Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional

Unidad Zacatenco

Departamento de Computación

Depuración Automática para Programas en Lenguaje C usando Algoritmos Genéticos.

Tesis que presenta

Pedro Eduardo Torres Jiménez

para obtener el Grado de

Maestro en Ciencias en Computación

Director de la Tesis

Dr. Pedro Mejía Álvarez

México, D.F. **Diciembre, 2014**

Dedicatoria

- A mis padres María Luisa y Pedro Francisco por su amor y apoyo incondicional.
- A mis abuelos Cristina Gallardo y Marcelo Jiménez por su gran amor y por todas las lecciones de vida que aprendí a su lado.
- A Nicolás mi hijo por inspirarme a ser cada día una mejor persona.
- A todos mis compañeros de clase, por compartir gratas experiencias juntos.

Agradecimientos

- Al *CINVESTAV por brindarme la oportunidad de formar parte de este gran centro de estudios.*
- *Al Consejo de Ciencia y Tecnología* CONACYT por el apoyo económico brindado y el impulso científico en México.
- Al Dr. Pedro Mejía Álvarez, por su conocimiento, entusiasmo brindado y por aceptarme en su tema de tesis.
- A todos los profesores del departamento de computación que ayudaron en mi formación académica.

Resumen

Actualmente el proceso de desarrollo de software ha adquirido madurez. Una de las características más importantes del software es sin duda la confiabilidad. La experiencia indica que es fácil introducir errores por parte de los desarrolladores de manera no intencional. Por esta razón se han desarrollado distintas estrategias para localizar defectos.

Nuestros servicios como energía, comunicaciones, transporte, etc. dependen de sistemas informáticos muy grandes. Por esta razón es importante detectar y eliminar la mayor cantidad posible de errores.

Los trabajos actuales de detección de defectos se basan en análisis estático, dinámico e inspecciones de software. Estos métodos son muy útiles, no obstante requieren mucho tiempo y son actividades tediosas.

Cuando el Software manifiesta un defecto, éste se encuentra generalmente dentro de un conjunto de datos grande. La depuración en este contexto es difícil y requiere mucho tiempo localizar la causa del defecto.

Este trabajo aprovecha las ventajas de las pruebas automáticas para automatizar el proceso de depuración. Se pretende ayudar al programador a simplificar código fuente y entradas de usuario de manera automática. Como resultado se entrega un subconjunto útil que preserve las características de un defecto y pueda ser corregido fácimente.

Se presenta una biblioteca de depuración automática de programas escritos en lenguaje C que permite identificar, simplificar y aislar defectos.

La biblioteca contiene los algoritmos clásicos de la literatura sobre depuración automática. Específicamente las técnicas de depuración delta, además se introduce un método de simplificación de casos de prueba basado en perfilaje y optimización usando un algoritmo genético, el cual entrega buenos resultados sin repetir pruebas.

Palabras clave: Simplificación de casos de prueba, aislamiento de defectos, depuración automática, depuración delta, algoritmo genético.

Abstract

Currently the software development process has aquired maturity. One of the most important features of the software is the reliability. Experience suggests that it isnt difficult to introduce errors by developers. For this reason different strategies have been developed to simplify and isolate failures.

Our services and infrastrucure like energy, communications, transportation, etc. Dependens of big computer systems. For this reason its important detect and remove as many bugs as possible.

Recent works are based on static, dynamic analysis and Software inspectios. These methods are useful but are expensive in terms of time and practice.

Debugging process are very difficult when the failures are in large sets of data. This is the most common scenario.

This work takes advantage of automated testing to automate the debugging process. This work help the developer to simplify user input and source code automatically. As a result we gain an useful subset which preserves the minimum characteristics of a defect and allow easily prevent failures.

A library of automatic debugging we present for programs written in C language. Allowing identify, simplify and isolate the defects on failure inducing inputs.

The library include the literature algorithms on automated debugging such delta debugging algoritmhs and a novel method that uses

profiling technique and genetic algorithm in the input data to avoids repeated tests.

Keywords: Simplification of test cases, failure isolation, automated debugging, delta debugging, genetic algorithm.

Índice general

Índice de figuras

Introducción

El Software es complejo, la mayoría de las veces no está libre de defectos y gran parte del tiempo que se tiene disponible para su creación se emplea para descubrir y corregir errores. El desarrollo de Software implica diversas etapas, entre las que se encuentran el análisis, diseño, codificación, pruebas y mantenimiento. La etapa de pruebas es imprescindible porque en ésta se mide la calidad y se manifiestan los defectos.

El proceso de detección y correción de errores es de gran importancia, cuando se omite repercute en la calidad y ocasiona pérdidas económicas, retrasos de entrega y problemas operativos. Al momento en que un programa falla, requiere atención inmediata. Lo cual implica detener su ejecución y averiguar la causa de la falla.

Nosotros como usuarios de Software, probablemente ya hemos aprendido a vivir con defectos que ocurren día a día, y en pocas ocasiones hemos aprendido a desarrollar Software libre de errores. Como desarrolladores de Software, se sabe que los defectos no se introducen por si solos en nuestros programas. Por el contrario, estos pueden introducirse por una mala interpretación de los requerimientos, un mal diseño o algún error introducido en la programación.

Nuestra capacidad para construir sistemas grandes y complejos ha mejorado drásticamente en los últimos años. Nuestros servicios e infraestructuras internacionales como energía, comunicaciones, transporte, etc. dependen de sistemas informáticos muy grandes, complejos y fiables.

La propiedad más importante de un sistema de Software es su confiabilidad y existen varias razones por las que ésta es la propiedad más importante de los sistemas:

- Los sistemas que no son fiables o que son inseguros o desprotegidos, son rechazados a menudo por sus usuarios. Si los usuarios no confían en un sistema, se negarán a utilizarlo.
- Los costos de los fallos de un sistema pueden ser enormes.
- Los sistemas no confiables pueden provocar pérdida de información.

Por estas razones los defectos deben corregirse lo antes posible y así evitar su propagación.

1.1. Antecedentes

En el proceso de desarrollo de Software se sabe que la mayor parte del tiempo y esfuerzo se usan para la detección y corrección de errores [1] y se sabe también que cerca del 50 % del tiempo en un proyecto se usa para depurar.

La figura 1.1 muestra que la etapa de pruebas es la que más tiempo requiere y dentro de las pruebas, la corrección de errores.

Además, se estima que el costo de los errores de Software es de 59,5 billones de dolares anualmente [2], y que estos errores influyen en el presupuesto, en las actividades empresariales y en el peor caso con vidas humanas.

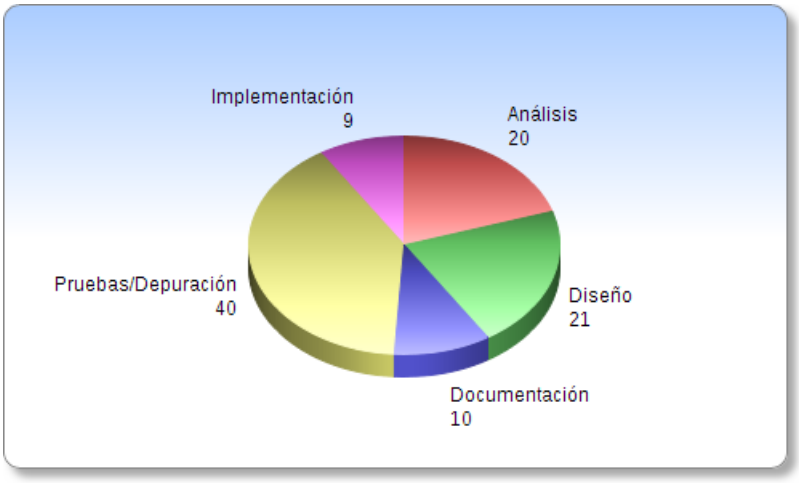

Figura 1.1: Distribución en tiempo de las etapas del Software, las cifras representan el porcentaje del tiempo de desarrollo [1].

Las pruebas manuales y las inspecciones de Software son las técnicas más usadas para la detección de errores, sin embargo estas son costosas.

En las pruebas se deben reunir ciertos requisitos como el conjunto de pruebas (o casos de prueba), el personal encargado de dirigirlas y el tiempo destinado para esta actividad. Los casos de prueba deben prepararse con la mayor amplitud posible, esto es, deben cubrir varios aspectos para probar el Software, los cuales son: la seguridad, la eficiencia, y los requerimientos.

Los encargados de las pruebas deben tener conocimiento del sistema, de manera que las pruebas sean planeadas para medir las métricas mencionadas. El tiempo destinado para esta actividad es proporcional al tamaño del sistema en desarrollo. Por lo tanto, las pruebas revelan defectos sólo si su preparación es adecuada.

Gracias a los avances en depuración automática es mas fácil detectar defectos en el Software [3]. Sin embargo es mejor aislar los defectos para su análisis y su reparación. En otras palabras, es conveniente tener un conjunto de casos de prueba reducido con el fin de generalizar un defecto y agilizar el proceso de detección y corrección de errores. Usando depuración automática se pueden encontrar defectos rápidamente con casos de prueba adecuados.

De manera similar, las inspecciones se usan para el mismo propósito, pertenecen al área del análisis estático, en donde no se tiene que ejecutar el programa. Esto implica algunas desventajas, como el tiempo utilizado, el costo y la intervención. Por esta razón la depuración automática tiene algunas ventajas respecto a las inspecciones, sin embargo las inspecciones sirven para detectar errores que no son posibles detectar con otras técnicas.

La detección de defectos usando depuracion automática puede ser aplicada a todas las circunstancias, en todo aquello que afecte la ejecucion de un programa y puede ser automatizada a diferencia de las inspecciones formales.

La automatización de estos procesos ha tenido gran auge en los últimos años [3] permitiendo a los programadores desarrollar Software de mejor calidad, logrando buenos resultados y reduciendo el costo y tiempo dedicados a esta tarea.

La etapa de pruebas de Software es crítica, además es la fase más costosa en el ciclo de desarrollo de Software. Las organizaciones desean probar a fondo el Software, sin embargo las pruebas exhaustivas son imprácticas debido a las limitaciones de recursos. Se conoce como caso de prueba al conjunto de entradas recibidas y salidas generadas por un proceso, programa o función. Un gran número de conjuntos de casos de prueba pueden ser generados utilizando herramientas automatizadas y el verdadero desafío es la selección de casos de prueba cruciales que validen un sistema.

Al conjunto de algoritmos que utiliza depuración automática para simplificar casos de prueba se le conoce como *depuración delta* [4]. Estos algoritmos usan pruebas automáticas y particiones ordenadas para reducir el conjunto de datos de entrada. El uso de particiones ordenadas ayuda a probar gradualmente el Software considerando las combinaciones posibles de datos. Sin embargo esto implica repetir pruebas. Por esta razón se incluye un algoritmo genético que usa como granularidad mínima las sentencias en código fuente o datos en el caso de entradas de usuario.

Posteriormente surgió la *depuración delta jerárquica* (hdd)[5]. Este trabajo

resulta más eficiente y preciso que la *depuración delta*.

1.2. Motivación

Una vez que se ha reproducido un problema, se debe simplificar, esto es, debemos encontrar qué circunstancias no son relevantes para el problema y omitirlas. Este proceso da como resultado un caso de prueba que contiene solo las circunstancias relevantes. En el mejor de los casos, un caso de prueba simplificado apunta directamente a la raiz del defecto.

Uno puede formularse preguntas como: ¿El problema depende realmente de la cantidad total del código fuente?, ¿Es necesario reproducir todos los pasos para manifestar el defecto?, ¿Se necesita inspeccionar toda una traza de ejecución?, etc.

Cada circunstancia debe ser analizada, se debe comprobar si es relevante o no para el problema, si no es así, se remueve del caso de prueba bajo estudio, en otro caso es necesaria y no se omite. Una circunstancia se refiere a cualquier aspecto que influye en el comportamiento del caso de prueba.

¿Cómo puede uno saber si una circunstancia es relevante? Esto se logra con la experimentación. i.e., se omite cierta circunstancia y se prueba nuevamente con la intención de reproducir el problema, si el problema no ocurre nuevamente, entonces tal circunstancia es relevante, por otro lado si el problema sigue ocurriendo, entonces la circunstancia es irrelevante. El propósito de simplificar un caso de prueba es hacerlo tan simple como sea posible que realizando cualquier cambio en él, altere el comportamiento del error.

Un reporte de bug o defecto encontrado debe ser lo más específico posible, de tal manera que se pueda reproducir el contexto en el que un programa falla, es decir, debe ser un proceso totalmente determinista.

Un caso de prueba debe ser lo más simple posible porque implica un contexto más general.

Formalmente:

Sea $c_{✗}$ un caso de prueba que no pase una función de prueba denotada por test, i.e., $c_{✗}$ manifieste un defecto f.

Un caso de prueba $c \subseteq c_{✗}$ es mínimo si no existe un subconjunto más pequeño de c que siga manifestando f

$c_{✓}$ indica que no manifiesta f en una ejecución específica bajo la función de prueba $test$.

Un caso de prueba $c \subseteq c_{✗}$ es llamado mínimo global de $c_{✗}$ si

$\forall c' \subseteq c_{✗}.\ |c'| \leq |c| \rightarrow test(c') \neq ✗$

En la práctica se desea encontrar el mínimo global, sin embargo esto es imprácticco porque para determinarlo se requieren probar $2^{|c_{✗}|}$ subcojuntos de $c_{✗}$ lo cual tiene complejidad exponencial.

Llamemos a c un caso de prueba mínimo si ninguno de sus subconjuntos causan que c falle, lo que queremos encontrar es un caso de prueba cuyos elementos sean todos significativos y necesarios para reproducir f.

En este trabajo se pretende resolver los siguientes dos problemas:

La simplificación de casos de prueba: encontrar un $c'_{✗}$ tal que $c'_{✗} \subseteq c_{✗}$, $test(c'_{✗}) = ✗$ y $c'_{✗}$ sea lo más pequeño posible.

El aislamiento de defectos: encontrar dos cunjuntos $c'_{✗}$ y $c'_{✓}$ tales que $c_{✓} \subseteq c'_{✓} \subset c'_{✗} \subseteq c_{✗}$ y la diferencia $\Delta = c'_{✗} - c'_{✓}$ sea mínima.

1.3. Contribuciones

- Se diseñó e implementó una biblioteca de depuración automática para programas escritos en lenguaje C.

En la solución de esta contribución se abordaron las siguientes contribuciones específicas:

Contribuciones Específicas:

- Implementación del algoritmo *ddmin* para reducir código fuente. El algoritmo original se implementó en la biblioteca.
- Implementación del algoritmo *ddmin* para reducir entradas de usuario de programas.
- Implementación del algoritmo *hdd* para reducir entradas de usuario de programas.
- Implementación del algoritmo *dd* para aislar defectos.
- Implementación de un algoritmo genético para reducir entradas y código que mitigue los problemas de ddmin.

- **Ventajas**. La biblioteca presenta de manera sencilla los algoritmos existentes del área, además de uno propuesto. Son de fácil configuración y no requiere intervención excesiva. Los resultados analizados indican que la detección de defectos es más sencilla.
- **Desventajas**. Pueden resultar falsos positivos por la naturaleza de los errores y los métodos de comprobación. Los cuales se detallan en el capítulo 3 y 4. Configurar una prueba depende mucho de la organización (estructura de directorios y dependencias de un proyecto).

1.4. Organización de la Tesis

En este capítulo se muestran: la motivación, objetivos y el planteamiento del problema.

En el capítulo 2 se introducen los conceptos básicos utilizados en este trabajo, luego se describe el proceso de pruebas de Software y de depuración, la importancia y limitantes de la depuración automática. Al final se muestran las herramientas existentes que intentan resolver los mismos problemas, estas estrategias combinan en la mayoría de las veces análisis estático y dinámico. Se discuten sus ventajas y desventajas.

En el capítulo 3 se muestran los algoritmos básicos sobre depuración automática, sus propiedades y su análisis.

El capítulo 4 contiene la teoría y aplicaciones de los algoritmos del capítulo 3, además una propuesta a los mismos problemas usando una metaheurística, se analizan ejemplos, propiedades y análisis.

El capítulo 5 muestra los resultados de las técnicas utilizadas a herramientas útiles de GNU Linux, y programas especiales para la detección de errores. Se realiza un análisis comparativo usando diversas métricas como el tiempo de ejecución, minimalidad y número de pruebas.

Finalmente en el capítulo 6 se presentan las conclusiones y el trabajo a futuro.

Capítulo 2

Depuración de Software

En este capítulo se introducen los conceptos de pruebas de Software necesarios para entender este trabajo. Se detallan los procesos utilizados en las pruebas de Software, sus características, sus ventajas y desventajas. Finalmente, se describe el proceso de depuración y sus limitaciones, asi como el trabajo relacionado del área y las herramientas desarrolladas.

En Ingeniería de Software se conocen términos técnicos referentes a las pruebas, a las etapas de desarrollo de Software y a la depuración. A continuación se describen los términos que se usan en este trabajo de tesis.

2.1. Proceso de Desarrollo de Software

Es el conjunto de actividades relacionadas a la producción de algún producto de Software, desde que surge como una necesidad hasta su aceptación. Específicamente estas actividades son al menos:

Especificación: se definen los requerimientos, se analizan y se determinan las restricciones y el ambiente operativo.

Diseño: es la etapa dedicada al desarrollo de modelos gráficos que representan al sistema y están basados en la especificación.

Codificación: se contruye el Software de acuerdo al diseño.

Validación: se evalúa el Software y se determina si está construido de acuerdo a las especificaciones.

