# Reparación de pruebas de interfaz de usuario en Android como un problema de búsqueda

Adrián Cantón Fernandez, José Antonio Parejo[0000−0002−4708−4606], Sergio Segura Rueda, and Antonio Ruiz-Cortés

University of Sevilla, Spain japarejo@us.es

**Abstract.** Las pruebas de interfaz de usuario son una técnica muy popular gracias a su capacidad para validar el comportamiento de la aplicación tal y como lo experimentaría el usuario, y por su facilidad para generar los casos de prueba. Sin embargo, una de las limitaciones más importantes de este tipo de pruebas es su fragilidad ante los cambios de la propia interfaz de usuario, que suelen producirse durante el desarrollo del sistema. En este artículo formulamos la reparación de estas pruebas ante cambios en la intefaz o funcionalidad de la aplicación como un problema de búsqueda. Además, proponemos un algoritmo heurístico para su resolución basado en GRASP. Esta propuesta se ha implementado y validado en el dominio especifico de aplicaciones móviles para dispositivos Android. Los resultados obtenidos demuestran su aplicabilidad con varios casos de estudio para cambios de diversa envergadura.

**Keywords:** e2e testing · GRASP · test repair

## 1 Introducción

Las interfaces gráficas de usuario (GUI, de *Graphical User Interface*) son una parte extremadamente importante de los sistemas software actuales. Para el usuario final ésta es el principal mecanismo interacción con las aplicaciones, hasta el punto que algunos autores afirman que "la interfaz es la aplicación para el usuario" [2]. Para asegurar el correcto funcionamiento del sistema y su interfaz es necesario realizar pruebas de interfaz, que ejecutan acciones sobre la GUI tal y como lo haría un usuario final y evalúan sus resultados. Este tipo de pruebas se han vuelto muy populares [9, 3], porque permiten expresar de manera natural como tests las historias de usuario/casos de uso, y porque la proliferación de herramientas para la generación de este tipo de pruebas (como por ejemplo Selenium [4]) ha simplificado en buena medida su desarrollo. Sin embargo, este tipo de tests tiene aún hoy en día un talón de Aquiles importante: son extremadamente fŕagiles, es decir, es muy fácil romper los tests previamente desarrollados (haciendo que no puedan llegar a ejecutarse, ni validarse) cuando cambiamos la propia interfaz de usuario. Por ejemplo, cambiar la localización del botón o su identificador interno, puede romper cualquier caso de prueba de interfaz que lo use. Dado que las metodologías ágiles promueven aceptar cambios a lo largo de todo el ciclo de vida del producto, y que la interfaz de usuario es el elemento

con el que más contacto tienen los usuarios, y por tanto el que más tienden a cambiar para casos de uso/historias de usuario ya implementadas, este escenario de rotura de los tests pre-existentes es muy común y frustrante. Además, el coste de reparar manualmente uno de estos tests es en muchos casos similar al de desarrollarlo desde 0 con las herramientas actuales. En este artículo presentamos una formulación de la reparación de tests de interfaz de usuario como un problema de búsqueda, y proponemos un algoritmo de búsqueda simple para su resolución basado en GRASP. La propuesta, ha sido implementada y validada en el escenario concreto del desarrollo de aplicaciones móviles en dispositivos Android. Esta validación empírica ha demostrado que el algoritmo propuesto es capaz de reparar tests con cambios no triviales en la secuencia de acciones a realizar. Además, se ha comprobado que para los tests usados en la validación, el algoritmo propuesto es más efectivo que un algoritmo alternativo basado en búsqueda aleatoria.

El resto del artículo se estructura tal y como se describe a continuación. En la siguiente sección de describe el trabajo relacionado. La sección 3 describe la formulación del problema de la reparación de casos de prueba como un problema de búsqueda. La sección 4 describe nuestra propuesta de algortimo para la reparación casos de prueba de interfaz de usuario basado en *GRASP*. La sección 5 describe la validación empírica realizada del funcionamiento del algoritmo propuesto. Finalmente, las sección 6 describe las conclusiones extraidas y el trabajo futuro a realizar.

## 2 Trabajo relacionado

El problema de generación de casos de prueba de interfaz de usuario es ampliamente conocido y ha sido abordado en la literatura usando gran variedad de estrategias diferentes. En [1] se presenta una revisión sistemática de la literatura reciente para el caso específico de aplicaciones móviles. La reparación de casos es un problema menos popular, pero que también ha sido abordado por diversos autores usando distintas estrategias. En [10] un conjunto de transformaciones para la reparación de casos de prueba de interfaz fue definido y aplicado para reparar test en aplicaciones de escritorio. Más allá de la identificación de patrones de error y transformaciones de reparación, podemos clasificar las propuestas concretas de algoritmos de reparación en base a dos criterios: i) dependiendo de si se requiere la intervención de un humano para la reparación, tendremos estrategias automáticas o semi-automaticas; ii) dependiendo de si las propuestas hacen uso o no de un modelo de la interfaz de usuario de la aplicación que les permita agilizar el proceso de toma de decisiones respecto de las acciones posibles dado un estado concreto de la interfaz de la aplicación. Una propuesta semi-automática para la reparación de casos de prueba en escritorio basada en modelos de interfaz es SITAR [7] Otra propuesta destacable basada en un modelo de GUI para la reparación automatica de casos de prueba en aplicaciones Android es ATOM [8]. También existen propuestas híbridas que combinan la generación estocásticas de pruebas con el uso de modelos para guiar su modificación, como [11]. Las prop-

uestas del presente artículo son algoritmos automáticos no basados en un modelo de interfaz.

