XV Jornadas de Ingeniería del Software y Bases de Datos
JISBD 2006
José Riquelme - Pere Botella (Eds)

TESTEO DE SOFTWARE CON DOS TÉCNICAS METAHEURÍSTICAS

Enrique Alba[1*], Francisco Chicano[1] y Stefan Janson[2]

1: Grupo GISUM, Departamento de Lenguajes y Ciencias de la Computación
E.T.S. Ingeniería Informática
Universidad de Málaga
Campus de Teatinos, 29071, Málaga, España
e-mail: {eat,chicano}@lcc.uma.es, web: http://neo.lcc.uma.es

2: Parallelverarbeitung und Komplexe Systeme
Fakultät für Mathematik und Informatik
Universität Leipzig
Augustusplatz 10/11, 04109 Leipzig
e-mail: janson@informatik.uni-leipzig.de, web: http://pacosy.informatik.uni-leipzig.de

Palabras clave: Testeo de Software, Testeo Evolutivo, Optimización Basada en Nubes de Partículas, Estrategias Evolutivas, Algoritmos Evolutivos, Metaheurísticas

Resumen. *En este artículo analizamos el uso de dos técnicas metaheurísticas para resolver el problema de la generación de casos de prueba en el testeo de software: Optimización Basada en Nubes de Partículas y Estrategias Evolutivas. Este problema es un paso fundamental y tedioso en el desarrollo software. Nosotros resolvemos el problema usando el criterio de cobertura de condiciones, que es más severo que el de cobertura de ramas ampliamente usado en la literatura previa. Comparamos las propuestas usando un conjunto de seis programas objeto que incluyen tanto algoritmos fundamentales en informática como programas complejos reales.*

1 Introducción

La generación automática de casos de prueba consiste en proponer de forma automática un "buen" conjunto de datos de entrada para probar un programa. Intuitivamente podemos decir que un buen conjunto de datos de entrada debería permitir detectar una gran cantidad de errores en un programa incorrecto. Para una definición más formal tenemos que remitirnos al concepto de *criterio de adecuación* (test adequacy criterion) [8].

En este artículo seguimos el paradigma de *testeo estructural* [7]. En este paradigma, el generador de casos de prueba usa información de la estructura del programa para guiar la búsqueda de nuevos datos de entrada (por eso también se conoce como testeo de caja blanca). Normalmente, la información estructural se toma del *grafo de control de flujo* del

programa. Los datos de entrada generados por el testeo estructural deben ser posteriormente contrastados sobre el programa para comprobar si dan lugar a un comportamiento incorrecto. Dentro de este paradigma podemos encontrar varias alternativas en la literatura tales como la *generación aleatoria* [3], la *generación simbólica* [9] y la *generación dinámica* [8] (usada en este artículo); así como combinaciones de las anteriores (DART [5] es un buen ejemplo que combina la generación simbólica y dinámica).

En este artículo analizamos la aplicación de dos técnicas metaheurísticas al problema de testeo de software: Optimización Basada en Nubes de Partículas (PSO) y Estrategias Evolutivas (ES). Hasta nuestro conocimiento, esta es la primera vez que se emplea PSO en este dominio. Las ESs fueron introducidas para el problema por primera vez en [1] por los autores de este trabajo. Además de presentar las técnicas y detallar la propuesta, mostramos resultados experimentales realizados sobre un conjunto de seis programas objeto. El resto del artículo está organizado como sigue. La siguiente sección ofrece una descripción general de los algoritmos empleados (PSO y ES). La Sección 3 presenta los detalles de nuestro generador de casos de prueba. En la Sección 4 analizamos los experimentos realizados y la Sección 5 presenta las conclusiones finales y el trabajo futuro.

