Tesis de Posgrado

Análisis del flujo de programas para reducir y estimar el costo de los criterios de cubrimiento de Testing

Marré, Martina

1997

Tesis presentada para obtener el grado de Doctor en Ciencias de la Computación de la Universidad de Buenos Aires

Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Análisis del Flujo de Programas para Reducir y Estimar el Costo de los Criterios de Cubrimiento de Testing

Trabajo de Tesis presentado por
Martina Marré
para optar al título de Doctor de la Universidad de Buenos Aires

Lugar de Trabajo: Departamento de Computación, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires

Directora de Tesis: Antonia Bertolino

- Oiga, hace rato que quiero preguntarle: ¿ qué es eso de la informática?

- Programas para que las computadoras hagan lo que uno quiere.

- ¿ Y usted se piensa salvar con eso? - preguntó sorprendido.

- No creo. En Europa tenía una buena situación pero se me dio por volver. Y como usted dice, ya es un poco tarde.

Osvaldo Soriano, Una sombra ya pronto serás

Resumen

El testing de un sistema de software consiste en ejecutarlo sobre un conjunto de datos de input conveniente y chequear que el output producido es el esperado. El testing es ampliamente usado para mejorar la calidad del software, y específicamente, para descubrir errores inevitablemente introducidos durante el proceso de desarrollo del software.

Uno de los problemas más difíciles en testing es saber cuándo detener el proceso de prueba. Por un lado, no es posible en general dar una respuesta a si un test suite garantiza la ausencia de defectos. Por otro lado, necesitamos una manera de limitar el costo del testing. Por consiguiente, es útil tener criterios para determinar cuándo se ha testeado "suficientemente" un programa. Idealmente, el proceso de testing se debe planear de antemano. Sin embargo, en la práctica se incorporan los tests de a uno, hasta se satisface que algún *criterio de terminación* o *criterio de adecuación*. En particular, se pueden usar diferentes cubrimientos para determinar cuándo se ha probado un programa lo suficiente. La idea es garantizar que cada instrucción, decisión, u otra parte de un programa se ha ejecutado por lo menos una vez bajo la prueba realizada.

Un problema mayor es que el testing emplea una cantidad considerable del tiempo y recursos gastados en la producción de software [9]. Por consiguiente, sería útil:

1. *Reducir* el costo del testing, y

2. *Estimar* el costo del testing.

En particular, *el número de tests a ser ejecutados* impacta fuertemente en el costo del testing. De hecho, el tiempo y los recursos necesarios para realizar el testing aumenta con el número de casos de prueba. Entonces, para reducir el costo del testing, el número de casos de prueba generados para satisfacer un criterio de selección de casos debe ser lo más pequeño posible. Además, una cota en el número de tests que tienen que ser ejecutados para satisfacer una estrategia de selección de casos de test puede ser usada por gerentes y testers para estimar el esfuerzo necesario para llevar a cabo la ejecución de los tests.

En práctica, un criterio de testing define una colección de *requisitos* que deben cumplirse. En el caso de criterios de cubrimiento estructurales, estos requisitos se mapean en un conjunto de *entidades* del flowgraph (o grafo del flujo de control) del programa, que se deben cubrir cuando se ejecutan los datos de test.

En este trabajo presentamos un método para reducir y estimar el número de tests necesarios para satisfacer criterios de cubrimiento estructurales, basado en el concepto nuevo de *spanning*

sets of entities o conjunto expandido de entidades. Este concepto se basa en la observación de que un test generalmente cubre más de una entidad. Sin embargo, este hecho tradicionalmente no se considera cuando se mide el cubrimiento de los tests y se identifica las entidades no cubiertas, o cuando se seleccionan más tests para aumentar el cubrimiento. Los métodos existentes para seleccionar tests generan un dato para cubrir una entidad seleccionada en forma arbitraria, y considerada en forma aislada a las otras entidades. Si el dato de test generado también ejecuta otras entidades, se considerarán cubiertas a posteriori. Pero no se realiza ningún esfuerzo para generar a priori datos de test que satisfagan varios requisitos simultáneamente.

Nuestro método supera estos inconvenientes, ya que identifica mediante análisis estático un subconjunto de entidades *mínimo* con la propiedad de que cualquier conjunto de tests que cubre este subconjunto cubre toda entidad en el programa. Llamamos a este subconjunto mínimo un "spanning set of entities". En este trabajo, primero definimos los conjuntos de entidades que se asocian con cada criterio en una familia entera de criterios de cubrimiento de test, muy populares. Luego presentamos un método generalizado para identificar un spanning set of entities para los criterios considerados.

Nuestro método puede ser automatizado. Una vez que ha sido incluido en el proceso de testing, la información puede ser usada para:

- Evaluar la adecuación del testing más efectivamente.
- Reducir el costo del testing.
- Estimar el costo del testing.
- Generar test suites.

En esta tesis presentamos nuestro estudio del uso de spanning sets of entities en testing.

Abstract

Testing a software system consists of executing it over a suitable sample of input data and then checking if the output produced matches what was expected. Testing is widely used to enhance software quality, and, specifically, to uncover bugs that are inevitably introduced during the software development process.

One of the most difficult problems in testing is knowing when to stop the testing process. On the one hand, it is not possible in general to give an answer to whether a test suite guarantees the absence of faults. On the other hand, we need a way to limit the cost of testing. Therefore, it is useful to have criteria to determine when a program has been tested "enough". Ideally, the testing process should be planned in advance. However, in practice the tests are incorporated little by little, until some adequacy criterion is satisfied. In particular, different coverages can be used to determine when the program has been tested enough. The idea is to guarantee that each statement, decision or other feature of the program has been executed at least once under some test.

A major problem is that testing takes a considerable amount of the time and resources spent on producing software [9]. Therefore, it would be useful to have ways

1. to *reduce* the cost of testing, and

2. to *estimate* this cost.

In particular, *the number of tests to be executed* impacts heavily on the cost of testing. In fact, the time and resources needed for testing increase as the number of test cases increases. Hence, to reduce the cost of testing, the number of test cases generated to satisfy a selected test criterion should be as small as possible. Moreover, a bound on the number of tests that have to be performed to satisfy a selected test strategy can be used by managers and testers to estimate the effort needed to carry out the tests.

A test criterion in practice sets a collection of *requirements* to be fulfilled. For structural coverage criteria, these requirements are mapped onto a set of *entities* in the program flowgraph that must be covered when the tests are executed.

In this work we present a method for reducing and estimating the number of tests needed to satisfy structural coverage criteria, based on the new concept of *spanning sets* of entities. This concept is based on the observation that one test generally covers more than one entity. However, this fact is not traditionally considered when coverage is measured and not covered entities are

identified, or when more tests have to be selected in order to augment coverage. Methods for selecting tests generate a test datum for covering one entity selected arbitrarily and considered in isolation from the other entities. If the generated test datum also exercises other entities, these will then be considered as covered a-posteriori. But no effort is made to generate a-priori test data that satisfy multiple requirements.

Our method overcomes these drawbacks by identifying with static analysis a *minimum* subset of entities with the property that any set of tests covering this subset covers every entity in the program. We call this minimum subset a "spanning set of entities". In this work, we first define the sets of entities that are associated with a whole family of popular test coverage criteria. Then we present a generalized method of identifying a spanning set of entities for the criteria considered.

Our method can be automated. Once it has been included in the software testing process, this information can be used for

- evaluating test adequacy more effectively;
- reducing the cost of testing;
- estimating the cost of testing;
- generating test suites.

In this thesis we present our study of the use of spanning sets of entities in coverage testing.

Contenidos

Capítulo 1

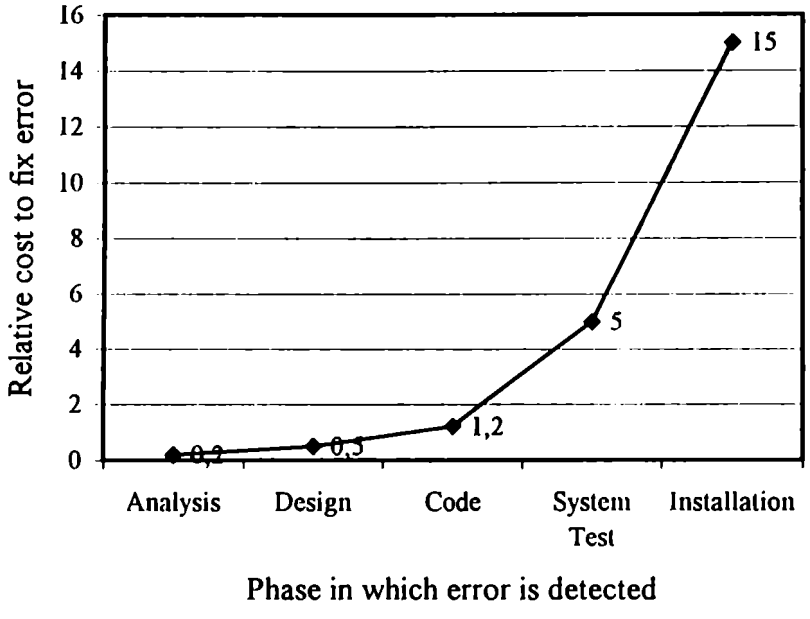
Introducción

> Beware of bugs in the above code; I have only proved it correct, not tried it.
>
> *Donald Knuth*

Cuando el producto de cualquier actividad de la ingeniería (por ejemplo, un puente, una plancha, hardware, software, etc.) es obtenido, queremos saber o garantizar que efectivamente funciona como deseamos. El proceso seguido para adquirir este conocimiento es llamado *verificación y validación* (V&V).

Desde la concepción de una idea de un sistema del software hasta que se entrega el producto e inclusive después de dicha entrega, el sistema sufre un desarrollo y una evolución graduales. Así, se dice que el software tiene un *ciclo de vida*, que consta de varias fases, relacionadas con especificación del software, el diseño del software, la implementación del software y la verificación del software, entre otros [33]. Los varios modelos del proceso de desarrollo del software existentes coinciden en considerar V&V como una parte esencial de este proceso. V&V se hace para verificar la equivalencia entre el producto y su especificación (normalmente conocido con el nombre de verificación) y para saber si el producto efectivamente trabaja como el cliente desea (normalmente, conocido con el nombre de validación).

En el modelo del proceso de desarrollo de software tradicional, la V&V de un sistema es una fase independiente confinada después de la integración y anterior al mantenimiento. Ya que esta fase tiene como objetivo encontrar problemas en el sistema, no tiene sentido comenzarla recién cuando se termina la implementación del sistema. Además, pueden resultar consecuencias severas al usar este método, ya que cuanto más tarde se encuentra un error, más alto es el costo de su



Figuras 1.1: Costo Relativo y Fase del Desarrollo del Software

corrección (ver Figura 1.1[1]). Además, varios estudios han mostrado que la mayoría de los errores en un sistema ocurren en las fases de análisis y diseño [69]. En la actualidad, los investigadores están de acuerdo en que V&V debe ser un proceso continuo, que comienza con una revisión de los requerimientos, y continúa a través de cada una de las fases del desarrollo de software.

Existen varias técnicas *estáticas* y *dinámicas* de V&V para sistemas del software (o programas). Las técnicas estáticas se relacionan con el *análisis* de representaciones del sistema tal como requisitos, diseño, y codificación del programa. Tales técnicas incluyen *walkthrough del código*, *métodos del análisis estático* (por ejemplo, chequeo de tipos), *inspecciones de código*, y *pruebas de corrección* [3, 76]. Las técnicas dinámicas o *testing* involucran la ejecución de la implementación del sistema. El testing es la técnica de verificación predominante usada en producción real de software.

Las técnicas estáticas son útiles para verificar la correspondencia entre un programa y su especificación (por ejemplo, demostrando la corrección de un programa con respecto a su especificación). Estas técnicas descubren algunos errores temprano en el proceso de desarrollo del

[1] Información recogida de [17], Resumen de IBM, GTE, y TRW.

software (por ejemplo, mediante una inspección de la especificación). Permiten eliminar categorías enteras de errores, a través de chequeo de sintaxis, strong typing, y chequeo de tipos, por ejemplo. Sin embargo, las técnicas estáticas no pueden demostrar que el software es operacionalmente útil, y tienen que ser usadas en combinación con testing.

El testing de programas se usa para verificar el comportamiento de un programa, y es esencial para encontrar errores[2], mejorar calidad, y proporcionar confianza en la calidad del programa. Un sistema del software puede testearse para alcanzar uno de los siguientes dos objetivos [26]:

1. Encontrar tantos defectos como sea posible en el sistema.

2. Evaluar la confiabilidad del sistema.

El segundo objetivo normalmente se refiere a testear el programa con el objetivo de predecir la conducta futura del sistema a partir de ejemplos pasados [26], o a establecer cierta probabilidad de corrección [38]. En general, se usan técnicas estadísticas para evaluar la confiabilidad de un sistema.

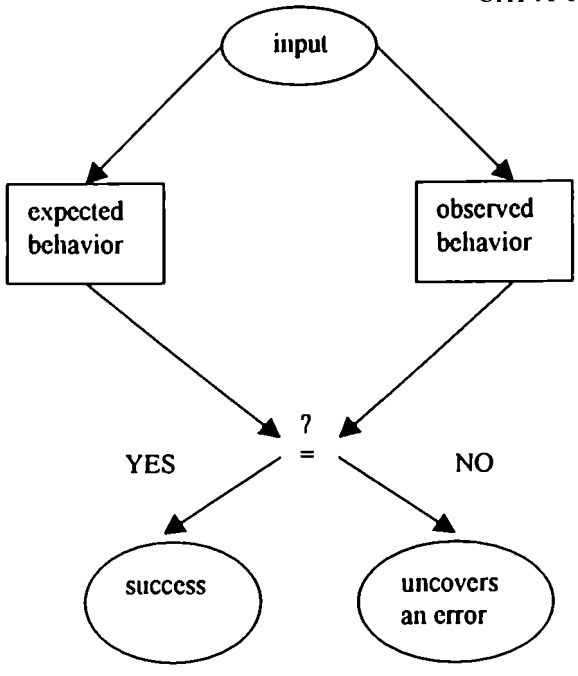
Esta tesis estudia el proceso de testing con el objetivo de encontrar defectos en un sistema (objetivo 1 más arriba).

Hay un acuerdo creciente en que un sistema de software debe satisfacer distintas calidades como robustez, corrección funcional, eficiencia, adaptabilidad, y confiabilidad. Distintos proyectos ponen énfasis en calidades diferentes, y de acuerdo con esto tienen diferentes requisitos de V&V. Toda técnica de V&V se debe ver como una componente importante entre los métodos que se deben utilizar para verificar que un producto es correcto. La manera en la cual se deben mezclar depende del proyecto en particular.

El testing y las técnicas estáticas previenen tipos diferentes de errores. No hay una manera única de realizar V&V. La meta de V&V es determinar y asegurar que un producto de software es de "alta calidad", pero los investigadores están de acuerdo en que no existe una noción fija y absoluta de "calidad" del software [1].

Se puede obtener una verificación completa de un programa en cualquier fase del desarrollo de software ejecutando un test por cada elemento del dominio, es decir, por cada input posible. Para un input específico, un test determina el comportamiento esperado del programa con ese input, ejecuta el programa, y observa el comportamiento real. Finalmente, compara el comportamiento real con el comportamiento esperado (ver Figura 1.2). Si coinciden, la instancia de test ha tenido

[2]Según las definiciones proporcionadas por el Standard IEEE/ ANSI [47], un *defecto* es un paso, proceso o definición de datos incorrecto en un programa; un sistema del software revela una *falla* si no hace lo que esperamos; un *error* es la acción humana que produce un resultado incorrecto.



Figuras 1.2: Testing de un input

éxito; de otra manera, se descubrió un defecto [3]. Si cada instancia posible (es decir, cada input posible del programa) tiene éxito, se verifica el programa; en otro caso, se asegura que hay un defecto en el programa.

Este método de testing exhaustivo es la única técnica de análisis dinámica que garantizaría la validez del programa. Desgraciadamente, esta técnica no es práctica. Frecuentemente, los dominios son infinitos o por lo menos suficientemente grandes como para hacer que el número de pruebas requeridas no sea factible [46].

Así, idealmente, el testing debe garantizar la corrección del sistema, pero en la mayoría de casos esto es imposible de alcanzar a través del testing de un programa en un conjunto finito de datos [46]. Por consiguiente, el testing de un programa consiste en la validación del programa a través de la selección de un subconjunto significativo de entre todas las ejecuciones posibles del programa, y en la verificación de que los outputs correspondientes son consistentes con lo que la especificación dice. El subconjunto de inputs seleccionado para testear el programa se llama *conjunto de datos de test* o *test suite*. El test suite debe ser bastante grande como para cubrir todo el dominio, pero a la vez lo suficientemente pequeño de manera de que el proceso de testing

[3] Algunos autores [63] dicen que un test input *tiene éxito* si descubre un defecto, y en otro caso, dicen que el test input *falla*.

se puede ejecutar para cada elemento en el test suite [45].

Para reducir el (potencialmente infinito) proceso de testing exhaustivo a un proceso factible, se puede guiar la selección de datos de test usando diferentes *estrategias* o *criterios* para seleccionar elementos representativos del dominio.

Una estrategia de testing agrupa en una clase aquellos inputs con "conducta similar"; es decir, se divide el dominio de la entrada en subdominios. La idea es que, para los propósitos del testing, todos los elementos en una clase o subdominio son esencialmente el mismo. En otro palabras, asumimos que todos los elementos en la clase revelarán los mismos defectos. De esta manera, se puede escoger un único input como un representante de cada clase.

Una vez que se divide el dominio de input en subdominios (posiblemente solapados), los criterios de selección requieren testear el programa usando por lo menos un elemento representativo de cada subdominio [84]. Los subdominios son determinados por casos de test. Los *casos de test* son predicados sobre el dominio de inputs. Un input seleccionado como el elemento representativo de un subdominio es un *dato de test.*

La subdivisión del dominio puede basarse en las funcionalidades o en la estructura de un programa. No existe ninguna controversia entre el uso de técnicas de testing funcional versus técnicas de testing estructurales: ambas son útiles, ambas tienen limitaciones, ambas atacan tipos diferentes de errores [9, 64].

En *testing funcional* [44, 68, 8] se mira al programa como una caja negra. Los datos de test se derivan a partir de la especificación del programa, con el objetivo de comparar la conducta real del programa con la conducta especificada, sin tener en cuenta la implementación del programa.

En *testing estructural* (o testing de la caja blanca, o glass box testing) la estructura del programa es examinada para analizar la consistencia de la implementación de una componente con sus requerimientos. El test suite trata de cubrir determinados rasgos predefinidos del flujo de control del programa (por ejemplo, instrucciones, decisiones) o del flujo de datos (es decir, relaciones entre una definición y un uso de una variable). Las técnicas basadas en el flujo de control y en el flujo de datos son técnicas de testing esenciales y complementarias. El testing estructural es probablemente la más extensamente usada clase de testing de programa [64]. La popularidad de estas técnicas se debe principalmente a su simplicidad y la disponibilidad resultante de herramientas de software para asistir al proceso de testing.