## 3 Formulación del problema

Para ilustrar los conceptos descritos en esta sección usaremos un ejemplo basado en una aplicación móvil de gestión de notas. En esta aplicación el usuario tiene la opción de crear, borrar, modificar y consultar notas que únicamente contienen un texto. Esta aplicación ha sido implementada como parte de la sección de validación. Un conjunto de capturas de la interfaz de usuario que ilustra un caso de uso de creación de una nota se muestra en la Figura 1.

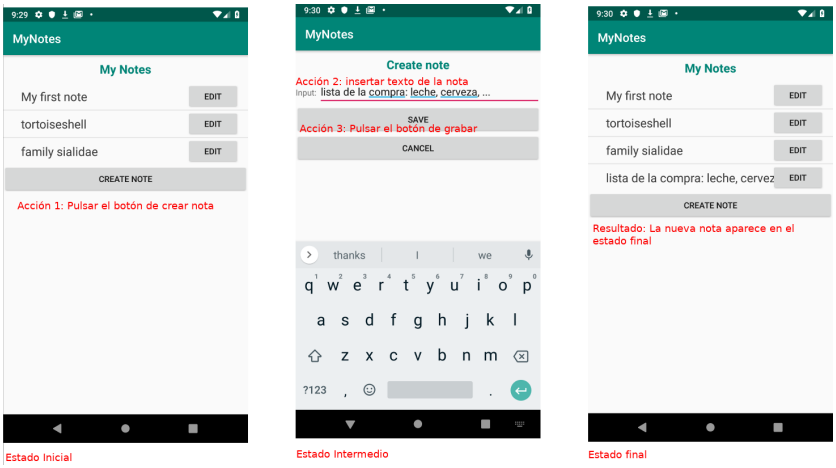


**Fig. 1.** Acciones y estados del caso de prueba de creación de notas

### 3.1 Definiciones básicas

Para nuestros propósitos asumiremos que una aplicación se compone de un conjunto de estados posibles (a nivel de su interfaz de usuario) $S = \{s_1, \ldots, s_n\}$, con un estado inicial $s_0 \in S$ que se muestra al arrancar la aplicación. En cada estado $x \in S$ de la aplicación tendremos disponbiles un conjunto de acciones distintas posibles dadas for la función $acc(x) = \{a_1^x, \ldots, a_m^x\}$. Llamaremos $A^*$ al conjunto unión de todas las acciones posibles sobre cualquier estado de la aplicación.

Cada acción $a_i$ tiene asociada una variable $v_i$, que define un valor asociado a la acción. Por ejemplo, para una acción de tipo texto, que supone insertar un texto concreto en un campo de texto de la interfaz, el valor de la variable será el texto insertado. Para las acciones de tipo botón de radio o campo de selección (que definen un conjunto de valores posibles de entre los que se puede seleccionar

uno), la variable contendrá el valor concreto seleccionado. Para las variables de tipo botón, la variable contendrá el nombre del botón. Definimos la función $v(a_i)$ que nos permitirá obtener dado una acción $a_i$ el valor de su variable asociada.

Así mismo definimos $perf(x, a) : S \times A^* \longrightarrow S \cup \epsilon$ como una función que dada un estado actual de la aplicación, y una acción nos devuelve bien otro estado de la aplicación $y \in S$ o un estado especial $\epsilon$. Si la acción proporcionada como parámetro está dentro del conjunto de acciones posibles del estado proporcionado como parámetro, el estado obtenido como resultado será un elemento de $S$, es decir la acción podrá realizarse sobre el estado de la interfaz. En caso contrario el resultado devuelto por la función será $\epsilon$ que representa un estado de error en el que hemos intentado realizar una acción que no era posible dado el estado actual de la interfaz. Por definición, el conjunto de acciones disponibles para el estado de error es vacío, $acc(\epsilon) = \{\}$.

Definiremos un caso de prueba de interfaz de usuario para una aplicación como una tupla compuesta por una secuencia de $n$ acciones $A =< a_1, a_2, \ldots, a_n >$, y un predicado $p$ definido sobre el conjunto de variables asociadas a las acciones en $A$, y dos conjuntos de valores especiales $init$ y $finish$, que contienen todo el conjunto de elementos textuales que muestra la interfaz de la aplicación en los estados inicial y final de la aplicación tras la ejecución del caso de prueba.

