Generación de pruebas unitarias mediante métodos de búsqueda y modelos de lenguaje

Marcos García[1][0009−0003−9307−6994], Javier Yuste[1][0000−0002−5956−9977], and Eduardo G. Pardo[1][0000−0002−6247−5269]

GRAFO, Departamento de Informática y Estadística,
Universidad Rey Juan Carlos, C/ Tulipán s/n, Móstoles, 28933, Spain
m.garciaga.2022@alumnos.urjc.es,{javier.yuste,eduardo.pardo}@urjc.es

Resumen Los sistemas *software* modernos crecen continuamente en complejidad, lo que hace que las tareas de verificación y validación sean cada vez más importantes para su correcto desarrollo. Desafortunadamente, la generación de pruebas de calidad es un proceso costoso, complejo y propenso a errores. En este contexto, los modelos de lenguaje de gran escala emergen como una alternativa prometedora para automatizar algunas de estas tareas. Este trabajo analiza y compara el rendimiento de enfoques basados en algoritmos de búsqueda y en modelos de lenguaje para la generación automática de pruebas unitarias, en el contexto de la disciplina *Search-Based Software Testing*. En particular, se evalúa DynaMOSA, considerado el estado del arte para la generación de pruebas unitarias, y se compara con un enfoque basado en generación y corrección iterativa de pruebas utilizando Gemini. Los resultados obtenidos sobre instancias de referencia muestran que el enfoque basado en modelos de lenguaje alcanza, en promedio, una mayor calidad en las soluciones generadas. No obstante, ambos métodos presentan un rendimiento competitivo y DynaMOSA supera a Gemini en un subconjunto particular de instancias. Para profundizar en estas diferencias, se realiza un análisis de correlación entre las características del código y los resultados obtenidos. En conjunto, este estudio evidencia la complementariedad de ambas aproximaciones y sugiere que su hibridación podría mejorar significativamente el rendimiento, lo que abre nuevas líneas de investigación.

Palabras clave: *Search-Based Software Engineering* · Generación de pruebas unitarias · Heurísticas · *Large Language Models*

1. Introducción

El desarrollo de *software* moderno se enfrenta a una creciente complejidad y a exigencias de calidad cada vez mayores. En el ciclo de vida de desarrollo de *software* (SDLC, del inglés *Software Development Life-Cycle*), la fase de pruebas es crítica para garantizar la fiabilidad del sistema. Desafortunadamente, se estima que las tareas de verificación y validación pueden llegar a consumir entre el 30 % y el 40 % de los recursos totales del proyecto [14]. La propensión a errores

y la limitada escalabilidad de las técnicas manuales se ven desafiadas por las crecientes demandas de calidad. Más aún, los sistemas *software* se ven sujetos a cambios continuos para evolucionar y satisfacer los requisitos de los usuarios [5]. Estos cambios continuos acrecientan la necesidad de procesos de verificación y validación eficientes y de calidad.

Para mitigar estos costes y asegurar pruebas de calidad, el área de *Search-Based Software Testing* (SBST) propone estudiar la generación automática de casos de prueba como un problema de optimización [9]. SBST se enmarca en el área de investigación de *Search-Based Software Engineering* (SBSE), una disciplina centrada en la reformulación de problemas de Ingeniería del *Software* como problemas de optimización y la implementación de métodos para resolverlos [16]. El término SBSE fue acuñado en 2001 por Harman y colaboradores [4]. En este contexto, SBST se centra en el uso de métodos de búsqueda para automatizar parcial o totalmente tareas de *testing*. Aunque los primeros trabajos en el área de SBST tuvieron lugar en 1976 [10], el interés en el área ha crecido en los últimos años, organizándose incluso competiciones internacionales sobre la temática [2]. Este creciente interés ha sufrido aún un mayor impulso debido a los Modelos de Lenguaje de Gran Escala (LLMs, del inglés *Large Language Models*) [15], que muestran un rendimiento notable para las tareas de generación de pruebas [17].

