Comparación de técnicas metaheurísticas para la generación automática de casos de prueba que obtengan una cobertura software

Eugenia Díaz, Raquel Blanco, Javier Tuya

Departamento de Informática, Universidad de Oviedo
Campus de Viesques s/n, Gijón, Asturias
33204 España
{eugenia, rblanco, tuya}@lsi.uniovi.es

Resumen. La prueba del software es un proceso difícil y caro que consume normalmente el 50% de los costes en un desarrollo software. Por ello el uso de técnicas que permiten la automatización de este proceso es muy importante. Los últimos métodos para la generación automática de casos de prueba para la obtención de cobertura software utilizan la técnica metaheurística denominada Algoritmos Genéticos. En este artículo se comparan los resultados obtenidos en los distintos trabajos con dicha técnica y los resultados obtenidos por un algoritmo propio basado en la técnica metaheurística Búsqueda Tabú.

1 Introducción

La prueba del software es una parte del proceso conocido como Verificación y Validación (V&V) [17], cuya finalidad es determinar si los productos desarrollados cumplen los requisitos y si el software satisface las necesidades del usuario [8]. El propósito de las pruebas de software es descubrir fallos ocultos para asegurar que éstos no vuelvan a aparecer y reducir el coste debido a ellos durante la vida del producto [14].

Según los aspectos software que se desean cubrir, las pruebas se pueden clasificar en pruebas de caja blanca y pruebas de caja negra [14]. Las pruebas de caja blanca examinan la estructura del programa basándose en una serie de criterios de cobertura tales como múltiple condición (cada una de las permutaciones de los valores de las variables booleanas en las condiciones deben ocurrir al menos una vez), funciones (cada una de las funciones debe ser llamada al menos una vez), sentencias (cada sentencia debe ejecutarse al menos una vez) o ramas (cada rama condicional debe ejecutarse al menos una vez), entre otros. Las pruebas de caja negra se basan únicamente en las especificaciones de los datos de entrada y salida del producto.

La tarea de probar el software es un proceso caro, que normalmente consume el 50% del coste total de un desarrollo software [1]. Con técnicas para automatizar la generación de casos de prueba se puede probar el software más eficientemente, disminuyendo el tiempo empleado en esta tarea. De este modo se reduce el coste y se incrementa la calidad del producto final.

2 Técnicas metaheurísticas para pruebas de cobertura

El número de casos de prueba necesarios para probar el software es infinito y por tanto es imposible lograr la prueba total del mismo. Por esta razón las técnicas para la generación de pruebas intentan encontrar un conjunto mínimo de pruebas sin pérdida de eficacia.

Existen diversos paradigmas para la generación automática de casos de prueba, entre los que se encuentran la generación aleatoria [15], la generación de tests simbólicos (técnicas estáticas) [4] y la generación dinámica de pruebas [10]. En el primero de ellos, las entradas se generan aleatoriamente hasta encontrar una útil, respecto a algún tipo de criterio que suele ser el de cobertura de código. Cuanto mayor es la complejidad del programa a probar o más complicado es el nivel de cobertura a conseguir, más difícil es encontrar los casos de prueba adecuados. En las técnicas estáticas se asignan valores simbólicos a las variables para obtener una expresión algebraica que represente lo que hace el programa. En la generación dinámica de pruebas se realiza una búsqueda directa de los casos de prueba a través de la ejecución del programa a probar.

Las técnicas dinámicas para la generación automática de pruebas más recientes utilizan técnicas de búsqueda metaheurísticas, en las cuales el problema de las pruebas es tratado como un problema de búsqueda o de optimización [3]. Entre estas técnicas se encuentran los algoritmos genéticos, el recocido simulado y la búsqueda tabú (figura 1).

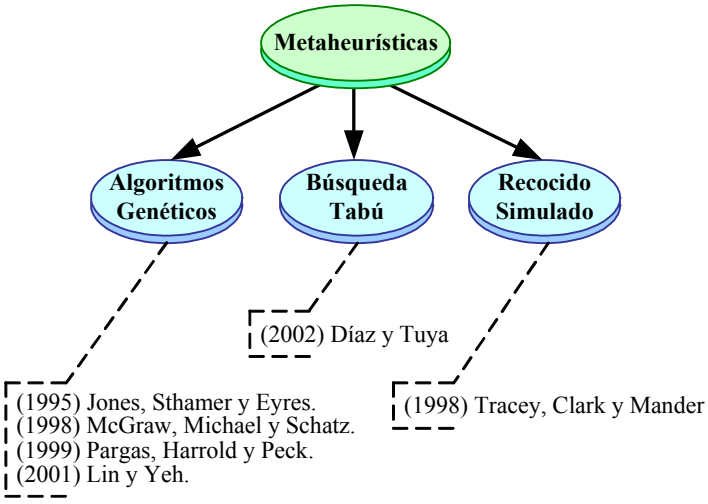

Figura 1: Técnicas de búsqueda metaheurísticas aplicadas a pruebas de software de caja blanca