Por ejemplo para el caso de prueba de creación de una notas descrito en la Figura 1, podríamos describir la secuencia de acciones del caso de prueba como:

1. BUTTON("buttonCreate","CREATE")
2. TEXT("input1","lista de la compra: leche, cerveza, ...")
3. BUTTON("buttonSave","SAVE")

El predicado del caso de prueba, que en lenguaje natual sería: *al terminar el caso de prueba se muestran mas notas que al comenzar* y *el texto introducido en el text box en la acción 2 (lista de la compra ...), aparece en la pantalla al terminar el caso de prueba*. Este predicado se puede expresar en nuestra formalización como[1]: $|init| < |finish| \wedge v(a_2) \in finish$.

Definimos la función $state(t, i)$ donde $t = \{A =< a_1, \ldots, a_n >, p\}$ es us un entero positivo menor o igual a la longitud de acciones del test cómo:

$$state(t,i) = \begin{cases} S_0 & si \quad i < 1 \\ perf(a_i, state(t, i-1)) & si\ 1 \leq i \leq |A| \\ state(t, i-1) & si \quad i \geq |A| \end{cases} \tag{1}$$

Es decir, state(t,i) nos devuelve el estado al que llega la aplicación tras ejectuar las $i$ primeras acciones del caso de prueba de interfaz de usuario.

Diremos que un caso de prueba de interfaz de usuario $t = \{A, p\}$ es inválido cuando existe al menos un $i$, tal que $0 < i \leq |A|$ y $state(t, i) = \epsilon$. Es decir cuando

[1] Una versión usable de este caso de prueba, usada en nuestra validación, esta disponible en el repositorio de código del artículo.

la ejecución de la secuencia de sus acciones nos lleva al estado de error. Al valor $i$ mínimo tal que $state(t, i) = \epsilon$ se le llama índice invalidación y a la acción del caso de prueba se la llama la *acción inválido*.

Diremos que un caso de prueba de interfaz de usuario detecta un fallo puede ejecutarse completo (no es inválido), pero el predicado $p$ no se cumple para la versión actual de la aplicación en desarrollo.

El problema de reparación de pruebas de interfaz de usuario consiste en, dado un caso de prueba inválido $t = \{A =< a_1, a_2, \ldots, a_n >, p\}$ con índice de invalidez $i$, encontrar una secuencia de acciones $< a'_1, \ldots, a'_m >$ tal que el caso de prueba $t' = \{< a_1, \ldots, a_i, a'_1, \ldots, a'_m >, p\}$ es válido y $p$ se cumple tras ejecutar $A$.

### 3.2 Función objetivo

Para formular el problema de reparación de pruebas de interfaz de usuario como un problema de búsqueda es necesario caracterizar el espacio de búsqueda y la función objetivo a optimizar.

La función más simple posible a optimizar sería:

$$f^{obj}_{falaz}(t) = \begin{cases} 1 \; si \text{ p se cumple tras ejecutar} A \\ \\ 0 \qquad\quad \text{en caso contrario} \end{cases} \tag{2}$$

Sin embargo, esta función es extremadamente falaz (o deceptiva), puesto que no proporciona información alguna de cuan lejos o cerca está la secuencia de acciones de hacer que se cumpla el predicado. Una función objetivo $f^{obj}_{pred}$ más útil desde la perspectiva de la definición de un algoritmo de búsqueda puede definirse si asumimos que el predicado se expresa como un conjunto de cláusulas en forma normal conjuntiva, y hacemos que esta función devuelva el número de clausulas que se cumplen.

### 3.3 Espacio de soluciones y estructuras de vecindario

El espacio de soluciones del problema viene dado por todo el conjunto de casos de prueba de interfaz de usuario tales que: i) no son inválidos, ii) $p$ es el predicado original del test a reparar, y iii) su conjunto de acciones es una permutación de un elemento del conjunto potencia de $A^*$, $P(A^*)$.

Podemos definir además dos estructuras de vecindario que serán útiles de cara a la creación de algoritmos de búsqueda para este problema.

**Vecindario por adición** Las soluciones vecinas por adición a una solución $t$ del problema de reparación de casos de prueba de interfaz de usuario son aquellas soluciones no inválidas que se obtienen al añadir una acción en cualquier posición. Formalmente, para $t = \{A =< a_1, a_2, \ldots, a_n >, p\}$, el conjunto de vecinas por adición se define como la unión para cada $i$ de las vecinas por adición en la posición $i$. Para cada $i$ tal que $0 \leq i \leq n$, éstas se definen como las $t' = \{A =<$

$a_1, \ldots, a_i, x, a_i + 1, \ldots, a_n >, p\}$, tales que: i) $x \in acc(state(t', i))$, es decir $x$ es una acción posible en el estado en que se ejecuta, y ii) $a_{i+1} \in acc(state(t', i+1))$ si $i < n$, es decir, si $x$ no es la última acción del caso de prueba de interfaz entonces la siguiente acción a esta debe ser una acción posible del estado al que nos lleva la ejecución de $x$. Si se cumplen estas dos condiciones el caso de prueba $t'$ no será inválido por definición.

