) es que el operador de variación no introduce material genético nuevo, sino que sólo reordena las secuencias de eventos existentes.

Además, las evaluaciones de casos de test son bastante caras, por lo que el algoritmo evolutivo no realiza muchas generaciones dentro del presupuesto de búsqueda otorgado. En el estudio de Harman et al. [24] mencionado se establece una cota máxima de 100 generaciones, por lo que la herramienta nunca realiza más de 100 generaciones en una hora (el presupuesto de búsqueda asignado). Es decir, teniendo en cuenta que cada generación utiliza 50 individuos con 5 casos de test cada uno, en una hora el algoritmo evolutivo no realiza más de 25 000 evaluaciones, lo cual es un número bastante bajo (por ejemplo, para generación de casos de test en cobertura de ramas es normal ver 100 000 evaluaciones de la función de *fitness* por cada rama [25]).

Todo esto junto es un fuerte indicio de que el algoritmo evolutivo no está aportando mucho a la herramienta. Por lo tanto, la primera variante a comparar contra SAPIENZ es una estrategia puramente aleatoria que también utiliza el componente MOTIFCORE y los genes *motif*, a la cual llamaremos **Random+MG**. Esto es, mientras haya presupuesto de búsqueda, se generan individuos aleatorios y se evalúa su función de *fitness*. Al finalizar, se devuelve el mejor individuo encontrado en todo el proceso para cada objetivo.

RQ2 (Aporte de los genes motif): Con respecto a la efectividad de SAPIENZ, ¿Cuánto aporta el hecho de utilizar genes *motif*?

Los enfoques más sencillos para automatizar el testing en ANDROID utilizan sólo eventos atómicos. Incluso con la combinación de tales eventos, la falta de conocimiento del estado y el contexto hace difícil descubrir interacciones complejas. Esta puede llegar a ser una de las razones por las cuales se encontró que muchas herramientas suelen dar peores resultados que MONKEY en el estudio realizado por Choudhary et al. [26].

En particular, la técnica SAPIENZ utiliza los llamados genes *motif* que representan acciones complejas que podría hacer el usuario (por ejemplo, llenar todos los campos de una pantalla y luego apretar un botón). Por lo tanto, la segunda variante a comparar contra

SAPIENZ es una estrategia que utiliza el mismo algoritmo evolutivo pero sin utilizar los genes *motif*, a la cual llamaremos **AE−MG**.

Esperamos de estas dos comparaciones poder determinar el aporte de los diferentes componentes innovadores de SAPIENZ, algoritmo evolutivo y genes *motif*, en su capacidad de superar a las otras técnicas mencionadas.

4. IMPLEMENTACIÓN

4.1. Sapienz

La técnica Sapienz fue implementada en el lenguaje de programación `Python` [27] utilizando el framework Deap [28], que provee varias herramientas [1] utilizadas comúnmente en algoritmos genéticos, como ser:

- Estrategias evolutivas.
- Optimización multi-objetivo (Ej. NSGA-II y *Tournament selection*).
- Paralelización de las evaluaciones.
- *Hall of Fame* de los mejores individuos que vivieron en la población.
- Algoritmo de *crossover* uniforme y de un punto.
- Algoritmo de reordenamiento al azar para mutar secuencias de eventos.

Sapienz instrumenta las aplicaciones Android y luego obtiene cobertura a nivel de linea utilizando EMMA [29], una herramienta de código abierto para medir y reportar cobertura de código Java [30].

EMMA es capaz de reportar cobertura a nivel de método, clase y paquete. Además, permite combinar la información de cobertura obtenida en diferentes corridas, lo cual se utiliza en Sapienz para obtener la cobertura total de un *test suite* luego de ejecutar cada uno de sus casos de test.

Para los genes atómicos, Sapienz soporta 10 tipos de eventos distintos, definidos por Android: `Touch`, `Motion`, `Rotation`, `Trackball`, `PinchZoom`, `Flip`, `Nav` (tecla de navegación), `MajorNav`, `AppSwitch`, `SysOp` (operaciones del sistema tales como "silenciar volumen" y "terminar llamada").

Con respecto a los genes *motif*, Sapienz provee un único patrón que de manera aleatoria ingresa las cadenas de texto obtenidas del código fuente en los campos de texto que hubiera y luego ejercita otros elementos clickeables en la misma pantalla.

En la Figura 4.1 puede verse un caso de test típico generado por Sapienz.

4.2. Estrategia puramente aleatoria con genes *motif*

Esta variante de la herramienta consiste en reemplazar el algoritmo evolutivo utilizado por una estrategia puramente aleatoria. Más precisamente, se mantiene el tamaño de población utilizando en Sapienz pero en cada generación todos los individuos son generados aleatoriamente, de la misma forma que se genera la población inicial en el algoritmo evolutivo.