En este trabajo se presenta una comparativa del rendimiento de métodos de búsqueda y LLMs para el problema de la generación automática de pruebas unitarias. En particular, se comparan el mejor método de búsqueda de la literatura (DynaMOSA [12]) y un *pipeline* propuesto en este trabajo, que usa un LLM de propósito general disponible públicamente (Gemini 3.1 Pro [3]). La comparativa se lleva a cabo sobre un conjunto de instancias publicado recientemente [2]. Finalmente, se realiza un análisis de la correlación entre distintas características del código de las instancias y los resultados obtenidos para ofrecer una visión más detallada de su desempeño.

El resto del trabajo se estructura de la siguiente manera. En la Sección 2 se define el problema de optimización. En la Sección 3, se describen los métodos comparados. En la Sección 4, se presentan los resultados obtenidos por cada método. En la Sección 5, se realiza un análisis de correlación entre las características de las instancias y los resultados obtenidos. Finalmente, las conclusiones y trabajos futuros se presentan en la Sección 6.

2. Definición del problema

Dentro de la familia de problemas de generación de casos de prueba [8], este trabajo se centra en la generación automática de pruebas unitarias, conocido comúnmente como TCG, del inglés *Test Case Generation*. Las pruebas unitarias tienen como objetivo verificar los componentes más pequeños del *software* de forma aislada (por ejemplo, una clase o un método). En este contexto, una instancia para el problema es una clase. Específicamente, en este trabajo, una clase Java. Una solución se compone de un conjunto de casos de prueba, donde cada caso de prueba es un test unitario compuesto de una serie de sentencias o

instrucciones de código. Debido a la naturaleza de las soluciones, el espacio de búsqueda podría ser infinito, ya que, incluso aunque se limite el tamaño de las pruebas, las variables y los parámetros de los métodos pueden tomar valores de conjuntos no finitos (por ejemplo, números enteros o reales, cadenas de texto, etc.).

Para evaluar la calidad de las soluciones, se define la función objetivo f como la cobertura de ramas o *Branch Coverage*. Esto es, el porcentaje de ramas de código que son ejecutadas al menos una vez, al procesar todos los casos de prueba de la solución (S). Formalmente:

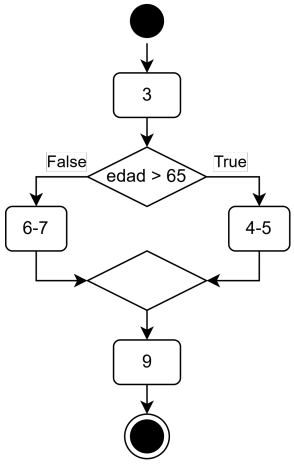
$$f(S) = \frac{\text{Ramas cubiertas por } S}{\text{Ramas totales}} \times 100.$$

El objetivo del problema es encontrar la solución que maximiza esta función objetivo. Es decir, que cubre todo el código a probar.

En la Figura 1a, se muestra una instancia de ejemplo consistente en el código fuente de una clase en Java con un método. En la Figura 1b se muestra el diagrama de actividad, con dos ramas, correspondiente al método anterior. El método recibe dos parámetros de entrada: *edad* y *precio*. Si el valor del parámetro edad es menor o igual a 65, entonces se ejecutan las líneas 3-4 y 6-9. En cambio, si el parámetro edad es mayor a 65, se ejecutan las líneas 3-5 y 9. El Código 1.1 muestra un test, llamado $testAplicarDescuento()$, con valores de parámetros $edad = 70$ y $precio = 10$. Este código es una solución para la instancia de la Figura 1. Como se puede observar, al ejecutar este test, solo se ejecutará la rama que contiene las líneas 4-5, dejando inexplorada la otra rama. Por lo tanto, esta solución tendría una calidad del 50 %, al cubrir la mitad de las ramas.

```
1  public class Tienda {
2      public double aplicarDescuento(
           int edad, double precio) {
3          double precioTotal;
4          if (edad > 65) {
5              precioTotal = precio *
                   0.5;
6          } else {
7              precioTotal = precio *
                   0.8;
8          }
9          return precioTotal;
10     }
11 }
```

(a) Código Java.



