Generación dinámica de casos de prueba utilizando metaheurísticas

Juan Pablo La Battaglia
III-LIDI, Facultad de Informática, UNLP,
La Plata, Argentina, 1900
juanlb@gmail.com

y

Laura Lanzarini
III-LIDI, Facultad de Informática, UNLP,
La Plata, Argentina, 1900
laural@lidi.info.unlp.edu.ar

Abstract

The resolution of optimization problems is of great interest nowadays and has encouraged the development of various information technology methods to attempt solving them. There are several problems related to Software Engineering that can be solved by using this approach. In this paper, a new alternative based on the combination of population metaheuristics with a Tabu List to solve the problem of test cases generation when testing software is presented. This problem is of great importance for the development of software with a high computational cost and which is generally hard to solve.

The performance of the solution proposed has been tested on a set of varying complexity programs. The results obtained show that the method proposed allows obtaining a reduced test data set in a suitable timeframe and with a greater coverage than conventional methods such as Random Method or Tabu Search.

Key words: Software Testing, Evolutionary Testing, Particle Swarm Optimization, Evolutionary Algorithms, Metaheuristics.

Resumen

La resolución de problemas de optimización es de gran interés en la actualidad y ha motivado el desarrollo de diversos métodos informáticos para tratar de resolverlos. Existen varios problemas pertenecientes a la Ingeniería de Software que pueden ser resueltos utilizando este enfoque. En este artículo se presenta una nueva alternativa basada en la combinación de una metaheurística poblacional con una lista Tabú para resolver el problema de la generación de casos de prueba en el testeo de software. Este problema es una tarea de suma importancia en el desarrollo de software que requiere un alto costo computacional y generalmente es difícil de resolver.

El desempeño de la solución propuesta ha sido probado sobre un conjunto de programas de distinta complejidad. Los resultados obtenidos muestran que el método propuesto permite obtener un conjunto de datos de prueba reducido, en un tiempo adecuado y con una cobertura superior a los métodos convencionales tales como Generación Random o Tabú Search.

Palabras Claves: Testeo de Software, Testeo Evolutivo, Optimización Basada en cúmulos de partículas, Algoritmos Evolutivos, Metaheurísticas.

1. Introducción

La generación automática de un conjunto de datos de prueba que permita medir el desempeño de un programa dado es una tarea de suma importancia en el desarrollo de software, que requiere un alto costo computacional y que generalmente es difícil de resolver.

La solución de este problema ha sido ampliamente investigada desde hace mucho tiempo. El primer paradigma utilizado fue el denominado *"generación de datos de prueba random*" que consistió en crear el conjunto de datos de prueba de manera aleatoria hasta que la condición de terminación fuera alcanzada o hasta que se hubieran generado un número máximo de conjuntos de prueba [Bir83].

Otra alternativa utilizada para resolver este problema es la *generación de datos de prueba simbólica*[Off91]. Consiste en utilizar valores simbólicos para las variables, en lugar de valores reales, permitiendo realizar una ejecución simbólica. A partir de dicha ejecución se obtienen restricciones algebraicas que determinan los casos de prueba.

Un tercer paradigma es la *generación dinámica de datos de prueba* [Mich91]. En este caso, el programa es modificado para brindarle información al generador permitiéndole verificar si determinado criterio fue alcanzado o no. De esta forma, si el criterio no fue alcanzado, podrá construir nuevos datos que servirán como entrada al programa. Bajo este paradigma, la generación de datos de prueba se convierte en un proceso de optimización ya que cada condición dentro del programa puede ser analizada como una función a minimizar. En particular, se han utilizado distintas metaheurísticas con el objetivo de generar dinámicamente los casos de prueba necesarios. Existen soluciones basadas en algoritmos genéticos[Par99], simulated annealing [Tra98] y sistemas inmunes [Bou07]. Algunas soluciones más recientes utilizan Tabu Search [Dia03] y Scatter Search [Sag06].

El objetivo de este artículo es presentar una nueva solución al problema de determinar el conjunto de datos de prueba adecuado para testear el desempeño de un programa utilizando una metaheurística poblacional basada en PSO combinada con una lista Tabú.

Se llevará a cabo un testeo de caja blanca, es decir que el generador de casos de prueba utilizará información de la estructura del programa para guiar la búsqueda de nuevos datos de entrada. 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.