Luego de generar una población entera, se evalúa la función de *fitness* para cada individuo, y se compara cada componente de esta función con el mejor histórico. Si se

[1] Listado completo de los operadores que provee Deap: `http://deap.readthedocs.io/en/master/api/tools.html`

```
type= raw events
count= -1
speed= 1.0
start data >>
LaunchActivity(arity.calculator,calculator.Calculator)
DispatchKey(461030,461030,0,20,0,0,-1,0)
DispatchKey(461135,461135,1,20,0,0,-1,0)
DispatchTrackball(-1,461574,2,-2.0,4.0,0.0,0.0,0,1.0,1.0,0,0)
DispatchTrackball(-1,461581,2,-4.0,0.0,0.0,0.0,0,1.0,1.0,0,0)
DispatchTrackball(-1,461583,2,3.0,3.0,0.0,0.0,0,1.0,1.0,0,0)
GUIGen(1)
DispatchPointer
   (466385,466385,0,657.0,687.0,0.0,0.0,0,1.0,1.0,0,0)
DispatchPointer
   (466385,466388,1,655.16595,687.2329,0.0,0.0,0,1.0,1.0,0,0)
DispatchPointer
   (466394,466394,0,519.0,209.0,0.0,0.0,0,1.0,1.0,0,0)
...
```

Figura 4.1: Caso de test de típico generado por SAPIENZ

encuentra un individuo que haya superado el mejor histórico de algún objetivo se lo guarda y se actualiza el histórico.

4.3. Estrategia algoritmo evolutivo sin genes *motif*

Esta variante de la herramienta consiste en modificar los casos de test creados por el componente MOTIFCORE para que no tengan genes *motif*. Dichos genes se expresan en el caso de test con el evento `GuiGen`, por lo que alcanza con remover dichas lineas.

4.4. Experimentación

El framework de experimentación fue implementado en el lenguaje de programación `Python` [27]. Algunos de los problemas técnicos que tuvimos que resolver fueron:

Inestabilidad de los dispositivos. En ocasiones, y probablemente debido al uso intensivo durante la experimentación, múltiples componentes del sistema operativo de los dispositivos empiezan a crashear. Por ejemplo: *Google Play Services, NetworkLocation, Contacts, Messaging, Simple Alarm Clock, Email, android.app.keyboard, etc.* Muchos de estos componentes son esenciales para el funcionamiento del dispositivo, por lo que la evaluación de los casos de test en un dispositivo en este estado es propensa a fallar.

Para poder sortear esta situación, una vez que se detecta que un dispositivo está fallando (por ejemplo, no es posible obtener el resultado de cobertura después de correr un caso de test), se lo resetea y quita momentáneamente de los dispositivos disponibles para evaluar casos de test. Una vez que pasa el tiempo de arranque y el dispositivo se estabiliza, es re-incorporado al experimento. Se observó que el tiempo de arranque hasta

que un dispositivo está completamente funcional suele ser de 2 minutos, por lo que este es el tiempo mínimo que se espera luego de reiniciarlo.

Además, de manera preventiva, se resetean todos los dispositivos antes de comenzar la experimentación.

Inestabilidad de Android Debug Bridge (ADB). Este componente es provisto por ANDROID y es el encargado de comunicar la PC con los dispositivos. Es decir, permite enviar comandos, instalar aplicaciones, obtener los datos de cobertura, etc. En particular, también se utiliza para resetear los dispositivos cuando estos empiezan a funcionar incorrectamente. Es por esta razón que si ADB funciona mal en este momento, la experimentación es incapaz de re-establecer el dispositivo afectado. Para tratar de maximizar el buen funcionamiento de ADB, se resetea el servidor local de ADB de manera preventiva antes de comenzar la experimentación.

Timeout de comandos. Para poder detectar cuando un dispositivo está fallando, por la razón que fuera, se pusieron timeouts en todos los comandos utilizados para comunicarse con los dispositivos. Además, se verifica que el código del resultado de los comandos sea exitoso.

Instalación de aplicaciones. Mencionamos que al resetear un dispositivo este puede tardar en estabilizarse. Uno de los problemas que puede traer esto es si tratamos de instalar una aplicación demasiado pronto, ya que puede pasar que el *package manager* no esté listo todavía. Para evitar esto, consideramos que un dispositivo terminó de arrancar si el *package manager* está funcionado (por ejemplo, es capaz de listar los paquetes instalados).

Evaluación en paralelo. La evaluación de casos de test es una actividad altamente paralelizable, dado que la evaluación de cada caso es independiente de las demás, y por lo tanto se pueden utilizar todos los dispositivos en paralelo. Desde el punto de vista técnico, para lograr esta paralelización en Python, se utilizaron los mecanismos que provee la biblioteca `multiprocessing`[2]. Con dicha librería es posible armar un *pool* de *threads* que van procesando los casos de test a medida que los dispositivos van terminando evaluaciones previas. Sin embargo, como cada una de estas evaluaciones puede fallar, y luego tirar una excepción, los *threads* pueden llegar a morir, y el *thread* padre se queda esperando para siempre a que termine la evaluación. Para resolver esto, tuvimos que envolver la función de entrada de estos *threads* en un `try-catch` que maneje las fallas de forma que no muera el *thread* hijo.

