Revista Española de Innovación, Calidad e Ingeniería del Software, Vol.2, No. 1, 2006

Generación automática de casos de prueba mediante búsqueda dispersa

Raquel Blanco, Eugenia Díaz, Javier Tuya
Departamento de Informática, Universidad de Oviedo
{rblanco | madiaz | tuya}@uniovi.es

Abstract

The test process is very expensive and the number of test cases needed to test a program is infinite, therefore it is impossible to obtain a fully tested program. For this reason, the techniques for the automatic generation of test cases try to efficiently find a small set of test cases that allow fulfilling an adequacy criterion. A method based on Scatter Search that automatically generates test cases to obtain branch coverage is presented in this article. Besides we show the results we have obtained compared with a method based on Tabu Search.

Resumen

El número de casos de prueba necesarios para probar un programa software es infinito, por lo que es imposible conseguir un programa totalmente probado, siendo además el proceso de prueba muy costoso. Por estos motivos las técnicas para la generación automática de casos de prueba tratan de encontrar de forma eficiente un conjunto pequeño de casos de prueba que permitan cumplir un determinado criterio de suficiencia. En este artículo se presenta un método basado en Búsqueda Dispersa que permite generar automáticamente casos de prueba para obtener cobertura de ramas. Además se muestran los resultados comparativos del método basado en Búsqueda Dispersa y de un método basado en Búsqueda Tabú.

Palabras clave: pruebas de software, automatización de la generación de casos de prueba, cobertura de ramas, técnicas de búsqueda metaheurísticas, búsqueda dispersa.

1 Introducción

La aplicación de algoritmos metaheurísticos para resolver problemas en Ingeniería del Software ha sido propuesto por la red SEMINAL (Software Engineering using Metaheuristic INnovative Algorithms) y se trata ampliamente en [4]. Una de esas aplicaciones es la selección de casos en el proceso de la prueba del software.

La prueba del software es el proceso de ejecutar un programa con el objetivo de encontrar errores [14].El número de casos de prueba necesarios para probar un programa software es infinito, por lo que es imposible conseguir un programa totalmente probado. Además el proceso de prueba es costoso y puede suponer el 50% del coste total del desarrollo software [1]. Por estos motivos las técnicas para la generación automática de casos de prueba tratan de encontrar de forma eficiente un conjunto pequeño de casos de prueba que permitan cumplir un determinado criterio de suficiencia. Entre las técnicas más recientes que son utilizadas para realizar esta automatización se encuentran las técnicas de búsqueda metaheurísticas, como los Algoritmos Genéticos [8][12][13][15][17], el Recocido Simulado [16] o la Búsqueda Tabú [5][6], aunque en la práctica la mayor parte de los trabajos utilizan Algoritmos Genéticos. Otra técnica metaheurística que también puede ser aplicada a la prueba del software es la Búsqueda Dispersa [7][10].

En este artículo se explica un desarrollo específico de la técnica Búsqueda Dispersa para satisfacer el criterio estructural de cobertura de ramas, ampliando lo descrito en [2][3].

2 La técnica de Búsqueda Dispersa

La Búsqueda Dispersa (Scatter Search)[7][10] 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. Esta técnica utiliza estrategias sistemáticas para avanzar en el proceso de búsqueda en vez de aleatorias, siendo ésta una de las principales diferencias con los ampliamente utilizados Algoritmos Genéticos.

El algoritmo Scatter Search comienza generando un conjunto P de soluciones diversas mediante un método de generación de diversidad. Las soluciones presentes en este conjunto pueden ser mejoradas con un método de mejora, el cual es opcional. Posteriormente se construye el conjunto de referencia con las mejores soluciones de P y las más diversas a las ya incluidas. A continuación comienza un proceso cíclico en el cual el algoritmo crea subconjuntos del conjunto de referencia, con un método de generación de subconjuntos, y aplica un método de combinación sobre dichos subconjuntos para obtener las nuevas soluciones. Posteriormente, sobre cada nueva solución aplica un método de mejora y evalúa si debe incorporarse al conjunto de referencia, mediante un método de actualización. El algoritmo se detiene cuando no se generan nuevas soluciones en el proceso de combinación.