(b) Diagrama de actividad.

Figura 1: Representación de una instancia del problema TCG: código fuente y su grafo de flujo de control asociado.

Código 1.1: Ejemplo de solución para la clase `Tienda`.

```
public class TiendaTest {
    @Test
    public void testAplicarDescuento() {
        Tienda tienda = new Tienda();
        double resultado = tienda.aplicarDescuento(70, 10.0);
        assertEquals(5.0, resultado, 0.001);
    }
}
```

3. Métodos para la generación de pruebas unitarias

En este trabajo se comparan dos métodos de generación de pruebas unitarias. En primer lugar, en la Sección 3.1, se introduce DynaMOSA [12], un método de la literatura basado en algoritmos genéticos. En segundo lugar, en la Sección 3.2, se introduce la aproximación propuesta basada en el uso de LLMs.

3.1. Método de búsqueda DynaMOSA

En SBST, existen varios entornos para la automatización de generación de casos de prueba. Entre ellos, uno de los más populares es EvoSuite [1], que ofrece múltiples herramientas para facilitar la implementación de métodos de búsqueda para el problema del TCG. DynaMOSA [12], considerado el actual estado del arte para el TCG, está implementado en este entorno.

DynaMOSA es una extensión de MOSA, un algoritmo genético propuesto en 2015 [11]. Este método está basado en la aplicación de operadores clásicos de cruce y de operadores de mutación basados en la eliminación, adición y modificación de sentencias de código. Históricamente, los métodos para el problema buscaban optimizar la cobertura de todo el conjunto de pruebas de manera global. No obstante, la aparición de MOSA [11] redefinió el problema al tratar cada rama como un objetivo independiente. La principal contribución de MOSA es la introducción de un criterio de preferencia particular, que guía la búsqueda hacia el cubrimiento de nuevas ramas. A pesar del avance de MOSA, este presenta la limitación de intentar optimizar todas las ramas simultáneamente, ignorando las dependencias estructurales del código, como la existencia de ramas anidadas.

DynaMOSA extiende el criterio de preferencia de MOSA. En particular, la principal contribución de DynaMOSA es la modificación dinámica del esfuerzo durante el proceso de búsqueda, concentrando la búsqueda en un subconjunto relevante de objetivos en cada iteración [12]. La idea es priorizar las ramas a cubrir en base a un criterio que establece la facilidad con la que se puede llegar a cubrir dichas ramas a partir de las soluciones actuales. DynaMOSA representa actualmente el estado del arte para el problema del TCG [13].

3.2. Método basado en *Large Language Models*

En este trabajo se propone utilizar una arquitectura que consiste en un flujo de automatización, que integra el acceso a LLMs con el ciclo de construcción de *software*, realizado mediante Maven. En particular, se utiliza la interfaz de línea de comandos GeminiCLI[1], que permite interactuar con los modelos de la familia Gemini. Esta herramienta selecciona dinámicamente el modelo a utilizar en función de la dificultad de la tarea. No obstante, para los experimentos realizados en este trabajo, eligió de manera consistente el modelo Gemini 3.1 Pro [3].

El pseudocódigo recogido en el Algoritmo 1 presenta el flujo de ejecución para la generación de casos de prueba mediante Gemini. El método recibe 4 parámetros de entrada: la clase para la que generar tests (CUT, del inglés *Class Under Test*); el tiempo límite (t_{max}); y dos plantillas para generar distintos *prompts* (P_1 y P_2). El método comienza generando una solución para la clase mediante Gemini utilizando un *prompt* generado a partir de la plantilla P_1 (línea 3). Esta plantilla se presenta en el Código 1.2. Como se puede observar, esta plantilla define al modelo como un experto sénior en Java, le obliga a emplear Mockito[2] para dependencias externas, y le insta a centrarse en crear pruebas para los escenarios más probables, casos límite y excepciones. Además, se da acceso al LLM a todo el proyecto al que pertenece la clase. Si la solución generada no es factible (por ejemplo, si el código generado no compila), entonces se intenta reparar la solución hasta que se alcanza el tiempo límite (línea 5). Para esta reparación, se utiliza un *prompt* generado con la segunda plantilla. Esta plantilla se presenta en el Código 1.3. Como se puede observar, en este segundo *prompt* se le facilitan al LLM los errores generados por la solución actual reportados por Maven. Finalmente, el método devuelve la solución generada (línea 7).