**Vecindario por sustracción** Las soluciones vecinas por sustracción a una solución $t$ del problema de reparación de casos de prueba de interfaz de usuario son aquellas soluciones no inválidas que se obtienen al eliminar una acción en cualquier posición. Formalmente, para $t = \{A =< a_1, a_2, \ldots, a_n >, p\}$, el conjunto de vecinas por sustracción se define como la unión para cada $i$ de las vecinas por sustracción en la posición $i$. Para cada $i$ tal que $1 \leq i \leq n$, éstas se definen como las $t' = \{A =< a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n >, p\}$, tales que: i) si $i < n$ entonces $a_{i+1} \in acc(state(t', i-1))$, es decir $a_{i+1}$ es una acción posible en el estado en que se ejecuta, y ii) si $i = 1$, entonces $a_{i+1} \in acc(S_0)$, es decir, si la accion eliminada era la primera, $a_{i+1}$ debe ser una acción posible en el estado inicial. Si se cumplen estas dos condiciones, el caso de prueba $t'$ no será inválido por definición.

## 4 Algoritmos para la reparación de casos de prueba de interfaz de usuario

En esta sección se definen 3 algoritmos para la reparación de casos de prueba de interfaz de usuario.

### 4.1 Algoritmo de reparación aditiva aleatorio desde cero

Dada la estructura de vecindario por adición definida en la sección anterior. Podemos definir un algoritmo de repararción como aquel que durante $maxIterations$ iteraciones, construye iterativamente un caso de prueba de $maxTestSize$ acciones, partiendo de una lista de acciones vacía, y añadiendo en cada iteración una acción posible aleatoria, dado el estado actual de la aplicación (se parte del estado inicial $S_0$). Cuando el número de acciones del caso de prueba alcanza un tamaño mínimo $minTestSize$, se realiza una evaluación del predicado del test tras realizar cada acción, para evaluar si hemos reparado o no el caso de prueba. El pseudocódigo del algoritmo está especificado en el listado 1.

### 4.2 Algoritmo de reparación aditiva aleatorio tras invalidez

El algoritmo anterior no utiliza en absoluto la información contenida en el caso de prueba original más allá del predicado a evaluar, sino que genera todas las acciones de cada una de las soluciones generadas de manera aleatoria de entre las siguientes acciones posibles hasta tener casos de prueba con un número máximo

**Algorithm 1** Algoritmo de reparación aleatorio desde cero

```
i ← 0
while i < maxIterations do
    testActions ← {}
    s ← S_0
    init ← currentState()
    availableActions ← acc(s)
    while |testActions| < maxTestSize do
        action ← chooseRandom(availableActions)
        s ← perf(action)
        if |testActions| < minTestSize then
            finish ← currentState()
            if eval(p, testActions, init, finish) then
                return testActions
            end if
        end if
        availableActions ← acc(s)
    end while
    i ← i + 1
end while
return {}
```

de acciones. Podemos modificar dicho algoritmo para usar el conjunto de acciones posibles presentes en el test original hasta el índice de invalidez $< a_1, \ldots, a_{i-1} >$, completando las acciones restantes desde $i$ hasta el máximo con acciones posibles aleatorias sucesivas. El pseudocódigo del algoritmo que surge como resultado de las modificaciones descritas se muestra en el listado 2. El conjunto de acciones posibles presentes en el test original hasta el índice de invalidez $< a_1, \ldots, a_{i-1} >$, se denota en el pseudocódigo como $previousValidActions$.

### 4.3 Algoritmo de reparación basado en GRASP (GRASPTeR)

Los algoritmos definidos previamente no aprovechan el conocimiento que poseemos respecto de los pasos del test que funcionaban en la versión anterior de la aplicación. Con el objetivo aprovechar dicho conocimiento y de adaptarnos mejor al desarrollo incremental de aplicaciones que promueven las metodologías actuales, en las que se introducen cambios más pequeños pero más frecuentemente, proponemos un algortmo basado en GRASP para resolver este problema.

El procedimiento de búsqueda voraz aleatorizado adaptativo (GRASP, del inglés Greedy Randomized Adaptive Search Procedure) [5] es un algoritmo de optimización iterativo que puede adaptarse para resolver distintos problemas de búsqueda. GRASP ha sido aplicado con éxito a gran cantidad de problemas reales y de investigación [6]. Su esquema de trabajo general se muestra en la figura 2. Cada iteración de GRASP consiste en dos pasos principales: (i) construcción de la solución y (ii) la mejora de la solución usando un algoritmo de búsqueda local.

