Generación Dinámica de Casos de Prueba Utilizando Metaheurísticas

Autor: Juan Pablo La Battaglia

Directora: Prof. Lic. Laura Lanzarini

Tesina de Licenciatura en Sistemas
Facultad de Informática
UNIVERSIDAD NACIONAL DE LA PLATA

A Ariel y Marina,
que desde el primer día
nunca dejaron de apoyarme.

A Laura,
porque gracias a ella
alcancé la meta.

A Sandra,
por regalarme cada día.

Al Maestro,
ya que sin Él,
nada es…

En el principio
no había existencia ni inexistencia;
todo este mundo era energía sin manifestarse...
El Ser Único respiraba, sin respiración,
por su propio poder.
Nada más existía...

- Himno de la Creación, Rig Veda

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 esta tesis se presenta una nueva alternativa basada en la combinación de una metaheurística poblacional (PSO) 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.

Palabras Claves: Testeo de Software, Testeo Evolutivo, Algoritmos Evolutivos, Metaheurísticas, Optimización Basada en Cúmulos de Partículas.

Trabajo aceptado para su publicación en los proceedings del *I Congreso de Inteligencia Computacional Aplicada* a realizarse en Buenos Aires en julio de 2009 (CICA 2009).

Índice General

Capítulo 1

Introducción

Dentro de las diversas problemáticas que se pueden encontrar en cada una de las etapas del desarrollo de soft en la Ingeniería de Software, la generación automática de un conjunto de datos de prueba que permita detectar los errores de un programa dado en la fase de testing, es una tarea de suma importancia que requiere un alto costo computacional y que generalmente es difícil de resolver.

Dicha tarea, inicialmente desempeñada a veces por los mismos desarrolladores, a veces por grupos específicos, comenzó a encontrar los obstáculos obvios de todo proceso informático que crece en tamaño y complejidad: el esfuerzo y tiempo humano necesarios para llevarla a cabo crece exponencialmente en relación al incremento de los módulos a testear.

La solución a 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]. Se advirtió prontamente que este método, si bien lograba cubrir gran parte del objetivo, estaba lejos de ser una solución definitiva ya que en muchos de los casos el mero azar no era suficiente para conducir el proceso de testing hacia los sectores de código más intrincados que, precisamente, son los más propensos a contener algún tipo de error.

Con el objetivo de resolver las falencias del método anterior, se propusieron soluciones que pueden clasificarse en dos grandes grupos: por un lado están aquellas basadas en un enfoque teórico del problema y por otro las que se apoyan en un punto de vista más práctico.

El primer grupo busca modelizar la situación y plantea una estrategia de generación de datos de prueba simbólica, la cual consiste en utilizar valores simbólicos para las variables en vez de valores reales dando lugar a una ejecución simbólica [Off91]. Luego, a partir de dicha ejecución se obtienen restricciones algebraicas que permiten obtener los casos de prueba. Si bien este método logra resultados correctos, tiene la dificultad de tener que resolver las ecuaciones propuestas mediante manipulaciones algebraicas muy complejas.

Por otro lado, con el método "práctico" para la generación de datos se llega al paradigma de generación dinámica de datos de prueba. En este caso, se utiliza un generador de datos de prueba que monitorea el resultado de distintas ejecuciones del programa a testear y decide, en función de un criterio establecido a priori, cual es el conjunto de prueba adecuado. Para llevar a cabo esta tarea, el programa, objeto del testeo, debe ser modificado a fin de poder brindar información relacionada con su ejecución al generador quien debe verificar si dicho criterio fue alcanzado o no. Cabe aclarar que las modificaciones introducidas en el programa objeto no alteran su normal desarrollo ni cambian de manera alguna su flujo.

Este último enfoque propone un proceso iterativo que permite construir el conjunto de datos de prueba adecuado para el programa a testear considerando como tal, al conjunto mínimo que satisfaga el criterio prefijado.

Planteada en estos términos, la construcción del conjunto de datos de prueba puede ser definida como un problema de optimización.

Se puede definir *Problema de Optimización* como el proceso de escoger una opción de entre un conjunto de ellas que sea mejor o igual que las demás. Este tipo de situaciones siempre están presentes de alguna forma sea cual fuere el ámbito en el que se observe, desde la vida cotidiana hasta la industria. A lo largo de la historia se ha dedicado mucho esfuerzo a encontrar técnicas que permitan resolver tales problemas. Dichas técnicas pueden clasificarse en dos grandes categorías: exactas o aproximadas. Las técnicas exactas realizan una búsqueda exhaustiva de todas las posibles soluciones a un problema, hasta encontrar la mejor de todas; si bien esto garantiza que se encontrará la solución óptima, se debe tener en cuenta que los problemas en general son del tipo np-completos, lo cual implica un tiempo de ejecución inaceptable. Por otro lado, con una técnica aproximada, a diferencia de las exactas, no se puede garantizar que se encontrará la solución óptima, pero sí que se encontrará una solución aceptable en un tiempo de búsqueda razonable. Esto lo logran aplicando estrategias particulares de exploración que les permiten, evaluando algunas características de las soluciones encontradas y algunos factores del entorno, "rastrear" las soluciones óptimas.

En el caso particular de los posibles valores que pueden tomar los argumentos de entrada de un programa que se desea testear, es fácil de ver que se trata de un número muy grande, lo que descarta inmediatamente la posibilidad de utilizar alguno de los métodos exactos. Por tal motivo, el objetivo general de esta tesis es la resolución de problemas de optimización utilizando técnicas aproximadas.

Si bien, sobre cada una de las técnicas de optimización aproximadas que se pueden encontrar, existe una basta documentación que provee el material necesario para su aplicación y configuración, algunas particularidades de la

generación dinámica de casos de prueba obligan a echar una segunda mirada a ciertos detalles de implementación de la estrategia que se desee aplicar. Cosas tales como funciones de adecuación (fitness) no continuas, optimizaciones paralelas independientes o búsqueda en la dirección del óptimo (más allá del óptimo en si mismo), plantean un reto muy interesante a ser solucionado, y del cual se puede obtener un gran provecho, que se trata, ni más ni menos, que de la colaboración en uno de los pasos que consumen más recursos en la Ingeniería de Software para el desarrollo de programas libres de errores evitables.

El objetivo específico de esta tesis es presentar una nueva estrategia perteneciente al grupo de técnicas aproximadas que permite generar dinámicamente el conjunto de datos de prueba adecuado para testear un programa. Su definición se basa en la combinación de un proceso de optimización utilizando cúmulos de partículas (PSO – Particle Swarm Optimization) con una lista tabú. El primero (PSO) permite analizar el efecto que cada caso de prueba produce en la ejecución del programa y decidir, a través de algunas modificaciones realizadas sobre el proceso de adaptación, cuales son los valores más adecuados para incluirlos dentro del conjunto esperado. La lista tabú se utiliza para evitar la ejecución reiterada del programa sobre los mismos datos de prueba. Esta es una de las tareas más costosas en tiempo y por tal motivo requiere de un análisis específico.

La generación de un conjunto adecuado de datos de prueba no es el único problema de optimización que puede presentarse en el área de la Ingeniería de Software. Por tal motivo, en el capítulo 2 se mencionan distintas etapas de la Ingeniería de Software haciendo hincapié en los problemas de optimización típicos que afectan a cada una de ellas, mencionando algunos de los trabajos de investigación que se ocupan de los mismos. Esto permite observar la relación estrecha que existe entre la disciplina del desarrollo del software y diversos problemas de optimización que exigen ser resueltos para lograr un resultado aceptable en tiempo y forma.

Antes de describir específicamente el método propuesto en esta tesis, se introduce a través del capítulo 3 una presentación más detallada de estas estrategias aproximadas. Allí se define el concepto de metaheurística junto con una breve reseña de las técnicas más utilizadas en la literatura. El capítulo 4 introduce con más detalle el proceso de optimización mediante cúmulos de partículas. Esto se debe a que es la base del método propuesto y es necesario conocer su funcionamiento para poder comprender las limitaciones que presenta a la hora de resolver el problema de generación de los casos de prueba.

El capítulo 5 contiene la definición del método propuesto en esta tesis. Esto incluye la descripción, definición y detalles de implementación de la generación dinámica de casos de prueba. En la sección 5.5 se detallan todas las características

que debieron modificarse de la metaheurística original para obtener un mejor provecho del proceso de optimización.

Los detalles de los experimentos llevados a cabo, así como los programas utilizados, los métodos empleados y las parametrizaciones realizadas son presentados en el capítulo 6. Por último, el capítulo 7 contiene las conclusiones y propone algunas líneas de trabajo futuras.

Capítulo 2

Problemas de optimización en Ingeniería de Software

Se puede definir a la *Ingeniería de Software* siguiendo al estándar IEEE 610.12-1990 [IEEE90] como "la aplicación de un método sistemático, disciplinado y cuantificable al desarrollo, operación y mantenimiento de software". Si bien el término de *Ingeniería de Software* fue acuñado durante una conferencia en 1968, las problemáticas que intenta abordar y resolver existen desde mucho antes.

En relación a otras ingenierías tiene una historia corta, que se relaciona inexorablemente con la aparición de las computadoras digitales en 1941. Inicialmente el software era relativamente simple y, por lo tanto, no resultaba complicado estructurar el desarrollo y asegurar la calidad del mismo.

Con el aumento de las prestaciones de las computadoras, se abría la posibilidad al desarrollo de software cada vez más complejo apoyado en todo momento por los cada vez más robustos lenguajes de programación y herramientas de más alto nivel. A lo largo de la historia, el desarrollo de software ha pasado de ser una tarea realizada por una sola persona en poco tiempo a convertirse en un conjunto de actividades interrelacionas que deben realizarse en grandes equipos de trabajo multidisciplinarios durantes meses, posiblemente en diferentes ubicaciones alrededor del mundo. Es en este contexto donde la estructuración de las etapas del desarrollo de proyectos cobra fuerza, y así fueron apareciendo los *procesos de desarrollo de software* o *ciclos de vida del software* como una herramienta para poner orden en este conjunto de actividades. Se han propuesto diversos procesos de desarrollo de software en las últimas décadas entre los que se destacan el ciclo de vida en cascada [Win70] y ciclo de vida en espiral [Bar88]. Estos procesos de desarrollo de software determinan un conjunto de actividades y un orden entre ellas.

El incremento de los procesos trajo aparejado el crecimiento de la problemática que deben enfrentar. Muchos de los obstáculos presentes en cada una de las etapas de la Ingeniería de Software están relacionados con los *Problemas de Optimización* lo que ha motivado el interés en la aplicación de estrategias adecuadas para tratar de resolverlos [Chi07].

Para poder comprender mejor la relación que esta tesis desea explotar entre la Ingeniería de Software y los Problemas de Optimización, se presenta una breve descripción de un conjunto de problemas que han sido planteados en la literatura. No se encontrará en ella una revisión completa y exhaustiva de trabajos, sino más bien una muestra en la que se aprecia la diversidad de los problemas que pueden definirse dentro la Ingeniería de Software. Por una cuestión de orden, se los presentará según la etapa del proceso en que aparecen. Las etapas que fueron consideradas son: análisis de requerimientos, diseño, implementación, testing, puesta en producción, mantenimiento y gestión. Todas ellas aparecen en los distintos procesos de desarrollo de software, ya que son fases ineludibles en la creación de software de calidad. A veces no está claro dónde colocar un problema en concreto; existen algunos problemas que se encuentran en el borde entre dos etapas y por lo tanto, podrían aparecer en ambas. Dentro de las categorías de mantenimiento y gestión se ha realizado una segunda clasificación de problemas, ya que la diversidad de éstos sugería tal división. En la figura 2.1 se muestra la clasificación propuesta.

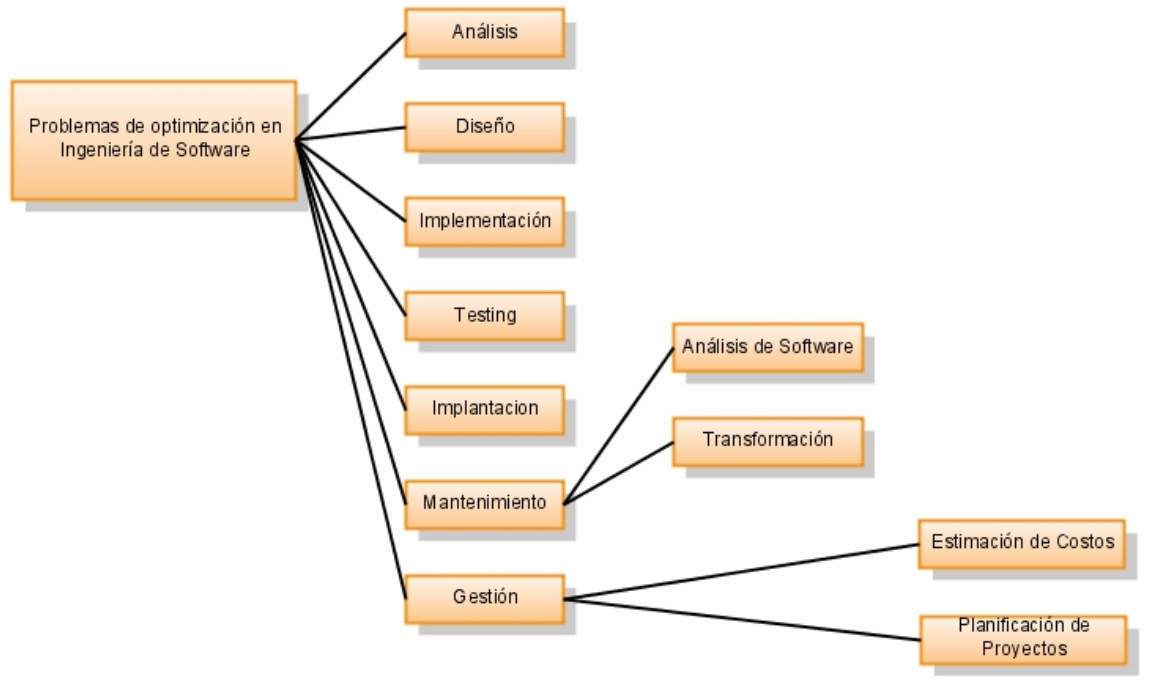

Figura 2.1: Clasificación de problemas de optimización en Ingeniería de Software.

A continuación se describen los problemas de optimización en el marco de cada una de las etapas mencionadas.

2.1. Análisis de Requerimientos

Es durante esta fase en la que se comienza a dar forma al concepto principal del sistema a desarrollar, donde el ingeniero de software y el cliente discuten sobre

lo que tiene que hacer el producto de software. Es una etapa fundamental, ya que de aquí se trazarán los lineamientos principales del desarrollo basados en la información que brinda el cliente en función de las expectativas y requisitos que tiene para el mismo. En los procesos de desarrollo de software utilizados actualmente se tiende a desarrollar y entregar el software de forma incremental. Del conjunto total de requisitos se seleccionan algunos y se realiza una iteración del proceso de desarrollo para, finalmente, entregar al usuario un producto de software que cumpla dichos requisitos. Tras esto, se vuelven a seleccionar algunos de los requisitos no cubiertos y se realiza otra iteración del proceso de desarrollo. Se sigue de esta forma hasta que el usuario esté completamente satisfecho con el producto. La decisión de qué requisitos deben cubrirse en una iteración concreta del proceso no es algo trivial. Existen numerosas restricciones y prioridades entre los requisitos que afectan al costo del producto y a la satisfacción del usuario. Por ejemplo, un requisito puede necesitar que otro requisito previo esté ya cubierto en el software, o que otro requisito se cubra a la vez que él. Es decir, existen dependencias entre requisitos. Por su parte, el usuario puede necesitar varios requisitos con urgencia mientras que otros son menos importantes para él. A esto se suma el hecho de que cada requisito tiene un costo de implementación asociado. En definitiva, la elección de requisitos a implementar en una iteración concreta del proceso de desarrollo software es una tarea no trivial que puede beneficiarse de las técnicas de optimización existentes.

Greer y Ruhe abordan el problema de la selección de requisitos para cada iteración del proceso de desarrollo software en [Des04]. Baker *et al*. [Bak06] resuelven el problema de la "siguiente versión", que consiste en seleccionar los requisitos que van a implementarse en la siguiente iteración del proceso de desarrollo.

2.2. Diseño

En la fase de diseño los ingenieros de software desarrollan una solución que cumple los requisitos recolectados en el análisis. En esta fase se decide el diseño arquitectónico y de comportamiento del software con suficiente detalle como para que pueda ser implementado, pero sin llegar a ser una implementación.

Cuando, en el desarrollo del software, se usa el paradigma de orientación a objetos, uno de los objetivos de la fase de diseño consiste en proponer el conjunto de clases que formarán la aplicación. Las clases son entidades con cierta autonomía que colaboran entre ellas. En un buen diseño, se espera que el acoplamiento (dependencia) entre clases sea bajo y la cohesión de cada clase (relación entre las tareas de los miembros de la clase) sea alta. Persiguiendo estos objetivos, Simons y Parmee [Sim06] plantean el diseño conceptual del software como un problema de optimización. Partiendo de un conjunto de métodos con parámetros y atributos que han sido identificados mediante los casos de uso, los

autores abordan el problema de encontrar una asignación de los métodos a clases que minimice el acoplamiento y maximice la cohesión.

Durante el desarrollo del software se generan distintos componentes, clases o módulos que deben ser integrados conjuntamente para formar el producto final. Estos componentes son testeados de forma aislada (prueba unitaria *o unit testing*) conforme se desarrollan y, tras integrarlos con el resto de componentes ya terminados, se prueban de nuevo para comprobar si la interacción entre ellos es correcta (prueba de integración o *integration testing*). En este último caso, es posible que un componente necesite otro que aún no está completo, siendo necesario sustituir el componente incompleto por un componente temporal que tiene la misma interfaz, conocido como *stub*. La implementación de estos *stubs* es costosa y puede evitarse o reducirse si se implementan los componentes siguiendo un orden adecuado, que dependerá fundamentalmente del grafo de interrelaciones de componentes. La determinación de este orden es un problema de optimización que ha sido abordado en [Bri02]. Se ha colocado este problema en la sección de diseño, a pesar de estar a mitad entre el diseño y la implementación, porque el objetivo del problema no es mejorar ningún aspecto de la implementación, sino más bien ahorrar costos del proyecto.

2.3. Implementación

En la fase de implementación es donde el diseño previamente creado se convierte en código fuente. Los ingenieros tienen que codificar en la plataforma seleccionada el diseño desarrollado en la fase anterior.

Los problemas de optimización que se han planteado en la fase de implementación tienen que ver con la mejora o creación de herramientas que ayuden al programador en su labor. Por ejemplo, Reformat *et al*. [Ref03] plantean el problema de la generación automática de clones. Este problema consiste en generar automáticamente un fragmento de código fuente (una función por ejemplo) que sea equivalente a otro dado. Esta técnica tiene aplicaciones en el desarrollo de software tolerante a fallos.

Los compiladores son herramientas imprescindibles en la implementación del software y, a su vez, una fuente de gran cantidad de problemas de optimización que deben abordarse de algún modo a la hora de generar el código objeto. La forma habitual de resolver estos problemas es mediante una heurística a medida que, si bien no ofrece una solución óptima en todos los casos, su tiempo de ejecución es corto, lo cual es una propiedad deseable para poder incorporarla en un compilador. Stephenson *et al*. [Ste03] plantean el problema de optimizar estas heurísticas.

En general, las optimizaciones que aplica un compilador no son conmutativas, es decir, diferentes secuencias de optimización dan lugar a distintos resultados (código objeto). Cooper *et al*. [Coo99] abordan el problema de encontrar una secuencia de optimización óptima para reducir el espacio que ocupa el código resultante, lo cual tiene una importante aplicación en el desarrollo de software para sistemas empotrados.

Otro problema resuelto de forma no óptima es la gestión de memoria dinámica. Del Rosso [Del06] aborda el problema de ajustar la configuración de la heap para reducir la fragmentación interna en listas libres segregadas mientras que Cohen *et al*. [Coh06] aplica técnicas de clustering a los threads con el objetivo de identificar grupos de threads que accedan a los mismos objetos. Estos grupos compartirán una heap, haciendo que el garbage collector no tenga que detener todos los threads para llevar a cabo su labor, sino tan sólo los del grupo afectado.

Otro interesante problema planteado en el área de los compiladores es la generación automática de código paralelo óptimo a partir de código secuencial. Este problema ha sido abordado por Nisbet [Nis98], Williams [Wil98] y Ryan [Rya00].

Dada una unidad de código fuente (función, clase, etc.) es posible aplicar una secuencia de transformaciones a dicho código que preserva la semántica pero que altera la sintaxis. Esta secuencia de transformaciones podría reducir el número de líneas de código. El problema de encontrar la secuencia de transformaciones que minimice la longitud del código resultante ha sido abordado en [Fat03] y [Fat04].