3 Descripción del generador de casos de prueba basado en Búsqueda Dispersa

El generador de casos de prueba desarrollado basado en Búsqueda Dispersa se denomina TCSS (Test Coverage Scatter Search) y utiliza el grafo de control de flujo asociado al programa bajo prueba para almacenar información relevante durante el proceso de búsqueda de casos de prueba. Con este grafo es posible determinar qué ramas han sido cubiertas debido a que el programa bajo prueba ha sido instrumentado para determinar el camino seguido por cada caso de prueba ejecutado en él.

El objetivo de TCSS es generar casos de prueba que permitan cubrir todas las ramas de un programa. Este objetivo se puede dividir en subobjetivos, consistiendo cada uno de ellos en encontrar casos de prueba que alcancen un determinado nodo del grafo de control de flujo.

Para alcanzar estos subobjetivos, TCSS trabaja con un conjunto de soluciones (conjunto de referencia) en cada nodo del grafo de control de flujo. Cada uno de estos conjuntos de soluciones se denomina S_k (donde k es el número de nodo) y contiene varios elementos $T_k^c = <\overline{x}_k^c, p_k^c, f_k^c>$, donde $\overline{x}_k^c$ es una solución (un caso de prueba) que alcanza el nodo k, p_k^c es el camino recorrido por la solución y f_k^c es la distancia que indica lo cerca que dicha solución está de pasar por su nodo hermano, es decir, el nodo cuya decisión de entrada es la negación de la decisión del nodo k. Cada una de las soluciones almacenadas en un nodo k está compuesta por los valores de las variables de entrada que hacen ciertas

tanto las decisiones de entrada a los nodos precedentes al nodo k en un determinado camino que permite llegar a él, como la del propio nodo k. La estructura del grafo de control de flujo asociado a un programa, junto con la información que se almacena en cada nodo se puede ver en la Figura 1.

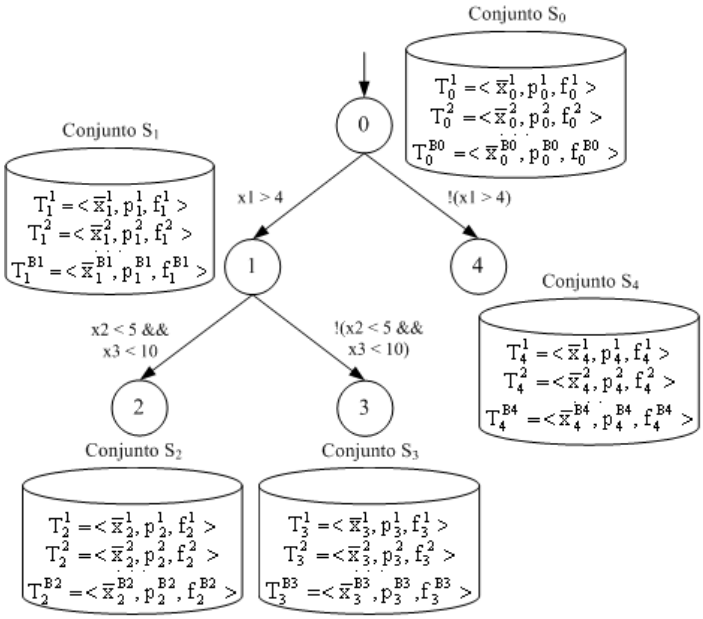

Figura 1. Información almacenada por TCSS en el grafo de control de flujo

El conjunto de soluciones de un nodo k (S_k tiene un tamaño máximo B_k. Este tamaño es distinto para cada nodo k y depende de la complejidad del código situado por debajo de dicho nodo k, que puede ser medida por medio de la complejidad ciclomática [11] del grafo de control de flujo resultante de tomar como raíz al propio nodo k. Este valor de complejidad es multiplicado por un factor fijo para disponer de una cantidad razonable de soluciones en cada conjunto S_k que permita generar otras nuevas.

