CMSA para el problema de la generación de casos de prueba priorizados en líneas de productos software

José Antonio Ortega Toro[1], Javier Ferrer[2], and Francisco Chicano[2]

[1] CERN, Suiza
joseaortegatoro@gmail.com
[2] Universidad de Málaga, España
{ferrer, chicano}@lcc.uma.es

Resumen En las líneas de producto software puede ser difícil o incluso imposible probar todos los productos de la familia debido al gran número de combinaciones de características que puede existir [1]. Esto conlleva la necesidad de buscar un subconjunto de productos de la familia que nos permita probar todas las posibles combinaciones. Los algoritmos del estado del arte basados en heurísticos junto con programación lineal entera (ILP) son lo bastante rápidos para instancias de tamaño pequeño o mediano. Sin embargo, existen algunas instancias del mundo real que son demasiado grandes para obtener una respuesta en un tiempo razonable, debido al crecimiento exponencial del espacio de búsqueda. Por otro lado, estos heurísticos no siempre conducen a las mejores soluciones. En este trabajo proponemos un nuevo enfoque basado en un algoritmo metaheurístico híbrido llamado *Construct, Merge, Solve & Adapt* (CMSA). Comparamos este enfoque con un algoritmo del estado del arte basado en programación entera y en algoritmos híbridos. El análisis muestra que el algoritmo propuesto conduce a soluciones de mayor calidad.

Keywords: Híbridos exactos/heurísticos, Aleatorización, Líneas de Productos Software, Optimización Combinatoria, Modelos de características, Testeo por pares.

1. Introducción

Las líneas de productos software, conocidas en inglés como *Software Product Lines* (SPLs) son usadas para conseguir un desarrollo software y un manejo de los productos software eficiente, reduciendo los costes y el tiempo de desarrollo, así como los gastos de mantenimiento [2].

Las líneas de productos se componen de productos con muchas características en común. Pero además de esto, poseen una gran variabilidad entre los productos de la misma familia. Esta variabilidad se debe a la personalización de los productos e implica un gran desafío cuando nos enfrentamos a la fase de pruebas de una línea de productos. Esta variabilidad implica una explosión combinatoria

en el número de productos [3]. Teniendo en cuenta estas dificultades, se han propuesto muchos enfoques [4,5], algunos de ellos basados en pruebas por pares (*pairwise testing*) [6,7,8], donde se requiere que todos los pares de características estén presentes en, al menos, un producto del conjunto de productos de prueba.

La imposibilidad de probar una línea de productos software completa en un tiempo razonable, ha dado lugar a un enfoque que usa prioridades para probar primero los productos más importantes atendiendo a un criterio. Estos pesos se pueden asignar a algunos productos de la línea para derivar los pesos de las características, o de forma inversa, se le asignan pesos a las características para calcular el peso de cada producto. El problema de optimización que queremos resolver consiste en encontrar el conjunto de productos mínimo que cubre la totalidad de pares de características con peso siguiendo un enfoque mono-objetivo.

Las últimas propuestas del estado del arte sobre pruebas por pares incluyen algoritmos híbridos, que mezclan heurísticas con algoritmos exactos [1]. Este método de pruebas y su métrica de cobertura se formalizan a través de modelos de características (*feature models*), donde los productos de una familia se definen como una combinación única de características [9].

La hipótesis científica para este trabajo es que un enfoque híbrido metaheurístico puede mejorar la calidad de las soluciones. El algoritmo que proponemos genera de forma probabilística soluciones del problema para combinarlas posteriormente y aplicar un algoritmo exacto que obtenga la mejor de las soluciones posibles a partir de la combinación. En particular, el algoritmo que utilizamos es *Construct, Merge, Solve, Adapt* (CMSA), propuesto por Blum et al. [10]. Comparamos los resultados con los de un algoritmo del estado del arte basado en programación entera no lineal, HINLP [1]. La comparación se realiza usando 16 modelos de características de líneas de productos software reales.

El resto del trabajo se organiza como sigue. En la Sección 2 introducimos el conocimiento de base necesario para comprender tanto el problema como el algoritmo que usamos en nuestro enfoque. En la Sección 3 explicamos el diseño e implementación de la adaptación del algoritmo CMSA al problema de la generación de productos para las pruebas por pares priorizadas. La Sección 4 discute los resultados de la implementación descrita, comparando los resultados con los de un algoritmo del estado del arte del problema. Para terminar, en la Sección 5 exponemos una serie de conclusiones sobre el trabajo presentado y presentamos las líneas de trabajo futuro más prometedoras.