2.4. Testing

Es en la etapa de testing donde el software desarrollado debe ser testeado para descubrir posibles errores y para verificar que lo que se implementó cumpla cabalmente con lo solicitado por el cliente en las especificaciones.

La resolución de esta etapa como un problema de optimización es el tema central de esta tesis. Si bien, como se menciona más adelante, es este uno de los problemas más estudiados en este ámbito, también es cierto que debido a su carácter crítico (los errores pasados por alto llegarán al cliente), su naturaleza tediosa para la realización manual y su gran consumo de recursos, es uno de lo que más atención requiere. Su evolución va de la mano del avance de las distintas técnicas de optimización. Por otro lado la flexibilidad de los nuevos lenguajes de computación facilita en gran medida la posibilidad de evaluar en forma externa condiciones y valores de un programa en ejecución, tarea fundamental para dirigir los procesos de optimización durante el testing.

En lo referente al proceso de testing, dependiendo de la envergadura del mismo, la evaluación puede ser de todo el sistema o subdividida por proceso o función donde se realizarán inicialmente las pruebas de unidad y luego se harán pruebas de integración, donde los test van abarcando cada vez más módulos hasta llegar a una prueba total del desarrollo.

Para testear un método o función se deben determinar los valores de los parámetros de entrada y una vez escogidos se corre el código en busca de errores y verificando que la salida sea la esperada. Un caso de prueba es una combinación concreta de valores para dichos parámetros de entrada asociados a una salida esperada. La labor del ingeniero de pruebas es crear un conjunto de casos de prueba y testar el software con él. Este conjunto de casos de prueba debe ser tal que maximice la probabilidad de descubrir los posibles errores del objeto de la prueba.

El proceso de generación dinámica de casos de prueba no puede asociar resultados esperados a los casos producidos, ya que no puede conocer el programa de ante mano, pero en cambio intenta conseguir que los casos generados recorran el programa en su totalidad. El enfoque más popular para conseguir conjuntos de casos de prueba adecuados es usar criterios de cobertura: un conjunto de casos de prueba se considera adecuado si consigue cubrir una gran cantidad de elementos (condiciones, instrucciones, ramas, predicados atómicos, etc.) del objeto de prueba.

El problema de la generación automática de casos de prueba es sin duda el más estudiado de los problemas de optimización en la Ingeniería de Software. Desde el primer trabajo de Miller y Spooner de 1976 [Mil76] hasta nuestros días, se han propuesto diversas formas de abordar esta tarea. Uno de los paradigmas empleados, es el estructural, hace uso de información estructural del programa para generar los casos de prueba [Alb05][Alb06]. Esta información proviene generalmente del grafo de control de flujo. En numerosos trabajos se han estudiado y propuesto diferentes funciones objetivo [Bar02][Bot03]. Así mismo, para aumentar la eficiencia y eficacia del proceso de generación de casos de prueba, se han combinado las estrategias de búsqueda con otras técnicas tales como *slicing* [Har02], transformación de código [Hie05], análisis de dependencia de datos [Kor03], uso de medidas software [Lam04] y búsqueda en dominios variables [Sag03]. Se han propuesto soluciones para algunos de los problemas principales que surgen en la generación de casos de prueba: la presencia de *flags* [Bar04][Bar03], la existencia de estados internos en el programa [Mcm03][Mcm05], las particularidades del software orientado a objetos [Wap05][Wap06] y la presencia de excepciones [Tra00]. Entre las técnicas más populares para la resolución del problema se encuentran los algoritmos genéticos [Gir05][Xan92], la búsqueda tabú [Dia05], la búsqueda dispersa [Sag06] y los algoritmos de estimación de distribuciones [Sag05].

Otro tema importante en la etapa de testing lo constituyen las pruebas de interacción las cuales consisten en comprobar todas las combinaciones posibles de entornos de ejecución del software: distintos sistemas operativos, distintas configuraciones de memoria, etc. Cuantos más factores se consideren mayor es el número posible de combinaciones, creciendo éste de forma exponencial. Por este motivo, se han propuesto estrategias basadas en optimización para reducir este enorme conjunto de combinaciones [Bry05][Coh03].

Por otro lado, existe un conjunto de trabajos en el que se comprueban aspectos no funcionales del software. La mayoría de ellos se centran en el tiempo de ejecución, con aplicaciones a los sistemas en tiempo real [Bri05]. Algunos trabajos extienden las técnicas para abordar software orientado a objetos y basado en componentes [Gro03]. También se han definido medidas que permiten conocer, tras un examen del código fuente del programa, si la aplicación de técnicas basadas en computación evolutiva es adecuada o no para comprobar las restricciones temporales [Gro01].

2.5. Implantación

Tras desarrollar y probar el software, procede la fase de implantación en la que el sistema software es instalado y desplegado en su entorno definitivo, donde será utilizado.

Monnier *et al*. [Mon98] abordan el problema de la planificación de tareas en un sistema distribuido de tiempo real como un problema de optimización. Dado un conjunto de tareas cuyo tiempo de ejecución está acotado y con dependencias entre ellas, el problema consiste en asignar a cada tarea una máquina para su ejecución y planificar los instantes de tiempo en los que se comunicarán las tareas usando la red.

2.6. Mantenimiento

La fase de mantenimiento es donde el software se modifica como consecuencia de las sugerencias de los usuarios o, simplemente, con el objetivo de mejorar su calidad. Al igual que ocurría en la fase de pruebas, existen muchos trabajos que afrontan problemas surgidos en la fase de mantenimiento usando técnicas de optimización. Para realizar una clasificación más fina de todos estos trabajos, son divididos en dos grupos: análisis de software y transformación.

2.6.1. Análisis de Software

Una de las técnicas usadas durante la fase de mantenimiento para llegar a una rápida comprensión de la estructura y los objetivos de un determinado fragmento de código fuente es la vinculación de conceptos (*concept binding*). Esta técnica consiste en asignar automáticamente a cada segmento del código fuente una palabra o concepto perteneciente a un mapa conceptual previamente definido. Durante una primera etapa, se asigna a cada línea o instrucción del código fuente un *indicador*. Estos indicadores señalan la posible vinculación de un determinado concepto al segmento de código en el que se encuentra el indicador. La siguiente etapa consiste en asignar conceptos a segmentos de código en función de los indicadores que se encuentren. Esta segunda etapa ha sido planteada como problema de optimización por Gold *et al*. [Gol06]. Otras técnicas utilizadas para extraer automáticamente una estructura de alto nivel del software a partir del código fuente son las técnicas de *software clustering*. La idea detrás de ellas es agrupar distintos elementos del software (clases, funciones, etc.) en módulos o subsistemas de acuerdo a su cohesión y acoplamiento. Este problema, originalmente propuesto por Mancoridis *et al*. en [Man98], ha sido investigado en profundidad en numerosos trabajos [Dov99][Hie02][Swi05][Mah03]. Mitchell *et at*. [Mit02] proponen un procedimiento completo de ingeniería inversa compuesto por una primera etapa en la que se aplican técnicas de s*oftware clustering* seguida de una segunda etapa de reconocimiento de estilos de interconexión entre los subsistemas inferidos en la primera etapa. La detección de clones consiste en descubrir fragmentos del código fuente que son idénticos o similares sintáctica o semánticamente. Sutton *et al*. [Sut05] resuelven el problema de detección de grupos de clones en el código fuente.

2.6.2. Transformación

Basados en la idea del software clustering, Seng *et al*. [Sen05] plantean el problema de mejorar la descomposición del software en subsistemas. Se trata de encontrar una partición de los elementos del software (clases o funciones) formando subsistemas de manera que se optimicen una serie de parámetros como la cohesión, el acoplamiento o la complejidad.

Slicing es una técnica usada para extraer automáticamente un fragmento no contiguo del código que resulta de valor para una acción concreta. Una forma de realizar esta extracción es seleccionar un conjunto de variables y obtener una proyección del software (un subconjunto de instrucciones) que posea la misma semántica que el original con respecto a esas variables. Una variante de esta técnica, denominada *slicing* amorfo (*amorphous slicing*), consiste en generar un fragmento de código que preserve la semántica del programa con respecto a las variables seleccionadas pero sin necesidad de preservar la sintaxis. En [Fat05], Fatiregun *et al*., abordan el *slicing* amorfo como un problema de optimización.

Refactoring es una técnica para reestructurar el código fuente, cambiando su estructura interna, sin modificar la semántica. La base de esta técnica se encuentra en una serie de pequeñas transformaciones del código que preservan el comportamiento del software. El objetivo del refactoring es encontrar una secuencia de transformaciones que den lugar a una estructura más clara, legible y fácil de mantener. Este problema se ha resuelto utilizando técnicas de optimización en numerosas ocasiones [Sal06][Har07][Oke06].

2.7. Gestión

La gestión de un proyecto software es una tarea que debe llevarse a cabo durante todo el proceso de desarrollo, controlándolo en todo momento y realizando los ajustes pertinentes. Para ello, es necesario medir diversos aspectos del producto y del proceso. Asimismo, entre las labores de gestión se encuentra la asignación de personal a tareas. De nuevo aquí se dividen los trabajos en dos categorías: estimación de costos y planificación de proyectos.

2.7.1. Estimación de Costos

La predicción del costo de un proyecto software a partir de un conjunto de datos básicos del mismo es una muy preciada técnica que ha sido resuelta mediante algoritmos de optimización. Sheta en [She06] plantea el problema de ajustar los parámetros de un modelo de predicción ya existente: COCOMO.

2.7.2. Planificación de proyectos

Aguilar Ruiz *et al.* [Agu01] han estudiado el problema de extraer reglas a partir de una simulación de un proyecto de software. Estas reglas ayudan al gestor del proyecto a realizar los ajustes pertinentes para conseguir reducir el costo y la duración del proyecto.

Chang *et al*. [Cha01] abordan el problema de asignar empleados a tareas teniendo en cuenta sus habilidades, su salario y su dedicación con el objetivo de minimizar el costo y la duración del proyecto. Alba y Chicano [Chi05] abordan este mismo problema analizando distintos proyectos automáticamente generados con un generador de instancias.

La asignación de *work packages* (WP) de un proyecto a equipos de programadores con el objetivo de reducir la duración del proyecto es un problema de optimización que ha sido abordado por Antoniol *et al*. [Ant04]. Los mismos autores amplían el problema en [Ant04-1] para optimizar también la asignación de programadores a equipos y considerar soluciones robustas que sean capaces de tolerar el posible abandono de WPs, errores, revisiones e incertidumbre en las estimaciones.

Como conclusión, luego de recorrer este resumen acerca de los problemas de optimización que surgen en la Ingeniería de Software, se puede observar, como se decía al principio del capítulo, que es un tema sobre el que cada vez se realizan más trabajos de investigación.

Si bien existen trabajos particulares sobre cada fase, sin lugar a dudas la más investigada es la de testing; esto da noticia de cuál es la etapa de la Ingeniería de Software en la que más preocupa su optimización. Es en esa dirección en la que está dirigida esta tesis. Como dice Glenford Myers en [Gle79], aproximadamente la mitad del tiempo de un proyecto de software y más de la mitad de su costo se dedica a la fase de prueba, lo cual explica el esfuerzo puesto y la cantidad de material existente en la investigación de la optimización de esta etapa.

Capítulo 3

Técnicas Metaheurísticas

Este capítulo presenta una breve reseña sobre las distintas técnicas de optimización, una posible clasificación y las particularidades de algunas de sus variantes con el objetivo de dar un marco de referencia general para el método específico propuesto en esta tesis.

En el capítulo 1 se introdujo una de las principales clasificaciones de las técnicas de optimización que las divide en: exactas y aproximadas. El problema de la generación dinámica de los casos de prueba se ubica dentro del contexto de las técnicas aproximadas. Esto implica que la solución alcanzada será *aceptable* dentro de los parámetros establecidos de antemano en un tiempo *razonable*.

Dentro de las técnicas aproximadas se pueden analizar tres tipos: las constructivas, las de búsqueda local y las Metaheurísticas.

Para la utilización de los métodos constructivos se debe tener un conocimiento muy claro del dominio del problema al que se quiere aplicar. Consiste en construir literalmente paso a paso una solución del problema. Usualmente son métodos deterministas y suelen estar basados en la mejor elección en cada iteración. Estos métodos han sido muy utilizados en problemas clásicos como el del viajante. Las soluciones propuestas, aunque en muchos casos son fáciles de encontrar, generalmente son de muy baja calidad.

En el caso de los métodos de búsqueda local, a diferencia del método anterior, comienzan con una solución completa del problema y la mejoran progresivamente. El procedimiento realiza en casa paso un movimiento de una solución a otra con mejor valor. El método finaliza cuando, para una solución, no existe ninguna solución accesible que la mejore.

El tercer tipo de técnicas aproximadas lo constituyen las técnicas Metaheurísticas, término utilizado por primera vez por Glover en [Glo86], las cuales surgen de la idea de la combinación de diversas heurísticas (técnicas que buscan soluciones casi óptimas a un costo computacional razonable) a un nivel más alto para conseguir una exploración del espacio de búsqueda más eficiente y efectiva.

La Figura 3.1 resume la clasificación hasta aquí descripta.

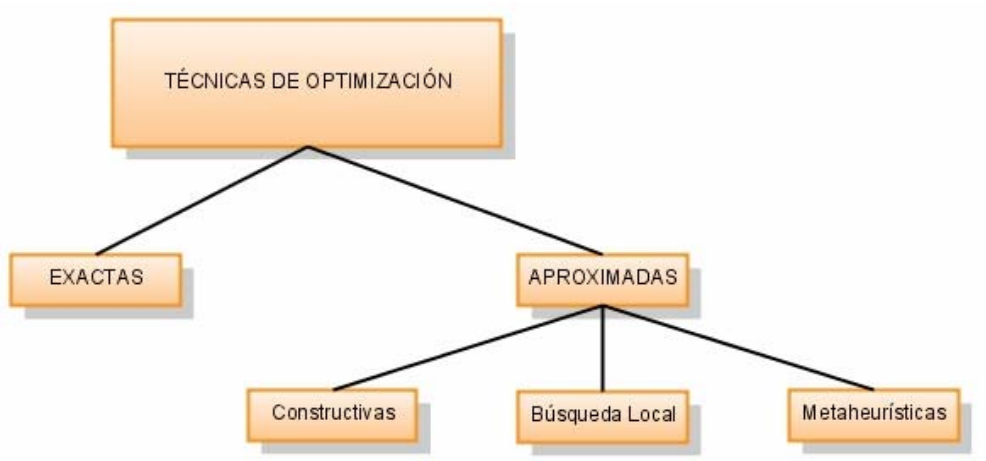

Figura 3.1: Clasificación de las técnicas de optimización.

Las características principales que describen en forma efectiva las técnicas metaheurísticas se pueden extraer de [Alb05-1][Blu03][Glo02]:

- Estrategias de alto nivel o plantillas generales que "guían" el proceso de búsqueda y tienen por objetivo encontrar soluciones (casi) óptimas.
- Algoritmos aproximados y no determinísticos que pueden incorporar mecanismos para evitar óptimos locales.
- Soluciones no específicas al problema que se intenta resolver, incorporando el conocimiento específico del problema o la experiencia (memoria) "desviando" la búsqueda. Utilizan funciones de bondad (funciones de fitness) para cuantificar el grado de adecuación de una determinada solución.
- Soluciones que poseen un amplio espectro de aplicación, desde técnicas sencillas (como búsqueda local) a técnicas complejas (procesos de aprendizaje, por ejemplo).

Se puede decir, resumiendo los puntos expuestos, que una metaheurística es una estrategia de alto nivel que utiliza diferentes métodos para explorar el espacio de búsqueda. Expresado como una solución para aplicar a un problema en particular, una metaheurística es una estructura genérica que debe ser rellenada con datos específicos del problema adaptados para la misma: representación de las soluciones, operadores para manipularlas, función de adecuación para el proceso de optimización, etc. En relación al tipo de problemas a los que se pueden (y

conviene) aplicar, son aquellos para los que no se conoce una resolución exacta, o que en caso de que se conozca, que la misma tenga para la magnitud de variables involucradas, un tiempo computacional inabarcable. Otro punto a tener en cuenta es que el problema debe permitir que los elementos de su espacio de búsqueda puedan adoptar una forma "mapeable" al formato de datos que maneja la metaheurística aplicada.

Dependiendo de las características que se tengan en cuenta, pueden armarse diversas clasificaciones para las técnicas metaheurísticas: si están basadas en algún proceso natural, si utilizan memoria en su desarrollo, si la función de adecuación se mantiene en el tiempo, etc. En este trabajo en particular se detallará una clasificación que se basa en que si en cada paso del proceso se maneja un único punto del espacio de búsqueda o se trabaja sobre un conjunto de ellos. La división entonces, se hace entre metaheurísticas basadas en trayectoria y metaheurísticas basadas en población. En la figura 3.2 se ilustra la clasificación mencionada, con las metaheurísticas más representativa de cada clase.

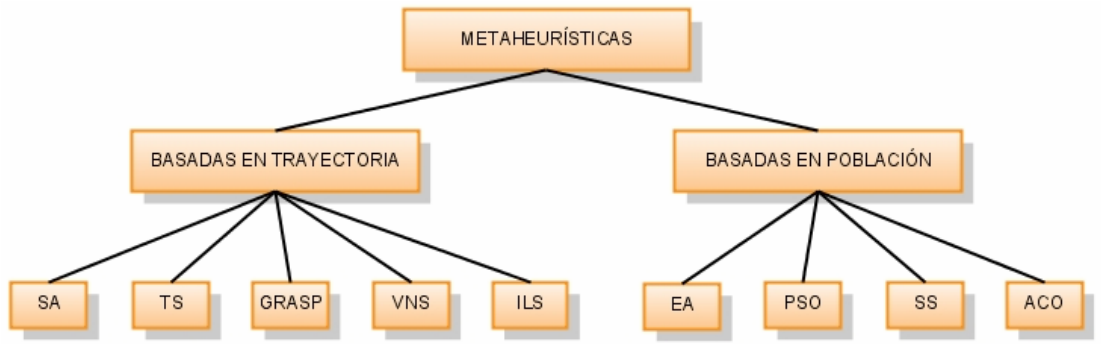

Figura 3.2: Clasificación de Metaheurísticas.

3.1. Metaheurísticas basadas en la Trayectoria

La principal característica de estos métodos es que parten de un punto y mediante la exploración de un vecindario van actualizando la solución actual, formando una trayectoria. La mayoría de estos algoritmos surgen como extensiones de los métodos de búsqueda local simples a los que se les añade alguna característica para escapar de los mínimos locales y se modifica su condición de fin en busca de una solución lo suficientemente aceptable o un número máximo de iteraciones.

3.1.1. Enfriamiento Simulado

El *Enfriamiento Simulado* o *Simulated Annealing* (SA) fue inicialmente presentado en [Kir83] y es una de las metaheurísticas más antiguas que incorpora una estrategia explícita para impedir óptimos locales. La idea del SA es simular el proceso físico del calentamiento de metales y del cristal. Se propone una similitud entre una buena estructura cristalina de metales y una buena estructura de soluciones para problemas de optimización combinatoria. Este algoritmo tiene por objetivo minimizar una función que representa la energía del sistema. Para evitar quedar atrapado en un óptimo local, el algoritmo permite elegir una solución peor que la solución actual.

En cada iteración se elige, a partir de la solución actual s, una solución s_0 del vecindario $N(s)$. Si s_0 es mejor que s (es decir, tiene un mejor valor en la función de fitness), se sustituye por s_0 como solución actual. Si la solución s_0 es peor, entonces es aceptada con una determinada probabilidad que depende de la temperatura actual T y la variación en la función de fitness, $f(s_0) - f(s)$ (caso de minimización).

Al comienzo la temperatura es alta y cualquier transición entre estados es permitida. Soluciones que empeoren la función objetivo pueden ser aceptadas con mayor probabilidad que más tarde, cuando la temperatura haya disminuido. La temperatura del sistema es controlada de acuerdo al enfriamiento sucesivo (función logarítmica) y recalentamientos periódicos, que permiten escapar de posibles óptimos locales.

3.1.2. Búsqueda Tabú

La *Búsqueda Tabú* o *Tabu Search* (TS) es una de las metaheurísticas que se ha aplicado con más éxito a la hora de resolver problemas de optimización combinatoria. Los fundamentos de este método fueron introducidos en [Glo86]. La idea básica de la búsqueda tabú es el uso explícito de un historial de la búsqueda (una memoria de corto plazo) que le permite escapar de óptimos locales y evitar exploraciones excesivas sobre una misma región. Esta memoria de corto plazo es implementada en esta técnica como una lista tabú (normalmente con una metodología FIFO), donde se mantienen las soluciones visitadas más recientemente para excluirlas de los próximos movimientos. En cada iteración se elige la mejor solución entre las permitidas y la solución es añadida a la lista tabú.