Una de las mayores deficiencias de las estrategias estructurales es que muchos defectos a lo largo de un camino de ejecución pueden descubrirse sólo si se ejecuta un programa con elementos específicos de su subdominio. Sin embargo, se debe notar que las estrategias que totalmente

descuidan la estructura del programa puede ser igualmente ineficaces para descubrir otros tipos de defectos (como por ejemplo, errores en la codificación específicos de una implementación particular que no aparecen en las especificaciones).

Esta tesis se centra en técnicas de testing estructurales, basadas en flujo de control y en flujo de datos.

Desgraciadamente, a la mayor parte del testing software aún le falta fundamentos teóricos. La relación entre un método de testing y las propiedades del software que se testea de acuerdo a ese método es todavía imprecisa. Los primeros trabajos, realizados por Goodenough y Gerahart [34] examinaron el papel teórico y práctico del testing en el desarrollo del software. Ellos propusieron una teoría que muestra que el testing puede establecer la ausencia de errores en un programa si los criterios de selección de testing satisfacen las propiedades correctas. Weyuker y Ostrand [84] extendieron y refinaron esta teoría. Sin embargo, se ha mostrado que no existe ningún algoritmo para encontrar criterios de testing que sean consistentes, confiables, válidos y completos [46]. En otras palabras, determinar si un test suite es realmente suficiente para demostrar la corrección de un programa es un problema indecidible. De hecho, el testing puede mostrar la presencia de errores, pero nunca su ausencia [23]. La teoría de testing proporciona algunos resultados de indecidibilidad [81] que definen los límites del testing. Sin embargo, ningún resultado ayuda a establecer qué o cuánto una técnica de testing particular verifica. Así, se debe desarrollar una teoría válida de testing de software que relacione propiedades de métodos de testing del software con la calidad real del software.

La falta de modelos aceptados para medir la efectividad del testing y su costo produjo un retardo en la evaluación de la efectividad de las estrategias de testing y en la comparación de su poder relativo para encontrar defectos. De hecho, relativamente poco trabajo se ha centrado en estudiar cuán buenos son para exponer defectos test suites que han sido seleccionados usando diferentes criterios de selección. Sólo en los últimos años, el problema de cómo las estrategias de testing de software se comparan una con otra en cuanto a su habilidad para exponer defectos [85] se ha realizado en forma experimental [30], mediante simulaciones [25, 37], y analíticamente [32, 82, 39].

Uno de los problemas más difíciles en testing es saber cuándo detener el proceso de prueba. Por un lado, no es posible en general dar una respuesta a si un test suite garantiza la ausencia de defectos. Por consiguiente, es útil tener criterios para determinar cuándo se ha testeado "suficientemente" un programa. Por otro lado, necesitamos una manera de limitar el costo del testing. De hecho, si pudieramos disponer de una cantidad ilimitada de recursos podríamos hacer

todo el testing que quisiéramos. En proyectos reales de software, siempre hay problemas de tiempo o dinero. Por consiguiente, necesitamos una manera de saber cuándo detener el proceso de testing.

Idealmente, el proceso de testing se debe planear de antemano. Sin embargo, en la práctica se incorporan los tests de a uno, hasta se satisface que algún *criterio de terminación* o *criterio de adecuación*. Desgraciadamente, el criterio de adecuación generalmente usado es *parar cuando el tiempo (o presupuesto) asignado para el testing termina* [63, 78]. En teoría un criterio de adecuación debe estar relacionado con alguna *frecuencia de errores* [62] o algún *cubrimiento* [35]. En el primer caso, se usan varias técnicas diferentes para determinar estimados estadísticos del número de defectos restantes en el programa. La idea básica es que cuánto más tiempo se prueba un programa sin observarse una falla, más grande es la probabilidad de que el producto esté libre de defectos. En el segundo caso, se usan cubrimientos diferentes para determinar cuándo se ha probado un programa lo suficiente. La idea es garantizar que cada instrucción, decisión, u otra parte de un programa se ha ejecutado por lo menos una vez bajo la prueba realizada.

Esta tesis se centra en criterios de cubrimiento de testing.

Salvo en el caso de sistemas pequeños, los sistemas no deben testearse como un todo. Los sistemas grandes deben ser construidos a partir de sub-sistemas, que se construyen a partir de módulos, que se componen de procedimientos y funciones. El proceso de testing debe proceder por consiguiente en fases, llevadas a cabo en forma incremental junto a la implementación del sistema [76, 86]. Hay tres fases distintas de testing de un sistema de software: *testing de unidad*, *testing de integración*, y *testing de sistema*. El testing de unidad se hace para mostrar que un módulo codificado satisface la especificación de la unidad. El testing de integración se lleva a cabo sobre grupos de módulos para asegurar que datos y control se pasan en forma correcta entre módulos. El testing de sistema se hace para verificar que el sistema entero (es decir, totalmente integrado) en su ambiente real de ejecución se comporta según la especificación de los requisitos del software.

Además, cuando se cambia un programa que está en funcionamiento (es decir, una vez entrado en producción), o cuando se cambia porque se ha encontrado una falla durante el testing, debemos verificar que los cambios requeridos se han llevado a cabo correctamente y debemos estar seguros que no se hicieron cambios no deseados. Así, una vez implementados los cambios deseados, se debe testear el programa contra datos de test previos, de manera de garantizar que la funcionalidad del resto del programa no se ha modificado. Este procedimiento se llama *testing de regresión* [76, 87].

Esta tesis se centra en testing de unidad.

Un problema mayor es que el testing emplea una cantidad considerable del tiempo y recursos gastados en la producción de software [9]. Por consiguiente, sería útil:

1. *Reducir* el costo del testing, y
2. *Estimar* el costo del testing.

En particular, *el número de tests a ser ejecutados* impacta fuertemente en el costo del testing. De hecho, el tiempo y los recursos necesarios para realizar el testing aumenta con el número de casos de prueba. Entonces, para reducir el costo del testing, el número de casos de prueba generados para satisfacer un criterio de selección de casos debe ser lo más pequeño posible. Además, una cota en el número de tests que tienen que ser ejecutados para satisfacer una estrategia de selección de casos de test puede ser usada por gerentes y testers para estimar el esfuerzo necesario para llevar a cabo la ejecución de los tests.

En práctica, un criterio de testing define una colección de *requisitos* que deben cumplirse. En el caso de criterios de cubrimiento estructurales, estos requisitos se mapean en un conjunto de *entidades* del flowgraph (o grafo del flujo de control) del programa, que se deben cubrir cuando se ejecutan los datos de test. Estas entidades pueden derivarse a partir del flujo de control del programa o del flujo de datos del programa. En este trabajo, presentamos un método para reducir y estimar el número de tests necesarios para satisfacer criterios de cubrimiento estructurales, basado en el concepto nuevo de *spanning sets of entities* o conjunto expandido de entidades.

Este concepto se basa en la observación de que un test generalmente cubre más de una entidad. Sin embargo, este hecho tradicionalmente no se considera cuando se mide el cubrimiento de los tests y se identifica las entidades no cubiertas, o cuando se seleccionan más tests para aumentar el cubrimiento. Los métodos existentes para seleccionar tests generan un dato para cubrir una entidad seleccionada en forma arbitraria, y considerada en forma aislada a las otras entidades [36]. Si el dato de test generado también ejecuta otras entidades, se considerarán cubiertas a posteriori. Pero no se realiza ningún esfuerzo para generar a priori datos de test que satisfagan varios requisitos simultáneamente.

Nuestro método supera estos inconvenientes, ya que identifica mediante análisis estático un subconjunto de entidades *mínimo* con la propiedad de que cualquier conjunto de tests que cubre este subconjunto cubre toda entidad en el programa. Llamamos a este subconjunto mínimo un "spanning set of entities". En este trabajo, primero definimos los conjuntos de entidades que se asocian con cada criterio en una familia entera de criterios de cubrimiento de test, muy populares.

Luego presentamos un método generalizado para identificar un spanning set of entities para los criterios considerados.

Recientemente, varios estudios empíricos [91, 71, 89] analizaron el efecto de reducir el tamaño de un conjunto de tests en el descubrimiento de fallas. Esto se realizó manteniendo el cubrimiento constante, y para diferentes criterios de cubrimiento. Los resultados muestran que al minimizar el conjunto de tests, para los programas considerados, se produce poca o ninguna reducción en la efectividad para descubrir fallas. Estos casos de estudio parecen indicar que tratar de minimizar el número de tests generados para alcanzar cubrimientos diferentes, como hacemos nosotros con los spanning set of entities, puede ayudar a reducir el costo del testing sin dañar la eficacia de la verificación.

Nuestro método puede ser automatizado. Una vez que ha sido incluido en el proceso de testing, la información puede ser usada para:

Evaluar la adecuación del testing más efectivamente: Sólo las entidades todavía no cubiertas en un spanning set son efectivas para aumentar el cubrimiento. Ya que un cubrimiento del 100% de un spanning set garantiza cubrimiento de todas las entidades, el porcentaje de entidades no cubiertas en un spanning set es lo que realmente importa: las entidades que no están en el spanning set no necesitan ser consideradas: de hecho, serán cubiertas por aquellos tests que son necesarios para garantizar el cubrimiento total del spanning set.

Reducir el costo del testing: La generación de casos de test puede ser dirigida a cubrir el (mínimo) spanning sets of entities, en vez de todas las entidades en el programa, minimizando así el número de tests y previniendo la selección de caminos redundantes.

Estimar el costo del testing: El número de entidades en un spanning set of entities puede ser usado como un estimado del número de tests necesarios para satisfacer un criterio de cubrimiento elegido.

Generar test suites: Pueden seleccionarse test paths, de manera de cubrir las entidades en un spanning set of entities aún no cubiertas.

En esta tesis presentamos nuestro estudio del uso de spanning sets of entities en testing. En [11, 12, 14], hemos identificado spanning sets of entities para el criterio de cubrimiento *branch* y hemos presentado algunas aplicaciones útiles. En [57], hemos identificado spanning sets of entities para el criterio *all-uses*. En [56], hemos generalizado estos resultados para una familia muy popular de criterios de cubrimiento de testing. En [12] hemos presentado un algoritmo para

construir un conjunto de caminos que cubre un spanning set of entities para el criterio branch. Este algoritmo puede adaptarse para construir conjuntos de caminos que cubren un spanning set of entities para un criterio de cubrimiento genérico. Esta extensión fue presentada en [13].

No hemos podido identificar ningún otro trabajo existente que identifique conjuntos de entidades mínimos que garanticen cubrimiento total para la familia de criterios considerada. Algunos autores [19, 4] han reconocido esta idea en forma independiente, para criterios de cubrimiento tales como branch y *statement*, pero no han generalizado la idea a otros criterios.

Sin embargo, algunos trabajos se han realizado para reducir el tamaño de un test suite. En particular, Gupta y Soffa [36] investigaron maneras de guiar la generación de tests, de manera de que un dato de test satisfaga más de un requisito. Ellos agrupan requisitos, de manera de que cada grupo pueda ser cubierto (potencialmente) por un único dato de test. Nuestro resultado mejora esta idea, ya que los spanning sets of entities proveen la manera óptima de agrupar entidades[4], ya que es el mínimo conjunto de entidades que garantiza cubrimiento total.

Una visión diferente consiste en minimizar el número de tests en un test suite. Ya que este problema es NP-completo [36], algunos autores propusieron el uso de heurísticas basadas en técnicas de minimización, como por ejemplo, [40, 91]. Sin embargo, estas técnicas se aplican luego de la generación del test suite. Por lo tanto, estos trabajos en realidad no reducen el esfuerzo de generar los tests.

Las estrategias de flujo de control y de flujo de datos analizan la estructura del programa usando una representación gráfica del mismo, llamada *flowgraph*, y toda la información usada para seleccionar casos de prueba está implícita en él. En el Capítulo 2 introducimos un modelo de flowgraph llamado *ddgraph* que usamos en este trabajo.

Como ya hemos dicho, un criterio de test determina un conjunto de entidades que debe ser cubierto para satisfacerlo. En el Capítulo 3 (re)definimos una familia muy popular de criterios de cubrimiento del flujo de control y del flujo de datos [9, 73, 31] identificando, para cada criterio, el correspondiente conjunto de entidades. Además, introducimos la noción de cubrimiento asociada a cada criterio. En realidad, la frase "cubrir una entidad" asume diferentes significados dependiendo del tipo de entidad considerada: definiremos esto en forma explícita.

En el Capítulo 4 introducimos primero los spanning sets of entities. Luego presentamos un método para identificar un spanning set of entities para las estrategias presentadas en el Capítulo 3, es decir, un método para encontrar un mínimo conjunto de entidades que garantiza cubrimiento

[4] Nosotros no hablamos explícitamente de grupos de entidades en nuestro trabajo, pero estos grupos pueden obtenerse fácilmente a partir del orden impuesto por la relación *subsumption* introducida en el Capítulo 4.

total. Este método se basa en las nociones de *subsumption* y *unconstrainedness*.

En los tres capítulos siguientes presentamos varias aplicaciones de los spanning sets of entities. Dividimos a las aplicaciones en tres partes:

1. En el Capítulo 5 estudiamos el uso de los spanning sets para *reducir* el costo del testing y prevenir la generación de caminos redundantes.

2. En el Capítulo 6 estudiamos el uso de los spanning sets para *estimar* el costo del testing en términos del número de caminos de test necesarios para obtener cubrimiento total.

3. En el Capítulo 7 presentamos el uso de los spanning set of entities para la *generación* de caminos de test.

Por último, en el Capítulo 8 presentamos las conclusiones de esta tesis.

Capítulo 2

Modelo para el Análisis de un Programa

En este capítulo se presenta el modelo para el análisis de un programa usado en esta tesis.

Las estrategias de flujo de control y de flujo de datos analizan la estructura del programa usando una representación gráfica del mismo, llamada *flowgraph*, y toda la información usada para seleccionar casos de prueba está implícita en él. El flujo de control de un programa puede mapearse en un flowgraph de diferentes maneras. En este trabajo usamos una representación llamada *ddgraph* (por decision-to-decision graph) [12]. En la primera parte de este capítulo introducimos este modelo.

El análisis del flujo de datos considera las posibles interacciones entre las definiciones y los usos de las variables del programa analizado. Para esto, el programa es representado usando un flowgraph etiquetado. En este trabajo, usamos ddgraphs etiquetados, llamdos *def-use ddgraphs* [57]. En la segunda parte de este Capítulo introducimos def-use ddgraphs.

2.1 El Modelo Ddgraph

En esta sección se usan conceptos básicos de la teoría de grafos (ver [10]), y de flowgraphs (ver [41]).

Un *grafo dirigido* o *digrafo* $G = (N, A)$ consta de un conjunto de *nodos* o *vértices* N y de un conjunto de *ejes dirigidos* o *arcos* A, donde un arco $e = (TAIL(e), HEAD(e))$ es un par ordenado de nodos *adyacentes*, llamado *Tail* y *Head* de e, respectivamente. Decimos que e *sale de* $TAIL(e)$

y *entra en* $HEAD(e)$. Si $HEAD(e) = TAIL(e')$, e y e' dicen arcos *adyacentes*. Para un nodo n en N, *indegree*(n) es el número de arcos que entra en él y *outdegree*(n) es el número de arcos que sale de él. Un nodo sin arcos que salen de él se dice una *hoja*.

El digrafo $G' = (N', A')$ es un *subgrafo* del digrafo $G = (N, A)$ si $A' \subseteq A$ y, para todo arco $e \in A'$, $HEAD(e), TAIL(e) \in N' \subseteq N$.

Un *camino* p de longitud q en un digrafo G es una secuencia $p = e_1, e_2, ..., e_q$, donde $TAIL(e_{i+1}) = HEAD(e_i)$ para $i = 1..., q-1$. p es entonces un camino de e_1 a e_q, o de $TAIL(e_1)$ a $HEAD(e_q)$. Si existe un camino desde el arco e (desde el nodo n) hasta el arco e' (hasta el nodo n'), entonces decimos que e (n) *alcanza* e' (n'). Si en un grafo no existe ningún camino desde el arco e hasta el arco e', ni existe ningún camino desde el arco e' hasta el arco e, decimos que e y e' son *incomparables*. Un camino p es *simple* si todos sus nodos, excepto tal vez el primero y el último, son diferentes. Un camino es *loop-free* si todo nodo es visitado a lo sumo una vez. Un digrafo se dice *acíclico* si cada camino en él es loop-free.

Dado un camino $p = e_1, e_2, ..., e_q$ en un ddgraph G, una secuencia de arcos $p' = e_{i_1}, ..., e_{i_r}$ en G se dice un *pseudo-camino* (o *pseudo-path*) de p si $\{i_1, ..., i_r\} \subseteq \{1, ..., q\}$ y $j = 1, ..., r-1 : i_j < i_{j+1}$. Notamos que un pseudo-camino no es necesariamente un camino; es decir, dos arcos consecutivos en un pseudo-camino pueden no ser adyacentes en el ddgraph. Si un pseudo-camino $p' = e_i, ..., e_j$ de p es un camino desde e_i hasta e_j, entonces se llama *subcamino* (o *subpath*) de p.

Una *componente fuertemente conexa* (o *strongly connected component*) en un digrafo G es un subgrafo G', donde cada arco o nodo puede alcanzar todo otro arco o nodo[1].

Un árbol (con raíz) (o *(rooted) tree*) $T = (N, A)$ es un digrafo con un nodo distinguido, llamado *raíz* (o *root*), al cual no llega ningún arco, tal que llega exactamente un arco a cada nodo excepto a la raíz, y existe un (único) camino desde la raíz a cada uno de los nodos. Sea $e = (n, m)$ un arco en T, entonces n es el *padre* (*parent*) de m y m es un *hijo* (*child*) de n.

2.1.1 Ddgraphs

Los digrafos son ampliamente usados para representar el flujo de control de un programa. En este caso, son llamados *flowgraphs*.

El flujo de control de un programa puede mapearse en un flowgraph de diferentes maneras. En el este trabajo usamos una representación llamada *ddgraph* (por decision-to-decision graph) [12]. Un nodo del ddgraph puede asociarse a una *decisión* (un punto en el programa en el cual

[1] Las componentes fuertemente conexas de un grafo $G = (N, A)$ se pueden encontrar en tiempo $O(|N| + |A|)$ [75].

el flujo de control diverge) o con una *empalme* (un punto en el programa en el cual el flujo de control converge). Un arco del ddgraph puede asociarse a un segmento del programa, o sea a una secuencia de instrucciones del programa no inturrumpida por decisiones ni empalmes. En algunos casos, se introduce un arco que no corresponde con un segmento del programa, si no que representa un posible flujo del flujo de control del programa (por ejemplo, un ELSE implícito, parte de una instrucción IF).

La siguiente es la definición formal de ddgraph.

Definición 2.1 *Ddgraph*

Un ddgraph es un digrafo $G = (N, A)$ *con dos arcos distinguidos* e_0 *y* e_k *(el único arco de entrada y el único arco de salida, respectivamente), tal que cualquier arco en* A *es alcanzado desde* e_0 *y alcanza* e_k*, y tal que para cada nodo* $n \in N$*, excepto para* $TAIL(e_0)$ *y* $HEAD(e_k)$*, se cumple que* $(indegree(n) + outdegree(n)) > 2$*, y* $indegree(TAIL(e_0)) = 0$ *y* $outdegree(TAIL(e_0)) = 1$*,* $indegree(HEAD(e_k)) = 1$ *y* $outdegree(HEAD(e_k)) = 0$.

