Generación automática de datos de prueba mediante un enfoque que combina Búsqueda Dispersa y Búsqueda Local

Raquel Blanco[1], Javier Tuya[1] y Belarmino Adenso-Díaz[2]

[1] Departamento de Informática
[2] Departamento de Administración de Empresas
Universidad de Oviedo
Campus de Viesques s/n, Gijón, Asturias, 33204, España
{rblanco, tuya, adenso}@uniovi.es

Resumen. Las técnicas empleadas en la generación automática de datos de prueba tratan de encontrar de forma eficiente un conjunto pequeño de datos de prueba que permitan satisfacer un determinado criterio de suficiencia, contribuyendo así a la reducción del coste de la prueba del software. En este artículo se presenta un enfoque basado en la técnica Búsqueda Dispersa denominado TCSS-LS. Este generador combina la propiedad de diversidad de la Búsqueda Dispersa con la intensificación de una Búsqueda Local. La diversidad es empleada para extender la búsqueda de datos de prueba a todas las ramas de un programa bajo prueba con el fin de generar datos de prueba que las cubran, mientras que la búsqueda local intensifica la búsqueda en determinados puntos para los cuales la diversificación encuentra dificultades. Además se presentan los resultados comparativos de TCSS-LS frente a un enfoque basado únicamente en Búsqueda Dispersa denominado TCSS.

Palabras Clave: Prueba software, generación automática de datos de prueba, cobertura de ramas, Búsqueda Dispersa, técnicas de búsqueda metaheurísticas.

1 Introducción

La prueba del software es la fase del ciclo de vida de un desarrollo software cuyo objetivo es descubrir los fallos que se han producido en el resto de fases, de tal forma que puedan ser solucionados, mejorando así la calidad del producto y disminuyendo el coste derivado de dichos de fallos [25]. Sin embargo esta fase es muy costosa y se estima que suele consumir entre un 30% [15] y un 50% [4] como mínimo del coste total de un desarrollo software.

Una parte de esta fase que conlleva una labor intensiva es la generación de los datos de prueba utilizados en la construcción de los casos de prueba del producto software. Esta tarea es crucial para el éxito de la prueba, debido a que es imposible obtener un programa software completamente probado (el número de casos de prueba necesarios para probar un programa software es infinito [25]) y un diseño adecuado de los casos de prueba podrá detectar un elevado número de fallos. Además la creación de los datos de prueba, al ser principalmente manual en la actualidad, es tal vez la tarea más costosa de la prueba del software, ya que puede suponer aproximadamente el 40% de su coste total [35]. Por estas razones, los diversos métodos desarrollados para la generación automática de los datos de prueba tratan de encontrar de forma eficiente un conjunto pequeño de dichos datos que permitan satisfacer un determinado criterio de suficiencia. De esta forma se reduce el coste de la prueba del software [12][27][35], por lo que los productos software pueden ser probados más eficientemente.

Entre las técnicas más recientes que realizan esta automatización se encuentran las técnicas de búsqueda metaheurísticas. La aplicación de los algoritmos metaheurísticos a la resolución de

problemas en Ingeniería del Software fue propuesto por la red SEMINAL (Software Engineering using Metaheuristic INnovative ALgorithms) y se trata ampliamente en [10]. Una de esas aplicaciones es la selección de datos de prueba en el proceso de la prueba del software, la cual es tratada como un problema de búsqueda u optimización, como se muestra en varios trabajos que llevan a cabo una revisión de la literatura [20][21].

La técnica metaheurística más ampliamente utilizada en este campo son los Algoritmos Genéticos. Esta técnica ha sido empleada en diversos trabajos para generar datos de prueba bajo criterios de suficiencia como cobertura de ramas [3][16][24][27][29][34], cobertura de condición [2], cobertura de condición-decisión [23][35], cobertura de caminos [1][7][18][19] [33] o criterio all-uses [13]. Otras técnicas que también han sido empleadas en la generación automática de datos de prueba son la Programación Genética [31], el Recocido Simulado [19][30][32][35], la Búsqueda Tabú [11], los Algoritmos Evolutivos [9][22], la Repulsión Simulada [8] o la Búsqueda Dispersa [6][28].

En este artículo se presenta un algoritmo basado en Búsqueda Dispersa denominado TCSS-LS para la generación automática de datos de prueba bajo el criterio de suficiencia de cobertura de ramas. Para ello TCSS-LS combina la propiedad de diversidad de la Búsqueda Dispersa con la intensificación de una Búsqueda Local. La diversidad es empleada para extender la búsqueda de datos de prueba a todas las ramas de un programa bajo prueba con el fin de generar datos de prueba que las cubran, mientras que la búsqueda local intensifica la búsqueda en determinados puntos para los cuales la diversificación encuentra dificultades. Este algoritmo es una extensión del algoritmo basado en Búsqueda Dispersa denominado TCSS [5][6].

2 Búsqueda Dispersa

La Búsqueda Dispersa (Scatter Search) [14][17] es un método evolutivo que opera sobre un conjunto de soluciones, llamado Conjunto de Referencia (RefSet). Las soluciones presentes en este conjunto son combinadas con el fin de generar nuevas soluciones que mejoren a las originales. Así, el Conjunto de Referencia almacena las mejores soluciones que se encuentran durante el proceso de búsqueda, considerando para ello su calidad y la diversidad que aportan al mismo.

El funcionamiento general de la Búsqueda Dispersa consiste en la generación de un conjunto P de soluciones diversas, que serán utilizadas para crear el Conjunto de Referencia sobre el que se trabaja. De ese conjunto se seleccionarán una serie de soluciones para realizar las combinaciones que dan lugar a las nuevas soluciones, las cuales serán evaluadas de acuerdo a una función objetivo para determinar si mejoran a algunas de las presentes en el Conjunto de Referencia. En caso afirmativo el Conjunto de Referencia será actualizado, incluyendo esas nuevas soluciones y eliminando las peores. De esta forma la solución final al problema planteado estará almacenada en el Conjunto de Referencia. El algoritmo de Búsqueda Dispersa finaliza su ejecución cuando dicho conjunto no es actualizado.