2. Fundamentos

2.1. Modelos de Características

Los modelos de características son usados en las SPLs para definir la funcionalidad de un producto de una familia como una combinación de características individuales. Los modelos pueden también representar las restricciones que existen entre características. Una estructura de árbol jerarquizado se usa para caracterizar el modelo de características, donde los nodos del árbol representan

las características y los ejes representan las relaciones entre ellas. La Figura 1 representa el modelo *Graph Product Line* (GPL) [11] de un problema clásico en la evaluación de metodologías para líneas de productos.

Hay cuatro tipos de relaciones jerárquicas entre características:

- *Características opcionales:* son representadas con un círculo vacío que indica que la característica puede o no ser seleccionada si su padre es seleccionado. Un ejemplo en GPL es la característica `Weight`.
- *Características obligatorias:* son representadas con un círculo relleno, y deben ser seleccionadas si su padre es seleccionado. En nuestra instancia de ejemplo son obligatorias `Driver` y `Benchmark`.
- *Relaciones Or-inclusivas:* son representadas como arcos independientes rellenos que abarcan un conjunto de líneas que conectan la característica padre con las características hijas. Indican que al menos una característica debe ser seleccionada si el padre es seleccionado. En nuestra instancia de ejemplo, al menos una del grupo {*Num, CC, SCC, Cycle, Shortest, Prim* y *Kruskal*} debe ser seleccionada.
- *Relaciones Or-exclusivas:* son representadas como arcos independientes vacíos que abarcan un conjunto de líneas que conectan la característica padre con las características hijas. Indican que exactamente una característica debe ser seleccionada si el padre es seleccionado. En nuestra instancia de ejemplo la relación entre *Search*, *DFS* y *BFS* es de este tipo.

Las características se pueden relacionar también con otras ramas del modelo de características por medio de las llamadas *Cross-Tree Constraints (CTC)*. En nuestro ejemplo podemos observar estas restricciones debajo de la estructura de árbol. Una de ellas es "Num requires Search" que indica que cuando *Num* es seleccionada, entonces *Search* debe serlo también. Podemos deducir fácilmente que estas restricciones se pueden formalizar usando lógica proposicional.

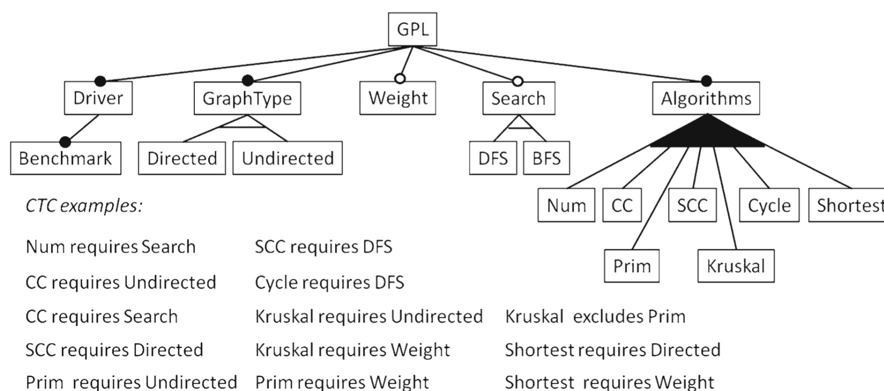

Figura 1: Modelo de características *Graph Product Line* (from Ferrer et al. [1]).

Una vez definidos los modelos de características, los usaremos con otros conceptos en la siguiente sección para formalizar el problema al que nos enfrentamos en este trabajo.

2.2. Formalización del Problema

Ahora presentamos la terminología usada en Pruebas de Interacción Combinatoria (*Combinatorial Interaction Testing - CIT)*. En este enfoque se construye un conjunto de muestras que nos permite probar las diferentes configuraciones del sistema [12]. Cuando aplicamos este enfoque a SPL, el conjunto de muestras es realmente un conjunto de productos de una familia. A continuación presentamos los conceptos que necesitamos para definir el problema de la generación de productos para pruebas por pares priorizadas:

Definición 1. ***Lista de características***. *Es el conjunto de características, FL, del modelo de características.*

Definición 2. ***Producto***. *Un producto se representa por un par* $(S, \bar{S})$*, donde* S *es un subconjunto de la lista de características FL,* $S \subseteq FL$*, y* $\bar{S} = FL - S$*.* S *representa el conjunto de características seleccionadas para el producto y* $\bar{S}$ *las no seleccionadas.*

Definición 3. ***Producto Válido***. *Decimos que un producto p es válido con respecto a un modelo de características FM si y solo si* $p.S$ *y* $p.\bar{S}$ *no violan ninguna de las restricciones definidas en el modelo de características. El conjunto de todos los productos válidos se denota por* P^{FM}*.*