Este proceso es propenso a la pérdida de información ya que buenas soluciones pueden ser excluidas del conjunto permitido. Para reducir este problema, se puede definir un criterio de aspiración que permita a una solución estar dentro del conjunto de soluciones permitidas aún cuando figure en la lista tabú.

La solución que esta tesis plantea a la generación automática de casos de prueba utiliza una estrategia de optimización basada en cúmulos de partículas combinada con una lista tabú. Si bien más adelante se detalla el método propuesto, se puede mencionar en este punto que se utiliza la lista tabú para evitar ejecutar más de una vez el programa testeado con un conjunto de prueba particular, siendo este sin duda, uno de los recursos más costosos en el proceso tratado en este trabajo.

3.1.3. Búsqueda Miope Aleatorizado y Adaptativo

El procedimiento de Búsqueda Miope Aleatorizado y Adaptativo o *The Greedy Randomized Adaptive Search Procedure* (GRASP) [Feo99] es una metaheurística simple que combina heurísticos constructivos y de búsqueda local. GRASP es un procedimiento iterativo compuesto de dos fases: primero una construcción de una solución y después un proceso de mejora. La solución mejorada es el resultado del proceso de búsqueda.

3.1.4. Búsqueda en Vecindario Variable

La *Búsqueda en Vecindario Variable* o *Variable Neighborhood Search* (VNS) es una metaheurística propuesta en [Mla97], que aplica explícitamente una estrategia para cambiar entre diferentes estructuras de vecindario de entre un conjunto definido al inicio del algoritmo. Las vecindades pueden ser elegidas arbitrariamente, aunque usualmente se utilizan vecindades de cardinalidad creciente. Este algoritmo es muy general y con muchos grados de libertad a la hora de diseñar variaciones e instanciaciones particulares.

3.1.5. Búsqueda Local Iterada

En la *Búsqueda Local Iterada* o *Iterated Local Search* (ILS) [Stu99] la solución actual es perturbada y a esta nueva solución se le aplica un método de búsqueda local para mejorarla. Este nuevo óptimo local obtenido por el método de mejora puede ser aceptado como nueva solución actual si pasa un test de aceptación.

3.2. Metaheurísticas basadas en Población

Los métodos basados en población se caracterizan por trabajar con un conjunto de soluciones (población) en cada iteración, a diferencia de los métodos que se vieron antes que únicamente utilizan un punto del espacio de búsqueda por

iteración. El resultado final proporcionado por este tipo de algoritmos depende fuertemente de la forma en que manipula la población.

3.2.1. Algoritmos Evolutivos

Los *Algoritmos Evolutivos* o *Evolutionary Algorithms* (EA) [Bac97] están inspirados en la teoría de la evolución de las especies de *Darwin*, donde el conjunto de individuos evoluciona en un proceso que combina el entrecruzamiento y la mutación para adaptarse a los cambios en su entorno. Esta familia de técnicas siguen un proceso iterativo y estocástico que opera sobre una población de individuos que evoluciona utilizando mecanismos de selección y construcción de nuevas soluciones candidatas mediante recombinación de características de las soluciones seleccionadas. El algoritmo se inicia con una población inicial, normalmente generada al azar. Cada individuo en la población tiene asignado un valor de aptitud, normalmente conocido como fitness, que representa que tan bien el individuo se "adapta" al entorno, que sería el problema que se está tratando. Los mejores individuos tendrán los fitness más altos y en ese valor se basa el proceso de optimización. En los métodos que siguen el esquema de los algoritmos evolutivos, la modificación de la población se lleva a cabo mediante tres operadores: selección, recombinación y mutación. Estos algoritmos establecen un equilibrio entre la explotación de buenas soluciones (fase de selección) y la exploración de nuevas zonas del espacio de búsqueda (fase de reproducción).

3.2.2. Búsqueda Dispersa

La *Búsqueda Dispersa* o *Scatter Search* (SS) [Glo02] es un algoritmo que se basa en mantener un conjunto relativamente pequeño de soluciones tentativas (llamado “conjunto de referencia”) que se caracteriza tanto por contener buenas soluciones como soluciones diversas. Este conjunto se divide en subconjuntos de soluciones a las cuales se les aplica una operación de recombinación y mejora. Para realizar la mejora o refinamiento de soluciones se suelen utilizar mecanismos de búsqueda local.

3.2.3. Colonias de Hormigas

Los sistemas basados en *Colonias de Hormigas* o *Ant Colony Optimization* (ACO) [Dor92] son metaheurísticas inspiradas en el comportamiento de las hormigas reales cuando realizan la búsqueda de comida. Este comportamiento es el siguiente: inicialmente, las hormigas exploran el área cercana a su nido de forma aleatoria. Tan pronto como una hormiga encuentra la comida, la lleva al nido. Mientras que realiza este camino, la hormiga va depositando una sustancia química denominada feromona. Esta sustancia ayudará al resto de las hormigas a

encontrar la comida. Esta comunicación indirecta entre las hormigas mediante el rastro de feromona las capacita para encontrar el camino más corto entre el nido y la comida. Esta funcionalidad es la que intenta simular este método para resolver problemas de optimización. En esta técnica, el rastro de feromona es simulado mediante un modelo probabilístico.

3.2.4. Cúmulos de Partículas

Los *Algoritmos Basados en Cúmulos de Partículas* o *Particle Swarm Optimization* (PSO) [Ken01] son técnicas metaheurísticas inspiradas en el comportamiento social del vuelo de las bandadas de aves o el movimiento de los bancos de peces. Se fundamenta en los factores que influyen en la toma de decisiones de un agente que forma parte de un conjunto de agentes similares. La toma de decisiones por parte de cada agente se realiza conforme a una componente social y una componente individual, mediante las que se determina el movimiento (dirección) de este agente para alcanzar una nueva posición en el espacio de soluciones. Simulando este modelo de comportamiento se obtiene un método para resolver problemas de optimización.

Se acaba de hacer una reseña sobre la definición, estructura y clasificación de las técnicas de optimización de problemas, en particular con la aplicación de metaheurísticas. Este trabajo continúa con la descripción en detalle de los algoritmos basados en Cúmulos de Partículas que son los que, modificados a tal efecto, son utilizados para resolver el problema de la generación dinámica de los casos de prueba.

Capítulo 4

Algoritmos basados en Cúmulos de Partículas (PSO)

Así como la teoría de la evolución de la especies fue la inspiración para la creación de los algoritmos evolutivos, los *Algoritmos Basados en Cúmulos de Partículas* (*PSO* o *Particle Swarm Optmization*) definen su funcionamiento a partir de enfoque conocido como "metáfora social", ejemplos del cual pueden encontrarse en el comportamiento social del vuelo de las bandadas de aves o el movimiento de los bancos de peces.

PSO fue originalmente desarrollado por el psicólogo Jammes Kennedy y por el ingeniero electrónico Russel Eberhart en 1995 [Ken95] y el algoritmo puede resumirse de la siguiente forma: los individuos que conviven en una sociedad tienen una opinión que es parte de un "conjunto de creencias" (espacio de búsqueda) compartido por todos los posibles individuos. Cada individuo puede modificar su propia opinión basándose en tres factores:

- Su conocimiento sobre el entorno (su valor de aptitud o fitness).
- Su conocimiento histórico o experiencias anteriores (su memoria o conocimiento cognitivo).
- El conocimiento histórico o experiencias anteriores de los individuos situados en su vecindario (su conocimiento social).

Debido a la comunicación que se da entre los individuos de la población, de a poco el esquema de creencias de cada uno va convergiendo hacia el de los individuos más exitosos. Llega un punto en el que gran parte de los individuos tienen un conjunto de creencias estrechamente relacionado. De esta manera y en forma de etapas sucesivas, en una primera instancia los individuos logran una exploración extensiva del campo de búsqueda, y en un segundo paso, cuando los mejores individuos saltan a la luz, los demás se acercan lentamente aumentando la explotación de la búsqueda del óptimo alrededor de los individuos exitosos.

El principio natural en el que se basa PSO es el comportamiento de una bandada de aves o de un banco de peces: supóngase que una bandada busca alimento en un área determinada, y que solamente existe una única pieza de comida en dicha área. Los pájaros no saben donde está la comida, y a cada uno por separado le tomaría un tiempo excesivo explorar la totalidad del área, por lo que la estrategia más eficaz para hallar la comida es seguir al ave que se encuentra más cerca de ella. PSO emula este escenario para resolver problemas de optimización.

Cada solución o partícula explora el espacio de búsqueda, está siempre en continuo movimiento y nunca muere.

El cúmulo de partículas es un sistema multiagente, es decir, las partículas son agentes simples que se mueven en el espacio de búsqueda y que guardan (y posiblemente comunican) la mejor solución que han encontrado. Cada partícula tiene un fitness, una posición y un vector velocidad que dirige su movimiento. El movimiento de las partículas por el espacio está guiado por las partículas óptimas en el momento actual.

Figura 4.1: Ejemplos de *swarm* en la naturaleza.

En las siguientes secciones se realiza una descripción formal del algoritmo PSO, junto a sus principales factores, parámetros, topologías de vecindario y aspectos avanzados de su desarrollo.

4.1. Descripción del Algoritmo

Un algoritmo PSO consiste en un proceso iterativo y estocástico que opera sobre un cúmulo de partículas. La posición de cada partícula representa una solución potencial al problema que se está resolviendo.

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 $pBest_i$)

El cúmulo se inicializa generando las posiciones y las velocidades iniciales de las partículas. Las posiciones se pueden generar aleatoriamente en el espacio de búsqueda (quizás con ayuda de un heurístico de construcción), de forma regular o con una combinación de ambas formas. Una vez generadas las posiciones, se calcula el fitness de cada una y se actualizan los valores de $fitness\_x_i$ y $fitness\_pBest_i$.

Una vez que la población está inicializada, las partículas comienzan a moverse por el espacio de búsqueda en el proceso iterativo. Un individuo se mueve desde una posición a otra en cada iteración simplemente añadiendo a su vector de posición x_i el vector velocidad v_i para obtener un nuevo vector de posición:

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

Con la nueva posición de la partícula, se reevalúa su fitness y se actualiza $fitness\_x_i$. En caso de que el nuevo fitness sea el mejor encontrado hasta el momento, se actualizan los valores de mejor posición $pBest_i$, y fitness $fitness\_pBest_i$. En cada iteración se actualiza el vector velocidad de la partícula utilizando la velocidad anterior, un componente *cognitivo* y un componente *social*. La expresión resultante sería entonces:

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

donde w representa el factor de inercia [Shi98], φ_1 y φ_2 son constantes de aceleración que representan la importancia que se le dará al conocimiento cognitivo y social respectivamente, *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].

En la figura 4.2 se muestra una representación gráfica del movimiento de una partícula en el espacio de soluciones.

En esta gráfica, las flechas de línea clara representan la dirección de los vectores de velocidad actual: v^k_{pBest} es la velocidad de la mejor posición tomada por la partícula, v^k_g es la velocidad de la mejor partícula encontrada en el vecindario y v^k es la velocidad actual de la partícula. La flecha de línea oscura representa la dirección que toma la partícula para moverse desde la posición x^k hasta la posición x^{k+1}. El cambio de dirección de esta flecha depende de la influencia de las demás direcciones que intervienen en el movimiento.

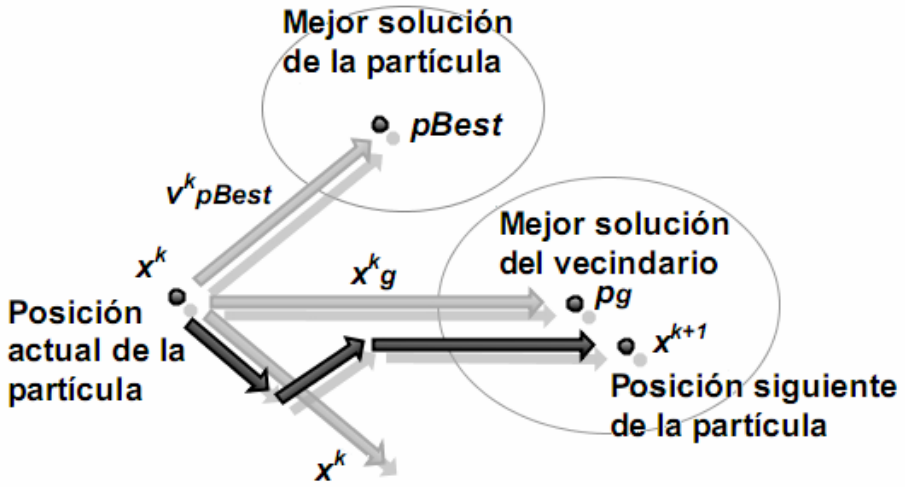

Figura 4.2: Movimiento de una partícula en el espacio de soluciones.

4.2. Tipos de Algoritmos basados en PSO

Sobre la base del algoritmo original de PSO se pueden aplicar variaciones en diversos factores de configuración para dar lugar a diferentes tipos de algoritmos.

Si las variaciones son realizadas, por ejemplo, en la influencia de los factores *cognitivo* y *social* (valores φ_1 y φ_2 respectivamente) sobre la dirección de la velocidad que toma una partícula en el movimiento, se pueden distinguir cuatro tipos de algoritmos:

- Modelo Completo: φ_1, $\varphi_2 > 0$. Tanto el componente cognitivo como el social intervienen en el movimiento.
- Modelo sólo Cognitivo: $\varphi_1 > 0$ y $\varphi_2 = 0$. Unicamente el componente cognitivo interviene en el movimiento.
- Modelo sólo Social: $\varphi_1 = 0$ y $\varphi_2 > 0$. Unicamente el componente social interviene en el movimiento.
- Modelo sólo Social exclusivo: $\varphi_1 = 0$, $\varphi_2 > 0$ y $g_i \neq x_i$. La posición de la partícula en sí no puede ser la mejor de su entorno.

Por otra parte, desde el punto de vista del vecindario, es decir, la cantidad y posición de las partículas que intervienen en el cálculo de la distancia en la componente social, se clasifican dos tipos de algoritmos: PSO Local y PSO Global.

En el PSO Local, se calcula la distancia entre la posición actual de partícula y la posición de la mejor partícula encontrada en el entorno local de la primera. El

entorno local consiste en las partículas inmediatamente cercanas en la topología del cúmulo. En La figura 4.3 se muestra el pseudocódigo de la versión Local de PSO también denominada *pBest PSO*.

Para el PSO Global, la distancia en el componente social viene dada por la diferencia entre la posición de la partícula actual y la posición de la mejor partícula encontrada en el cúmulo completo $gBest_i$, como se muestra en la figura 4.4.

Algoritmo 1 PSO Local

$S \leftarrow InicializarCumulo()$
while no se alcance la condición de parada **do**
  for $i = 1$ to $size(S)$ **do**
    evaluar cada partícula x_i del cumulo S
    if $fitness(x_i)$ es mejor que $fitness(pBest_i)$ **then**
      $pBest_i \leftarrow x_i$; $fitness(pBest_i) \leftarrow fitness(x_i)$
    end if
  end for
  for $i = 1$ to $size(S)$ **do**
    Escoger $lBest_i$, la partícula con mejor fitness del entorno de x_i
    $v_i \leftarrow \omega \cdot v_i + \varphi_1 \cdot rand_1 \cdot (pBest_i - x_i) + \varphi_2 \cdot rand_2 \cdot (lBest_i - x_i)$
    $x_i \leftarrow x_i + v_i$
  end for
end while
Salida: la mejor solución encontrada

Figura 4.3: Algoritmo PSO Local o *pBest PSO*

La versión Global converge más rápido pues la visibilidad de cada partícula es mejor y se acercan más a la mejor del cúmulo favoreciendo la intensificación, por esta razón, también cae más fácilmente en óptimos locales. El comportamiento de la versión Local es el contrario, es decir, le cuesta más converger favoreciendo en este caso la diversificación, pero no cae fácilmente en óptimos locales.

Algoritmo 2 PSO Global

$S \leftarrow InicializarCumulo()$
while no se alcance la condición de parada **do**
  for $i = 1$ to $size(S)$ **do**
    evaluar cada partícula x_i del cúmulo S
    if $fitness(x_i)$ es mejor que $fitness(pBest_i)$ **then**
      $pBest_i \leftarrow x_i$; $fitness(pBest_i) \leftarrow fitness(x_i)$
    end if
    if $fitness(pBest_i)$ es mejor que $fitness(gBest)$ **then**
      $gBest \leftarrow pBest_i$; $fitness(gBest) \leftarrow fitness(pBest_i)$
    end if
  end for
  for $i = 1$ to $size(S)$ **do**
    $v_i \leftarrow \omega \cdot v_i + \varphi_1 \cdot rand_1 \cdot (pBest_i - x_i) + \varphi_2 \cdot rand_2 \cdot (gBest - x_i)$
    $x_i \leftarrow x_i + v_i$
  end for
end while
Salida: la mejor solución encontrada

Figura 4.4: Algoritmo PSO Global o *gBest PSO*

4.3. Topologías del Cúmulo de Partículas

La manera en que una partícula interacciona con las demás de su vecindario es un aspecto muy importante que debe ser tenido en cuenta. El desarrollo de una partícula depende tanto de la topología del cúmulo como de la versión del algoritmo. Las topologías definen el entorno de interacción de una partícula individual con su vecindario, y pueden ser de dos tipos:

- Geográficos: se calcula la distancia de la partícula actual al resto y se toman las más cercanas para componer su entorno.
- Sociales: se define a priori una lista de vecinas para cada partícula, independientemente de su posición en el espacio.

En la figura 4.5 puede observarse gráficamente la relación de conocimiento de las partículas para cada una de las topologías mencionadas.

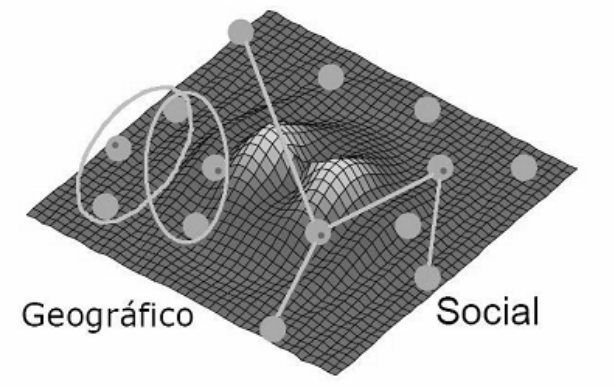

Figura 4.5: Ejemplos de entornos sociales y geográficos en un espacio de soluciones.

4.4. Aspectos avanzados de Algoritmos Basados en PSO

Un problema habitual en los algoritmos de PSO es que la magnitud del vector velocidad de las partículas puede llegar a ser muy grande, provocando que las mismas se muevan demasiado rápido, degradando las propiedades de búsqueda del algoritmo. El rendimiento puede disminuir si no se fija adecuadamente el valor de v_{max}, la velocidad máxima de cada componente del vector velocidad. En [Ebe00] se comparan dos métodos para controlar el excesivo crecimiento de las velocidades: Un *factor de inercia*, ajustado dinámicamente y un *coeficiente de constricción*.

El factor de inercia, que se ajusta dinámicamente a medida que avanza el algoritmo, se multiplica en cada iteración por el vector velocidad, logrando paulatinamente ir reduciendo la velocidad de las partículas. Su decremento debe ser correctamente ajustado para no influir negativamente en el proceso de búsqueda, una ecuación que puede lograr esto es:

$$w = w_{max} - (w_{max} - w_{min} / iter_{max}) \cdot iter$$

Donde w_{max} es el peso inicial y w_{min} el peso final, $iter_{max}$ es el número máximo de iteraciones y *iter* es la iteración actual. *w* debe mantenerse entre 0.9 y 1.2. Valores altos provocan una búsqueda exhaustiva (más *diversificación*) y valores bajos una búsqueda más localizada (más *intensificación*).

Por otra parte, el coeficiente de constricción introduce una nueva ecuación para la actualización de la velocidad. Este coeficiente asegura la convergencia, y se realiza con las siguientes ecuaciones:

$$v_i(t + 1) = K[w.v_i(t) + \varphi_1.rand_1.(pBest_i - x_i(t)) + \varphi_2.rand_2.(g_i - x_i(t))]$$

$$K = \frac{2}{2 - \varphi - \sqrt{\varphi^2 - 4\varphi}}$$

Otro aspecto a tener en consideración es el tamaño del cúmulo de partículas, pues determina el equilibro entre la calidad de las soluciones obtenidas y el costo computacional.