Los algoritmos genéticos constituyen la técnica metaheurística más ampliamente usada y fueron establecidos por [7] y posteriormente por [6]. Un algoritmo genético es un método de búsqueda paralelo basado en los procesos de evolución natural. El primer estudio para generar pruebas automáticas de software es [9] y su objetivo es obtener cobertura de ramas. Para ello utilizan el grafo de control de flujo del progra-

ma a probar y como función objetivo el recíproco de la distancia de Hamming. En [12] el objetivo es lograr cobertura de ramas y de condición-decisión. La generación dinámica de casos de prueba se realiza a través de su propio sistema llamado GADGET (Genetic Algorithm Data Generation Tool). Este sistema está diseñado para trabajar con programas grandes escritos en C y C++ y sin limitar el conjunto de construcciones de C válidas. GADGET combina el enfoque de minimización de una función de [13] y [10] con la estrategia de [2] consistente en usar una tabla de cobertura para marcar qué ramas se han probado y cuáles no en cada instante. Se basa en la idea de encontrar un camino para alcanzar la localización del código dónde se quiere que el criterio de cobertura sea satisfecho y convertir ese criterio en una función que pueda ser minimizada.

Otro sistema para la generación de casos de prueba, denominado TGEN, se describe en [16] y trata de conseguir cobertura de objetivos. Para ello utiliza el grafo de control de dependencias, en el cual los nodos representan sentencias y los ejes representan las dependencias entre sentencias (predicados). La función objetivo compara el conjunto de predicados actual con el conjunto de predicados que tienen que cumplirse para llegar al nodo objetivo. Una solución que cubre la mayoría de los predicados tendrá una evaluación alta de la función objetivo.

Por otro lado, [11] tiene como objetivo alcanzar cobertura de caminos, apoyándose en el grafo de control de flujo. Cada rama del grafo es etiquetada y cada rama del programa es instrumentada para generar una etiqueta al ejecutarse. Como el número de caminos es infinito, se seleccionan un conjunto de ellos para realizar las pruebas, normalmente los más difíciles de cubrir mediante pruebas aleatorias. El sistema genera los casos de prueba y se evalúan, ejecutando el programa, para determinar si se ha satisfecho el criterio de prueba adecuado. La función objetivo que emplea es una extensión de la distancia de Hamming llamada NEHD (Normaliza Extended Hamming Distance) que mide la distancia entre dos caminos y por tanto permite determinar el camino conocido que esté más cerca del camino objetivo.

La técnica de recocido simulado ha sido utilizada por [18] para generar automáticamente casos de prueba basados en las especificaciones de un sistema y para la prueba de condiciones que elevan excepciones en un programa, pero no ha sido empleada para la generación de pruebas de cobertura. El recocido simulado es un método de búsqueda local que trata de encontrar un conjunto de posibles soluciones, reduciendo la posibilidad de quedarse atascado en un óptimo local malo mediante movimientos a soluciones inferiores controlados por un esquema aleatorio.

La Búsqueda Tabú [5] se basa en el algoritmo de los k-vecinos junto con el mantenimiento de una lista tabú que evita repetir la búsqueda dentro de un área del espacio de soluciones. Nuestra propia técnica está basada en búsqueda tabú y utiliza como criterio de prueba la obtención de cobertura de ramas. Para ello emplea el grafo de control de flujo del programa a probar y una función objetivo que se crea específicamente para cada tipo de relación lógica presentes en las condiciones.

En la tabla 1 se puede ver una comparativa de las distintas técnicas planteadas.

Tabla 1: Comparativa de funcionamiento de las distintas técnicas.

	Técnica	Criterio de cobertura
(1995) Jones, Sthamer y Eyres	Algoritmos Genéticos	Ramas
(1998) McGraw, Michael y Schatz	Algoritmos Genéticos	Ramas Condición-decisión
(1998) Tracey, Clark y Mender	Recocido Simulado	No generan pruebas de cobertura
(1999) Pargas, Harrold y Peck	Algoritmos Genéticos	Objetivos: implementado para sentencias y ramas
(2001) Lin y Yeh	Algoritmos Genéticos	Caminos
(2002) Técnica propia	Búsqueda Tabú	Ramas

3 Resultados

En esta sección se presentan los resultados obtenidos por los diferentes autores utilizando un famoso benchmark para pruebas software: el problema de clasificación del triángulo.