3 La búsqueda de datos de prueba mediante Búsqueda Dispersa

En esta sección se describe la aplicación de la técnica metaheurística Búsqueda Dispersa a la generación automática de datos de prueba y su combinación con un método de Búsqueda Local. Esta combinación ha dado lugar al algoritmo generador de datos de prueba denominado TCSS-LS.

3.1 Planteamiento del problema

Durante el proceso de búsqueda de datos de prueba bajo el criterio de suficiencia de cobertura de ramas, TCSS-LS necesita conocer las ramas que han sido cubiertas por los datos de prueba

previamente generados y las que aún no lo han sido. Además necesita almacenar una serie de información que le sirva de base para la generación de nuevos datos de prueba que permitan aumentar la cobertura de ramas. Para ello utiliza el grafo de control de flujo asociado al programa software bajo prueba (SUT: Software Under Test), donde cada requisito de prueba a cumplir se representa mediante un nodo, es decir, cada evaluación cierta o falsa de una decisión del SUT dará lugar a un nodo. Con este grafo es posible determinar qué ramas han sido cubiertas debido a que el SUT es instrumentado para recordar el camino seguido por cada dato de prueba ejecutado en él.

Mediante el uso del grafo de control de flujo, el objetivo general de obtener datos de prueba que cubran todas las ramas del SUT se puede dividir en subobjetivos, consistiendo cada uno de ellos en encontrar datos de prueba que alcancen un determinado nodo del grafo de control de flujo.

Para alcanzar dichos subobjetivos, los nodos almacenan determinada información durante todo el proceso de generación de datos de prueba, la cual permite mantener el conocimiento de las ramas cubiertas y es utilizada para avanzar en el proceso de búsqueda. Cada nodo almacena esta información en un conjunto de soluciones, llamado Conjunto de Referencia o RefSet. A diferencia del algoritmo de Búsqueda Dispersa general, TCSS-LS trabajan con varios Conjuntos de Referencia. Cada uno de estos conjuntos se denomina S_k (donde k es el número de nodo) y contiene varios elementos $T_k^c = < \overline{x}_k^c, p_k^c, fb_k^c, fc_k^c > c \in \{1, 2, \ldots, B_k\}$, donde $\overline{x}_k^c$ es una solución (un dato de prueba) que alcanza el nodo k y está formada por los valores de las variables de entrada que hacen ciertas las decisiones de los nodos del camino que llega al nodo k , p_k^c es el camino recorrido por la solución, fb_k^c es la distancia que indica lo cerca que dicha solución está de pasar por su nodo hermano y fc_k^c es la distancia que indica lo cerca que está la solución de pasar por el nodo hijo que no ha sido alcanzado por dicha solución.

Las distancias son calculadas utilizando las decisiones de entrada a los nodos no alcanzados durante la ejecución de la solución en el SUT, es decir, decisiones que se evalúan a falso. Las funciones empleadas para su cálculo pueden consultarse en [11].

La estructura del grafo de control de flujo del SUT con la información almacenada en los Conjuntos de Referencia S_k puede verse representada en la Fig. 1.

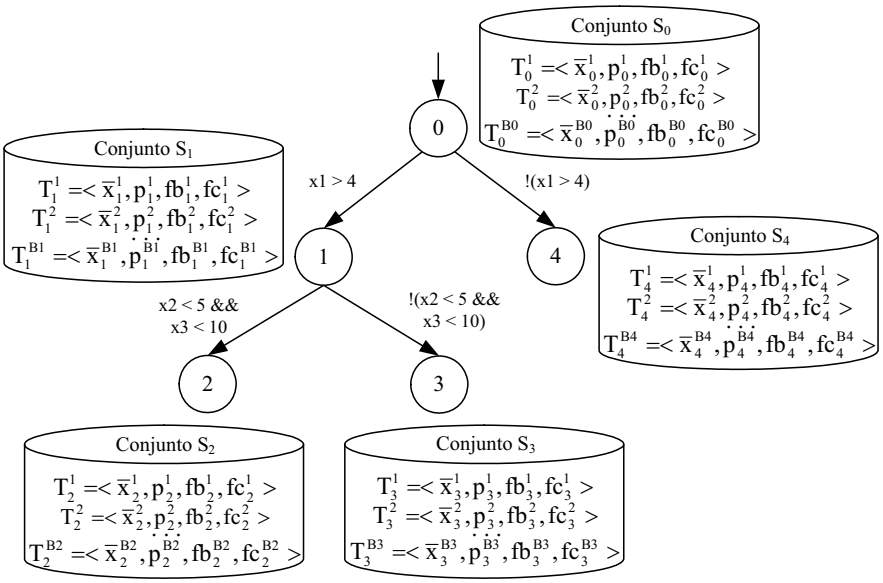

Fig. 1. Grafo de Control de Flujo de TCSS-LS

El conjunto de soluciones de un nodo n_k (S_k) tiene un tamaño máximo B_k. Este tamaño es diferente para cada nodo n_k y depende de la complejidad del código fuente situado por debajo del nodo n_k, de forma que las nuevas soluciones construidas a partir de las almacenadas en el conjunto

S_k puedan alcanzar los nodos que representan dicho código fuente. La forma en la que se determina cada tamaño B_k se puede consultar en [5].

TCSS-LS tratará de hacer los conjuntos S_k lo más diversos posibles para generar soluciones que puedan cubrir distintas ramas del programa.