Recientemente se han propuesto algunas variantes que adaptan heurísticamente el tamaño del cúmulo, de manera que, si la calidad del entorno de la partícula ha mejorado pero la partícula es la peor de su entorno, entonces es eliminada. Para la solución propuesta en esta tesis en particular, como se detallará mas adelante, se agregó otro factor en el control del tamaño del cúmulo, y es el

acercamiento que se da entre ellas a medida que avanza el algoritmo. Al detectar que muchas partículas se agrupan, para evitar que el algoritmo pierda su poder de exploración, son elegidas las de peor fitness y se eliminan del cúmulo.

Finalmente, existen trabajos que proponen valores adaptativos para los coeficientes de aprendizaje φ_1 y φ_2. Los pesos que definen la importancia de los componentes *cognitivo* y *social* pueden definirse dinámicamente según la calidad de la propia partícula y del entorno.

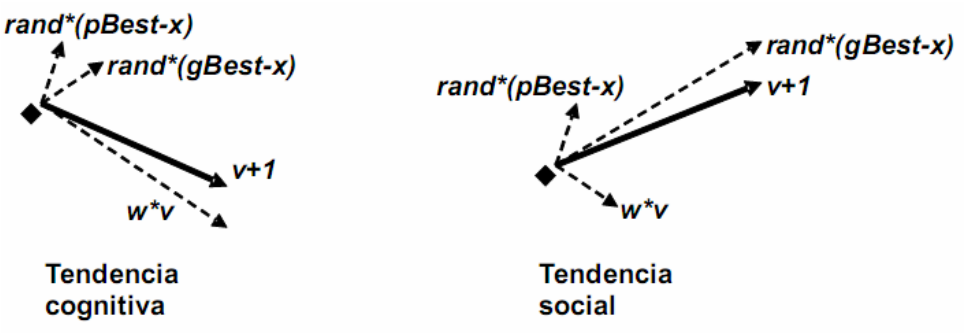

Figura 4.6: Adaptación de coeficientes de aprendizaje.

Según la Figura 4.6, cuanto mejor es una partícula, más se tiene en cuenta a sí misma (tendencia cognitiva) y no tiene en cuenta factores pasados ni externos. Por el contrario, cuento mejor es el vecino, más tiende a ir hacia él (tendencia social).

Capítulo 5

Generación de casos de prueba

La generación automática de casos de prueba para la fase de test en Ingeniería de Software es una tarea a la que se ha dedicado mucho tiempo de investigación, ya que, como se mencionó antes, es una de las etapas del proceso de desarrollo que consume mas recursos, tanto humanos como de máquina.

En la etapa de testing se intenta eliminar la mayor cantidad de “errores evitables” en el código que se ha desarrollado. También busca detectar si hay diferencias entre la salida esperada y la salida real de los procesos en evaluación. Se puede definir "errores evitables" como aquellos que son inherentes al proceso de desarrollo, errores típicos de codificación, que pueden ser detectados tempranamente con el testeo adecuado. Muchas veces se relacionan con acceso a posiciones fuera de rango en las estructuras, condiciones de terminación de loops o llamadas inválidas a subrutinas.

La generación dinámica de casos de prueba consiste en proponer de forma automática un “buen” conjunto de datos de entrada para probar un programa. Intuitivamente se puede 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 debe remitirse al concepto de criterio de adecuación (test adequacy criterion) [Mic01].

La estrategia que utiliza el generador de casos de prueba se basa en el conocimiento de información de la estructura del programa que está tratando para guiar la búsqueda de nuevos datos de entrada. Este método es conocido también como *testeo de caja blanca*. El generador utiliza, para buscar los casos de prueba que cubran el objetivo final del test, el algoritmo de PSO, con las adaptaciones pertinentes de representación de datos y con algunas variantes particulares relacionadas con la peculiaridad del problema que se está resolviendo.

Se verán en este capítulo detalles acerca del algoritmo del generador de pruebas, del método utilizado para conocer información de la estructura del programa, del criterio de adecuación usado, la función de fitness y de la implementación de PSO; como así también de las pruebas realizadas con diferentes programas característicos de este campo de investigación.

5.1. El criterio de adecuación

Diversos son los factores que deben definirse al momento de armar la estrategia de la generación de casos de prueba. Uno de ellos es determinar formalmente cuando es que un programa está *bien* testeado. Dentro de este contexto, en forma intuitiva la primera aproximación a esta idea es la de pretender que se testee la *totalidad* del programa. Así se encuentra en la literatura el término *criterio de adecuación* que no es otra cosa que el criterio que se tendrá en cuenta a la hora de determinar que tan completo ha sido un test. En este trabajo se usa el criterio de *cobertura de condiciones*. Este criterio requiere que todas las condiciones atómicas de un programa tomen los dos valores lógicos: *true* y *false*. Otros criterios muy extendidos son el de *cobertura de ramas*, que busca 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 se encuentra un conjunto de datos de entrada para el que todas las condiciones toman los dos valores lógicos se puede asegurar que todas las ramas factibles 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. Se ha elegido el criterio de cobertura de condiciones por ser más estricto que los otros dos.

Llevando este concepto concretamente al código que se va a testear, se toman las *condiciones* como las estructuras de control de selección (*if* y *case*) y las de iteración (*for* y *while*). En el caso de las de selección, se considera cubierta cuando toma ambos valores lógicos y en el caso de las de iteración, cuando se alcanza a ejecutar bloque interior al loop, ya que tanto en el caso de un *for* o un *while* si ejecutó el cuerpo, se sabe que en algún momento ejecutará el código siguiente. La pregunta inmediata que surge es acerca de cómo trataría el generador un programa que contenga un loop infinito, pero como bien se sabe por el *Problema de la Detención* formulado por *Alan Turing*, este inconveniente no puede ser resuelto computacionalmente, dejando la solución librada a la intervención humana.

Para cumplir con la meta de cumplir con todo el criterio de adecuación para un programa, el generador descompone el objetivo global en varios objetivos parciales consistiendo cada uno en hacer que una condición tome un determinado valor lógico. Luego, 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. Dicho de otra forma, una vez que se logró determinar un caso de prueba para que una de las condiciones tome alguno de los valores de verdad, el generador intentará a su turno, buscar un caso de prueba que no solo alcance la misma condición, sino que además invierta dicho valor. Para resolver el problema de minimización se utilizarán las técnicas de optimización que se detallarán más adelante.

5.2. Función de fitness

En el desarrollo del proceso de PSO, el parámetro principal que utilizan las partículas para explorar el espacio de búsqueda, es el de saber qué partícula es *mejor* o *peor* que otras. Este valor es conocido en el campo de las metaheurísticas como *fitness* y el cálculo para cada partícula depende de la *función de fitness*. En el caso del problema que se está describiendo, el valor de fitness de una partícula (que representa un caso de prueba sobre una condición particular) va a ser la distancia que la separa del valor que hace que la condición tratada tome el valor de verdad opuesto. Es decir, si un caso de prueba evalúa en *true* para una condición dada, su fitness será un valor numérico que representa la distancia a la que se encuentra la prueba de llevar a la condición al valor *false*. Cuanto más cerca se encuentre de invertir el valor de verdad, el fitness será cada vez más cercano a cero, reduciendo de esta forma cada iteración del proceso de generación de casos de prueba a un problema de minimización.

La función de fitness asociada a cada tipo de condición impone la restricción de que los datos de entrada al programa a testear sean numéricos. La tabla 5.1 resume la manera de calcular la aptitud del individuo en cada caso.

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

Tabla 5.1: Funciones de Fitness según el tipo de condición.

5.3. Modificaciones al programa evaluado

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

Para poder llevar a cabo esta tarea, el programa a ser evaluado debe ser capaz de brindar información al generador de casos de prueba en tiempo de ejecución. Esto llevó a insertar, de manera automática, líneas de código dentro del programa objeto del testeo. La intención en este aspecto era la de desarrollar un sistema fácil de usar para que cualquier programa que cumpliese con los requisitos mínimos pudiera ser testeado.

Se logró dividir el problema en dos partes:

- Por un lado, un proceso del tipo *parsing*, que toma como entrada el código que se va a testear y mediante el uso simple de expresiones regulares, agrega símbolos inocuos a la ejecución (de tipo *comentario*).

- Por otro lado, un asistente de ejecución, que dependiendo de los símbolos introducidos por el proceso anterior, es capaz de capturar información sobre los valores de cada variable y entregarlos al generador.

La salida tiene un formato muy simple: es el mismo código fuente, pero anexado con la información que necesita el generador para evaluar el paso. Los datos agregados *para cada condición* son:

- **Identificador único**: cada condición recibe un identificador único que la representará durante todo el algoritmo y con el cual se mostrará en los resultados finales las estadísticas de cobertura de la misma.

- **Resultado booleano**: valor de verdad que recibió la condición cada una de las veces que fue evaluada.

- **Valores de la desigualdad**: El valor numérico de cada uno de los terminos de la desigualdad.

Cabe aclarar que el programa se ejecuta *completo* cada vez que se evalúa un caso de prueba. Si bien en un principio parece un desperdicio de ciclos de procesador, (lo aparentemente lógico sería evaluar el programa hasta llegar a la condición que se está optimizando) se pudo observar que a medida que avanza el programa, es una particularidad que amplió en gran medida la capacidad de exploración del generador, ya que condiciones que no habían sido cubiertas o alcanzadas en el momento de su proceso tienen una “segunda chance” cuando se analizan las condiciones vecinas.

5.4. Descripción del proceso de generación

El método de generación de casos de prueba se basa en un algoritmo que en forma cíclica ajusta los valores de los conjuntos de variables de entrada en función a los valores de las condiciones que, por turno, va optimizando.

Cada conjunto de variables de entrada representará un individuo de PSO. Cada variable se mapeará internamente con una dimensión en el vector de posición del individuo.

En un principio se crea un individuo en forma aleatoria, esta es la primera entrada que recibe el programa. A partir de ese momento, el algoritmo se comporta en forma cíclica. Se toma la primera condición alcanzada pero no cubierta: esto significa que con los valores ingresados se llegó a ejecutar la condición, pero sólo alcanzó a tomar un valor de verdad.

Como se explicó, el objetivo global del proceso es dividido en objetivos individuales de optimización, esto se traduce en la implementación como una instancia de PSO en cada una de las condiciones del programa. Esto significa que cada una tendrá su propia población y los individuos de una no se mezclan con la de los demás.

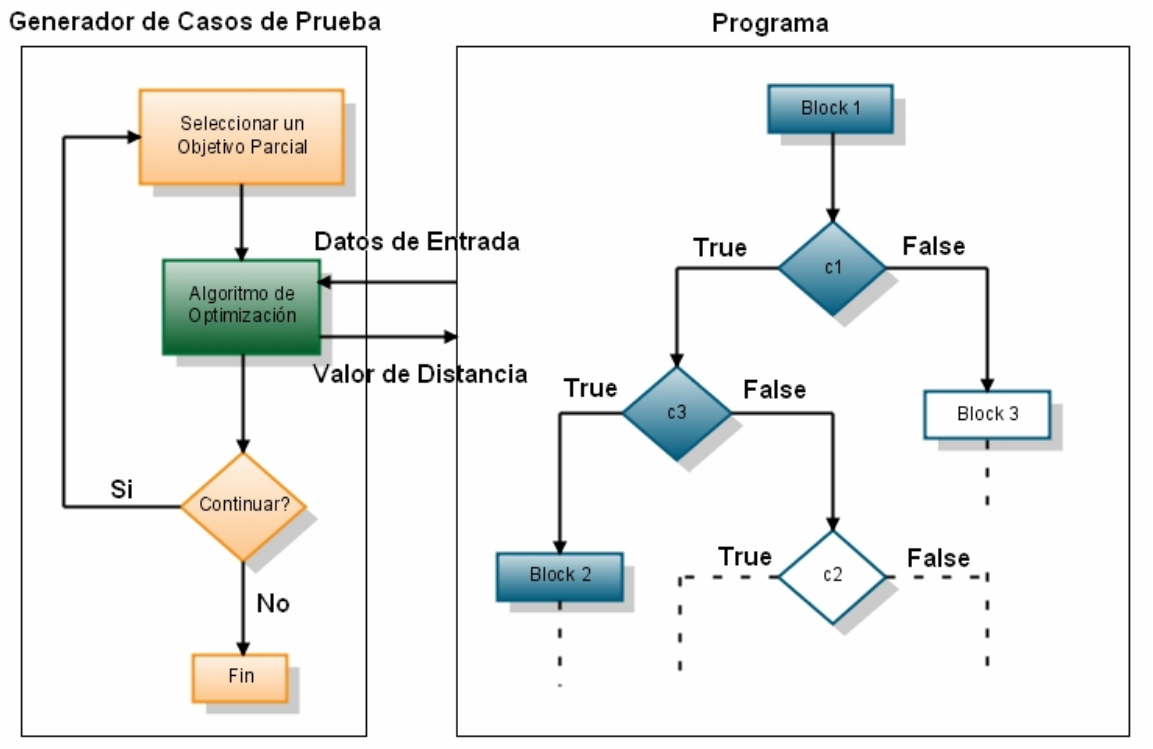

Figura 5.1: Generación de casos de prueba.

En el momento en que una condición es alcanzada en el proceso de generación por un caso de prueba, este es agregado a la población correspondiente. Si la población tiene solo un individuo (es la primera vez que la condición es alcanzada), se crea una población a partir del individuo recién llegado aplicándole variaciones con un criterio determinado: es deseable que los nuevos individuos exploren valores *cercanos* y *lejanos* para cada una de las dimensiones del individuo. Este método tiene la ventaja de que muchos de los individuos creados tendrán una gran posibilidad de volver a llegar a la condición que se está evaluando (el método se explica en las variaciones aplicadas al algoritmo de PSO). Si la condición ya tiene una población, el caso de prueba se agregará como nuevo individuo solo si no existía en dicha población.

El siguiente paso es la ejecución del algoritmo de PSO sobre la población (recién creada o aumentada) de la condición que se está intentando cubrir, y con cada uno de los nuevos casos de prueba generados (los individuos movidos) se re-ejecuta el programa original.

En cada iteración se evalúa nuevamente la condición, en caso de haberse alcanzado el objetivo (cubrir ambos valores de verdad) se selecciona la siguiente condición que haya sido alcanzada y no cubierta. En caso contrario, se vuelve a ajustar la entrada moviendo la población. Este proceso se repite hasta cubrir la condición o hasta alcanzar un número de repeticiones estipulado.

Algoritmo 3 Generador de Casos de Prueba

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

Figura 5.2: Pseudocódigo del Generador

En la Figura 5.1 se muestra un esquema de la solución, y el la figura 5.2 se presenta el pseudocódigo del método propuesto:

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, indicada en el **while**, 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 (Aplicar_PSO_Modificado). Cuando la población contiene un único individuo, se generan a partir de él la variaciones mencionadas, 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:

$$vi \leftarrow 0.75 \,.\, rand1 \,.vi + 0.75 \,.\, rand2 \,.\, (gBest - xi)$$

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

$$vi \leftarrow 0.75 \,.\, rand1 \,.vi + 0.25 \,.\, rand2 \,.\, (gBest - xi)$$

Como se explicará en la siguiente sección, 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*.

5.5. Variaciones aplicadas al algoritmo de PSO

El problema de optimización que plantea la generación de casos de prueba tiene ciertas particularidades que hicieron que la utilización del algoritmo de PSO para su resolución deba ser adaptado convenientemente.

Una de las tareas de investigación del presente trabajo fue la de detectar y comprender estas particularidades que inicialmente plantearon problemas para adaptar el proceso de optimización y convertirlas en verdaderos puntos de apoyo para el funcionamiento del algoritmo.

Se expone aquí el listado de las características que fueron encontradas en el proceso de optimización de la generación automática y de las modificaciones que se aplicaron sobre el algoritmo de PSO original para resolverlas/aprovecharlas:

5.5.1. Variaciones guiadas para creación de poblaciones

Una población lo suficientemente dispersa es una cualidad que proporciona una notable mejoría en el desarrollo de una solución basada en una metaheurística.

En la solución propuesta en esta tesis se encuentra el caso particular de tener que crear una población a partir de un único individuo. En las primeras pruebas se sumaban y restaban valores random a todas las dimensiones del vector de posición de este individuo para generar uno a uno los individuos de la nueva población. Este método tenía un inconveniente: como las modificaciones afectaban a todas las dimensiones a la vez podía ocurrir que ninguno de los nuevos individuos alcanzasen la condición en la próxima ejecución, quedando nuevamente el individuo original solo.

Evaluando esta situación se observó que el método que utiliza Tabú Search para la exploración del campo de búsqueda podría ser apropiado para la etapa de la creación de población.

Es así que la nueva población para cada condición, cuando solo tiene un individuo se crea modificando a este individuo de a una dimensión a la vez

aplicando cuatro variaciones: un valor cercano y lejano en sentido positivo y negativo.

La lejanía y cercanía de los valores se determina según el rango de valores de entrada del programa que se esté evaluando.

5.5.2. Optimización multi objetivo

La optimización que se realiza puede ser considerada *multi objetivo*, ya que busca cubrir cada una de las condiciones del programa evaluado. Esto significa que cada condición mantiene un proceso de búsqueda independiente de los demás. Por más que haya un individuo igual (mismos vectores) en la población de más de una condición, el movimiento de uno de ellos *no* representará el movimiento de los demás. Cada condición dentro del proceso utiliza su propia población independiente de las demás.

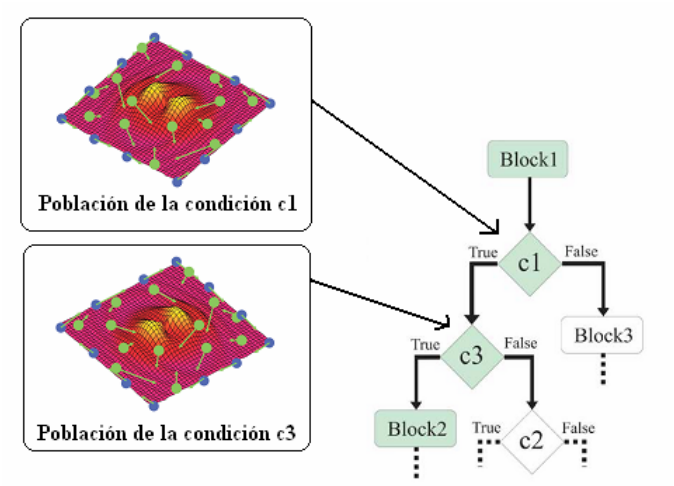

Figura 5.3: Poblaciones independientes para cada condición.

5.5.3. Capacidad de exploración

Siendo la exploración del campo de búsqueda el concepto más importante que maneja el algoritmo, fue acertado agregar ciertos comportamientos que beneficien e impulsen dicha actividad. Uno de los agregados que tuvo un gran impacto positivo en la capacidad de exploración, fue el de incorporar un individuo con valores random en cada iteración del algoritmo de PSO. Si el individuo “cae” dentro de la condición, es aceptado y tratado como uno más. En caso de no volver a entrar en la misma condición el individuo es descartado.

Esta pequeña variación brinda al algoritmo una constante fuente de diversidad, sin atentar con el desarrollo de la técnica metaheurística, lo que se traduce en una exploración más eficiente a "bajo costo".

5.5.4. Modificación del factor de inercia

Dentro del generador de casos de prueba, la utilización del algoritmo de PSO tiene un objetivo levemente distinto al de una optimización ordinaria. Lo que se busca en este caso no es que las partículas se acerquen concretamente a un valor determinado, sino más bien que se muevan en la dirección de los mejores (los valores cercanos a cero) y "crucen" este valor. La función de fitness asigna los valores cercanos a cero a aquellas partículas que están más próximas a *invertir* el valor de verdad de la condición, así por ejemplo, para la condición $a > 0$, si el proceso está intentado obtener el valor *false*, una partícula x que en su dimensión a tenga el valor 1 será "mejor" que la partícula y que tenga el valor 10. En este sentido, lo que debe pretender el algoritmo no es que y se acerque a x, sino que vaya en esa dirección y logre cruzarlo para cumplir el objetivo para esa condición.

Para resolver esto se reemplazó el parámetro w, también denominado *inercia*, normalmente usado en PSO para disminuir progresivamente la velocidad de los individuos y evitar que den pasos muy largos en una etapa avanzada del algoritmo, por una reducción lineal que ejerce mucho menos presión al momento de desacelerar las partículas. De esta forma se logró que las partículas tomaran los valores requeridos sin que el movimiento dentro del cúmulo sea descontrolado.