Definición 4. ***Par***. *Un par pr es una tupla* $(s, \bar{s})$*, donde s y* $\bar{s}$ *representan diferentes características de la lista de características FL, es decir,* $pr.s, pr.\bar{s} \subseteq FL \ \wedge \ pr.s \ \cap \ pr.\bar{s} = \emptyset$*. Un par es cubierto por un producto p si, y sólo si* $pr.s \subseteq p.S \ \wedge \ pr.\bar{s} \subseteq p.\bar{S}$*.*

Definición 5. ***Par válido***. *Un par pr es válido en un modelo de características FM si existe un producto que cubra pr. El conjunto de todos los pares válidos de un modelo de características FM se denota por* VPR^{FM}*.*

Definición 6. ***Conjunto de pruebas***. *Un conjunto de pruebas ts para un modelo de características FM es un conjunto válido de productos de FM. Un conjunto de pruebas es completo si cubre todos los pares válidos en* VPR^{FM}*:* $\{ \ ts \mid \forall pr \in VPR^{FM}, \exists p \in ts \ \text{tal que} \ p \ \text{cubre} \ pr \ \}$

Definición 7. ***Producto priorizado***. *Un producto priorizado pp es una tupla* (p, w)*, donde p representa un producto válido en un modelo de características FM y* $w \in \mathbb{R}$ *representa su peso.*

Definición 8. ***Configuración***. *Una configuración c es una tupla* (pr, w) *donde pr es un par válido y* $w \in \mathbb{R}$ *representa su peso. w se calcula de la siguiente manera. Sea PP el conjunto de todos los productos priorizados y* PP_c *el subconjunto de PP, tal que* PP_c *contiene todos los productos priorizados de PP que cubren pr, tal que,* $PP_c = \{ \ p \in PP \mid p \ \text{cubre} \ c \ \}$*. Entonces* $w = \sum_{p \in PP_c} p.w$

Definición 9. ***Covering Array***. *Un* covering array *CA para un modelo de características FM y un conjunto de configuraciones C es un conjunto válido de productos P que cubre todas las configuraciones en C cuyos pesos son mayores a cero:* $\forall c \in C\ (c.w > 0 \rightarrow \exists p \in CA$ *tal que* p *cubre* $c.pr)$.

Definición 10. ***Cobertura***. *Dado un* covering array *CA y un conjunto de configuraciones C, definimos cov(CA) como la suma de todos los pesos de las configuraciones en C cubiertas por cualquier configuración en CA dividido por la suma de todos los pesos de las configuraciones en C, es decir:*

$$cov(CA) = \frac{\sum\limits_{c \in C,\ \exists p \in CA,\ p\ covers\ c.pr} c.w}{\sum\limits_{c \in C} c.w}$$

El problema de optimización que nos interesa consiste en encontrar un *covering array CA* con el mínimo número de productos $|CA|$ que con la restricción de que $cov(CA) = 1$ (cobertura total).

2.3. Construct, Merge, Solve & Adapt

Las metaheurísticas híbridas son técnicas que combinan las metaheurísticas con otras técnicas de optimización tales como programación dinámica, programación lineal entera (ILP), etc. El algoritmo CMSA es una metaheurística híbrida para optimización combinatoria presentada por Blum et al. [10]. Antes de describir el algoritmo presentaremos algunos conceptos.

Dado un problema P, sea C el conjunto de todos los posibles componentes de una solución para una instancia de un problema I. Una solución válida S de I se representa como un subconjunto de componentes de la solución: $S \subseteq C$. Una sub-instancia I' de I viene caracterizada por un subconjunto C' del conjunto de componentes de una solución: $C' \subseteq C$.

La forma de proceder del algoritmo es la siguiente (ver el Algoritmo 1). Mientras no se llega al tiempo limite establecido:

1. Se generan n_a sub-instancias de la instancia principal del problema I.
2. Todos los componentes que pertenecen a instancias previas son mezclados en una única sub-instancia, C'.
3. Un algoritmo exacto se aplica a la sub-instancia C', para obtener la mejor solución posible de la sub-instancia.
4. La solución se compara con la mejor obtenida hasta el momento y C' se actualiza con respecto a la política de envejecimiento, usualmente eliminando componentes de soluciones que no han sido usadas en soluciones óptimas durante un número prefijado de iteraciones.

El algoritmo tiene dos componentes clave que deben ser adaptados específicamente para el problema en cuestión:

- **Generador de sub-instancias**: Se basa en alguna estrategia probabilística para generar soluciones del problema inicial.
- **Un resolutor exacto para las sub-instancias**: Por ejemplo, basado en ILP u otro algoritmo exacto.