TCSS tratará de hacer los conjuntos S_k lo más diversos posibles, utilizando una función de diversidad, para generar soluciones que puedan cubrir distintas ramas del programa.

El objetivo de TCSS es obtener la máxima cobertura de ramas, por lo que deben encontrarse soluciones que permitan cubrir todos los nodos del grafo de control de flujo. Como dichas soluciones se almacenan en los nodos, el objetivo de TCSS es, por tanto, que todos los nodos tengan al menos un elemento en su conjunto S_k.

3.1 Cálculo de la distancia asociada a una solución

Para que una solución alcance un determinado nodo, debe cumplirse la decisión de entrada al mismo, por ello, si se desea calcular lo cerca que la solución está de pasar por el nodo hermano se debe utilizar la decisión de entrada a dicho nodo hermano. Por ejemplo, en la Figura 1 la solución $\overline{x}_1^1$ alcanza el nodo 1 y el valor de distancia f_1^1 se calcula utilizando la decisión de entrada al nodo 4 (!(x1>4)). El cálculo de las distancias f_k^c se muestra en la Tabla 1.

Condición	**eval(Condición, $\overline{x}$)**
x=y, x≠y, x<y, x≤y, x>y, x≥y	|x-y|
Decisión	**f_k^c**
C1 AND C2	$\sum \text{eval}(C_j, \overline{x})\ \forall C_j$ False
C1 OR C2	$\text{Min}(\text{eval}(C_j, \overline{x}))\ \forall C_j$
¬C	Aplicar leyes de De Morgan

Tabla 1. Cálculo de las distancias f_k^c

En primer lugar se evalúa cada condición que forma parte de la decisión de entrada al nodo hermano, según los valores de las variables de entrada que constituyen la solución. Posteriormente, se calcula el valor f_k^c de la decisión. Si en la decisión intervienen operadores AND, la distancia f_k^c será el resultado de sumar la evaluación de las condiciones falsas, pues son las que impiden que se alcance al nodo hermano. Si en la decisión intervienen operadores OR, la distancia f_k^c será el valor mínimo de las evaluaciones de las condiciones, ya que todas ellas son falsas y con que se cumpla una sola se alcanzaría el nodo hermano. Si la decisión está negada, simplemente se aplican las leyes de De Morgan.

3.2 Generación de nuevas soluciones

En cada iteración del algoritmo, TCSS selecciona un nodo para generar las nuevas soluciones del proceso de búsqueda por medio de la combinación de las soluciones almacenadas en su conjunto S_k. Los criterios que rigen la selección del nodo con el que trabajar pueden consultarse en [3].

TCSS selecciona un número constante b de elementos $T_k^c = < \overline{x}_k^c, p_k^c, f_k^c >$ del conjunto S_k que previamente no hayan sido utilizados para generar nuevas soluciones, intentado que esas soluciones ($\overline{x}_k^c$) cubran caminos distintos (p_k^c) y tengan menor valor de distancia (f_k^c). Con las soluciones seleccionadas se forman todos los posibles pares ($\overline{x}_k^j$, $\overline{x}_k^h$), con j≠h, y a partir de cada uno de ellos se generan cuatro nuevas soluciones como resultado de aplicar las siguientes combinaciones elemento a elemento: $\overline{x}_k^{ji} \pm \Delta_i$, $\overline{x}_k^{hi} \pm \Delta_i$, donde $\Delta_i = |\overline{x}_k^{ji} - \overline{x}_k^{hi}|/2$ y el índice i recorre todas las variables de entrada. Cada nueva solución es examinada para determinar si los valores de sus variables de entrada se encuentran situados dentro del rango que cada una de ellas puede tomar. De no ser así, se debe aplicar sobre dicha solución el método de mejora, que se limita a modificar los valores de las variables para que no sea considerada inválida.