Algoritmo 1 Pseudocódigo del procedimiento de generación de tests con LLMs

```
1: procedure GeneraciónLLM(CUT, t_max, P_1, P_2)
2:     t ← 0
3:     S ← GeneraTests(CUT, P_1)
4:     while ¬factible(S) ∧ t < t_max do
5:         S ← GeneraTests(CUT, P_2)
6:     end while
7:     return S
8: end procedure
```

[1] https://geminicli.com/
[2] https://site.mockito.org/

Código 1.2: Plantilla del *prompt* de generación.

```
"You are a senior Java 8 and JUnit 4 expert specialized in Software Testing.
Your goal is to create a robust test suite for the class: {target_class_path}.
TASKS:
1. Generate a JUnit 4 test class named {os.path.basename(target_class_path).replace('.java', 'GEMTest')}.
2. Use Mockito for all external dependencies needed.
3. OBJECTIVE: Achieve high branch coverage. Include tests for:
- Happy path (standard usage).
- Edge cases (null inputs, empty strings, boundary values).
- Exception handling (ensure expected exceptions are thrown and caught).
4. Maintain the original package structure and all necessary imports.
5. Use only existing constructor and methods of the class, do NOT invent new ones.
6. You can check the context of the project provided in the --include-directories argument to understand the code.
TARGET CLASS PATH: {target_class_path}
Respond only with the Java code inside a single \`\`\`java block."
```

Código 1.3: Plantilla del *prompt* de reparación.

```
"You are a senior Java 8 and JUnit 4 expert specialized in Software Testing.
Your goal is to repair the test class: {test_path}.
CONTEXT:
1. The compilation/runtime errors are in the file: {error_file} (provided via --include-directories).
2. The source code of the class under test is at: {target_class_path}.
TASKS:
1. READ and ANALYZE the specific errors in {error_file}.
2. FIX the test class to resolve all identified errors:
    - If COMPILATION ERROR: Focus on fixing missing imports, incorrect method signatures, or constructor mismatches, as well as any other compilation issue.
    - If RUNTIME ERROR: Focus on fixing the test logic, object initialization, or Mockito stubbing.
3. STRICT RULE: Cross-reference the class under test ({target_class_path}) to verify constructors and methods. DO NOT invent any method signature.
4. Maintain the original package and all necessary imports.
Respond only with the corrected Java code inside a single \`\`\`java block."
```

4. Resultados experimentales

Para comparar el rendimiento de los métodos descritos, se ha seleccionado un subconjunto de instancias del benchmark oficial de la 10ª edición de la *Java Unit Testing Competition* (SBST 2022) [2]. Este *benchmark* está compuesto por 65 clases Java, que fueron seleccionadas por los organizadores de entre un total de 584 clases analizadas. Estas clases pertenecen a cuatro proyectos de código abierto ampliamente reconocidos: FastJSON[3], Guava[4], Seata[5] y Spoon[6]. Todas las instancias tienen una complejidad ciclomática de McCabe [7] superior a cinco, la presencia de al menos una bifurcación lógica y una testabilidad mínima verificada. De las 65 instancias disponibles, 21 se han descartado por errores de ejecución en EvoSuite, dejando un *benchmark* final de 44 instancias.