Algorithm 1: Construct, Merge, Solve & Adapt (CMSA)

```
1  input: Componentes del problema C, valores para los parámetros n_a y age_max
2  bestSolution := C
3  subInstance := ∅
4  age[c] := 0 for all c ∈ C
5  while Tiempo de CPU no alcanzado do
6  |  for t = 1, ..., n_a do
7  |  |  probSolution := ProbabilisticSolution(C)
8  |  |  for all x ∈ probSolution and x ∉ subInstance do
9  |  |  |  age[x] := 0
10 |  |  |  subInstance := subInstance ∪ {x}
11 |  |  end
12 |  end
13 |  probSolution := ExactSolver(subInstance)
14 |  if probSolution es mejor que bestSolution then
15 |  |  bestSolution := probSolution
16 |  end
17 |  Adapt(subInstance, probSolution, age_max)
18 end
19 output: bestSolution
```

3. CMSA para SPL

Para aplicar el algoritmo CMSA a nuestro problema necesitamos definir los tres métodos descritos en el Algoritmo 1: *ProbabilisticSolution*, *ExactSolver* y *Adapt*. Antes de eso, tenemos que plantear qué es un componente de la solución para nuestro problema. En este caso, dado que queremos minimizar la cardinalidad del conjunto de productos que cubre todas las posibles configuraciones del modelo de características, un componente de una solución sería un producto válido del modelo de características FM, siendo P el conjunto de todos los productos válidos de FM. Ahora pasamos a describir la idea de los tres componentes de CMSA que tenemos que adaptar.

3.1. Generación de sub-instancias

Al comienzo del algoritmo tenemos que generar diferentes subconjuntos de P (sub-instancias de I) que se mezclarán en la sub-instancia final que resolveremos luego. Seguimos tres estrategias diferentes para generar las sub-instancias:

- Enfoque *priorizado*: En este enfoque restringimos el conjunto de componentes del problema al de los productos priorizados. Este método restringe nuestro espacio de búsqueda, perdiendo algunas posibles soluciones cuya calidad se espera que sea baja.
- Enfoque *aleatorio*: En este caso consideramos todo el espacio de búsqueda P, generando productos aleatorios que son válidos para el modelo de ca-

racterísticas FM. Esta estrategia garantiza que encontraremos una solución mejor que la anterior, pero a costa de un rendimiento peor.

- Enfoque *híbrido*: Este último enfoque cambia entre los dos espacios de búsqueda anteriores para encontrar buenas soluciones. Se basa sobre todo en los productos priorizados pero introduce algunos del espacio de búsqueda completo P.

Como resultado, para el método *ProbabilisticSolution* tenemos entonces tres versiones, una por cada estrategia descrita anteriormente: priorizada, aleatoria e híbrida. El pseudocódigo de estos tres enfoques se describe en los Algoritmos 2, 3 y 4.

Algorithm 2: ProbabilisticSolutionGeneration (priorizado) (CMSA$_p$)

```
1  input: instancia problema I, conjunto de productos priorizados C
2  solution := ∅
3  available := C
4  uncovered := validPairs(I)
5  while uncovered no vacío do
6  |  x := selecciona aleatoriamente de available
7  |  solution := solution ∪ {x}
8  |  uncovered := uncovered − configurations(x)
9  |  available := {x ∈ available | configurations(x) ∩ uncovered ≠ ∅}
10 end
11 output: solution
```

Tanto en el enfoque aleatorio como en el híbrido necesitamos generar productos válidos de manera aleatoria. Para los enfoques aleatorio e híbrido hemos implementado un analizador del árbol de características para tener en cuenta todas las restricciones del modelo cuando validamos los productos generados aleatoriamente.

Esta generación se realiza por medio de un analizador del árbol de características que tiene en cuenta las restricciones estructurales (no las CTC) del modelo. Esta generación se realiza en dos pasos:

1. Se genera un producto seleccionando nodos del modelo. Partiendo desde la característica o nodo raíz, se decide aleatoriamente si incluir a cada uno de los nodos hijos recursivamente, teniendo en cuenta la relación jerárquica que existe entre los dos nodos.
   Por ejemplo, si un nodo tiene dos hijos con una relación del tipo or-exclusivo, se elegirá solo a uno de los dos. Si la relación es or-inclusiva, se seleccionará uno o más de uno de sus nodos. De esta manera nos aseguramos que el producto generado es coherente con la estructura del modelo de características.
2. El producto generado se valida frente a las CTC, asegurándonos de que no viola ninguna de las restricciones.

En modelos de características con muchas restricciones del tipo CTC, éstas pueden dar lugar a un mayor tiempo de ejecución cuando se generan productos aleatorios, ya que en ocasiones los productos generados pueden no cumplir con alguna de las CTC.