Con estas combinaciones se pretende generar soluciones que se alejen de las originales y soluciones que se sitúen entre ellas. Además con los criterios de selección de las soluciones se intentan combinar soluciones diversas (recorren distintos caminos) que estén próximas a producir un salto de rama.

Si el nodo seleccionado para generar nuevas soluciones no posee al menos dos soluciones que puedan ser empleadas para realizar las combinaciones se lleva a cabo un proceso de backtracking que puede ser consultado en [3].

El programa bajo prueba es ejecutado con cada nueva solución. Cada una de estas ejecuciones puede causar la actualización de los conjuntos S_k de los nodos alcanzados. Esta actualización se describe en la siguiente subsección.

3.3 Actualización de los conjuntos de soluciones

La actualización de un conjunto de soluciones S_k tiene en cuenta el estado de dicho conjunto. Así si el conjunto S_k no ha excedido su tamaño máximo B_k, la nueva solución es aceptada. En otro caso, la actualización se realiza mediante la función de diversidad. Esta función determina si una nueva solución puede ser añadida a un determinado conjunto S_k y, en ese caso, qué solución debe ser eliminada del mismo.

Cuando se incluye una nueva solución en un conjunto S_k y su tamaño máximo B_k ha sido sobrepasado, se aplica la función de diversidad para determinar la solución que debe abandonarlo, pudiendo ser incluso la solución que provocó el desbordamiento del conjunto.

La función de diversidad se aplica sobre el subconjunto $S_{p^*}=\{T_{p^*}{}^1=<\overline{x}_{p^*}{}^1;p_{p^*};f_{p^*}{}^1>,...,T_{p^*}{}^q=<\overline{x}_{p^*}{}^q;p_{p^*};f_{p^*}{}^q>\} \subseteq S_k$, que representa las soluciones almacenadas en el nodo k que cubren el camino (p_{p^*}) con más ocurrencias en el conjunto S_k. La solución más similar al resto de soluciones que recorren el mismo camino (la menos diversa) abandonará el conjunto S_k. El valor de diversidad de una solución se calcula según la función de diversidad definida como:

$$div(<x^m_{p^*}; p_{p^*}>; S_{p^*}) = \sum_{y=1..q}\left(\sum_{i=1..n}\frac{\left|\overline{x}^{m_i}_{p^*} - \overline{x}^{y_i}_{p^*}\right|}{range_i}\right)$$

donde el índice y recorre las soluciones del conjunto S_{p^*}, el índice i recorre las variables de entrada y $rango_i$ es el rango de valores de la variable de entrada i.

Si existen dos o más soluciones con el mismo valor de diversidad, la solución que será eliminada del conjunto S_k será aquella con mayor valor de distancia, es decir, la menos cercana a producir un salto de rama.

La aplicación de la función de diversidad sobre el subconjunto de soluciones que recorren el camino con más ocurrencias dentro del conjunto S_k tiene como objetivo equilibrar el número de soluciones que cubren diferentes caminos, consiguiendo así mayor diversidad.

4 Resultados

En esta sección, se presentan los resultados obtenidos con un programa que determina la posición de una línea respecto a un rectángulo. Su grafo de control de flujo aparece en la Figura 2.Este programa tiene ocho variables de entrada enteras, cuatro de ellas (xr1, xr2, yr1, yr2) representan las coordenadas de un rectángulo, y las otras cuatro (xl1, xl2, yl1, yl2) representan las coordenadas de la línea. El grafo de control de flujo de este programa tiene nodos difíciles de alcanzar debido a la existencia de igualdades y decisiones compuestas a un alto nivel de anidamiento.