Obtención de resultado de cobertura. Dado que la experimentación se realizó utilizando dispositivos Samsung Galaxy S3 con ANDROID versión 4.4 (KitKat), hubo que resolver algunos problemas con respecto a dónde se guarda, en el dispositivo, la información parcial de cobertura generada por la librería EMMA.

Las versiones de ANDROID anteriores a KitKat exponían un único volumen de almacenamiento externo a las aplicaciones. Este volumen podía ser una tarjeta SD removible, o simplemente alguna ubicación en el disco *flash* interno. El acceso de lectura/escritura

[2] `https://docs.python.org/library/multiprocessing.html`

completo a cualquier ubicación dentro de este volumen estaba protegido por un solo permiso llamado `WRITE_EXTERNAL_STORAGE`. El acceso de lectura no requería ningún permiso especial.

Hay dos cambios principales en KitKat: el acceso de lectura ahora requiere el permiso `READ_EXTERNAL_STORAGE` (para aplicaciones que no tenían ya activado el permiso de escritura), y los datos de los directorios especiales de una aplicación en el almacenamiento externo (por ejemplo /Android/data/[PACKAGE_NAME]) no requieren ningún permiso para la aplicación que es dueña de esos archivos en cuestión.

En KitKat, la API de almacenamiento externo fue dividida para incluir múltiples volúmenes: uno "primario" y otro "secundario". El volumen primario es, para todos los efectos, exactamente lo mismo que antes era el volumen único. Todas las APIs que existían antes que KitKat hacen referencia al almacenamiento externo primario. El volumen (o volúmenes) secundario modifica los permisos de escritura un poco. Se pueden leer globalmente utilizando el permiso mencionado anteriormente. Los directorios fuera del directorio especial de la aplicación (/Android/data/[PACKAGE_NAME]) no son posibles de escribir bajo ningún concepto por la aplicación en cuestión.

Por esta razón, fue necesario cambiar la instrumentación de los sujetos de prueba para que guarden el resultado de la cobertura en la carpeta especial mencionada. A raíz de esto, otro cambio fue necesario. Para obtener una evaluación de casos de test consistente, los datos de la aplicación se limpian (utilizando el comando `adb pm clear [package_name]`) antes de cada evaluación. Pero, por lo mencionado anteriormente, los datos de cobertura parcial se guardan en esta carpeta especial. Debido a esto, es necesario resguardar este resultado de cobertura parcial en la tarjeta SD (por ejemplo, en la ruta /mnt/sdcard) antes da cada limpieza, y restaurarlos antes de cada evaluación.

Comandos Unix. Si bien todos los dispositivos ANDROID corren arriba de un sistema Linux, algunos de ellos tienen un conjunto reducido de los comandos habituales que se pueden encontrar en sistemas Unix. En particular, los dispositivos Samsung Galaxy S3 utilizados no cuentan con comandos como `cp` o `awk`, por lo que fue necesario arreglar esta situación utilizando otros comandos (por ejemplo, `cp src dest` equivale a `cat src > dest`).

Casos de test sin genes motif. Para quitar los genes *motif* de los casos de test se utilizó el comando `sed` que permite eliminar lineas de un archivo.

Batería de los dispositivos. A medida que pasa el tiempo durante la experimentación, es posible que los dispositivos vayan decrementando su nivel de batería. Para que las evaluaciones sean consistentes, y buscando evitar evaluaciones cuando el dispositivo se encuentra en un estado de "ahorro de energía", se monitorea los niveles de batería. Una repetición de la experimentación solo comienza si todos los dispositivos a utilizar tienen un porcentaje de batería mayor a 75 %. Además, para ayudar a prevenir la descarga, todos los dispositivos estuvieron conectados a alimentación de energía durante la experimentación.

5. EVALUACIÓN EMPÍRICA

Esta sección describe el estudio empírico realizado para comparar las distintas variantes de SAPIENZ.

5.1. Elección de parámetros

Tamaño de población: Cada generación del algoritmo evolutivo tiene una población de tamaño fijo: 50 individuos (*test suites*) de 5 casos de test cada uno. Este parámetro es igual al que se utiliza en SAPIENZ por defecto.

Tamaño de los casos de test: Cada caso de test tiene un largo variante: entre 20 y 500 lineas. Estas cotas son las que utiliza SAPIENZ por defecto.

Probabilidad de crossover: Los individuos del algoritmo evolutivo se recombinan con probabilidad $p_c = 0{,}7$ utilizando *crossover* uniforme. Este parámetro es igual al que se utiliza en SAPIENZ por defecto.

Tamaño de mutación: Los individuos del algoritmo evolutivo se mutan con probabilidad $p_m = 0{,}3$ utilizando *crossover* de punto fijo entre dos casos de test del individuo. Este parámetro es igual al que se utiliza en SAPIENZ por defecto.

Operador de selección: Los individuos del algoritmo evolutivo se seleccionan para pertenecer a la nueva población utilizando el operador de selección NSGA-II [18].

Criterio de terminación: Todas las variantes terminan al acabarse el tiempo máximo de búsqueda de 1 hora, o al llegar a 100 generaciones, lo primero que suceda.