Algorithm 3: ProbabilisticSolutionGeneration (aleatorio) (CMSA$_r$)

```
1  input: instancia problema I
2  solution := ∅
3  uncovered := validPairs(I)
4  while uncovered no vacío do
5  |  x := generar producto aleatorio válido
6  |  if configurations(x) ∩ uncovered ≠ ∅ then
7  |  |  solution := solution ∪ {x}
8  |  |  uncovered := uncovered − configurations(x)
9  |  end
10 end
11 output: solution
```

Algorithm 4: ProbabilisticSolutionGeneration (híbrido) (CMSA$_h$)

```
1  input: instancia problema I, conjunto de productos priorizados C, tasa
     determinista d_rate
2  solution := ∅
3  available := C
4  uncovered := validPairs(I)
5  while uncovered no vacío do
6  |  Elige un número aleatorio δ ∈ [0, 1]
7  |  if δ ≤ d_rate then
8  |  |  x := selecciona aleatoriamente de available
9  |  else
10 |  |  do
11 |  |  |  x := generar producto aleatorio válido
12 |  |  while not (configurations(x) ∩ uncovered ≠ ∅);
13 |  end
14 |  solution := solution ∪ {x}
15 |  uncovered := uncovered − configurations(x)
16 |  available := {x ∈ available | configurations(x) ∩ uncovered ≠ ∅}
17 end
18 output: solution
```

Nótese que el hecho de que no haya configuraciones no cubiertas en cada subinstancia generada nos asegura que seremos siempre capaces de cubrir todas las configuraciones ponderadas cuando resolvamos la instancia final. Con respecto a las otras dos partes del algoritmo, es común para estos tres enfoques.

3.2. Sub-instancias

Después de generar las sub-instancias y mezclarlas en la sub-instancia final, usamos un algoritmo exacto para determinar el número de productos que necesitamos para cubrir todas las configuraciones ponderadas del modelo de características.

En este caso, usamos un modelo de programación lineal entera para seleccionar un subconjunto de los productos de la sub-instancia que, cubriendo todos los pares con pesos, tiene cardinalidad mínima. Este problema es equivalente al problema de minimización del conjunto de casos de prueba (*test suite minimization*), en su versión mono-objetivo [13].

3.3. Adaptando la sub-instancia

El mecanismo de envejecimiento funciona como el propuesto por Blum et al. [10]. En cada iteración del algoritmo, tenemos un subconjunto de componentes del problema C que se resuelven usando el método *ExactSolver*. Este método devuelve un subconjunto de C, llamémoslo S. Entonces, la edad de todos los componentes que son parte de la solución S son inicializadas a 0, mientras que el resto de componentes incrementan su edad en 1. Si algún componente alcanza el máximo de edad, entonces se elimina de C. En Algoritmo 5 mostramos el pseudocódigo del método *Adapt*.

Algorithm 5: Adapt

```
1  input: sub-instancia C, sub-instancia solución S
2  for x in S do
3  |  age[x] := 0
4  end
5  for x in C − S do
6  |  age[x] := age[x] + 1
7  |  if age[x] = age_max then
8  |  |  C := C − {x}
9  |  end
10 end
11 output: C
```

4. Experimentos

En esta sección describimos como se ha realizado la evaluación de nuestra propuesta. Primero, se presenta el algoritmo HINLP, objeto de la comparación. Después, describimos el *benchmark* usado para la evaluación. Finalmente explicamos la configuración de los experimentos y los resultados obtenidos.

4.1. Algoritmo híbrido basado en programación entera no lineal (HINLP)

Este algoritmo, propuesto por Ferrer et al. [1], es un algoritmo híbrido que combina una heurística ávida con un algoritmo exacto basado en programación entera no lineal. En [1], este algoritmo fue comparado con otros dos: *Prioritized Pairwise Genetic Solver* (PPGS) [14] y *Prioritized-ICPL* [15]. Se concluía que HINLP funciona mejor en calidad de las soluciones y tiempo de computación.

4.2. Benchmark

Los modelos de características que usamos en la comparación son 16 SPLs realistas, calculando las prioridades sobre características no funcionales medidas con SPL Conqueror [16]. La Tabla 1 describe los 16 modelos: nombre, numero de características, número de productos, configuraciones y porcentaje de productos que están ponderados.

Model name	**FN**	**PN**	**CN**	**PP %**
Apache	10	256	192	75,0
BerkeleyDBFootprint	9	256	256	100,0
BerkeleyDBMemory	19	3840	1280	33,3
BerkeleyDBPerformance	27	1440	180	12,50
Curl	14	1024	68	6,6
LinkedList	26	1440	204	14,1
Linux	25	$\approx 3E7$	100	$\approx 0,0$
LLVM	12	1024	53	5,1
PKJab	12	72	72	100,0
Prevayler	6	32	24	75,0
SensorNetwork	27	16704	3240	19,4
SQLiteMemory	40	$\approx 5E7$	418	$\approx 0,0$
Violet	101	$\approx 1E20$	101	$\approx 0,0$
Wget	17	8192	94	1,15
x264	17	2048	77	3,7
ZipMe	8	64	64	100,0

Tabla 1: Modelos de características usados.