Evolución y Mantenimiento: es la etapa donde el Software se modifica cuando se detectan errores o cuando se requiere adecuarlo a un nuevo ambiente de trabajo o a nuevas necesidades.

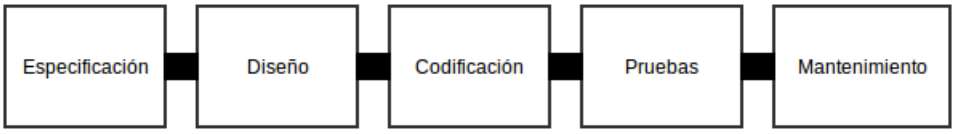

Figura 2.1: Etapas del proceso de desarrollo de Software.

La figura 2.1 muestra las diferentes etapas del proceso de desarrollo de Software.

Existen diversas metodologías de desarrollo de Software que ejecutan cada una de estas actividades de forma distinta.

2.2. Pruebas de Software

Es el proceso de evaluar un sistema o sus componentes con la intención de verificar si satisface los requerimientos especificados o no.

De acuerdo al estándar ANSI/IEEE 1059, las pruebas se definen como el proceso de analizar un artefacto de Software para detectar las diferencias que existen entre su ejecución correcta y la ejecución real.

Cuando se prueba una aplicación se intenta medir su calidad o confiabilidad y en consecuencia, la mejora en la confiabilidad del Software significa encontrar y remover errores.

Debido a que es difícil producir Software libre de errores, es necesario ejecutar pruebas lo suficientemente exhaustivas que permitan descubrir el origen de los errores y su posible reparación.

En esta etapa del desarrollo de Software se desarrolla la depuración.

Caso de prueba

Los casos de prueba son un conjunto de condiciones, variables o valores (información en general) bajo los cuales se determina si un requisito de una aplicación es correcto. Un caso de prueba contiene conjuntos de entradas que se proporcionan al sistema para que este nos de una salida. Para cada conjunto de entradas, la salida encontrada es comparada contra una salida esperada y una correcta (proporcionada por un oráculo) para verificar si el sistema contiene errores.

Una vez concluida la ejecución de un caso de prueba, es necesario detectar cuales de los conjuntos de entrada producen errores, para luego determinar su origen y su posible remoción.

Pueden diseñarse muchos casos de prueba para encontrar distintos tipos de errores o fuentes de ejecución errónea.

En la figura 2.2 se abstrae el concepto de caso de prueba. Un caso de prueba consta de una entrada, un proceso (programa o función), una salida generada y una salida conocida.

Generalmente, se realizan diversos casos de prueba para determinar si un componente es completamente satisfactorio. Con el propósito de comprobar que todos los componentes de una aplicación son revisados, debe haber al menos un caso de prueba para cada componente. Algunas metodologías recomiendan crear por

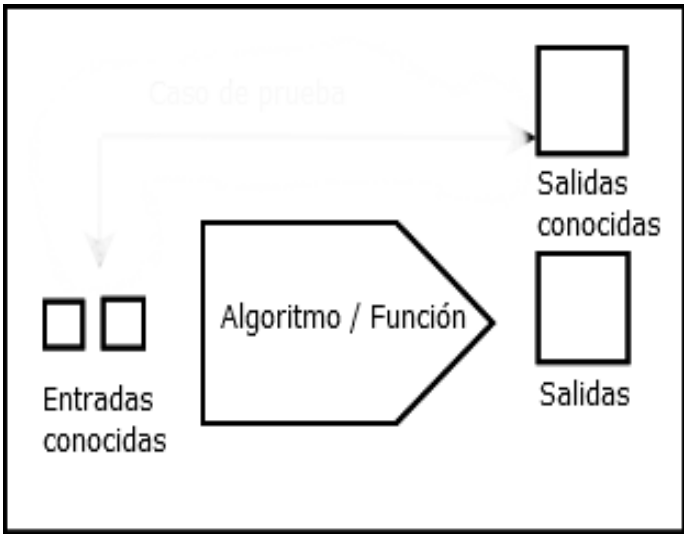

Figura 2.2: Caso de prueba

lo menos dos casos de prueba para cada componente. Uno de ellos debe realizar la prueba positiva de los requisitos y el otro debe realizar una prueba negativa.

Los casos de prueba incluyen una descripción de la funcionalidad que se desea probar y sus características.

Suite de pruebas

Es una colección de casos de prueba que se usa para analizar algún comportamiento en especial del Software o componente. Por lo general, contiene información adicional detallada acerca de los objetivos que las pruebas persiguen.

Oráculo

Es un mecanismo usado para determinar si una prueba ha sido exitosa o ha fallado. Se usa para comparar salidas generadas por un caso de prueba de un programa o sistema contra las salidas esperadas. En el caso de que las salidas reales sean iguales a las esperadas, se dice que esa prueba es exitosa. En caso contrario, se puede decir que para estas entradas el sistema contiene errores. Estos mecanismos se preparan de manera independiente, y pueden ser ejecutados

antes de las pruebas o durante la ejecución de las pruebas.

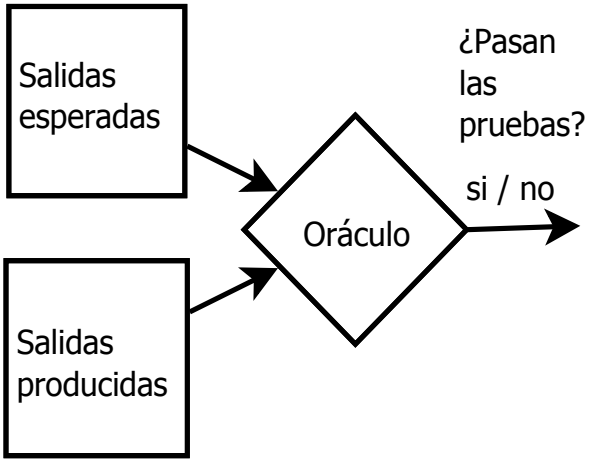

Figura 2.3: Oráculo

En la figura 2.3 se abstrae el concepto de un oráculo. Un oráculo es un comparador entre una salida esperada y una salida generada.

Un oráculo completo debe tener los siguientes componentes:

- Un *generador*, para proveer resultados predecidos o esperados para cada prueba.
- Un *comparador*, para comparar los resultados obtenidos contra los esperados.
- Un *evaluador*, para determinar si la prueba es correcta o no.

Depuración

El proceso de pruebas consiste en descubrir la existencia de un error en el Software. Por otro lado, el proceso de depuración sigue después de las pruebas y permite descubrir y corregir errores.

Cada vez que se descubre un error, la depuración permite seguir paso a paso la ejecución del programa para:

- Descubrir el lugar donde se produce el error.
- Descubrir las causas que llevaron a la manifestacion del error. Esto involucra descubrir los datos, los procedimientos o los objetos involucrados en la producción del error.
- Modificar el programa para corregir la parte del Software que produce los errores.

2.2.1. Análisis dinámico de Software

A continuación se enlistan algunos términos asociados al análisis dinámico o en tiempo de ejecución de programas.

Grafo de llamadas

Es un grafo dirigido donde los vértices representan las funciones o métodos de un programa y los arcos las llamadas a tales entidades; este grafo representa la historia que recorre secuencialmente el código, pero no necesariamente una ejecución. La pila de llamadas si almacena la historia de las funciones ejecutadas.

Árbol de Sintaxis Abstracta

Es una representación de la sitaxis de alguna estructura de código fuente bien formada (de acuerdo a su sintaxis) de un lenguaje de programación. Sirve para manejar la información semántica de un código. La forma más eficiente de manejar la información proveniente de un lenguaje de programación es usando esta estructura de datos.

Traza de ejecución

Se refiere a las sentencias en un lenguaje de alto nivel o instrucciones en un lenguaje de bajo nivel que se han ejecutado hasta el momento donde se detenga la ejecución de un programa, ya sea por un error o intencionalmente mediante un depurador. Se puede usar distinta granularidad, e.g., funciones en vez de sentencias.

Perfilador

En inglés, *profiler* es una herramienta de análisis dinámico que genera trazas de ejecución, ayudando a enteder qué pasos se han ejecutado en un programa; además guarda un historial de las trazas para cada nueva ejecución y permite medir los tiempos por cada llamada a función. Resulta muy útil para localizar las partes que más se tardan en ejecutar y permite inferir qué partes del código se pueden optimizar.

2.2.2. Clasificación

En esta sección, se describen los tipos de pruebas de acuerdo a diversos criterios.

La figura 2.4 muestra la clasificación de pruebas.

Las pruebas se clasifican en *manuales* o *automáticas* de acuerdo a la forma en que se ejecutan.

Las **pruebas manuales** se construyen por algún programador o probador que toma el rol de usuario final. La intención es encontrar algún defecto o comportamiento inesperado, haciendo uso del sistema construido. En este tipo de pruebas se usan las pruebas de unidad, de integración y de aceptación, las cuales se detallan más adelante.

Se crea un plan, el cual incluye casos de prueba o escenarios para asegurar la completitud del dominio de entradas que puedan repercutir en la calidad del Software.

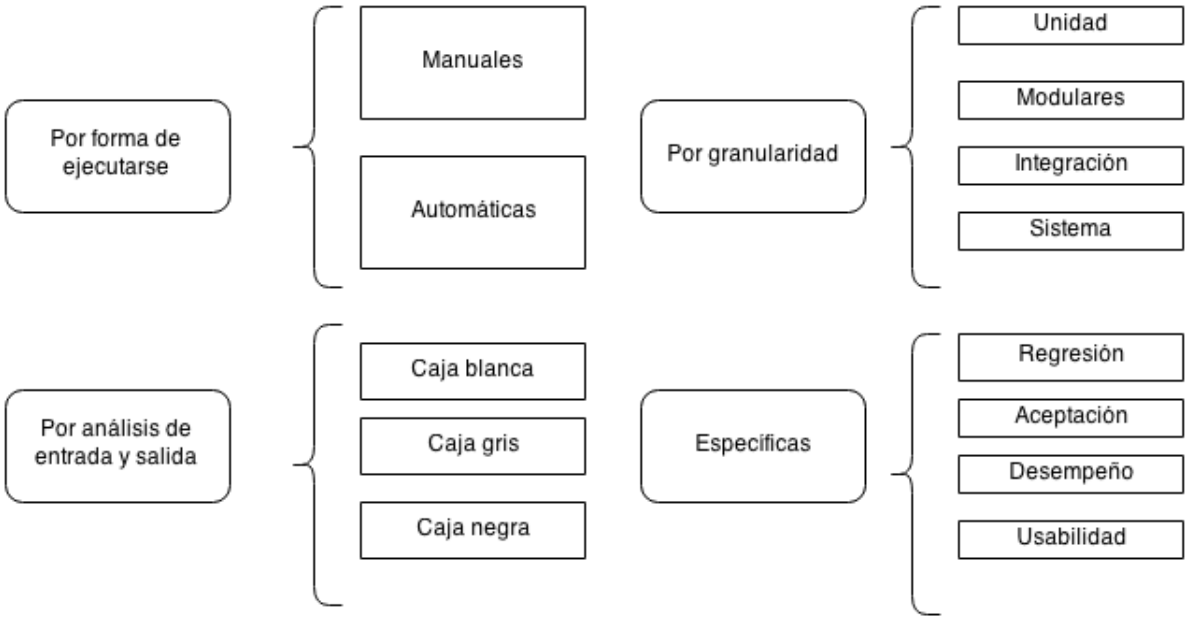

Figura 2.4: Clasificación de pruebas

En las **pruebas automáticas**, los encargados de las pruebas escriben *scripts* y usan Software especial. El proceso involucra la automatización de las pruebas manuales en las cuales los escenarios se reejecutan de manera rápida y repetitiva.

Las pruebas automáticas se usan principalmente para realizar pruebas de estrés, desempeño y regresión. Estas pruebas ayudan a incrementar la cobertura de las pruebas.

En las pruebas no es siempre posible automatizar todas las actividades del proceso, sin embargo cualquier escenario donde exista una gran cantidad de usuarios que puedan acceder simultáneamente o acciones deterministas como acceso de usuarios a una aplicación, altas, bajas, etc se pueden probar eficientemente por este tipo de pruebas.

La automatización es muy demandada en proyectos críticos y largos en donde se requiere probar requerimientos que no cambian.

En la literatura se mencionan tres tipos de pruebas, los cuales son pruebas de

caja negra, de caja blanca y de caja gris.

Las **pruebas de caja negra** abstraen el proceso interno de un componente o módulo de manera que solo nos enfoquemos en las entradas y salidas obtenidas. El probador no tiene acceso al código fuente de la aplicación y solo interactúa como usuario final.

Las ventajas en este enfoque de pruebas son:

- No es necesario el acceso al código fuente.
- Eficientes y útiles para largos segmentos de código.
- Separa claramente la perspectiva del usuario y del desarrollador.
- No se requiere conocimiento técnico de la aplicación en cuanto a su construcción.

Las desventajas:

- Cobertura limitada por el número de escenarios probados.
- Pruebas ineficientes debido al conocimiento limitado de la aplicación.
- Casos de prueba difíciles de diseñar.

Por otro lado, en las **pruebas de caja blanca** si importa la lógica interna y estructura de los módulos y del código fuente. En este caso, los probadores necesitan tener conocimiento del Software para poder revisar y encontrar secciones de código que se comporten de forma erronea.

Algunas ventajas son:

- Ayudan a optimizar el código.

- Pueden ayudar a remover segmentos de código innecesarios o que causen defectos escondidos.
- Maximizan la cobertura.

Las desventajas:

- Por la necesidad de probadores que conozcan la estructura interna, los costos se incrementan.
- Debido a la explosión combinatoria de estados de ejecución, muchas rutas de ejecución son dificiles de probar.
- Se requieren herramientas para automatizar este tipo de pruebas.

Las **pruebas de caja gris** se desempeñan con poco conocimiento de la estructura interna de una aplicación. Los probadores tienen acceso a documentos de diseño y bases de datos, con el fin de ser capaces de preparar mejor los casos de prueba y los escenarios.

Sus ventajas son:

- Ofrecen beneficios tanto de pruebas de caja begra y blanca.
- Debido a la información limitada, es posible crear buenos escenarios para encontrar fallos.
- Las pruebas están completas desde el punto de vista del usuario final y no del desarrollador.

Sus desventajas son:

- La cobertura es limitada.

- Solo son complementarias y no deben usarse como único método de prueba.
- No es ideal para pruebas unitarias (funciones, métodos y algoritmos).

También las pruebas pueden clasificarse por niveles y granularidad.

Los niveles de pruebas incluyen diferentes metodologías que pueden ser usadas mientras el Software se desarrolla.

Dentro de los niveles de pruebas se tienen las *pruebas funcionales* y las *no funcionales*.

Dentro de las **pruebas funcionales** se tienen las pruebas de unidad, de integración, de sistema, de regresión y de aceptación. Dentro de las **no funcionales** se tienen las pruebas de desempeño, usabilidad, seguridad, portabilidad, etc.

Las **pruebas de unidad** se desarrollan para probar métodos o funciones particulares, i.e., de granularidad más pequeña en un sistema complejo. El objetivo es aislar cada parte de una aplicación para asegurarse de su funcionalidad (de acuerdo a los requerimientos) y poder probar a mayor escalar posteriormente.

Las **pruebas de integración** combinan partes de una aplicación de mayor granularidad que las pruebas de unidad, e.g., estas pueden ser pruebas que contengan a su vez una serie de funciones o módulos probados previamente, con el objetivo de verificar si un subsistema mayor funciona adecuadamente.

Las **pruebas de sistema** se refieren a probar en mayor escala, i.e., el sistema como un "todo". Una vez que los componentes han sido integrados y probados se debe proceder con esta técnica.

Las **pruebas de regresión** sirven para verificar si nuevas versiones del código fuente funcionan adecuadamente. Estas pruebas consisten en probar los casos de pruebas previos reiteradamente para cada versión nueva del Software. Estas pruebas son importantes debido a que minimizan problemas para nuevas versiones;

prueban los nuevos cambios, mitigan los riesgos, se incrementa la cobertura y son rápidas.

Las **pruebas de aceptación** sirven para asegurarse que un sistema o aplicación cumple con los requerimientos especificados. Sirven también para mejorar detalles desde estéticos hasta operativos. De acuerdo a las características más relevantes o importantes determinadas por los usuarios estas pruebas se pueden preparar de distintas maneras.

Por último, se detallan las *pruebas no funcionales*.

Las **pruebas de desempeño** se usan para medir la calidad en características como: retardos en una red, velocidad de procesamiento, transacciones de bases de datos o acceso a memoria, balanceo de carga entre servidores, etc. Ayudan a mejorar la velocidad, capacidad, estabilidad y escalabilidad de un sistema.

Las **pruebas de usabilidad** son utilizadas para verificar si las interfaces gráficas de usuario y todas las actividades relacionadas con la interacción del sistema pueden ser usadas fácilmente por sus usuarios. Las pruebas de seguridad son utilizadas para mejorar la integridad, confidencialidad, autenticación etc, elaborando ataques simples y observando vulnerabilidades en el sistema. Mientras que las pruebas de portabilidad se usan para la migración de un sistema a otra plataforma.

En este trabajo se usa el enfoque de pruebas automáticas, de caja negra y de unidad. Las pruebas automáticas dirigen a los algoritmos de búsqueda, es decir, son el criterio para la elección de subconjuntos. Las pruebas de caja negra se usan al momento de leer los mensajes que el sistema operativo manda a la salida estándar de error y pruebas de unidad se usan por la jerarquía del programa iniciando siempre en la función *main*.