5.2. Sujeto

Para la experimentación se eligió uno de los sujetos utilizados en el Estudio 2 de Harman et al. [24]. Dichos sujetos son 10 aplicaciones elegidas aleatoriamente del repositorio F-Droid [31]. El detalle de dichos sujetos se puede ver en la Tabla 5.1.

De estas 10 aplicaciones, se eligió aleatoriamente para el experimento una: **Arity**.

5.3. Procedimiento experimental

Dada la naturaleza aleatoria de las variantes a comparar, se decidió correr cada una de ellas 20 veces en el sujeto de prueba. Para cada variante y aplicación elegida, se corrió el experimento un total de una hora. Este presupuesto de búsqueda es el que se suele utilizar en la literatura, y el que se utiliza en el Estudio 2 de Harman et al. [24]. Por lo tanto, la experimentación llevó en total $1\ \textit{aplicación} \times 3\ \textit{variantes} \times 20\ \textit{repeticiones} \times 1\ \textit{hora c/u} = 60\textit{hs}$ de cómputo. Es decir, la experimentación requirió en total 2,5 días de ejecución.

Sujeto	Descripción	Ver.	Fecha	LOC
Arity	Calculadora científica	1.27	2012-02-11	2,821
BabyCare	Temporizador para cuando alimentar al bebe	1.5	2012-08-23	8,561
BookWorm	Administrador de colecciones de libros	1.0.18	2011-05-04	7,589
DroidSat	Visor de satélites	2.52	2015-01-11	15,149
FillUp	Calculador de consumo de combustible	1.7.2	2015-03-10	10,400
Hydrate	Establecer metas para la ingesta de agua	1.5	2013-12-09	2,728
JustSit	Temporizador para meditación	0.3.3	2012-07-26	728
Kanji	Reconocimiento de caracteres	1.0	2012-10-30	200,154
L9Droid	Ficción interactiva	0.6	2015-01-06	18,040
Maniana	Anotador de tareas	1.26	2013-06-28	20,263

Tabla 5.1: Sujetos de prueba utilizados en el Estudio 2 de Sapienz [24]

Figura 5.1: Rack de 9 dispositivos Samsung Galaxy S3.

En cada corrida, almacenamos la máxima cobertura obtenida, mínimo largo promedio y máxima cantidad de *crashes* alcanzados.

Para evaluar los casos de test se utilizaron 9 dispositivos Samsung Galaxy S3. Para evitar la descarga de batería durante la ejecución de los experimentos los dispositivos fueron conectados a varios Hubs USB que proveen la alimentación necesaria. Estos Hubs son a la vez conectados a una computadora con un procesador Intel i7 (de 8 cores) y 6Gb de RAM, corriendo Ubuntu 16.04. Se puede ver en la Figura 5.1 una foto de esta configuración.

Al analizar los datos obtenidos del estudio empírico seguimos las pautas para la ingeniería de software experimental presentadas por Arcuri et al. [32]. Para cada par de variantes $\mathcal{A}$, $\mathcal{B}$ y métrica objetivo, utilizamos el test no paramétrico *U de Mann-Whitney* para analizar la **hipótesis nula**: dada una corrida al azar para cada una de las variantes, es igual de probable que $\mathcal{A}$ sea mejor que $\mathcal{B}$ para el objetivo seleccionado que lo contrario. Es decir, la hipótesis nula es que no hay diferencia estadística entre la efectividad de las

dos variantes con respecto a la métrica objetivo elegida.

Por ser no paramétrico, el test *U de Mann-Whitney* no requiere como asunción que las muestras sean tomadas de una distribución normal, a diferencia de otros como el test paramétrico *t*.

En aquellos pares de variantes que muestran una diferencia estadística, utilizamos la medida de efecto no-paramétrica $\hat{A}_{12}$ de *Vargha y Delaney* para determinar la magnitud de dicha diferencia. Esta medida de efecto es sencilla de interpretar comparada con otras disponibles. Suponiendo dos variantes $\mathcal{A}$ y $\mathcal{B}$, un valor $\hat{A}_{12} = 0{,}7$ significa que uno obtendría mejores resultados 70 % del tiempo utilizando $\mathcal{A}$. Las diferencias entre variantes son caracterizadas como pequeñas, medianas y largas cuando $\hat{A}_{12}$ supera 0,56, 0,64 y 0,71 respectivamente.

La fórmula para calcular la medida de efecto $\hat{A}_{12}$ de *Vargha y Delaney* es:

$$\hat{A}_{12} = (R_1/m - (m+1)/2)/n$$

Dónde m es la cantidad de observaciones para la primer muestra de datos, y n la cantidad para la segunda. Para nuestro experimento, $m = n$, dado que corrimos la misma cantidad de veces todas las variantes. R_1 es la suma del *ranking* del primer grupo de datos bajo comparación. Por ejemplo, tomemos los datos $X = \{42, 11, 7\}$ e $Y = \{1, 20, 5\}$. Entonces, X tendría rankings $\{6, 4, 3\}$, cuya suma es 13, mientras que Y tendría rankings $\{1, 5, 2\}$. La suma de *rankings* es un componente básico del test *U de Mann-Whitney*, por lo que la mayoría de las herramientas estadísticas lo proveen. En nuestro caso, todos los resultados estadísticos se calcularon utilizando el lenguaje de programación `R`.