Después de describir el algoritmo HINLP, podemos inferir algunas conclusiones prestando atención a los modelos. Por una parte, en los modelos en los que el porcentaje de productos que ponderamos es del 100 %, aplicar la heurística a todos los componentes de la solución P (sin generar sub-instancias) debe proporcionar resultados similares a HINLP. Por otra parte, esperamos que la versión priorizada de CMSA ($CMSA_p$) funcione peor que la aleatoria e híbrida ($CMSA_r$) y ($CMSA_h$) en los modelos con bajo porcentaje de productos ponderados *(Linux, Violet, SQLiteMemory)* debido al enorme espacio de búsqueda que se descarta restringiendo la búsqueda a productos priorizados. Aunque se puede dar la circunstancia de que si la solución óptima es una combinación de los productos ponderados, encuentre esta solución más fácilmente por la misma razón.

4.3. Configuración de los experimentos

El algoritmo HINLP se ha implementado en Java y CMSA en Python. Los experimentos se han ejecutado en un clúster con máquinas Intel Core 2 Quad Q9400 de 4 núcleos por procesador, sistema operativo Ubuntu 16.04 LTS, Java 1.8 y Python 2.6. Como HINLP es determinista, sólo necesitamos una ejecución del algoritmo. Para el algoritmo CMSA, hemos realizado experimentos preliminares con varias configuraciones de sub-instancias por iteración (n_a) y tiempo máximo para la política de envejecimiento (age_{max}). Como resultado escogimos la mejor configuración que es n_a=5 y $age_{max} = 4$. Para cada estrategia de generación de soluciones y cada instancia del problema realizamos un total de 30 ejecuciones. El parámetro d_{rate} del enfoque híbrido se fijó a 0,15.

Con respecto al número de iteraciones de CMSA, el algoritmo realiza al menos una. Si el tiempo de una iteración es menor a tres segundos, el algoritmo se detiene después de terminar la iteración que pasa de los tres segundos desde el inicio de la ejecución. Para la comparación de las soluciones se ha utilizado el test *Wilcoxon rank-sum* para comparación por pares con un nivel de significación de 0,05 y corrección de Bonferroni.

4.4. Resultados

La Tabla 2 resume los resultados de la ejecución de HINLP y las versiones del algoritmo CMSA para cada uno de los modelos para cobertura 100 %. El tiempo medio de ejecución se muestra en segundos y la desviación estándar se muestra como subíndice en aquellos casos en que no es cero (porque el algoritmo es HINLP, que es determinista, o se obtiene el mismo resultado en todas las ejecuciones independientes).

En la Tabla 2 podemos apreciar como el algoritmo HINLP es claramente el más rápido, gracias a su estrategia *greedy*. Por el contrario, en calidad de las soluciones encontramos que tan sólo es capaz de obtener el mejor resultado en un modelo, empatando con los otros tres algoritmos. Además, estadísticamente hay diferencias significativas entre los demás algoritmos y HINLP en 14 de 16 modelos, la mayoría de las ocasiones siendo su resultado claramente peor que el de las otras propuestas.

Analizando las diferentes versiones de CMSA, podemos observar diferencias significativas entre la versión priorizada y alguna de las otras dos versiones de CMSA en seis modelos: Curl, Violet, LinkedList, LLVM, SQLiteMemory y Wget. Además, $CMSA_h$ y $CMSA_r$ alcanzan el mejor resultado en 13 de 16 modelos, mientras que $CMSA_p$ lo alcanza en 7 de 16 modelos. En cuanto a tiempo de ejecución, podemos concluir que $CSMA_p$ sólo es competitivo con modelos grandes donde aprovecha que su espacio de búsqueda se restringe a los productos priorizados generados. Además, la heurística utilizada por HINLP funciona bien en las instancias grandes, como en el caso del modelo Violet donde obtiene mejores resultados que $CMSA_p$ y $CMSA_r$.