La comparación de TCSS ha sido realizada respecto al generador basado en Búsqueda Tabú TSGen [5][6]. Para ambos generadores se muestran resultados para diferentes rangos (16 bits, 24 bits, 32 bits) y variables de tipo entero. En los experimentos, el criterio de parada utilizado para ambos generadores ha sido alcanzar el 100% de

cobertura de ramas o generar 1.000.000 de casos de prueba. Para cada rango se han llevado a cabo 10 ejecuciones, presentando como resultado el promedio de casos de prueba y tiempo que cada generador necesita para obtener un determinado porcentaje de cobertura de ramas para el programa bajo prueba.

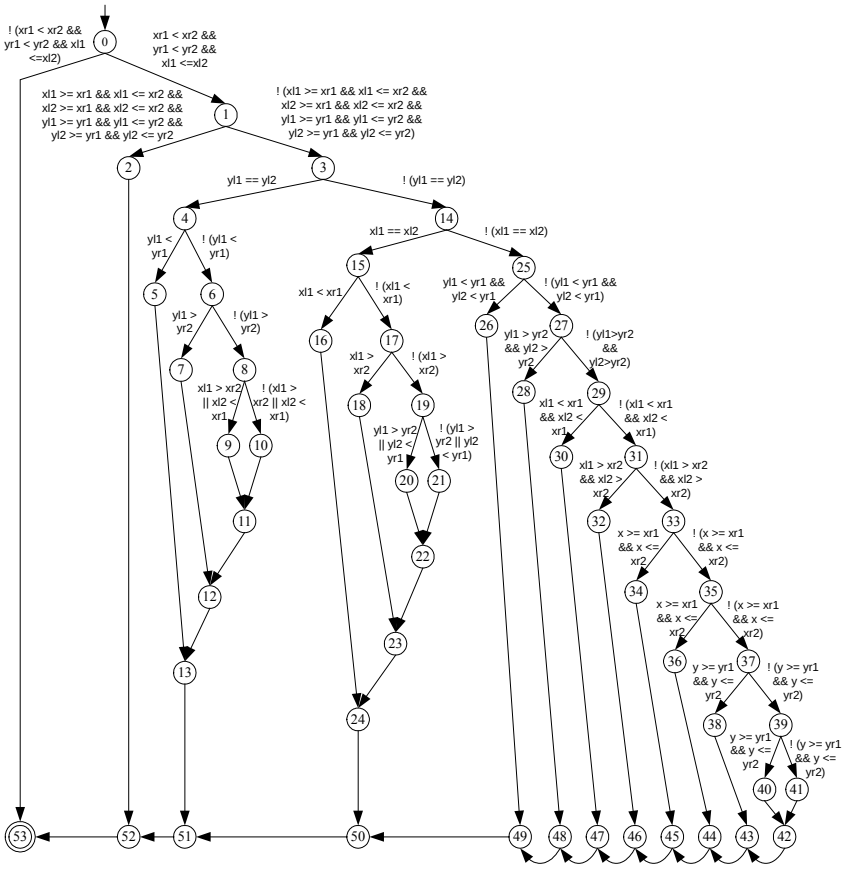

Figura 2. Grafo de control de flujo del programa Position_Line_Rectangle

Los resultados obtenidos por los dos generadores se pueden ver en la Tabla 2. Para cada rango de las variables de entrada se muestra el número de casos de prueba generados, el porcentaje de cobertura alcanzado con dicho número de casos y el tiempo en segundos empleado en ello. TCSS necesita generar menos casos de prueba que TSGen para alcanzar el 100% de cobertura cuando el rango de entrada es de 16 bits, pero cuando dicho rango aumenta TCSS genera más casos de prueba que TSGen. Además TCSS no alcanza el 100% de cobertura en todas las ejecuciones para rangos de 24 bits (2 ejecuciones no logran la cobertura total) y 32 bits (4 ejecuciones no logran la cobertura total), mientras que TSGen

siempre alcanza el 100% de cobertura. Respecto al tiempo consumido por ambos generadores, TCSS consume menos tiempo que TSGen para todos los rangos de entrada.