El problema de clasificación del triángulo tiene tres variables de entrada (A, B, C) que representan la longitud de los lados de la figura. El programa determina si representa un triángulo, y en ese caso su tipo. El grafo de control de flujo del problema del triángulo aparece en la figura 2.

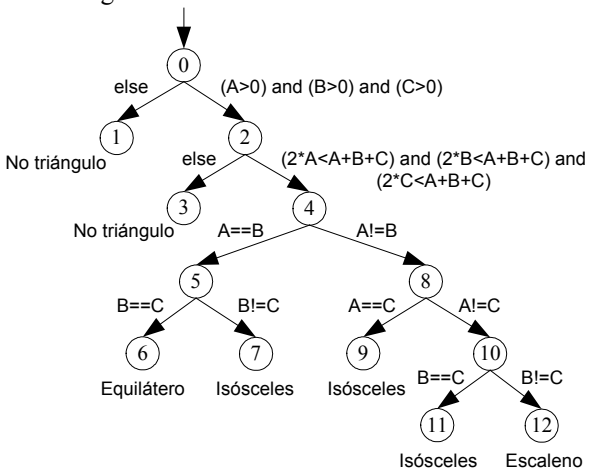

Figura 2: Grafo de control de flujo del problema del triángulo

Cada autor compara su técnica con un algoritmo aleatorio. Por otro lado, es difícil comparar entre sí los resultados obtenidos por los diferentes autores porque cada uno de ellos utiliza rangos distintos para las variables de entrada (y quizás diferente criterio de cobertura).

En la tabla 2 se muestras los resultados obtenidos por Jones, Sthamer y Eyres [9] para un 100% de cobertura de ramas. El algoritmo posee una población de 101 individuos y el máximo número de generaciones es 2000.

Tabla 2: Comparación del algoritmo genético de Jones, Sthamer y Eyres con el algoritmo aleatorio: número de casos de prueba generados y tiempo empleado.

	Aleatorio		Genético	
Rango	**Nº casos prueba**	**Tiempo (seg)**	**Nº casos prueba**	**Tiempo (seg)**
± 100	160752	3.0	17789	8.4
± 200	571983	10.4	48490	23.5
± 400	1967674	37.9	126943	60.4

El algoritmo aleatorio necesita generar un número considerablemente mayor de casos de prueba para lograr el 100% de cobertura de ramas, sin embargo el tiempo total que emplea en alcanzar dicha cobertura es también menor (aproximadamente la mitad del tiempo empleado por el algoritmo genético).

El algoritmo genético diseñado por McGraw, Michael y Schatz [12] tiene como criterio la cobertura de ramas. Estos autores no especifican el tamaño de la población, pero sí el número máximo de generaciones, que se sitúa en 8000. Ante un rango de variables de entrada que cubre todo el dominio entero, el algoritmo aleatorio alcanza el 48.6% de cobertura y el algoritmo genético el 94.29%. Desde el punto de vista de la cobertura alcanzada, el algoritmo genético obtiene mejores resultados. Sin embargo en el citado trabajo no se indican los tiempos empleados ni los casos de pruebas generados.

Pargas, Harrold y Peck tienen como objetivo lograr cobertura de ramas con su algoritmo genético [16]. El tamaño de la población es de 100 individuos y el número máximo de generaciones es el necesario para alcanzar el 100% de cobertura. Los resultados mostrados por los autores se corresponden con un porcentaje de cobertura del 100%, para el cual el algoritmo aleatorio necesita generar 484100 casos de prueba y el algoritmo genético necesita 41500 casos. Puesto que no se indica el rango de las variables de entrada ni se muestran los tiempos empleados, no se puede determinar qué algoritmo se comporta mejor.

El algoritmo genético de Lin y Yeh [11] tiene como objetivo la cobertura de caminos. Para ello emplean una población de 1000 individuos y el número máximo de generaciones es el necesario para alcanzar el camino objetivo. El algoritmo genético necesita generar 10100 casos de prueba para alcanzar el camino más difícil de cubrir. Los autores no indican el rango de las variables de entrada, no muestran los tiempos empleados y no realizan una comparación con el algoritmo aleatorio, por lo que no se puede determinar si su comportamiento es mejor.

El algoritmo propio basado en Búsqueda Tabú tiene como criterio de prueba la cobertura de ramas. El algoritmo concluye cuando se alcanza el 100% de cobertura o se supera un número máximo de iteraciones. Para el ejemplo del triángulo en cada iteración se generan 12 vecinos, 4 por cada variable de entrada. En todos los experimentos se obtuvo el 100% de cobertura y los resultados se muestran en la tabla 3.