3.2 Proceso de búsqueda

Como se ha comentado previamente, el objetivo de TCSS-LS es obtener la máxima cobertura de ramas, por lo que deben encontrarse soluciones (datos de prueba) que permitan cubrir todos los nodos del grafo de control de flujo. Puesto que dichas soluciones se almacenan en los nodos, el objetivo de TCSS-LS es, por tanto, que todos los nodos tengan al menos un elemento en su conjunto S_k. Sin embargo, este objetivo no puede ser alcanzado cuando el SUT posee ramas inalcanzables. Por ese motivo TCSS-LS también finaliza la búsqueda tras la generación de un número máximo de datos de prueba. Inicialmente los conjuntos de soluciones S_k están vacíos y serán rellenados en las sucesivas iteraciones.

En la Fig. 2 se muestra gráficamente el esquema del proceso de búsqueda de TCSS-LS. El proceso de búsqueda comienza con la generación de las soluciones aleatorias que serán almacenadas en el Conjunto de Referencia del nodo raíz (S_0), el cual actúa como el conjunto P del algoritmo estándar de Búsqueda Dispersa. Cada solución es ejecutada sobre el SUT, lo que da lugar al recorrido de un cierto camino. A continuación los Conjuntos de Referencia S_k (o RefSet) de los nodos pertenecientes al camino recorrido serán actualizados con aquellas soluciones que los alcancen, por lo que dichas soluciones además de ser diversas presentan un cierto grado de calidad. A partir de este momento comienzan las iteraciones del proceso de búsqueda, en las cuales se debe seleccionar un nodo del grafo de control de flujo (nodo en evaluación) para crear los subconjuntos de soluciones de su Conjunto de Referencia, que serán utilizados por las reglas de combinación para generar las nuevas soluciones. Estas nuevas soluciones, una vez han pasado por un proceso de mejora, son también ejecutadas sobre el SUT para realizar la posterior actualización de los Conjuntos de Referencia S_k de los nodos alcanzados, cerrando así el ciclo de ejecución.

Cuando el Conjunto de referencia del nodo seleccionado no posee suficientes soluciones para continuar con el proceso de búsqueda es necesario realizar un proceso de backtracking para encontrar otro conjunto más apropiado con el que trabajar. Si el proceso de backtracking llega a su fin, sin conseguir el objetivo perseguido, se lleva a cabo una regeneración de los Conjuntos de Referencia que se han visto afectados por dicho backtracking.

El final del proceso de búsqueda viene determinado por el cumplimiento de alguno de los criterios anteriormente expuestos: todos los nodos has sido alcanzados o se ha generado un número máximo de datos de prueba permitido.

La solución final de TCSS-LS está compuesta por los datos de prueba que cubren las ramas del SUT, los cuales están almacenados en los conjuntos S_k de los nodos del grafo de control de flujo, así como el porcentaje de cobertura de ramas alcanzado y el tiempo consumido en el proceso de búsqueda.

Los criterios utilizados en la selección del nodo del grafo de control empleado en cada iteración, la formación de los subconjuntos de soluciones a partir de un conjunto S_k, las reglas de combinación empleada para generar las nuevas soluciones, el proceso de mejora y los criterios de actualización de los conjuntos S_k pueden consultarse en [5][6].