5.4. Resultados

La Tabla 5.2 muestra los $p-valores$ obtenidos del test *U de Mann-Whitney*. La Tabla 5.3 muestra las medidas de efecto $\hat{A}_{12}$ de *Vargha y Delaney*. Para determinar si se puede descartar o no la hipótesis nula vamos a tomar nivel de significancia $\alpha = 0{,}05$.

Las Figuras 5.2, 5.3 y 5.4 muestran los datos agregados por estrategia para cada métrica objetivo.

Porcentaje de cobertura alcanzado. Sólo es posible descartar la hipótesis nula para la comparación Random+MG vs. Sapienz, con un $p-valor$ de 0,04, y la medida de efecto $\hat{A}_{12}$ para esa misma comparación es 0,689, por lo que podemos decir que aproximadamente 69 % de las veces Random+MG encuentra casos de test con mayor cobertura que Sapienz.

Para las comparaciones AE−MG vs. Sapienz y Random+MG vs. AE−MG no es posible descartar la hipótesis nula. En particular, en el caso de Random+MG vs. AE−MG el $p-valor$ obtenido es muy cercano a 1, con lo que desde el punto de vista estadístico estas dos estrategias son prácticamente indistinguibles. Esto se ve reflejado también en la medida de efecto $\hat{A}_{12}$ obtenida para dicha comparación, que es 0,503. Es decir, la mitad de las veces es mejor utilizar una estrategia y la otra mitad es mejor utilizar la otra.

Es interesante notar que, a pesar de que Random+MG y AE−MG no puedan ser diferenciadas desde el punto estadístico, y que Random+MG obtiene en promedio mejor cobertura que Sapienz, no es posible descartar la hipótesis nula para AE−MG vs. Sapienz. Una intuición de porqué pasa esto puede verse en la Figura 5.2: la media de

Objetivo	Comparación entre estrategias		
	Random+MG vs. SAPIENZ	**AE−MG vs.** SAPIENZ	**Random+MG vs. AE−MG**
Cobertura	0.040	0.437	0.989
#Crashes	0.023	0.973	0.040
Largo	0.001	0.058	0.0000001

Tabla 5.2: $p - valores$ obtenidos del test *U de Mann-Whitney.*

Objetivo	Comparación entre estrategias		
	Random+MG vs. SAPIENZ	**AE−MG vs.** SAPIENZ	**Random+MG vs. AE−MG**
Cobertura	0.689	0.573	0.503
#Crashes	0.655	0.496	0.650
Largo	0.801	0.324	0.994

Tabla 5.3: Medida de efecto $\hat{A}_{12}$ de *Vargha y Delaney*

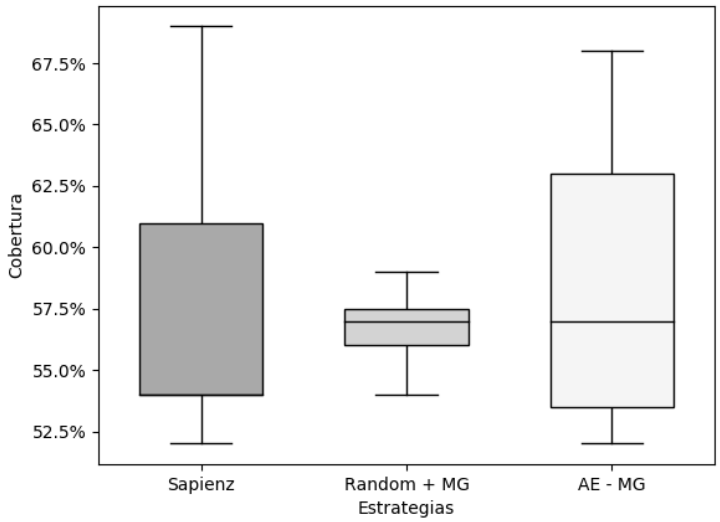

Figura 5.2: Resultados para el objetivo "cobertura de lineas"

Random+MG es más alta que la de SAPIENZ y con una varianza bastante menor, mientras que la media de AE−MG es parecida a la de Random+MG pero con una varianza mucho mayor.

Cantidad de crashes encontrados. Podemos descartar la hipótesis nula para las comparaciónes Random+MG vs. SAPIENZ, con un $p - valor$ de 0,023 y medida de efecto 0,655, y para Random+MG vs. AE−MG, con un $p - valor$ de 0,04 y medida de efecto 0,65.

No es posible descartar la hipótesis nula para la comparación AE−MG vs. SAPIENZ.