En la Figura 2 mostramos un boxplot con la comparativa de los resultados en cuanto a calidad de las soluciones obtenidas por los cuatro algoritmos

Modelo	HINLP		$CMSA_p$		$CMSA_r$		$CMSA_h$	
	#Prod.	Tiempo	#Prod.	Tiempo	#Prod.	Tiempo	#Prod.	Tiempo
Apache	7	0,66	6	45,30	6	4,38	6	4,84
BerkeleyDBFootprint	8	0,65	6	46,47	6	4,61	6	4,81
BerkeleyDBMemory	21	1,34	20	8,34	20	4,11	20	3,62
BerkeleyDBPerformance	10	0,94	9	33,35	9	3,50	9	3,34
Curl	9	0,73	$8,37_{0,48}$	43,10	8	8,20	8	10,64
LinkedList	13	1,06	15	3,85	11	4,78	11	3,74
Linux	11	4,42	$10,07_{0,25}$	157,31	10	447,82	10	516,01
LLVM	10	0,75	7	45,21	6	5,32	$6,07_{0,25}$	7,59
PKJab	7	0,60	6	45,09	6	3,15	6	3,18
Prevayler	6	0,49	6	45,54	6	3,18	6	3,21
SensorNetwork	14	1,40	$10,43_{0,62}$	48,82	$10,20_{0,40}$	4,74	$10,23_{0,42}$	4,77
SQLiteMemory	27	3,40	$30,83_{0,69}$	37,09	$23,03_{0,60}$	18,65	$22,57_{0,62}$	20,96
Violet	12	3,43	16	3,78	$13,80_{0,54}$	443,32	$11,17_{0,58}$	313,74
Wget	12	0,81	$10,50_{0,50}$	46,97	$9,13_{0,34}$	29,68	$9,90_{0,30}$	44,99
x264	12	0,78	$9,13_{0,34}$	37,47	$9,97_{0,60}$	74,18	9	3,81
ZipMe	7	0,58	6	45,64	6	45,82	6	46,08
Media	11,63	1,38	11,02	43,33	10,01	69,09	9,81	61,90

Tabla 2: Media y desviación estándar del número de productos para el *benchmark* propuesto. Se marca el mejor resultado para cada modelo.

considerando los resultados de los 16 modelos. Este gráfico vuelve a poner de manifiesto que HINLP es globalmente el peor algoritmo con una mediana de 11 productos. Además, también muestra que $CMSA_p$ es el algoritmo con mayor rango intercuartílico y con *outliers* muy por encima de los resultados de las demás propuestas. Con lo cual, sabemos que en algunas ejecuciones independientes $CMSA_p$ obtiene peores resultados que los obtenidos por los otros tres algoritmos.

Observando los resultados de las versiones aleatoria e híbrida, podemos decir que los resultados son casi los mismos, como se podía esperar. Entre ellos sólo hay diferencias significativas en el modelo Violet y x264, donde $CMSA_h$ obtiene mejores resultados y en Wget donde $CMSA_r$ es mejor. De cualquier forma, con un número de ejecuciones significativamente mayor los dos enfoques deben converger a la misma solución, por el hecho de que la aleatorización puede llegar a cubrir todo el espacio de búsqueda.

5. Conclusiones

En este trabajo hemos presentado un nuevo enfoque para resolver el problema de la generación de casos de prueba priorizados, con el objetivo de aliviar las tareas de pruebas en grandes líneas de productos software. Nuestra principal contribución es la adaptación del algoritmo CMSA al problema de la generación de casos de prueba priorizados para SPL.

Presentamos tres enfoques o estrategias diferentes del algoritmo CMSA (priorizada, aleatoria e híbrida) a la hora de resolver el problema. La estrategia priorizada ha permitido al algoritmo realizar más iteraciones, debido a que la búsqueda se restringe al conjunto de productos ponderados. El algoritmo se ahorra el

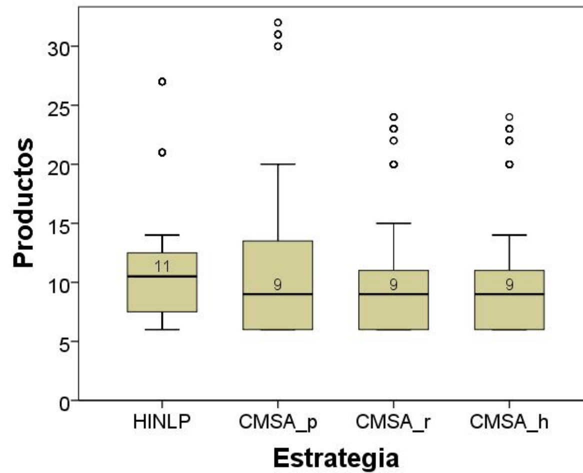

Figura 2: Comparación de los algoritmos según el número de productos obtenido.

tiempo de la generación de nuevos productos válidos, que puede ser un proceso muy costoso en tiempo, dependiendo de las restricciones del modelo. Este comportamiento es beneficioso si la solución óptima se encuentra en el conjunto de productos ponderados. Por otro lado, las otras dos estrategias se basan en una generación aleatoria de productos, que implica un mayor cómputo para obtener un nuevo producto válido, pero por contra, son capaces de introducir más variabilidad en la búsqueda, que finalmente ha sido muy beneficioso para la obtención de mejores resultados.