2.3. El Proceso de depuración

A grandes rasgos, el proceso de depuración implica:

1. Averiguar por qué el Software se comporta de forma errónea inesperadamente.
2. Modificar el Software para componer los errores encontrados.
3. Evitar modificar cualquier otro artefacto que no tenga que ver con los defectos, o evitar la propagación de los defectos cuando se realizen las modificaciones.
4. Mantener o mejorar la calidad (confiabilidad, arquitectura, cobertura de pruebas, desempeño, etc) del código.
5. Asegurar que el mismo problema no vuelva a ocurrir de nuevo.

La mayoría de los desarrolladores usualmente omiten el proceso de depuración o lo realizan de forma limitada en algunas secciones del código o cuando lo realizan solo depuran los componentes y no el sistema de forma global.

El proceso de depuración consiste de cuatro pasos:

1. **Reproducción**: encontrar una manera confiable y conveniente de reproducir el problema bajo estudio.
2. **Diagnóstico**: consiste en construir una hipótesis, probarla y desarrollar experimentos hasta que se asegure la causa de un defecto.
3. **Correción**: diseño e implementación de cambios que arreglen el problema, manteniendo o mejorando la calidad.
4. **Reflexión**: aprendizaje del fallo, saber qué fue exactamente lo que se hizo mal, y por supuesto evitar el mismo error en un futuro.

La figura 2.5 presenta las etapas y su seguimiento en el proceso de depuración.

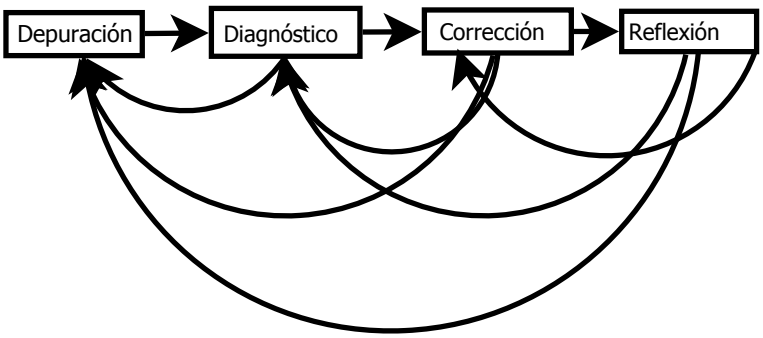

Figura 2.5: Etapas del proceso de depuración.[6]

Muchas veces se presentan varios errores a la vez y se desea corregirlos en paralelo. Sin embargo, esta costumbre no es adecuada puesto que puede requerir mucho más tiempo del estimado [6]. Mientras más cuidado se tenga, mayor es la probabilidad de encontrar y entender los defectos, usando los experimentos basados en la hipótesis.

La mayoría de los defectos son causados por simples descuidos. De vez en cuando, nos enfrentamos con algo muy sutil, sin embargo las cosas simples no deben pasarse por alto.

La *reproducción* es importante porque si un problema no puede repetirse, es muy difícil entender qué origina el problema y, por ende corregirlo. El proceso empírico depende de nuestra habilidad y experiencia de entender el Software bajo la presencia de cierto error. Si no se logra hacer que el Software se comporte anormalmente, no se puede comenzar la corrección aún teniendo la hipótesis correcta.

El *diagnóstico* es un elemento clave en la depuración. Aquí es donde se inicia la correción y se llega a la comprensión de la raíz del comportamiento anómalo que se observa.

El proceso de diagnóstico es un proceso intelectual y muy difícilmente computable.

El método científico puede trabajar de dos maneras distintas aquí. En un caso, se comienza con una hipótesis y se intentan crear experimentos. Los resultados de los experimentos obtenidos nos ayudan a aceptar o refutar la idea. En otro caso, se comienza con una observación que no se ajuste a nuestra teoría y, como resultado, esta teoría se va modificando con el tiempo.

Por esa razón, lo primero que hay que hacer es observar distintas ejecuciones de la aplicación y crear una hipótesis basada en nuestra intuición, experiencia y razonamiento. Posteriormente se procede a diseñar algún experimento. Si éste no desaprueba la hipótesis, se piensa en una nueva hipótesis y se repite este proceso. Si con un experimento nuestra hipótesis es válida, se construyen más experimentos hasta estar seguros que esa es la razón del problema a corregir.

La figura 2.6 muestra un diagrama de flujo sobre la forma en que se debe depurar.

Una de las reglas básicas de la construcción de experimentos es que se debe hacer un solo cambio a la vez. Si se realiza un solo cambio en el código en alguna decisión de diseño o en un caso de prueba, se pueden observar cero o más errores y entonces se puede estar seguro de que al menos ese cambio puede tener algo que ver en la aparición del error. En cambio, si se realizan varios cambios a la vez puede ser difícil estar seguros de este razonamiento.

Una técnica que es bastante útil es la de divide y vencerás. Específicamente la búsqueda binaria es útil porque permite aislar una amplia variedad de defectos. Este tipo de búsqueda es la base de los algortimos que se verán en este trabajo de tesis.

Otra forma de trabajar sobre la depuración es obviamente usar depuradores. Estos varían dramáticamente en capacidades porque existen desde los simples y basados en línea de comandos hasta aquellos que están completamente integrados

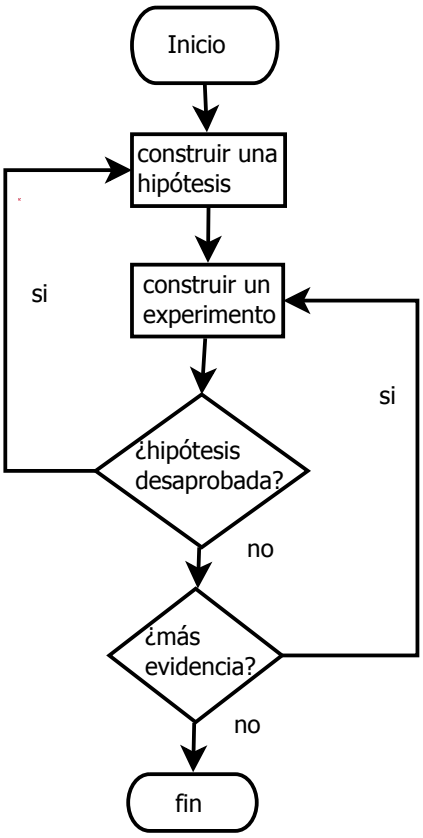

Figura 2.6: Algoritmo de depuración aplicando el método científico. [3]

en un IDE y cuentan con una interfaz gráfica. Los depuradores pueden examinar el código, poner puntos de ruptura, ejecutar simple o individualmente las instrucciones del código o sentencias, y examinar los estados de un programa.

Cuando la etapa de diagnóstico ha concluido debe continuarse con la *reparación*. Esta etapa no debe ser la más difícil a menos que se haya notado un error en los requerimientos. El principio básico aquí es corregir la causa no los síntomas.

La refactorización es importante en esta etapa, pues ayuda a dejar un código limpio y claro. Con la popularidad de las metodologías ágiles se han propuesto muchas maneras de mejorar la construcción de código fuente. Una manera de hacer esto es usando la refactorización, la cual permite mejorar el código para facilitar su cambio y su organización sin cambiar la funcionalidad original.

Otro factor importante en esta etapa son las revisiones rápidas de código, como son las inspecciones no formales propuestas en técnicas de programación extrema, usando la programación por parejas.

Por último, tenemos la etapa de la *reflexión* en la cual debemos de ser capaces de responder preguntas como ¿Funcionan adecuadamente los cambios generados? ¿Cuándo y por qué se generó el problema? Y evitar el mismo error en implementaciones posteriores.

Cuando nos aseguramos que un problema no vuelve a ocurrir y se sabe que fue lo que estuvo mal, se dice que se ha depurado correctamente. Es decir, acertadamente se supo la causa del error.

2.4. Depuración automática

Otra manera de realizar el proceso de depuración es mediante su automatización total o parcial. La idea detrás de la depuración automática es tener una herramienta que guíe al usuario interactivamente a depurar un programa.

De acuerdo a Butcher[6], en la depuración automática se requieren las siguientes suposiciones:

1. Asumir que un resultado incorrecto R es una secuencia de orígenes $o_1, o_2, \ldots o_n$
2. Para cada origen o_i algorítmicamente verificar si o_i contribuye a un defecto.
3. Si uno de esos orígenes conduce al defecto, regresar al paso 1 con $R = o_i$
4. Cuando R sea lo suficientemente pequeño se procede a reparar.
5. De otra manera, todos los orígenes o_i son correctos, entonces la infección no se presenta más. El proceso termina.

En un programa largo e interactivo, existen miles de funciones que se ejecutan. Varias de esas funciones se comunican vía estructuras de datos compartidas, en vez de tener simples argumentos y retorno de valores. Peor aún, las estructuras de datos pueden estar siendo accedidas simultáneamente. Por estas razones, la depuración automática funciona mejor con lenguajes de programación lógicos, funcionales y secuenciales, ya que no se modifican estados compartidos y el usuario no tiene que revisar el estado general del programa.

En términos más generales, la mayoría de las técnicas existentes solo se enfocan en la selección de subconjuntos potenciales de sentencias defectuosas y se les asigna un valor de peso de acuerdo a ciertos criterios.

Vale la pena mencionar que se ha encontrado que las herramientas de depuración automática ayudan a los desarrolladores más experimentados encontrando defectos más rápido.Weiser [7] propuso una de las primeras técnicas que soportan la depuración automática y en particular la localización de defectos: *program slicing.*

2.5. Trabajo relacionado

A continuación se describen los trabajos del área agrupados por los diversos criterios de detección de errores. En general de pueden separar los trabajos que usan herramientas de análisis estático y análisis dinámico. Todos los trabajos incluyen depuración delta como base. Los trabajos mencionados no siguen algún orden en especial.

2.5.1. Trabajos basados en depuración delta y análisis de cobertura

Yu [8] presenta la combinación de análisis de cobertura con depuración delta para aislar cambios que inducen defectos. El objetivo de este trabajo es una herramienta que soporta pruebas de regresión.

Autor	Herramienta	Validación de errores	Características	Ventajas	Desventajas
▪ Yu	▪ ADD	▪ pruebas de regresión	▪ Análisis de cobertura ▪ Generación de conjunto sospechoso	▪ Regresa menos falsos positivos que dd. ▪ Realiza menos pruebas que dd.	▪ Requiere muchos casos de prueba. ▪ No es completamente automático.

2.5.2. Trabajos basados en depuración delta y model checking

Groce [9] usa un método automático para encontrar múltiples versiones de un mismo error. Se analizan esas ejecuciones para producir una descripción importante de elementos clave de un error. Las descripciones producidas incluyen la identi-

ficación de porciones del código que representan fuentes cruciales para distinguir ejecuciones correctas y ejecuciones fallidas.

Weimer [10] presenta un algoritmo que explota la existencia de trazas correctas para localizar la causa de un error en una traza errónea, reportando un error por traza. Se implementó usando la herramienta Slam [11] el cual es un verificador de modelos que evalúa propiedades temporales de seguridad en programas escritos en C.

Autor	Herramienta	Validación de errores	Características	Ventajas	Desventajas
▪ Groce	▪ Java Path Finder	▪ Análisis estático	▪ Uso de verificación de modelos de estados explícitos. ▪ Análisis de condiciones de carrera y asserts en rutas exhaustivas. ▪ Usa un solver SAT.	▪ Evita caer en óptimos locales. ▪ Salidas válidas. ▪ Salidas en promedio 25 veces menores que dd.	▪ Reducción manual subsecuente
▪ Ball	▪ Slam	▪ Análisis estático	▪ Uso de trazas correctas. ▪ Uso de propiedades LTL.	▪ Tiene Complejidad lineal respecto al tamaño de entrada.	▪ Varios falsos positivos.

2.5.3. Trabajos basados en depuración delta y slicing dinámico

JINSI [12] es una herramienta de depuración automática la cual trata una ejecución como una serie de interacciones entre objetos, e.g, llamadas a métodos y retornos, que enventualmente producen un defecto. La herramienta puede almacenar y reproducir esas interacciones resoloviendo el problema de la reproducción

de defectos. Esta herramienta solo requiere de una ejecución que falle para poder minimizar las interacciones entre sus objetos a la cantidad requerida para la reproducción del problema. Esta herramienta utiliza la técnica de depuración delta y *slicing* dinámicos en llamadas a métodos. El resultado es una prueba unitaria con los objetos que precisan la reproducción. Además integra un enfoque estadístico en la presencia de múltiples ejecuciones.

El trabajo de Gupta [13] integra el potencial de la depuración delta con los beneficios del slicing dinámico de programas para reducir el espacio de búsqueda para código defectuoso. Se inicia usando depuración delta, para después usar esta salida y procesas *slices* dinámicos.

Autor	Herramienta	Validación de errores	Características	Ventajas	Desventajas
▪ Orso	▪ JINSI	▪ Análisis dinámico	▪ Resultado se regresa arreglado como una prueba unitaria. ▪ Enfoque estadístico. ▪ Basado en interacciones.	▪ Completamente automático. ▪ Solo requiere una ejecución fallida.	▪ Muy lenta
▪ Gupta		▪ Análisis estático y dinámico	▪ Intersección de sentencias de slices dinámicos. ▪ Utiliza analizadores estáticos Diablo. ▪ Utiliza valgrind como analizador dinámico.	▪ Defectos rápidos de detectar	▪ Muy acoplada a ciertos errores. ▪ No supone que un programa falla inicialmente por lo que detectar errores puede consumir mucho tiempo.

2.5.4. Trabajos basados en depuración delta, metaheurísticas y pruebas de mutación

Otro enfoque conocido es el uso de programación genética [14] para evolucionar variantes de programas hasta que se encuentre un individuo i.e., un programa que se ajuste a los requerimientos y a la vez no contenga el defecto en cuestión. La técnica entonces toma un programa como entrada, un conjunto de casos de prueba exitosos y un caso de prueba que demuestre un defecto. La herramienta se detiene hasta que se haya evolucionado una variante del programa original que pase todos los casos de prueba. Debido a que la programación genética introduce cambios irrelevantes al código, se usan algoritmos de diferencia estructurada basados en árboles y depuración delta en un post proceso para generar un parche con minimalidad 1.

Autor	Herramienta	Validación de errores	Características	Ventajas	Desventajas
▪ Weimer		▪ Análisis dinámico	▪ Uso de pruebas de mutación para generar variantes de programas. ▪ Uso de programación genética para mutar programas.	▪ Rapidez en obtener resultados	▪ Pasa por muchos pasos independientes y puede generar muchos falsos positivos

2.5.5. Trabajos basados en depuración delta y análisis de impacto

Para eliminar el esfuerzo excesivo gastado en la depuración, se desarrolló Autoflow [15], la cual es una herramienta de depuración automática para Software escrita en AspectJ. Autoflow integra el potencial de la depuración delta con el beneficio

del análisis del impacto en los cambios para reducir la búsqueda en la localización de cambios que encuentren fallos.

Autor	Herramienta	Validación de errores	Características	Ventajas	Desventajas
▪ Zhang	▪ Autoflo	▪ Pruebas de regresión. ▪ Análisis dinámico.	▪ Uso de heurística de ranking para validar un defecto. ▪ Uso de analizador Celadon de impacto estadístico. ▪ Escrito en AspectJ. ▪ Soporta pruebas de regresión	▪ Muy práctico. ▪ Disponible como plugin para Eclipse.	▪ Falsos positivos recurrentes. ▪ Se requieren varios casos de prueba exitosos y que fallen.

2.6. Sumario

Las pruebas automáticas marcan la pauta para poder procesar programas en tiempo de ejecución. El análisis dinámico de programas es sin duda una metodología bastante útil para análisis de errores, comportamiento y desempeño. Todos los enfoques existentes usan ya sea combinación de análisis estático o dinámico con depuración delta.

La depuración sigue siendo un proceso difícil, depende en gran medida de como se organizan las pruebas y como se procede a detectar un error.

Capítulo 3

Depuración Delta

La depuración es el proceso de detección y corrección de errores. Actualmente se han logrado automatizar estas etapas, especialmente la etapa de detección. Aunque existen muchos trabajos que intentan resolver ambas, se han logrado resultados simples. En general, cada etapa es un problema difícil.

En este trabajo de tesis se analiza solo la etapa de detección, en la cual se han obtenido mejores resultados en los últimos años. La detección automática de errores es una área aún en reciente progreso de investigación [16].

En la depuración se desean analizar fragmentos de código pequeños o condiciones simples con la intención de resolver un problema más fácilmente. En el capítulo anterior se mencionaron los pasos de la depuración. Siguiendo esta idea, en esta tesis se desean analizar casos de prueba que fallen con el fin de obtener una reducción en el código a depurar respecto al código del caso de prueba original. Esto es uno de los objetivos de la depuración delta, la cual tiene el propósito de ayudar a corregir más fácilmente un defecto y entender su naturaleza.

3.1. Simplificación y aislamiento de casos de prueba

Dos problemas importantes se tratan en este capítulo. El primero consiste en la simplificación de casos de prueba, la cual es muy importante en la depuración; es la primer tarea que se requiere para llegar a entender un defecto. El segundo problema es el aislamiento de defectos que tiene como fin la captura de los defectos.

La figura 3.1 muestra el principio de la simplificación y aislamiento de casos de prueba defectuosos usando pruebas automáticas. La partición del conjunto de datos es sucesiva hasta que no pueda simplificarse o aislarse más.