Todos los experimentos se han ejecutado en el mismo entorno computacional: un servidor con un procesador AMD EPYC 7282, 16 cores, 8 GB de RAM y el sistema operativo Ubuntu 20.04.6 LTS. Los lenguajes de programación utilizados son Java 8 en DynaMOSA y Python 3.13.13 para el *pipeline* con Gemini. La implementación de DynaMOSA es la disponible en el entorno de EvoSuite v1.2.0[7]. Para garantizar una comparación justa, el tiempo límite dado a DynaMOSA para cada instancia es de 180 segundos, de acuerdo con las especificaciones observadas en artículos relacionados. Este mismo tiempo se ha proporcionado al proceso de generación de pruebas mediante LLMs.

La Tabla 1 muestra los resultados obtenidos. Por cada proyecto, se muestra la calidad media de las pruebas generadas por cada método (BC, del inglés *Branch Coverage*), la desviación media con respecto a la mejor solución encontrada por cualquiera de los métodos (Dev. (%)) y el número de instancias para las que cada método ha obtenido una solución igual o mejor que el otro (# Mejores).

Como se puede observar, el método denotado como Gemini es capaz de generar casos de prueba con una calidad superior a los generados por DynaMOSA, con un 74,82 % de cobertura frente a un 67,91 %. Además, genera mejores casos de prueba para 13 instancias más que DynaMOSA. Sin embargo, es importante destacar que Gemini tiene una desviación superior al 20 %, lo que indica que existen clases para las que genera casos de prueba de mucha peor calidad que DynaMOSA. En particular, observando el rendimiento de DynaMOSA se aprecia que mejora a Gemini para la mayoría de clases del proyecto Seata.

En la Figura 2 se muestra un *boxplot* que resume la distribución de los resultados obtenidos por ambos métodos en las 44 instancias evaluadas. El gráfico permite visualizar no solo la superioridad en la mediana por parte del *pipeline* basado en LLMs, sino también la presencia de valores atípicos con 0 % de cobertura. Estos resultados se deben a instancias para las que Gemini no fue capaz de generar casos de prueba válidos dentro del tiempo límite establecido, a pesar del proceso de reparación.

[3] https://github.com/alibaba/fastjson
[4] https://github.com/google/guava
[5] https://github.com/seata/seata
[6] https://github.com/INRIA/spoon
[7] https://github.com/EvoSuite/evosuite/releases/tag/v1.2.0

Tabla 1: Comparativa de los resultados obtenidos por DynaMOSA y Gemini para las 44 clases pertenecientes a cuatro proyectos distintos.

Proyecto (#Clases)	**DynaMOSA**			**Gemini**		
	BC	**# Mejores**	**Desv.**	**BC**	**# Mejores**	**Desv.**
FastJSON (11)	74,92 %	4	15,03 %	**84,15 %**	**9**	**10,42 %**
Guava (13)	61,22 %	2	36,29 %	**93,23 %**	**11**	**1,21 %**
Seata (10)	**77,60 %**	**10**	**0,00 %**	40,00 %	4	60,00 %
Spoon (10)	59,19 %	3	25,68 %	**75,45 %**	**8**	**20,00 %**
Total (44)	67,91 %	19	**20,32 %**	**74,82 %**	**32**	21,15 %

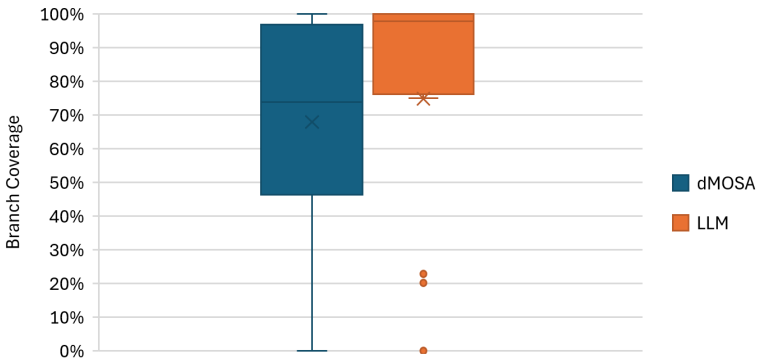


Figura 2: *Boxplot* para la comparación de la calidad de los casos de prueba generados por DynaMOSA y Gemini.