2. Optimización mediante Cúmulos de Partículas

Un algoritmo basado en Cúmulos de Partículas, también llamado Particle Swarm Optimization (PSO), es una metaheurística poblacional donde cada individuo representa una posible solución del problema y realiza su adaptación teniendo en cuenta tres factores: su conocimiento sobre el entorno (su valor de fitness), su conocimiento histórico o experiencias anteriores (su memoria), el conocimiento histórico o experiencias anteriores de los individuos situados en su vecindario [Ken95]. Su objetivo es evolucionar su comportamiento de manera de asemejarse a los individuos con más éxito dentro de su entorno. En este tipo de técnica, cada individuo permanece en continuo movimiento dentro del espacio de búsqueda y nunca muere. Por su parte, la población puede verse como un sistema multiagente donde cada individuo o partícula se mueven dentro del espacio de búsqueda guardando y eventualmente comunicando, la mejor solución que ha encontrado [Nie06].

Existen distintas versiones de PSO; las más conocidas son *gBest PSO* que utiliza como criterio de vecindad a la población completa y *lBest PSO* que, por el contrario, utiliza un tamaño de vecindad pequeño [Ken95][Shi99]. El tamaño de la vecindad influye en la velocidad de convergencia del algoritmo así como en la diversidad de los individuos de la población. A mayor tamaño de vecindad, la convergencia del algoritmo es más rápida pero la diversidad de individuos es menor.

Cada partícula p_i está compuesta por tres vectores y dos valores de fitness:

- El vector $x_i = (x_{i1}, x_{i2}, \ldots, x_{in})$ almacena la posición actual de la partícula en el espacio de búsqueda.
- El vector $pBest_i = (p_{i1}, p_{i2}, \ldots, p_{in})$ almacena la posición de la mejor solución encontrada por la partícula hasta el momento.
- El vector velocidad $v_i = (v_{i1}, v_{i2}, \ldots, v_{in})$ almacena el gradiente (dirección) según el cual se moverá la partícula.
- El valor de fitness $fitness\_x_i$ almacena el valor de aptitud de la solución actual (vector x_i).

- El valor de fitness $fitness\_pBest_i$ almacena el valor de aptitud de la mejor solución local encontrada hasta el momento (vector $pBest_i$)

La posición de una partícula se actualiza se la siguiente forma

$$x_i(t+1) = x_i(t) + v_i(t+1) \quad (1)$$

Como se explicó anteriormente, el vector velocidad se modifica teniendo en cuenta su experiencia y la de su entorno. La expresión es la siguiente:

$$v_{ij}(t+1) = w \,.\, v_i(t) + \varphi_1 \,.\, rand_1 \,.\, (pBest_i - x_i(t)) + \varphi_2 \,.\, rand_2 \,.\, (g_i - x_i(t)) \quad (2)$$

donde *w* representa el factor de inercia [Shi98], φ_1 y φ_2 son constantes de aceleración, *rand1* y *rand2* son valores aleatorios pertenecientes al intervalo (0,1) y g_i representa la posición de la partícula con el mejor *pBest_fitness* del entorno de p_i (*lBest* o *localbest*) o de todo el cúmulo (*gBest* o *globalbest*). Los valores de *w*, φ_1 y φ_2 son importantes para asegurar la convergencia del algoritmo. Para más detalles sobre la elección de estos valores puede consultar [Cle02] y [Ber02].

La figura 1 contiene el algoritmo PSO básico

```
S ← InicializarCumulo()
while no se alcance la condición de terminación do
    for i = 1 to size(S)  do
       evaluar la partícula xi del cúmulo S
       if fitness(xi) es mejor que fitness(pBesti) then
             pBesti ← xi; fitness(pBesti) ← fitness(xi)
        end if
    end for
    for i = 1 to size(S)  do
        Elegir gi según el criterio de vecindad utilizado
        vi ← w . vi + φ1 . rand1 . (pBesti − xi) + φ2 . rand2 . (gi − xi)
        xi ← xi + vi
    end for
end while
Salida : la mejor solución encontrada
```

Figura 1

3. Descripción de la solución propuesta

La generación dinámica de casos de prueba implica conocer durante la ejecución del programa si el criterio de cobertura se cumple o no. Para ello, se modifica el programa original insertando instrucciones que le permitan al generador recolectar la información necesaria. Nuevos datos de entrada son incorporados al conjunto de prueba hasta que el criterio deseado sea alcanzado. De esta forma, este problema perteneciente al área de la Ingeniería de Software es transformado en un problema de optimización ya que se busca minimizar una cierta distancia a un criterio de cobertura preestablecido. El método utilizado para realizar dicha minimización está basado en el algoritmo de optimización mediante cúmulo de partículas.