En la Figura 2.1 presentamos el programa ejemplo SORT que usaremos a lo largo de la tesis, y el correspondiente ddgraph G_{SORT}. Los arcos distinguidos de G_{SORT} son e_1 (el arco de entrada) y e_8 (el arco de salida). La correspondencia entre las instrucciones en el programa (según la numeración en la Figura 2.1) y los arcos y nodos en el ddgraph se muestra en la Tabla 2.1[2].

Un *camino completo* (o *complete path*) en un ddgraph G es un camino desde el arco de entrada hasta el arco de salida de G. El camino característico (o *characteristic path*) p_c de G se define como la secuencia más larga de arcos (posiblemente no adyacentes), tales que para cualquier camino completo p en G, p_c es un pseudo-camino de p[3].

Un camino p en un ddgraph G de un programa σ se corresponde con un caso de test para σ. De hecho, p determina un predicado sobre el domino de inputs de σ. Cada nodo decisión en p podría agregar una nueva restricción a los inputs que siguen ese camino. Por ejemplo, consideremos el cammino $p = e_1, e_2, e_7, e_8$ en G_{SORT} de la Figura 2.1. En particular,

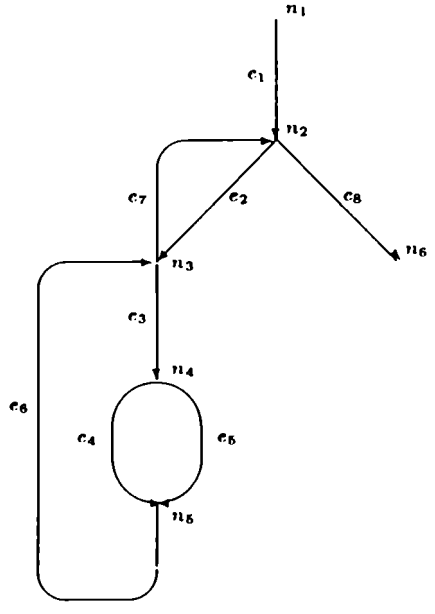
- el nodo n_2 = HEAD(e_1) corresponde a la línea 10 del programa SORT, es decir, a la instrucción **while** (*sortupto* < *n*);
- el nodo n_3 = HEAD(e_2) corresponde a la línea 13 del programa SORT, es decir, a la instrucción **while** (*index* <= *n*).

[2] Los arcos e_5 y e_3 no se corresponden con ninguna instrucción del programa.

[3] p_c puede obtenerse fácilmente usando el árbol de dominadores de G [55] (var Chapter 4, Definición 4.3), ya que coincide con el camino en el árbol que va desde la raíz, que representa al arco de entrada, hasta la hoja que representa el arco de salida.

```
void sort(a, n) /* SORT Program */
1    int a[];
2    int n;
3  {
4      int sortupto;
5      int maxpos;
6      int mymax;
7      int index;

8      sortupto = 1;
9      maxpos = 1;
10     while (sortupto < n) {
11         mymax = a[sortupto];
12         index = sortupto + 1;
13         while (index <= n) {
14             if (a[index] > mymax) {
15                 mymax = a[index];
16                 maxpos = index;
17             }
18             index = index + 1;
19         }
20         index = a[sortupto];
21         a[sortupto] = mymax;
22         a[maxpos] = index;
23         sortupto = sortupto + 1;
24     }
25 }
```



Figuras 2.1: Program SORT and ddgraph G_{SORT}

Lineas de Código	1-9	10	11-12	13	14	15-17	18-19	20-24	25
Componente del Ddgraph	e_1	n_2	e_2	n_3	n_4	e_4	e_6	e_7	e_8

Tablas 2.1: Correspondencia entre instrucciones de SORT y arcos y nodos en el ddgraph G_{SORT}

Procedure REDUCE ($G = (N, A)$:digraph): ddgraph;
begin
A*:= A; N*:= N;
while $\exists$ e_i, $e_j \in$ A* **such that**
$e_i \neq e_j$, TAIL(e_j)=HEAD(e_i), indegree(HEAD(e_i))=1, outdegree(TAIL(e_j))=1
do begin
N*:= (N*-{HEAD(e_i)});
A*:= (A-{e_i,e_j})$\cup${e_{i-j}}, where TAIL(e_{i-j})=TAIL(e_i) y HEAD(e_{i-j})=HEAD(e_j)
end
return(G^*)
end procedure.

Figuras 2.2: Procedure REDUCE

Por lo tanto, p se corresponde con el siguiente predicado:
$(1 < n)$ y (not$(2 <= n)$) y (not$(2 < n)$)[4].

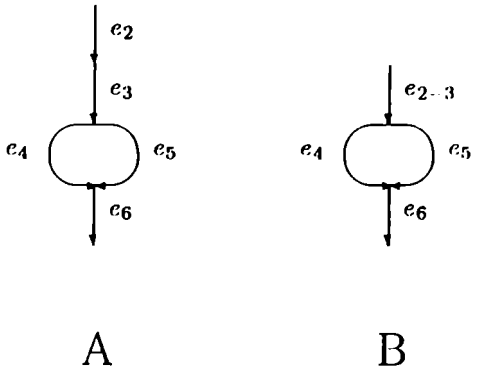
Sub-ddgraphs

Ahora introducimos el concepto de *sub-ddgraph* para un ddgraph G y dos arcos en G dados. Notamos al sub-ddgraph de G desde e_a hasta e_b como sub-ddgraph(G,e_a,e_b). Intuitivamente, un nodo (o arco) en G está en sub-ddgraph(G,e_a,e_b), si existe un camino desde $HEAD(e_a)$ hasta $TAIL(e_b)$ en G que incluye a ese nodo (o arco), pero no incluye a e_a ni a e_b. Un sub-ddgraph es un ddgraph.

Los sub-ddgraph de un ddgraph G pueden derivarse visitando G. Cuando esto sucede, puede pasar que la eliminación de algunos arcos deje un *procedure node*, es decir, un nodo con un único arco que entra en él e_{in} y un único arco que sale de él e_{out}. Pero esto no puede pasar en un ddgraph. Por lo tanto, es necesario eliminarlos, uniendo el arco e_{in} con el arco e_{out}. Para esto, introducimos el procedimiento REDUCE de la Figura 2.2, que transforma a un digrafo G en un ddgraph G^*, en tiempo $O(|A|)$.

Introducimos ahora la definición formal de sub-ddgraph.

[4]Notamos que, a partir del análisis del predicado asociado, podemos concluir que este camino no es factible.



Figuras 2.3: Sub-ddgraph(G_{SORT}, e_2, e_6)

Definición 2.2 *Sub-ddgraph*

Sea $G = (N, A)$ un ddgraph con arco de entrada e_0 y arco de salida e_k. Sean e_a y e_b dos arcos en A tales que e_a alcanza a e_b. El sub-ddgraph de G desde e_a hasta e_b, escrito sub-ddgraph(G,e_a,e_b), es el ddgraph obtenido después de reducir vía el procedimiento REDUCE (en Figura 2.2) al digrafo $G' = (N', A')$, donde

- *e_a y e_b no están en A';*
- *e'_0 y e'_k son el arco de entrada y de salida, respectivamente, en A', con $HEAD(e'_0) = HEAD(e_a)$ y $TAIL(e'_k) = TAIL(e_b)$ en N' y con $HEAD(e'_k)$ y $TAIL(e'_0)$ dos nodos nuevos y distintos en N' (no en N);*
- *si $e \in A$ y existe un camino p desde $HEAD(e_a)$ hasta $TAIL(e_b)$ en G, que no contiene a e_a ni a e_b, tal que e está en p, entonces $e \in A'$ y TAIL(e) y HEAD(e) están en N'.*

La Figura 2.3 muestra un ejemplo de cómo construir el sub-ddgraph(G_{SORT}, e_2, e_6) del ddgraph G_{SORT}. Primero se construye un digrafo (ver Figura 2.3A). Se identifa el procedure node $n_3 = HEAD(e_2) = TAIL(e_3)$ en este digrafo, que se elimina reduciendo los arcos e_2 y e_3 a un único arco e_{2-3}, y se obtiene el sub-ddgraph(G_{ex}, e_2, e_6) de G_{SORT} (ver Figura 2.3B).

El procedimiento SUB-DDGRAPH que, dado un ddgraph $G = (N, A)$ y dos arcos e_a y $e_b \in A$, devuelve G'=SUB-DDGRAPH(G,e_a,e_b) fue presentado en [12] (ver Figura 2.4). El algoritmo SUB-DDGRAPH toma tiempo $O(|A|)$.

La siguiente notación es usada en el algoritmo de la Figura 2.4:

- los conjuntos de arcos A' y A'' y los conjuntos de nodos N' y N'' son conjuntos auxiliares;
- una flag *label* se asocia a cada nodo y se inicializa en *false* antes de entrar a cada ciclo *repeat*. Tan pronto como el nodo n es alcanzado y es includido en el conjunto de nodos N' en el primer *repeat* o en el conjunto de nodos N'' en el segundo *repeat*, *label*(n) se modifica a *true*;
- una flag *used* se asocia a cada arco y se inicializa en *false* antes de entrar a cada ciclo *repeat*. Tan pronto como el arco es includido en el conjunto de arcos A' en el primer *repeat* o en el conjunto de arcos A'' en el segundo *repeat*, *used* (e) se modifica a *true*;
- se usa un conjunto auxiliar de nodos Q, que contiene los nodos que han sido alcanzados pero aún no han sido procesados.

2.2 Def-Use Ddgraphs

El análisis del flujo de datos considera las posibles interacciones entre las definiciones de las variables y los usos de las variables en un programa. Para analizar estas interacciones, los programas se representan como flowgraphs etiquetados. En este trabajo usamos un tipo de ddgraphs etiquetados, llamados *def-use ddgraphs* [57]. Describimos aquí los control flowgraph etiquetados y definimos los def-use ddgraphs.

La ocurrencia de una variable en un programa puede estar asociada a los siguientes eventos [9]:

Definition Una instrucción que almacena un valor en un lugar de memoria de una variable crea una definición (de esa variable).

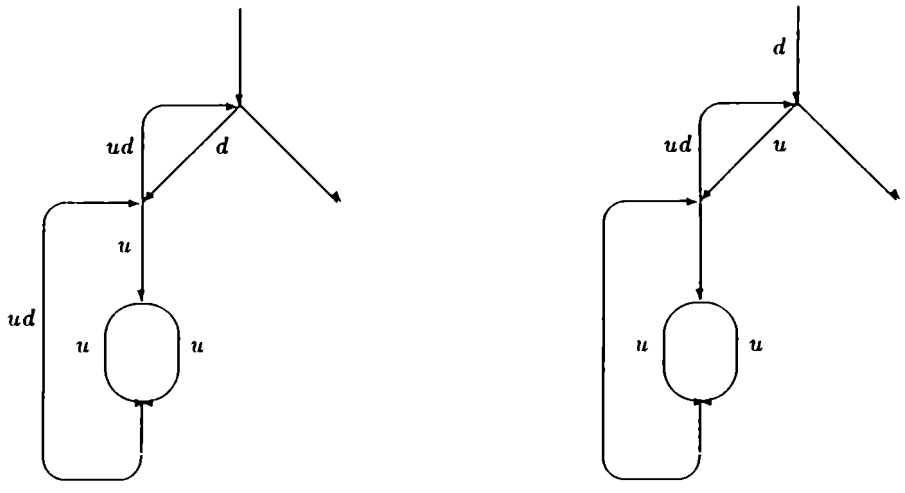
Use Una instrucción que usa un valor de un lugar de memoria de una variable es un uso de la definición que está activa en ese momento de la variable. En particular, cuando una variable aparece en el lado derecho de una asignación, se llama *computational use*; cuando la variable aparece en el predicado de una instrucción de transferencia de control, se llama *predicate use*.

Killing Una instrucción mata a la definición que está activa en ese momento de una variable cuando el valor asociado queda unbound.

Consideramos definiciones y usos *globales*. Un computational use de una variable X en un arco e es un *uso global* si ninguna definición de X precede al uso de X en e, y X fue definida

```
Procedure SUB-DDGRAPH (G = (N, A): ddgraph; c_a, c_b: arcs): ddgraph
begin
    N':=∅; A':=∅;
    for each n∈N do label(n):=false;
    for each e∈A do used(e):=false;
    label(H(e_a)):=true; Q:={H(e_a)}; used(e_a)=true;
    repeat
        select y remove n∈Q;
        N':=N'∪ {n};
        for each e∈A such that not(used(e)) and T(e)=n do
            begin
            used(e):=true;
            A':=A'∪ {e};
            if label(H(e))=false then
                begin
                label(H(e)):=true;
                Q:=Q∪ {H(e)};
                end
            end
    until Q=∅;
    N'':=∅;    A'':=∅
    for each n∈N' do label(n):=false;
    for each e∈A' do used(e):=false;
    label(T(e_b)):=true; Q:={T(e_b)}; used(e_b)=true;
    repeat
        select y remove n∈Q;
        A'':=A''∪ {n};
        for each e∈E' such that not(used(e)) and H(e)=n do
            begin
            used(e):=true;
            A'':=A''∪ {e};
            if label(T(e))=false then
                begin
                label(T(e)):=true;
                Q:=Q∪ {T(e)};
                end
            end
    until Q=∅;
    new(v_1); new(v_2);
    N'':=N''∪{v_1, v_2};
    e'_0:=(v_1, H(e_a)); e'_k:=(T(e_b), v_2);
    A'':=A''∪{e'_0, e'_k};
    return(REDUCE(G''=(N'', A'')));
end procedure.
```

Figuras 2.4: Procedure SUB-DDGRAPH



Figuras 2.5: Def-use ddgraphs para el programa SORT y las variables *index* y *a*, respectivamente

en algún otro arco. Los predicate uses (de variables ya definidas) son siempre globales. Una definición de una variable X en un arco e es una *definición global* si es la última definición de X en e y existe al menos un camino def-clear con respecto a X desde e hasta algún uso global de X (ver más adelante).

Dado un ddgraph correspondiente a un programa, y una variable en el programa, anotamos los arcos del ddgraph con los símbolos u, d, k, que representan el uso, definición y muerte de la variable de interés, en el segmento del programa representado por tales arcos. El ddgraph etiquetado se llama *def-use ddgraph* para la variable considerada. Formalmente, un *def-use ddgraph* para un programa dado y una variable dada es el ddgraph del programa, en el cual cada arco ha sido etiquetado con una secuencia (que puede ser vacía) de símbolos, que denotan la secuencia de operaciones de los datos en ese arco, con respecto a la variable de interés. En particular, para un predicate use (que en nuestro modelo se asocia a un nodo de decisión), anotamos con el símbolo u a cada arco que sale del nodo al cual está asociado el predicado.

En la Figura 2.5 presentamos los def-use ddgraphs correspondientes al programa SORT y la variable *index*, y al programa SORT y la variable *a*, respectivamente.

En adelante, asumimos que G es un def-use ddgraph para la variable X en el programa σ.

Un *camino def-clear* o *def-clear path* (o definition clear path) con respecto a una variable X es un camino $p = e, e_1, e_2, ..., e_q, e'$ en G, con $q \geq 0$, y tal que X puede estar definida en e, y no está redefinida ni matada en ningún arco entre $e_1, e_2, ..., e_q$. Por ejemplo, $p_1 = e_2, e_3, e_5, e_6$ es un def-clear path con respecto a *index* en G_{SORT} (porque *index* está definida en e_2 y no está redefinida ni matada en e_3 ni e_5). De la misma manera, $p_2 = e_2, n_2$ es un def-clear path con respecto a *index* en G_{SORT}. Por otro lado, $p_3 = e_2, e_3, e_5, e_6, e_7$ no es un def-clear path con respecto a *index* en G_{SORT}.

Un concepto importante en data flow testing es el de asociación definición-uso o *dua*.

Definición 2.3 *Dua*

Sean d y u dos arcos en G. Decimos que $T=[d, u, X]$ es una dua si la variable X tiene una definición global en d, un uso global en u, y existe un def-clear path con respecto a X desde d hasta u.

Por ejemplo, $T_1=[e_2, e_7, index]$ y $T_2=[e_6, e_4, index]$ son duas en G_{SORT}. Por otro lado, $T_3=[e_7, e_4, index]$ no es una dua.

Como veremos más adelante, duas y *clases de duas* constituyen los elementos básicos a cubrir de acuerdo a los varios criterios de data flow testing. El conjunto de todas las duas en un ddgraph G se nota como $D(G)$. Además, distinguiremos entre el conjunto $D_c(G)$ de c-duas, donde una dua $T=[d, u, X]$ está en $D_c(G)$ si u es un c-use, y el conjunto $D_p(G)$ de p-duas, donde una dua $T=[d, u, X]$ está en $D_p(G)$ si u es un p-use. Claramente, $D(G) = D_c(G) \cup D_p(G)$.

Seguiendo, listamos el conjunto de duas para SORT, $D(G_{SORT})$:

- $[e_1, e_8, n]$, $[e_1, e_2, n]$, $[e_1, e_7, n]$, $[e_1, e_3, n]$,
- $[e_1, e_2, a]$, $[e_1, e_7, a]$, $[e_7, e_2, a]$, $[e_7, e_7, a]$, $[e_7, e_4, a]$, $[e_7, e_5, a]$, $[e_1, e_4, a]$, $[e_1, e_5, a]$,
- $[e_1, e_2, sortupto]$, $[e_1, e_7, sortupto]$, $[e_7, e_8, sortupto]$, $[e_1, e_8, sortupto]$, $[e_7, e_2, sortupto]$, $[e_7, e_7, sortupto]$,
- $[e_1, e_7, maxpos]$, $[e_4, e_7, maxpos]$,
- $[e_2, e_7, mymax]$, $[e_2, e_4, mymax]$, $[e_4, e_7, mymax]$, $[e_4, e_4, mymax]$, $[e_2, e_5, mymax]$, $[e_4, e_5, mymax]$,
- $[e_2, e_3, index]$, $[e_6, e_7, index]$, $[e_2, e_7, index]$, $[e_2, e_4, index]$, $[e_6, e_4, index]$, $[e_6, e_3, index]$, $[e_2, e_5, index]$, $[e_6, e_5, index]$, $[e_2, e_6, index]$, $[e_6, e_6, index]$.

Otro concepto importante en este trabajo es el de *agrupar duas en clases*. Dos duas están en una misma clase si la misma variable está definida en el mismo arco del ddgraph en ambas duas. Entonces, agrupamos en una clase S_d^X todas las duas en $G=(N, A)$ tales que la variable X está definida en el arco d, i.e.:

$S_d^X = \{T \in D(G)\colon \exists u \in A,\ T = [d, u, X]\}$.

En lo que sigue, notaremos al conjunto de todas las clases de duas de un ddgraph G como $C(G)$[5]. Las duas en una misma clase se llaman *akin duas con respecto a* d *y* X.

Por ejemplo, las clases de duas para SORT y las variables a y $index$ son:

$S_{e_1}^a = \{\ [e_1, e_2, a],\ [e_1, e_7, a],\ [e_1, e_4, a],\ [e_1, e_5, a]\}$;

$S_{e_7}^a = \{\ [e_7, e_2, a],\ [e_7, e_7, a],\ [e_7, e_4, a],\ [e_7, e_5, a]\}$;

$S_{e_2}^{index} = \{\ [e_2, e_3, index],\ [e_2, e_7, index],\ [e_2, e_4, index],\ [e_2, e_6, index],\ [e_2, e_5, index]\}$;

$S_{e_6}^{index} = \{\ [e_6, e_7, index],\ [e_6, e_4, index],\ [e_6, e_3, index],\ [e_6, e_5, index],\ [e_6, e_6, index]\}$.