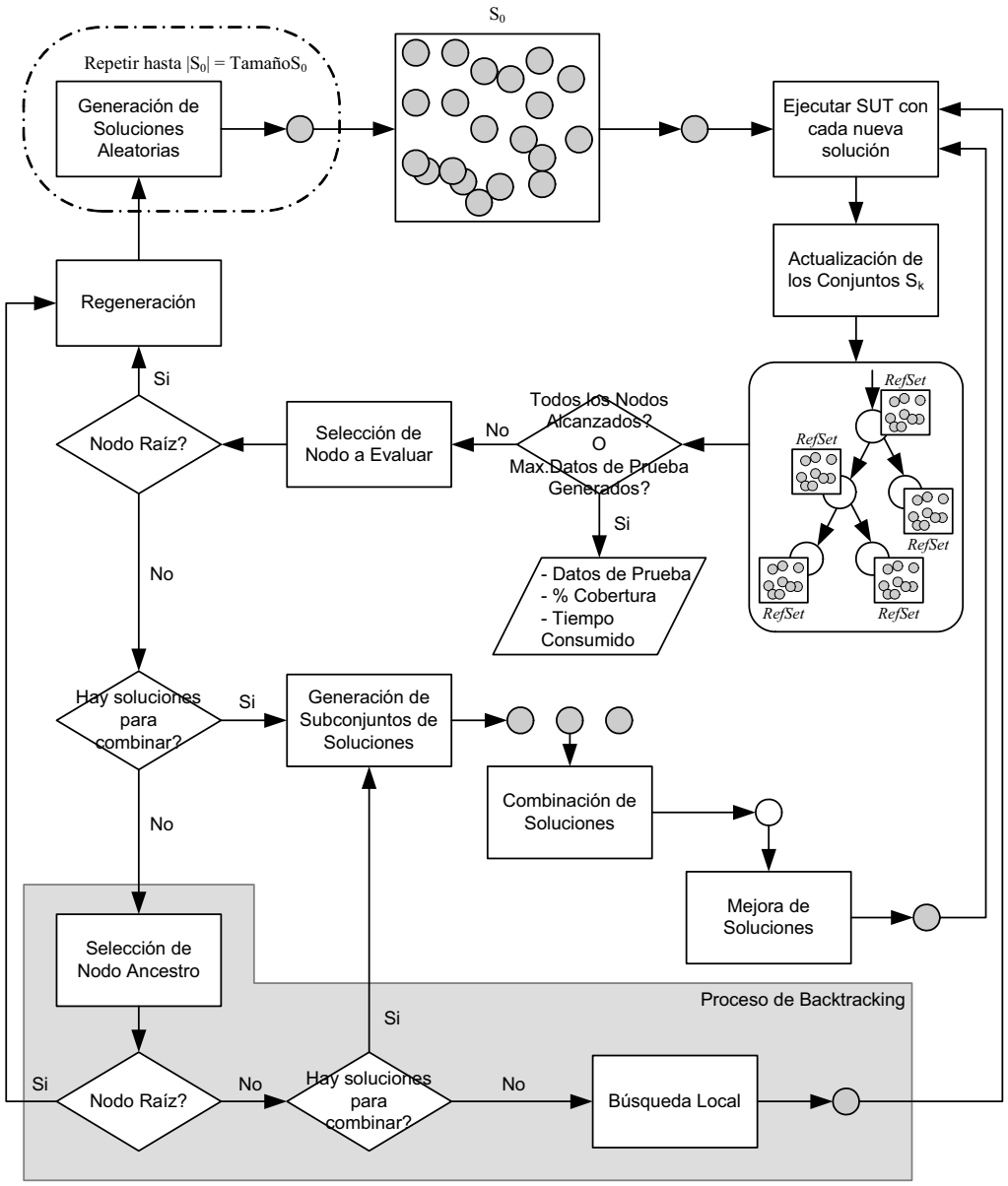

Fig. 2. Esquema de funcionamiento de TCSS-LS

3.3 El proceso de backtracking

La mayor dificultad del proceso de búsqueda surge cuando el nodo n_k en evaluación no posee al menos dos soluciones en su conjunto S_k que permitan realizar las combinaciones que dan lugar a las nuevas soluciones. Esto sucede cuando no existe ningún dato de prueba que cubra el nodo n_k (o sólo uno fue capaz de alcanzarlo) y además todos los demás candidatos están en la misma situación. Por tanto TCSS-LS no puede avanzar en el proceso de búsqueda y en consecuencia se debe realizar un proceso de backtracking. Este proceso trata de incrementar el tamaño del conjunto S_k del nodo n_k por medio de la generación de nuevas soluciones utilizando el nodo padre.

Para generar las nuevas soluciones durante el proceso de backtracking TCSS-LS tiene dos opciones:

a. Realizar combinaciones: esta opción se lleva a cabo cuando el nodo padre posee en su conjunto S_k soluciones que no han sido utilizadas en combinaciones previas. En este caso, las combinaciones se realizan utilizando dichas soluciones.
b. Utilizar un método de búsqueda local: esta opción se efectúa cuando el nodo padre ya ha utilizado todas sus soluciones en combinaciones previas. De este modo, TCSS-LS lleva a cabo un proceso de intensificación para intentar cubrir los nodos que no han sido alcanzados en el proceso de diversificación.

El SUT instrumentado es ejecutado con cada nueva solución y los conjuntos S_k de los nodos alcanzados durante dicha ejecución se actualizan. Si no se consigue generar ninguna solución en el nodo padre que permita llegar al nodo hijo, se deberá realizar nuevamente un backtracking para intentar generar soluciones en el nodo abuelo, llegando si fuera necesario hasta el nodo raíz. Si el backtracking llega al nodo raíz TCSS-LS lleva a cabo un proceso de regeneración puesto que las soluciones almacenadas hasta el momento no permitieron obtener la cobertura total de ramas. Durante dicha regeneración se limpian los conjuntos S_k afectados por el backtracking, pero se recuerda una de sus soluciones para que TCSS-LS no tenga que realizar un esfuerzo adicional buscando soluciones que lleguen a un nodo que previamente había sido alcanzado.

3.4 El proceso de Búsqueda Local

El método de Búsqueda Local trata de generar soluciones que cubran un determinado nodo n_k que provocó el backtracking. Para ello selecciona una solución del conjunto S_k del nodo ancestro seleccionado en cada iteración del backtracking (padre, abuelo, …), denominada solución original ($\overline{os}$), y lleva a cabo una serie de iteraciones a partir de ella para alcanzar el nodo n_k. La solución $\overline{os}$ seleccionada es aquella que posee el menor valor de distancia al nodo hijo n_k que se pretende alcanzar (fc_i^c), intentando así guiar la búsqueda de soluciones que cubran dicho nodo. El número de iteraciones a realizar a partir de una $\overline{os}$ tiene un límite máximo (MAX_INTENTOS) para evitar que se produzca un estancamiento de la búsqueda por el uso de una solución de que no da buen resultado.

En cada iteración del proceso de Búsqueda Local se generan 2n nuevas soluciones (donde n es el número de variables de entrada de la solución) a partir de la mejor solución de la iteración anterior, denominada solución actual ($\overline{cs}$). Las nuevas soluciones se diferencian de $\overline{cs}$ en el valor de una sola variable de entrada y se obtienen de la siguiente forma:

- $\overline{ns1} = \overline{cs}^i + \delta_i$
- $\overline{ns2} = \overline{cs}^i - \delta_i$

donde el índice *i* recorre todas las variables de entrada y δ_i es definida de la siguiente forma:

$$\delta_i = \begin{cases} \dfrac{fc_{cs}}{reductor\_salto} & \text{si} \begin{cases} \text{TCSS - LS calcula } \overline{ns1} \text{ y } \delta_i \le max\_saltoderecho \\ \text{or TCSS - LS calcula } \overline{ns2} \text{ y } \delta_i \le max\_saltoizquierdo \end{cases} \\ \dfrac{max\_saltoderecho}{reductor\_salto} \cdot Ln\left(\dfrac{fc_{cs}\cdot(e-1)}{fc_{os}}+1\right) & \text{si TCSS - LS calcula } \overline{ns1} \text{ y } \delta_i > max\_saltoderecho \\ \dfrac{max\_saltoizquierdo}{reductor\_salto} \cdot Ln\left(\dfrac{fc_{cs}\cdot(e-1)}{fc_{os}}+1\right) & \text{si TCSS - LS calcula } \overline{ns2} \text{ y } \delta_i > max\_saltoizquierdo \end{cases}$$

donde *reductor_salto* es un parámetro que se incrementa para reducir el salto y los saltos máximos se definen como:

- max_saltoderecho = Rango Superior Variable i – $\overline{cs}^i$
- max_saltoizquierdo = $\overline{cs}^i$ – Rango Inferior Variable i

Antes de calcular las nuevas soluciones se deben determinar los saltos máximos que cada variable puede dar sin producir un desbordamiento del rango Así *max_saltoderecho* es el mayor salto que se puede sumar a una variable y *max_saltoizquierdo* es el mayor salto que se puede restar a una variable.