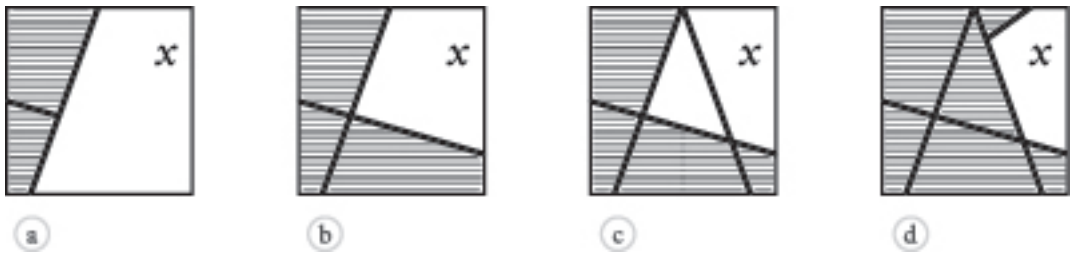

Figura 3.1: Simplificación de casos de prueba

El área sombreada representa las regiones del caso de prueba que no preservan un defecto. Sin embargo siempre se requiere verificar si el defecto se debe a los resultados emitidos por estos algoritmos. Usualmente esta verificación se realiza con herramientas de análisis estático o dinámico.

Detectar y aprobar la raiz de un defecto es difícil, debido a que las técnicas existentes que automatizan este proceso aproximan un resultado que debe ser evaluado. Aunque en muchas ocasiones se logre detectar adecuadamento los defectos, en otras ocasiones nos ayuda a encontrar síntomas de fragilidad en nuestras aplicaciones.

La depuración delta (DD) está compuesta por técnicas de búsqueda y reducción

de conjuntos de datos. Estas reducciones se producen usando pruebas automáticas y algoritmos de obtención de subconjuntos que preservan ciertas características. La DD es la pionera del área y muchos enfoques la usan como base.

3.2. Depuración delta

DD fue propuesta por Andreas Zeller [4], y consiste de un conjunto de algoritmos que usan funciones de prueba automáticas para resolver dos problemas: la simplificación de casos de prueba y el aislamiento de defectos. En realidad se trata de algoritmos de búsqueda con criterios de elección de nuevos subconjuntos (deltas) para reducir el espacio de búsqueda.

3.2.1. Simplificación de Casos de Prueba

En esta sección se explican a detalle los procedimientos de simplificación de casos de prueba. En el contexto de la simplificación, nos referimos a un caso de prueba como entradas de usuario (arreglos, valores enteros, cadenas, argumentos a funciones, etc) o como el código de un programa. Los casos de prueba simplificados son útiles por las siguiente razones:

- Ayudan a entender un defecto más fácilmente, en un contexto más general.
- Un caso de prueba simple es más fácil de depurar ya que involucra una traza de ejecución más corta, i.e.involucra menos estados de ejecución de un programa.
- Ayudan a detectar defectos duplicados, lo cual es una situación común.
- Se ahorra tiempo en el proceso de detección y corrección de errores.
- Son de gran valor a la hora de realizar un reporte de fallo.

Un caso de prueba corto, no solo permite descripciones más precisas de un problema, sino que también ayuda al diseño de los informes actuales y futuros de errores. El reporte de un defecto debe ser lo más específico posible, de manera que se pueda recrear el contexto en el que un programa falla.

Simplificar significa reducir casos de prueba, haciendo que cada parte de éstos sea importante o relevante para reproducir un defecto en particular. Comúnmente, las personas que encuentran un problema tardan mucho tiempo investigando que entradas producen un error y cuales otras no lo afectan.

Sabemos que aún teniendo un programa pequeño, puede involucrar cientos de posibles estados de ejecución. Un defecto por lo general queda atrapado en un subconjunto de estos estados [6], y aún para las mejores herramientas de depuración y personal capacitado en la detección de un defecto, este proceso sigue siendo tedioso y complejo.

Programa 3.1: Programa corto con muchos estados

```
#include <assert.h>
#include <pthread.h>

int i = 0;
pthread_mutex_t mutex;

void *thread( void *e ) {
    pthread_mutex_lock( &mutex );
     ++ i;
    pthread_mutex_unlock( &mutex );
    return 0;
}

int main() {
    pthread_t tid;
    pthread_mutex_init( &mutex, 0 );
    pthread_create( &tid, 0, thread, 0 );

    pthread_mutex_lock( &mutex );
    ++i;
    pthread_mutex_unlock( &mutex );

    pthread_join( tid, 0 );
    assert( i == 2 );
    return 1;
}
```

El programa 3.1 escrito en Lenguaje C tiene un amplio conjunto de estados, como se muestra en la figura 3.2.

La primer actividad en la depuración es la reproducción de un defecto. Es esencial asegurar bajo qué circunstancias el problema vuelve a ocurrir. Una vez que un reporte de error está en alguna base de datos, o se haya descubierto, debe ser procesado por alguien para poder repararlo. La primer tarea para el programador debe ser entonces reproducir el defecto. Reproducir un problema es importante por dos razones. Primero para observarlo y entenderlo (si alguien no es capaz de reproducir el fallo, no puede analizarlo, y por lo tanto no entenderá la causa). Mientras que la segunda razón consiste en verificar si el problema ha sido corregido.

El proceso de reproducción consiste de dos pasos:

1. Reproducir las *circunstancias del problema*: El escenario en el que ocurre.
2. Reproducir la *historia del problema*: Los pasos necesarios para recrearlo.

Si un problema ocurre en circunstancias específicas, la mejor manera de estudiarlo es exactamente bajo esas circunstancias. Sin embargo, en la práctica no es posible obtener el escenario ideal fácilmente. Encontrar el escenario donde el problema ocurre no es suficiente, sino que también debemos ser capaces de recrear los pasos que hacen al defecto manifestarse.

El requisito de las técnicas de simplificación involucra a casos de prueba que fallen, ya que estos son las entradas de los algoritmos que se muestran en las siguientes secciones.

La reparación de defectos es manual, por lo que automatizar este proceso es muy difícil. Sin embargo, la simplificación si se puede automatizar. Cuando un defecto se observa por primera vez, algún usuario debe añadirlo a una base de datos.

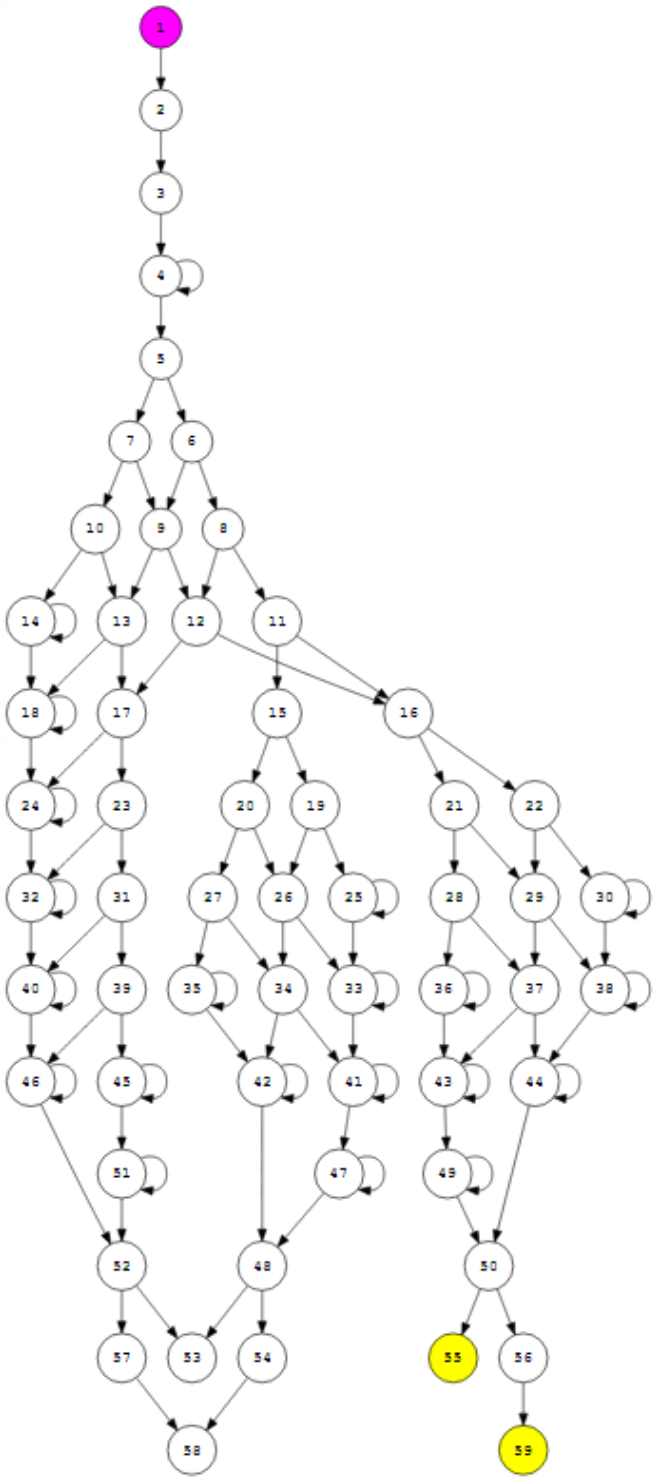

Figura 3.2: Posibles estados del programa 3.1

Ejemplificaremos el proceso de reporte de defectos con el llevado a cado por *Bugzilla*, la cual es una comunidad que cuenta con una base de datos de acceso libre a diferentes reportes de errores de Software y un conjunto de desarrolladores que se encargan de procesar y corregir errores. El proceso inicia cuando se incluye a la base de datos algún posible error encontrado por algún usuario. La categoría que es asignada a este problema se conoce como defecto abierto (open bug) debido a que ningún experto lo ha revisado hasta ese momento. Dentro de esta categoría, existen diversos estados tales como:

- **No confirmado**: Se asigna cuando un reporte es nuevo y nadie ha validado si se trata de un defecto o no. Los moderadores pueden cambiar el estado a nuevo o, pueden directamente marcarlo como resuelto o asignado.
- **Nuevo**: En este caso, el error ha sido añadido recientemente a una lista oficial y debe ser procesado. De este estado pasa a ser asignado a algún desarrollador en específico para posteriormente pasar al estado resuelto, en caso de que sea un error confirmado.
- **Asignado**: En este caso el bug no se ha resuelto aún, pero se está trabajando en él por alguien en particular.
- **Reabierto**: Este bug ha sido resuelto una vez, pero la solución ha resultado incorrecta.
- **Listo**: El bug tiene suficiente información, asi que se puede iniciar la reparación. En este caso, se cuenta con los casos de prueba requeridos, detalles específicos y la forma de reproducirlo, etc.

Despues que en reporte ha sido abierto, pasa por alguno de los siguientes estados:

- **Corregido**: Un defecto ha sido reparado y probado efectivamente.
- **Inválido**: El problema encontrado no es un defecto.

- **Wontfix (No reparable)**: El defecto nuncá se reparará.
- **Duplicado**: Aunque el problema luce diferente, es el duplicado de uno existente.
- **Worksforme (Esperando por nueva información)**: Todos los intentos de reproducir el defecto fallan, aún usando condiciones similares. Pero si nueva información aparece, el defecto puede reabrirse.
- **Incompleto**: El bug es descrito vagamente sin los pasos necesarios para su reproducción.

Una de las peticiones de los moderadores antes de comenzar a analizar un problema de estos, es un caso de prueba simple; mientras más pequeño mejor.

En este trabajo nos centraremos en la simplificación de programas escritos en lenguaje C. En la siguiente sección se muestra un ejemplo de simplificación de caso de prueba de un programa y de entradas de usuario. Posteriormente se detallan y discuten los algoritmos existentes en la literatura de depuración automática. Al conjunto de estos algoritmos se le conoce como Depuración Delta. Posteriormente se analizan las propiedades, ventajas y desventajas del uso de estos algoritmos.

3.2.2. Ejemplo de Simplificación de un Caso de Prueba

El proceso general de simplificación sigue una regla simple: Para cualquier componente del caso de prueba, se procede a revisar si es relevante para que el problema ocurra, en caso de no serlo se remueve. El propósito de simplificar un caso de prueba es hacerlo tan simple que cambiando cualquier componente cuase que el defecto no se vuelva a manifestar [1].

A continuación se muestra un ejemplo de un programa simple en lenguaje C. En este programa (3.2) se muestra un código que intenta crear e insertar nodos en un árbol binario.

Programa 3.2: Ejemplo 1

```
#include <stdlib.h>
#include <stdio.h>

struct nodo{
        int num;
        struct nodo * izq;
        struct nodo * der;
};

typedef struct nodo Nodo;

Nodo * creaNodo(int n)
{
        Nodo * raiz = (Nodo*)malloc(sizeof(Nodo));
        raiz->izq = NULL;
        raiz->der = NULL;
        raiz->num = n;
        return raiz;
}

void inserta(Nodo * ptr, int n)
{
        ptr = NULL;
        ptr = creaNodo(n);
        if(ptr != NULL)
                printf("nodo insertado\n");
        (ptr)->num++;
}

int main()
{
        Nodo * raiz = creaNodo(10);
        inserta(raiz->izq,4);
        inserta(raiz->izq->izq,1);
        inserta(raiz->izq->der,5);
        inserta(raiz->der,16);
        inserta(raiz->der->izq,14);
        inserta(raiz->der->der,22);
        return 0;
}
```

En 3.3 se muestra el código más simple que preserva el defecto al compilar y ejecutar 3.2.

Programa 3.3: Ejemplo 1 simplificado

```
#include <stdlib.h>
```

```
struct nodo{
        struct nodo * izq;
};

typedef struct nodo Nodo;

Nodo * creaNodo(int n)
{
        Nodo * raiz = (Nodo*)malloc(sizeof(Nodo));
        return raiz;
}

void inserta(Nodo * ptr, int n)
{
        ptr = creaNodo(n);
}

main()
{
        Nodo * raiz = creaNodo(10);
        inserta(raiz->izq,4);
        inserta(raiz->izq->izq,1);
        return 0;
}
```

En 3.4 se muestra el código correcto.

Programa 3.4: Ejemplo 1 correcto

```
#include <stdlib.h>
#include <stdio.h>

struct nodo{
        int num;
        struct nodo * izq;
        struct nodo * der;
};

typedef struct nodo Nodo;

Nodo * creaNodo(int n)
{
        Nodo * raiz = (Nodo*)malloc(sizeof(Nodo));
        raiz->izq = NULL;
        raiz->der = NULL;
        raiz->num = n;
        return raiz;
}
```

```
void inserta(Nodo ** ptr, int n)
{
        *ptr = NULL;
        *ptr = creaNodo(n);
        if(*ptr != NULL)
                printf("nodo insertado\n");
        (*ptr)->num++;
}

void preorden(Nodo * ptr)
{
        if(ptr->izq != NULL)
                preorden(ptr->izq);
        if(ptr->der != NULL)
                preorden(ptr->der);
        printf("%d\n", ptr->num);
}

main()
{
        Nodo * raiz = creaNodo(10);
        inserta(&raiz->izq,4);
        inserta(&raiz->izq->izq,1);
        inserta(&raiz->izq->der,5);
        inserta(&raiz->der,16);
        inserta(&raiz->der->izq,14);
        inserta(&raiz->der->der,22);
        preorden(raiz);
        return 0;
}
```

En el programa 3.2, se crea un nodo con la llamada a la función *creaNodo* en la línea 32. Esta sentencia es válida y se asigna memoria dinámica para el tipo de dato Nodo $*$. Después, en la línea 33 se llama a la función *inserta*, en la cual se intenta asignar al hijo izquierdo de la raiz del árbol binario un nodo. Hasta este punto, compilar y ejecutar el programa no causa ningún problema, la semántica del programa refleja que al llamar a *inserta* varias veces, después de crear el nodo raiz, causan el llenado del arbol binario paso a paso. Sin embargo, el problema es que falta pasar por referencia en *inserta* el argumento para que después de salir de la llamada a esta función, el árbol conserve los nodos agregados. Por esta razón, la entrada original "arbol.c", al momento de ejecutarlo, genera un fallo de segmentación por acceder a un espacio de memoria no inicializado.

El programa 3.3 es una versión simplificada del programa original, en donde quitando cualquier sentencia se pierde la generación del defecto que en este caso es un error en tiempo de ejecución (fallo de segmentación por tratar de acceder a memoria no asignada). Por último el programa 3.4 es la versión corregida. El hecho de obtener y analizar el caso simplificado nos indica directamente que la causa del defecto es el acceso a memoria.

Un punto importante en este problema es el uso de la granularidad, que consiste en obtener la unidad mínima arbitraria a remover usando los algoritmos de simplificación, la granularidad mínima posible en este caso son los caracteres. Sin embargo, en el ejemplo se usaron las sentencias como unidad mínima porque involucran menos pruebas. Estos detalles se tratan más adelante.

En el siguiente ejemplo, se muestra una simplificación de entradas dadas por el usuario. En este caso, es un arreglo de enteros, el cual es procesado por una implementación del algoritmo de ordenamiento quicksort mostrada en 3.5. Para este caso, la función de prueba incluye un oráculo. Puesto que se trata de un algoritmo de ordenamiento, el oráculo puede ser el código mostrado en 3.6.