Largo promedio de los casos de test que encuentran crashes. Podemos descartar la hipótesis nula para las comparaciónes Random+MG vs. SAPIENZ, con un $p - valor$ de 0,001 y medida de efecto 0,801, y para Random+MG vs. AE−MG, con un $p - valor$ de 0,0000001

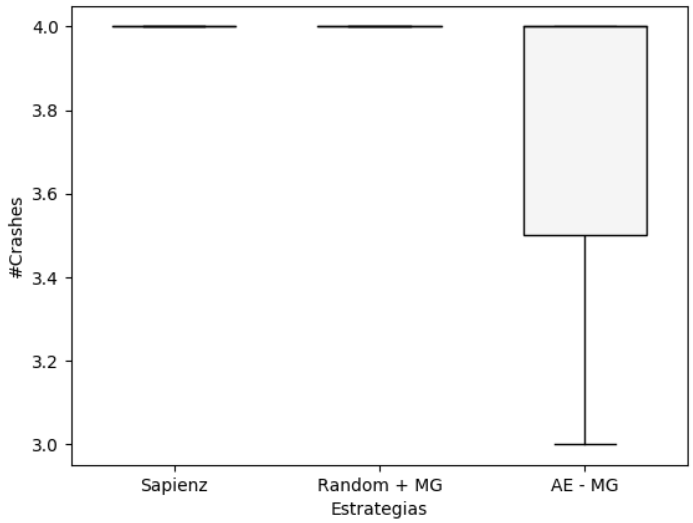

Figura 5.3: Resultados para el objetivo "cantidad de crashes"

y medida de efecto 0,994.

No es posible descartar la hipótesis nula para la comparación AE−MG vs. Sapienz, aunque el $p - valor$ obtenido (0,058) no está tan lejos del nivel de significancia $\alpha = 0{,}5$.

La diferencia significativa mencionada es probable que se deba a que Random+MG logra generar casos de test de longitud muy chica que sólo contienen genes *motif*, y que a su vez encuentran crashes. En este sentido, los genes *motif* provocan una comparación injusta, dado que son la agrupación de uno o más genes atómicos. En el estudio realizado por Harman et al. [24] los genes *motif* se deconstruyen en genes atómicos para realizar la comparación, pero no dan instrucciones en como llevar a cabo esta tarea y dicho código no se encuentra en la implementación código abierto de Sapienz.

Degradación de los dispositivos. Como se mencionó en la sección de Implementación de la Experimentación, la inestabilidad de los dispositivos hace que muchas veces la evaluación de un caso de test en uno de ellos falle. Al ocurrir esta situación el dispositivo afectado es reseteado y quitado momentáneamente de los dispositivos disponibles para evaluar casos de test.

Por esta razón, definimos una medida de *degradación* que indica el porcentaje de evaluaciones fallidas en un dispositivo. Es decir, si un dispositivo tuvo degradación 5 % en una corrida, significa que falló en evaluar $\frac{5*n}{100}$ casos de test, con n la cantidad total de casos de test que fue dado a evaluar.

Para mostrar que la comparación entre las distintas estrategias es justa, mostramos en la Figura 5.5 un boxplot agrupando por cada una la degradación obtenida .

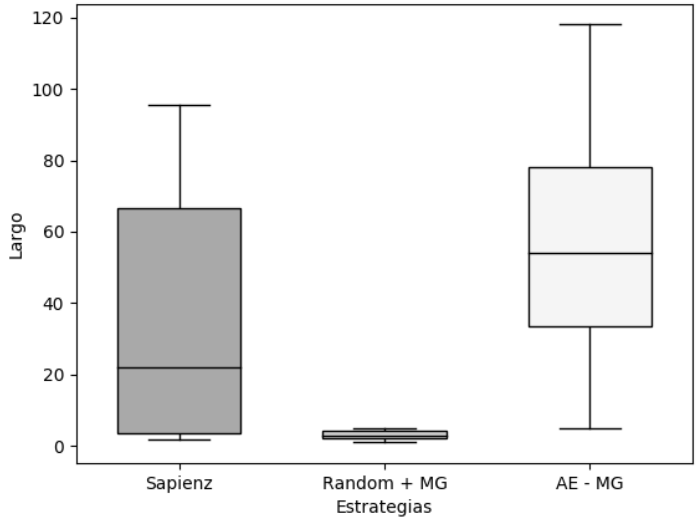

Figura 5.4: Resultados para el objetivo "largo promedio de casos de test que encuentran crashes"

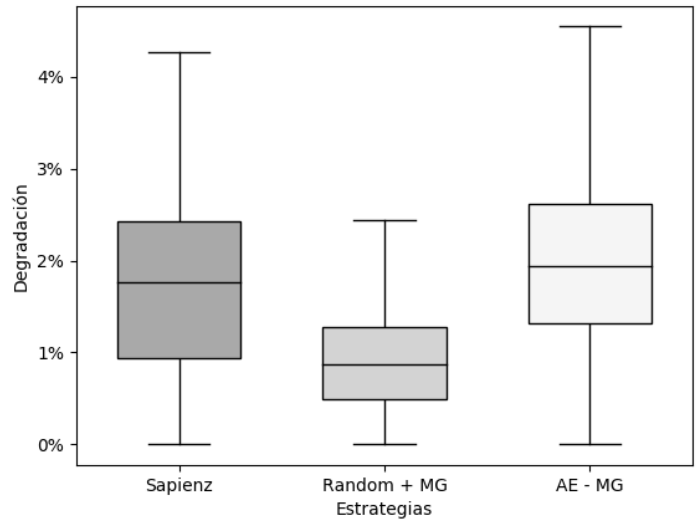

Figura 5.5: Degradación de las evaluaciones en los dispositivos

6. CONCLUSIONES

It is good to have an end to journey towards;
but it is the journey that matters, in the end.
– Ursula K. Le Guin, The Left Hand of Darkness.