5.5.5. Función de fitness no continua

Debido a los diferentes flujos de ejecución que pueden tomar los programas que son sometidos al proceso de testing, se encontró el inconveniente de que las funciones de fitness que son determinadas por las condiciones no son continuas, ya que para un conjunto de valores de entrada puede ocurrir que el programa no alcance a ejecutar la condición que se está evaluando. En este caso, luego de mover a la población puede ocurrir que más de un individuo *no tenga* un fitness asociado. Esto representaba un gran problema, ya que el algoritmo se basa en dicho valor para tomar todas las decisiones. Una de las posibles soluciones era la de asignar un valor muy alto a los individuos que quedasen sin fitness, pero se llegó en ocaciones a poblaciones enteras que caían "fuera de la condición" y, por lo tanto, buscaban a ciegas.

La solución que se encontró para este caso fue agregar un comportamiento particular a los individuos por el cual si luego de moverse caían "fuera de la condición", retrocedían su paso y volvían a intentarlo en la próxima ronda de movimiento. De esta forma se logró que todos los individuos de una población

hayan sido determinados por la condición que están evaluando y así mantener un significativo aporte *social* a la búsqueda.

5.5.6. Ejecución del programa en proceso de testing.

Se sabe que el recurso más costoso que tiene el proceso de generación de casos de prueba es la ejecución del programa que se está evaluando. Para llevar adelante el proceso de optimización debe ejecutarse el código entero para *cada uno* de los individuos de las poblaciones, y así poder calcular el fitness de los conjuntos de prueba con los que se testea el programa.

Siendo un recurso tan delicado y sabiendo que una vez que el programa se ejecutó con un caso de prueba en particular ya se obtienen todos los datos posibles para esa ejecución (volver a ejecutar el programa con la misma entrada no aporta ningún dato nuevo), se añadió una parte del algoritmo de Tabú Search al proceso de generación. La variante aplicada agrega una lista en la que se almacena cada uno de los conjuntos de prueba (individuos) con los que alguna vez se ejecutó el programa. Antes de volver a correr el mismo, se revisa dicha lista y, como en el algoritmo de *Tabu Search*, en caso de que el conjunto esté presente, se procede a seleccionar al próximo conjunto, evitando de esta forma ejecutar reiteradamente el programa con un argumento ya utilizado.

En la implementación de la lista se utiliza un método de acceso de tipo hash para evitar que la búsqueda en una gran cantidad de individuos termine bajando el rendimiento global del proceso de generación.

5.5.7. Condiciones anidadas

Puede ocurrir que algunas condiciones internas en el programa sean muy difíciles de alcanzar, principalmente si se encuentran anidadas dentro de alguna secuencia de condiciones anteriores. Esto dificulta el tratamiento de dichas condiciones ya que en muchos casos, una vez que los individuos son movidos, no vuelven a ejecutar la misma condición. Para tratar este problema se implementaron dos modificaciones en el comportamiento del generador:

- Cada vez que se ejecuta el programa con un individuo o caso de prueba, este se corre en su totalidad y el individuo que está siendo evaluado es copiado en cada una de las poblaciones de las condiciones por las que pasa. De esta forma se aprovechan todos los conjuntos de prueba y se pueden encontrar soluciones particulares para una condición mientras se está minimizando otra aprovechando al máximo cada una de las corridas. De esta forma se aumenta la probabilidad de cubrir todas las condiciones, aún las de acceso difícil.

- La cantidad de individuos en las poblaciones fue un parámetro que se estudió particularmente en la implementación del método propuesto. La intención del trabajo era implementar una solución que sirva a quien la desee tomar, lo suficientemente flexible para que se adapte a cualquier programa que cumpliese los requerimientos mínimos. Se pudo observar entonces que fijar el tamaño de las poblaciones dentro del algoritmo no iba a ser una buena idea ya que los programas que se iban a tratar podían ser de cualquier tipo. Aún dentro de un mismo programa, cada una de las condiciones necesita una población distinta, por lo tanto la parametrización tampoco era una opción (llegar a determinar el tamaño de la población para un programa con cientos de condiciones podría resultar más complicado que el testeo mismo). Por lo tanto se decidió utilizar poblaciones variables, en donde los individuos se agregan cada vez que se encuentra uno que alcance la condición. Solo se eliminan cuando se aproximan mucho entre si y le quitan la capacidad de exploración a la solución. Así el algoritmo puede regular la cantidad de individuos que necesita para alcanzar cada objetivo parcial.

5.6. Métricas de cobertura

Finalmente para poder obtener la bondad de las diferentes pruebas que se realizan para los programas a testear, se debe definir algún tipo de métrica que permita determinar si los resultados obtenidos se asemejan a los esperados o no. En este caso, el objetivo último del proceso de testing es lograr una cobertura completa del programa, saber si se han ejecutado la totalidad de las líneas de código y así poder afirmar que, si en el proceso no se detectaron errores, entonces el programa está libre de "errores evitables".

La métrica que se utilizó es la *métrica de cobertura*, que indica el porcentaje de condiciones cubiertas sobre el total de condiciones existentes. Para dicha métrica, si una condición es alcanzada pero no cubierta (solo se llegó a obtener uno de los valores de verdad), la condición estará cubierta "por la mitad". Para verlo en un ejemplo, si un programa tiene una totalidad de 10 condiciones, si luego de ejecutar el generador de casos de testing todas las condiciones fueron alcanzadas pero no cubiertas, entonces el porcentaje de cobertura de la prueba será del 50%.

Capítulo 6

Experimentos

Se expondrá en esta sección las pruebas que se llevaron a cabo con el modelo propuesto con el objetivo de demostrar la mejora que este trabajo aporta al área de la generación automática de casos de prueba.

6.1. Programas

Para determinar los programas que se utilizarían en la sección de pruebas, solo fue necesario una recorrida por la bibliografía que trata el tema [Alb06][Dia05][Tuy04]. Existen ya algunos problemas clásicos en este campo y eso permite comparar los resultados obtenidos con otros trabajos existentes.

Se presentan entonces los experimentos realizados sobre un conjunto de cinco programas codificados en *Ruby*. En primer lugar se eligieron dos programas sin bucles: *triangle* que recibe la longitud de los lados de un triángulo y lo clasifica, el cual posee una gran cantidad de condiciones complejas anidadas en el que se realizan comprobaciones de igualdad entre enteros y *calday* que recibe una fecha, y una vez validada informa a qué día de la semana corresponde, cuyos objetivos parciales están muy dispersos en el espacio de búsqueda. De entre los programas con bucles, se eligieron dos técnicas de ordenación diferentes: *select* y *quicksort* y por último, *bessel*, un programa de cálculo numérico que resuelve las funciones de Bessel.

La mayoría de los códigos fuentes se han extraído del libro "C Recipes" [Pre02]. Los programas están listados en la tabla 6.1, donde se presenta la información sobre el número de condiciones, las líneas de código (LdC), el número de argumentos que recibe y una breve descripción de su objetivo.

Programa	Condiciones	LdC	Args.	Descripción
Triangle	5	31	3	Clasifica triángulos.
Calday	21	158	3	Calcula el día de la semana.
Quicksort	5	37	10	Ordenamiento utilizando QuickSort.
Select	11	83	11	k-ésimo elemento de una lista desordenada.
Bessel	14	167	2	Funciones de Bessel J_n y Y_n

Tabla 6.1: Programas utilizados en los experimentos.

6.2. Comparación con otros métodos

El otro punto a tener en cuenta para que las pruebas sean realmente objetivas es ejecutar las mismas optimizaciones pero con otros métodos para comparar sus resultados.

Los métodos que se escogieron para comparar los test son dos. Uno es el primer método que se utilizó para la generación automática de casos de prueba: el *random*. El segundo es el algoritmo basado en Tabu Search.

6.3. Parametrización

Los resultados aquí expuestos para todos los algoritmos (PSO modificado, Random y Tabu Search), son los mejores obtenidos después de haber ajustado las parametrizaciones específicas para cada uno de ellos.

En el caso de los programas testeados, el generador permite parametrizar los datos de entrada: cantidad de argumentos de entrada, el rango para cada uno de los argumentos en forma independiente, la cantidad de veces que intentará cubrir cada condición (en caso de superar el número, se supone no cubierta) y luego los parámetros específicos de funcionamiento para PSO. La cantidad y rango de los argumentos de entrada se ajustaron para cada programa en forma particular, en los demás parámetros se utilizaron los valores indicados en la tabla 6.2:

PSO	
Cantidad de partículas	Dinámico, depende de la condición.
w	Eliminado, se reduce linealmente un 25% cada vuelta.
c1	0,5
c2	0,9
Condición de Parada	Alcanzar el objetivo o 150 intentos por cada condición

Tabla 6.2: Parámetros utilizados para la implementación de PSO.

6.4. Resultados

Los resultados obtenidos después de las pruebas se muestran en la tabla 6.3.

Las variables que están en juego son el porcentaje de cobertura (*Cov.*) y la cantidad de evaluaciones promedio (*Evals.*) que debieron ejecutarse para lograr el objetivo. Se pretende que la cobertura sea la mayor posible, bajando la cantidad de evaluaciones necesarias.

Programa	PSO con variación		Random		Tabu Search	
	Cov.	Evals.	Cov.	Evals.	Cov.	Evals.
triangle	**100**	**50,72**	95,75	34,76	73,75	55,58
calday	**99,4**	**512,74**	98,43	197,21	83,04	1491,26
select	**100**	72,56	**100**	**16,55**	98,83	139,13
bessel	**100**	**483,76**	99,03	320,08	96,63	2116,25
quicksort	**100**	2,21	**100**	**2,1**	**100**	7,99

Tabla 6.3: Resultados de los experimentos.

Se puede ver que, atento a la cobertura final de los programas examinados, el método propuesto de PSO adaptado resulta en general superior a los demás. Algo a tener en cuenta mientras se observa la cantidad de evaluaciones de cada uno de los test es que la generación de individuos para las poblaciones de PSO se realizan tomando el primer individuo que alcanzó la condición y creando variaciones específicas para cada una de sus dimensiones. Por eso en programas que se resuelven en muy pocas generaciones, como son el caso del *select* o el *quicksort*, hay una menor tasa de evaluaciones para el método random ya que este no necesita crear población alguna; aún así la cobertura final de ambos métodos, PSO modificado y random, es igual y el tiempo computacional utilizado por ambos es despreciable.

Cabe recordar que el método propuesto utiliza cúmulos de partículas de tamaño variable y por tanto resulta de interés verificar el tamaño de las poblaciones para cada una de las condiciones dentro de los programas evaluados. El objetivo es verificar que el tamaño de las poblaciones se mantuviese dentro de márgenes razonables, para que no afecte el rendimiento de la ejecución sin perder capacidad de exploración. Para ello se observaron las cantidades promedio de las problaciones en cada una de las condiciones, las cuales se detallan en la tabla 6.4.

	Bessel
Cond	PSO var
1	4,05
2	4,32
3	4,23
4	3,68
5	4,97
6	4,29
7	1
8	5,15
9	6,92
10	4,34
11	1
12	1
13	11,64
14	2
15	30,13

	Calday
Cond	PSO var
1	5,67
2	3,27
3	4,79
4	4,51
5	4,76
6	4,5
7	4,48
8	4,47
9	4,88
10	4,51
11	15,47
12	6,7
13	5,16
14	6,16
15	5,31
16	6,83
17	5,89
18	4,54
19	6,59
20	8,05
21	5,67
22	6,69

	Select
Cond	PSO var
1	2,52
2	1,73
3	1
4	3,45
5	4,17
6	5,08
7	2,78
8	4,63
9	3,12
10	1,41
11	2,62
12	3,29

	Quicksort
Cond	PSO var
1	1
2	2,4
3	2,47
4	3,65
5	2,84
6	2,33

	Triangle
Cond	PSO var
1	4,51
2	3,41
3	4,38
4	3,21
5	5,97
6	5,83

Tabla 6.4: Tamaños de las poblaciones promedio al finalizar el proceso.

Observando en detalle los números se advierte que las poblaciones se encuentran en rangos razonables y que solo crecen (llegando a un máximo de 30) cuando intenta cubrir alguna condición difícil de alcanzar, como puede ser la última condición de un algoritmo numérico complejo.

Por último, otra de las medidas que se deseaba comprobar era la ingerencia que tenía la utilización de PSO en la generación de nuevos individuos. Dentro del desarrollo del algoritmo hay algunos puntos donde algunos individuos son generados al azar, este comportamiento aporta una diversidad que resulta importante en el proceso de exploración, pero es importante verificar que la generación random no "tome el control" de la aplicación, lo que significaría que la gran mayoría de los individuos sean movidos aleatoriamente en vez de aplicando el algoritmo de PSO, lo cual haría significar que las variaciones aplicadas no estarían aportando mayor beneficio. A continuación se detallan las proporciones

de creación/movimiento de individuos en forma random y aplicando PSO dentro del método propuesto:

	Calday	Triangle	Bessel	Select	Quicksort
PSO	648,83	124,59	369,08	19,64	1,85
Random	129,27	9,82	35,27	75,98	1,15

Tabla 6: Relación PSO – Random.

En este caso la evaluación vuelve a ser positiva. Si bien los números de las evaluaciones de los algoritmos de ordenación, *Select* y *Quicksort* no son los esperados, esto tiene una explicación: cuando el programa evaluado logra cubrir todas sus condiciones muy rápidamente, generalmente a causa de la simpleza de su estructura (un algoritmo de ordenación por complejo que sea nunca tendrá mas de dos condiciones anidadas), lo que ocurre es que la generación random de los primeros individuos ya logra el objetivo final del problema, sin permitir a PSO explotar la solución. Son programas que son cubiertos muy rápidamente, por lo cual no representan un problema a la solución planteada.

Por otro lado, cuando el problema tiene una complejidad estructural mayor y la cobertura de las condiciones es una tarea de búsqueda más fina, el algoritmo de PSO modificado puede explotar todo su potencial, y se verifica esto en los programas *Calday*, *Triangle* y *Bessel*, donde la tasa de utilización random apenas alcanza el 10%.

Capítulo 7

Conclusiones

Durante el desarrollo de esta tesis se ha hablado de la importancia de la Ingeniería de Software como procedimiento para el desarrollo de sistemas libres de errores. Se ha detallado la gran diversidad de problemas que se encuentran normalmente en su desarrollo y se ha relacionado la resolución de estos problemas a los algoritmos que tratan problemas de optimización.

Enunciando los tipos y características de los problemas de optimización, se llegó a las metaheurísticas para finalmente detallar historia, particularidades e implementación del algoritmo de PSO.

Habiendo observado que de entre las etapas de la Ingeniería de Software, la que mas ha captado el interés de investigar, resolver y automatizar sus problemas es la fase de Testing, se presentó una nueva alternativa para su tratamiento con un Generador Dinámico de Casos de Prueba utilizando Metaheurísticas, que en este caso se trató de PSO adaptado a las particularidades de las condiciones de resolución del problema.

En la etapa de experimentación con la solución propuesta, se pudo comprobar que el aporte mejora las soluciones existentes, aumentando el nivel de cobertura y bajando la cantidad de proceso necesario para hacerlo. Las pruebas fueron comparadas utilizando programas conocidos y algoritmos ya tratados en el campo de la generación automática.

El presente trabajo deja implementado en leguaje Ruby toda la plataforma necesaria para la utilización del Generador con cualquier programa que cumpla las restricciones mínimas de funcionamiento.

Dado que la estrategia de optimización propuesta se aplica independientemente a cada una de las condiciones, y siendo el tiempo un recurso muy valioso en esta fase, se propone como línea de trabajo futuro la paralelización de la solución planteada, con el objetivo de aumentar su rendimiento.

Otro aspecto interesante y no desarrollado en esta tesis es el análisis e implementación de nuevas funciones de fitness que permitan al método propuesto liberarse de la restricción de utilizar datos de entrada numéricos.

Anexo A

Consideraciones sobre el lenguaje Ruby

Para la implementación del presente trabajo se eligió el lenguaje de programación Ruby. Es un lenguaje relativamente nuevo que, si bien tiene apenas un poco más de diez años de vida, ha ganado muchísimo terreno desde que se publico en 1995 en Japón por Yukihiro "Mats" Matzumoto.

Se trata de un lenguaje dinámico, reflexivo y orientado a objetos, dotado de una gran flexibilidad, con implementaciones en las plataformas más importantes, que permite verificar el estado y tomar el control de los programas que ejecuta. Esta característica facilitó en gran medida la implementación del asistente de ejecución para la verificación del estado de las variables dentro de las condiciones ejecutadas. Parte de la implementación del mismo fue tomada de la librería XPM Filter, programada por Mauricio Fernández, que entrega bajo la licencia de distribución de Ruby.

Si bien se puede decir que aún Ruby no es muy utilizado en el ambiente académico, tiene dos ventajas que fueron de gran ayuda en su elección, por un lado saber que su marcado crecimiento, gracias a su flexibilidad y potencia, hará que en poco tiempo se comience a oír su presencia en este ambiente, y por otro lado se pudo comprobar que la traducción del código fuente de lenguajes usados actualmente (C#, Java) es casi directa, pudiendo implementarse con mucha facilidad un conversor automático.

Anexo B

Sobre la programación

Se entrega todo el código del generador de casos de prueba implementado. Cada uno los archivos *.rb* que componen el sistema está precedido por el nombre en **negrita**. Se agrega también uno de los programas utilizados para crear el conjunto de casos de prueba, se trata de **triangulo.rb**. La configuración del generador está lista para correr dicho programa.

El código es multiplataforma y fue testeado y utilizado sobre Windows XP y sobre Ubuntu Hardy Heron, con la versión de Ruby 1.8.6