Tabla 3: Comparación del algoritmo propio basado en Búsqueda Tabú con el algoritmo aleatorio y un algoritmo genético propio: número de casos de prueba generados y tiempo empleado.

Algoritmo	Rango –64 a 63		Rango –128 a 127		Rango –256 a 255	
	Nº casos de prueba	Tiempo	Nº casos de prueba	Tiempo	Nº casos de prueba	Tiempo
Aleatorio	45210	1.31 seg	178834	2.29 seg	725033	6.08 seg
Genético (propio)	7002	1.43 seg	26948	2.89 seg	105857	8.58 seg
Búsqueda Tabú	587	1.09 seg	1091	1.23 seg	4330	3.0 seg

El algoritmo basado en Búsqueda Tabú necesita generar un número de casos de prueba considerablemente menor que los algoritmos aleatorio y genético para alcanzar el 100% de cobertura. Así mismo el tiempo empleado por el algoritmo basado en búsqueda tabú también es menor que el empleado por los algoritmos aleatorio y genético.

4 Conclusiones

Existen pocos trabajos que utilicen algoritmos genéticos para generación de casos de prueba de cobertura software. En ellos los resultados no reflejan todos los parámetros necesarios para realizar una comparativa (rango de las variables de entrada, tiempos empleados, número de casos de prueba generados). Cada autor compara su técnica con el algoritmo aleatorio pero ninguno realiza la comparación con los otros algoritmos genéticos. Los dos parámetros fundamentales para comparar algoritmos de generación automática de casos de prueba son el porcentaje de cobertura acumulado (mide la eficacia) y el tiempo empleado para alcanzar dicho porcentaje (mide la eficiencia). Por ello para comparar algoritmos de similar eficacia se necesita conocer el tiempo utilizado, sin embargo sólo uno de los autores estudiados proporciona dichos tiempos.

Los resultados aquí mostrados se corresponden con el único benchmark que es utilizado por la mayoría de los autores. De hecho cada autor muestra otros resultados obtenidos para un programa bajo prueba propio del que no existe código fuente disponible y por tanto, es inaccesible para posteriores comparativas con otros algoritmos. La existencia de una serie de benchmarks ampliamente difundidos sería de gran utilidad para realizar las comparaciones entre distintos algoritmos.

Agradecimientos

Este trabajo ha sido financiado por el Ministerio de Ciencia y Tecnología (España) bajo el Programa Nacional para Investigación, Desarrollo e Innovación, proyecto TIC2001-1143-C03-03.

Referencias

1. Beizer, B., Software testing techniques, 2nd. Ed. Van Nostrand Reinhold. 1990
2. Chang, K., Cross, J., Carlisle, W. and Liao, S., A performance evaluation of heuristics-based test case generation methods for software branch coverage, International Journal of Software Engineering and Knowledge Engineering, 6(4), 1996.
3. 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, IEEE Proceedings – Software, 150(3): 161-175, 2003
4. DeMillo, R. A., Offutt, A. J., Constraint-based automatic test data generation, IEEE Transactions on Software Engineering, 17(9). 1991.
5. Glover, F. Tabu search part i, ii, ORSA Journal on Computing, 1(3). 1989.
6. Goldberg, D. Genetic Algorithms in search, optimization, and machine learning, Addison-Wesley, Reading, MA, 1989.
7. Holland, J. H., Adaptation in natural and artificial systems, University of Michigan Press, Ann Arbor, MI, 1975.
8. IEEE Standards Software Engineering, Volume One, 1999
9. Jones, B., Sthamer, H.and Eyers, D., Automatic structural testing using genetic algorithms, The Software Engineering Journal 11, 1996
10. Korel, B., Automated software test data generation, IEEE Transactions on Software Engineering, 16(8), 1990.
11. Lin, J., Yeh, P., Automatic test data generation for path testing using GAs, Information Sciences 131, 2001.
12. McGraw, G., Michael C., Schatz, M., Generating software test data by evolution, Technical Report RSTR-018-97-01, RST Corporation, 1998.
13. Miller, W., Spooner, D., Automatic generation of floating point test data, IEEE Transaction of Software Engineering, SE-2(3), 1976.
14. Myers, G., The art of software testing, 1977.
15. Ntafos, S., On random and partition testing, Intl. Symp. on Software Testing and Analysis, 1998
16. Pargas, R.P., Harrold, M.J., Peck, R.R., Test data generation using genetic algorithms, The journal of software testing, verification and reliability, 9, 1999
17. Pressman, R.S. Ingeniería del Software, Un enfoque práctico, 4ª Edición Ed. McGraw Hill, 1997.
18. Tracey, N., Clark, J., Mander, K., Automated program flaw finding using simulated annealing, International Symposium on software testing and analysis, ACM/SIGSOFT, 1998.