En base a los resultados obtenidos en el estudio empírico, responderemos las preguntas de investigación planteadas anteriormente.

RQ1 (Aporte del algoritmo evolutivo): Con respecto a la efectividad de SAPIENZ, ¿Qué tanto aporta el hecho de utilizar un algoritmo evolutivo?

En el estudio empírico realizado, para todas las métricas objetivo, logramos mostrar con significancia estadística ($\alpha = 0{,}5$) que existe diferencia entre la efectividad de Random+MG y la SAPIENZ. Además, la medida de efecto $\hat{A}_{12}$ nos permite afirmar que esta diferencia se inclina favorablemente hacía el lado de Random+MG.

Por esta razón, podemos afirmar que el algoritmo evolutivo que utiliza SAPIENZ no sólo no aporta sino que resta cuando lo comparamos con un enfoque puramente aleatorio mucho más sencillo. Las posibles causas de esto, cómo se mencionó previamente, parecen ser: poca cantidad total de evaluaciones realizadas durante el tiempo de búsqueda y un operador de mutación que no introduce nuevo material genético.

RQ2 (Aporte de los genes motif): Con respecto a la efectividad de SAPIENZ, ¿Qué tanto aporta el hecho de utilizar genes *motif*?

En el estudio empírico realizado, no logramos ver ninguna diferencia significativa entre AE−MG y SAPIENZ. Si logramos ver diferencia entre Random+MG y AE−MG para las métricas de cantidad de crashes y largo promedio de los casos de test que encuentran crashes, pero esto es muy probable que se deba a que la estrategia aleatoria es más eficiente que el algoritmo evolutivo.

Esto quiere decir que, al menos para el sujeto de prueba utilizado, los genes *motif* no están aportando una efectividad que les permita destacar a SAPIENZ de la variante AE−MG.

6.1. Peligros de validez

Como en cualquier estudio empírico, hay factores que pueden llegar a afectar la validez de los resultados experimentales.

Peligros de validez interna Los peligros de validez interna son aquellos factores en la metodología experimental que pueden afectar los resultados.

Para el estudio realizado, se seleccionó una aplicación de las 10 utilizadas en el experimento realizado por Harman et al. [24], lo cual puede haber resultado en un sesgo en la selección. Para mitigar este problema, la selección se realizó de manera aleatoria.

Con respecto a la implementación de SAPIENZ y las otras estrategias, se utilizó como base el código público disponible al momento. Dicha implementación utiliza un sólo gen

motif que llena todos los campos de texto en la pantalla y luego acciona los elementos clickeables. La capacidad de SAPIENZ puede que mejore considerando otros genes *motif*, pero no podría ser peor, dado que el gen *motif* actual siempre estaría disponible.

La elección de parámetros para cada una de las estrategias puede afectar su capacidad significativamente. Para reducir este peligro, utilizamos la configuración por defecto que provee SAPIENZ sin realizar ningún tipo de ajuste de parámetros.

Peligros de validez externa Los peligros de validez externa aparecen cuando los resultados experimentales no pueden ser generalizados. Debido a restricciones de tiempo, estuvimos limitados en la cantidad de sujetos de prueba a los cuales pudimos aplicar las diferentes variantes.

Los resultados presentados puede que no generalicen más allá de la aplicación utilizada. En particular, aplicaciones que contengan mayor cantidad de pantallas con campos de texto pueden verse favorecidas por el gen *motif* implementado. Además, la experimentación se realizó sobre una sola versión de la plataforma ANDROID.

6.2. Trabajo futuro

Como trabajo futuro, creemos que hay varias preguntas que quedan abiertas y que son interesantes de investigar:

- Los resultados presentados, ¿Se mantienen o cambian al aumentar la cantidad de repeticiones utilizadas en el estudio empírico?
- Los resultados presentados, ¿Son generalizables a otras aplicaciones?
- ¿Es posible sacar alguna conclusión con respecto a la *eficiencia* (por ejemplo, cobertura alcanzada por unidad de tiempo) de las diferentes estrategias presentadas?
- Con respecto al algoritmo evolutivo utilizado por SAPIENZ, ¿Cambia la efectividad de la herramienta si cambiamos el algoritmo actual $(\mu + \lambda)$ por otro ?

Bibliografía

[1] Google announces over 2 billion monthly active devices on android - the verge. https://www.theverge.com/2017/5/17/15654454/android-reaches-2-billion-monthly-active-users. (Accessed on 05/01/2018).