2 Algoritmos

En un problema de optimización, el objetivo es generalmente encontrar la mejor de entre varias alternativas. En algunos problemas esto se puede hacer en un tiempo razonable visitando todas las soluciones posibles del espacio de búsqueda o mediante alguna técnica que reduce el problema. Sin embargo, existen problemas no simplificables con un espacio de búsqueda demasiado grande para explorarlo por completo con técnicas exhaustivas. En estos casos tiene sentido aplicar las *técnicas metaheurísticas* [4]. De forma breve podemos definir las metaheurísticas como estrategias de alto nivel estructuradas en etapas para explorar espacios de búsqueda. La información del problema que suelen requerir estas técnicas es una función de adecuación o *fitness* que asigna un valor real a cada solución del espacio de búsqueda indicando su calidad. A continuación describimos brevemente las dos metaheurísticas que usamos en este artículo: PSO y ES.

2.1 Optimización Basada en Nubes de Partículas

El algoritmo PSO [6] mantiene un conjunto de soluciones (vectores de reales), también llamadas *partículas*, que son inicializadas aleatoriamente en el espacio de búsqueda. El movimiento de una partícula $\mathbf{x}^i$ está influenciado por su velocidad $\mathbf{v}^i$ (otro vector de reales) y las posiciones donde se encontraron buenas soluciones. En la versión estándar del PSO, la velocidad de una partícula depende de la mejor solución global que ha sido encontrada. Sin embargo, existe otra variante en la que cada partícula tiene asociado un conjunto de partículas vecinas y la velocidad se actualiza de acuerdo a la mejor solución encontrada por alguna partícula de dicho vecindario. La actualización de la velocidad y la posición de cada partícula viene dada por las siguientes ecuaciones:

$$v_j^i(t+1) = w \cdot v_j^i(t) + c_1 \cdot r_1 \cdot (p_j^i - x_j^i(t)) + c_2 \cdot r_2 \cdot (n_j^i - x_j^i(t)) \quad (1)$$
$$x_j^i(t+1) = x_j^i(t) + v_j^i(t) \quad (2)$$

donde w es la *inercia* y controla la influencia de la velocidad anterior, c_1 y c_2 permiten ajustar la influencia de la mejor solución personal (p_j^i) y la mejor solución del vecindario (n_j^i) y los valores r_1 y r_2 son números reales aleatorios dentro del intervalo $[0, 1]$ generados en cada iteración. Para una descripción más detallada del algoritmo consultar [6].

2.2 Estrategia Evolutiva

ES [11] pertenece a una familia de algoritmos basados en población denominados Algoritmos Evolutivos (EA) por su inspiración en la evolución natural [2]. En una ES existe un conjunto de μ soluciones tentativas (individuos) que evolucionan durante la ejecución del algoritmo. Cada individuo está formado por tres vectores de reales: el vector con la solución $\mathbf{x}$, un vector con desviaciones estándar σ y un vector con ángulos ω. Los dos últimos se usan en el operador de variación principal del algoritmo: la mutación Gausiana, para incorporar auto-adaptación a la búsqueda. Al principio, este conjunto de soluciones se crea de forma aleatoria. Después, el algoritmo entra en un bucle donde λ individuos son elegidos de forma aleatoria de la población, se les aplica el operador de mutación y se les evalúa. La siguiente generación se forma escogiendo los mejores μ individuos de la unión de la población previa y los λ hijos. Este bucle se detiene cuando se cumple un cierto criterio de parada, por ejemplo, cuando se encuentra una solución óptima o se alcanza un número determinado de evaluaciones. Para más detalles del algoritmo ver [11].

3 El Generador de Casos de Prueba

En la generación dinámica el programa se instrumenta para pasar información al generador, que comprueba si el criterio de adecuación se cumple o no con el conjunto de datos propuesto. En caso negativo crea nuevos datos de entrada para el programa. La generación de casos de prueba es transformada así en un problema de minimización, donde la función a minimizar es algún tipo de "distancia" a una ejecución donde el criterio de adecuación está más cerca de cumplirse. Por tanto, para diseñar un generador de casos de prueba dinámico necesitamos determinar el criterio de adecuación, la función de distancia, la instrumentación del programa y los algoritmos de búsqueda. Ya hemos presentado los algoritmos en la sección anterior y a continuación se discutirán el resto de los aspectos mencionados junto con otros dos: el proceso completo de generación de casos de prueba y las métricas de cobertura de programas.