**Algorithm 2** Algoritmo de reparación aleatorio tras invalidez

```
i ← 0
while i < maxIterations do
    s ← S_0
    init ← currentState()
    testActions ← previousValidActions
    for all a ∈ previousValidActions do
        s ← perf(a)
    end for
    availableActions ← acc(s)
    while |testActions| < maxTestSize do
        action ← chooseRandom(availableActions)
        s ← perf(action)
        if |testActions| < minTestSize then
            finish ← currentState()
            if eval(p, testActions, init, finish) then
                return testActions
            end if
        end if
        availableActions ← acc(s)
    end while
    i ← i + 1
end while
return {}
```

**Fase de construcción**. Durante la fase de construcción, GRASP comienza creando una soución vacía. Los elementos son añadidos a la solución en construcciones de manera iterativa, hasta que se obtiene una solución competa y válida al problema. Por ejemplo, en el caso de la reparación de casos de prueba de interfaz, la solución vacía no contendría ninguna acción desde el índice de invalidezl test. En cada iteración, se añadiría una acción a la secuencia de acciones del test, generando tests válidos pero que en general no harán que se cumpla el predicado del test. El conjunto de acciones posibles para añadir en el test en cada iteración, serán aquellas que puedan realizarse sobre el estado actual de la interfaz de usuario de la aplicación (determinadas por la secuencia previa de acciones).

Para añadir un elemento a la solución parcial actual durante la fase de construcción, el algoritmo realiza tres pasos. Primero se determinal el conjunto acciones posibles a realizar. Por ejemplo, en nuestro ejemplo motivador, si el índice de invalidez fuera 3, el conjunto de acciones posibles en el estado intermedio tras introducir el texto sería: i) pulsar el botón Salvar, ii) introducir un texto alternativo aleatorio, o iii) pulsar el botón Cancelar. A continuación, un subconjunto de acciones candidatas prometedoras es seleccionada del conjunto de acciones disponibles. Este subconjunto es conocido como el conjunto de cadidatos restringido (RCL, del inglés *Restricted Candidate List*). La selección de los elemento de la RCL debería ser *voraz* y *adaptativa*.

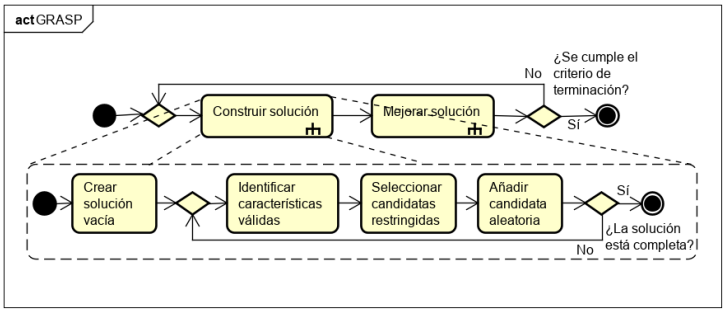


**Fig. 2.** Estructura general del algoritmo GRASP como diagrama de actividad UML

Por *voraz* queremos decir que el criterio utilizado para seleccionar elementos para formar parte de la RCL debe promover la selección de las acciones más prometedoras. Por ejemplo, un criterio voraz en el caso de nuestro problema podría ser incluir aquellas acciones que ya aparecieran en el caso de pruebas correcto antes del cambio. Al establecer este criterio suponemos que dichas acciontes tenderán a generar casos de prueba que nos lleven a resultados similares a los del caso de prueba correcto anterior. Es decir, en nuestro caso de ejemplo, estarían dentro de la RCL dos acciones: Pulsar el botón Salvar, e introducir un texto.

Por adaptativo queremos decir que el criterio usado debería tener en cuenta la solución parcial actual. Por ejempo, como criterio adaptativo tendremos que el algoritmo no incluye en el conjunto de candidatos restringido las acciones ya realizadas en la secuencia, siempre y cuando queden acciones de la sencuencia del caso de prueba original por realizar. Por tanto en nuestro ejemplo sólo quedaría en la RCL la pulsación del botón Salvar.

Si no existieran acciones que cumplieran ambos criterios descritos anteriormente, la RCL estaría formada en nuestro caso por todas las acciones posibles. Por tanto en este último caso la búsqueda se vuelve totalmente aleatoria, puesto que al final de la iteración de la fase de construción, un elemento de la RCL es seleccionado aleatoriamente y es añadido a la solución actual.

**Fase de mejora**. En la fase de mejora, un algoritmo de búsqueda local es ejecutado usando como solución inicial la generada durante la fase de construcción. El algoritmo de búsqueda local concreto implementado en nuestra propuesta de algoritmo es una búsqueda en escalada con criterio de terminación por número de iteraciones. La estructura de vecindario para este algoritmo de búsqueda elige un índice dentro de la lista de acciones del caso de prueba actual, y selecciona un vecino aleatoriamente de entre los vecinos por adición o por substracción para ese índice (tal y como se definen en la sección 3.3).