main.rb

```
# Inclusiones de las demas clases
require 'xpmfilter'
require 'conversor'
require 'test_set_pso'
require 'if_object'
require 'helper'

#Toma el código de entrada y le hago las modificaciones (# =>)
#Llega como parámetro el nombre del archivo que se testeará
conversor = Conversor.new
code = conversor.modificar_archivo(ARGF.read)

#Elección del método de optimización
metodos = ["pso"]
#metodos = ["pso", "tabu", "random"]

# VARIABLES
cant_parameters = 3 #la cantidad de parametros de entrada que acepta el programa
max_corridas = 10 #Cantidad de veces que se ejecuta la busqueda

#para Tabu
mu = 1
la = 4

# Seteo de los intervalos de valor para cada uno de los argumentos de entrada (dimensiones), con el fomato [MIN, MAX]
value_range = [[-2,15],
  [-2,15],
  [-2,15]]

# De la misma forma se setea un intervalo de velocidad válida distinta para cada dimensión
velocity_range = [[-1.5,1.5],
  [-0.3,0.3],
  [-2.5,2.5]]

# Cantidad de veces que intentará invertir el valor de verdad de una condición
vueltas_por_if = 150

if_array = Array.new
```

```
#Cuando se agota un if, busca con random hasta que lo encuentra, la cobertura será siempre 100% (pero puede tardar indefinidamente)
buscar_agotado = false

resultados_finales = Hash.new

# Muetra los resultados parciales de cada ejecución
verbose = false
# Muestra el resultado de cada iteración del programa
super_verbose = false
# Muestra el código modificado por el proceso de instrumentación
show_code = false

# Valores de c1 y c2 para PSO
const_local = 0.5
const_global = 0.9


TestSetPso.set_params(const_local,const_global, value_range, velocity_range)

#CORRIDAS ==========================================================

metodos.each do |metodo|
  max_corridas.times do |nro_corrida|

    #VARIABLES DE LA CORRIDA
    sum_iteraciones = 0 #cantidad de vueltas que hace el algoritmo para cubrir todo
    sum_ejecuciones = 0 #Cantidad de veces que se llegó a ejecutar el programa
    evaluaciones = 0 #cantidad de ejecuciones totales de un if
                # Contadores de individuos por métodos si lograron entrar una condición
    modo_alcance = {:pso => {:if_creador => 0, :otros => 0},:random => {:if_creador => 0, :otros => 0}, :cuatroxn => {:if_creador =>
0, :otros => 0}}
    # Contadores de creaciones de indivuduos
    modo_creacion = {:pso => 0, :random => 0, :cuatroxn => 0}

    # Generar el primer individuo
    params = TestSetPso.new  #params es el primer test
    params.set_values(random_array_round(value_range))
    params.modo_creacion = :random
    modo_creacion[:random] += 1

    # Crear la lista de objetos que va a representar cada if.
    #           va a tener una lista de individuos que arranca vacía
    #Recorro el codigo modificado y armo la lista de IF
    if_array = Array.new
    array = code.split "\n"

                #Identificación de condicione: IF y WHILE
    array.size.times do |i|
      if !(/^\s*IF\s(.*)#(IF.*)\s--conector:(.*)/i =~ array[i]).nil?
        if_array << IfObject.new(i, Regexp.last_match(3), "#{Regexp.last_match(2)} - #{Regexp.last_match(1)}", :if)
      end
      if !(/^\s*WHILE\s(.*)#(WHILE.*)\s(.*)/i =~ array[i]).nil?
        if_array << IfObject.new(i, Regexp.last_match(3), "#{Regexp.last_match(2)} - #{Regexp.last_match(1)}", :while)
      end
    end

    #EJECUCION DEL PROGRAMA
    # Correr el programa
    # en res_array se obtiene el código fuente modificado con los resultados de las evaluaciones
    res_array = evaluate(code, params.to_s)
    sum_ejecuciones += 1
    puts res_array if show_code


    #EVALUO la ejeción
    if_array.each do |if_obj|
      if_obj.check(res_array, params, if_obj.id, modo_alcance)
    end

    # cargar el test_set en la lista tabu
    soluciones = Hash.new
```

```
    soluciones[params.id] = params

    #Comienzo el algoritmo PSO
    agotados = Hash.new
    ok = true
    while ok do
      sum_iteraciones += 1
      #busco el primer if evaluado pero no cubierto disponible
      if buscar_agotado
        evaluados = (if_array.select { |item| item.evaluado? })
      else
        evaluados = (if_array.select { |item| !item.agotado and item.evaluado? })
      end

      if evaluados.size == 0
        #estan todos cubiertos
        ok = false
      else
        #de todos los que encontró tomo el primero
        if_evaluar = evaluados.first
        #sumo 1 a su cantidad de evaluaciones
        agotados[if_evaluar.id] = agotados[if_evaluar.id].nil? ? 1 : agotados[if_evaluar.id] + 1
        #digo que se agotó si ya sumó "vueltas_por_if"
        if_evaluar.agotado = true if agotados[if_evaluar.id] == vueltas_por_if
                            #imprime la vuelta si está seteado
        puts "IF a tratar: #{if_evaluar.id} --- Intento #{agotados[if_evaluar.id]} --- mejor: #{if_evaluar.test_set_condicion_evaluada} ---
fitness: #{if_evaluar.fitness_condicion_evaluada} "  if super_verbose

        # Ejecuta la optimización según el método elegido
        if buscar_agotado and if_evaluar.agotado
          nuevo_random = TestSetPso.new  #params es el primer test
          nuevo_random.set_values(random_array_round(value_range))
          nuevo_random.modo_creacion = :random
          modo_creacion[:random] += 1
          vecinos = Array.new
          vecinos << nuevo_random
        else
          case metodo
          when "tabu"
            ###
            #TABU SEARCH
            vecinos = Array.new
            var = [mu, la, mu * -1, la * -1]

            soluciones[if_evaluar.test_set_condicion_evaluada].values.to_a.size.times do |i|
              var.each do |v|
                arr = Array.new(soluciones[if_evaluar.test_set_condicion_evaluada].values.to_a)
                arr[i] = arr[i] + v
                ts = TestSetPso.new
                ts.set_values(arr)
                ts.modo_creacion = :cuatroxn
                modo_creacion[:cuatroxn] += 1
                vecinos << ts
              end
            end
            ###
          when "pso"
            ###
            #PSO
            vecinos = if_evaluar.mover_pso( agotados, soluciones, vueltas_por_if, cant_parameters, modo_creacion)
            ###
          when "random"
            nuevo_random = TestSetPso.new  #params es el primer test
            nuevo_random.set_values(random_array_round(value_range))
            nuevo_random.modo_creacion = :random
            modo_creacion[:random] += 1
            vecinos = Array.new
            vecinos << nuevo_random
          end
        end
```

```
# Una vez terminado el case, en la variable "vecinos" están los individuos con los que tienen que correr nuevamente el programa.

      vecinos.each do |ts|
        # Verifico que no esté en al lista tabú
       if soluciones[ts.id].nil?
        puts "evaluando vecino #{ts.id}"  if super_verbose
        soluciones[ts.id] = ts
        res_array = evaluate(code, ts.to_s)
        sum_ejecuciones += 1
        if_array.each do |if_obj|
          evaluaciones += if_obj.check(res_array, ts, if_evaluar.id, modo_alcance) unless if_obj.cubierto?
        end
       else
        # Está en la lista tabú y no se ejecuta
        puts "skipped #{ts.id}"  if super_verbose
       end
      end

    end
  end
  IfObject.imprimir if_array  if verbose

  #Estadísiticas
  cubiertos = 0
  evaluados = 0
  total = if_array.size
  if_array.each do |if_obj|
    cubiertos += 1 if if_obj.cubierto?
    evaluados += 1 if if_obj.evaluado?
  end

  puts "-------"  if verbose
  puts "Cantidad de vueltas de while #{sum_iteraciones}"  if verbose
  resultados_finales[:sum_iteraciones] = sumar_resultado_final(resultados_finales, :sum_iteraciones, sum_iteraciones)
  puts "Cantidad de ejecuiones del programa #{sum_ejecuciones}"  if verbose
  resultados_finales[:sum_ejecuciones] = sumar_resultado_final(resultados_finales, :sum_ejecuciones, sum_ejecuciones)
  puts "Total de condiciones evaluadas: #{evaluaciones}"  if verbose
  resultados_finales[:evaluaciones] = sumar_resultado_final(resultados_finales, :evaluaciones, evaluaciones)
  puts "Condiciones:                #{total}"  if verbose
  puts "Condiciones cubiertas:      #{cubiertos}"  if verbose
  puts "Condiciones alcanzadas:     #{evaluados}"  if verbose
  porcentaje = (100 / (total.to_f * 2)) *((cubiertos * 2) + evaluados)
  resultados_finales[:porcentaje] = sumar_resultado_final(resultados_finales, :porcentaje, porcentaje)
  puts "Porcentaje cubierto         #{porcentaje}%"  if verbose
  puts "--------------------------------------------------------------------------------"      if verbose

  pso = modo_creacion[:pso]
  resultados_finales[:pso] = sumar_resultado_final(resultados_finales, :pso, pso)
  puts  "Creados con pso: #{pso}"  if verbose

  random = modo_creacion[:random]
  resultados_finales[:random] = sumar_resultado_final(resultados_finales, :random, random)
  puts  "Creados con random #{random}"  if verbose

  cuatroxn = modo_creacion[:cuatroxn]
  resultados_finales[:cuatroxn] = sumar_resultado_final(resultados_finales, :cuatroxn, cuatroxn)
  puts  "Creados con 4 x n #{cuatroxn}"  if verbose


  pso_creado = modo_alcance[:pso][:if_creador]
  resultados_finales[:pso_creado] = sumar_resultado_final(resultados_finales, :pso_creado, pso_creado)
  puts  "Entraron en el mismo if, despues de moverlos con pso: #{pso_creado}"  if verbose

  random_creador = modo_alcance[:random][:if_creador]
  resultados_finales[:random_creador] = sumar_resultado_final(resultados_finales, :random_creador, random_creador)
  puts  "Entraron en el mismo if, despues de crearlos random  : #{random_creador}"  if verbose

  cuatroxn_creador = modo_alcance[:cuatroxn][:if_creador]
  resultados_finales[:cuatroxn_creador] = sumar_resultado_final(resultados_finales, :cuatroxn_creador, cuatroxn_creador)
  puts  "Entraron en el mismo if, despues de crearlos 4 x n  : #{cuatroxn_creador}"  if verbose
```

```
    pso_otros = modo_alcance[:pso][:otros]
    resultados_finales[:pso_otros] = sumar_resultado_final(resultados_finales, :pso_otros, pso_otros)
    puts  "Entraron en otro if, despues de  moverlos con pso: #{pso_otros}"  if verbose

    random_otros = modo_alcance[:random][:otros]
    resultados_finales[:random_otros] = sumar_resultado_final(resultados_finales, :random_otros, random_otros)
    puts  "Entraron en otro if, despues de crearlos random   : #{random_otros}"  if verbose

    cuatroxn_otros = modo_alcance[:cuatroxn][:otros]
    resultados_finales[:cuatroxn_otros] = sumar_resultado_final(resultados_finales, :cuatroxn_otros, cuatroxn_otros)
    puts  "Entraron en otro if, despues de crearlos 4 x n   : #{cuatroxn_otros}"  if verbose

    if_array.each  do |if_obj|
      resultados_finales[if_obj.id.to_sym] = sumar_resultado_final(resultados_finales, if_obj.id.to_sym, if_obj.population.size)
      puts "#{if_obj.estado} - #{if_obj.id} : #{if_obj.population.size}"  if verbose
    end

    puts "vuelta:#{nro_corrida} - while corrido: #{sum_iteraciones} veces - programa ejecutado: #{sum_ejecuciones} veces - cubierto
en un #{porcentaje.to_f.precision(2)}%#{porcentaje.to_i == 100 ? ":  ' ***'}"
    ind = ""
    if_array.each  do |if_obj|
      ind += "#{if_obj.id[0..3].strip}#{(if_obj.estado[0..0] != 'C') ? (' **' + if_obj.estado[0..0] + '** ') : (' ')}#{if_obj.population.size} - "
    end
    puts ind
  end #max_corridas.times do


  puts "-------------------------------------------------------------------------------"
  puts "------------------------  Resultados totales ------------------------"
  puts "-------------------------------------------------------------------------------"
  puts "Método: #{metodo}"
  puts "Modo #{buscar_agotado ? 'Si se agota un if, busca con random hasta encontrar' : 'Si se agota un if, lo deja y busca otro'}"
  puts ""

  puts  "Creados con pso: #{(resultados_finales[:pso].to_f / max_corridas).precision(2)}"
  puts  "Creados con random #{(resultados_finales[:random].to_f / max_corridas).precision(2)}"
  puts  "Creados con 4 x n #{(resultados_finales[:cuatroxn].to_f / max_corridas).precision(2)}"
  puts "Cantidad de vueltas promedio de while #{(resultados_finales[:sum_iteraciones].to_f / max_corridas).precision(2)}"
  puts "Cantidad de ejecuiones promedio del programa #{(resultados_finales[:sum_ejecuciones].to_f / max_corridas).precision(2)}"
  puts "Total de condiciones evaluadas promedio: #{(resultados_finales[:evaluaciones].to_f / max_corridas).precision(2)}"
  puts "Porcentaje cubierto en todas las corridas promedio #{(resultados_finales[:porcentaje].to_f / max_corridas).precision(2)}%"
  puts "-------------------------------------------------------------------------------"

  puts  "Entraron en el mismo if, despues de moverlos con pso: #{(resultados_finales[:pso_creado].to_f / max_corridas).precision(2)}"
  puts      "Entraron en el mismo if, despues de crearlos random      : #{(resultados_finales[:random_creador].to_f      /
max_corridas).precision(2)}"
  puts      "Entraron en el mismo if, despues de crearlos 4 x n      : #{(resultados_finales[:cuatroxn_creador].to_f      /
max_corridas).precision(2)}"
  puts  "Entraron en otro if, despues de  moverlos con pso: #{(resultados_finales[:pso_otros].to_f / max_corridas).precision(2)}"
  puts  "Entraron en otro if, despues de crearlos random   : #{(resultados_finales[:random_otros].to_f / max_corridas).precision(2)}"
  puts  "Entraron en otro if, despues de crearlos 4 x n   : #{(resultados_finales[:cuatroxn_otros].to_f / max_corridas).precision(2)}"

  puts "Promedio de poblaciones"
  if_array.each  do |if_obj|
    puts "      #{if_obj.id} : #{((resultados_finales[if_obj.id.to_sym]).to_f  / max_corridas).to_f.precision(2)}"
  end
end #metodos.each do |metodo|
```

conversor.rb

```
class Conversor
  def buscar_condicion(linea)
    # Busca el tipo de desigualdad de la condición
    cond = if linea.include? "=="
      "=="
    elsif linea.include? "!="
      "!="
    elsif linea.include? "<="
      "<="
    elsif linea.include? ">="
      ">="
    elsif linea.include? "<"
```

```
    "<"
  elsif linea.include? ">"
    ">"
  else
    "ERROR"
  end
  return cond
 end

 def crear_agregados_codigo(condicion)
  condicion.gsub!(/\sor\s/i, ' OR ')
  condicion.gsub!(/\sand\s/i, ' AND ')
  if (condicion.upcase.include?('OR') and condicion.upcase.include?('AND'))
    return "ERROR", "ERROR", "ERROR"
  else
    if condicion.upcase.include?('OR')
      conector = "OR"
    elsif condicion.upcase.include?('AND')
      conector = "AND"
    else
      conector = "--NO_CONECTOR--"
    end

    operadores = Array.new
    parciales = Array.new
    elementos = condicion.split(conector)
    elementos.each do |elem|
      desigualdad = buscar_condicion(elem)
      if desigualdad == 'ERROR'
        return "ERROR", "ERROR", "ERROR"
      else
        opers = elem.split(desigualdad)
        operadores << "[#{opers[0]},#{opers[1]},'#{desigualdad}']"
        parciales << "[#{opers[0]} #{desigualdad} #{opers[1]}]"
      end
    end
    operadores_s = "[#{operadores.join(',')}]"
    parciales_s = "[#{parciales.join(',')}]"
    return operadores_s, parciales_s, conector
  end

 end

 def modificar_archivo(code)
  salida = []
  if_id = 0
  lines = code.split("\n") #separo en lineas
  lines.each do |line|
    if (/^((\s*)(IF|WHILE)\s(.*))/i =~ line).nil?
      #no es if
      salida << line
    else
      if_comp = Regexp.last_match(1)
      if_spc = Regexp.last_match(2)
      control = Regexp.last_match(3)
      if_cond = Regexp.last_match(4)

      operadores,  parciales, conector = crear_agregados_codigo(if_cond.strip.gsub(/do$/i, " "))

      salida << "#{if_spc}#{operadores} # =>"
      salida << "#{if_spc}#{parciales} # =>"
      salida << "#{if_spc}#{if_cond.strip.gsub(/do$/i, " ")} # =>"
      salida << "#{if_spc}#{if_comp.strip} ##{control.upcase}#{if_id}  --conector:#{conector}"
      if_id += 1
    end
  end
  return salida.join("\n")
 end
end
```

test_set_pso.rb

```
require 'matrix'

class TestSetPso
  attr_accessor :values
  attr_accessor :values_anterior
  attr_accessor :values_pbest
  attr_accessor :velocity
  attr_accessor :velocity_anterior
  attr_accessor :fitness_pbest
  attr_accessor :modo_creacion
  attr_accessor :volvio_a_pasar

  @fitness

  @@const_local = 0
  @@const_global = 0
  @@value_range = 0
  @@velocity_range = 0

  def recuperar_posicion_anterior
    @values = @values_anterior
    @velocity = @velocity_anterior
  end

  def guardar_posicion_anterior
    @values_anterior = @values
    @velocity_anterior = @velocity
  end

  def self.set_params(const_local, const_global, value_range, velocity_range)
    @@const_local = const_local
    @@const_global = const_global
    @@value_range = value_range
    @@velocity_range = velocity_range
  end

  def self.value_range
    @@value_range
  end

  def self.const_local
    @@const_local
  end

  def self.const_global
    @@const_global
  end

  def to_s
    "[" + @values.to_a.join(", ") + "]"
  end

  def initialize(*list)
    @values = Vector.elements(list)
    @volvio_a_pasar = false
    set_velocity
  end

  def set_values(values)
    @values = Vector.elements(values)
    set_velocity
  end

  def set_velocity
    #la velocidad se inicializa random
    aux = random_array(@@velocity_range)
    @velocity = Vector.elements(aux)
    @fitness_pbest = 1000000
  end

  def get_fitness
```

```
    @fitness
  end

  def fitness(fitness, test_set)
    @fitness = fitness
    if @fitness_pbest.abs > fitness.abs
      @fitness_pbest = @fitness
      @values_pbest = test_set.values.clone
    end
  end
  def id
    @values.to_a.join(" | ").to_sym
  end
end
```