[5] Un conjunto vacío S_d^X no es una clase, i.e., si $\{T \in D(G)\colon \exists u \in A,\ T = [d, u, X]\} = \emptyset$, entonces S_d^X no está en $C(G)$.

Capítulo 3

Una Familia de Estrategias de Testing Estructural

Para ciertos programas (aquéllos que contienen al menos un loop), existen infinitos caminos desde la entrada hasta la salida del mismo. Existen varios criterios o estrategias que pueden usarse para seleccionar un subconjunto adecuado y finito del conjunto potencialmente infinito de inputs con el cual realizar el testing. Un criterio de test define una colección de requisitos que deben cumplirse cuando se testea el programa. En particular, los criterios de cubrimiento estructural requieren que un conjunto de *entidades* del flowgraph de un programa sean cubiertos cuando se ejecutan los tests. Dependiendo del criterio seleccionado, estas entidades pueden derivarse del flujo de control del programa o del flujo de datos del programa.

Es este capítulo redefinimos una familia muy popular de criterios de cubrimiento del flujo de control y del flujo de datos [9, 73, 31]. Para esto, identificamos, para cada criterio, el conjunto de entidades que deben cubrirse para satisfacerlo. Además, establecemos en forma precisa la noción de cubrimiento asociada a cada entidad. De hecho, "cubrir una entidad" tiene distintos significados según el tipo de entidad considerada: en la Section 3.2 definimos este significado en forma explícita.

Los resultados presentados en este capítulo fueron publicados en [56].

3.1 Criterios Estructurales

En lo que sigue, sea $G = (N, A)$ un ddgraph, y $\wp$ el conjunto de todos los caminos completos en G. Para cada criterio de test c, notaremos como $E_c(G)$ al correspondiente conjunto de entidades.

Criterios basados en el Flujo de Control

Criterios basados en el flujo de control es una familia de estrategias estructurales de testing, basadas en seleccionar un conjunto de caminos de test a través del programa, de manera de cubrir ciertas entidades del flujo de control. Son las más viejas de las técnicas estructurales, y las primeras en haber sido estudiadas.

All-Paths El criterio All-Paths requiere que todos los posibles caminos del flujo de control del programa sean ejecutados. Es la más fuerte entre las técnicas del flujo de control, y generalmente imposible de satisfacer ya que los programas con loops tienen un número infinito de caminos que deberían cubrirse.

En un ddgraph G, las entidades que deben cubrirse para satisfacer el criterio All-Paths son todos los caminos completos en G, i.e.,

$E_{All-Paths}(G) = \wp$.

All-k-Paths Este criterio (también llamado "structured testing" [64]) es una restricción del anterior, que limita las iteraciones de los loops en un camino. Los caminos seleccionados son aquellos que no iteran un loop más de k veces, para un entero k dado.

En un ddgraph G, las entidades son:

$E_{All-k-Paths}(G) = \{p \in \wp$: p es un camino completo en G y ningún loop en p se itera más de k veces$\}$.

All-Branches Las estrategia All-Branches requiere que cada alternativa branch en el programa (i.e., cada posible divergencia del flujo de control) sea ejecutada al menos una vez bajo algún test.

En un ddgraph G, la divergencia del flujo de control se modela como varios arcos saliendo de un nodo (decisión). El criterio All-Branches se satisface entonces si cada arco en G está cubierto, es decir:

$E_{All-Branches}(G) = A$.

All-Statements El criterio All-Statements es la forma más simple de criterio basado en el flujo de control. Requiere que todas las instrucciones del programa sean ejecutadas al menos una vez bajo algún test.

En el modelo ddgraph, las entidades son los arcos del ddgraph que se corresponden con instrucciones del programa (por ejemplo, un arco que representa la parte ELSE implícita de una instrucción IF-THEN no estaría incluido):

$E_{All-Statements}(G) = \{e \in A$: e está asociado con al menos una instrucción del programa$\}$.

Criterios basados en el Flujo de Datos

A pesar de que gran parte de la detección de errores del flujo de datos puede hacerse mediante análisis estático, muchos errores sólo pueden detectarse cuando se ejecuta el programa. Existe una familia de estrategias de testing del flujo de datos [53, 65, 73, 21, 80] que se basa en la selección de caminos del flujo de control del programa con el objetivo de explorar las secuencias de eventos relaccionadas con el estado de los datos. Estas estrategias seleccionan segmentos de caminos que satisfacen requisitos del flujo de datos para todas las variables del programa.

Siguiendo, presentamos varias estrategias de testing basadas en el flujo de datos.

All-du-Paths Este criterio requiere testear todos los caminos simples def-clear desde cada definición de cada variable en el programa hasta cada posible uso de esa definición. Una cota superior del número de caminos necesarios para satisfacer esta estrategia es 2^d, donde d es el número de decisiones en el programa [79]. Sin embargo, hay evidencia empírica de que muchos errores pueden encontrarse testeando el programa con no demasiados casos [16, 79].

En nuestra terminología, el criterio All-du-Paths se satisface si para cada dua $T = [d, u, X]$, todos los caminos simples y def-clear con respecto a X desde d hasta u están cubiertos. Por lo tanto, las entidades a ser cubiertas son estos caminos.

$E_{All-du-Paths}(G) = \{p \in p$: $\exists T = [d, u, X] \in D(G)$ tal que p es un camino simple y def-clear con respecto a X desde d hasta $u\}$.

All-Uses Este criterio requiere que se cubra al menos un camino def-clear desde cada definición de cada variable en el programa hasta cada posible uso de esta definición. Esta estrategia es probablemente la que tiene mejor relación costo/beneficio [9]. Se ha demostrado que descubre muchos errorres [29], mientras que se necesitan relativamente pocos casos para satisfacerla [79]. En el modelo ddgraph, las entidades son las duas.

$E_{All\text{-}Uses}(G) = D(G)$.

All-p-Uses Este criterio requiere que se cubra al menos un camino def-clear desde cada definición de cada variable en el programa hasta cada posible p-uso de esta definición. Es decir, se deben ejecutar todas las asociaciones def-p-use.

En un ddgraph G, las entidades son las p-duas, i.e., una dua $T=[d, u, X]$ es una entidad si el uso u es un p-use.

$E_{All\text{-}p\text{-}Uses}(G) = D_p(G)$.

De la misma manera, definimos el criterio *All-c-Uses* e identificamos el conjunto de entidades asociadas, es decir, $E_{All\text{-}c\text{-}Uses}(G) = D_c(G)$.

All-p-Uses/Some-c-Uses Este criterio asume que si se ha cubierto una p-dua, entonces no hay necesidad de seleccionar un c-use de la definición de la p-dua. Es decir, se debe incluir al menos un camino def-clear desde cada definición de cada variable hasta cada p-uso; si existen definiciones de las variables que no han sido cubiertas por los casos así seleccionados, entonces se deben agregar los casos necesarios para cubrirlas (mediante un c-uso).

En un ddgraph G, las entidades son cada asociación def-p-use y al menos una asociación def-c-use para cada definción no cubierta por los p-usos.

$E_{All\text{-}p\text{-}Uses/Some\text{-}c\text{-}Uses}(G) = \{\{T\}\colon T \in D_p(G)\} \cup \{S_d^X \in C(G)\colon \forall\, T = [d', u', X'] \in D_p(G),\ d \neq d' \text{ o } X \neq X'\}$.

De la misma manera, definimos el criterio *All-c-Uses/Some-p-Uses* e identificamos el conjunto de entidades asociadas, es decir, $E_{All\text{-}c\text{-}Uses/Some\text{-}p\text{-}Uses}(G) = \{\{T\}\colon T \in D_c(G)\}$ $\cup\ \{S_d^x \in C(G)\colon \forall\, T = [d', u', X'] \in D_c(G),\ d \neq d' \text{ o } X \neq X'\}$.

All-Defs Este criterio requiere que se cubra cada definición de cada variable con al menos un uso (computacional o de predicado) de esa variable. El criterio All-Defs se satisface si para cada conjunto S_d^X en G, se cubren al menos una de las duas en él.

$E_{All\text{-}Defs}(G) = \{S_d^X\colon \exists T = [d', u', X'] \in D(G)$ ta que $d = d'$ y $X = X'\}$.

La Tabla 3.1 resume los conjuntos de entidades para cada criterio de cubrimiento analizado. Presentamos ahora algunos ejemplos.

- El conjunto de entidades para el ddgraph $G_{SORT} = (N_{SORT}, A_{SORT})$ y el criterio All-Branches es

  $E_{All\text{-}Branches}(G_{SORT}) = A_{SORT} = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8\}$;

Coverage Criterion c	Set $E_c(G)$ of entities for ddgraph G and coverage criterion c
All-Paths	$\wp$
All-k-Paths	$\{p \in \wp$: no loop in p is iterated more than k times$\}$
All-Branches	Λ
All-Statements	$\{e \in \Lambda$: e is associated with at least one instruction$\}$
All-du-Paths	$\{p \in \wp$: $\exists T' = [d, u, X] \in D(G)$ such that p contains a simple, def-clear path wrt X from d to $u\}$
All-Uses	$D(G)$
All-p-Uses	$D_p(G)$
All-c-Uses	$D_c(G)$
All-p-Uses/Some-c-Uses	$\{\{T\}$: $T \in D_p(G)\} \cup \{S_d^x \in C(G)$: $\forall T' = [d', u', X'] \in D_p(G) \Rightarrow d \neq d'$ or $X \neq X'$ $\}$
All-c-Uses/Some-p-Uses	$\{\{T\}$: $T \in D_c(G)\} \cup \{S_d^x \in C(G)$: $\forall T' = [d', u', X'] \in D_c(G) \Rightarrow d \neq d'$ or $X \neq X'$ $\}$
All-Defs	$\{S_d^x$: $\exists T = [d', u', X'] \in D(G)$ such that $d = d'$ and $X = X'\}$

Tablas 3.1: Entidades y Cubrimiento para los Criterios de Cubrimiento Estructural

- el conjunto de entidades para el ddgraph G_{SORT} y el criterio All-Statements es

  $E_{All-Statements}(G_{SORT}) = \{e_1, e_2, e_4, e_6, e_7, e_8\}$;

- el conjunto de entidades para el ddgraph G_{SORT} y el criterio All-Uses es

  $E_{All-Uses}(G_{SORT}) = D(G_{SORT})$ (ver el Capítulo 2 donde se presenta una lista de las duas en este conjunto);

- el conjunto de entidades para el ddgraph G_{SORT} y el criterio All-Defs es

  $E_{All-Defs}(G_{SORT}) = \{ S_{e_1}^n, S_{e_1}^a, S_{e_7}^a, S_{e_1}^{sortupto}, S_{e_7}^{sortupto}, S_{e_1}^{maxpos}, S_{e_4}^{maxpos}, S_{e_2}^{mymax}, S_{e_4}^{mymax}, S_{e_2}^{index}, S_{e_6}^{index}\}$.

3.2 ¿Cuál es el Significado de "Cubrimiento"?

Hasta ahora, hemos definido las entidades para un programa, a partir del flowgraph del programa y para distintos criterios de cubrimiento. Un conjunto de caminos en el flowgraph satisface un criterio si *cubre* al conjunto de entidades asociado a ese criterio. Las entidades asociadas a los crierios pueden ser de distinto tipo. Y el término "cubrimiento" tiene distintos significados, de acuerdo al tipo de entidad considerada.

En general, para los criterios en la familia considerada, hay cuatro tipos de entidades:

- arco
- dua
- clase de duas
- camino

En particular, los arcos son las entidades para los criterios All-Branches y All-Statements; las duas son las entidades para los criterios All-Uses, All-p-Uses, y All-c-Uses; las clases de duas son las entidades para los criterios All-p-Uses/Some-c-Uses, All-Defs, y All-c-Uses/Some-p-Uses; y los caminos son las entidades para los criterios All-Paths, All-k-Paths y All-du-Paths.

Definimos ahora el significado de cubrimiento para los cuatro tipos de entidades consideradas.

Definición 3.1 *Cubrimiento*
Un camino completo p cubre

- *un* arco *si p contiene a ese arco;*
- *una* dua $T = [d, u, X]$ *si p tiene un sub-camino def-clear con respecto a X desde d hasta u;*
- *una* clase de duas S *si* $\exists T \in S$ *tal que p cubre T;*
- *un* camino p' *si p' es un sub-camino de p.*

Un conjunto de caminos completos $\wp$ cubre un arco (o una dua, o una clase de duas, o un camino) si alguno de los caminos en $\wp$ lo hace.

Presentamos algunos ejemplos. Consideramos el camino $p = e_1, e_2, e_7, e_8$, en el ddgraph G_{SORT}. Por la definición 3.1,

- p cubre al arco e_1, pero no cubre al arco e_3;
- p cubre la dua $[e_2, e_7, index]$ y la dua $[e_1, e_2, a]$, pero no cubre la dua $[e_2, e_3, index]$ ni la dua $[e_7, e_2, a]$;
- p cubre la clase de duas $S^{index}_{e_2}$, pero no cubre la clase de duas $S^{index}_{e_6}$ ni la clase $S^{a}_{e_7}$;
- p cubre al camino e_1, e_2, pero no cubre al camino e_7, e_2.

Capítulo 4

Spanning Sets, Subsumption y Unconstrainedness

Tres Conceptos Nuevos para el Análisis de un Programa

Una estrategia de test estructural selecciona un subconjunto de caminos que deben ser ejecutados al hacer el testing. Como hemos dicho, esta selección está orientada a cubrir un conjunto particular de entidades de la estructura del programa, derivadas del flujo de control o del flujo de datos.

Los métodos existentes para seleccionar tests generan un dato para cubrir una entidad seleccionada en forma arbitraria, y considerada en forma aislada de las otras entidades [36]. Si el dato de test generado también ejecuta otras entidades, se considerarán cubiertas a posteriori. Pero no se realiza ningún esfuerzo para generar a priori datos de test que satisfagan varios requisitos simultaneamente.

Sin embargo, observamos que un caso de test cubre, en general, a más de una entidad. Es entonces posible, en general, identificar un subconjunto de entidades *mínimo* con la propiedad de que cualquier conjunto de tests que cubre este subconjunto cubre toda entidad en el programa. Llamamos a este subconjunto mínimo un "spanning set of entities".

Intuitivamente, existe un orden entre las entidades, de acuerdo a cuán facilmente pueden ser

cubiertas. Llamamos *subsumption* a este orden entre entidades. Las entidades maximales en este orden, es decir, aquellas cuyo cubrimiento no puede ser garantizado por ninguna otra entidad, se dicen *unconstrained*. Un spanning set of entities está compuesto por entidades unconstrained.

En este capítulo primero introducimos los spanning sets of entities. Después, presentamos un método para identificar un spanning set of entities para las estrategias estructurales presentadas en el último capítulo, es decir, un método que permite identificar un conjunto de entidades mínimo que garantiza cubrimiento total. Este método está basado en la noción de subsumption.

Los resultados presentados aquí fueron publicados en [56].

4.1 Spanning Sets of Entities

Debería ser claro en este momento que un criterio de cubrimiento c determina para un ddgraph G dado, un conjunto de c-entidades $E_c(G)$ que deben ser cubiertas para satisfacer el criterio. Observamos ahora que es en general posible derivar un subconjunto de $E_c(G)$ con la propiedad de que un conjunto de caminos completos que cubre todas las entidades en él, cubrirá también a toda c-entidad en $E_c(G)$. Por ejemplo, para el ddgraph G_{SORT}, cualquien conjunto de caminos completos que cubre los arcos:

$\{e_3, e_4, e_5, e_6\} \subseteq E_{All-Branches}(G_{SORT})$

cubrirá cada arco en $E_{All-Branches}(G_{SORT})$. Esto sucede porque cualquier camino completo que cubre estos arcos, debe cubrir también a los arcos e_1, e_2, e_7 y e_8.

Para los propósitos del testing, estamos interesados en el (o los) subconjunto(s) más chico de c-entidades con esta propiedad. Estas son las c-entidades de $E_c(G)$ "más importantes", pero también aquellas más difíciles de ejecutar. Conociendo este subconjunto a priori, el esfuerzo del testing puede centrarse en cubrir las c-entidades de este subconjunto, y por lo tanto satisfacer el criterio c con menos esfuerzo.

Un subconjunto mínimo de $E_c(G)$ tal que un conjunto de caminos que lo cubre cubre toda entidad en $E_c(G)$ es llamado un *spanning set of entities*. El término spanning set of entities significa que "se expande por" todas las entidades para un ddgraph G dado y un criterio c dado.

Definición 4.1 *Spanning Set of Entities*

Sea G un ddgraph y c un criterio de cubrimiento de los presentados. Un subconjunto U de $E_c(G)$ es un spanning set of entities *para G y c si*

1. *cualquier conjunto de caminos p que cubre todas las entidades en U cubre también todas las entidades en $E_c(G)$;*

2. *para cualquier conjunto* $U' \subseteq E_c(G)$, *tal que cualquier conjunto de caminos que cubre todas las entidades en* U' *cubre también todas las entidades en* $E_c(G)$, *se tiene que* $|U| \leq |U'|$.

4.2 Cómo Encontrar un Spanning Set of Entities

En la última sección, introducimos el concepto de spanning set of entities. Ahora presentaremos un método general para encontrar un spanning set of entities, para un ddgraph G dado y un criterio de cubrimiento c, para cualquier criterio en la familia definida en el Capítulo 3.

Como ya hemos dicho, existe un orden entre las entidades, de acuerdo a cuán fácilmente pueden ser cubiertas. Llamamos a este orden entre entidades, *subsumption*. La idea intuitiva es que si una c-entidad E_1 *subsume* a otra c-entidad E_2, entonces nos podemos "olvidar" de E_2 siempre que "recordemos" a E_1.

Definición 4.2 *Subsumption*

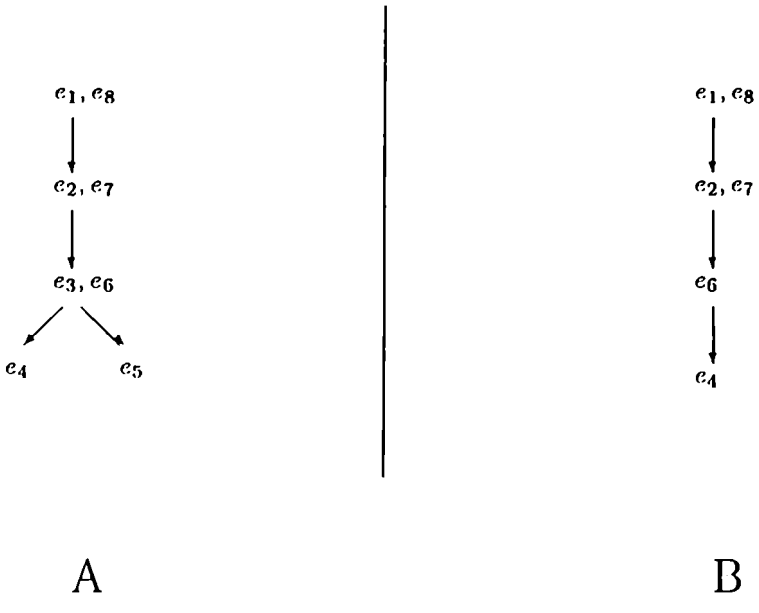
Sea G *un ddgraph,* c *un criterio de cubrimiento, y* E_1 *y* E_2 *dos entidades en* $E_c(G)$. *Entonces,* E_1 *subsume a* E_2 *si todo camino que cubre a* E_1 *cubre también a* E_2.