5. Análisis de los resultados

Para analizar las diferencias en el rendimiento de los métodos descritos anteriormente, se ha realizado un análisis de correlación entre la calidad de las soluciones obtenidas por cada método y distintas características de las instancias. El objetivo es comprobar qué características de las instancias afectan positiva o negativamente a los métodos a la hora de generar casos de prueba de calidad. Este análisis se realiza a nivel de método, evitando que las métricas globales de una clase opaquen comportamientos específicos en métodos de distinta naturaleza.

La Tabla 2 resume las características extraídas de las instancias, clasificadas en cuatro categorías: complejidad, acoplamiento, parámetros de entrada y visibilidad. La primera categoría incluye métricas de complejidad ampliamente conocidas en Ingeniería del *Software*, como el número de líneas de código o el número de comparaciones; la segunda categoría contiene métricas relacionadas con la interacción entre clases y/o métodos; la tercera categoría incluye métricas relacionadas con el tipo de parámetros de entrada que recibe el método; y, finalmente, la cuarta categoría incluye una métrica que recoge la visibilidad del

método, indicando si es público. Para cada métrica recogida, se incluye una descripción en la tabla. Las métricas de complejidad y acoplamiento se han extraído mediante la herramienta CK[8]. El resto de métricas se han extraído mediante un *script* propio escrito en Python, haciendo uso de la librería javalang[9]. Las herramientas mencionadas ofrecen más métricas que las recogidas en esta tabla. No obstante, por cuestiones de legibilidad, se ha optado por presentar únicamente aquellas métricas más relevantes.

Tabla 2: Descripción de métricas empleadas para el análisis de correlación.

Categoría	**Métrica**	**Descripción**
Complejidad	Complejidad de McCabe	Recuento de instrucciones de bifurcación.
	Máximo anidamiento	Nivel máximo de anidamiento de bloques dentro del método.
	LOC	Cantidad de líneas de código fuente (sin contar comentarios ni líneas vacías).
	Comparaciones	Cantidad de instrucciones de comparación lógica.
Acoplamiento	Métodos invocados	Número de llamadas a otros métodos.
	Fan-out	Número de dependencias a otras clases.
Parámetros	Cantidad	Número total de parámetros de entrada en la firma del método.
	Object	Cantidad de parámetros de tipo `Object` (tipos complejos).
	String	Cantidad de parámetros de tipo `String`.
	short	Cantidad de parámetros de tipo `short`.
	long	Cantidad de parámetros de tipo `long`.
	char	Cantidad de parámetros de tipo `char`.
	int	Cantidad de parámetros de tipo `int`.
	boolean	Cantidad de parámetros de tipo `boolean`.
	double	Cantidad de parámetros de tipo `double`.
	byte	Cantidad de parámetros de tipo `byte`.
Visibilidad	público	Indica si el método es público.

La Tabla 3 contiene un análisis de la correlación de las características descritas anteriormente y la calidad de los casos de prueba generados por cada método. En particular, se ha realizado un análisis de correlación utilizando los coeficientes de Spearman con $p < 0,05$. Los valores negativos indican que existe un impacto negativo en la calidad de las soluciones. Es decir, cuanto mayor es el valor de la métrica, peor es la calidad de las soluciones.

[8] https://github.com/mauricioaniche/ck
[9] https://pypi.org/project/javalang/

Tabla 3: Coeficientes de correlación de Spearman obtenidos.