## 5 Implementación y validación experimental

La validación de las propuestas de este artículo tiene dos objetivos principales: por un lado corroborar que el problema de reparación de casos de prueba de interfaz de usuario puede resolverse como un problema de búsqueda; y por otro, evaluar si el algoritmo basado en GRASP propuesto es capaz reparar casos de prueba, y si su rendimiento es mejor que el de los algoritmos de búsqueda puramente aleatorios para los casos de pruebas seleccionados.

Para realizar la validación se han creado dos aplicaciones Android. Para cada una de esas aplicaciones, se han realizado modificaciones de interfaz que rompían un total de 9 casos de prueba. Concretamente para la aplicación de gestión de notas que hemos usado como ejemplo durante este artículo se generaron 7 casos de prueba inválidos a partir de 6 casos de prueba válidos, probando las acciones básicas de creación, edición y borrado de notas, y sus respectivas cancelaciones.

Además se ha creado otra aplicación android con una interfaz de usuario mucho más compleja (varias ventanas distintas y decenas de acciones posibles desde el estado inicial), y se han implementado casos de prueba que exijen la navegación a través de varias pantallas para conseguir el cumplimiento de los predicados del test (que hacen referencia a un mensaje de confirmación que se muestra en la pantalla principal de la aplicación)[2].

### 5.1 Detalles de la implementación

Para la implementación de los algoritmos descritos en este artículo[3] se ha usado *Android Studio* y *Google UI Automator*. Se ha desarrollado un formato textual para la descripción de casos de prueba de interfaz de usuario, que incluye: i) el paquete de la aplicación a probar (que permite ejecutarla para comenzar la prueba y cerrarla al terminar), ii) la semilla del generador de números aleatorios usada para la ejecución del test que permite hacerlo replicable, iii) una descripción de las acciones a ejecutar muy similar a la mostrada en la sección 3.1, y iv) la definición del predicado a evaluar para validar el comportamiento esperado. Para la evaluación de los predicados se ha usado el evaluador del lenguaje de expresiones del framework Spring (SPEL). Esto permite tener un nivel de expresividad similar al del propio lenguaje java en la definición de los predicados, considerando que el predicado no se cumple en caso de que la evaluación no devuelva un resultado de tipo booleano.

### 5.2 Detalles del experimento

Se ha ejecutado cada algoritmo 10 veces para cada caso de prueba, recopilando el número de veces que se ha encontrado un caso de prueba correcto y válido,

[2] La especificación de todos los casos de prueba tanto válidos como inválidoss para la versión actual de las apliaciones están disponbiles en el paquete de laboratorio de este artículo

[3] El código fuente de los algoritmos junto con las propias aplicaciones de pruebas desarrolladas están disponibles en https://github.com/adrcanfer/automatic-uitests-repair.

el tiempo de cada ejecución, y el número de evaluaciones de la función objetivo de cada ejecución. Los parámetros usados para la ejecución de cada algoritmo se detallan en la tabla 1

| Algoritmo | Parámetro | Valor |
|---|---|---|
| GRASP | NIteraciones | 10 |
| | NIteracionesMejora | 5 |
| Aleatorio desde 0 | NIteraciones | 50 |
| | minTestSize | Tamaño del test original |
| | maxTestSize | Doble del tamaño del test original |
| Aleatorio incremental | NIteraciones | 50 |
| | minTestSize | Tamaño del test original |
| | maxTestSize | Doble del tamaño del test original |

**Table 1.** Valores de los parámetros de cada algoritmo

Los experimentos fueron ejecutados en un PC con procesador Intel i7 con 8 núcleos a 3.7 Ghz, 16 Gigas de RAM, sistema operativo Ubuntu Linux 18.04. En cuanto a los entornos de desarrollo y ejecución de las pruebas, se usaron Android Studio 3.3.2, y el emulador basado en x86 virtualizando un teléfono Nexus X5 para un nivel de la API de Android 28 (Android 9.0).

### 5.3 Resultados

La tabla 2 muestra el éxito de los distintos algoritmos a la hora de reparar los tests del experimento. Puede apreciarse claramente como el algoritmo con mayor

| Algoritmo | Reparados | PorcentajeReparados |
|---|---|---|
| 1 GRASPTeR | 89 | 98.89 |
| 2 Aleatorio desde 0 | 77 | 85.56 |
| 3 Aeatorio incremental | 85 | 94.44 |

**Table 2.** Capacidad de los algoritmos para reparar los tests

éxito fue la propuesta basada en GRASP, seguida de el algoritmo aleatorio incremental que tiene un porcentaje de pruebas no reparadas de más del cuádruple del primero. Finalmente el peor algoritmo en estos términos es el algoritmo aleatorio desde cero, con un porcentaje de pruebas no reparadas muy superior a los anteriores.