3.1. Criterio de Cobertura

El criterio utilizado en este artículo, para determinar si un programa se encuentra adecuadamente cubierto o no, es el de cobertura de condiciones. Esto implica que todas las condiciones atómicas del programa deben tomar los dos valores de verdad posibles. Existen otros criterios como el de cobertura de instrucciones que requiere la ejecución de todas las instrucciones del programa o la cobertura de ramas que busca pasar por todas las ramas del programa. Sin embargo, el criterio seleccionado, al exigir que

todas las condiciones alcancen ambos valores de verdad, garantiza que todas las ramas serán cubiertas y por lo tanto, todas las instrucciones del programa serán ejecutadas; por este motivo ha sido seleccionado.

Para llevar a cabo esta tarea, cada una de las condiciones del programa es analizada de manera independiente. Para cada una de ellas se utiliza la estrategia descripta en la sección anterior. Dado que la verificación del estado de cada condición implica la ejecución del programa, podrá avanzarse en el criterio de cobertura en más de una condición durante una misma ejecución.

3.2. Modificaciones al método PSO

El método de optimización utilizado es una versión modificada del algoritmo básico de PSO considerando las particularidades del problema a resolver:

- La optimización es multi-objetivo utilizando una población diferente para cada condición.
- PSO, por ser una estrategia de optimización, desplaza los individuos de la población dentro del espacio de soluciones buscando acercarse al óptimo. Esto lleva en ocasiones a la pérdida de diversidad [Bir06]. En el caso de la generación de los casos de prueba, la función a minimizar para cada condición es una expresión que permite invertir el valor de verdad. Para que funcione adecuadamente es preciso conservar la inercia de cada partícula; es decir, que el parámetro *w* no se utiliza de la manera habitual.
- Cada población asociada a una condición está formada por individuos que permiten evaluarla. Todos ellos darán como resultado el mismo valor de verdad. El objetivo del método aquí propuesto es utilizarlos para obtener el valor de verdad contrario. Es importante notar que, la ejecución del programa utilizando como entrada uno de estos individuos en su nueva posición podría no llegar a evaluar la condición deseada impidiendo la asignación de un valor de aptitud. Esto sería equivalente a utilizar un espacio de soluciones no continuo, donde los individuos al moverse quedan fuera del espacio de interés. Por tal motivo, se ha modificado PSO a fin de permitir sólo los desplazamientos que se mantienen dentro del espacio de soluciones; el resto de los individuos conserva su posición actual.

3.3. Función de aptitud

La solución aquí propuesta sólo se aplica a variables de entrada numéricas. La función de aptitud utilizada en cada caso es la indicada en la tabla 1 y tiene por objetivo retornar un valor positivo el cual se irá acercando a cero en la medida en que el individuo de la población, que representa los datos de entrada utilizados para la ejecución del programa, avance en la dirección adecuada para obtener el valor de verdad contrario.

Tabla 1 : Función de aptitud utilizada para cada tipo de condición

Condición	Función de fitness
x=y, x≠y	abs(x-y)
x<y, x≤y	y-x
x>y, x≥y	x-y
x∧y	Min(cost(x),cost(y))

x∨y	if x = TRUE and y = TRUE then
	Min(cost(x),cost(y))
	else $\sum_{c_i FALSE} \cos t(c_i)$
	end if

3.4. Método de Generación

El método para la generación de casos de prueba elegido es del tipo *caja blanca*, por lo tanto es necesario conocer el valor de las variables involucradas en cada condición en el momento de la ejecución.

Para lograr esto se utilizó un método compuesto de dos módulos

- Un asistente de ejecución, que dependiendo de algunos símbolos introducidos en el código fuente (inocuos para la ejecución), genera información sobre los valores de cada variable.

- Un proceso que tomando por entrada cualquier programa que se quiera tratar, agrega los símbolos antedichos.

Toda esta información es evaluada en forma automática por el generador de casos de prueba.

La Figura 2 resume el método propuesto.

```
 1. P ← CrearEstructuraInicial {todas las poblaciones están vacías}
 2. DatosPrueba ← Generar_1_solucion_AlAzar
 3. EjecutarPrograma(DatosPrueba)
    {las condiciones que fueron alcanzadas, ahora tienen un individuo
    en su población}
 4. ListaRta = DatosPrueba
 5. while no se alcance la condición de terminación do
 6.    idNoC ← {identificar la 1ra.condición evaluada y no cubierta}
 7.    DatosPrueba ← Aplicar_PSO_Modificado( P(idNoC) )
 8.    for i = 1 to size(DatosPrueba)  do
 9.      if DatosPrueba(i) no fue probado then
10.        cambios = EjecutarPrograma(DatosPrueba(i))
11.        if cambios>0 then
12.          Agregar DatosPrueba(i) al conj. ListaRta;
13.        end if
14.        Agregar DatosPrueba(i) a la lista de los ya probados.
15.      end if
16. end while
```

Figura 2

En este pseudocódigo, la estructura P contiene tantas poblaciones como condiciones haya en el programa. El primer juego de datos de entrada se genera al azar. Este se utiliza para ejecutar el programa y se registra en cada una de las poblaciones cuya condición haya podido ser evaluada.