	Rango: 16 bits (-32768, 32767)			Rango: 24 bits (-8388608, 8388607)			Rango: 32 bits (-2147483648, 2147483648)		
	Casos prueba	% cober-tura	Tiempo (seg.)	Casos prueba	% cober-tura	Tiempo (seg.)	Casos prueba	% cober-tura	Tiempo (seg.)
TCSS	3356	100	1,10	595251	98,3	113,77	858759	97,5	168,34
TSGen	27312	100	57,10	65091	100	147,05	177967	100	454,41

Tabla 2. Resultados obtenidos para TCSS y TSGen

La evolución para cada rango de entrada del número de casos de prueba respecto al porcentaje de cobertura alcanzado se muestra en la Figura 3. TCSS genera menos casos de prueba que TSGen para el rango de 16 bits. Cuando el rango aumenta TCSS genera menos casos de prueba hasta un porcentaje de cobertura del 70%. A partir de ese punto TSGen genera menos casos de prueba.

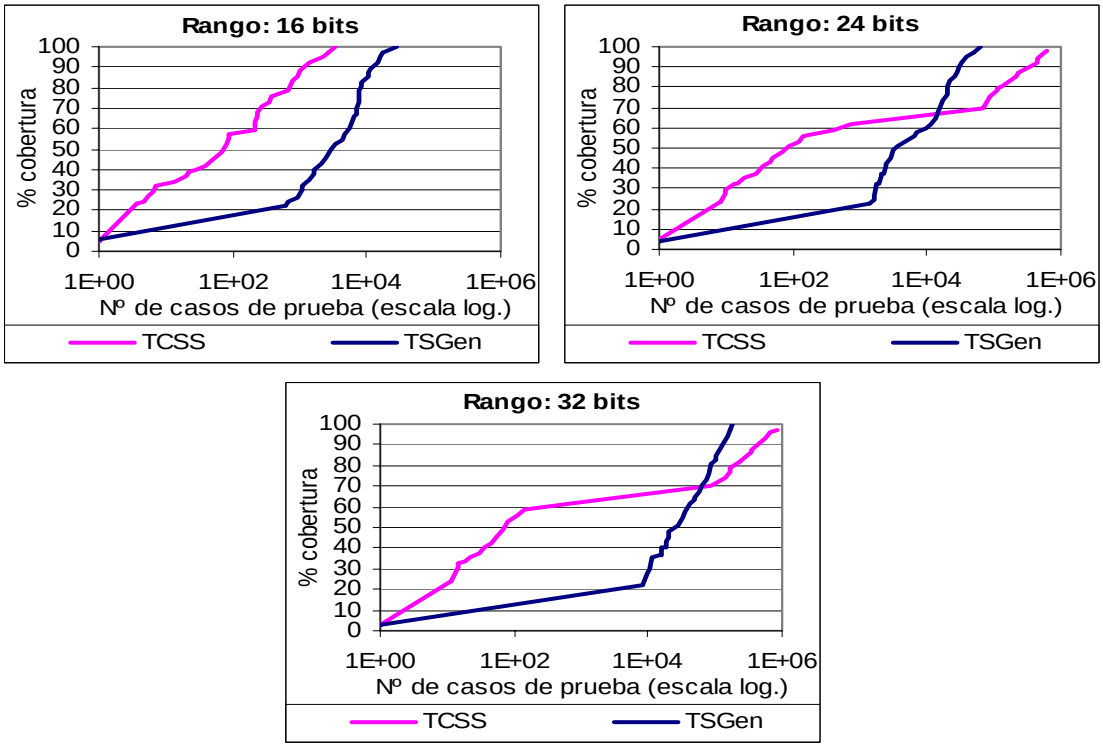

Figura 3. Número de casos de prueba respecto al porcentaje de cobertura alcanzado para cada rango de las variables de entrada