Otro indicador interesante de la calidad de los tests generados es su longitud, ya que permite apreciar hasta que punto estas pruebas son óptimas (en términos de tener la longitud mínima necesaria para que se cumplan sus predicados) o incluyen acciones superfluas. La tabla 3 muestra la longitud media de los tests generados por cada algoritmo para cada caso de prueba a reparar.

| Test | GRASP | TeR | Aleatorio desde 0 | Aleatorio incremental |
|---|---|---|---|---|
| 1 Broken-CancelCreationTestCase | 2.60 | | 4.40 | 3.80 |
| 2 Broken-CancelEditionTestCase | 3.20 | | 3.00 | 3.10 |
| 3 Broken-ComplexTestCase | 2.00 | | 2.20 | 2.00 |
| 4 Broken-CreationTestCase | 3.70 | | 3.30 | 3.00 |
| 5 Broken-DeletionTestCase | 2.00 | | 3.00 | 2.60 |
| 6 Broken-EditionTestCase | 3.00 | | 5.20 | 3.40 |
| 7 Broken2-CreationTestCase | 3.40 | | 3.90 | 3.50 |
| 8 Broken2-DeletionTestCase | 1.90 | | 2.00 | 2.00 |
| 9 Broken2-EditionTestCase | 3.40 | | 4.20 | 4.00 |

**Table 3.** Longitud de las secuencias de acciones generadas

Puede apreciarse claramente cómo el algoritmo basado en GRASP ofrece los mejores resultados medios para todos los casos excepto dos. Además en uno de esos casos la diferencia entre las medias (con el algoritmo aleatorio incremental), es de tan solo una décima. Por ello consideramos que el algoritmo es superior a los otros respecto de esta métrica.

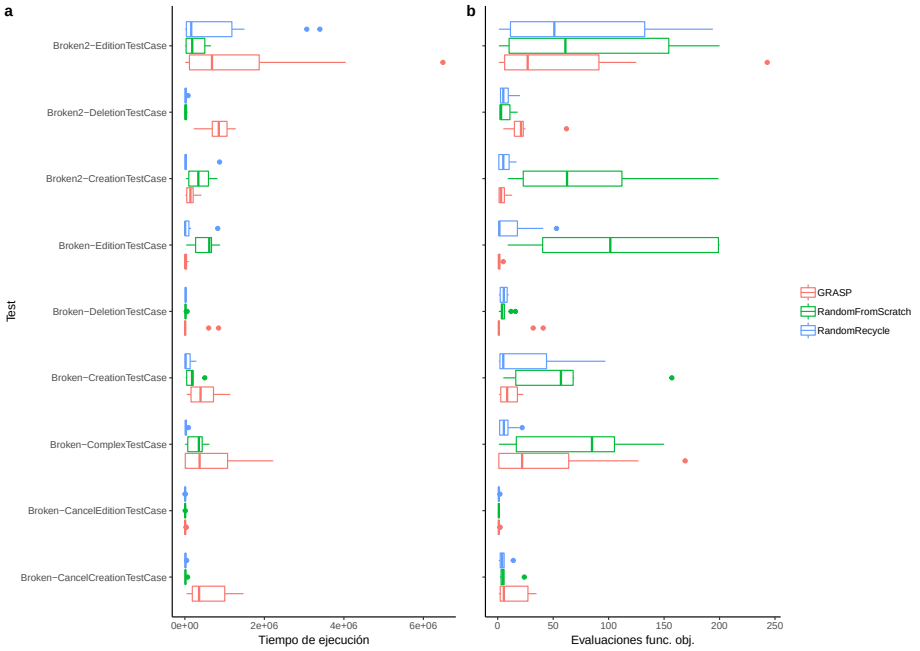


**Fig. 3.** Boxplots del realizadas por cada algoritmo para cada prueba a reparar

Las figuras 3(a) y 3(b) muestran los boxplots de resultados de cada algoritmo para el tiempo de ejecución, y el número de evaluaciones de la función objetivo respectivamente. Puede apreciarse como el comportamiento de GRASP (en rojo) es bastante bueno en términos del número evaluaciones de la función objetivo.

Sin embargo, el tiempo de ejecucion del algoritmo GRASP (en rojo) es el más alto de los 3 en general. Esto se debe al uso de una exploración exahustiva de las estructuras de vecindario en los índices seleccionados durante la fase de mejora, que implica la ejecución completa desde la primera a la última acción de su secuencia para de cada vecino, mientras que los algoritmos de búsqueda aleatoria y la fase de construcción escogen una única acción a ejecutar de entre las disponibles. Sin embargo, dado que la mayoría de las ejecuciones del algoritmo están por debajo de los 10 segundos (nótese que las unidades del eje vertical de la figura 3(a) son milisegundos), consideramos que este comportamiento no limita en demasía la aplicabilidad de esta propuesta. Se han comparado los resultados de los algoritmos en términos de tiempo de ejecución, número de ejecuciones en que se encontro el óptimo, y número de evaluaciones de la función objetivo. Dado que las distribuciones de los datos no eran normales en general (comprobación realizada usando el test de Shapiro-Wilk) usamos el tests de Mann-Withney para comparar los resultados de los algoritmos, y el indicador $A^{12}$ para evaluar el tamaño del efecto. Los resultados concretos confirman en general todo lo descrito anteriormente, y no serán incluidos como parte del artículo en aras de la brevedad, pero están disponibles en el paquete de laboratorio del artículo (en formato R Markdown).