Programa 3.5: Quick Sort

```
void qs(int *a, int size)
{
    if(size<=1)
        return;
    int pivot = a[0];
    int left = 0, right = 0;
    for(left = 1, right = size-1; left <= right;){
        if(a[left] >= pivot && a[right] <= pivot) {
            swap(left,right,a);
        }
        if(a[left] < pivot) left++;
        if(a[right] > pivot)right--;
    }
    swap(0,right,a);
    qs(a,right-1);
    qs(&(a[right+1]),size-right-1);
}
```

Programa 3.6: Oráculo

```
bool oraculo(int * a, int n)
```

```
{
        for(int i = 1; i < n ; i++){
                if(*(a + i -1) > *(a + i))
                        return false;
        }
        return true;
}
```

La entrada 3,1,4,9,6,2,7,5 da como resultado el arreglo 2,1,3,4,5,6,7,9 por lo tanto se puede verificar que el programa no realiza el ordenamiento correctamente. Si procedemos a simplificar la entrada para depurar más fácilmente el programa, nos encontramos con que la entrada 3,1,2 es suficiente para demostrar que este programa falla.

3.2.3. Algoritmo de Simplificación

El algoritmo clásico de simplificación de casos de prueba llamado *ddmin*, propuesto por Zeller y Hildebrandt [4], generaliza y minimiza un caso de prueba que falla de acuerdo a una función de prueba (el prefijo dd significa depuración delta).

El algoritmo requiere como entrada un caso de prueba que falle, el cual se simplifica mediante pruebas automáticas sucesivas. El algoritmo se detiene cuando un caso de prueba mínimo es alcanzado, es decir cuando cualquier cambio en este hace que el defecto no sea visible nuevamente.

Usando pruebas una y otra vez podemos discriminar partes del caso de prueba original identificando lo que realmente es útil o no para la reproducción del defecto bajo estudio. Gracias a las pruebas automáticas este proceso de búsqueda es posible de manera automática.

En general, asumimos que la ejecución de un programa es determinado por un número de circunstancias, las cuales incluyen el código fuente, datos de usuario o de dispositivos y la plataforma en la que se ejecuta.

Definición 3.2.1. Denotemos el conjunto de posibles configuraciones como C.

Cada $r \in C$ denota una ejecución específica de un programa.

También definimos a $r_{✗} \in C$ como una ejecución que falla. Es decir, existe una función $test(C)$ que puede dar como resultado $r_{✓}$ (pasa la prueba $test$), y $r_{✗}$ no pasa la prueba o incluso genera un resultado indeterminado.

La función de prueba es totalmente personalizable y puede ser guiada por un oráculo que puede ser comúnmente algún error en tiempo de ejecución o una implementación como mecanismo de verificación. Nos concentraremos en la diferencia entre $r_{✓}$ y $r_{✗}$. Esta diferencia en muchos casos, resulta ser la causa de un defecto en particular, en donde, mientras menor sea la diferencia mejor se califica la causa del error.

De acuerdo al estándar de POSIX 1000.3 de pruebas, existen 3 posibles maneras de calificar el resultado de una prueba [4]:

1. Prueba exitosa, denotada por el símbolo ✓.
2. Prueba fallida, denotada por el símbolo ✗.
3. Prueba indeterminada, denotada por el símbolo ?.

Definición 3.2.2. Una función de prueba $test : C \rightarrow ✗, ✓, ?$ determina para una ejecución específica r si ocurre algún error (✗) o no (✓) o un estado indeterminado (?).

Algunos libros de texto de depuración explican como proceder a aislar una posible causa de fallo, sin embargo con la ayuda de las pruebas automáticas podemos aislar estas causas usando los mismo principios.

En el libro *The Practice of Programming* [17], el proceso de simplificación se propone hacer manualmente con la recomendación de usar el algoritmo de búsqueda binaria, i.e., revisando un caso de prueba por la mitad y para cada

parte que falle partirlo nuevamente hasta encontrar una sección más pequeña que manifieste un fallo.

Sea c un caso de prueba que pase alguna función de prueba *test*, $c_{\text{✗}}$ un caso que no pase *test*. Entonces minimizando la diferencia entre c y $c_{\text{✗}}$ da como resultado la simplificación de $c_{\text{✗}}$.

Un caso de prueba $c \subseteq c_{\text{✗}}$ se dice que es mínimo si no existe un subconjunto más pequeño de $c_{\text{✗}}$ que haga fallar al programa bajo estudio usando la función *test*.

Formalmente:

Definición 3.2.3. Un conjunto $c \subseteq c_{\text{✗}}$ es llamado un mínimo global de $c_{\text{✗}}$ si $\forall c' \subseteq c_{\text{✗}}$ se cumple que $|c'| < |c| \rightarrow test(c') \neq x$

En la práctica sería ideal obtener un mínimo global. Sin embargo es impráctico tratar de obtenerlo pues se requieren $2^{|c_{\text{✗}}|}$ pruebas y esto es muy costoso computacionalmente, pues tiene complejidad exponencial.

Definición 3.2.4. Un caso de prueba $c \subseteq c_{\text{✗}}$ es un mínimo local de $c_{\text{✗}}$ si $\forall c' \subset c$ se cumple $test(c') \neq x$.

Lo que se desea obtener es un caso de prueba que falle y en donde sus elementos sean todos significativos. Sin embargo determinarlos sigue requiriendo $2^{|c|} - 2$ pruebas.

Una solución al problema es encontrar una aproximación, la cual consiste en encontrar un caso de prueba que al remover cualquiera de sus elementos sigue siendo significativo para producir el defecto que se estudia. La n-minimalidad de un caso de prueba es una propiedad interesante que consiste en remover cualquier combinación de n elementos para desaparecer el fallo.

Lo que nos interesa son los conjuntos de prueba con minimalidad 1. Un caso de prueba c tiene minimalidad 1 si removiendo cualquier cambio simple causamos

que desaparesca el defecto.

Definición 3.2.5. Un caso de prueba $c \subseteq c_{✗}$ tiene minimalidad n si $\forall c' \subset c$ se cumple que $|c| - |c'| \leq n \rightarrow test(c') \neq ✗$.

A continuación se muestra el algoritmo *ddmin*, el cual parte de la idea del algoritmo de búsqueda binaria y de particiones ordenadas. En el caso de que el tamaño de $c_{✗}$ sea 1 entonces por definición $c_{✗}$ es mínimo. De otra manera, particionamos a $c_{✗}$ y a cada partición le llamaremos delta y la denotaremos con el símbolo Δ_i

Data: caso de prueba $c_{✗}$ test($c_{✗}$) = ✗
Result: $c'_{✗}$ tal que $c'_{✗} \subseteq c_{✗}$, test($c'_{✗}$) = ✗ y $c'_{✗}$ tenga minimalidad 1
ddmin($c_{✗}$,2):
if $\exists i \in 1,\ldots,n$ $test(\Delta_i)$=✗ **then**
    ddmin(Δ_i, 2);
else
    if $\exists i \in 1,\ldots,n$ $test(\nabla_i)$= ✗ **then**
        ddmin($\nabla_i, max(n-1, 2)$);
    else
        if $n < |c_{✗}|$ **then**
            ddmin($c'_{✗}, min(c'_{✗}, |2n|)$);
        else
            regresa $c_{✗}$;
        end
    end
end

Algoritmo 1: Algoritmo *ddmin*

El algoritmo necesita un caso de prueba que falle $c_{✗}$ y el tamaño de la partición n. Dos subconjuntos Δ_1 y Δ_2 con tamaños similares se prueban, esto da tres posibilidades

1. **Reducir a** Δ_1. Esto es, Δ_1 falla, entonces Δ_1 es un caso de prueba más pequeño.

2. **Reducir a** Δ_2, Esto es, Δ_1 no falla pero Δ_2 si, entonces Δ_2 es un caso de prueba más pequeño.

3. **Estado de ignorancia**, esto ocurre cuando ambas pruebas no pasan, ni Δ_1, Δ_2 califican para ser reducidos.

En los dos primeros casos, podemos simplemente continuar con la búsqueda en los subconjuntos que fallan.

Con estas reglas, podemos partir entonces cualquier subconjunto de la entrada e ir tomando como el nuevo caso de prueba a aquel que demuestre un fallo, ¿pero qué pasa cuando no ocurre esto?

Analizando el peor de lo casos, después de separar a c en n subconjuntos y probarlos, despues de verificar si todas las pruebas pasan o no se resuelven, debe tenerse alguna manera de obtener un conjunto más pequeño, puesto que sabemos que el conjunto más grande c falla. Debemos contar con algún mecanismo que nos garantize y aumente la probabilidad de encontrar una prueba que falle probando con algún subconjunto de c.

Probando subconjuntos largos de c incrementamos la probabilidad de que las pruebas en el nuevo c' obtenido fallen, siendo obviamente $|c'| < |c|$ y la diferencia entre c y c' es más pequeña, sin embargo una diferencia pequeña significa una progresión más lenta. Por otro lado, probando con subconjuntos pequeños de c, conseguimos una progresión más rápida pero la probabilidad de que las pruebas fallen es menor.

Estas formas de pruebas se pueden combinar particionando a c en un número mayor de subconjuntos y probando cada nuevo Δ_i más pequeño como también utilizando complementos más largos.

De esta manera, tenemos particiones y pruebas ordenadas que llevan a obtener la convergencia. La desventaja es que mientras más particiones tenemos, más evaluaciones de pruebas se requieren.

Entonces probando los distintos Δ_i y sus respectivos complementos pueden ocurrir cuatro casos:

1. **Reducir a subconjunto**: Si probando cualquier Δ_i falla alguno, entonces Δ_i es un caso de prueba más pequeño, se debe continuar reduciendo a Δ_i con $n = 2$ subconjuntos.

2. **Reducir a complemento**: Si probando cualquier $\nabla_i = c_{\text{✗}} - \Delta_i$ falla, entonces ∇_i es un caso de prueba más pequeño, se debe continuar reduciendo a ∇_i con $n - 1$ subconjuntos.

3. **Incrementar granularidad**: En este caso las pruebas no fallan, entonces se aumenta el número de subconjuntos a $2n$ con esto aumentamos la probabilidad de encontrar una prueba que falle. En el caso en que $|2n| > |c_{\text{✗}}|$ entonces el valor límite para $n = |c_{\text{✗}}|$

4. **Fin**: El proceso se repite hasta que la granularidad no pueda ser incrementada. En este caso se ha probado removiendo cualquier número de cambios sin que las pruebas fallen, el conjunto resultantes es mínimo.

El algoritmo va probando particiones ordenadas. La figura 3.3 muestra las primeras 10 llamadas recursivas a *ddmin* inicialmente con $n = 2$ usando potencias de 2.

En 3.3.*a* La parte sombreada representa la primer mitad a probar. Se inicia con n = 2. Como no falla entonces en *b* se prueba la segunda mitad. De igual manera, al no falla entonces se prueban sus respectivos complementos repitiendo *a* y *b*. Posteriormente se prueba con $n = 4$. en *c*,*d*,*e* y *f* el proceso se repite. En *g* , *h*, *i* y *j* se prueban los complementos con $n = 4$.

En el programa 3.5 referente a la versión errónea del quicksort, bajo la entrada $3, 1, 4, 9, 6, 2, 7, 5$ da como resultado el arreglo $2, 1, 3, 4, 5, 6, 7, 9$.

El algoritmo *ddmin* entonces actua de la siguiente manera sobre la entrada.

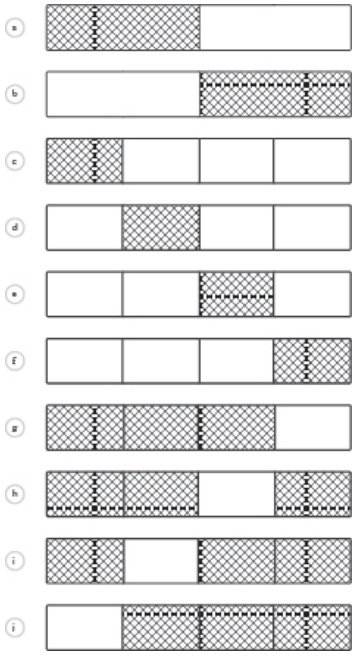

Figura 3.3: Particiones ordenadas. Primeras 10 evaluaciones de ddmin. Iniciando con $n = 2$

El número de pruebas llevado por *ddmin* en el peor caso es $|c_{\text{✗}}|^2 + 3|c_{\text{✓}}|$, y por lo tanto tiene complejidad $O(n^2)$.

3.3. Aislamiento de Entradas

Para este nuevo algoritmo, cada vez que una entrada pasa las pruebas i.e., $c' = \text{✓}$, una entrada más grande es requerida. Esto es debido a que ahora se requieren dos casos, $c'_{\text{✓}}$ y $c'_{\text{✗}}$, donde $c'_{\text{✗}}$ será minimizado y $c'_{\text{✓}}$ será maximizado para converger a una posible causa de error.

En este problema, aislar significa entonces encontrar solo una parte relevante que al removerla el defecto considerado deja de existir. En general, el aislamiento es más eficiente que la simplificación. Si se tiene una entrada larga defectuosa, aislar la diferencia llevará a la causa del defecto más rápido que la simplificación. Además nos entrega una explicación más detallada del problema o al menos

Inicio	
$c_{\text{✗}}$	$[3, 1, 4, 9, 6, 2, 7, 5]$
iniciando $ddmin(c_{\text{✗}}, 2)$	
$\Delta_1 = [3, 1, 4, 9]$	✓
$\Delta_2 = [6, 2, 7, 5]$	✓
$\Delta_1 = [3, 1]$	✓
$\Delta_2 = [4, 9]$	✓
$\Delta_3 = [6, 2]$	✓
$\Delta_4 = [7, 5]$	✓
$\nabla_1 = [4, 9, 6, 2, 7, 5]$	✓
$\nabla_2 = [3, 1, 6, 2, 7, 5]$	✗
$\Delta_1 = [3, 1]$	✓
$\Delta_2 = [6, 2]$	✓
$\Delta_3 = [7, 5]$	✓
$\nabla_1 = [6, 2, 7, 5]$	✓
$\nabla_1 = [3, 1, 7, 5]$	✓
$\nabla_1 = [3, 1, 6, 2]$	✗
$\Delta_1 = [3, 1]$	✓
$\Delta_2 = [6, 2]$	✓
$\Delta_1 = [3]$	✓
$\Delta_2 = [1]$	✓
$\Delta_3 = [6]$	✓
$\Delta_4 = [2]$	✓
$\nabla_1 = [1, 6, 2]$	✓
$\nabla_2 = [3, 6, 2]$	✓
$\nabla_3 = [3, 1, 2]$	✗

buenos argumentos del defecto, o los síntomas de un problema mayor.

Al utilizar una función exitosa de prueba $pass(c')$ el programador requiere una nueva configuración. Esta puede no tener características extras, por ejemplo, si $fail(c')$ verifica si existe un fallo de segmentación, pass(c') simplemente podría ser lo opuesto a $fail(c')$.

Tener un aislamiento como salida resulta muy útil, por ejemplo si un caso simplificado consiste de 100 sentencias, un aislamiento puede consistir de solo 2.

Otro punto importante es que frecuentemente el tiempo de ejecución de un

programa es proporcional al tamaño de su entrada, por lo que en algunos casos la simplificación requiere mayor cantidad de pruebas pero menor tiempo de ejecución, debido a entradas más pequeñas. En la práctica, resulta útil tanto la simplificación como el aislamiento.

Se necesita extender a *ddmin* de manera que se encuentren dos conjuntos $c'_{\checkmark}$ y $c'\text{✗}$ tales que $\emptyset = c_{\checkmark} \subseteq c'_{\checkmark} \subseteq c'\text{✗} \subseteq c_{\text{✗}}$ y $\Delta = c'_{\text{✗}} - c'_{\checkmark}$ sea mínima.

Definición 3.3.1. La diferencia $\Delta = c'_{\text{✗}} - c'_{\checkmark}$ es mínima si $\forall \Delta_i \subset \Delta$, test$(c'_{\checkmark} \cup \Delta_i)$ $\neq$ ✓$\wedge$ test$(c'_{\text{✗}} - \Delta_i) \neq$ ✗

Como el número de $|\Delta|$ es exponencial, se limita a calcular una aproximación usando el concepto de minimalidad.

Definición 3.3.2. Diferencial de minimalidad n. $\Delta = c'_{\text{✗}} - c'_{\checkmark}$ tiene minimalidad n si $\forall \Delta_i \subseteq \Delta, |\Delta_i| \leq n \rightarrow test(c'_{\checkmark} \cup \Delta_i) =$ ✓ $\wedge\ test(c'_{\text{✗}} - \Delta_i) \neq$ ✗

Consecuentemente Δ tiene minimalidad 1 si $\forall \Delta_i \in \Delta|$ test$(c'_{\checkmark} \cup \Delta_i) \neq$ ✓$\wedge$ test$(c'_{\text{✗}} - \Delta_i) \neq$ ✗.

El objetivo es aislar una diferencia con minimalidad 1 entre un caso de prueba exitoso y uno defectuoso.

En todo momento $c'_{\text{✗}}$ y $c'_{\checkmark}$ actúan como cota inferior y superior respectivamente. Los cambios que *ddmin* necesita son:

- Extenderlo de tal manera que trabaje con $c'_{\text{✗}}$ y $c'_{\checkmark}$ inicialmente con $c'_{\text{✗}} = c_{\text{✗}}$ y $c'_{\checkmark} = \emptyset$.
- Calcular subconjuntos $\Delta_i = c'_{\text{✗}} - c'_{\checkmark}$ en vez de subconjuntos de $c'_{\text{✗}}$.
- Cambiar la regla de reducir a subconjunto tal que $c'_{\checkmark} \cup \Delta_i$ se pruebe
- Introducir dos reglas adicionales:

1. Incrementar a complemento. Si $c'_{\text{✗}} - \Delta_i$ pasa para cualquier subconjunto Δ_i, entonces $c'_{\text{✗}} - \Delta_i$ es un caso de prueba exitoso más largo. Se debe continuar reduciendo la diferencia entre $c'_{\text{✗}} - \Delta_i$ y $c'_{\text{✗}}$.
2. Incrementar a subconjunto. Si $c'_{\text{✓}} \cup \Delta_i$ pasa para cualquier subconjunto Δ_i. Entonces $c'_{\text{✓}} \cup \Delta_i$ es un caso de prueba que pasa más largo.