El salto δ_i se calcula a partir de la distancia al nodo hijo que se pretende alcanzar (fc). Esa distancia es dividida por un reductor que permite controlar la amplitud del salto. Si se está calculando $\overline{ns1}$, el salto no debe ser mayor que *max_saltoderecho*. En caso contrario el salto se recalcula teniendo en cuenta el salto máximo y un ratio obtenido en función de los valores de la distancia al nodo hijo de la solución original (fc_{os}) y de la solución actual (fc_{cs}). Este ratio se encuentra situado entre 0 y 1, por lo que suaviza el salto generado. Si se está calculando $\overline{ns2}$, el salto no debe ser mayor que *max_saltoizquierdo* y en caso contrario se sigue el proceso descrito para $\overline{ns1}$.

Posteriormente TCSS-LS ejecuta el SUT con las nuevas soluciones. Estas soluciones son evaluadas para determinar si alguna de ellas alcanza el nodo ancestro y disminuye el valor de la distancia fc de $\overline{cs}$, de modo que se utilice la mejor solución como nueva solución actual. Si ninguna solución mejora $\overline{cs}$ y la Búsqueda Local no ha llevado a cabo MAX_INTENTOS iteraciones a partir de la solución original $\overline{os}$, se reduce el salto utilizado en δi para generar soluciones más cercanas a $\overline{cs}$ en la próxima iteración. Si la Búsqueda Local ha realizado MAX_INTENTOS iteraciones a partir de $\overline{os}$ y el nodo hijo n_k no ha sido alcanzado, el método selecciona otra solución ($\overline{os}$) del conjunto S_k del nodo ancestro para efectuar la búsqueda. Cuando todas las soluciones del nodo ancestro han sido utilizadas por el método de Búsqueda Local, TCSS-LS realiza un backtracking y utiliza las soluciones de otro ancestro para alcanzar el nodo n_k. El proceso de Búsqueda Local finaliza cuando el nodo n_k se cubre o cuando todas las soluciones del nodo ancestro han sido utilizadas en la búsqueda.

4 Resultados

En esta sección se presenta la experimentación realizada con TCSS-LS y con el generador basado únicamente en Búsqueda Dispersa denominado TCSS [5][6]. Para ello se han empleando los benchmarks que se muestran en la

Tabla **1**. En esta tabla figura para cada benchmark su nombre y la abreviatura que lo representa, el número de ramas que contiene, su máximo nivel de anidamiento, su complejidad ciclomática y la referencia desde la cual fueron obtenidos.

Para llevar a cabo los experimentos se definen una serie de instancias, que especifican el SUT empleado por TCSS-LS y TCSS para generar los datos de prueba bajo el criterio de cobertura de ramas. Una instancia representa un benchmark con un determinado tipo de variables de entrada (C para variables de tipo char, F para variables de tipo float e I para variables de tipo entero), las cuales toman valores dentro de un cierto rango (L para rangos de 8 bits, M para rangos de 16 bits y H para rangos de 32 bits), así por ejemplo una instancia estaría definida por el benchmark TrianguloMyers con variables de entrada de tipo entero y rango de entrada de 8 bits. TCSS-LS y TCSS realizan 10 ejecuciones con cada instancia, de forma que los resultados obtenidos (porcentaje de cobertura alcanzada, datos de prueba generados y tiempo consumido) se corresponden con los valores medios de esas ejecuciones.

En todos los experimentos el criterio de parada utilizado por TCSS-LS y TCSS consiste en alcanzar el 100% de cobertura de ramas o generar 200.000 datos de prueba. La máquina utilizada para llevar a cabo la experimentación es un Pentium 1.50GHz con 512 MB de memoria RAM

Tabla 1. Lista de benchmarks empleados en la experimentación

Benchmark	Abr.	Nº de ramas	Nivel de Anidamiento	Complejidad Ciclomática	Referencia
Atof	AF	30	2	17	[34]
BisectionMethod	BM	8	3	5	[26]
ComplexBranch	CB	24	3	14	[34]
CalDay	CD	22	2	12	[2]
LineRectangle	LR	36	12	19	[11]
NumberDays	ND	86	10	44	[11]
QuadraticFormula	QF	4	2	3	[26]
QuadraticFormulaSthamer	QFS	6	3	4	[29]
RemainderSthamer	RS	18	5	10	[29]
TriangleMyers	TM	12	5	7	[25]
TriangleMichael	TMM	20	6	11	[23]
TriangleSthamer	TS	26	12	14	[29]
TriangleWegener	TW	26	3	14	[34]

En la Tabla 2 se resumen los resultados obtenidos por TCSS y TCSS-LS. Para cada instancia se muestra el porcentaje de cobertura alcanzada, el número de datos de prueba generados para lograr dicha cobertura y el tiempo consumido (en segundos). Para llevar a cabo una comparación correcta el número de datos de prueba creados y el tiempo consumido debe ser comparado cuando ambos generadores obtienen el mismo porcentaje de cobertura. Por este motivo cuando TCSS-LS alcanza mayor cobertura que TCSS se muestra entre paréntesis el número de datos de prueba generados y en tiempo consumido por TCSS-LS para alcanzar la misma cobertura obtenida por TCSS.