Estamos interesados en identificar a los objetos maximales en ese orden. Esas entidades se dicen "unconstrained", ya que el cubrimiento de las entidades unconstrained no está garantizado por el cubrimiento de ninguna otra entidad. Más precisamente, una entidad E se dice *unconstrained* si no existe ninguna otra entidad E' que subsuma a E sin que suceda que E subsume a E'.

Veremos ahora cómo usar esta relación para encontrar un spanning set of entities.

En general, una relación R sobre un conjunto S es un *preorden* si es reflexiva, i.e., (s, s) está en R para cada s en S, y transitiva, i.e., si (s_1, s_2) y (s_2, s_3) están en R entonces (s_1, s_3) también está en R. La relación de subsumption es un preorden. Entonces, dado un ddgraph G y un criterio de cubrimiento c, podemos construir un grafo dirigido que representa a la relación de subsumption entre las entidades. Los nodos son las entidades para G y c. Hay un arco desde en nodo E_2 hasta el nodo E_1 si E_1 subsume a E_2. Este digrafo se llama *c-subsumption digraph* de G, y se nota como $S_c(G)$.

Cualquier entidad en una componente fuertemente conexa M de $S_c(G)$ puede elegirse para representar a otra. Haremos referencia a la entidad elegida para representar a una componente M como la *representante* de M, y la notaremos $rep(M)$. Reduciendo las componentes fuertemente conexas de $S_c(G)$, se obtiene un DAG (directed acyclic digraph). Este grafo se llama *reduced c-subsumption digraph*, y se nota $R_c(G)$.



Figuras 4.1: Reduced Subsumption Digraph para G_{SORT} y los criterios All-Branches y All-Statements, respectivamente

Consideraremos las hojas del reduced c-subsumption digraph. Sea U un conjunto de representantes de estas hojas. Puede probarse fácilmente que U es un spanning set of (unconstrained) entities.

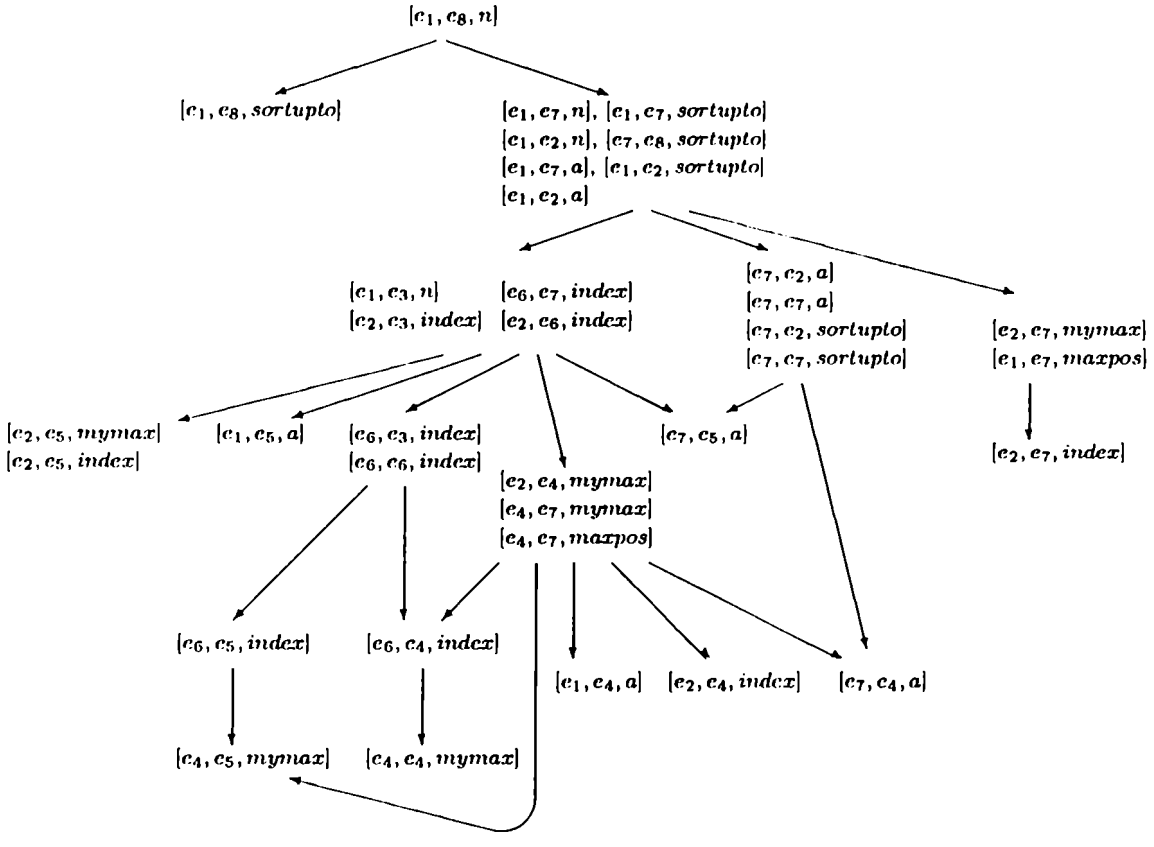
Teorema 4.1 *Sea G un ddgraph, c un criterio de cubrimiento, y $R_c(G)$ el reduced c-subsumption digraph de G. Sea U un conjunto de c-entities de $E_c(G)$ representando las hojas de $R_c(G)$, i.e., cada entidad en U representa una hoja en $R_c(G)$ y todas las hojas en $R_c(G)$ tienen un representante en U. Entonces U es un spanning set of entities para G y c.*

La demostración puede verse en el Apéndice A

Por ejemplo, presentamos $U_{All-Branches}(G_{SORT})$, el único spanning set of entities para el ddgraph $G_{SORT} = (N_{SORT}, A_{SORT})$ y el criterio All-Branches. Puede obtenerse seleccionando los representantes de las hojas del reduced $All-Branches$-subsumption digraph para G_{SORT} de la Figura 4.1 A,

$U_{All-Branches}(G_{SORT}) = \{e_4, e_5\}$.

El único spanning set of entities para el ddgraph G_{SORT} y el criterio All-Statements puede obtenerse seleccionando los representantes de las hojas del reduced $All-Statements$-subsumption digraph para G_{SORT} de la Figura 4.1 B,



Figuras 4.2: Reduced Subsumption Digraph para G_{SORT} y el criterio All-Uses

Procedure **FIND-A-SPANNING-SET-OF-ENTITIES** (G: ddgraph; $E_c(G)$: set of entities): set of entities;

1. **for each** $E_1, E_2 \in E_c(G)$, $E_1 \neq E_2$, **do** SUBSUMPTION(E_1, E_2, G);
2. construct $S_c(G) = (V_{S_c(G)}, E_{S_c(G)})$;
3. construct $R_c(G) = (V_{R_c(G)}, E_{R_c(G)})$;
4. U = {rep(M): M is a leaf of $R_c(G)$};
5. **return**(U).

Figuras 4.3: Procedure FIND-A-SPANNING-SET-OF-ENTITIES

$U_{All\text{-}Statements}(G_{SORT}) = \{e_4\}$.

Un spanning set of entities para G_{SORT} y el criterio All-Uses puede obtenerse seleccionando los representantes de las hojas del reduced $All-Uses$-subsumption digraph para G_{SORT} de la Figura 4.2. En este caso existen dos spanning sets of entities, ya que o bien $[e_2, e_5, index]$ o bien $[e_2, e_5, index]$ pueden elegirse como representantes de la hoja que las contiene:

$U_{All-Uses}(G_{SORT}) = \{ [e_1, e_8, sortupto], [e_2, e_5, index], [e_2, e_4, index], [e_2, e_7, index], [e_1, e_5, a], [e_7, e_5, a], [e_1, e_4, a], [e_7, e_4, a], [e_4, e_5, mymax], [e_4, e_4, mymax]\}$,

$U'_{All-Uses}(G_{SORT}) = \{ [e_1, e_8, sortupto], [e_2, e_5, mymax], [e_2, e_4, index], [e_2, e_7, index], [e_1, e_5, a], [e_7, e_5, a], [e_1, e_4, a], [e_7, e_4, a], [e_4, e_5, mymax], [e_4, e_4, mymax]\}$,

Notar que para simplificar las Figuras 4.1 A y B y la Figura 4.2, hemos dibujado sólo los arcos significativos.

Resumiendo, para encontrar un spanning set of entities U para un ddgraph G y un criterio de cubrimiento c, primero se obtiene el conjunto de entidades $E_c(G)$. Luego, se sigue el procedure FIND-A-SPANNING-SET-OF-ENTITIES($G, E_c(G)$), descripto en la Figura 4.3.

El análisis de la complejidad de este procedimiento se presenta en la Sección 4.4.

4.3 La Relación Subsumption

El primer paso del procedure FIND-A-SPANNING-SET-OF-ENTITIES presentado en la sección anterior (es decir, evaluar si una entidad E_1 *subsume* a una entidad E_2) depende de la noción de "cubrimiento" asociada al tipo particular de entidad considerada (ver Definición 3.1). Entonces

necesitamos una implementación distinta del procedimiento SUBSUMPTION(E_1,E_2,G), para cada tipo posible de entidad (es decir, arco, dua, clase de duas, y camino). Notamos que la implementación no depende del criterio, depende del tipo de entidad. En esta sección discutiremos cómo puede implementarse el procedimiento SUBSUMPTION para los cuatro tipos de entidades. Introducimos los siguientes procedimientos:

- SUBSUMPTION-BETWEEN-ARCS,
- SUBSUMPTION-BETWEEN-DUAS,
- SUBSUMPTION-BETWEEN-CLASSES-OF-DUAS, y
- SUBSUMPTION-BETWEEN-PATHS.

4.3.1 Relaciones Dominance, Post-Dominance y Alignment

Los conceptos que presentamos aquí serán usados en la implementación del procedimiento SUBSUMPTION.

Una relación muy conocida de la teoría de grafos es la de *dominance* [41]. Esta relación impone un orden parcial entre los nodos o arcos de un flowgraph[1]. Estamos interesados en usar esta relación sobre los arcos de un ddgraph.

Definición 4.3 Dominance en ddgraphs

Sea G un ddgraph, e_0 el arco de entrada de G. Un arco e' domina *al arco e si todo camino p desde e_0 hasta e contiene a e'.*

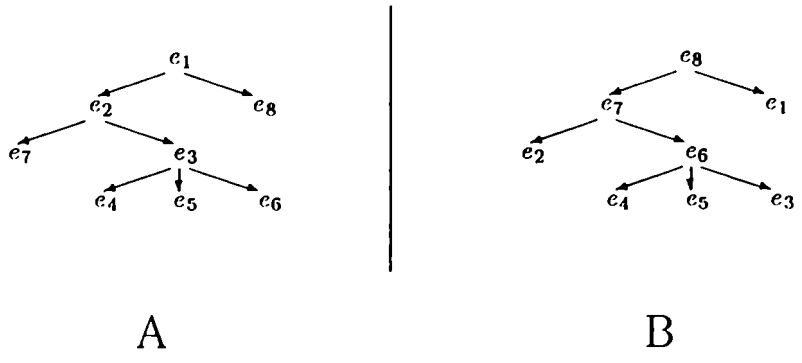
Existen muchos algoritmos para encontrar dominadores en un digrafo, por ejemplo [55].

El *dominador inmediato* e' de un arco e es un dominador de e tal que cualquier otro dominador de e también domina a e'.

Esta relación puede representarse como un árbol, cuyos nodos son los arcos del ddgraph G y cuya raíz es e_0. Se llama *dominator tree* y se nota $DT(G)$. Existe un arco (e, e') entre dos nodos e y e' en el dominator tree, si e es el dominador inmediato de e'. En la Figura 4.4 A, presentamos el dominator tree del ddgraph G_{SORT} (de la Figura 2.1).

El mejor algoritmo conocido para construir un dominator tree lleva tiempo $O(|A|\alpha(|A|,|N|))$ [55], donde A es el conjunto de arcos y N el conjunto de nodos de G, y α es la inversa de la función de Ackermann.

[1] Un orden parcial es una relación reflexiva, asimétrica y transitiva. Una relación R en un conjunto S es asimétrica si (s_1, s_2) está en R entonces (s_2, s_1) no está en R.



Figuras 4.4: $DT(G_{SORT})$ y $IT(G_{SORT})$

Ahora introducimos la simétrica de esta relación, llamada *post-dominance*. Esta relación aparece en la literatura con distintos nombres (por ejemplo, follow node [7], inverse-dominance [19], implication [12]).

Definición 4.4 Post-dominance en ddgraphs

Sea G un ddgraph, e_k el arco de salida de G. Un arco e' post-domina *al arco e si todo camino P desde e hasta e_k contiene a e'.*

La relación de post-dominance en un ddgraph G puede obtenerse a partir de la relación de dominance. La representación de la relación de post-dominance sobre los arcos de G como un arco con raíz se llama *post-dominator tree* (o *implication tree*) y se nota $IT(G)$. El post-dominator tree del ddgraph G_{SORT} se presenta en la Figura 4.4 B.

Ahora introducimos la relación de alignment entre tres arcos de un ddgraph.

Definición 4.5 Alignment

Decimos que los arcos a, b y c en un ddgraph G están aliñados *si todo camino desde a hasta c contiene b, y se nota $AL(a,b,c)$.*

El siguiente resultado muestra cómo implementar la relación alignment.

Proposición 4.1 *$AL(a,b,c)$ vale si y sólo si b pertenece al camino característico del sub-ddgraph(G,a,c).*

La demostración puede verse en el Apéndice A.

Entonces el orden de tiempo de saber si se satisface AL(a, b, c) es $O(|A|\alpha(|A|, |N|))$.

El siguiente resultado sale directamente de las Definiciones 4.3, 4.4 y 4.5. Muestra que las relaciones de dominance y post-dominance son instancias particulares de la de alignment.

Proposición 4.2 *Para dos arcos cualquiera a y b en un ddgraph G con arco de entrada e_0 y arco de salida e_k, se tiene que*

a domina *b si y solo si AL(e_0,a,b) , y*

b post-domina *a si y solo si AL(a,b,e_k).*

4.3.2 Implementación de SUBSUMPTION-BETWEEN-ARCS

Mostramos aquí dos procedimientos para chequear si un arco e subsume un arco e'.

Teorema 4.2 *Sean e y e' dos arcos en un ddgraph G con arco de entrada e_0 y arco de salida e_k. Entonces, e subsume e' si y solo si e' domina o post-domina e.*

La demostración puede verse en el Apéndice A.

Entonces, para implementar SUBSUMPTION-BETWEEN-ARCS(e,e',G) podemos chequear si e' domina o post-domina a e. Esto puede hacerse en tiempo $O(|A|\alpha(|A|, |N|))$ si $G = (N, A)$.

Otra manera de implementar SUBSUMPTION-BETWEEN-ARCS es chequeando si AL(e_0,e',e) o AL(e,e',e_k). Esto es gracias al siguiente teorema:

Teorema 4.3 *Sean e y e' dos arcos en el ddgraph G con arco de entrada e_0 y arco de salida e_k. Entonces, e subsume e' si y solo si AL(e_0,e',e) o AL(e,e',e_k).*

La demostración puede verse en el Apéndice A.

Por lo tanto, para implementar SUBSUMPTION-BETWEEN-ARCS(e,e',G) podemos chequear si AL(e_0,e',e) o AL(e,e',e_k), en tiempo $O(|A|\alpha(|A|, |N|))$ time if $G = (N, A)$, si A es el conjunto de arcos y N el conjunto de nodos de G, usando la Proposición 4.1.

ALL-PATHS(a, b: arcs or nodes; G = (N, A): ddgraph): digraph;
begin
A_p={$e \in A$: there exists a path in G from a to b containing e};
N_p={$n \in N$: there exists an arc $e \in A_p$, n=TAIL(e) or n=HEAD(e)};
return(N_p, A_p)
end

Figuras 4.5: Procedure ALL-PATHS

ALL-DEF-CLEAR-PATHS(n_1, n_2: nodes; X: variable; G = (N, A): digraph): digraph;
begin
A_1 = {$e \in A$: e contains no definition of X};
N_1 = {n: $\exists e \in A_1$, n=TAIL(e) or n=HEAD(e)};
return(ALL-PATHS($n_1, n_2, (N_1, A_1)$))
end

Figuras 4.6: Procedure ALL-DEF-CLEAR-PATHS

4.3.3 Implementación de SUBSUMPTION-BETWEEN-DUAS

Sea T_1=$[d_1, u_1, X_1]$ y T_2=$[d_2, u_2, X_2]$ dos duas en el ddgraph G. En esta sección presentamos un método para chequear si T_1 subsume a T_2.

Se usan las siguientes operaciones sobre caminos:

- $\langle\ e\ \rangle$ devuelve el camino formado por el arco e;
- $p_1 \cdot p_2$ devuelve un camino que consiste en el camino p_1 seguido por el camino p_2, si se cumple que la HEAD del último arco en p_1 es la TAIL del primer arco en p_2.

Sean a y b dos arcos o nodos en un digraph G; el procedimiento ALL-PATHS(a, b, G) de la Figura 4.5 construye el digrafo representando a todos los caminos desde a hasta b en G.

Sean n_1 y n_2 dos nodos, y X una variable en un ddgraph G; la siguiente proposición muestra que el procedimiento ALL-DEF-CLEAR-PATHS(n_1, n_2, X, G) de la Figura 4.6 construye el digrafo que representa a todos los caminos desde n_1 hasta n_2 en G que no contienen una definición de X.

```
COVERING-PATHS(G = (N, A): ddgraph; [d, u, X]: dua): digraph;
begin
    G1 = (N1, A1) = ALL-PATHS(ENTRY-ARC(G), TAIL(d), G);
    G2 = (N2, A2) = ALL-DEF-CLEAR-PATHS(HEAD(d), TAIL(u), X, G);
    G3 = (N3, A3) = ALL-PATHS(HEAD(u), EXIT-ARC(G), G)
    N' = N1 ∪ N2 ∪ N3;
    A' = A1 ∪ A2 ∪ A3 ∪ {d, u};
    return(N', A')
end
```

Figuras 4.7: Procedure COVERING-PATHS

Proposición 4.3 *Sea $G=(N, A)$ un digrafo. Sean $n_1, n_2 \in N$, y X una variable en G.*

Entonces, $\{p$: p es un camino desde n_1 hasta n_2 en $ALL\text{-}DEF\text{-}CLEAR\text{-}PATHS(n_1, n_2, X, G)\}$ = $\{p$: p es un camino desde n_1 hasta n_2 en G que no contiene una definición de $X\}$.

La demostración puede verse en el Apéndice A.