Este algoritmo es conocido como *dd*, el cual se muestra a continuación.

Data: $c'_{\text{✗}}$,$c'_{\text{✓}}$, y una función de prueba test
Result: $(c'_{\text{✓}}.c'_{\text{✗}})|\emptyset = c_{\text{✓}} \subseteq c'_{\text{✓}} \subseteq c'_{\text{✗}} \subseteq c_{\text{✗}}$ y $\Delta = c'_{\text{✗}} - c'_{\text{✓}}$ tenga minimalidad 1
$\mathrm{dd}(c_{\text{✓}}, c_{\text{✗}}, 2)$:
if $\exists i \in 1, \ldots, n\ test(c'_{\text{✓}} \cup \Delta_i) = \text{✗}$ **then**
  $\mathrm{dd}(c'_{\text{✓}}, c'_{\text{✓}} \cup \Delta_i, 2)$
  else if $\exists i \in 1, \ldots, n\ test(c'_{\text{✗}} - \Delta_i) = \text{✓}$ **then**
    $\mathrm{dd}(c'_{\text{✗}} - \Delta_i, c'_{\text{✗}}, 2)$
    else if $\exists i \in 1, \ldots, n\ test(c'_{\text{✓}} \cup \Delta_i) = \text{✓}$ **then**
      $\mathrm{dd}(c'_{\text{✓}} \cup \Delta_i, c'_{\text{✗}}, max(n-1, 2))$
      else if $\exists i \in 1, \ldots, n\ test(c'_{\text{✗}} - \Delta_i) = \text{✗}$ **then**
        $\mathrm{dd}(c'_{\text{✓}}, c'_{\text{✗}} - \Delta_i, max(n-1, 2))$
        else if $n \leq |\Delta|$ **then**
          $\mathrm{dd}(c'_{\text{✓}}, c'_{\text{✗}}, min(2n, |\Delta|))$
          else
            regresa $(c'_{\text{✓}}, c'_{\text{✗}})$
          end
        end
      end
    end
  end
end

Algoritmo 2: Algoritmo *dd*

El número de pruebas llevado por *dd* en el peor caso es $|c_{\text{✗}}|^2 + 3|c_{\text{✓}}|$, de hecho *ddmin* es un caso específico de *dd*. Sin embargo *dd* es más eficiente que *ddmin* si no existen casos sin resolver. En el mejor caso de *dd* se usa la mitad de pruebas

que en *ddmin*.

3.4. Depuración Delta Jerárquica

La depuración delta jerárquica (*hdd*) es un algoritmo simple y efectivo comparado con el algoritmo convencional *ddmin*, además se caracteriza por su calidad en las salidas usando entradas jerárquicas.

Hdd utiliza el AST de un programa. Un ejemplo de AST se presenta a continuación en la figura 3.4 con su respectivo fragmento de código (3.7).

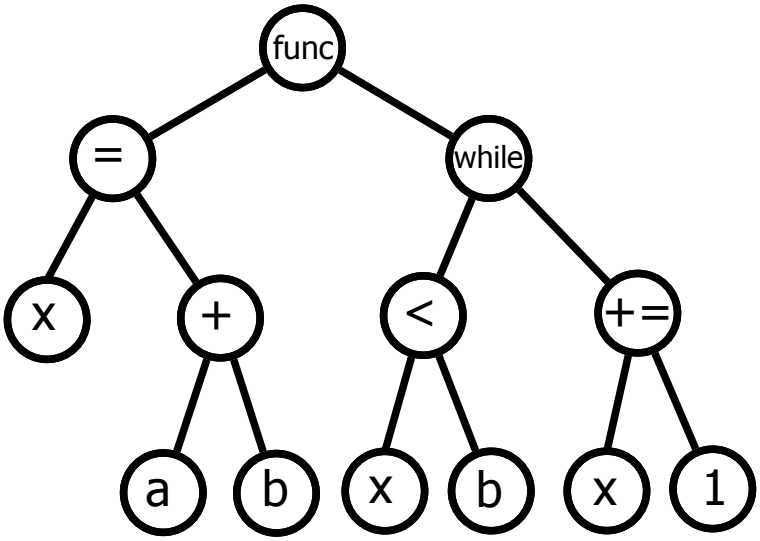

Figura 3.4: Árbol abstracto sintáctico del programa 3.7

Programa 3.7: Ejemplo de un fragmento de código.

```
void func()
{
        x = a + b;
        while(x < b){
                x += 1;
        }
}
```

Cada nivel que no sea el primero del árbol contiene hijos de los nodos del nivel anterior. El primer nivel es el 0 y pertenece a la raíz del árbol, en el último nivel se encuentran las hojas del AST.

En vez de tratar a las entradas como una lista plana y atómica, se aplica DD aprovechando la estructura de la entrada en forma de árbol. En particular, aplicamos el algoritmo original a cada nivel de la entrada del programa trabajando del nivel más general al más específico. Después se procede con una poda de porciones irrelevantes de la entrada de manera prematura. Todas las entradas generadas son configuraciones sintácticamente válidas con lo que se reduce el número de salidas inconclusas.

Hdd produce salidas de calidad y realiza menos pruebas que *ddmin*, es capaz de escalar a entradas de tamaño considerable que *ddmin* no puede procesar en la práctica. en general *hdd* es bastante práctico para entradas jerárquicas.

Un programa puede hacer uso de *hdd* cuando es considerado como entrada para un compilador o un intérprete, si un programa largo causa un fallo específico *hdd* opera através de su árbol de sintaxis abstracto (AST). La configuración mínima es encontrada para cada nivel de dicho árbol, e.g., primero se encuentra la configuración mínima de prototipos, variables globales y funciones de primer nivel, para después determinar sentencias, declaraciones locales, etc. Este proceso se desarrolla hasta el nivel más bajo del AST.

Se limita a un nivel por cada llamada local a *ddmin*, la idea es examinar pequeños fragmentos de la entrada para su minimización, esto también explota la independencia relativa de los nodos en diferentes ramas del árbol.

El algoritmo original *ddmin* produce casos de prueba que son difíciles de leer al minimizar identificadores, y removiendo secciones de código. La ventaja de *hdd* es que reemsambla el programa más cercano a su estructura original.

3.4.1. Algoritmo HDD

Usando el árbol de sintaxis se manipula la entrada y se generan los nuevos casos.

El algoritmo comienza procesando el nivel más alto o la raiz del AST. Se determina inicialmente la configuración mínima global necesaria para la reproducción del defecto, i.e., variables globales, prototipos de funciones, y definiciones de tipos personalizados.

Cuando se obtiene una configuración mínima de un nivel se procede al siguiente nivel. Se usa el algoritmo estándar *ddmin* para determinar la configuración mínima de todos los nodos por cada nivel. Cada subárbol podrá ser tratado individualmente.

Las entradas de *hdd* deben ser jerárquicas para sacar provecho de su estructura. La idea básica parte de encontrar un caso de prueba simple del nivel más general al particular, i.e., , del nivel 0 al $n - 1$ en el árbol.

Al limitar la simplificación a cada nivel del árbol a la vez, se poda información más inteligentemente. En cada nivel se deben probar múltiples configuraciones para determinar una salida simplificada del caso de prueba, esto involucra llamar

a *ddmin* en cada nivel.

Data: caso de prueba $c_{\text{✗}}$ test($c_{\text{✗}}$) = ✗
Result: $c'_{\text{✗}}$ tal que $c'_{\text{✗}} \subseteq c_{\text{✗}}$, test($c'_{\text{✗}}$) = ✗ y $c'_{\text{✗}}$ tenga minimalidad 1
hdd($c'_{\text{✗}}$):
nivel ← 0 ;
nodos ← etiquetarNodos(arbol,nivel);
while *nodos* ≠ 0 **do**
    configMin ← ddmin(nodos);
    poda(arbol,nivel,configMin);
    nivel ← nivel +1;
    nodos ← etiquetarNodos(arbol,nivel);
end

Algoritmo 3: Algoritmo HDD

Ahora de discute la complejidad con respecto al número de pruebas realizadas. La complejidad en el peor caso de *hdd* es la misma que *ddmin*, complejidad cuadrática respecto al número de pruebas del tamaño original de la entrada. El peor caso para *hdd* ocurre cuando la entrada es una lista plana y secuencias sin estructura jerárquica.

Para entradas con estructuras jerárquicas como programas, *hdd* se desempeña mejor.

Supongamos que *hdd* se ejecuta en un árbol balanceado de tamaño n con un factor constante b referente a las ramas tal que para cada padre exactamente un hijo permanece en la configuración. Hay entonces $log_b n$ niveles en ese árbol. En cada nivel, se invoca a *ddmin*, y se requieren $O(b^2)$ pruebas. Consecuentemente ejecutamos $O(b^2 log_b n)$ pruebas, o simplemente O(log n) porque b es constante

En el nivel más general, se examina la raiz del árbol. Para cada nivel subsecuente i, se examinan b nodos para cada nodo padre incluido o $b(m^{i-1})$ nodos. Se ejecuta *ddmin* a cada nivel, por lo tanto se prueban casi $O((b(m^{i-1}))^2)$ veces por nivel, sumando sobre el árbol entero se tiene:

$1 + \sum_{i=1}^{log_b n}(bm^{i-1})^2$

$= 1 + b^2 \sum_{i=1}^{log_b n}(m^{i-1})^2$

$<= 1 + b^2(\sum_{i=1}^{log_b n}(m^{i-1}))^2$

$= 1 + b^2(m^{log_b n} - 1/m - 1)^2$ cuando $m \neq 1$

hdd en el peor caso:

$O((m^{log_b n} - 1/m - 1)^2) = O(m^{2log_b n})$ pruebas, cuando m es distinto de 1, en otro caso $1 + b^2 log_b n$. por lo tanto $O(log_b n)$ pruebas.

Si el árbol está bien balanceado, se puede esperar eliminar porciones largas de la entrada siempre que se remueva un nodo de nivel superior en el árbol. Esta propiedad es la que permite obtener mejores cotas asintóticas.

3.5. Sumario

Hdd en particular ayuda a la calidad de las salidas generadas. Como se mencionó anteriormente, idealmente se desea obtener el mínimo global de una entrada que induce defectos. Desafortunadamente eso es bastante difícil, de hecho el problema de la minimalidad global de un árbol es un problema NP Completo.

En lugar de tratar de alcanzar una meta tan difícil, se desea obtener cierta mini malidad con respecto a los vecinos inmediatos. Con tal objetivo, sólo podemos examinar todos los vecinos de la configuración actual hasta que no se induzca el error, por lo tanto se obtienen salidas con mínimos locales.

Ddmin manifiesta algunas debilidades importantes e.ga mayor tamaño tenga una entrada simplificada, mayor cantidad de llamadas a una función de prueba se requieren. Se pueden emplear diferentes métodos para evitar usar muchas pruebas e.gdetener la simplificación transcurrido cierto límite de tiempo o establecer un

umbral de logitud del nuevo caso de prueba. Sin embargo existe una mejor manera que consiste en reducir una entrada defectuosa e incrementar una entrada que no presente el mismo defecto.

Capítulo 4

Depuración automática usando algoritmos genéticos.

En este apartado se detallan los algoritmos propuestos. Estos algoritmos usan una metaheurística debido a que los problemas que se intentan resolver en esta tesis pueden ser modelados como problemas de optimización combinatoria discreta.

En la siguiente sección se incluyen conceptos y teoría relacionados con algoritmos genéticos.

4.1. Algoritmos genéticos

Se propone una metaheurística para resolver el problema de la simplifiación y el aislamiento de defectos. La idea es modelar el problema como un caso especial del problema de la mochila 0-1, que consiste en visualizar al conjunto de datos o de código como una lista de bits (0 se utiliza, 1 no se utiliza) y trabajar sobre ellos usando un algoritmo genético.

Al final la intención es obtener casos de prueba con minimalidad aceptable en

la práctica, que probablemente puedan llegar a ser más grandes que con *ddmin* o *hdd* pero obtenidos de manera rápida evitando pruebas repetidas y muy útiles en la práctica.

El problema de la mochila es un ejemplo de problema de optimización combinatoria, se resuelve mejorando una solución actual usando información de otras soluciones candidatas. Se asume que la mochila tiene volumen entero positivo (capacidad), que existen distintos artículos con volumen y beneficio específico, el objetivo es ocupar el mayor espacio posible sin exceder el volumen de la mochila y al mismo tiempo obtener el mayor beneficio posible.

Se puede obtener una aproximación a la solución del problema de la mochila usando distintos algoritmos, como por ejemplo, los genéticos.

Los algoritmos genéticos fueron propuestos por Goldberg y se inspiran en la teoría de la evolución de Darwin, estos algoritmos utilizan la regla que dictamina "solo las especies más aptas sobreviven", y que éstas están sometidas al proceso de reproducción, cruza y mutación.

Una solución generada por un algoritmo genético es llamada cromosoma, en el caso de entradas o de programas estos cromosomas pueden ser modelados por una lista de sentencias o datos. A la colección de cromosomas que se tiene se conoce como población. Un cromosoma es compuesto por genes, el valor de cada gen puede ser binario, numérico, caracteres,etc.

Estos cromosomas se somenten al proceso de evaluación de la función de aptitud para medir la calidad de la solución entragada por el algoritmo genético. Una generación consiste de la evaluación, selección, cruza y mutación de estos cromosomas. Un algoritmo maneja cierta cantidad de generaciones como método de paro o convergencia.

En general un algoritmo genético consta de:

1. Determinar el número de cromosomas, generaciones, tasa de mutación y

cruza.

2. Generar la población inicial de manera aleatoria.
3. Mientas el número de generaciones no es alcanzada, seguir con el siguiente paso, de otra manera saltar al paso 9.
4. Calcular el valor de aptitud para cada cromosoma usando una función objetivo.
5. Seleccionar los cromosomas más aptos.
6. Proceso de cruza.
7. Proceso de mutación.
8. Nuevos cromosomas se obtienen, repetir paso 3.
9. Entregar una lista o el mejor candidato.

4.2. Depuración automática usando algoritmos genéticos

Dada la naturaleza de los algoritmos vistos hasta el momento, se observa que éstos son algoritmos de búsqueda de tipo voraz, i.e., en el momento que encuentran una posible y mejor solución que la actual (reducción en el contexto de este trabajo) se elige como nuevo caso de prueba sin revisar otros posibles mejores candidatos.

Los algoritmos de búsqueda voraz pueden generalizarse con los siguientes elementos: un operador de transformación, una función de aptitud y un mecanismo que prueba si la elección ha sido mejorada. Trabajan repetidamente mejorando la solución actual usando la evaluación de la función de aptitud y reemplazándola por una mejor.

El operador de transformación determina qué se elige como nueva solución, la cual consigue tener un mejor valor de aptitud, este valor es calculado mediante la función de aptitud.

Tanto *ddmin* como *dd* son instancias de búsqueda voraz, donde la aptitud de una entrada que induce defectos es el inverso del tamaño de su entrada. El operador de tranformación remueve fragmentos de la entrada, en la práctica resulta útil utilizar otro operador de transformación. Regerh [18] propone varios modelos para esta operación e.g., sustituir variables por constantes o utilizar nuevas variables para operaciones usadas a menudo. En este trabajo solo se trabaja con la eliminación de fragmentos de una entrada como la operación de tranformación, la función de aptitud es $1/|c_{\text{✗}}|$.

La búsqueda voraz resulta útil en este contexto, sin embargo existe un problema importante: La ejecución de una función de prueba (denotada por *test* en los algoritmos vistos en el capítulo anterior) son costosas debido a que requieren la construcción del nuevo código fuente, la compilación, la revisión y la asignación de valores, y al final una ejecución del nuevo programa que depende de su propia implementación.

Otro problema importante es que con *dd* y *ddmin*, muchas pruebas se repiten, se podría considerar el guardar una lista tabú para no permitir pruebas repetidas sobre los mismo datos, sin embargo al final resulta en ejecutarse todos los subconjuntos posibles de la entrada.

4.3. Simplificación de Casos de Prueba

1. Evaluación: El primer paso consiste en evaluar la función objetivo, la cual está en función del tamaño de la entrada, e.g., sentencias para código fuente. Esto se hace para cada cromosoma o individuo de la población.

2. Selección: los cromosomas más aptos distinguidos por el valor de su aptitud

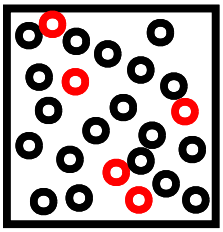

Figura 4.1: Proceso de evaluación

tienen mayor probabilidad de ser elegidos para la siguiente generación. Hay un parámetro llamado probabilidad de aptitud, para calcularlo se debe tener la aptitud para cada cromosoma. Una buena manera de hacerlo es evitar la división por cero usando $aptitud[i] = 1/1 + fobj(i)$, donde $i \in n$, con n el tamaño de la población.

Se calcula la sumatoria de las aptitudes y la probabilidad mencionada se calcula como $p(i) = aptitud(i)/total$. Para la selección se usa el método de la ruleta, por lo tanto se deben calcular las probabilidad acumulativas y por último en base a un número aleatorio entre $0 - 1$ se decide que cromosomas se usan para continuar con el algoritmo.