if_object.rb

```
require 'helper'
class IfObject

  @line = 0
  #@cond = ""
  @conector = ""
  @values
  @id = nil
  @fitness_anterior = ""

  #PSO

  attr_accessor :population
  attr_accessor :agotado
  attr_accessor :estructura

  def initialize(line, conector, id, estructura)
    @line = line
    @conector = conector
    @id = id
    @values = Hash.new
    @population = Array.new
    @estructura = estructura
    self.agotado = false
    @fitness_anterior = ""

  end

  def id
    @id
  end

  #Recibe el array con las lineas evaluadas del programa
  #Busca sus propios resultados
  #y llama a check_each para evaluar cada vuelta de ejecución
  #devuelve la cantidad de veces que se el programa ejecutó la línea
  def check(res_array, test_set, if_id, modo_alcance)

    #busco las evaluaciones del if: "true, false, false..."
    if !(/.*# =>\s(.*)/ =~ res_array[@line]).nil?
      resultados = Regexp.last_match(1).split ","
      resultados.map {|item| item.strip!}
      #Si se evaluó más de una vez...
      resultados.size.times do |i|
        check_each(res_array, test_set, eval(resultados[i]) ,i, if_id, modo_alcance)
        if self.cubierto?
          break
        end
      end
      return resultados.size
    else
      return 0
    end
  end
```

```
 def array_min(array)
   min = 99999999
   array.each do |i|
     min = i if i < min
   end
   min
 end

 def reducir_poblacion
   #armo una matriz con las distancias
   mat = Array.new
   @population.size.times do |i|
     arr_aux = Array.new
     @population.size.times do |j|
       arr_aux << distancia(@population[i].values, @population[j].values)
     end
     mat << arr_aux
   end

   #busco la distancia menor que cada uno tiene con un individuo
   minimos = Array.new
   mat.each do |array|
     minimos << array_min(array)
   end
   #busco la menor distancia
   minimo = array_min(minimos)
   cant_min = (minimos.select { |item| item == minimo }).size
   if (cant_min > 2) and (@population.size > cant_min)
     minimos.size.times do |i|
       if minimos[i] == minimo
         @population.delete_at(i)
       end
     end
   end

 end

 def check_each(res_array, test_set, res, index, if_id, modo_alcance)
   #Busco si el individuo que entró ya estaba
   ts =  @population.find {|item| item.eql? test_set}  #La igualdad solo da true si es el MISMO objeto. Como al agregarlo se clona, ya
es otro (es uno distinto por if)
   if ts.nil? #NO esta
     #Lo tengo que agregar a la población
     ts = test_set.clone
     @population << ts
     agrego_individuo = true #si algregó individuo al final del metodo elimino
   end
   test_set = ts
   test_set.volvio_a_pasar = true

   #contar como entro: random o PSO
   if if_id == self.id
     modo_alcance[test_set.modo_creacion][:if_creador] = modo_alcance[test_set.modo_creacion][:if_creador] + 1
   else
     modo_alcance[test_set.modo_creacion][:otros] = modo_alcance[test_set.modo_creacion][:otros] + 1
   end

   #Calculo el fitness de la ejecución
   fitness = calcular_fitness(res_array, index)

   test_set.fitness(fitness, test_set) #lo almaceno porque pso necesita saber el fitness de cada individuo

   if res
     if !@values[:true].nil?
       if @values[:true][:fitness] > fitness

         @values[:true][:fitness] = fitness
         @values[:true][:test_set] = test_set.id
       end
     else
       @values[:true] = {:fitness => fitness, :test_set => test_set.id}
```

```
    end
    if self.estructura == :while #Si es WHILE y dió true, es porque el false ya lo tiene
      if !@values[:false].nil?
        if @values[:false][:fitness] < fitness

          @values[:false][:fitness] = fitness
          @values[:false][:test_set] = test_set.id
        end
      else
        @values[:false] = {:fitness => fitness, :test_set => test_set.id}
      end
    end
  else
    if !@values[:false].nil?
      if @values[:false][:fitness] < fitness

        @values[:false][:fitness] = fitness
        @values[:false][:test_set] = test_set.id
      end
    else
      @values[:false] = {:fitness => fitness, :test_set => test_set.id}
    end
  end
  reducir_poblacion if agrego_individuo
end

def fitness_desigualdad(op1, op2, cond)
  return case cond
  when "==", "!="
    (op1 - op2).abs
  when "<", "<="
    op2 - op1
  when ">", ">="
    op1 - op2
  else
    nil
  end
end

def fitness_minimo(evaluacion)
  minimo = Array.new
  evaluacion.each do |eva|
    minimo <<  fitness_desigualdad(eva[0].to_f, eva[1].to_f, eva[2].strip)
  end
  return minimo.sort{|x,y| x.abs <=> y.abs}[0]
end


#Recibe los resultados, la vuelta de ejecución, busca sus operadores y calcula el fitness en funcion a la condición
def calcular_fitness(res_array, index)
  /.*# =>(.*)/ =~ res_array[@line - 2]
  evaluacion = eval("[#{Regexp.last_match(1)}]")[index]
  if evaluacion.size == 1
    #no tiene and ni or
    return fitness_desigualdad(evaluacion[0][0].to_f, evaluacion[0][1].to_f, evaluacion[0][2].strip)
  else
    #      tiene conector
    if @conector == "AND"
      return fitness_minimo(evaluacion)
    elsif @conector == "OR"
      /.*# =>(.*)/ =~ res_array[@line - 1]
      resultados = eval("[#{Regexp.last_match(1)}]")[index]
      if resultados.select{|i| !i[0]}.size == 0
        #no hay false
        return fitness_minimo(evaluacion)
      else
        # hay false
        #sumatoria de los costos falsos
        res = 0
        resultados.size.times do |j|
          if !resultados[j][0]
```

```
          res += fitness_desigualdad(evaluacion[j][0].to_f, evaluacion[j][1].to_f, evaluacion[j][2].strip)
        end
      end
      return res
    end
  end
end

def estado
  if self.agotado
    res = "AGOTADO"
  else
    if @values[:true].nil? and @values[:false].nil?
      res = "NO EVALUADO"
    else
      if !@values[:true].nil?
        if !@values[:false].nil?
          res = "CUBIERTO"
        else
          res = "EVALUADO"
        end
      else
        res = "EVALUADO"
      end
    end
  end
  res
end

def cubierto?
  !@values[:true].nil? and !@values[:false].nil?
end

def evaluado?
  (@values[:true].nil? and !@values[:false].nil?) or (!@values[:true].nil? and @values[:false].nil?)
end

def true?
  !@values[:true].nil? and @values[:false].nil?
end

def false?
  !@values[:false].nil? and @values[:true].nil?
end

def test_set_condicion_evaluada
  #devuelve el test_set de la condicion evaluada.
  #si no es "evaluado" devuelve nil
  if self.evaluado?
    if self.true?
      return @values[:true][:test_set]
    else
      return @values[:false][:test_set]
    end
  else
    return nil
  end
end

def fitness_condicion_evaluada
  #devuelve el fitness de la condicion evaluada.
  #si no es "evaluado" devuelve nil
  if self.evaluado?
    if self.true?
      return @values[:true][:fitness]
    else
      return @values[:false][:fitness]
    end
  else
    return nil
```

```
  end
end

def imprimir
  puts @id.to_s
  puts self.estado
  puts "TRUE: " + (@values[:true].nil? ? "--" : "fitness=#{@values[:true][:fitness]} test_set=#{@values[:true][:test_set]}")
  puts "FALSE: " + (@values[:false].nil? ? "--" : "fitness=#{@values[:false][:fitness]} test_set=#{@values[:false][:test_set]}")
  puts "-------------"
end

#Imprime los resultados
def self.imprimir(if_hash)
  if_hash.each do |if_obj|
    if_obj.imprimir
  end
end

def crear_individuo_con_variacion(poblacion_retorno, mejor_clonado, nro_atrib, signo, factor, paso, modo_creacion)

  esta = true
  while esta
    #tomo el vector
    variacion = mejor_clonado.values.to_a
    #lo modifico
    variacion[nro_atrib] = variacion[nro_atrib] + (signo * factor * rand * (TestSetPso.value_range[nro_atrib][1])).round

    nuevo = TestSetPso.new  #params es el primer test
    nuevo.set_values(variacion)
    nuevo.modo_creacion = :random
    modo_creacion[:random] += 1

    esta = poblacion_retorno.find{|ind| ind.id == nuevo.id}
    factor = factor - signo * paso
  end
  poblacion_retorno << nuevo
end


#PSO
def mover_pso(vueltas, soluciones, max_iteraciones, cant_parameters, modo_creacion)
  #Cada if tiene su propia población, creada a partir del mejor test_set
  #En función a los elementos que están en la población, se va a generar una nueva, para evaluar en la próxima corrida

  #Crear la población para devolver
  poblacion_retorno = Array.new

  #Si hay 0 ó 1 elemento creo 4*N haciendo que los pasos cortos no den como resultado un individuo igual
  if @population.size == 1
    #Obtengo el mejor
    mejor_clonado = soluciones[test_set_condicion_evaluada].clone

    cant_parameters.times do |nro_atrib|
      crear_individuo_con_variacion(poblacion_retorno, mejor_clonado, nro_atrib, 1, 1, 0.5,modo_creacion)
      crear_individuo_con_variacion(poblacion_retorno, mejor_clonado, nro_atrib, -1,1, 0.5,modo_creacion)
      crear_individuo_con_variacion(poblacion_retorno, mejor_clonado, nro_atrib, 1, 0.1, 0.1,modo_creacion)
      crear_individuo_con_variacion(poblacion_retorno, mejor_clonado, nro_atrib, -1, 0.1, 0.1,modo_creacion)

    end
  else
    #Como primer elemento le envío un random (va siempre)
    #TODO se puede agregar solo si el fitness no mejoro
    if @fitness_anterior == ""
      @fitness_anterior = fitness_condicion_evaluada
      agregar_random = true
    else
      if @fitness_anterior == fitness_condicion_evaluada
        agregar_random = true
      else
        agregar_random = false
      end
```

```
end

# if agregar_random
random = TestSetPso.new  #params es el primer test
random.set_values(random_array_round(TestSetPso.value_range))
random.modo_creacion = :random
poblacion_retorno << random
modo_creacion[:random] += 1
# end
#Busco el máximo global

# Creo una lista de fitness, la ordeno, tomo el primer y segundo fitness diferente
#De todos los que tengan ese fitness tomo dos al azar y promedio. ESO será lo que siga la poblacion
array_fitness = Array.new
aux  =  @population.map {|ind| ind.get_fitness}.sort.uniq
array_fitness << aux[0]
array_fitness << aux[1]
array_fitness = array_fitness.select {|fit| !fit.nil?}   #Si el aux tenía un solo elemento, le quito el nil
mejores_candidatos =  Array.new
mejores =  Array.new
array_fitness.each do |mejor_fit|
  mejores_candidatos += @population.select {|pop| pop.get_fitness == mejor_fit}
end
#desordeno el array
mejores_candidatos.sort {|x,y| rand <=> rand}
if mejores_candidatos.size > 1
  mejores << mejores_candidatos[0]
  mejores << mejores_candidatos[1]
  mejor_direccion = (mejores[0].values + mejores[1].values) * 0.5
else
  mejor_direccion = mejores_candidatos[0].values
end
gv_best = mejor_direccion

#Modifico la posición de las particulas
@population.each do |tsp|
  if not tsp.volvio_a_pasar
    tsp.recuperar_posicion_anterior
  end
end
@population.each do |tsp|

  rand_local = rand
  rand_global = rand
  #Nuevo calculo, solo lo reduzco en un 75%
  w_local = 0.75

  aux_w = tsp.velocity * w_local

  aux_local = tsp.values_pbest - tsp.values
  aux_local = aux_local * (TestSetPso.const_local * rand_local)

  aux_global = gv_best - tsp.values
  aux_global = aux_global * (TestSetPso.const_global * rand_global)

  tsp.guardar_posicion_anterior
  tsp.velocity = aux_w + aux_local + aux_global
  #tsp.velocity = Vector.elements(tsp.velocity.to_a.map {|i| i.precision(2)})

  tsp.values = tsp.values + tsp.velocity
  tsp.values = Vector.elements(tsp.values.to_a.map {|elem| elem.round})

  #control para que se salgan del rango
  arr = tsp.values.to_a
  arr.size.times do |j|
    arr[j] = ((arr[j] <TestSetPso.value_range[j][0]) ? TestSetPso.value_range[j][0] : arr[j])
    arr[j] = ((arr[j] >TestSetPso.value_range[j][1]) ? TestSetPso.value_range[j][1] : arr[j])
  end

  tsp.modo_creacion = :pso
  modo_creacion[:pso] += 1
```

```
      tsp.values = Vector.elements(arr)
      tsp.volvio_a_pasar = false
      poblacion_retorno << tsp
    end
  end
  return poblacion_retorno
 end
end
```

xpmfilter.rb

```
#!/usr/bin/env ruby
# Copyright (c) 2005-2006 Mauricio Fernandez <mfp@acm.org> http://eigenclass.org
#
# Use and distribution subject to the terms of the Ruby license.

class XMPFilter
 VERSION = "0.2.0"

 MARKER = "!XMP#{Time.new.to_i}_#{Process.pid}_#{rand(1000000)}!"
 XMP_RE = Regexp.new("^" + Regexp.escape(MARKER) + '\[([0-9]+)\] (=>|~>) (.*)')
 VAR = "_xmp_#{Time.new.to_i}_#{Process.pid}_#{rand(1000000)}"
 WARNING_RE = /-:([0-9]+): warning: (.*)/

 INITIALIZE_OPTS = {:interpreter => "ruby", :options => [], :libs => [],
  :include_paths => []}
 def initialize(opts = {})
  options = INITIALIZE_OPTS.merge opts
  @interpreter = options[:interpreter]
  @options = options[:options]
  @libs = options[:libs]
  @include_paths = options[:include_paths]
 end

 def add_markers(code, min_codeline_size = 50)
  maxlen = code.map{|x| x.size}.max
  maxlen = [min_codeline_size, maxlen + 2].max
  ret = ""
  code.each do |l|
   l = l.chomp.gsub(/ # (=>|!>).*/, "").gsub(/\s*$/, "")
   ret << (l + " " * (maxlen - l.size) + " # =>\n")
  end
  ret
 end

 def add_annotations(code, params)
  annotate(code, :annotation, params)
 end

 def add_assertions(code)
  annotate(code, :unittest)
 end

 private
 def annotate(code, mode, params)

  code = "params = #{params} \n" + code

  raise "Unknown mode" unless [:annotation, :unittest].include? mode

  idx = 0
  newcode = code.gsub(/^(.*) # =>.*/) do |l|
   line = $1
   case mode
   when :annotation
    prepare_line_annotation(line, idx += 1)
   when :unittest
    prepare_line_assertion(line, idx += 1)
   end
  end
  stdout, stderr = execute(newcode)
  output = stderr.readlines
```

```
  results, exceptions = extract_data(output)
  idx = 0
  annotated = code.gsub(/^(.*) # =>.*/) do |l|
    expr = $1
    if /^\s*#/ =~ l
      l
    else
      case mode
      when :annotation
        "#{expr} # => " + (results[idx+=1].map{|x| x[1]} || []).join(", ")
      when :unittest
        indent =  /^\s*/.match(l)[0]
        assertions(expr.strip, results, exceptions, idx+=1).map{|x| indent + x}.join("\n")
      end
    end
  end.gsub(/ # !>.*/, '').gsub(/# (>>|~>)[^\n]*\n/m, "");
  ret = final_decoration(annotated, output)
  if mode == :annotation and (s = stdout.read) != ""
    ret << s.inject(""){|s,line| s + "# >> #{line}".chomp + "\n" }
  end
  ret
end

def prepare_line_annotation(expr, idx)
  v = "#{VAR}"
  %!((#{v} = (#{expr}); $stderr.puts("#{MARKER}[#{idx}] => " + #{v}.class.to_s + " " + #{v}.inspect) || #{v}))!
end

def prepare_line_assertion(expr, idx)
  basic_eval = prepare_line_annotation(expr, idx)
  %|begin; #{basic_eval}; rescue Exception; $stderr.puts("#{MARKER}[#{idx}] ~> " + $!.class.to_s); end|
end

if /win|mingw/ =~ RUBY_PLATFORM && /darwin/ !~ RUBY_PLATFORM
  require 'tempfile'
  def execute(code)
    stdin, stdout, stderr = (1..3).map{ Tempfile.new("xmp_filter") }
    stdin.puts code
    [stdin, stdout, stderr].each{|x| x.close}
    exe_line = <<-EOF.map{|l| l.strip}.join(";")
      $stdout.reopen('#{stdout.path}', 'w')
      $stderr.reopen('#{stderr.path}', 'w')
      $0.replace '#{stdin.path}'
      ARGV.replace(#{@options.inspect})
      load #{stdin.path.inspect}
    EOF
    system(*(interpreter_command << "-e" << exe_line))
    stdout.open
    stderr.open
    [stdout, stderr]
  end
else
  require 'open3'
  def execute(code)
    stdin, stdout, stderr = Open3::popen3(*interpreter_command)
    stdin.puts code
    stdin.close
    [stdout, stderr]
  end
end

def interpreter_command
  r = [@interpreter, "-w"]
  r << "-I#{@include_paths.join(":")}" unless @include_paths.empty?
  @libs.each{|x| r << "-r#{x}" } unless @libs.empty?
  r
end

def extract_data(output)
  results = Hash.new{|h,k| h[k] = []}
  exceptions = Hash.new{|h,k| h[k] = []}
```

```
  output.grep(XMP_RE).each do |line|
    result_id, op, result = XMP_RE.match(line).captures
    case op
    when "=>"
      klass, value = /(\S+)\s+(.*)/.match(result).captures
      results[result_id.to_i] << [klass, value]
    when "~>"
      exceptions[result_id.to_i] << result
    end
  end
  [results, exceptions]
end

def final_decoration(code, output)
  warnings = {}
  output.join.grep(WARNING_RE).map do |x|
    md = WARNING_RE.match(x)
    warnings[md[1].to_i] = md[2]
  end
  idx = 0
  ret = code.map do |line|
    w = warnings[idx+=1]
    w ? (line.chomp + " # !> #{w}") : line
  end
  output = output.reject{|x| /^-:[0-9]+: warning/.match(x)}
  if exception = /^-:[0-9]+:.*/m.match(output.join)
    ret << exception[0].map{|line| "# ~> " + line }
  end
  ret
end

def assertions(expression, results, exceptions, index)
  if !(wanted = results[index]).empty? || !exceptions[index]
    case wanted[0]
    when nil
      ["assert_nil(#{expression})"]
    else
      case wanted.size
      when 1
        value_assertions(wanted[0], expression)
      else
        # discard values from multiple runs
        ["#xmpfilter: WARNING!! extra values ignored"] +
          value_assertions(wanted[0], expression)
      end
    end
  else
    ["assert_raise(#{exceptions[index][0]}){#{expression}}"]
  end
end

OTHER = Class.new
def value_assertions(klass_value_txt_pair, expression)
  klass_txt, value_txt = klass_value_txt_pair
  value = eval(value_txt) || OTHER.new
  # special cases
  value = nil if value_txt.strip == "nil"
  value = false if value_txt.strip == "false"
  case value
  when Float
    ["assert_in_delta(#{value.inspect}, #{expression}, 0.0001)"]
  when Numeric, String, Hash, Array, Regexp, TrueClass, FalseClass, Symbol, NilClass
    ["assert_equal(#{value_txt}, #{expression})"]
  else
    [ "assert_kind_of(#{klass_txt}, #{expression})",
      "assert_equal(#{value_txt.inspect}, #{expression}.inspect)" ]
  end
rescue
  return [ "assert_kind_of(#{klass_txt}, #{expression})",
    "assert_equal(#{value_txt.inspect}, #{expression}.inspect)" ]
end
```

```
end

def evaluate(code, params)

  xmp = XMPFilter.new(:interpreter => 'ruby', :options => [],
    :include_paths => [], :libs => [])
  return xmp.add_annotations(code, params)
end
```

helper.rb

```
def random_range(min, max)
  (rand * (max - min)) + min
end

def random_range_round(min, max)
  random_range(max, min).round
end

def random_array(array_range)
  set = Array.new
  array_range.each do |range|
    set << random_range(range[0], range[1])
  end
  return set
end

def random_array_round(array_range)
  set = Array.new
  array_range.each do |range|
    set << random_range_round(range[0], range[1])
  end
  return set
end

def sumar_resultado_final(resultado_final, key, valor)
  resultado_final[key].nil? ? valor : resultado_final[key] + valor
end

def distancia(vec_1, vec_2)
  #está tocada para que la distancia a si mismo sea un número grande
  if vec_1.eql?(vec_2)
    return 99999
  else
    arr_aux = (vec_1 - vec_2).to_a
    sum = 0
    arr_aux.map {|i| sum += i ** 2}
    return Math.sqrt(sum)
  end

end

class Float
  def precision(pre)
    mult = 10 ** pre
    (self * mult).truncate.to_f / mult
    #self.round
  end
end
```

triangulo.rb

```
a = params[0]
b = params[1]
c = params[2]

if a > 0 and b > 0 and c > 0
  if  2 * a < a + b + c and 2 * b < a + b + c and 2 * c < a + b + c

    if a == b
      if b == c
        res = "equilatero"
```

```
      else
        res = "isosceles"
      end
    else
      if a == c
        res = "isosceles"
      else
        if b == c
          res = "isosceles"
        else
          res = "escaleno"
        end
      end
    end

  else
    res = "NO"
  end
else
  res = "NO"
end

puts res
```

Anexo C

Paper aceptado en CICA 2009

Se presenta en este anexo el paper "Generación dinámica de casos de prueba utilizando metaheurísticas", basado en la presente tesina, el cual fue aceptado para su publicación en el **I Congreso de Inteligencia Computacional Aplicada**, a realizarse en Buenos Aires durante el mes de julio de 2009.

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

and

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

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.

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

1. Introducción

La generación automática 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.

Otra alternativa utilizada para resolver este problema es la *generación de datos de prueba simbólica.* Consiste en utilizar valores simbólicos para las variables en vez de valores reales dando lugar a una ejecución simbólica. A partir de dicha ejecución se obtienen restricciones algebraicas que permiten obtener los casos de prueba.

Un tercer paradigma es la *generación dinámica de datos de prueba*. En este caso, el programa es modificado para brindarle información al generador de casos de prueba 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, hill-climbing y simulated annealing. Algunas soluciones más recientes utilizan Tabu Search y Scatter Search.

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 técnica heurística 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.

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:

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

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(x_i) es mejor que fitness(pBest_i) then
             pBest_i ← x_i; fitness(pBest_i) ← fitness(x_i)
        end if
    end for
    for i = 1 to size(S)  do
        Elegir gi según el criterio de vecindad utilizado
        v_i ← w . v_i + φ1 . rand1 . (pBest_i − x_i) + φ2 . rand2 . (g_i − x_i)
        x_i ← x_i + v_i
    end for
end while
Salida : la major solución encontrada
```

Figura 1

3. Descripción de la solución propuesta

El método de generación de casos de prueba se basa en un algoritmo que en forma cíclica ajusta los valores de las variables de entrada en función a los valores de las condiciones que, por turno, va optimizando.

A cada vuelta se evalúa nuevamente la condición, si se alcanzó el objetivo (cubrir la condición) se pasa a la siguiente condición y si no se vuelve a ajustar la entrada. Este proceso se repite hasta cubrir la condición o hasta alcanzar un número repeticiones estipulado por condición.

El criterio que se utilizó para determinar si un programa era cubierto o no, fue el de cobertura por condición, que indica que un programa fue completamente cubierto si cada una de sus condiciones tomó ambos valores de vedad. A diferencia del trabajo realizado en [x] las condiciones se evaluaron con sus conectores lógicos en vez de en forma atómica, con lo que se logró alcanzar una mejor cobertura respetando el criterio mencionado.

El método de optimización que se utilizó fue PSO con algunas variantes debido a la particularidad del problema a resolver.

Dichas características con nuestras soluciones son:

- La optimización es multi objetivo, ya que debe ejecutarse por cada una de las condiciones del programa evaluado, por lo cual se utilizaron poblaciones independientes para cada una
- La optimización del problema no busca llegar a un valor específico, sino ir en la dirección del mismo y poder "cruzarlo" para invertir el valor de verdad de la condición. Para resolver esto se reemplazo el parámetro *w*, normalmente usado en PSO para disminuir progresivamente la velocidad de los individuos con una reducción lineal y de esta forma se logró no mermar su capacidad de exploración a la vez que se mantenían dentro de parámetros de movimiento aceptable.
- Por los diferentes flujos de ejecución que puede tomar un programa nos encontramos con el inconveniente que las funciones que determinan las condiciones no son continuas, ya que para un conjunto de valores de entrada puede ocurrir que no alcanza a ejecutarse la condición que se está evaluando. En este caso, luego de mover a la población nos podemos encontrar con que más de un individuo no tiene un fitness asociado. En este caso se agregó un comportamiento particular a los individuos por cual si luego de moverse caían "fuera de la población", retrocedían su paso, y volvían a intentarlo en la próxima ronda de movimiento. Con esto se logra que todos los individuos de una población hayan sido determinados por la condición que están evaluando.

Función de fitness

La función de fitness para este tipo de problema tiene que indicar un valor cada vez más cercano a cero cuanto mas cerca estén los valores numéricos de la condición a tomar el valor de verdad opuesto. Así el objetivo final del algoritmo es minimizar dicha función.

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

Modificaciones al programa evaluado.

El método para la generación de casos de prueba elegido es del tipo de 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 utilizamos un método compuesto de dos partes: por un lado, un asistente de ejecución, que dependiendo de algunos símbolos introducidos en el código fuente (inocuos para la ejecución), agrega información sobre los valores de cada variable, y por otro 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.

4. Aspectos de implementación

La solución se implementó en Ruby, un lenguaje de programación interpretado, reflexivo y orientado a objetos, sumamente flexible que permite no solo 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.

Programas

Para poder medir si los resultados de las pruebas del generador, se implementaron algunos programas característicos en el ámbito del testing de datos. Particularmente elegimos:

- 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 ordenamiento de lista.
- Bessel: algoritmo que resuelve las funciones de Bessel J_n e Y_n

Otra medición que debíamos tener en cuenta para verificar que nuestra propuesta con PSO esté representando una mejora, es la comparación con otros métodos de generación de casos de prueba, por lo cual los mismos programas fueron testeados con el método Tabú Search y con una generación totalmente Random.

5. Resultados

Los resultados promedio luego de 100 ejecuciones de cada una de las pruebas nos dan esta tabla de resultados:

Programa	PSO con variación		Random		Tabu Search	
	Cov.	Evals.	Cov.	Evals.	Cov.	Evals.
triangle	**100**	**50,72**	95,75	34,76	73,75	55,58
calday	**99,4**	**512,74**	98,43	197,21	83,04	1491,26
select	**100**	72,56	**100**	**16,55**	98,83	139,13
bessel	**100**	**483,76**	99,03	320,08	96,63	2116,25
quicksort	**100**	2,21	**100**	**2,1**	**100**	7,99

Podemos ver que, atento a la cobertura final de los programas examinados, el método propuesto con PSO resulta superior. Algo a tener en cuenta mientras observamos la cantidad de ejecuciones de cada uno los test es que la generación de individuos para las poblaciones de PSO se realizan tomando el primer individuo que alcanzó la condición y creando variaciones específicas para cada una de sus dimensiones, por eso en programas que se resuelven en muy pocas vueltas de generación, como el select, hay una menor tasa de ejecuciones en caso del random que en el de PSO, aún así, la cobertura sigue superior.

6. Conclusiones

En el artículo se presentó un método para generación de casos de prueba utilizando PSO con adaptaciones particulares al problema.

Además de su implementación se realizaron casos de prueba con diferentes algoritmos y se comparó con otros métodos conocidos de generación de casos.

Quedó implementado un sistema de evaluación y ejecución asistida para programas escritos en Ruby.

Los resultados obtenidos de para cada uno de los programas en los diferentes métodos, nos señalan que la nuestra propuesta es válida, aumentando la cobertura en todos los casos a cambio de incrementar levemente la cantidad de ejecuciones totales, condición que parece razonable teniendo en cuenta que el fin último de este tipo de problemáticas e la maximización de la cobertura.

Bibliografía

[Agu01] J. Aguilar-Ruiz, I. Ramos, J. C. Riquelme, and Miguel Toro. An evolutionary approach to estimating software development projects. Information and Software Technology, 43(14):875–882, 2001.

[Alb05] Enrique Alba and J. Francisco 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, Crete, Greece, September 2005.

[Alb05-1] E. Alba. Parallel Metaheuristics: A New Class of Algorithms. John Wiley & Sons, October 2005.

[Alb06] Enrique Alba, Francisco Chicano, and Stefan Janson. Testeo de software con dos técnicas metaheurísticas. In XI Jornadas de Ingeniería del Software y Bases de Datos, pages 109–118, Sitges,Barcelona, Spain, October 2006.

[Alb07] E. Alba and F. Chicano. Observations in using parallel and sequential evolutionary algorithms for automatic software testing. Computers & Operations Research, 177(11):2380–2401, June 2007.

[Ant04] Giulio Antoniol, Massimiliano Di Penta, and Mark Harman. Search–based techniques for optimizing software project resource allocation. In Genetic and Evolutionary Computation Conference (GECCO 2004), 2004.

[Ant04-1] Giuliano Antoniol, Massimiliano Di Penta, and Mark Harman. A robust search-based approach to project management in the presence of abandonment, rework, error and uncertainty. In IEEE METRICS, pages 172–183. IEEE Computer Society, 2004.

[Bak97] T. Bäck, D. Fogel, and Z. Michalewicz. Handbook of Evolutionary Computation. IOP Publishing and Oxford University Press, New York and Bristol (UK), Feb 1997.

[Bak06] Paul Baker, Mark Harman, Kathleen Steinhofel, and Alexandros Skaliotis. Search based approaches to component selection and prioritization for the next release problem. In ICSM, pages 176–185. IEEE Computer Society, 2006.

[Bar02] Andr´e Baresel, Harmen Sthamer, and Michael Schmidt. Fitness function design to improve evolutionary structural testing. In GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference, pages 1329–1336. Morgan Kaufmann Publishers, 2002.

[Bar03] Andre Baresel and Harmen Sthamer. Evolutionary testing of flag conditions. In Genetic and Evolutionary Computation (GECCO-2003), volume 2724, pages 2442–2454. Springer-Verlag, 2003.

[Bar04] André Baresel, David Wendell Binkley, Mark Harman, and Bogdan Korel. Evolutionary testing in the presence of loop–assigned ?ags: A testability transformation approach. In International Symposium on Software Testing and Analysis (ISSTA 2004), pages 108–118, 2004.

[Bar88] Barry W. Boehm. A spiral model of software development. Computer, 21(5):61–72, May 1988.

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

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

[Blu03] C. Blum and A. Roli. Metaheuristics in Combinatorial Optimization: Overview and Conceptual Comparison. ACM Computing Surveys, 35(3):268-308, 2003.

[Bot03] Leonardo Bottaci. Predicate expression cost functions to guide evolutionary search for test data. In Genetic and Evolutionary Computation – GECCO-2003, volume 2724 of LNCS, pages 2455–2464. Springer-Verlag, 2003.

[Bou06] Salah Bouktif, Houari Sahraoui, and Giuliano Antoniol. Simulated annealing for improving software quality prediction. In GECCO ’06: Proceedings of the 8th annual conference on Genetic and evolutionary computation, pages 1893–1900. ACM Press, 2006.

[Bri02] Lionel C. Briand, Jie Feng, and Yvan Labiche. Using genetic algorithms and coupling measures to devise optimal integration test orders. In SEKE ’02: Proceedings of the 14th international conference on Software engineering and knowledge engineering, pages 43–50. ACM Press, 2002.

[Bri05] Lionel C. Briand, Yvan Labiche, and Marwa Shousha. Stress testing real-time systems with genetic algorithms. In GECCO ’05: Proceedings of the 2005 conference on Genetic and evolutionary computation, pages 1021–1028. ACM Press, 2005.

[Bry05] Ren´ee C. Bryce and Charles J. Colbourn. Constructing interaction test suites with greedy algorithms. In ASE, pages 440–443. ACM, 2005.

[Cha01] Carl K. Chang, Mark J. Christensen, and Tao Zhang. Genetic Algorithms for Project Management. Annals of Software Engineering, 11:107–139, 2001.

[Chi05] E. Alba and J. F. Chicano. Management of software projects with GAs. In Metaheuristics International Conference (MIC-2005), pages 13–18, Viena, Austria, August 2005.

[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

[Coh03] Myra B. Cohen, Peter B. Gibbons, Warwick B. Mugridge, and Charles J. Colbourn. Constructing test suites for interaction testing. In ICSE '03: Proceedings of the 25th International Conference on Software Engineering, pages 38–48. IEEE Computer Society, 2003.

[Coh06] Myra Cohen, Shiu Beng Kooi, and Witawas Srisa-an. Clustering the heap in multi-threaded applications for improved garbage collection. In GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation, pages 1901–1908. ACM Press, 2006.

[Coo99] Keith D. Cooper, Philip J. Schielke, and Devika Subramanian. Optimizing for reduced code space using genetic algorithms. In Proceedings of the ACM Sigplan 1999 Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'99), volume 34.7, pages 1–9. ACM Press, 1999.

[Chi07] José Francisco Chicano García. Metaheurísticas e Ingeniería del Software. PhD 2007.

[Del06] Christian Del Rosso. Reducing internal fragmentation in segregated free lists using genetic algorithms. In WISER '06: Proceedings of the 2006 international workshop on Workshop on interdisciplinary software engineering research, pages 57–60. ACM Press, 2006.

[Des04] Des Greer and G¨unther Ruhe. Software release planning: an evolutionary and iterative approach. Information & Software Technology, 46(4):243–253, 2004.

[Dia05] Eugenia Díaz, Raquel Blanco, and Javier Tuya. Applying tabu and scatter search to automated software test case generation. In Proceedings of the 6th Metaheuristic International Conference, pages 290–297, Vienna, Austria, August 2005.

[Dor92] M. Dorigo. Optimization, Learning and Natural Algorithms. PhD thesis, Dipartamento di Elettronica, Politecnico di Milano, 1992.

[Dov99] D. Doval, S. Mancoridis, and B. S. Mitchell. Automatic clustering of software systems using a genetic algorithm. In Intl. Conference on Software Tools and Engineering Practice, 1999.

[Ebe00] R. Eberhart and Y. Shi. Comparing Inertia Weights and Constriction Factors in Particle Swarm Optimization. In Proceedings of the International Congress on Evolutionary Computation, volume 1, pages 84□88, July 2000.

[Fat03] Deji Fatiregun, Mark Harman, and Robert Hierons. Search based transformations. In Genetic and Evolutionary Computation – GECCO-2003, volume 2724, pages 2511–2512. Springer-Verlag, 2003.

[Fat04] Deji Fatiregun, Mark Harman, and Robert Mark Hierons. Evolving transformation sequences using genetic algorithms. In Fourth IEEE International Workshop on Source Code Analysis and Manipulation, pages 65–74. IEEE Press, 2004.

[Fat05] Deji Fatiregun, Mark Harman, and Rob Hierons. Search-based amorphous slicing. In 12th International Working Conference on Reverse Engineering (WCRE 05), pages 3–12, 2005.

[Feo99] T. Feo and M. Resende. Greedy Randomized Adaptive Search Procedures. Journal of Global Optmization, 6:109-133, 1999.

[Gir05] Moheb R. Girgis. Automatic test data generation for data flow testing using a genetic algorithm. J. UCS, 11(6):898–915, 2005.

[Gle79] G. Myers. The Art of Software Testing. John Wiley and Sons, New York, 1979.

[Glo86] F. Glover. Future Paths for Integer Programming and Links to Artifcial Intelligence. Computers & Operations Research, 13:533-549, 1986.

[Glo02] F. Glover and G. Kochenberger. Handbook of Metaheuristics. Kluwer Academic Publishers, Norwell, MA, 2002.

[Gol06] Nicolas Gold, Mark Harman, Zheng Li, and Kiarash Mahdavi. Allowing overlapping boundaries in source code using a search based approach to concept binding. In International Conference on Software Maintenance (ICSM), pages 310–319. IEEE Computer Society, 2006.

[Gro01] Hans-Gerhard Groß. A prediction system for evolutionary testability applied to dynamic execution time analysis. Information & Software Technology, 43(14):855–862, 2001.

[Gro03] H.-G. Groß and N. Mayer. Search-based execution-time verification in object-oriented and component-based real-time system development. In WORDS, page 113. IEEE Computer Society, 2003.

[Gro05] Concettina Del Grosso, Giuliano Antoniol, Massimiliano Di Penta, Philippe Galinier, and Ettore Merlo. Improving network applications security: a new heuristic to generate stress testing data. In GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation, pages 1037–1043. ACM Press, 2005.

[Har02] Mark Harman, Lin Hu, Robert Mark Hierons, Chris Fox, Sebastian Danicic, Andr´e Baresel, Harmen Sthamer, and Joachim Wegener. Evolutionary testing supported by slicing and transformation. In IEEE International Conference on Software Maintenance, page 285, 2002.

[Har04] Mark Harman, Lin Hu, Robert M. Hierons, Joachim Wegener, Harmen Sthamer, Andr´e Baresel, and Marc Roper. Testability transformation. IEEE Trans. Software Eng, 30(1):3–16, 2004.

[Har07] Mark Harman and Laurence Tratt. Pareto optimal search based refactoring at the design level. In GECCO 2007: Proceedings of the Genetic and Evolutionary Computation Conference, 2007.

[Hie02] Mark Harman, Robert Hierons, and Mark Proctor. A new representation and crossover operator for search-based optimization of software modularization. In GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference, pages 1351–1358. Morgan Kaufmann Publishers, 2002.

[Hie05] Robert M. Hierons, Mark Harman, and Chris Fox. Branch-coverage testability transformation for unstructured programs. Computer Journal, 48(4):421–436, 2005.

[IEEE90] IEEE. IEEE standard glossary of software engineering terminology -description (610.12-1990), 1990.

[Ken95] J. Kennedy and R. Eberhart. Particle Swarm Optimization. In Proceedings of the IEEE International Conference on Neural Networks, volume 4, pages 1942-1948, Perth, Australia, Nov 1995.

[Ken01] J. Kennedy, R. Eberhart, and Y. Shi. Swarm Intelligence. San Francisco:Morgan Kaufmann Publishers, 2001.

[Kho04] Taghi Khoshgoftaar, Yi Liu, and Naeem Seliya. A Multiobjective Module-Order Model for Software Quality Enhancement. IEEE Transactions on Evolutionary Computation, 8(6):593–608, 2004.

[Kir83] S. Kirkpatrick, C. Gelatt, and M. Vecchi. Optimization by Simulated Annealing. Science, 220(4598):671□680, 1983.

[Koe03] Bogdan Korel, S. Chung, and P. Apirukvorapinit. Data dependence analysis in automated test generation. In Proceedings: IASTED International Conference on Software Engineering and Applications, pages 476–481, 2003.

[Lam04] Frank Lammermann, André Baresel, and Joachim Wegener. Evaluating evolutionary testability with software-measurements. In GECCO (2), volume 3103, pages 1350–1362. Springer, 2004.

[Mah03] Kiarash Mahdavi, Mark Harman, and Robert Mark Hierons. A multiple hill climbing approach to software module clustering. In ICSM'03: Proceedings of the International Conference on Software Maintenance, pages 315–324. IEEE Computer Society Press, 2003.

[Man98] S. Mancoridis, B. S. Mitchell, C. Rorres, Y. Chen, and E. R. Gansner. Using automatic clustering to produce high-level system organizations of source code. In Proceedings of the International Workshop on Program Comprehension. IEEE Computer Society Press, 1998.

[Mcm03] Phil McMinn and Mike Holcombe. The state problem for evolutionary testing. In Erik Cantú Paz et al., editor, Proceedings of the Genetic and Evolutionary Computation Conference, volume 2724, pages 2488–2498, Chicano, Illinois, USA, 2003. Springer-Verlag.

[Mcm04] Phil McMinn and Mike Holcombe. Hybridizing evolutionary testing with the chaining approach. In GECCO (2), volume 3103 of LNCS, pages 1363–1374. Springer, 2004.

[Mcm05] Phil McMinn and Mike Holcombe. Evolutionary testing of state-based programs. In GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation, pages 1013–1020. ACM Press, 2005.

[Mcm06] Phil McMinn, Mark Harman, David Binkley, and Paolo Tonella. The species per path approach to search-based test data generation. In International Symposium on Software Testing and Analysis (ISSTA 06), pages 13–24, 2006.

[Mic01] 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.

[Mil76] Webb Miller and David L. Spooner. Automatic generation of floating-point test data. IEEE Trans.Software Eng., 2(3):223–226, 1976.

[Mit02] Brian S. Mitchell, Spiros Mancoridis, and Martin Traverso. Search based reverse engineering. In Proceedings of the 14th international conference on Software engineering and knowledge engineering, pages 431–438. ACM Press, 2002.

[Mla97] N. Mladenovic and P. Hansen. Variable Neighborhood Search. Computers Oper. Res, 24:1097□1100, 1997.

[Mon98] Yannick Monnier, Jean-Pierre Beauvais, and Anne-Marie D´eplanche. A genetic algorithm for scheduling tasks in a real-time distributed system. In EUROMICRO, pages 20708–20714. IEEE Computer Society, 1998.

[Nis98] Andy Nisbet. GAPS: A compiler framework for genetic algorithm (GA) optimised parallelisation. In High-Performance Computing and Networking, International Conference and Exhibition, HPCN Europe 1998, Amsterdam, The Netherlands, April 21-23, 1998, Proceedings, volume LNCS 1401, pages 987–989. Springer, 1998.

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

[Oke06] Mark O'Keffe and Mel O'Cinneide. Search-based software maintenance. In Conference on Software Maintenance and Reengineering (CSMR'06), pages 249–260, 2006.

[Pre02] 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.

[Ref03] Marek Reformat, Xinwei Chai, and James Miller. Experiments in automatic programming for general purposes. In ICTAI, pages 366–373. IEEE Computer Society, 2003.

[Rya00] Conor Ryan. Automatic re-engineering of software using genetic programming. Kluwer, 2000.

[Sag03] 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.

[Sag05] Ram´on Sagarna and Jose A. Lozano. On the performance of estimation of distribution algorithms applied to software testing. Applied Artificial Intelligence, 19(5):457–489, May-June 2005.

[Sag06] Ramón Sagarna and Jos´e A. Lozano. Scatter search in software testing, comparison and collaboration with estimation of distribution algorithms. European Journal of Operational Research, 169(2):392–412, March 2006. (available online).

[Sal06] Salah Bouktif, Giuliano Antoniol, Ettore Merlo, and Markus Neteler. A novel approach to optimize clone refactoring activity. In GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation, pages 1885–1892. ACM Press, 2006.

[Sen05] Olaf Seng, Markus Bauer, Matthias Biehl, and Gert Pache. Search-based improvement of subsystem decompositions. In GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation, pages 1045–1051. ACM Press, 2005.

[Sha07] Yuanyuan Zhang, Mark Harman, and S. Afshin Mansouri. The multi-objective next release problem. In GECCO 2007: Proc. of the Genetic and Evolutionary Computation Conference, 2007

[She06] Alaa F. Sheta. Estimation of the COCOMO model parameters using genetic algorithms for NASA software projects. Journal of Computer Science, 2(2):118–123, 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

[Sim06] C. L. Simons and I. C. Parmee. Single and multi-objective genetic operators in object-oriented conceptual software design. In GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation, pages 1957–1958. ACM Press, 2006.

[Ste03] M. Stephenson, S. Amarasinghe, M. Martin, and U.-M. O'Reilly. Meta optimization: improving compiler heuristics with machine learning. In PLDI '03: Proc. of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, pages 77–90. ACM Press, 2003.

[Stu99] T. Stützle. Local Search Algorithms for Combinatorial Problems Analysis, Algorithms and New Applications. Technical report, DISKI Dissertationen zur Künstliken Intelligenz. Sankt Augustin, Germany, 1999.

[Sut05] Andrew Sutton, Huzefa Kagdi, Jonathan I. Maletic, and L. Gwenn Volkert. Hybridizing evolutionary algorithms and clustering algorithms to find source-code clones. In GECCO '05: Proc. of the 2005 conference on Genetic and evolutionary computation, pages 1079–1080. ACM Press, 2005.

[Swi05] Mark Harman, Stephen Swift, and Kiarash Mahdavi. An empirical study of the robustness of two module clustering fitness functions. In GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation, volume 1, pages 1029–1036. ACM Press, 2005.

[Tra00] Nigel Tracey, John Clark, Keith Mander, and John McDermid. Automated test-data generation for exception conditions. Software Practice and Experience, 30(1):61–79, 2000.

[Tra00-1] N. Tracey. A search-based automated test-data generation framework for safety-critical software. PhD thesis, University of York, 2000.

[Tra02] Nigel Tracey, John Clark, John McDermid, and Keith Mander. A search-based automated test-data generation framework for safety-critical systems. Systems engineering for business process change: new directions, pages 174–213, 2002.

[Tuy04] Javier Tuya, Eugenia Díaz, Raquel Blanco. A Modular Tool for Automated Coverage in Software Testing. In Proceedings of the Eleventh Annual International Workshop on Software Technology and Engineering Practice. IEEE Press. 2004.

[Wap05] Stefan Wappler and Frank Lammermann. Using evolutionary algorithms for the unit testing of object-oriented software. In GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation, pages 1053–1060. ACM Press, 2005.

[Wap06] Stefan Wappler and Joachim Wegener. Evolutionary unit testing of object-oriented software using strongly-typed genetic programming. In GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation, pages 1925–1932. ACM Press, 2006.

[Wil98] Kenneth Peter Williams. Evolutionary Algorithms for Automatic Parallelization. PhD thesis, University of Reading, UK, 1998.

[Win70] Winston W. Royce. Managing the development of large software systems. In Proceedings of the IEEE Western Electronic Show and Convention, pages 328–338, LA, USA, 1970. IEEE Press.

[Xan92] S. Xanthakis, C. Ellis, C. Skourlas, A. Le Gall, S. Katsikas, and K. Karapoulios. Application of genetic algorithms to software testing (Application des algorithmes g´en´etiques au test des logiciels). In 5th International Conference on Software Engineering and its Applications, pages 625–636, 1992.