3.1 El Criterio de Adecuación

En este artículo usamos como criterio de adecuación el criterio de *cobertura de condiciones* [1, 8]. Este criterio requiere que todas las condiciones atómicas de un programa

objeto tomen los dos valores lógicos: *true* y *false*. Otros criterios muy extendidos son el de cobertura de ramas, que busca pasar por todas las ramas del programa y el de cobertura de instrucciones, en el que todas las instrucciones del programa deben ejecutarse. El criterio de cobertura de condiciones es más severo que los otros dos, es decir, si encontramos un conjunto de datos de entrada para el que todas las condiciones toman los dos valores lógicos podemos asegurar que todas las ramas factibles serán tomadas y, como consecuencia, todas las instrucciones alcanzables serán ejecutadas. No obstante, lo inverso no es cierto: ejecutar todas las instrucciones alcanzables o tomar todas las ramas factibles no asegura que todas las condiciones tomen los valores lógicos posibles. Hemos elegido el criterio de cobertura de condiciones por ser más estricto que los otros dos.

Nuestro generador descompone el objetivo global (el criterio de cobertura de condiciones) en varios objetivos parciales consistiendo cada uno en hacer que una condición tome un determinado valor lógico [1]. Después, cada objetivo parcial es tratado como un problema de optimización en el que la función a minimizar es una distancia entre la entrada actual y una entrada que satisface el objetivo parcial. Para resolver el problema de minimización usamos las técnicas de optimización global presentadas en la Sección 2.

3.2 Función de Distancia

La función de distancia depende de la expresión de la condición atómica particular asociada al objetivo parcial y de los valores de las variables del programa cuando la condición es alcanzada. Por tanto, sólo puede ser calculada si el flujo del programa alcanza dicha condición atómica, en otro caso la distancia toma por defecto el mayor valor posible para los números reales en una máquina. En la Tabla 1 mostramos la función de distancia para cada tipo de condición y cada valor lógico deseado. Las expresiones de las funciones de distancia están pensadas para que su minimización implique la satisfacción del objetivo parcial. Algunas de ellas aparecen en trabajos previos [8].

Condición Atómica	Expresión para *true*	Expresión para *false*
$a < b$	$a - b$	$b - a$
$a <= b$	$a - b$	$b - a$
$a == b$	$(b - a)^2$	$(1 + (b - a)^2)^{-1}$
$a \mathrel{!}= b$	$(1 + (b - a)^2)^{-1}$	$(b - a)^2$
a	$(1 + a^2)^{-1}$	a^2

Tabla 1: Funciones de distancia para los distintos tipos de condiciones y valores lógicos. Las variables a y b son numéricas (enteras o reales)

3.3 Instrumentación del Programa Objeto

Para obtener información sobre el valor de distancia y las condiciones alcanzadas durante una ejecución añadimos ciertas instrucciones al código fuente del programa objeto. En nuestro caso lo hacemos de forma automática con una aplicación desarrollada por nosotros mismos que analiza el código fuente en C y genera un nuevo programa modificado listo para ejecutar el programa original y devolver información sobre su funcionamiento.

Durante la evaluación de una entrada, cuando el generador ejecuta el programa objeto modificado, se elabora un informe (condiciones alcanzadas y los valores de distancia) que es transmitido al generador. Con esta información el generador mantiene una *tabla de cobertura* donde almacena por cada condición dos conjuntos de datos de entrada: los que hacen la condición cierta y los que la hacen falsa. Diremos que una condición ha sido *alcanzada* si al menos uno de los conjuntos es no vacío. Por otro lado, diremos que una condición ha sido *cubierta* si los dos conjuntos son no vacíos.

3.4 El Proceso de Generación

El bucle principal del generador se muestra en la Figura 1. Al comienzo de la generación se crean algunas entradas aleatorias (10 en nuestros experimentos) que alcanzan sólo algunas condiciones. Después, comienza el bucle principal del generador donde, en primer lugar, se selecciona un objetivo parcial no cubierto, es decir, una condición alcanzada pero no cubierta. La elección del objetivo parcial no es aleatoria, siempre se elige un objetivo parcial con una condición asociada previamente alcanzada. En concreto, se elige el objetivo que cumpliendo lo anterior es el siguiente al objetivo recién abordado.