Los resultados obtenidos por ambos generadores indican que TCSS-LS siempre alcanza, al menos, la cobertura lograda por TCSS. De las 40 instancias TCSS alcanza el 100% de cobertura en 24 de ellas, mientras que TCSS-LS mejora la cobertura obtenida por TCSS al alcanzar el 100% en 33 instancias. En las 7 instancias restantes donde ambos generadores no alcanzan el 100% de cobertura, TCSS-LS incrementa la cobertura obtenida por TCSS en 4 de ellas.

Respecto al número de datos de prueba generados y al tiempo consumido para alcanzar la misma cobertura (27 instancias), TCSS-LS necesita menos datos de prueba en 22 instancias, en las cuales también consume menos tiempo (instancias 3, 6, 7, 15-22, 24-29, 32, 34-37). Sólo en 4 instancias TCSS-LS genera mayor número de datos de prueba y consume más tiempo que TCSS (instancias 9-11, 31).

De las 13 instancias en las que la cobertura alcanzada por TCSS es incrementada por TCSS-LS, éste genera menos datos de prueba para obtener mayor cobertura en 7 de ellas (instancias 1, 2, 4, 5, 12, 30, 33), consumiendo además menos tiempo en todas excepto en una (instancia 12). En las 6 instancias restantes TCSS-LS genera más datos de prueba para incrementar la cobertura, pero en 4 de ellas requiere menos datos de prueba y consume menos tiempo para obtener la máxima cobertura alcanzada por TCSS (instancias 8, 13, 14, 39). En las otras dos instancias los resultados de ambos generadores para obtener la máxima cobertura lograda por TCSS son iguales (instancias 38, 40).

Por otro lado, TCSS-LS obtiene mejores resultados (mayor cobertura, menor número de datos de prueba y menor tiempo) que TCSS en todos los benchmarks excepto en LR, donde obtiene peores resultados en los tres rangos enteros, y en TMM, donde obtiene peores resultados en el rango menor.

La mejora aportada por TCSS-LS también se puede ver analizando las diferencias entre los datos de prueba creados por cada generador, siendo éstas mayores cuando TCSS-LS genera menos datos de prueba que TCSS (por ejemplo, en la instancia 17) y son menores en caso contrario (por ejemplo, en la instancia 9). Por otro lado, cuando el rango de las variables de entrada se incrementa las diferencias también aumentan, por lo que la mejora aportada por TCSS-LS es mayor con rangos grandes.

Tabla 2. Resultados obtenidos por TCSS-LS y TCSS

Instancia	Benchmark	Tipo	Rango	Resultados para TCSS			Resultados para TCSS-LS		
				% Cob.	Datos Prueba	Tiempo (seg)	% Cob.	Datos Prueba	Tiempo (seg)
1	AF	C	L	97,19	101144	141,11	100,00	17133 (12167)	8,28 (4,69)
2	BM	F	H	88,00	52679	39,82	100,00	233 (194)	0,22 (0,22)
3	CB	I	L	100,00	6206	34,24	100,00	4880	19,67
4	CB	I	M	91,92	121223	608,09	100,00	5384 (3838)	15,18 (10,80)
5	CB	I	H	74,23	31210	41,21	100,00	26750 (472)	39,37 (0,82)
6	CD	I	M	100,00	38074	8,82	100,00	558	0,58
7	CD	I	H	100,00	39209	9,07	100,00	561	0,68
8	LR	F	H	97,49	4838	2,29	100,00	5939 (2601)	2,07 (1,21)
9	LR	I	L	100,00	1985	1,18	100,00	4213	1,37
10	LR	I	M	100,00	1580	1,03	100,00	3538	1,25
11	LR	I	H	100,00	2188	1,22	100,00	5817	1,69
12	ND	I	L	99,79	98173	193,19	100,00	76271 (76271)	269,13 (269,13)
13	ND	I	M	6,17	42410	17,91	100,00	115379 (2992)	361,66 (1,32)
14	ND	I	H	2,87	100912	43,32	99,04	148880 (1298)	164,52 (0,56)
15	QF	F	H	100,00	25839	3,75	100,00	50	0,02
16	QF	I	L	100,00	110	0,03	100,00	48	0,02
17	QF	I	M	100,00	20537	3,31	100,00	50	0,02
18	QF	I	H	100,00	26234	3,79	100,00	50	0,02
19	QFS	F	H	100,00	25159	4,25	100,00	2977	0,36
20	QFS	I	L	100,00	1176	0,31	100,00	1096	0,15
21	QFS	I	M	100,00	25052	4,24	100,00	2357	0,28
22	QFS	I	H	100,00	23838	4,06	100,00	1958	0,22
23	RS	I	L	100,00	100	0,09	100,00	100	0,09
24	RS	I	M	100,00	27676	4,79	100,00	482	0,28
25	RS	I	H	100,00	27743	4,82	100,00	484	0,28
26	TM	F	H	100,00	806	0,25	100,00	405	0,15
27	TM	I	L	100,00	367	0,16	100,00	260	0,13
28	TM	I	M	100,00	880	0,26	100,00	370	0,15
29	TM	I	H	100,00	951	0,22	100,00	607	0,15
30	TMM	F	H	96,00	9995	6,16	100,00	4518 (624)	2,35 (0,75)
31	TMM	I	L	100,00	1028	0,76	100,00	1568	0,79
32	TMM	I	M	100	26431	15,20	100,00	5147	2,64
33	TMM	I	H	92,00	117416	73,56	100,00	29419 (16395)	9,64 (5,22)
34	TS	F	H	88,46	1307	1,03	88,46	1121	0,90
35	TS	I	L	100,00	20028	8,83	100,00	18161	4,47
36	TS	I	M	88,46	1162	0,92	88,46	943	0,68
37	TS	I	H	88,46	1527	0,95	88,46	1046	0,77
38	TW	F	H	11,53	4	0,01	96,15	3692 (4)	2,45 (0,01)
39	TW	I	M	92,30	3561	2,25	96,92	17486 (1531)	7,84(1,56)
40	TW	I	H	11,53	4	0,01	96,15	3692 (4)	2,45 (0,01)

Para comprobar que las diferencias observadas entre TCSS-LS y TCSS son significativas se realiza un análisis estadístico con α=0,05. Las hipótesis a verificar son las siguientes:

- Hipótesis H1: El número de datos de prueba generados por TCSS-LS es significativamente menor que los datos de prueba generados por TCSS.
- Hipótesis H2: El tiempo consumido por TCSS-LS es significativamente menor que el tiempo consumido por TCSS
- Hipótesis H3: El número de veces que TCSS-LS obtiene mejores resultados que TCSS es significativo.