La condición de terminación utilizada (línea 5) es haber alcanzado una cantidad máxima de generaciones o haber cubierto todas las condiciones, lo que ocurra primero.

Las condiciones se ordenan según su aparición dentro del código. Luego de la primera ejecución al menos una condición ha sido evaluada.

En cada iteración se identifica la primera condición evaluada y no cubierta y se utiliza su población para generar nuevos datos de entrada utilizando una versión modificada de PSO (línea 6). Cuando la población contiene un único individuo, se generan a partir de él doce variaciones, la mitad dentro del 10% del rango permitido y la otra mitad un poco más lejos, dentro del 50% de dicho rango. Si la población contiene más de un individuo se aplica una variante de PSO global. El valor gBest se obtiene promediando los vectores de posición de los dos mejores individuos. Todos los individuos de la población excepto los dos mejores calculan su vector velocidad de la siguiente forma:

$$v_i \leftarrow 0.75 \cdot rand_1 \cdot v_i + 0.75 \cdot rand_2 \cdot (gBest - x_i)$$

mientras que los dos mejores utilizan menos presión para permanecer en el lugar cambiando la actualización de su vector velocidad de la siguiente forma:

$$v_i \leftarrow 0.75 \cdot rand_1 \cdot v_i + 0.25 \cdot rand_2 \cdot (gBest - x_i)$$

Como se dijo anteriormente no se utiliza el concepto de inercia de la manera habitual ya que el efecto esperado es que la partícula pase a través del óptimo logrando que la condición invierta su valor de verdad. Los nuevos datos de entrada a considerar serán las posiciones de los individuos luego de sumarles sus correspondientes vectores velocidad.

El proceso *EjecutarPrograma* es el encargado de aplicar los datos de entrada e identificar las condiciones que han cambiado de estado, ya que con cada ejecución pueden aparecer nuevas condiciones cubiertas o evaluadas. Durante este proceso, las condiciones que han sido evaluadas, incorporan a su población los datos de entrada utilizados, reemplazando al individuo que le dio origen. A diferencia del algoritmo PSO convencional, aquellos individuos, que al desplazarse dieron lugar a nuevos datos de entrada, pero que al momento de ejecutar el programa no permitieron evaluar la condición, no serán registrados en la población, dejando al individuo que le dio origen en la misma posición.

Cada conjunto de datos de entrada, utilizado para ejecutar el programa, es registrado en una lista a fin de reducir el tiempo de cómputo. Los datos de entrada que hayan modificado el estado de alguna condición son incorporados al conjunto de datos de prueba de salida, *ListaRta*.

4. Resultados

La solución se implementó en Ruby, un lenguaje de programación interpretado, reflexivo y orientado a objetos, sumamente flexible que permite no sólo la rápida modificación de la solución, sino la implementación del asistente de ejecución que indica al generador de casos de prueba cual es el valor de las variables en cada condición.

El desempeño del método propuesto fue probado en la generación de datos de prueba para algunos programas característicos en el ámbito del testing de datos:

- Triángulos: recibe el largo de los tres lados un triángulo e indica qué tipo de triángulo es.
- Calday: recibe una fecha e indica a qué día de la semana pertenece.
- Select: recibe un arreglo con una lista desordenada y un índice k y devuelve el k-ésimo menor elemento.
- QuickSort : método de ordenación de una lista.
- Bessel : algoritmo que resuelve las funciones de Bessel J_n e Y_n.

La Tabla 2 muestra el promedio de los resultados obtenidos luego de realizar 100 ejecuciones independientes del método propuesto considerando un número de 150 iteraciones como máximo. También se incluyen los resultados de aplicar el método Tabú Search [Dia03] y una generación totalmente Random en las mismas condiciones.

Tabla 2: Resultados obtenidos con el método propuesto y su comparación con otras dos soluciones existentes.