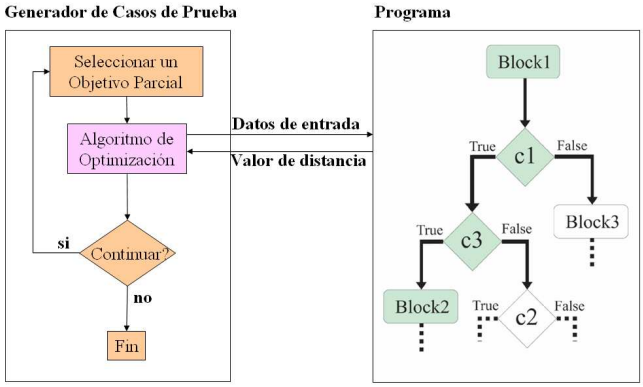

Figura 1: Generación de casos de prueba

Cuando el objetivo ha sido elegido, se usa el algoritmo de optimización para buscar casos de prueba que hagan que la condición tome el valor no cubierto aún. El algoritmo de optimización es inicializado con al menos una entrada que permite alcanzar la condición elegida. El algoritmo explora diferentes entradas y usa los valores de distancia para guiar la búsqueda. Durante esta búsqueda se pueden encontrar datos que cubran otros objetivos parciales aún por satisfacer. Estos datos son usados también para actualizar la tabla de cobertura. Tras ejecutar el algoritmo de optimización e independientemente del éxito de la búsqueda, el cuerpo del bucle principal se ejecuta de nuevo y se elige otro objetivo parcial (el siguiente). Este esquema se repite hasta que se consigue cobertura total o se alcanza un número preestablecido de fracasos del algoritmo de optimización.

3.5 Métricas de Cobertura

Para terminar con la descripción del generador debemos discutir las métricas de cobertura usadas para presentar los resultados del generador. La métrica más simple es el cociente entre los objetivos parciales cubiertos y el número total de objetivos parciales. Este valor expresado en porcentaje se conoce como *porcentaje de cobertura* (porcentaje de cobertura de condiciones, en nuestro caso). Aunque ésta es la forma más simple de medir la eficacia del generador, no es la más apropiada. La razón es que existen programas para los que es imposible conseguir una cobertura total, ya que tienen objetivos parciales inalcanzables. En este caso se produce una pérdida de cobertura que es independiente de la técnica usada para la generación de casos de prueba. Por ejemplo, un bucle infinito tiene una condición que siempre es verdadera y nunca falsa. En este caso hablamos de *pérdida de cobertura dependiente del código*. No obstante, hay otro factor que puede producir una pérdida de cobertura inevitable: el entorno en el que el programa se ejecuta. Por ejemplo, si un programa pide una pequeña cantidad de memoria dinámica y después comprueba si la asignación tuvo éxito lo más probable es que lo tenga en todas las ejecuciones del programa y la condición que comprueba si hay error sea siempre falsa. En este caso decimos que hay una *pérdida de cobertura dependiente del entorno*. Cuando se da una de estas situaciones ningún generador de casos de prueba es capaz de conseguir cobertura total y puede parecer ineficaz cuando, en realidad, no es así.

Nosotros buscamos una métrica de cobertura que tenga en cuenta las pérdidas en la medida de lo posible. Por ello hemos introducido otra métrica que denominamos *cobertura corregida* y que se calcula como el cociente entre el número de objetivos parciales cubiertos y alcanzables. Esta medida, que es la que usamos en los experimentos, evita la pérdida dependiente del código. La pérdida dependiente del entorno es más difícil de evitar y la nueva métrica no la considera. Con la cobertura corregida podemos ordenar los programas de acuerdo a su dificultad para un generador dado. Si usáramos la medida simple de cobertura podríamos clasificar a un programa como difícil cuando, en realidad, tiene muchos objetivos parciales inalcanzables pero los objetivos alcanzables son fáciles de cubrir.