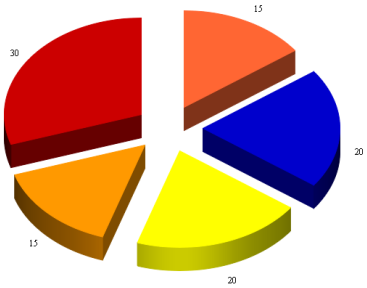

Figura 4.2: Proceso de selección

3. Cruza: Se usa cruza básica usando solo un punto de corte i.e., se selecciona una posición aleatoria entre dos cromosomas y se subintercambian. La cruza es controlada por la tasa de cruza.

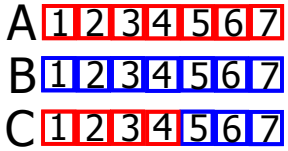

Figura 4.3: A con B se cruzan en la posición 4 y resulta C

4. Mutación: El número de cromosomas que se mutan en la población es determinado por la tasa de mutación. Uno o más genes de los cromosomas de la población se cambian de valor. En el problema de la mochila modelado consiste en negar el bit seleccionado.

   Una vez que se tenga un cromosoma con el mejor valor i.e., la entrada simplificada más pequeña que siga fallando, se reconstruye el código o las entradas.

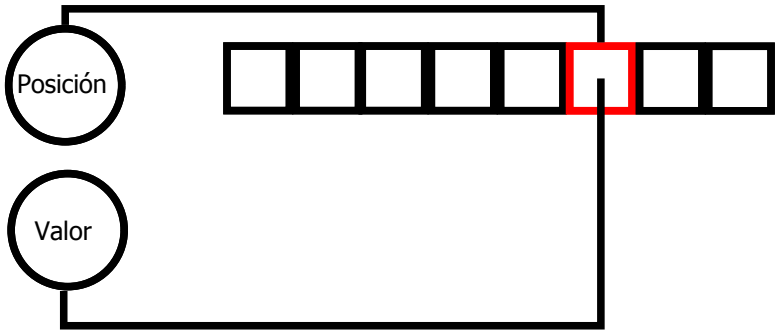

Figura 4.4: Proceso de mutación

Una nota importante es que las sentencias se rigen por un modelo de puntuación propuesto equivalente al volumen de cada artículo, aunque la ini-

cialización de los cromosomas es aleatoria, la declaración y asignación de variables son sentencias importantes, ya que sin ellas una versión de un programa no es sintácticamente correcta.

A continuación se muestra el algoritmo de la simplificación de casos de prueba.

Data: caso de prueba $c_{\text{✗}}$ test$(c_{\text{✗}}) = \text{✗}$
Result: $c'_{\text{✗}}$ tal que $c'_{\text{✗}} \subseteq c_{\text{✗}}$, test$(c'_{\text{✗}}) = \text{✗}$ y $c'_{\text{✗}}$ tenga minimalidad aceptable
Determinar el número de cromosomas, generaciones, tasa de mutación y cruza. ;
Generar la población inicial.;
while *generaciones* $\neq$ *No generaciones* **do**
    Evaluación del valor de aptitud para cada cromosoma usando una función objetivo;
    Selección de cromosomas.;
    Proceso de cruza.;
    Proceso de mutación.;
end
Entregar una lista o el mejor candidato.;
Algoritmo 4: Algoritmo genético de simplificación de casos de prueba.

Debido a que la inicialización, la selección, la cruza y la mutación requieren solo n pasos, donde n es el tamaño de la lista de sentencias, la complejidad es de $O(n)$. El factor de las generaciones y la población no afectan la complejidad.

4.4. Aislamiento de defectos de casos de prueba

Este algoritmo resuelve el mismo problema que *dd*. Básicamente este algoritmo ejecuta dos veces el algoritmo visto en la sección anterior, solo que usando dos objetivos, no al mismo tiempo la maximización y minimización. La idea es operar sobre el conjunto de datos obtenidos de la minimización y posteriormente la maximización, obteniendo una buena aproximación del aislamiento del defecto.

El algortimo es el siguiente.

1. Se opera el algoritmo de la sección anterior.
2. La salida es procesada por el paso 1 usando maximización como objetivo.

Data: caso de prueba $c_{\text{✗}}$ $\text{test}(c_{\text{✗}}) = \text{✗}$
Result: $c'_{\text{✗}}$ tal que $c'_{\text{✗}} \subseteq c_{\text{✗}}$, $\text{test}(c'_{\text{✗}}) = \text{✗}$ y $c'_{\text{✗}}$ tenga minimalidad aceptable
Se opera el algoritmo de la sección anterior.;
La salida es procesada por el paso 1 usando maximización como objetivo.;
Algoritmo 5: Algoritmo genético de aislamiento de casos de prueba.

La maximización y la minimización deben usarse en el orden descrito. estas operaciones no son conmutativas en este caso. El algoritmo de igual manera que en el caso de la simplificación opera linealmente respecto al conjunto de datos, osea $O(n)$.

Inicialmente se obtienen aleatoriamente la construcción de los datos, se pasan por el proceso de evaluación, selección, cruza y mutación. Esto al cabo de las generaciones determinadas culmina con una nueva mejor solución.

4.5. Sumario

En este capítulo se concluye que el uso de una heurística de búsqueda para el problema del aislamiento y la simplificación puede resultar muy práctica. Es un hecho que el tiempo total de ejecución es importante, aunque como es claro no resultan mejores soluciones que usando los algoritmos del capítulo anterior.

En realidad es posible el uso de otras formas de búsqueda. Es interesante ver el comportamiento de los algoritmos genéticos de este capítulo variando la taza de cruza y mutación. En muchos casos puede resultar igual de efectivo usar *ddmin* o *hdd*.

Capítulo 5

Experimentos

En este capítulo se muestran los resultados de los experimentos realizados. Se analizaron comandos básicos de Linux que mostraron algún error en tiempo de ejecución y programas especiales generados por una herramienta de programas aleatorios.

5.1. Diseño de experimentos

Todos los defectos reportados se encontraron utilizando la técnica de pruebas aleatorias. Para cada código fuente de los programas se utilizaron 3 algoritmos de simplificación: ddmin, hdd y el algoritmo propuesto en el capítulo anterior.

El número de pruebas es un factor muy importante en estos métodos. Mientras menos pruebas se realizan mejor. Esto depende de la naturaleza del error y del algoritmo elegido para analizar la entrada. Las pruebas son costosas. Cada una requiere reconstruir, recompilar, probar un programa, además de su procesamiento lógico y manejo de datos necesario.

Los programas analizados en estos resultados corresponden a algunos coman-

dos recolectados de paquetes de GNU como core utils, a2ps y groff [19] [20] [21]. Cada experimento requiere un fichero de configuración. Este fichero incluye mensajes del sistema operativo como el tipo de error ocurrido, el tiempo máximo de espera por ejecución, el consumo de memoria máximo que el programa a analizar puede ocupar, la ruta de los archivos compilados y la ruta del nuevo ejecutable, asi como los argumentos con los que se ejecuta.

5.2. Resultados

Las tablas que se muestran a continuación corresponden a los resultados experimentales. Cada experimento consistió de un programa con argumentos o ficheros de entrada específicos.

Los autores de ddmin muestran los resultados de su implementación en una herramienta llamada WYNOT usando el programa 5.1 [4]. Originalmente el programa contiene 755 caracteres. En las primeras dos pruebas la minimalidad resultante fue de 377 y 188 caracteres respectivamente. La gráfica 5.1 muestra el tamaño de la salida respecto al número de pruebas. Al final el algoritmo utilizó 733 pruebas hasta converger. De esta manera se observa una cantidad de pruebas excesivas debido a la granularidad usando caracteres. Se observa también que el uso de caracteres como unidad mínima de simplificación desencadena la pérdida del significado del programa, por ejemplo, el nombre de la función *mult* queda alterado.

Programa 5.1: Ejemplo de reducción

```
#define SIZE 20
double mult(double z[],int n)
{
        int i,j;
        i = 0;
        for(j = 0; j < n; j++){
                i = i + j + 1;
                z[i] = z[i] * (z[0] + 1.0);
        }
        return z[n];
}
```

```
void copy(double to[], double from[], int count)
{
        int n = (count + 7) / 8;
        switch(count % 8) do{
                case 0 : *to++ = *from++;
                case 7 : *to++ = *from++;
                case 6 : *to++ = *from++;
                case 5 : *to++ = *from++;
                case 4 : *to++ = *from++;
                case 3 : *to++ = *from++;
                case 2 : *to++ = *from++;
                case 1 : *to++ = *from++;

        }while(-- --n > 0);
        return mult(to,2);
}

int main(int argc, char * argv[])
{

        double x[SIZE], y[SIZE];
        double *px = x;

        while(px < x + SIZE)
                *px++ = (px - x) * (SIZE + 1.0);

        return copy(y, x, SIZE);
}
```

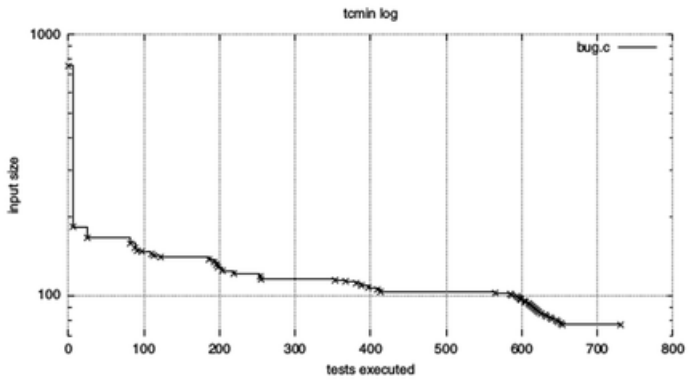

Figura 5.1: Número de pruebas vs tamaño de la entrada[4].

La granularidad en este trabajo se refiere a la unidad elegida para procesar los

datos y parsear el código fuente. Existen dos tipo de granularidad, la primera es usando caracteres (grano fino) y la segunda usando sentencias (grano grueso).

En este trabajo no se usaron caracteres como granularidad mínima, puesto que se requiere el análisis de las distintas funciones o métodos diseñados por los usuarios de la aplicación. En un sentido estricto, resulta en una mayor minimización pero se pierde precisión en el significado del programa para los programadores.

Los experimentos se dividen en dos bloques. El primer bloque consiste de comandos de linux que mostraron algún error en tiempo de ejecución. Se uso el paquete coreutils 6,10.

El segundo bloque de pruebas consiste de 5 programas especiales generados por Csmith. Csmith es una herramienta que genera programas aleatorios que conforman dinámica y estáticamente el estándar C99. Es utilizado para realizar pruebas de estrés en compiladores de C y otras herramientas que procesan código en C.

5.3. Análisis y comparaciones

A continuación se enlistan los experimentos realizados en este trabajo para el primer grupo de programas.

Experimento	Comando	Ejecución
e1	a2ps	a2ps −Efoo+ /dev/null
e2	tac	echo > x; tac −r x x
e3	pr	perl -e print a, bx100, t | pr e300 dev/null
e4	mkdir	mkdir −Z no−such−context DIR
e5	paste	./paste −d aaaaaaaaa
e6	cut	./cut −c1234567890− −−output−d=: foo
e7	dd	./dd if=/dev/cdrom of=cd1.iso
e8	expr	./expr "x": "[[:alpha:]]"
e9	sort	./sort < big-file | less
e10	grops	grops -c a
e11	lookbib	./lookbib -idata storage bibrefer
e12	groff	groff -man -html xxx > xxx.html
e13	pic	pic bug.pic

Cuadro 5.1: Lista de experimentos del primer bloque de programas

A continuación se enlistan los experimentos realizados en estre trabajo para el segundo grupo de programas.

Experimento	Comando	Ejecución
e14	csmith –bitfields -s 133910020	gcc t1.c -o t1 -O2
e15	csmith –bitfields -s 1014372711	gcc t2.c -o t2 -O3
e16	csmith –bitfields -s 2087907740	gcc t3.c -o t3 -Os
e17	csmith –bitfields -s 3230950649	gcc t4.c -o t4 -O3
e18	csmith –bitfields -s 253603242	gcc t5.c -o t5 -Ofast

Cuadro 5.2: Lista de experimentos del segundo bloque de programas

La tabla 5.3 muestra los resultados del primer bloque de programas utilizando el algoritmo ddmin. Se muestran el tiempo de ejecución y el número de pruebas realzidas. Observamos en todos los casos que nunca se llegó al peor caso en complejidad. El número de pruebas está alrededor del doble de sentencias en el código.

Experimento	Tiempo de ejecución (s)	Número de pruebas
e1	146	408
e2	53	133
e3	321	752
e4	37	98
e5	183	308
e6	112	393
e7	87	194
e8	84	208
e9	193	329
e10	31	76
e11	22	58
e12	30	125
e13	16	28

Cuadro 5.3: Evaluación del primer grupo de programas usando ddmin

La tabla 5.4 muestra los resultados de 5.3 en términos de minimalidad obtenida. El nivel de minimalidad no es mejor comparado con *hdd*. Esto es debido a que los errores detectados se encuentran en ciertos niveles anidados dentro del código.

Experimento	Tamano inicial (bytes)	Tamano final (bytes)	Minimalidad porcentual
e1	34897	12559	0.63
e2	18985	4730	0.75
e3	80091	23007	0.71
e4	6107	3395	0.44
e5	12164	2896	0.76
e6	23916	6530	0.71
e7	37715	8090	0.78
e8	17776	12948	0.26
e9	83814	12755	0.84
e10	45323	2415	0.94
e11	3347	737	0.77
e12	5703	4232	0.25
e13	11855	1392	0.88

Cuadro 5.4: Minimalidad obtenida usando ddmin

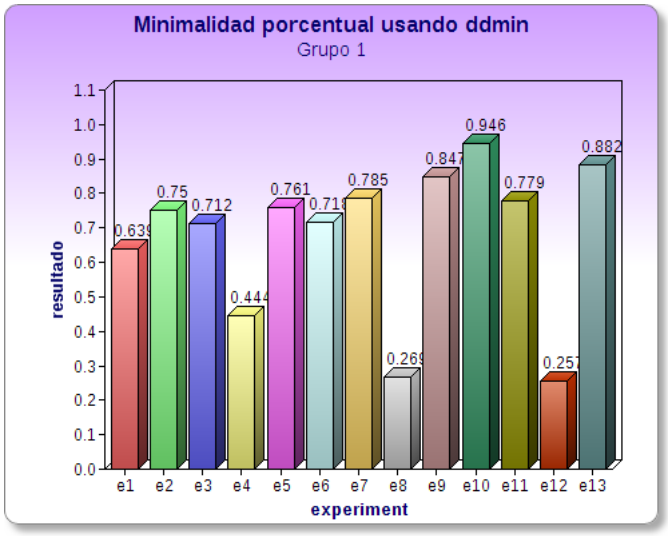

Figura 5.2: Minimalidad porcentual usando ddmin

La tabla 5.5 muestra los resultados del primer bloque de programas utilizando el algoritmo *hdd*. Se muestran el tiempo de ejecución y el número de pruebas realizadas.

Experimento	Tiempo de ejecución (s)	Número de pruebas
e1	104	206
e2	41	65
e3	229	272
e4	23	21
e5	57	44
e6	81	65
e7	66	40
e8	56	111
e9	124	286
e10	20	57
e11	17	41
e12	12	92
e13	5	11

Cuadro 5.5: Evaluación del primer grupo de programas usando hdd

Claramente se observan menos pruebas comparado con el método *ddmin* y por lo tanto el tiempo de ejecución total para cada prueba. El tiempo para generar el AST de cada programa no impacta en el tiempo de ejecución del algoritmo para efectos de comparación.

La tabla 5.6 muestra los resultados de 5.5 en términos de minimalidad obtenida. Además de obtener menor tiempo de ejecución se obtuvo mayor minimalidad. Esto significa que los defectos se pudieron aislar mejor. Este algoritmo genera mayor minimalidad que los otros dos.

Experimento	Tamano inicial (bytes)	Tamano final (bytes)	Minimalidad porcentual
e1	34897	11903	0.65
e2	18985	3980	0.79
e3	80091	20538	0.74
e4	6107	2906	0.52
e5	12164	1750	0.85
e6	23916	4403	0.81
e7	37715	7766	0.79
e8	17776	12033	0.32
e9	83814	12699	0.84
e10	45323	2150	0.95
e11	3347	590	0.82
e12	5703	3850	0.32
e13	11855	1204	0.89

Cuadro 5.6: Minimalidad obtenida usando hdd

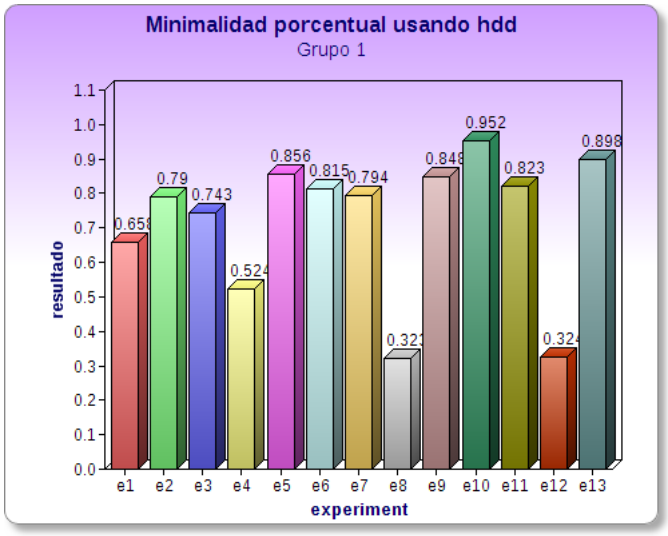

Figura 5.3: Minimalidad porcentual usando hdd

La tabla 5.7 muestra los resultados del primer bloque de programas utilizando el algoritmo genético propuesto en el capítulo anterior. Se muestran el tiempo de ejecución y el número de pruebas realizadas. Las pruebas se realizaron usando ún número fijo de generaciones y cromosomas.