	Tabu Search		**Random**		**PSO modificado (Método propuesto)**	
	Cobertura	**Evaluaciones**	**Cobertura**	**Evaluaciones**	**Cobertura**	**Evaluaciones**
Triángulos	73,75%	55,58	95,75%	34,76	100%	50,72
Calday	83,04%	1491,26	98,43%	197,21	99,4%	512,74
Select	98,83%	139,13	100%	16,55	100%	72,56
QuickSort	96,63%	2116,25	99,03%	320,08	100%	483,76
Bessel	100%	7,99	100%	2,1	100%	2,21

Como puede observarse, la cobertura final alcanzada por el método aquí propuesto basado en una modificación de PSO, para los programas examinados, resulta superior. Un aspecto a considerar, a partir de la cantidad de evaluaciones promedio de cada método, es la aplicación de estrategias de búsqueda para resolver problemas que requieren una escasa cantidad de iteraciones. Puede observarse que el método random permite determinar un conjunto de datos de prueba adecuado para el programa *Select* con muy pocas evaluaciones. En este caso particular, la solución es fácil de encontrar y la aplicación de una estrategia de búsqueda sólo limita la capacidad exploratoria del método; situación que no se presenta en la generación aleatoria. De todas formas, aunque la cantidad de evaluaciones es superior, la cobertura del método propuesto sigue siendo adecuada.

5. Conclusiones

Se ha presentado un nuevo método para la generación de casos de prueba, basado en una versión modificada del algoritmo de PSO, que utiliza poblaciones específicas asociadas a cada condición del programa.

Se ha implementado un sistema de evaluación y ejecución asistida para programas escritos en Ruby a través del cual se ha medido el desempeño de esta propuesta.

Los resultados obtenidos para cada uno de los programas en los diferentes métodos, indican que el método propuesto es válido, aumentando la cobertura en todos los casos y decrementando levemente la cantidad de ejecuciones totales, demostrando de esta forma que se realizó un aporte significativo en el área.

Referencias

[Ber02] Van den Bergh F. An analysis of particle swarm optimizers. Ph.D. dissertation. Department Computer Science. University Pretoria. South Africa. 2002

[Bir06] Bird S. and Li X. Adaptively Choosing Niching Parameters in a PSO. *Proceeding of Genetic and Evolutionary Computation Conference 2006 (GECCO'06)*, eds. M. Keijzer, et al., p.3 - 9, ACM Press. 2006.

[Cle02] Clerc M., Kennedy J. The particle swarm – explosion, stability and convergence in a multidimensional complex space. *IEEE Transactions on Evolutionary Computation*. Vol 6,nro. 1, pp. 58-73. Feb.2002

[Ken95] Kenedy J. and Eberhart R. Particle Swarm Optimization. *Proceedings of IEEE International Conference on Neural Networks*. Vol IV, pp.1942-1948. Australia 1995

[Nie06] Nieto José. Algoritmos basados en Cúmulos de Partículas para la resolución de problemas complejos. Universidad de Málaga. Tesis. 2006.

[Shi98] Shi Y., Eberhart R. Parameter Selection in Particle Swarm Optimization. *Proceedings of the 7th International Conference on Evolutionary Programming*. pp. 591-600. Springer Verlag 1998. ISBN 3-540-64891-7

[Shi99] Shi Y., Eberhart R. An empirical study of particle swarm optimization. Proceeding on IEEE Congress Evolutionary Computation. pp.1945-1949. Washington DC, 1999.

[Bir83] D. Bird and C. Munoz. Automatic generation of random self-checking test cases. IBM Systems Journal, 22(3):229-245, 1983.

[Mich91] 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.

[Off91] J.Offutt. An integrated automatic test data generation system. Journal of Systems Integration, 1(3):391-409, November 1991.

[Dia03] Díaz E., Tuya J., Blanco R. Automated Software Testing using a Metaheuristic Techmique based on Tabu Search. 18 th IEEE International Conference on Automated Software Engineering. pp.310- 313. ISBN: 0-7695-2035-9. 2003

[Sag06] Sagarna R., Lozano J. Scatter Search in software testing, comparison and collaboration with Estimation of Distribution Algorithms. European Journal of Operational Research 169 (2006) 392–412

[Par99] Pargas R., Harrold M., Peck R. Test-Data generation using Genetic Algorithms. Journal of Software Testing, Verification and Reliability. Vol 9. pp.263-282.1999

[TRA98] Tracey N., Clark J., Mander K. Automated program flaw finding using simulated annealing. International Symposium on Software Testing and Analysis. 1998. pp. 73-81. ACM/SIGSOFT

[Bou07] Bouchachia A. An Immune Genetic Algorithm for Software Test Data Generation. Seventh International Conference on Hybrid Intelligent Systems. 2007. pp.84-89