4 Experimentos

En esta sección presentamos los experimentos realizados sobre un conjunto de seis programas en C. En primer lugar hemos usado dos programas sin bucles: `triangle` que posee numerosas condiciones complejas comprobando la igualdad de enteros y `calday` con objetivos parciales dispersos en el espacio de búsqueda. De entre los programas con bucles, `select` lo hemos escogido como paradigma de los algoritmos de ordenación (incluye el algoritmo de ordenación shell), `bessel` representa a los programas de cálculo numérico y `sa` y `netflow` son programas complejos con aplicación real: resolución de una instancia del viajante de comercio (TSP) con enfriamiento simulado y optimización de una red de comunicaciones. La mayoría de los códigos fuente se han extraído del libro "C Recipes" [10]. Los programas están listados en la Tabla 2, donde presentamos información

sobre el número de condiciones, las líneas código (LdC), el número de argumentos de entrada, una breve descripción de su objetivo y cómo se pueden conseguir.

Programa	Conds.	LdC	Args.	Descripción	Fuente
`triangle`	21	53	3	Clasifica triángulos	Ref. [8]
`calday`	11	72	3	Calcula el día de la semana	`julday`
`select`	28	200	21	k-ésimo elemento de una lista desordenada	`selip`
`bessel`	21	245	2	Funciones de Bessel J_n y Y_n	`bessj*,bessy*`
`sa`	30	332	23	Enfriamiento simulado	`anneal`
`netflow`	55	112	66	Optimización de una red	Wegener [13]

Tabla 2: Programas objeto usados en los experimentos. La columna "Fuente" contiene el nombre de la función en C-Recipes

4.1 Representación y Función de Fitness

Los valores de entrada de los programas objeto usados en los experimentos son números reales o enteros. Tanto en PSO como en ES se representan con un valor real. Este valor se redondea a entero cuando el argumento que representa es de ese tipo. La ventaja de usar esta representación es que podemos explorar el espacio de búsqueda completo. Esto contrasta con otras técnicas que limitan el dominio de los argumentos de entrada a una región acotada, como ocurre en [12].

La función de fitness usada en la búsqueda no es exactamente la función de distancia. Queremos evitar valores negativos de fitness para poder aplicar operadores de selección que dependen del valor absoluto de la función de fitness tales como la selección por ruleta. Por esta razón usamos la siguiente expresión:

$$fitness(\mathbf{x}) = \pi/2 - \arctan(distance(\mathbf{x})) + 0.1 \tag{3}$$

En la expresión anterior multiplicamos el `arctan` por -1 debido a que nuestros algoritmos están diseñados para maximizar la función de fitness. Además, necesitamos añadir el valor $\pi/2$ a la expresión para obtener siempre un valor positivo. Finalmente, el 0.1 se usa para no obtener valores negativos cuando existe alguna pérdida de precisión en el cálculo.

4.2 Resultados Computacionales

En esta sección presentamos los resultados obtenidos por los generadores de casos de prueba con PSO y ES como motores de búsqueda. En ambos casos hemos realizado un estudio previo de parámetros y mostramos los resultados obtenidos con la mejor parametrización. En el caso del PSO los mejores resultados se obtuvieron al incorporar un operador que añade un valor aleatorio a las velocidades de la mitad de las partículas (escogidas aleatoriamente) si la mejor solución de la nube no mejora tras una iteración. La magnitud de la perturbación añadida va creciendo exponencialmente (se multiplica por 10) conforme aumenta el número de iteraciones consecutivas sin mejora. Los parámetros de los algoritmos se presentan en la Tabla 3.