Categoría	**Métrica**	**DynaMOSA**	**Gemini**
Complejidad	Complejidad de McCabe	-0,73	-0,67
	Máximo anidamiento	-0,71	-0,63
	LOC	-0,61	-0,55
	Comparaciones	-0,57	-0,58
Acoplamiento	Métodos Invocados	-0,53	-0,42
	Fan-Out	-0,37	-0,36
Parámetros	Cantidad	-0,31	-0,22
	Object	-0,37	-0,21
	String	-0,11	-0,15
	short	0,09	0,08
	long	-0,08	0,01*
	char	-0,08	-0,06*
	int	0,05*	-0,04*
	boolean	0,03*	0,04*
	double	0,01*	0,06*
	byte	0,01*	0,01*
Visibilidad	Público	0,22	0,12

* Valor no significativo ($p > 0,05$).

Como se puede observar, la complejidad de McCabe es la métrica con mayor impacto en la calidad de las soluciones, obteniendo coeficientes de -0,73 y -0,67. Otras métricas relacionadas con la complejidad, como el nivel máximo de anidamiento, las líneas de código o la cantidad de comparaciones, tienen también correlaciones negativas similares, en el rango de -0,71 a -0,55.

En cuanto al acoplamiento, ambos enfoques se ven afectados negativamente, tanto por el número de métodos invocados como por el número de dependencias a otras clases. En el caso del *Fan-Out*, el impacto es similar en ambos métodos. En el caso del número de métodos invocados, sin embargo, se observa mayor diferencia. Mientras que los resultados obtenidos por DynaMOSA presentan una correlación de -0,53, los resultados obtenidos por Gemini presentan una correlación de -0,42.

En cuanto a los parámetros de entrada del método, se puede observar una correlación significativa con el número de parámetros y con el número de parámetros de tipo *Object*, que representan objetos complejos. Respecto a los parámetros de tipos primitivos (como *int*, *boolean* o *double*), se observa que sus coeficientes son muy cercanos a cero y, en gran parte de los casos, carecen de significancia estadística. Esto implica que la presencia de estos tipos simples no es un factor determinante en el éxito o fracaso de la generación de pruebas, situando el verdadero desafío en la gestión de estructuras de datos complejas.

Finalmente, la única característica que parece tener un efecto positivo pronunciado es la visibilidad pública del método, especialmente en el caso de DynaMOSA. Este resultado es coherente con lo observado en la literatura [6]. La ejecución de ramas de métodos públicos se puede alcanzar llamando directa-

mente a dichos métodos. Sin embargo, las ramas de métodos privados deben ejecutarse indirectamente mediante llamadas a otros métodos accesibles.

En líneas generales, las características del código tienen menor impacto en la calidad de las soluciones de Gemini que en la calidad de las soluciones de DynaMOSA. Sorprendentemente, una de las pocas características que tiene un mayor impacto negativo en Gemini que en DynaMOSA es el número de parámetros de entrada de tipo *String*. Esto puede contrastar con la expectativa inicial de que un LLM sea capaz de generar *Strings* adecuados con mayor facilidad que un método de búsqueda. No obstante, la diferencia es pequeña, por lo que sería necesario realizar más experimentos para confirmar esta observación.

6. Conclusiones y trabajos futuros

En este trabajo se ha comparado el rendimiento de dos métodos para el problema de la generación automática de casos de prueba en Java. El primer método, DynaMOSA, basado en algoritmos genéticos, es el estado del arte en métodos de búsqueda para el problema. El segundo método, propuesto en este trabajo, se basa en el uso de LLMs. En particular, se ha empleado el modelo Gemini 3.1 Pro. Los resultados obtenidos sobre un conjunto de instancias estándar en la literatura demuestran que, aunque el uso de LLMs en esta tarea da lugar a resultados de calidad, los métodos de búsqueda del estado del arte ofrecen resultados competitivos. Además, se han observado instancias para las que DynaMOSA es capaz de obtener mejores resultados que Gemini. Esto indica que una hibridación de ambas aproximaciones puede ser prometedora.

Complementariamente, se ha analizado la correlación entre las características de las instancias y los resultados obtenidos, con el objetivo de encontrar las características que llevan a un método a obtener mejores resultados que el otro. Aunque el análisis realizado muestra que el rendimiento de Gemini se ve menos afectado por la complejidad del código que DynaMOSA, no se ha evidenciado ninguna característica del código que permita predecir qué aproximación dará lugar a los mejores resultados.