Las pruebas que permiten verificar la hipótesis H1 (prueba t para muestras pareadas o la prueba de los signos de Wilcoxon para muestras pareadas) dependen de la normalidad de la distribución. En este caso se trata de la normalidad de la variable $D_{DatosPrueba}$=$DatosPrueba_{TCSS-LS}$-$DatosPrueba_{TCSS}$. Por ello, en primer lugar se realiza la prueba de Kolmogorov-Smirnov. El p-value obtenido es muy pequeño (< 0,001) y no se puede asumir que siga una distribución normal.

Por tanto se aplica la prueba de los signos de Wilcoxon para muestras pareadas para verificar la hipótesis nula sobre la igualdad de medianas (H_0:m_D=0, H_1:m_D<0). El p-value obtenido en el análisis es más pequeño que 0,001<α y en consecuencia la hipótesis H_0:m_D=0 puede ser rechazada. Además la media de los datos de prueba generados por TCSS-LS es menor que la media de los datos de prueba generados por TCSS. Así pues se puede asumir que TCSS-LS genera menos datos de prueba que TCSS.

Para verificar la hipótesis H2 también es necesario comprobar la normalidad de la variable D_{Tiempo}=$Tiempo_{TCSS-LS}$-$Tiempo_{TCSS}$ para determinar la prueba a realizar. El p-value obtenido nuevamente es menor que 0,001 y no se puede asumir que siga una distribución normal.

Al aplicar la prueba de los signos de Wilcoxon para muestras pareadas, la hipótesis H_0:m_D=0 puede volver a ser rechazada ya que el p-value<0,001<α. Además la media del tiempo consumido por TCSS-LS es también menor que la media del tiempo consumido por TCSS. Por lo tanto se puede asumir que TCSS-LS consume menos tiempo que TCSS.

Para verificar la hipótesis H3 se emplea la prueba de McNemar que trata de validar la hipótesis nula que indica que los sujetos confrontados tienen la misma probabilidad de derrotar al otro, es decir,

$$H_0:\ \mathrm{Prob}\left[\frac{\text{vecesmejor}}{\text{vecesmejor}+\text{vecespeor}}\right] = \mathrm{Prob}\left[\frac{\text{vecespeor}}{\text{vecesmejor}+\text{vecespeor}}\right]$$

Para aplicar la prueba de McNemar se considera que un generador es mejor que otro cuando alcanza mayor porcentaje de cobertura o cuando crea menos datos de prueba si ambos generadores alcanzan el mismo porcentaje de cobertura.

En la Tabla 2 se puede observar que TCSS-LS obtiene mejores resultados que TCSS en 35 instancias y obtiene peores resultados en 4 instancias. Ambos generadores obtienen el mismo resultado en una instancia. El p-value obtenido es menor que 0,0001<α, por lo que la diferencia entre ambos generadores es significativa y TCSS-LS obtiene estadísticamente mejores resultados que TCSS.

4 Conclusiones

En este artículo se ha presentado el generador automático de datos de prueba TCSS-LS, que está basado en la técnica metaheurística Búsqueda Dispersa. TCSS-LS también utiliza un procedimiento de búsqueda local para intensificar la búsqueda de datos de prueba en determinados puntos. Este generador utiliza el grafo de control de flujo asociado al programa bajo prueba, el cual permite guiar el proceso de búsqueda, ya que almacena en los Conjuntos de Referencia que cada nodo posee tanto datos de prueba como el conocimiento adquirido durante dicho proceso.

Los resultados de los experimentos muestran que TCSS-LS consigue aumentar el porcentaje de cobertura obtenido por TCSS, y genera además menos datos de prueba y consume menos tiempo,

siendo la mejora aportada estadísticamente significativa. Así mismo, el análisis de los resultados indica que la combinación de varias técnicas, como la Búsqueda Dispersa y la Búsqueda Local, mejora la eficiencia de la generación de datos de prueba, puesto que se pueden aprovechar los principales beneficios de cada una de ellas.

A la vista de los resultados obtenidos, una línea de trabajo consiste en combinar la Búsqueda Dispersa con otra técnica que trabaje con una búsqueda local más especializada, la Búsqueda Tabú, con el fin de desbloquear más rápidamente aquellos puntos en los que la Búsqueda Dispersa encuentra dificultades. Otras líneas de trabajo futuras consisten en emplear TCSS-LS con otros criterios de suficiencia, como cobertura de caminos o MC/DC, así como la aplicación de la Búsqueda Dispersa a la generación de datos de prueba para sentencias de acceso a bases de datos y composiciones de servicios web.

Agradecimientos

Este trabajo ha sido financiado por el Ministerio de Educación y Ciencia y los fondos FEDER dentro del Plan Nacional de I+D+I, Proyectos INT2TEST (TIN2004-06689-C03-02), Test4SOA (TIN2007-67843-C06-01) y RePRIS (TIN2005-24792-E, TIN2007-30391-E).