Realizamos 30 ejecuciones independientes de cada generador para cada programa. En la Tabla 4 presentamos la media de la cobertura corregida y del número de evaluaciones

PSO		ES	
Partículas	10	Población	25
w	0.729	Selección	Aleatoria
$c1$	1.494	Mutación	Gausiana
$c2$	1.494	Hijos(λ)	5
Prob. de perturb.	0.5	Reemplazo	$(\mu + \lambda)$
Parada	Objetivo o 1000 evaluaciones	Parada	Objetivo o 1000 evaluaciones

Tabla 3: Parámetros de los algoritmos

necesarias para obtener la cobertura alcanzada. Hemos añadido a la comparativa los resultados obtenidos al usar un algoritmo genético (GA) como motor de búsqueda, ya que ha sido hasta ahora el método más empleado en el testeo de software con metaheurísticas. Los parámetros usados en el GA son también los que mejores resultados obtuvieron en un estudio previo. Estos parámetros son: cruce de dos puntos (con probabilidad 1.0) y mutación mediante perturbación aleatoria siguiendo una distribución normal de media 0 y varianza 1 (con probabilidad 0.6). La población del GA es de 25 individuos y el número de hijos es 5 (igual que en ES) [1].

Program	PSO		ES		GA	
	Cov.	Evals.	Cov.	Evals.	Cov.	Evals.
`triangle`	93.98	11295.77	**99.84**	**2370.03**	99.67	3209.47
`calday`	**100.00**	179.33	98.18	3166.47	90.91	**75.03**
`select`	**88.89**	380.13	83.33	**13.27**	83.33	83.20
`bessel`	**97.56**	**116.90**	**97.56**	350.63	**97.56**	533.03
`sa`	**100.00**	**165.67**	99.94	2337.30	96.72	176.63
`netflow`	97.77	4681.70	**98.17**	**307.77**	96.42	917.90

Tabla 4: Resultados obtenidos con PSO y ES. Mostramos también resultados previos obtenidos con GA

En primer lugar, podemos observar que GA obtiene de forma general la peor cobertura. En segundo lugar, comparando PSO y ES observamos que su eficacia es aproximadamente la misma. En `calday`, `select`, y `sa` la cobertura media que obtiene PSO es mayor que la de ES, pero ES supera a PSO en `triangle` y `netflow`. Además, un análisis estadístico (que no mostramos por falta de espacio) demuestra que las únicas diferencias significativas son las de `triangle` y `select`.

Hemos observado que en `triangle` PSO obtiene menor cobertura que ES debido a la perturbación que añadimos a la técnica (los experimentos previos que realizamos para decidir los parámetros de los algoritmos así lo demuestran). La perturbación después de cada paso del algoritmo aumenta de forma drástica la capacidad de exploración del algoritmo. Eso ayuda de forma crucial a programas como `calday` o `select`, que poseen objetivos parciales en lugares muy alejados de la región donde se encuentran las partículas inicialmente. Sin embargo, la densidad de objetivos parciales en `triangle` es mayor en la región inicial y la exploración perjudica la búsqueda en ese caso. Por otro lado, ES tiene una mayor componente de explotación y menor de exploración, lo cual explica que no alcance la cobertura que obtiene PSO en `calday` y `select`.

El uso de la cobertura corregida en la presentación de nuestros experimentos ha permitido eliminar la pérdida debido al código, sin embargo, podemos observar que los mejores

resultados de cobertura en algunos problemas se mantienen por debajo del 100%. Esto se debe en parte a la pérdida debida al entorno (`select`, por ejemplo, usa memoria dinámica) y a la ineficacia del generador (en `bessel` existe una condición de igualdad entre números reales, que es muy difícil de satisfacer con técnicas de generación dinámica).

La comparación del número de evaluaciones es justa solamente cuando los algoritmos obtienen la misma cobertura. En otro caso, podemos concluir que el algoritmo que requiere menos evaluaciones es mejor si la cobertura obtenida es mayor. Este es el caso del PSO en los programas `bessel` y `sa` y de ES en el programa `triangle`.

Es importante mencionar que para `netflow` los algoritmos encuentran varias entradas para las que no termina la ejecución (nunca antes reportado), lo cual es un claro ejemplo de que la metodología seguida funciona y tiene utilidad práctica. No obstante, para la realización de los experimentos tuvimos que incorporar un mecanismo que detiene el programa objeto cuando lleva cierto tiempo en ejecución. En este programa en particular nuestra ES obtiene un 98.17% de cobertura de condiciones con 308 evaluaciones que, analizando el código fuente, resulta ser equivalente al 99% de cobertura de ramas que Wegener et al. [13] obtienen usando 40703 evaluaciones. La conclusión es que obtenemos la misma cobertura con una centésima parte del número de evaluaciones, a pesar del hecho de que ellos usan una función de fitness más sofisticada.