La evolución para cada rango de entrada del tiempo empleado en alcanzar cada porcentaje de cobertura se muestra en la Figura 4. TCSS necesita consumir menos tiempo que TSGen para obtener cada porcentaje de cobertura. A medida que el rango de entrada aumenta, la diferencia de tiempos se hace menor a partir de un porcentaje de cobertura del 70%.

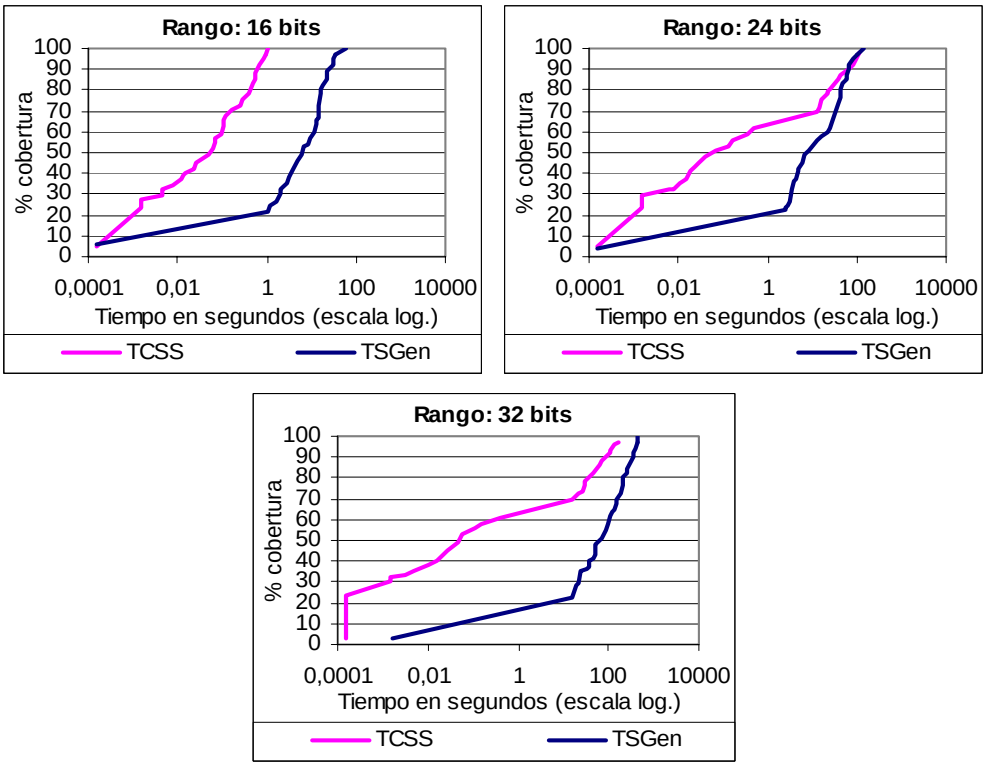

Figura 4. Tiempo empleado respecto al porcentaje de cobertura alcanzado para cada rango de las variables de entrada

5 Conclusiones

En este artículo se ha descrito la adaptación realizada de la técnica metaheurística Búsqueda Dispersa a la generación automática de casos de prueba para cobertura de ramas, dando como resultado el desarrollo del generador de casos de prueba TCSS. Este generador utiliza el grafo de control de flujo asociado al programa bajo prueba y maneja un conjunto de soluciones en cada nodo, que facilita la división del objetivo general de obtener la máxima cobertura de ramas posible en subobjetivos.