## 6 Conclusiones

En este trabajo se ha formalizado el problema de reparación de casos de prueba de interfaz de usuario como un problema de búsqueda, y se han presentado tres algoritmos para su resolución. Mediante un experimento controlado, se ha validado que dichos algoritmos pueden llegar a resolver el problema de búsqueda planteado tanto para casos de prueba simples (de 3 o menos pasos y predicados con hasta 2 cláusulas en forma normal conjuntiva) como relativamente complejos (decenas de pasos y predicados con 4 o más cláusulas). Además se ha comprobado que el algoritmo basado en GRASP propuesto es más eficiente que los algoritmos aleatorios tanto en términos de número de casos de prueba reparados como de evaluaciones de la función objetivo realizadas.

Una lección aprendida que se ha extraido de este trabajo es que para mejorar la robustez y reparabilidad de los casos de prueba de interfaz de usuario, los predicados debe ser los más declarativos, precisos e independientes del contexto posibles. Por ejemplo, cuando definimos un caso de prueba de borrado de notas para nuestra aplicación, originalmente definimos el predicado como $|finish| < |init|$. El resultado de esta definición simplista fue que muchas de las reparaciones generadas por los algoritmo nos llevan a la ventana de edición de notas de la aplicación, que contiene pocos elementos, y que por tanto, hacía que se cumpliera el predicado incluso aún cuando no se hubiera borrado ninguna nota. Resolvimos esto reformulando el predicado como $finish \subseteq init \wedge |finish| = |init| - 1$.

Como trabajo futuro los autores del trabajo pretenden incrementar la flexibilidad del lenguaje de predicados para soportar el uso de un conjunto más amplio de acciones (permitiendo por ejemplo la ejecución de pulsaciones sobre imágenes

o el uso de gestos), aplicar los algoritmos a aplicaciones de código abierto, y comparar el rendimiento de las propuestas con el de las aproximaciones basadas en modelos de GUI.

## Replicabilidad y paquete de laboratorio

El paquete de laboratiorio de este artículo incluye todos los casos de prueba tanto válidos como inválidos, los resultados de ejecución de los algoritmos, y los scripts de análisis estadístico de resultados, está disponibles en https://exemplar.us.es/demo/JAParejoJISBD2019.

El código fuente de las implementaciones (algoritmos y aplicaciones) están disponibles en https://github.com/adrcanfer/automatic-uitests-repair.

## Agradecimientos

Este trabajo ha sido posible gracias al apoyo del gobierno español en el marco de los proyectos BELI (TIN2015-70560-R)y and HORATIO (RTI2018-101204-B-C21), y de la Junta de Andalucía a través del proyecto COPAS (P12-TIC-1867)

## References

1. Arnatovich, Y.L., Wang, L.: A systematic literature review of automated techniques for functional gui testing of mobile applications. arXiv preprint arXiv:1812.11470 (2018)
2. Atwood, J.: The user interface is the application (2005), https://blog.codinghorror.com/the-user-interface-is-the-application/
3. Banerjee, I., Nguyen, B., Garousi, V., Memon, A.: Graphical user interface (gui) testing: Systematic mapping and repository. Information and Software Technology **55**(10), 1679–1694 (2013)
4. Ellis, D.: Selenium: Front End Testing and Continuous Integration. O'Reilly Media, Inc., 1st edn. (2017)
5. Festa, P., Resende, M.G.: An annotated bibliography of grasp–part i: Algorithms. International Transactions in Operational Research **16**(1), 1–24 (2009)
6. Festa, P., Resende, M.G.: An annotated bibliography of grasp–part ii: Applications. International Transactions in Operational Research **16**(2), 131–172 (2009)
7. Gao, Z., Chen, Z., Zou, Y., Memon, A.M.: Sitar: Gui test script repair. Ieee transactions on software engineering **42**(2), 170–186 (2016)
8. Li, X., Chang, N., Wang, Y., Huang, H., Pei, Y., Wang, L., Li, X.: Atom: Automatic maintenance of gui test scripts for evolving mobile applications. In: 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST). pp. 161–171. IEEE (2017)
9. Memon, A.M.: Gui testing: Pitfalls and process. Computer (8), 87–88 (2002)
10. Memon, A.M.: Automatically repairing event sequence-based gui test suites for regression testing. ACM Transactions on Software Engineering and Methodology (TOSEM) **18**(2), 4 (2008)
11. Su, T.: Fsmdroid: Guided gui testing of android apps. In: 2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C). pp. 689–691 (May 2016)