Dado un ddgraph G, con arco de entrada e_0 y arco de salida e_k, y una dua T, usamos los procedimientos ALL-PATHS y ALL-DEF-CLEAR-PATHS para derivar COVERING-PATHS(G, T), que es el digrafo que representa todos los caminos completos en G que cubren a T. COVERING-PATHS se construye usando el procedimiento de la Figura 4.7.

En la construcción de COVERING-PATHS suponemos que ningún arco pertenece a más de uno de los siguientes conjuntos: A_1, A_2, A_3 y $\{d, u\}$. De otra manera, podemos asumir que usamos una función que renombra los arcos. Por ejemplo, podemos renombrar a cada arco e en el conjunto A_i como un nuevo arco e_i, i.e., $rename(e, A_i) = e_i$. Supongamos ahora que consideramos un camino en el digrafo COVERING-PATHS. Para obtener un camino en el grafo original, consideramos la inversa de la función de renombre, $rename^{-1}$. Usando esta función, dado un arco e_i en COVERING-PATHS obtenemos el arco e en el ddgraph original.

La siguiente proposición establece que todo camino en COVERING-PATHS(G, T) es un camino en G_1, seguido del arco d, luego de un camino en G_2, luego del arco u, y luego de un camino en G_3. Inversamente, cada camino compuesto por tres caminos "completos" p_1, p_2, p_3 en G_1, G_2 y G_3, respectivamente, se asocia con un camino en COVERING-PATHS(G, T).

Proposición 4.4 *Sea $G=(N, A)$ un ddgraph, e_0 el arco de entrada de G, e_k el arco de salida de G. Sea $T=/d, u, X/$ una dua en G. Sean G_1, G_2 y G_3 los digrafos definidos en la Figura 4.7.*

Entonces, $\{p$: p es un camino desde e_0 hasta e_k en COVERING-PATHS(G, T)$\}$ = $\{p$: $\exists p_1, p_2, p_3$, tal que $p = p_1 + \langle\ d\ \rangle + p_2 + \langle\ u\ \rangle + p_3$; p_1 es un camino en G_1 desde e_0 hasta TAIL(d); p_2 es un camino en G_2 desde HEAD(d) hasta TAIL(u); y p_3 es un camino en G_3 desde HEAD(u) hasta $e_k\}$.

La demostración puede verse en el Apéndice A.

El siguiente corolario dice que todo camino completo en COVERING-PATHS(G, T) cubre a T.

Corolario 4.1 *Sea $G=(N, A)$ un ddgraph, e_0 el arco de entrada de G, e_k el arco de salida de G. Sea $T=/d, u, X/$ una dua en G. Entonces, todo camino desde e_0 hasta e_k en COVERING-PATHS(G, T) cubre a T.*

La demostración puede verse en el Apéndice A.

La siguiente proposición establece que el digrafo obtenido reduciendo COVERING-PATHS(G, T) (usando el procedimiento REDUCE de la Figura 2.2) es un ddgraph.

Proposición 4.5 *Sea $G=(N, A)$ un ddgraph, e_0 el arco de entrada de G, e_k el arco de salida de G. Sea $T=/d, u, X/$ una dua en G. Entonces, $G^* = (N^*, A^*)$ = REDUCE(COVERING-PATHS(G, T)) es un ddgraph.*

La demostración puede verse en el Apéndice A.

El siguiente teorema establece que el ddgraph REDUCE(COVERING-PATHS(G, T)) representa todos los caminos completos en G que cubren a T.

Teorema 4.4 *Sea G un ddgraph, e_0 el arco de entrada de G, e_k el arco de salida de G. Sea $T=/d, u, X/$ una dua en G. Sea G' = REDUCE(COVERING-PATHS(G, T)).*

Entonces, $\{p$: p es un camino completo en $G'\}$ = $\{p$: p es un camino completo en G que cubre a $T\}$.

La demostración puede verse en el Apéndice A.

Por lo tanto, el ddgraph G' = REDUCE(COVERING-PATHS(G, T_1)) identifica todos los caminos en el ddgraph G que cubren a la dua T_1. Para saber si T_1 subsume a T_2, necesitamos ver si todo camino completo en G' cubre a T_2. Esto puede hacerse viendo si:

1. todo camino completo en G' pasa por d_2, y

2. todo camino completo en G' pasa por u_2, y

3. todo camino completo en G' contiene un camino def-clear con respecto a X_2 desde d_2 hasta u_2.

La Condición 1 se cumple si y solo si d_2 pertence al camino característico de G'. De la misma manera, la Condición 2 se cumple si y solo si u_2 pertenece al camino característico de G'. Suponiendo que se cumplen las Condiciones 1 y 2, sea G''=SUB-DDGRAPH(G', d_2, u_2). Entonces, la Condición 3 se cumple si y solo si ningún arco e en G'' distinto de d_2 y u_2 contiene una definición de X_2. El siguiente teorema prueba que este procedimiento funciona.

Teorema 4.5 *Sea G un ddgraph, e_0 el arco de entrada de G, e_k el arco de salida de G. Sean T_1=[d_1, u_1, X_1] y T_2=[d_2, u_2, X_2] dos duas en G. Sea G' = REDUCE(COVERING-PATHS(G, T_1)). Sea G''=SUB-DDGRAPH(G', d_2, u_2).*

Entonces, T_1 subsume a T_2 si y solo si

1. *d_2 pertenece al camino característico de G', y*

2. *u_2 pertenece al camino característico de G', y*

3. *ningún arco distinto de d_2 y u_2 en G'' contiene una definición de X_2.*

La demostración de este teorema puede verse en el Apéndice A.

En conclusión, dado un ddgraph G, y dos duas T_1 y T_2 en G, para implementar SUBSUMPTION-BETWEEN-DUAS(G, T_1, T_2), se puede usar el procedimiento de la Figura 4.8. Construyendo G' = REDUCE(COVERING-PATHS(G, T_1)) seleccionamos los caminos en G que cubren a T_1. Luego, chequeamos si todo camino que cubre a T_1 en G atraviesa d_2 y u_2, lo que puede hacerse viendo si d_2 y u_2 están en el camino de e_0 hasta e_k en el dominator tree de G'. Finalmente, para saber si todo camino en G' cubre a T_2, vemos que ningún arco distinto de d_2 y u_2 en SUB-DDGRAPH(G', d_2, u_2) contenga una definición de X_2.

Analizamos ahora la complejidad temporal de SUBSUMPTION-BETWEEN-DUAS. Construir G' = REDUCE(COVERING-PATHS(G, T_1)) lleva tiempo $O(|A|)$. Construir $DT(G')$ y ver si d_2 y u_2 están en el camino desde e_0 hasta e_k en $DT(G')$ lleva tiempo $O(|A|\alpha(|A|, |N|))$. Ver si ningún arco distinto de d_2 y u_2 en SUB-DDGRAPH(G', d_2, u_2) contiene una definición de X_2 lleva tiempo $O(|A|)$. Por lo tanto, el procedimiento SUBSUMPTION-BETWEEN-DUAS toma tiempo $O(|A|\alpha(|A|, |N|))$.

SUBSUMPTION-BETWEEN-DUAS(G: ddgraph; T_1, T_2: duas): bool;
begin
construct G' = REDUCE(COVERING-PATHS(G, T_1));
construct $DT(G')$, the dominator tree for G';
if d_2 y u_2 are in the path from e_0 to e_k in $DT(G')$ **and**
$\forall e \neq d_2, u_2,\ e \in$ SUB-DDGRAPH(G', d_2, u_2) $\Rightarrow e$ does not contain a definition of X_2
then return(TRUE)
else return(FALSE)
end

Figuras 4.8: Procedure SUBSUMPTION-BETWEEN-DUAS

4.3.4 Implementación de SUBSUMPTION-BETWEEN-CLASSES-OF-DUAS

Sea G un ddgraph, X_1, X_2 dos variables en G definidas en los arcos d_1 y d_2, respectivamente. Queremos saber si $S_{d_1}^{X_1}$ subsume $S_{d_2}^{X_2}$.

Primero identificamos para una dua dada T todas las clases de duas S_d^X tales que T garantiza el cubrimiento de S_d^X, es decir, tales que todo camino completo que cubre T, cubre a alguna dua en S_d^X. El conjunto de todas estas clases es TOP(T).

Definición 4.6 TOP

Sea G un ddgraph, T una dua de G. Entonces, TOP(T) = $\{S_d^X$: $\exists T' = [d, u, X]$, T subsume $T'\}$.

Ahora damos un procedimiento para encontrar a TOP(T), para una dua dada T. Sea $R_{All-Uses}(G)$ el reduced subsumption digraph de G y criterio All-Uses. El conujnto TOP(T) puede obtenerse visitando $R_{All-Uses}(G)$ desde el nodo que contiene a T hasta la raíz. Para cada dua $T' = [d, u, X]$ en cada nodo visitado, el conjunto S_d^X está en TOP(T).

Por las Definiciones 3.1 y 4.2, S_d^X subsume $S_{d'}^{X'}$ si y solo si para todo camino completo p que cubre la dua $T \in S_d^X$ existe una dua $T' \in S_{d'}^{X'}$ tal que p cubre T'. Entonces, la clase S_d^X subsume la clase $S_{d'}^{X'}$ si y solo si para cada dua $T \in S_d^X$, $S_{d'}^{X'} \in$ TOP(T); i.e., el cubrimiento de cualquier dua $T \in S_d^X$ garantiza el cubrimiento de $S_{d'}^{X'}$. El conjunto de clases de duas a las cuales la clase S_d^X subsume se llama SUB(S_d^X).

Definición 4.7 SUB

Sea G un ddgraph, X una variable en G definida en el arco d. Entonces, SUB(S_d^X) = $\{S_{d'}^{X'}$: $S_{d'}^{X'} \in$ TOP(T), $\forall T \in S_d^X\}$.

El siguiente teorema demuestra que las definiciones dadas pueden usarse para chequear si una clase S_d^X subsume otra clase $S_{d'}^{X'}$.

Teorema 4.6 *Sea G un ddgraph, X una variable en G definida en el arco d, X' una variable en G definida en el arco d'. S_d^X subsume $S_{d'}^{X'}$ si y solo si $S_{d'}^{X'} \in SUB(S_d^X)$.*

La demostración puede verse en el Apéndice A.

Es decir, para saber si S_d^X subsume $S_{d'}^{X'}$ necesitamos chequear si $S_{d'}^{X'} \in \mathrm{SUB}(S_d^X)$; i.e., si $S_{d'}^{X'} \in \mathrm{TOP}(T)$, $\forall T \in S_d^X$. En otras palabras, para cada $T \in S_d^X$, visitamos $R_{All-Uses}(G)$ desde el nodo que contiene a T hacia arriba, y vemos si durante esa visita pasamos por una dua T', tal que $T'=[d', u, X']$.

Analizamos ahora la complejidad temporal de SUBSUMPTION-BETWEEN-CLASSES-OF-DUAS. Construir el reduced subsumption digraph para G y el criterio All-Uses lleva tiempo $O(|D|^2|A|\alpha(|A|,|N|))$, donde D es el conjunto de todas las duas en $G = (N, A)$. El procedimiento SUBSUMPTION-BETWEEN-CLASSES-OF-DUAS(S_d^X,$S_{d'}^{X'}$,G) para evaluar si la clase de duas S_d^X subsume a la clase de duas $S_{d'}^{X'}$ lleva tiempo $O(|S_d^X| * |D| + |D|^2|A|\alpha(|A|,|N|))) \leq O(|D|^2|A|\alpha(|A|,|N|))$.

4.3.5 Implementación de SUBSUMPTION-BETWEEN-PATHS

La implementación de SUBSUMPTION-BETWEEN-PATHS es directa, ya que consiste en chequear si un camino dado es subcamino de otro camino dado. Esto puede hacerse en tiempo $O(|A|)$.

4.4 Análisis de la Complejidad

Analizamos ahora la complejidad temporal del procedimiento FIND-A-SPANNING-SET-OF-ENTITIES para un ddgraph $G = (N, A)$ (ver Figura 4.3). Sea D el conjunto de todas las duas de G. Ya que el procedimiento SUBSUMPTION toma tiempo $O(t)$, donde t es

$$\begin{cases} O(|A|\alpha(|A|,|N|)) & \text{si las entidades son arcos o duas} \\ O(|D|^2|A|\alpha(|A|,|N|)) & \text{si las entidades son clases de duas} \\ O(|A|) & \text{si las entidades son caminos} \end{cases}$$

el primer paso toma tiempo $O(|E_c(G)|^2 * t)$. Construir $S_c(G)$ toma tiempo $O(|E_c(G)|^2)$. Derivar el reduced subsumption digraph $R_c(G) = (N_{R_c(G)}, A_{R_c(G)})$ toma tiempo $O(|N_{R_c(G)}| + |A_{R_c(G)}|)$.

Seleccionar las hojas de $R_c(G)$ toma tiempo $O(|N_{R_c(G)}|)$. Por lo tanto, obtener un spanning set of entities para $G = (N, A)$ y c toma tiempo $O(|E_c(G)|^2 * t)$[2].

4.5 Aplicaciones de los Spanning Sets of Entities

En lo que sigue de este trabajo, presentaremos varias aplicaciones de los spanning sets of entities. Estudiamos tres tipos de aplicaciones:

1. El uso de spanning sets para reducir el costo del testing. Este tema se estudia en el próximo capítulo.

2. El uso de spanning sets para estimar el costo del testing. Esto se hace estimando el número de caminos de test necesarios para obtener un cubrimiento completo y se describe en el Capítulo 6.

3. Guía en la generación de caminos de test. El uso de spanning set of entities en la generación de caminos de test se presenta en el Capítulo 7.

[2]Notamos que este algoritmo tomará un tiempo exponencial en el número de arcos si el número de entidades es exponencial en el número de arcos (por ejemplo, si c es el criterio All-Paths).

Capítulo 5

Reducción del Costo del Testing

El testing emplea una cantidad considerable del tiempo y recursos gastados en la producción de software [9]. Por consiguiente, sería útil tener una manera de *reducir* el esfuerzo del testing. En particular, *el número de tests a ser ejecutados* impacta fuertemente en el esfuerzo del testing. De hecho, el tiempo y los recursos necesarios para realizar el testing aumenta con el número de casos de prueba. En este capítulo, analizamos el uso de spanning sets of entities para reducir el número de tests que deben plancarse para obtener un cierto cubrimiento durante el desarrollo del software.

En la siguiente sección, presentamos los trabajos existentes en la generación de test suites de tamaño reducido. Luego, analizamos el uso de los spanning sets of entities para reducir el número de tests en un test suite. Tradicionalmente, los datos de test necesarios para obtener un cierto cubrimiento se generan con el objetivo de cubrir el conjunto de entidades asociado al cubrimiento. Sin embargo, sabemos que no es necesario cubrir todas las entidades: es suficiente con cubrir las entidades en un spanning set of entities. Usando esta idea, en la última sección de este capítulo presentamos una nueva familia de criterios de cubrimiento estructural.

Los resultados presentados en este capítulo han sido parcialmente publicados en [14] para el caso del criterio All-Branches y en [57] para el caso del criterio All-Uses.

5.1 Trabajos Existentes

En esta sección, analizamos los trabajos existentes en la generación de test suites de tamaño reducido.

Hace varios años, el problema de hallar un conjunto de caminos que satisfaga el criterio de

cubrimiento All-Branches ha sido atacado por Krause, Smith y Goodwin. En [52], se genera un conjunto de caminos escogiendo repetidamente el camino más eficaz, identificando aquel que cubre la mayor cantidad de branches entre aquellas aún no cubiertas.

Unos años más tarde, el problema de hallar un conjunto de caminos que satisfaga el criterio de cubrimientto All-Branches con el *mínimo número de caminos* ha sido atacado por Ntafos y Hakimi [66].

En su trabajo, presentan dos métodos por resolver este problema. El problema de hallar un conjunto de caminos *mínimo* se transforma en un problema de flujo mínimo y en un problema de matching máximo [18], respectivamente. Sin embargo, cuando se usa un test suite de cardinalidad mínima para probar un programa, la presencia de loops reducirá el número de tests requerido. Esto no refleja el número de tests requerido en la práctica.

Estas soluciones están asociadas al criterio All-Branches. Otros criterios de testing no se tienen en cuenta.

Los métodos usados en la generación de un conjunto de caminos de test que satisfacen un criterio de cobertura generan un camino de test para cubrir una entidad seleccionada en forma aislada de las otras entidades. Si el dato de test generado también ejecuta otras entidades, se considerarán cubiertas a posteriori. Pero no se realiza ningún esfuerzo para generar a priori datos de test que satisfagan varios requerimientos simultáneamente. Sin embargo, observamos que en general ese caso de test cubre a más de una entidad. Así, se debe usar una manera de reducir el número de casos de test diferente.

En particular, Gupta y Soffa [36] investigaron cómo guiar la generación de test, de manera de que un único caso de test satisfaga varios requerimientos. Ellos agrupan requerimientos de cubrimiento, de manera que cada grupo pueda ser potencialmente cubierto por un único caso de test. Nuestro resultado mejora esta solución. De hecho, los spanning sets of entities proporcionan la manera óptima de agrupar entidades[1], ya que son el conjunto de entidades *mínimo* que garantiza cubrimiento total.

Un acercamiento diferente consiste en minimizar el número de tests en un test suite o el número de caminos en un conjunto de caminos de test. Este problema es NP-completo [36]. En [77], el problema de hallar un conjunto de caminos de cubrimiento es tratado como un problema de decisión y es formulado como un problema de programación entera [18], de manera de minimizar el número de caminos necesarios para satisfacer el criterio All-Branches. Se presenta un modelo

[1] Realmente, no hablamos explícitamente de grupos de entidades, pero éstos se pueden derivar fácilmente usando el orden impuesto por la relación de *subsumption* introducida en este trabajo (ver Capítulo 4).

generalizado que sirve para varias estrategias del flujo de control [77].

Otros autores han propuesto heurísticas basadas en técnicas de minimización par reducir el número de tests, por ejemplo, [40, 91]. En [40], se asocia a cada requerimiento un conjunto de tests que lo satisfacen. Luego, se selecciona un subconjunto de tests que incluye por lo menos un test del conjunto asociado a cada requerimiento. En [91], los datos de test son generados al azar. Luego, se reduce el tamaño del test suite manteniendo constante el cubrimiento de tipo All-Uses. Sin embargo, estas técnicas se aplican a un conjunto de test redundante sólo después de que el test suite ha sido generado. Así, tales soluciones no reducen realmente el esfuerzo de generar los tests.

En nuestro trabajo usamos pre-análisis para reducir el número de casos de test. De esta manera, evitamos la generación de pruebas redundantes (ver Sección 5.2.2), minimizando el número de casos de test. El uso spanning sets of entities para reducir el costo del testing se estudia en la próxima sección.

5.2 Uso de los Spanning Sets para Reducir el Costo del Testing

En esta sección discutimos el uso de los spanning sets para reducir el costo del testing, incluyendo:

1. chequeo de la completitud del cubrimiento del testing;
2. prevención de la generación de caminos redundantes;
3. guía en la selección de entidades para aumentar el cubrimiento;
4. tratamiento del problema de no-factibilidad de caminos.

5.2.1 Chequeo de la Completitud del Testing

El mejor uso de los criterios de cubrimiento es como criterios de adecuación del testing realizado. Específicamente, la proporción entre las entidades cubiertas y el número total de entidades en el programa se usa para evaluar la minuciosidad del proceso de testing sobre un programa.