Los resultados obtenidos en los experimentos muestran que TCSS se comporta mejor que TSGen con rangos pequeños y en las primeras iteraciones de la búsqueda, debido a la

utilización de la función de diversidad, ya que con esta función TCSS trata de generar casos de prueba que cubran diversas ramas a partir de las soluciones de un determinado nodo. En la cobertura de los nodos más difíciles TSGen se comporta mejor, gracias a la búsqueda local que implementa.

Por lo tanto los resultados sugieren una línea de integración de los generadores TCSS y TSGen aprovechando las mejores características de cada uno, de modo que se puedan mejorar los resultados de ambos.

Actualmente estamos trabajando en la incorporación de una búsqueda local al generador TCSS para mejorar la cobertura de los nodos más difíciles, así como en la utilización de una memoria que permita diferenciar buenas y malas soluciones y la realización de nuevos experimentos que permitan efectuar la comparación de TCSS con otras técnicas de generación automática de casos de prueba.

Agradecimientos

Este trabajo ha sido financiado por el Ministerio de Educación y Ciencia dentro del Plan Nacional de I+D+I, Proyectos TIN2004-06689-C03-02 (IN2TEST) y TIN2005-24792-E (REPRIS).

Referencias

[1] Beizer B, "Software testing techniques", 2ª edición, Van Nostrand Reinhold, 1990.
[2] Blanco R, Díaz E, Tuya J, "Algoritmo Scatter Search para la generación automática de pruebas de cobertura de ramas", IX Jornadas de Ingeniería del Software y Bases de Datos, pp 375-386, 2004.
[3] Blanco R, Tuya J, Díaz E, Díaz BA, "A Scatter Search approach for automated coverage in software testing", International Conference on Knowledge Engineering and Decision Support, pp 387-394, 2004.
[4] Clarke J, Dolado JJ, Harman M, Hierons RM, 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.
[5] Díaz E, Tuya J, Blanco R, "Automated Software Testing Using a Metaheuristic Technique Based on Tabu Search", 18th IEEE International Conference on Automated Software Engineering. IEEE Computer Society Press, pp 310-313, 2003.
[6] Díaz E, Tuya J, Blanco R, "Pruebas automáticas de cobertura de software mediante una herramienta basada en Búsqueda Tabú". VIII Jornadas de Ingeniería del Software y Bases de Datos, pp 283-292, 2003.
[7] Glover F, Laguna M, Martí R, "Fundamentals of Scatter Search and Path Relinking", Control and Cybernetics, 39(3):653-684, 2000.

[8] Jones B, Eyres D, Sthamer H, "A strategy for using genetic algorithms to automate branch and fault-based testing", Computer Journal, 41(2): 98-107, 1998.
[9] Korel B,"Automated software test data generation",IEEE Transactions on Software Engineering,16(8):870-870,1990.
[10] Laguna M, Martí R, "Scatter Search: Methodology and Implementations in C", Kluwer Academic Publishers, Boston, 2002.
[11] McCabe TJ, "A complexity measure", IEEE Transaction Software Engineering, 2(4):308-320, 1976.
[12] Michael C, McGraw G, Schatz M, "Generating Software Test Data by Evolution", IEEE Transactions on Software Engineering, 27(12):1085-1110, 2001.
[13] Mansour N, Salame M, "Data generation for path testing", Software Quality Journal, 12(2):121-136, Kluwer Academic Publishers, 2004
[14] Myers G, "The art of software testing", Ed. John Wiley & Sons, 1979.
[15] Pargas R, Harrold MJ, Peck R, "Test-data generation using genetic algorithms", Software Testing, Verification and Reliability, 9(4): 263-282, 1999.
[16] Tracey N, Clark J, Mander K, "Automated program flaw finding using simulated annealing", International Symposium on software testing and analysis, ACM/SIGSOFT, pp 73-81, 1998.
[17] Wegener J, Baresel A, Sthamer H, "Evolutionary test environment for automatic structural testing", Information & Software Technology, 43(14):841-854, 2001.