En general, los resultados de las variantes de CMSA muestran una mejora en términos de la calidad de la solución con respecto a HINLP. De hecho, la mejor estrategia ($CMSA_h$) siempre consigue resultados iguales o mejores que HINLP, por lo que podemos decir que CMSA es claramente superior a la estrategia greedy (HINLP) objeto de la comparación en calidad de solución.

No podemos olvidar que HINLP ha resultado ser muy superior en tiempo de ejecución, por lo que una línea de trabajo futuro es un análisis en profundidad de métodos eficientes para la generación de productos aleatorios válidos. La mejora de este procedimiento puede redundar positivamente en un decremento del tiempo de ejecución de las estrategias aleatoria e híbrida.

Agradecimientos

Esta investigación ha sido parcialmente financiada por CELTIC C2017/2-2 en colaboración con las empresas EMERGYA y SECMOTIC en los contratos #8.06/5.47.4997 y #8.06/5.47.4996. También agradecemos el apoyo del Ministerio de Economía y Competitividad y de los fondos FEDER, proyectos: TIN2014-57341-R (http://moveon.lcc.uma.es), TIN2016-81766-REDT (http://cirti.es) y TIN2017-88213-R (http://6city.lcc.uma.es), y a la Universidad de Málaga.

Referencias

1. Ferrer, J., Chicano, F., Alba, E.: Hybrid algorithms based on integer programming for the search of prioritized test data in software product lines. In Squillero, G., Sim, K., eds.: EvoApps 2017, LNCS 10200, The Netherlands, Springer (2017) 3–19
2. Pohl, K., Böckle, G., van Der Linden, F.J.: Software product line engineering: foundations, principles and techniques. Springer Science & Business Media (2005)
3. Engström, E., Runeson, P.: Software product line testing – a systematic mapping study. Information and Software Technology **53**(1) (2011) 2 – 13
4. Al-Hajjaji, M., Thüm, T., Meinicke, J., Lochau, M., Saake, G.: Similarity-based prioritization in software product-line testing. In: Proceedings of the 18th International Software Product Line Conference - Volume 1. SPLC '14, New York, NY, USA, ACM (2014) 197–206
5. Cohen, M.B., Dwyer, M.B., Shi, J.: Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach. IEEE Transactions on Software Engineering **34**(5) (Sept 2008) 633–650
6. Perrouin, G., Oster, S., Sen, S., Klein, J., Baudry, B., Le Traon, Y.: Pairwise testing for software product lines: comparison of two approaches. Software Quality Journal **20**(3-4) (2012) 605–643
7. Oster, S., Markert, F., Ritter, P. In: Automated Incremental Pairwise Testing of Software Product Lines. Springer Berlin Heidelberg, Berlin, Heidelberg (2010) 196–210
8. Lopez-Herrejon, R.E., Chicano, F., Ferrer, J., Egyed, A., Alba, E.: Multi-objective optimal test suite computation for software product line pairwise testing. In: Software Maintenance (ICSM), 2013 29th IEEE International Conference on, IEEE (2013) 404–407
9. Batory, D. In: Feature Models, Grammars, and Propositional Formulas. Springer Berlin Heidelberg, Berlin, Heidelberg (2005) 7–20
10. Blum, C., Pinacho, P., López-Ibáñez, M., Lozano, J.A.: Construct, merge, solve & adapt a new general algorithm for combinatorial optimization. Comput. Oper. Res. **68**(C) (April 2016) 75–88
11. Lopez-Herrejon, R.E., Batory, D.: A standard problem for evaluating product-line methodologies. In: International Symposium on Generative and Component-Based Software Engineering, Springer (2001) 10–24
12. Nie, C., Leung, H.: A survey of combinatorial testing. ACM Comput. Surv. **43**(2) (February 2011) 11:1–11:29
13. Arito, F., Chicano, F., Alba, E.: On the application of sat solvers to the test suite minimization problem. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) **7515 LNCS** (2012) 45–59
14. Lopez-Herrejon, R.E., Javier Ferrer, J., Chicano, F., Haslinger, E.N., Egyed, A., Alba, E.: A parallel evolutionary algorithm for prioritized pairwise testing of software product lines. In: Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation, ACM (2014) 1255–1262
15. Johansen, M., Haugen, Ø., Fleurey, F., Eldegard, A., Syversen, T.: Generating better partial covering arrays by modeling weights on sub-product lines. Model Driven Engineering Languages and Systems (2012) 269–284
16. Siegmund, N., Rosenmüller, M., Kästner, C., Giarrusso, P.G., Apel, S., Kolesnikov, S.S.: Scalable prediction of non-functional properties in software product lines: Footprint and memory consumption. Information and Software Technology **55**(3) (2013) 491 – 507 Special Issue on Software Reuse and Product Lines.