[2] Smartphone os share installed base worldwide 2015-2017 — statistic. https://www.statista.com/statistics/385001/smartphone-worldwide-installed-base-operating-systems/. (Accessed on 05/01/2018).

[3] Number of available android applications - appbrain. http://www.appbrain.com/stats/number-of-android-apps. (Accessed on 05/03/2018).

[4] Boris Beizer. *Software testing techniques (2. ed.)*. Van Nostrand Reinhold, 1990.

[5] Edsger W. Dijkstra. In *Software Engineering Techniques*, page 16. NATO Science Committee, 1969.

[6] Jon Edvardsson. A survey on automatic test data generation.

[7] Ting Chen, Xiao-song Zhang, Shi-ze Guo, Hong-yuan Li, and Yue Wu. State of the art: Dynamic symbolic execution for automated test generation. *Future Generation Computer Systems*, 29(7):1758–1773, September 2013.

[8] Phil McMinn. Search-based software test data generation: a survey. *Software Testing, Verification and Reliability*, 14(2):105–156.

[9] Glenford J. Myers and Corey Sandler. *The Art of Software Testing*. John Wiley & Sons, Inc., USA, 2004.

[10] Gordon Fraser and Andrea Arcuri. Whole test suite generation. *IEEE Trans. Software Eng.*, 39(2):276–291, 2013.

[11] James E. Baker. Adaptive selection methods for genetic algorithms. In *Proceedings of the 1st International Conference on Genetic Algorithms*, pages 101–111, Hillsdale, NJ, USA, 1985. L. Erlbaum Associates Inc.

[12] G. Pavai and T. V. Geetha. A Survey on Crossover Operators. *ACM Comput. Surv.*, 49(4):72:1–72:43, December 2016.

[13] A. Arcuri. It does matter how you normalise the branch distance in search based software testing. In *2010 Third International Conference on Software Testing, Verification and Validation*, pages 205–214, April 2010.

[14] Allen Goldberg, T. C. Wang, and David Zimmerman. Applications of feasible path analysis to program testing. In *Proceedings of the 1994 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '94, pages 80–94, New York, NY, USA, 1994. ACM.

[15] José Campos, Yan Ge, Gordon Fraser, Marcelo Eler, and Andrea Arcuri. An empirical evaluation of evolutionary algorithms for test suite generation. In Tim Menzies and Justyna Petke, editors, *Search Based Software Engineering - 9th International Symposium, SSBSE 2017, Paderborn, Germany, September 9-11, 2017, Proceedings*, volume 10452 of *Lecture Notes in Computer Science*, pages 33–48. Springer, 2017.

[16] Benjamin Doerr, Carola Doerr, and Franziska Ebel. From black-box complexity to designing new genetic algorithms. *Theor. Comput. Sci.*, 567:87–104, 2015.

[17] Carlos M. Fonseca and Peter J. Fleming. Genetic algorithms for multiobjective optimization: Formulation, discussion and generalization. In *Proceedings of the 5th International Conference on Genetic Algorithms*, pages 416–423, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

[18] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, Apr 2002.

[19] Android fragmentation report august 2015 - opensignal. `https://opensignal.com/reports/2015/08/android-fragmentation/`. (Accessed on 05/03/2018).

[20] M. E. Joorabchi, A. Mesbah, and P. Kruchten. Real challenges in mobile app development. In *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 15–24, Oct 2013.

[21] P. S. Kochhar, F. Thung, N. Nagappan, T. Zimmermann, and D. Lo. Understanding the test automation culture of app developers. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–10, April 2015.

[22] Ui/application exerciser monkey. `https://developer.android.com/studio/test/monkey.html`. (Accessed on 05/03/2018).

[23] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. Dynodroid: an input generation system for android apps. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, pages 224–234, 2013.

[24] Ke Mao, Mark Harman, and Yue Jia. Sapienz: multi-objective automated testing for android applications. In *International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany*, pages 94–105, 2016.

[25] M. Harman and P. McMinn. A theoretical and empirical study of search-based testing: Local, global, and hybrid search. *IEEE Transactions on Software Engineering*, 36(2):226–247, March 2010.

[26] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. Automated test input generation for android: Are we there yet? (E). In *International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA*, pages 429–440, 2015.

[27] Python programming language. `https://www.python.org/`. (Accessed on 05/08/2018).

[28] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner, Marc Parizeau, and Christian Gagné. DEAP: Evolutionary algorithms made easy. *Journal of Machine Learning Research*, 13:2171–2175, jul 2012.

[29] Emma: a free java code coverage tool. `http://emma.sourceforge.net/`. (Accessed on 05/05/2018).

[30] Java programming language. `https://docs.oracle.com/javase/8/docs/technotes/guides/language/index.html`. (Accessed on 05/08/2018).

[31] F-droid - free and open source android app repository. `https://f-droid.org/en/`. (Accessed on 05/06/2018).

[32] Andrea Arcuri and Lionel C. Briand. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Softw. Test., Verif. Reliab.*, 24(3):219–250, 2014.