Referencias

1. Ahmed, M.A., Hermadi, I.: GA-based multiple paths test data generator. Computers and Operations Research 35(10), 3107-3124 (2008).
2. Alba, E., Chicano, F.: Observations in using parallel and sequential evolutionary algorithms for automatic software testing. Computers and Operations Research 35(10), 3161-3183 (2008).
3. Alshraideh, M., Bottaci, L.: Search-based software test data generation for string data using program-specific search operators. Software Testing Verification and Reliability 16(3), 175-203 (2006).
4. Beizer, B.: Software testing techniques. Second edition, Van Nostrand Reinhold (1990).
5. Blanco, R., Díaz, E., Tuya, J.: Algoritmo Scatter Search para la generación automática de pruebas de cobertura de ramas. In: Actas de IX Jornadas de Ingeniería del Software y Bases de Datos, pp. 375-386 (2004).
6. Blanco, R., Díaz, E., Tuya, J.: Generación automática de casos de prueba mediante Búsqueda Dispersa. Revista Española de Innovación, Calidad e Ingeniería del Software 2(1), 24-35 (2006).
7. Bueno, P.M.S., Jino, M.: Automatic test data generation for program paths using Genetic Algorithms. International Journal of Software Engineering and Knowledge Engineering 12(6), 691-709 (2002).
8. Bueno, P.M.S., Wong, W.E., Jino, M.: Improving random test sets using the diversity oriented test data generation. In: Proceedings of the Second International Workshop on Random Testing, pp. 10-17 (2007).
9. Bühler, O., Wegener, J.: Evolutionary functional testing. Computers and Operational Research 35(10), 3144-3160 (2008).
10. Clarke, J., Dolado, J.J., Harman, M., Hierons, R.M., Jones, B., Lumkin, M., Mitchell, B., Mancoridis, S., Rees, K., Roper, M., Shepperd, M.: Reformulating software engineering as a search problem. IEE Proceedings – Software 150(3), 161-175 (2003).
11. Díaz, E., Tuya, J., Blanco, R., Dolado, J.J.: A tabu search algorithm for Software Testing. Computers and Operational Research 35(10), 3052-3072 (2008).
12. Ferguson, R., Korel, B.: The Chaining Approach for Software Test Data Generation. ACM Transactions on Software Engineering and Methodology 5(1), 63-86 (1996).
13. Girgis, M.R.: Automatic test data generation for data flow testing using a genetic algorithm. Journal of Universal Computer Science 11(6), 898-915 (2005).
14. Glover, F.: A template for Scatter Search and Path Relinking, Artificial Evolution, Lecture Notes in Computer Science 1363, Springer-Verlag, pp. 13-54 (1998).
15. Hartman, A.: Is ISSTA Research Relevant to Industry? ACM SIGSOFT Software Engineering Notes 27(4), 205-206 (2002).
16. Jones, B.F., Eyres, D.E., Sthamer, H.H.: A strategy for using Genetic Algorithms to automate branch and fault-based testing. The Computer Journal 41(2), 98-107 (1998).

17. Laguna, M., Martí, R.: Scatter Search: Methodology and Implementations in C. Kluwer Academic Publishers, Boston, MA, USA (2002).
18. Lin, J., Yeh, P.: Automatic test data generation for path testing using Gas. Information Sciences 131 (1-4) 47-64 (2001).
19. Mansour, N., Salame, M.: Data generation for path testing. Software Quality Journal 12, 121-136 (2004).
20. Mantere,T., Alander, J.T.: Evolutionary software engineering, a review. Applied Soft Computing 5(3), 315-331 (2005).
21. McMinn, P.: Search-based software test data generation: a survey. Software Testing Verification and Reliability 14(2), 105-156 (2004).
22. McMinn, P., Holcombe, M.: Evolutionary testing using an extended chaining approach. Evolutionary Computation 14(1), 41.64 (2006).
23. Michael, C., McGraw, G., Schatz, M.: Generating software test data by evolution. IEEE Transactions on Software Engineering 27(12), 1085-1110 (2001).
24. Miller, J., Reformat, M., Zhang, H.: Automatic test data generation using genetic algorithm and program dependence graphs. Information and Software Technology 48, 586-605 (2006).
25. Myers, G.: The art of software testing, Ed. John Wiley & Sons (1979).
26. Offut, A.J., Pan, J., Tewary, K., Zhang, T.: Experiments with Data Flow and Mutation Testing. Technical Report ISSE-TR-94-105, 1994.
27. Pargas, R.P., Harrold, M.J., Peck, R.R.: Test data generation using genetic algorithms. Journal of Software Testing Verification and Reliability 9, 263-282 (1999).
28. Sagarna, R., Lozano, J.A.: Scatter Search in software testing, comparison and collaboration with Estimation of Distribution Algorithms, European Journal of Operational Research 169, 392-412 (2006).
29. Sthamer, H.H.: The automatic generation of software test data using genetic algorithms. PhD Thesis, University of Glamorgan (1996).
30. Tracey, N., Clark, J., Mander, K.: Automated program flaw finding using simulated annealing. In: Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 73-81 (1998).
31. Vergilio, S.R., Pozo, A.: A grammar-guided genetic programming framework configured for data mining and software testing. International Journal of Software Engineering and Knowledge Engineering 16(2), 245-267 (2006).
32. Waeselynck, H., Thévenod-Fosse,P., Abdellatif-Kaddour, O.: Simulated annealing applied to test generation: landscape characterization and stopping criteria. Empirical Software Engineering 12(1), 35-63 (2007).
33. Watkins, A., Hufnagel, E.M.: Evolutionary test data generation: a comparison of fitness functions. Software Practice and Experience 36, 95-116 (2006).
34. Wegener, J., Baresel, A., Sthamer, H.: Evolutionary test environment for automatic structural testing, Information & Software Technology 43(14), 841-854 (2001).
35. Xiao, M., El-Attar, M., Reformat, M., Miller, J.: Empirical evaluation of optimization algorithms when used in goal-oriented automated test data generation techniques. Empirical Software Engineering 12(2), 183-239 (2007).