Experimento	Tiempo de ejecución (s)	Número de pruebas
e1	38	120
e2	43	120
e3	65	120
e4	24	120
e5	51	120
e6	70	120
e7	75	120
e8	78	120
e9	160	120
e10	55	120
e11	48	120
e12	43	120
e13	48	120

Cuadro 5.7: Evaluación del primer grupo de programas usando un algoritmo genético

La tabla 5.8 muestra los resultados de 5.7 en términos de minimalidad obtenida. Estos experimentos se repitieron 5 veces. La media obtenida se muestra en la última columna, la desviación estándar de todo el bloque de experimentos fue de 3937 bytes.

Experimento	Tamano inicial (bytes)	Tamano final (bytes)	Minimalidad porcentual
e1	34897	15720	0.54
e2	18985	7123	0.62
e3	80091	50211	0.37
e4	6107	3366	0.44
e5	12164	4109	0.66
e6	23916	8939	0.62
e7	37715	10182	0.73
e8	17776	13991	0.21
e9	83814	19260	0.77
e10	45323	24402	0.46
e11	3347	1538	0.54
e12	5703	4848	0.14
e13	11855	3533	0.7

Cuadro 5.8: Minimalidad obtenida usando un algoritmo genético

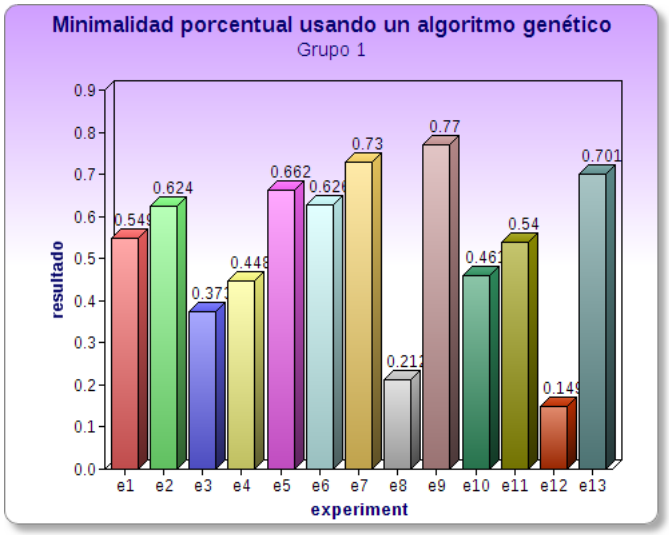

Figura 5.4: Minimalidad porcentual usando un algoritmo genético

El siguiente bloque de experimentos se realizó sobre programas generados por csmith.

En este grupo de experimentos el compilador gcc 4.0.0 y clang obtuvieron un error. Es decir aunque el programa generado cumple con el estándar c99 y es correctamente sintáctico no fue posible generar un binario.

Los programas son más cortos que en el grupo anterior. Para hacer romper al compilador se usaron banderas de optimización. Todas estas pruebas se realizaron usando un procesador intel core i3, gcc 4.0, y ubuntu 10.04.

La tabla 5.9 muestra los resultados del segundo bloque de programas utilizando el algoritmo ddmin.

Experimento	Tiempo de ejecución (s)	Número de pruebas
e14	206	118
e15	179	105
e16	225	123
e17	109	88
e18	206	96

Cuadro 5.9: Evaluación del segundo bloque de programas usando ddmin

Se observa que la minimalidad máxima oscila entre la quinta parte del programa original.

La tabla 5.10 muestra los resultados de 5.9 en términos de minimalidad obtenida.

Experimento	Tamano original (bytes)	Tamano final (bytes)	Minimalidad porcentual
e14	140227	27692	0.8
e15	25808	8767	0.66
e16	142397	29518	0.79
e17	76332	11542	0.84
e18	95217	17203	0.81

Cuadro 5.10: Minimalidad obtenida usando ddmin

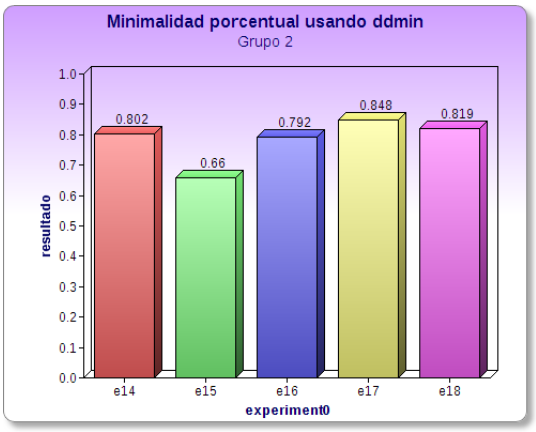

Figura 5.5: Minimalidad porcentual usando ddmin

Nuevamente el algoritmo *hdd* muestra usar menos pruebas.

La tabla 5.11 muestra los resultados del segundo bloque de programas utilizando el algoritmo ddmin.

Experimento	Tiempo de ejecución (s)	Número de pruebas
e14	125	76
e15	131	54
e16	116	70
e17	59	36
e18	161	44

Cuadro 5.11: Evaluación del segundo bloque de programas usando hdd

Nuevamente el algoritmo *hdd* muestra obtener la mayor minimalidad en ambos grupos de programas. La tabla 5.12 muestra los resultados de 5.11 en términos de minimalidad obtenida.

Experimento	Tamano original (bytes)	Tamano final (bytes)	Minimalidad porcentual
e14	140227	25500	0.81
e15	25808	7312	0.71
e16	142397	29042	0.79
e17	76332	9830	0.87
e18	95217	16331	0.82

Cuadro 5.12: Minimalidad obtenida usando hdd

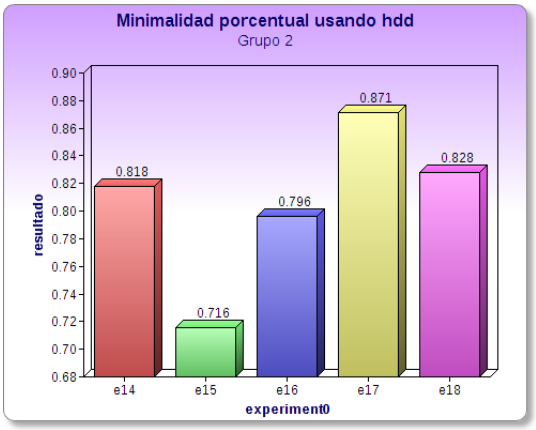

Figura 5.6: Minimalidad porcentual usando hdd

La tabla 5.13 muestra los resultados de aplicar el algoritmo genético. Se realizaron 60 pruebas y el experimento se repitió 5 veces.

La media obtenida de la minimalidad se presenta en la última columna, con desviación estándar en este grupo de 1031 bytes. La tabla 5.14 muestra los resultados de 5.13 en términos de minimalidad obtenida.

Experimento	Tiempo de ejecución (s)	Número de pruebas
e14	109	60
e15	97	60
e16	110	60
e17	88	60
e18	120	60

Cuadro 5.13: Evaluación del segundo bloque usando un algoritmo genético

Experimento	Tamano original (bytes)	Tamano final (bytes)	Minimalidad porcentual
e14	140227	35201	0.74
e15	25808	10191	0.6
e16	142397	33794	0.76
e17	76332	12650	0.83
e18	95217	18050	0.81

Cuadro 5.14: Minimalidad obtenida usando un algoritmo genético

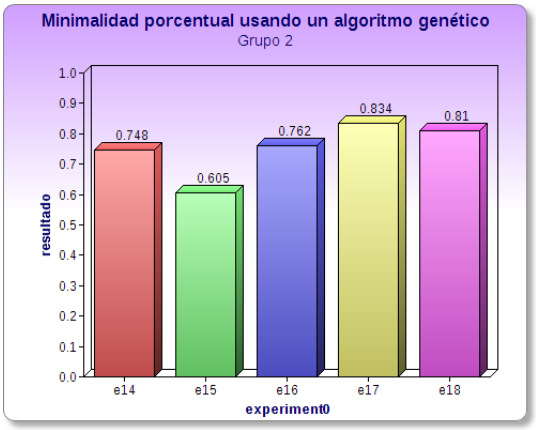

Figura 5.7: Minimalidad porcentual usando un algoritmo genético

5.4. Discusión

El aislamiento resulta más eficiente que la simplificación y esto se debe al uso de la cota superior e inferior (aumentar y a la vez reducir dos conjuntos de datos. i.e.el que falla y el que pasa las pruebas exitosamente). Esto implica que se requieran menos pruebas y por lo tanto un menor costo computacional. Sin embargo, por razones prácticas, resulta útil usar los dos enfoques combinados para entender mejor la causa de un error estudiado. Es preferible usar la simplificación para ayudar al programador, por tratarse de un conjunto pequeño pero con suficiente información.

Como se puede ver en los resultados, el algoritmo hdd resulta mejor que el algoritmo genético y ddmin en minimalidad. Es posible ver que la implementación de hdd es más complicada pues requiere hacer llamadas a ddmin. El algoritmo hdd presentado en este trabajo resuelve el problema de la simplificación de casos de prueba y es el más útil en la práctica, puesto que aprovecha la estructura jerárquica de la entrada (como el nivel de anidamiento en el código fuente). Los resultados mostraron en todos los casos que se ocupan menos pruebas en hdd que en ddmin y además se obtienen conjuntos con mayor minimalidad.

También ddmin puede aprovecharse y ser igual de eficiente que hdd solo con entradas planas o con pocos niveles de anidamiento cuando los errores se manifiesten en los niveles más generales del AST, con la ventaja de no parsear excesivamente el código fuente. Con este tipo de entradas en ddmin se obtiene mejor tiempo de ejecución. Los programas simples usan mas eficientemente el algoritmo ddmin.

El uso de recursión en ddmin puede llegar a generar consecuencias no deseadas. Recordando que cada llamada a ddmin necesita ejecutar un programa usando determinados archivos o argumentos de entrada. El algoritmo propuesto ayuda a mitigar este problema. Cualquier otro algoritmo de búsqueda discreta puede ser utilizado, como por ejemplo: búsqueda tabú, beam search, hill climbing, etc.

Conviene usar dd o el algoritmo genético cuando el código fuente tiende a ser

plano, tambien cuando se tengan recursos limitados de tiempo, poca memoria o para programas que usan entradas no estructuradas como arreglos o cadenas de caracteres. El algoritmo genético puede utilizar menos tiempo de ejecución.

Se usó gdb, así como un analizador dinámico como método de rectificación de un error. Cada programa usa un shell script que analizó mensajes del sistema operativo o de un oráculo. Los mensajes del sistema operativo fueron tales como fallo de segmentación, sobreflujo en la pila, o falla de aserciones. Valgrind es una herramienta de análisis dinámico que muestra la pila de llamadas y detección de errores . Por esta razón se utilizó como segundo criterio de elección de subconjuntos analizando trazas de ejecución.

Los experimentos y resultados sugieren que , al minimizar la entrada defectuosa se obtiene el beneficio de analizar más fácil y cómodamente la causa del error. Cuando se descubre que una entrada muy larga e.g., ordenamiento de números o letras falla, causa un efecto preocupante al observar una entrada de cientos o miles de elementos. Los resultados experimentales nos indican en todos los casos que las entradas no deben ser largas para mostrar el defecto. Por lo que se reduce el nivel de incertidumbre en las pruebas de Software reales.

Por último se enfatiza la dificultad de la creación de oráculos automáticos para muchos problemas de la vida real, ya que se realiza un programa para obtener una solución desconocida, sin embargo se pueden usar pruebas diferenciales o varias versiones de programas que entreguen mismos resultados para problemas deterministas, esto se hace creando más de dos oráculos para el mismo problema y se considera correcta aquella salida que resulte igual a los que entreguen la mayoría de los oráculos.

Por esta razón el uso de pruebas automáticas usando oráculos se limita a cierto tipo de problemas. En general los resultados muestran que usando estas herramientas adecuadamente resultan útiles.

Capítulo 6

Conclusiones

Las técnicas vistas en esta tesis resultan útiles en la práctica. La depuración es una técnica necesaria para los programadores. Se debe tener en cuenta que al Software siempre se le encuentran vulnerabilidades, las cuales deben ser atendidas. A futuro sería ideal que los entornos de desarrollo integrados se equipen con algún analizador y simplificador de defectos basados en depuración automática.

Las pruebas automáticas no solo se usan en el contexto de la depuración automática, de hecho se están adoptando en los enfoques de desarrollo.

Aunque los entornos modernos de desarrollo contienen herramientas especiales de depuración como el uso de bibliotecas especiales, la capacidad de depurar un programa concurrente, multihilo y la capacidad de depurar un programa en un proceso por separado o remotamente, la depuración automática puede ser una nueva herramienta robusta para los programadores.

Muchos programas dependen de circunstancias como el hardware, el tiempo de ejecución, la capacidad del cpu, del sistema operativo, etc. Muchas veces los errores no pueden ser detectados debido a que las técnicas de simplificación y aislamiento de defectos requieren que el mismo problema se repita. Por esta razón un trabajo a futuro deberá resolver este problema, i.e., poder encontrar circuns-

tancias importantes para la reproducción de un error que rara vez se manifieste.

Son útiles estos métodos de simplificación a problemas grandes, sin embargo, para errores de omisión no resultan ser útiles. Los algoritmos de este trabajo resultan especialmente aplicables a defectos que no se propagan en cadena e identificables para un programador experimentado. Se sugiere que se combinen con el uso de herramientas de análisis estático como model checking, ejecución simbólica o reducción de orden parcial.

Poder seleccionar entre varias formas de búsqueda es posible como se mencionó, así como usar pruebas aleatorias para detección de errores.

Cuando se descubre que una entrada muy larga e.g., ordenamiento de números o letras falla, causa un efecto preocupante al observar una entrada de cientos o miles de elementos. Los resultados experimentales nos indican en todos los casos que las entradas no deben ser largas para mostrar el defecto. Por lo que se reduce el nivel de incertidumbre en las pruebas de Software reales.

6.1. Trabajo a futuro

Las pruebas automáticas apuntan a formar parte de los futuros entornos de desarrollo y más aún la depuración delta como plugins. Actualmente hay herramientas basadas en depuración delta usando varios criterios de validación de errores. Mientras más criterios se usen, la búsqueda es más precisa.

Por último se enfatiza la dificultad de la creación de oráculos automáticos para muchos problemas de la vida real, ya que se realiza un programa para obtener una solución desconocida, sin embargo se pueden usar pruebas diferenciales o varias versiones de programas que entreguen mismos resultados para problemas deterministas, i.e., se crean más de dos oráculos para el mismo problema y se considera correcta aquella salida que resulte igual a los que entreguen la mayoría de los oráculos.

Referencias

[1] S. McConnell, *Code Complete: A Practical Handbook of Software Construction*. Microsoft Press, 2nd edition ed., 2004. 3, 4, 42

[2] H. Jaakkola, "Towards a globalized software industry," vol. 6, 2009. 3

[3] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?," *ISSTA*, 2011. 4, 5, 26

[4] Zeller and Hidelbrandt, "Simplifying and isolating failure-inducing input," *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, vol. 28, february 2002. 5, 37, 47, 48, 74, 75

[5] Misherghi and Su, "Hdd: hierarchical delta debugging," *ICSE '06 Proceedings of the 28th international conference on Software engineering.*, pp. 142–151, may 2006. 5

[6] Butcher, *Debug It!: Find, Repair, and Prevent Bugs in Your Code*. Pragmatic Bookshelf, 1st edition ed., 2009. 24, 28, 38

[7] Weiser, "Program slicing," *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING,*, vol. 4, July 1984. 28

[8] Yu, "Improving failure-inducing changes identification using coverage analysis," *ICSE 12*, pp. 1604–1606, June 2012. 29

[9] Groce, "What went wrong: explaining counterexamples," *SPIN 03*, pp. 121–136, 2003. 29

[10] W. et al., "From symptom to cause: Localizing errors in counterexample traces," *Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, vol. 53, May 2010. 30

[11] B. et al., "From symptom to cause: Localizing errors in counterexample traces," *POPL 03*, January 2010. 30

[12] O. et al., "Isolating relevant component interactions with jinsi," *WODA 06*, May 2006. 30

[13] G. et al., "Locating faulty code using failure-inducing chops," *ASE 05*, November 2005. 31

[14] W. et al., "Automatic program repair with evolutionary computation," *Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, vol. 53, pp. 470 – 471, May 2010. 32

[15] Z. et al., "Autoflow: An automatic debugging tool for aspectj software.," *IEEE International Conference on Software Maintenance*, vol. 4, pp. 470 – 471, July 2008. 32

[16] Zeller, "Automated debugging: Are we close," vol. 34, pp. 26–3q, August 2002. 35

[17] Kernighan and Pike, *The Practice of Programming*. Addison-Wesley, 1st edition ed., 1999. 48

[18] R. et al., "Test-case reduction for c compiler bugs," *SIGPLAN*, June 2012. 67

[19] Coreutils, "Gnu coreutils package," 2000. 74

[20] a2ps, "Gnu a2ps package," 2000. 74

[21] groff, "Gnu groff package," 2000. 74