Observamos que una medida más significativa es proporcionada por la proporción entre las entidades cubiertas en un spanning set y el número total de entidades en él (i.e., el número de hojas en el reduced subsumption digraph). Para medir el cubrimiento con respecto a un spanning set of entities para el criterio considerado, el tester sabrá cuántas entidades unconstrained quedan

para ser cubiertas y cuáles se deben elegir para aumentar el cubrimiento más eficazmente. Por ejemplo, si ha quedado una única hoja no cubierta, el tester sabrá que un test que cubra una entidad unconstrained que representa a esa hoja será suficiente para obtener cobertura total, independientemente de la medida de cobertura sobre el conjunto total de entidades.

Consideremos el programa SORT de la Figura 2.1. Supongamos que hemos generado el siguiente test suite para testear el programa:

Υ_1 = { i_1 = ([5,7,6],0), i_2 = ([5,7,6],1), i_3 = ([5,7,6],3), i_4 = ([7,6,5],3) }.

Los Inputs i_1 y i_2 ejecutan el camino e_1, e_8 en el ddgraph de la Figura 2.1. El Input i_3, ejecuta el camino $e_1, e_2, e_3, e_4, e_6, e_3, e_5, e_6, e_7, e_2, e_3, e_4, e_6, e_7, e_8$. El Input i_4, ejecuta el camino $e_1, e_2, e_3, e_5, e_6, e_3, e_5, e_6, e_7, e_2, e_3, e_5, e_6, e_7, e_8$.

Queremos ver si Υ_1 satisface el criterio All-Uses. Esto puede hacerse viendo si todas las duas en el ddgraph están cubiertas por Υ_1. O bien, se puede hacer viendo si el conjunto reducido de duas en un spanning set of duas S está cubierto por Υ_1. En el primer caso, necesitamos comprobar el cubrimiento de 36 duas. En el segundo caso, sólo necesitamos comprobar el cubrimiento de 10 duas. Los Inputs i_1 y i_2 cubren la dua $[e_1, e_8, sortupto]$. De las duas que quedan en S, i_3 cubre las duas $[e_7, e_4, a]$, $[e_1, e_4, a]$, $[e_1, e_5, a]$, $[e_4, e_5, mymax]$ y $[e_2, e_4, index]$; y i_4 cubre las duas $[e_7, e_5, a]$ y $[e_2, e_5, index]$. Por lo tanto, en este ejemplo las siguientes duas en S no están cubiertas por Υ_1: $[e_4,e_4,mymax]$ y $[e_2,e_7,index]$.

5.2.2 Previniendo la Generación de Caminos Redundantes

Enfocar la generación de casos de test en el cubrimiento de un spanning set of entities ayuda a prevenir la selección de caminos redundantes. De hecho, en la situación común de que nuevos caminos de test tienen que ser seleccionados para aumentar el cubrimiento, los caminos más útiles son aquéllos que cubren alguna entidad unconstrained aún no cubierta (notar que hay al menos una, o el cubrimiento sería del 100%). Si se escoge un camino de test P que cubre sólo entidades unconstrained ya seleccionadas, P será en algún momento un camino redundante. De hecho, para cada entidad no unconstrained E en P, o bien alguna de las entidades unconstrained ya cubiertas la subsume; o bien en algún momento se seleccionará otra entidad unconstrained que la subsume.

Por ejemplo, supongamos que necesitamos aumentar el cubrimiento en el ejemplo presentado en la última sección. Las duas en G que no son cubiertas por Υ_1 son las siguientes: $[e_6,e_4,index]$, $[e_4,e_4,mymax]$, $[e_1,e_7,maxpos]$, $[e_2,e_7,mymax]$ y $[e_2,e_7,index]$. Las duas en S no cubiertas por Υ_1 son las siguientes: $[e_4,e_4,mymax]$ y $[e_2,e_7,index]$. Supongamos que elegimos la dua $[e_6,e_4,index]$ para aumentar el cubrimiento; i.e., no usamos la información provista por el spanning set of

duas. En particular, si ejecutamos el camino $(e_1, e_2, e_3, e_5, e_6, e_3, e_4, e_6, e_7, e_2, e_3, e_4, e_6, e_7, e_8)$, dicha dua será cubierta. Por ejemplo, el input ([6,5,7],3) ejecuta ese camino. Sin embargo, ese input no cubre ninguna dua unconstrained no cubierta hasta ahora. Entonces, no se aumenta realmente el cubrimiento ya que todavía necesitamos cubrir las duas $[e_4,e_4,mymax]$ y $[e_2,e_7,index]$ para completar el cubrimiento. De hecho, si cubrimos la dua $[e_4,e_4,mymax]$, estamos cubriendo también la dua $[e_6,e_4,index]$.

5.2.3 Guías en la Selección de Entidades para Aumentar el Cubrimiento

Supongamos que tenemos que testear un programa y satisfacer un criterio de cubrimiento específico. Hemos seleccionado varios caminos de test, pero no se satisface el requisito de cubrimiento. Así, necesitamos seleccionar más caminos de test. En tal situación se pueden usar las entidades unconstrained para guiar la generación de los tests. De hecho, la generación de casos de test puede ser dirigida a cubrir un spanning set of entities. Ya que un spanning set of entities contendría un menor número de entidades que el conjunto completo de entidades en un programa, el esfuerzo de la generación de los casos de test puede disminuir considerablemente.

Por ejemplo, supongamos que queremos completar el conjunto Υ_1 de la Sección 5.2.1, para obtener cubrimiento de tipo All-Uses. Sabemos que si agregamos más tests a Υ_1 para cubrir las duas $[e_4,e_4,mymax]$ y $[e_2,e_7,index]$, obtendremos cubrimiento de tipo All-Uses.

Para cubrir la dua $[e_4,e_4,mymax]$, necesitamos un input que entre dos veces el loop interno del programa, entrando en la condición if las dos veces. Por ejemplo, el siguiente camino de control haría esto: $(e_1, e_2, e_3, e_4, e_6, e_3, e_4, e_6, e_7, e_2, e_3, e_5, e_6, e_7, e_8)$. Podemos seleccionar el input ([5, 6, 7[, 3), que ejecuta ese camino, y entonces cubre la dua $[e_4,e_4,mymax]$.

Para cubrir la dua $[e_2,e_7,index]$, necesitamos un input que entre el loop externo pero no entre el loop interno del programa. Esto no es posible, ya que para esto n debería ser mayor que *sortupto*, y menor que *sortupto*+1, y n y *sortupto* son variables enteras. En conclusión, esa dua no es factible y no existe ningún input que la ejecute. Por lo tanto, el cubrimiento no puede ser completado. A estas alturas, el conjunto $\Upsilon_2 = \Upsilon_1 \cup \{i_5 = ([5,6,7[, 3)\}$ es el conjunto de tests que cubre todas las duas en S - $\{[e_2,e_7,index]\}$. El cubrimiento fue aumentado. Sin embargo, el cubrimiento puede ser aumentado todavía más, como veremos en la Sección 5.2.4.

Este tema será extendido en el Capítulo 7, donde usamos los spanning sets of entities para generar caminos de test.

5.2.4 Tratamiento de No-Factibilidad

Supongamos que no puede encontrarse ningún dato de test para cubrir una cierta entidad E, i.e., E se considera *no-factible*. Entonces, toda entidad E' que subsume a E es también no-factible. De hecho, si existe un camino que cubre a E', ese camino cubriría también a E. En particular, si E es el representante de una componente fuertemente conexa M, entonces sabemos que todas las demás entidades en M son también no-factibles.

Supongamos ahora que S es un spanning set of entities que contiene a E. Entonces, en este caso un spanning set of entities no garantiza cubrimiento del 100%. En particular, E no puede ser cubierta por ningún camino en el ddgraph. Consideremos las entidades que tienen a E como uno de sus hijos en $R_c(G)$ (el reduced c-subsumption digraph, presentado en el Capítulo 4). Tenemos los siguientes dos casos. En el primer caso, supongamos que para toda entidad E' que tiene a E como uno de sus hijos en $R_c(G)$, existe una entidad unconstrained factible que subsume a E'. Entonces, E puede ser eliminada de S. En el otro caso, (i.e., $\exists E'$ que tiene a E como uno de sus hijos en $R_c(G)$ tal que ninguna entidad en S que subsume a E' es factible), entonces S debe ser modificado para garantizar el cubrimiento máximo posible. Sustituimos a E (y los otros hijos de E') en S con todas las entidades E' tales que E es un hijo de E' en $R_c(G)$ y ninguna entidad en S que subsume a E' es factible. Este conjunto de entidades garantiza el máximo cubrimiento posible.

Notamos que hemos usado la información que contiene $R_c(G)$. De hecho, la estructura misma del reduced c-subsumption digraph puede ser útil para el testing.

En el ejemplo presentado en la Sección 5.2.1, ha sido demostrado que la dua $|e_2,e_7,index|$ es no-factible. El conjunto de tests Υ_2 cubre todas las duas en S - $\{|e_2,e_7,index|\}$. Y, a partir de la información del reduced subsumption digraph $R_{All-Uses}(SORT)$ de la Figura 4.2, sabemos que Υ_2 cubre a todas las duas con excepción de $|e_1,e_7,maxpos|$, $|e_2,e_7,mymax|$ y $|e_2,e_7,index|$.

Entonces, para aumentar el cubrimiento necesitamos reemplazar a la dua $|e_2,e_7,index|$ en S con la dua $|e_2,e_7,mymax|$ (o la dua $|e_1,e_7,maxpos|$). En este caso, el input i_4 cubre a la dua $|e_2,e_7,mymax|$, y por lo tanto el conjunto $\{i_1, i_2, i_3, i_4, i_5\}$ es el conjunto de datos de test que garantiza el máximo cubrimiento posible en el ejemplo considerado.

5.3 Resultados Experimentales

Como en el caso de muchas técnicas de reducción del costo en el área de testing, el costo de aplicar la técnica se debe comparar con el ahorro real que produce. En este caso, es importante

investigar empiricamente la proporción entre el número total de entidades y el número menor de entidades en un spanning set. Ya que la identificación de un spanning set of entities toma tiempo polinómico en el número de entidades y se puede automatizar, es de esperar que sea mucho más eficaz que seleccionar tests al azar hasta que se alcance un cubrimiento del 100%.

En esta sección presentamos algunos resultados experimentales preliminares en la reducción real del número de entidades para el caso de cubrimiento All-Uses. De esta manera queremos estimar en cuánto se reduce el costo del testing cuando se usan spanning sets of entities.

Dado un programa en lenguaje C, queremos calcular el conjunto total de duas en el programa y elegir un spanning set of duas. Luego, queremos comparar los tamaños de estos dos conjuntos. Estas funcionalidades han sido implementadas en un prototipo de una herramienta llamado UAT (por All-Uses Analysis and Testing). Para obtener el conjunto total de duas, UAT usa ATAC [43]. ATAC hace análisis del flujo de datos de programas C. Luego, UAT construye un spanning set of duas para el criterio All-Uses.

Se ha hecho experimentación con esta funcionalidad de UAT. Se ha usado sobre varios programas tomados de [90]. En la Tabla 5.1 presentamos algunos resultados preliminares del experimento de la reducción del número de duas al usar spanning sets en cubrimiento de tipo All-Uses. Para cada programa C considerado, damos el número de duas, el número de duas en un spanning set y la reducción de uno a otro. Podemos observar una reducción real en el número de duas a ser consideradas. Sin embargo, aún cuando estos resultados prometen, se necesita más experimentos para apoyar la hipótesis de que usar spanning sets para seleccionar tests es mucho más eficaz que seleccionar tests al azar hasta obtener cubrimiento del 100%.

5.4 Una Nueva Familia de Criterios de Cubrimiento de Testing

En esta sección introducimos una familia nueva de criterios de cubrimiento de testing. Hemos visto que para satisfacer un criterio de cubrimiento dado, podemos tratar de cubrir las entidades en un spanning set. El resultado final será el mismo que tratar de cubrir las entidades en el ddgraph, pero el proceso nuevo será probablemente más barato que el tradicional.

Basándonos en esta idea, para cada criterio de cubrimiento c, definimos un nuevo criterio de cubrimiento *Spanning* c[2] como sigue. Supongamos que para satisfacer el criterio de cubrimiento c necesitamos cubrir todas las entidades en el conjunto $E_c(G)$.

[2]Spanning All-Statements criterion, Spanning All-Branches criterion, etc.

Function	#duas	#duas in a spanning set	reduction (in %)
sort	34	10	70.6
find	95	50	27.4
copi	17	4	73.5
qsorti	37	11	70.3
print_buf	52	17	67.3
animate	82	48	41.5
if	27	14	48.1
save_pic	34	12	65.7
getop	77	34	55.8
itoa	16	5	68.8
base	28	10	64.3
prime	20	10	50.0
length	28	16	42.9
getone	73	35	52.1
substitute	66	42	36.4

Tablas 5.1: Reducción en el número de duas

Supongamos que U es un spanning set of entities para G y c. Entonces, el nuevo criterio de cubrimiento *Spanning c* se define de manera de que para satisfacer *Spanning c* necesitamos cubrir todas las entidades en U.

Para cada criterio de cubrimiento c presentado en el Capítulo 3, definimos un criterio de cubrimiento nuevo *spanning c* como dijimos más arriba. Estos criterios nuevos se llaman *spanning test coverage criteria*. La familia nueva de criterios satisface todo las propiedades buenas de spanning set of entities estudiadas en esta tesis.

Consideremos los criterios redefinidos en el Capítulo 3. Para los programas que contienen caminos no-factibles, puede ser imposible satisfacer un criterio dado. Notamos que para cada criterio y cada programa no es decidible si existen datos de test que testean adecuadamente el programa. Entonces, en [31], Frankl y Weyuker presentan una familia de criterios de adecuación derivados de los criterios de testing presentados en el Capítulo 3. Estos criterios se obtienen a partir de los criterios correspondientes eliminando las entidades no factibles. Son llamados *feasible test coverage criteria*. La ventaja de estos criterios factibles sobre los criterios tradicionales es que para todo programa y todo criterio existe algún conjunto de caminos que satisface ese criterio. La desventaja es que no es decidible si un conjunto de caminos particular es adecuado para un programa particular y un criterio de cubrimiento particular.

De la misma manera podemos definir una familia de *criterios spanning factibles*, basados en los criterios de cubrimiento de test introducidos más arriba. Un criterio en esta familia tiene la propiedad que cada entidad considerada es factible, y que es un conjunto mínimo con esa propiedad.

Capítulo 6

Estimación del Costo del Testing

El testing toma una cantidad considerable del tiempo y los recursos gastados en la producción de software [9]. Por consiguiente, sería útil tener una manera de *estimar* este esfuerzo de testear. De hecho, saber de antemano cuánto esfuerzo se necesitará para testear un programa dado, es esencial para que los managers puedan planear los recursos requeridos durante el proceso de desarrollo. Idealmente, se requiere una estimación del costo global de la fase de testing. Esta estimación es difícil de obtener, debido a la multiplicidad de factores involucrados, muchos de los cuales no son medibles de una manera objetiva (por ejemplo, la especialización del tester, o incluso su compromiso). Sin embargo, una cota en el número de tests que tienen que ser ejecutados para satisfacer una estrategia de test puede usarse para estimar el esfuerzo necesario para llevar a cabo los tests. De hecho, el *número de tests* a ser ejecutados es un factor importante del esfuerzo de testear un programa. El tiempo y los recursos requeridos para el testing aumenta a medida que aumenta el número de casos de test necesarios.

En este capítulo analizamos el problema de establecer cuántos casos de test se deben planear para alcanzar un cubrimiento particular durante el desarrollo del software. En particular, estudiamos la aplicación de spanning sets of entities para estimar este número.

El cómputo de una cota del número de casos de test necesarios para alcanzar un cubrimiento particular ya ha sido estudiado en la literatura. En la próxima sección presentamos y analizamos análisis teórico [66, 83, 64, 6] y resultados experimentales [16, 79] introducidos durante los últimos años. Luego, introducimos nuestras cotas para el criterio de cubrimiento All-Branches. Definimos intuitivamente qué contamos, definimos formalmente las cotas α y β, y discutimos su relación con trabajos anteriores. Además, presentamos algunos resultados experimentales en

la cota $\beta_{All-Branches}$. Luego, generalizamos estos resultados para los criterios de cubrimiento introducidos en el Capítulo 3. Finalmente, presentamos conclusiones y sugerimos desarrollos futuros.

Los resultados introducidos en este capítulo se han publicado parcialmente en [14] para el caso del criterio All-Branches y en [57] para el caso del criterio All-Uses.

6.1 Trabajos Previos

En esta sección discutimos las cotas existentes en el número de casos de test requeridos para alcanzar un cubrimiento particular. Primero, presentamos análisis teóricos llevados a cabo durante los años últimos, y luego introducimos varios resultados experimentales.

En [83], Weyuker determina *cotas superiores* del número de tests necesarios para satisfacer cada criterio en la familia [73]. Sea $G = (N, A)$ un ddgraph representando al programa σ. Sea v el número de definiciones en σ, y supongamos que G contiene d nodos decisión (bi-direccionales). Entonces, ella estableció que los criterios All-Statements y All-Branches requieren a lo sumo $d + 1$ casos de test. All-Defs requiere a lo sumo v casos de test. All-p-Uses/Some-c-Uses, All-c-Uses/Some-p-Uses, All-p-Uses y All-Uses requieren a lo sumo $1/4(d^2 + 4d + 3)$ casos de test. All-du-Paths requiere a lo sumo 2^d casos de test.

Ntafos [64] también estableció *cotas superiores* del número de tests necesarios para satisfacer varios criterios estructurales. Si $G = (N, A)$ es un ddgraph representando un programa, All-Paths puede requerir un número infinito de caminos de test. All-du-Paths puede requerir un número de caminos de test que es función exponencial de $|A|$. All-Uses, All-p-Uses/Some-c-Uses, All-c-Uses/Some-p-Uses, All-p-Uses pueden requerir un número de caminos de test de orden $O(|A|)$. Para All-Branches, All-Defs y All-Statements esta cota es función lineal de $|A|$.

Varios años atrás, McCabe [58] propuso el uso del número ciclomático $v(G)$ [75] del grafo de un programa $G = (N, A)$, con

$$v(G) = |A| - |N| + 2,$$

no sólo para medir la complejidad estructural del programa considerado, si no también como una base metodológica para el testing. El número ciclomático del flowgraph del programa es a menudo (mal)usado como una cota en el número de casos de test necesarios para testear el programa asociado. Esta propuesta extensamente usada es sin embargo criticable [9]. De hecho, el uso de McCabe del número ciclomático [58] provee el número de casos de test necesarios para testear el

programa si se usa el criterio de testing llamado "structured testing" [59] y no está relacionado con ningún otro criterio estructural. En la literatura de grafos, el número ciclomático, también conocido como "circuit rank" [75], es el máximo número de ciclos linealmente independientes. Para un grafo G con $|N|$ nodos, $|A|$ ejes y $|P|$ componentes, $v(G) = |A| - |N| + |P|$. En realidad, no hay ninguna relación entre los ciclos y los caminos en un flowgraph. Sin embargo, este número se usa comúnmente para predecir el número de casos necesarios para testear un programa.