5 Conclusiones

En este artículo hemos presentado el uso de la Optimización Basada en Nubes de Partículas y las Estrategias Evolutivas para resolver el problema del testeo de software. Además de detallar la propuesta hemos realizado una serie de experimentos sobre un conjunto de seis programas que implementan algoritmos representativos de diversos dominios de la Informática. Los resultados demuestran que ambas técnicas poseen una eficacia similar en general. Mientras que en algunos problemas PSO aventaja a ES, en otros los mejores resultados son obtenidos por ES. De cualquier forma, concluimos que ambos algoritmos resultan mejores que GA, el cual es la base de numerosos trabajos en el dominio.

Como trabajo futuro, planeamos hibridar ambas técnicas para tratar de aunar lo mejor de cada una de ellas. Hemos visto en la sección experimental que nuestro PSO con perturbación tiene una importante componente exploradora y ES, por el contrario, explota mejor las regiones del espacio de búsqueda. La combinación de ambas técnicas podría dar lugar al equilibrio necesario de exploración y explotación que permite abordar todos los programas de forma robusta. También pensamos extender la técnica para poder usar datos de entrada no numéricos como cadenas de caracteres, registros y arrays, entre otros.

Agradecimientos

Agradecemos a Joachim Wegener el acceso al código fuente del programa `netflow`. Este trabajo está parcialmente financiado por el Ministerio de Educación y Ciencia y FEDER con número de proyecto TIN2005-08818-C04-01 (proyecto OPLINK). Francisco Chicano disfruta de una beca de la Junta de Andalucía (BOJA 68/2003).

REFERENCIAS

[1] E. Alba and J. F. Chicano. Software testing with evolutionary strategies. In *Proceedings of the 2nd International Workshop on Rapid Integration of Software Engineering Techniques*, volume 3943 of *LNCS*, pages 50–65, Heraklion, Greece, September 2005.

[2] T. Bäck. *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms*. Oxford University Press, NY, 1996.

[3] D. Bird and C. Munoz. Automatic generation of random self-checking test cases. *IBM Systems Journal*, 22(3):229–245, 1983.

[4] C. Blum and A. Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Computing Surveys*, 35(3):268–308, 2003.

[5] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 213–223, New York, NY, USA, 2005. ACM Press.

[6] J. Kennedy. Small worlds and mega-minds: effects of neighborhood topology on particle swarm performance. In *Proceedings of IEEE Congress on Evolutionary Computation (CEC 1999)*, pages 1931–1938, Piscataway, NJ, USA, 1999.

[7] P. McMinn. Search-based software test data generation: a survey. *Software Testing, Verification and Reliability*, 14(2):105–156, June 2004.

[8] C. C. Michael, G. McGraw, and M. A. Schatz. Generating software test data by evolution. *IEEE Transactions on Software Engineering*, 27(12):1085–1110, 2001.

[9] J. Offutt. An integrated automatic test data generation system. *Journal of Systems Integration*, 1(3):391–409, November 1991.

[10] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C. The Art of Scientific Computing*. Cambridge University Press, 2002. Available online at http://www.library.cornell.edu/nr/bookcpdf.html.

[11] G. Rudolph. *Evolutionary Computation 1. Basic Algorithms and Operators*, volume 1, chapter 9, Evolution Strategies, pages 81–88. IOP Publishing Lt, 2000.

[12] R. Sagarna and J.A. Lozano. Variable search space for software testing. In *Proceedings of the International Conference on Neural Networks and Signal Processing*, volume 1, pages 575–578. IEEE Press, December 2003.

[13] J. Wegener, A. Baresel, and H. Sthamer. Evolutionary test environment for automatic structural testing. *Information and Software Technology*, 43(14):841–854, 2001.