Como trabajo futuro, se plantean dos líneas de investigación. En primer lugar, profundizar en el análisis de las diferencias de rendimiento entre los métodos comparados. En segundo lugar, explorar diferentes estrategias de hibridación entre ambas aproximaciones. Se espera que la hibridación entre ambas herramientas permita alcanzar coberturas mayores a las obtenidas de forma individual.

Agradecimientos Esta investigación ha sido parcialmente financiada por los siguientes proyectos: PID2024-160226OB-C22, y PID2021-125709OA-C22, financiado por MCIN/AEI y "ERDF A way of making Europe"; CIRMA-CM Ref. TEC-2024/COM-404, financiado por Comunidad Autónoma de Madrid; TSI-100930-2023-3, "Cátedra AI4DDS: Artificial Intelligence for Data Drive Solutions" (Ref. ID MCA07); "Cátedra de Innovación y Digitalización Empresarial entre Universidad Rey Juan Carlos y Second Episode" (Ref. ID MCA06); y "Cátedra Institucional de Transición Energética" (Ref. ID MCA09).

Referencias

1. Fraser, G., Arcuri, A.: Evosuite: automatic test suite generation for object-oriented software. In: Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering. pp. 416–419 (2011)
2. Gambi, A., Jahangirova, G., Riccio, V., Zampetti, F.: Sbst tool competition 2022. In: Proceedings of the 15th Workshop on Search-Based Software Testing. pp. 25–32 (2022)
3. Google DeepMind: Gemini 3.1 pro - model card. Tech. rep., Google DeepMind (2026), `https://deepmind.google/models/model-cards/gemini-3-1-pro/`, Último acceso: 2026-05-08
4. Harman, M., Jones, B.F.: Search-based software engineering. Information and software Technology **43**(14), 833–839 (2001)
5. Lehman, M.M.: Programs, life cycles, and laws of software evolution. Proceedings of the IEEE **68**(9), 1060–1076 (1980)
6. Ma, L., Zhang, C., Yu, B., Sato, H.: An empirical study on the effects of code visibility on program testability. Software Quality Journal **25**(3), 951–978 (2017)
7. McCabe, T.J.: A complexity measure. IEEE Transactions on software Engineering (4), 308–320 (1976)
8. McMinn, P.: Search-based software test data generation: a survey. Software testing, Verification and reliability **14**(2), 105–156 (2004)
9. McMinn, P.: Search-based software testing: Past, present and future. In: 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops. pp. 153–163. IEEE (2011)
10. Miller, W., Spooner, D.L.: Automatic generation of floating-point test data. IEEE Transactions on Software Engineering (3), 223–226 (1976)
11. Panichella, A., Kifetew, F.M., Tonella, P.: Reformulating branch coverage as a many-objective optimization problem. In: 2015 IEEE 8th international conference on software testing, verification and validation (ICST). pp. 1–10. IEEE (2015)
12. Panichella, A., Kifetew, F.M., Tonella, P.: Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. IEEE Transactions on Software Engineering **44**(2), 122–158 (2017)
13. Pecorelli, F., Grano, G., Palomba, F., Gall, H.C., De Lucia, A.: Toward granular search-based automatic unit test case generation. Empirical Software Engineering **29**(4), 71 (2024)
14. Pressman, R.S.: Software Engineering: A Practitioner's Approach. McGraw-Hill, 7th edn. (2010)
15. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł., Polosukhin, I.: Attention is all you need. Advances in neural information processing systems **30** (2017)
16. Yuste, J., Pardo, E.G., Chicano, F.: Search-based software engineering problems. In: Handbook of Heuristics, pp. 1621–1649. Springer (2025)
17. Zhang, Q., Fang, C., Gu, S., Shang, Y., Chen, Z., Xiao, L.: Large language models for unit testing: A systematic literature review. arXiv preprint arXiv:2506.15227 (2025)