Unos años más tarde, el problema de hallar un conjunto de caminos que satisfaga cubrimiento All-Branches con el número del mínimo de caminos, ha sido abordado por Ntafos y Hakimi [66], usando teoría de redes. Ellos generalizaron el teorema de Dilworth [24], originalmente para grafos no-dirigidos, para el caso de grafos dirigidos. También dan dos métodos para encontrar ese número. A pesar de proveer una base matemática sólida, creemos que no es una cota útil para el caso de cubrimiento All-Branches. Antes de discutir este problema, introducimos una propuesta más reciente, que coincide con la de Ntafos y Hakimi con respecto al cubrimiento All-Branches.

Más recientemente, una teoría unificada ha sido introducida por Bache y Müllerburg [6] para medir la *testability* o *testabilidad*, es decir, el número de casos de test requerido por varias estrategias de test basadas en el flujo de control (estrategias All-Paths, Simple Path [72], Structured Testing [59], All-Branches y All-Statements). Ellos aplicaron los resultados de la teoría de descomposición de programas de Fenton-Whitty [27]. Esta teoría asume que existe un número de flowgraphs básicos, llamados *primos*, y que cualquier flowgraph estructurado puede construirse mediante la composición recursiva de estos primos, mediante secuencia o inclusión anidada. De la misma forma, un flowgraph puede descomponerse en primos. En consecuencia, se pueden constuir "métricas jerárquicas", simplemente asignando un valor $m(G)$ a cada primo y definiendo cómo calcular $m(G)$ para las operaciones de secuencialidad y anidamiento.

De esta manera, Bache y Müllerburg computaron el mínimo número de casos de test requerido para satisfacer varias estrategias estructurales. Observamos que, en aprticular, para el caso de All-Branches, el número calculado coincide con la cota de Ntafos y Hakimi en la generalización del teorema de Dilworth [66].

Por otro lado, se han hecho estudios experimentales en el número de casos de test necesarios para alcanzar cubrimientos particulares. El criterio All-du-Paths fue estudiado experimentalmente en [16]. En casi todos los 143 subprogramas analizados, el criterio fue satisfecho testeando un "número razonable" de caminos completos.

En [79], Weyuker describe un estudio empírico del costo de los criterios All-c-Uses, All-p-Uses, All-Uses y All-du-Paths. Para los programas analizados, encontró que cada criterio es lineal en

el número de nodos decisión en el "peor caso empírico".

6.2 Estimando el Costo del Criterio All-Branches

En esta sección analizamos por qué las medidas teóricas propuestas hasta el momento no son útiles desde el punto de vista de la práctica del testing. Luego, proporcionamos alternativas teóricas a la pregunta de cuántos casos de test se requieren para obtener cubrimiento All-Branches.

6.2.1 Análisis de las Cotas Existentes

Ahora analizamos por qué las medidas teóricas propuestas hasta el momento no son útiles desde el punto de vista de la práctica del testing, para el criterio All-Branches. En realidad, los mismos problemas ocurren con todos los tipos de cubrimiento estructurales.

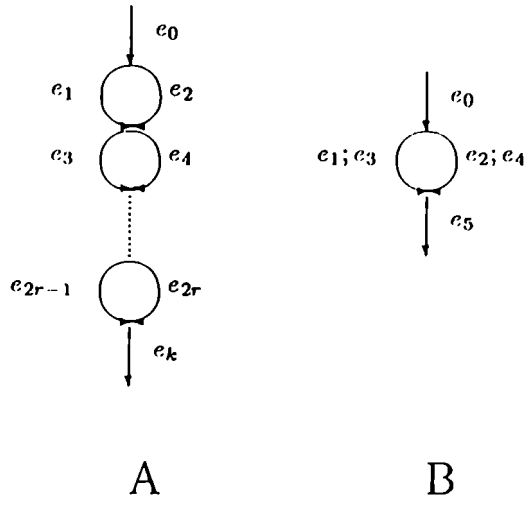
Primero, notamos que el número de casos de test requerido en práctica es normalmente considerablemente menor que lo implicado por las cotas superiores [64, 16, 79].

Ahora mostramos por qué el número ciclomático no es el número que buscamos. Considere por ejemplo el simple programa de la Figura 6.1A, que consiste en un número r de *ifs* en cascada. A través de inspección directa, podemos ver que necesitamos dos casos para satisfacer el criterio All-Branches. Sin embargo, $v(G)$ es $r+1$. En [58], McCabe explica por qué el número ciclomático es 3. La respuesta es que esto pasa porque el programa que realmente testeamos al ejecutar sólo dos caminos podría ser representado por un flowgraph menos complejo. Se obtiene este flowgraph quitando una decisión, por ejemplo como en la Figura 6.1B. Sin embargo, este argumento no es suficiente para convencernos de que se deben planear $r+1$ casos de test en práctica para el criterio All-Branches en el programa de la Figura 6.1A. El testing de tipo All-Branches debe cubrir todas las ramas posibles en el programa, y ésto puede ser hecho planificando sólo dos casos de test.

Ahora analizamos el acercamiento de Bache y Müllerburg [6] para medir la *testability*. Por ejemplo, consideramos de nuevo el programa de la Figura 6.1A, obtenida poniendo en secuencia r primos IF-THEN-ELSE.

Para el criterio All-Branches, cada primo IF-THEN-ELSE requiere dos casos de test, y la función del secuenciamiento para r flowgraphs $F_1 ..., F_r$ mantiene el máximo valor de testability entre los flowgraphs en la sucesión. Entonces, el número de casos de test requerido sería dos, como uno intuitivamente piensa.

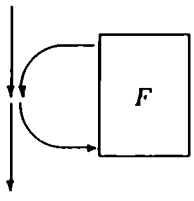
Sin embargo, aunque esta medida es correcta de un punto de vista teórico, creemos que aparecen problemas en la práctica si queremos testear programas con loops. Para ver por qué,



Figuras 6.1: Número de casos de test: ifs en cascada

basta con considerar como un ejemplo la medida de Bache y Müllerburg para un flowgraph obtenido anidando un flowgraph de complejidad arbitraria en un primo WHILE-DO (ver la Figura 6.2). La testability siempre da uno, i.e., se requiere para el criterio All-Branches un único caso de test. Más en general, cuando se usa un conjunto de tests de cardinalidad mínima para medir la testability del programa, la presencia de loops reducirá el número de casos de test requeridos, sin reflejar el número de casos requeridos en la práctica.

Resumiendo, hemos examinado las métricas de cubrimiento All-Branches existentes, y hemos concluido que no cumplen sus propósitos, principalmente porque representan una medida matemática sin un significado asociado al testing, i.e., no se tiene en cuenta el uso de este número en la práctica. Creemos que una especificación clara no sólo de qué atributo se mide, si no también de *por qué* se mide se debe aclarar antes de dar la definición de tal medida. En este capítulo introducimos métodos nuevos para evaluar el número de casos de test necesarios para alcanzar distintos cubrimientos.



Figuras 6.2: Anidando un flowgraph F de complejidad arbitraria en un primo WHILE-DO

6.2.2 La Cota $\alpha_{All-Branches}$

Primero, introducimos un resultado que será útil en lo que sigue.

Proposición 6.1 *Sea $G = (N, A)$ un ddgraph. Entonces, existe un único spanning set of entities S para G y el criterio All-Branches.*

La demostración puede verse en el Apéndice A.

Como una consecuencia, un arco e es unconstrained si y sólo si e pertenece al único spanning set.

Ahora notamos que los spanning set of entities son útiles para estimar el costo del testing. En particular para el criterio All-Branches, el conjunto de entidades es el conjunto de los arcos del ddgraph del programa, y un spanning set of entities es un subconjunto de los arcos del ddgraph. Ya que el cubrimiento de spanning set of entities garantiza un cubrimiento del 100%, se puede tomar a la cardinalidad de tal conjunto como una estimación del número de casos de test necesario para satisfacer el criterio All-Branches. Para ser precisos, éste es el número de casos de test requerido en la peor hipótesis de que se toma un camino diferente para cubrir cada arco en un spanning set. En este sentido, el número de arcos unconstrained en un spanning set se puede ver como una cota inferior "segura" del número de casos de test necesarios para satisfacer el criterio All-Branches.

Definición 6.1 $\alpha_{All-Branches}$

Sea $G = (N, A)$ un ddgraph, $S \subseteq A$ un spanning set of entities para G y el criterio All-Branches. Entonces, $|S|$ una cota inferior del número de casos de test necesarios para satisfacer el criterio All-Branches. Esta cota será llamada $\alpha_{All-Branches}$, i.e.,

$$\alpha_{All-Branches}(G) = |S|.$$

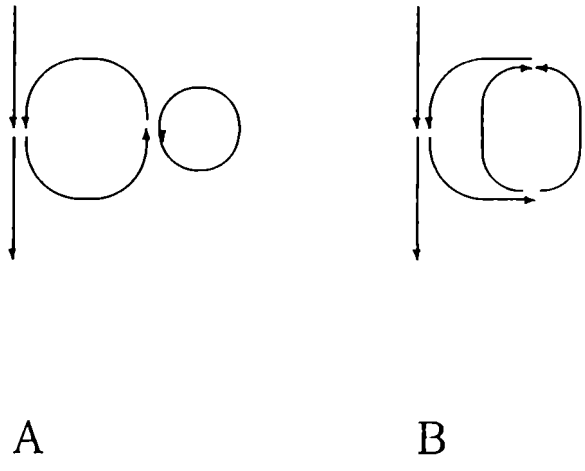
6.2.3 La Cota $\beta_{All-Branches}$

Hemos dicho que $\alpha_{All-Branches}$ es una cota inferior del número de caminos de test necesarios para satisfacer el criterio All-Branches, considerando un camino diferente para cubrir cada branch en S. Sin embargo, un camino generalmente cubre más de un arco unconstrained. Entonces, el número real de casos de test depende de cómo se combinan en los caminos completos los arcos unconstrained. Por ejemplo, podríamos encontrar el número mínimo de tests en teoría, combinando en caminos completos tantos arcos unconstrained como sea posible. Pero esta cota mínima podría no ser útil en la práctica. De hecho, cuanto más complejo (i.e., más largo) es un camino, es más probable que el camino no sea factible [9]. Es un hecho normalmente aceptado que el problema *real* de las estrategias de test basadas en el flujo de control es derivar caminos de test que sean *ejecutables* [42]. Entonces, argumentamos que el conjunto de caminos de test que se debe pensar para satisfacer un criterio particular, y, en particular, el criterio All-Branches, debe ser un conjunto con *significado*, es decir, correspondiente a un uso posible del programa y, lo que es más importante, factible. Algunos trabajos experimentales recientes [93] han dado evidencia estadística a la idea muy intuitiva que es más probable que sean factibles los caminos con un bajo número de predicados. Basándonos en este hecho, desarrollamos un método para combinar arcos unconstrained para formar caminos que sean lo más cortos posible.

Con este propósito, introducimos las nociones siguientes. Consideraremos "con significado" a los caminos que contienen un bajo número de decisiones. En particular, tales caminos no combinan arcos unconstrained que requieren entrar en diferentes loops (obviamente, excluye el caso de loops anidados), y si se entra en un loop, entonces será iterado sólo una vez. En nuestra propuesta, capturamos esta noción intuitiva de combinar arcos unconstrained para formar caminos con significado dentro de un marco matemático riguroso, introduciendo la noción de *incomparabilidad débil*. Este concepto nos permite establecer formalmente la noción de caminos de test con significado introducido más arriba.

Para ejemplificar nuestra propuesta, consideramos el caso muy simple de un primo WHILE-DO y un primo IF-THEN-ELSE ambos anidados en un primo WHILE-DO (Figuras 6.3A y 6.3B). ¿Cuántos casos de test se deben planear para alcanzar cubrimiento All-Branches? McCabe diría tres en ambos casos, y Bache y Müllerburg (con Ntafos y Hakimi) dirían uno en ambos casos.

Nosotros decimos que planificar un caso de test es suficiente para el flowgraph en la Figura



Figuras 6.3: Número de casos de test: Programas simples con loops

6.3A. De hecho, un camino de test que ejercita el WHILE-DO interno[1] cubriría cualquier otro arco. Y, recíprocamente, se debe ejercer necesariamente por lo menos una vez ese arco para alcanzar cubrimiento All-Branches. Así, en este ejemplo, estamos de acuerdo con el número mínimo teórico. En cambio, para el flowgraph en la Figura 6.3B, decimos que deben ser planeados dos casos de test: uno que cubre el THEN y el otro que cubre el ELSE de la instrucción IF-THEN-ELSE. Es verdad que en teoría, si todos los caminos fueran ejecutables, entonces sería suficiente con un único camino. Pero en práctica, esto requeriría hallar un input que forzara la ejecución del flujo de control a entrar en el WHILE-DO por lo menos dos veces, pasando una vez por el THEN y una vez por el ELSE. Este camino podría ser factible, pero posiblemente no lo sería. Más pragmaticamente, diríamos que se deben encontrar dos inputs par hacer el test, uno que ejecute el THEN y otro que ejecute el ELSE.

6.2.4 Definición de la Cota $\beta_{All-Branches}$

Ahora formalizamos las nociones introducidas más arriba. Buscamos la cardinalidad de un conjunto de caminos "con significado" para garantizar cubrimiento All-Branches. Con este propósito, buscamos un conjunto de caminos que cubran el conjunto de arcos unconstrained; de hecho, la definición un spanning set garantiza que este conjunto de caminos cubrirá todo arco en el ddgraph. Más precisamente, buscamos un conjunto de caminos p que agrupe los arcos unconstrained de

[1] Notamos que éste es el único arco unconstrained para este ddgraph.

una manera útil; i.e., requerimos que un camino en $\wp$ cubre a más de un arco unconstrained sólo si se satisfacen algunas condiciones precisas. En particular, si un camino $p \in \wp$ cubre un arco unconstrained e perteneciente a un ciclo, el camino p no puede entrar al ciclo más de una vez, y no puede entrar en otro ciclo en el ddgraph (con la excepción obvia de posibles ciclos anidados).

Ahora definiremos cuándo dos arcos en G no pueden ser cubiertos por el mismo camino. En este caso los llamaremos dos arcos *débilmente incomparables*. Intuitivamente, dos arcos son débilmente incomparables si:

1. cada uno de ellos puede ser cubierto por un camino sin loops, pero los dos juntos no pueden ser cubiertos por el mismo camino sin loops, o
2. uno de ellos puede ser cubierto por un camino sin loops y el otro no se puede cubrir con un camino sin loops, o
3. ambos pertenecen al mismo ciclo en G y uno alcanza al otro sólo entrando al ciclo por lo menos dos veces, o
4. ambos pertenecen a ciclos diferentes en G (no anidados uno dentro del otro).

Por ejemplo, consideramos el programa C de la Figura 6.4, la función "GETOP" tomada de [50]. El ddgraph correspondiente G_{GETOP} se presenta en la Figura 6.5. En el ddgraph G_{GETOP}:

- los arcos e_{12} y e_{13} son (débilmente) incomparables (de acuerdo al punto 1);
- los arcos e_{12} y e_{16} son débilmente incomparables (de acuerdo al punto 2);
- los arcos e_{16} y e_{17} son débilmente incomparables (de acuerdo al punto 3), pero no son incomparables: uno alcanza al otro entrando al ciclo dos veces;
- los arcos e_{16} y e_1 son débilmente incomparables (de acuerdo al punto 4);
- los arcos e_{16} y e_{18} no son débilmente incomparables, porque perteneces al mismo ciclo y e_{16} alcanza a e_{18} entrando al ciclo sólo una vez.

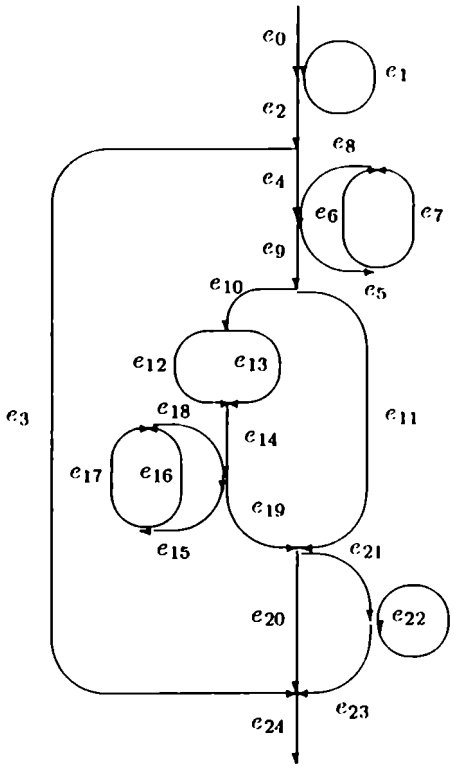
Definimos formalmente esta relación:

Definición 6.2 *Arcos Débilmente Incomparables*

Sea $G = (N, A)$ un ddgraph. Sea $e, e' \in A' \subseteq A$ y $e \neq e'$. Los arcos e y e' son llamados dos arcos débilmente incomparables en A' si para todo camino completo en G que contiene e y e', es siempre posible derivar un pseudo-camino que contiene sólo a uno de ellos y que es un camino completo en G.

```
getop (s, lim) /* get next operator or operand */
char s[ ];
int lim;
{
int i, c;
while ((c=getch()) == ' ' || c == ' \t' || c == '\n' ) /* decision node 1 */
    ;
if (c != '.'  && (c<'0' || c>'9' ))                    /* decision node 2 */
    return(c);
s[0]=c;
for (i=1; (c=getchar()) ≥ '0' && c ≤ '9'; i++)      /* decision node 3 */
    if (i < lim)                                    /* decision node 4 */
        s[i]=c;
if (c=='.')  { /* collect fraction */                  /* decision node 5 */
    if (i < lim)                                    /* decision node 6 */
        s[i]=c;
    for (i++; (c=getchar()) ≥ '0' && c ≤ '9'; i++) /* decision node 7 */
        if (i < lim)                                /* decision node 8 */
            s[i]=c;
}
if (i < lim) { /* number is ok */                    /* decision node 9 */
    ungetch(c);
    s[i] = ' \0';
    return (NUMBER);
} else { /* it's too big; skip rest of line */
    while ( c != ' \n' && c != EOF)                 /* decision node 10 */
        c=getchar();
    s[lim-1]=' \0';
    return (TOO BIG);
}
}
```

Figuras 6.4: Program GETOP



Figuras 6.5: Ddgraph G_{GETOP}

Ahora, analizamos en general cuántos caminos necesitamos para cubrir todos los arcos en un subconjunto $A' \subseteq A$. Suponemos que hemos identificado un conjunto de arcos débilmente incomparables L en A' de máxima cardinalidad, i.e., dos arcos cualesquiera en L son débilmente incomparables, y todo arco en $A' - L$ no es débilmente incomparable con algún arco en L. Entonces, podemos construir un conjunto $\wp$ de $|L|$ caminos "con significado", tal que cada arco en L es cubierto por un camino diferente en $\wp$. Sea S el spanning set of entities (arcos) para G y el criterio All-Branches. En particular, nos interesamos en el caso $A' = S$. De hecho, si $A' = S$, este conjunto de caminos contiene caminos que cubren todos los arcos en A. Presentamos una definición formal del conjunto de máxima cardinalidad de arcos débilmente incomparables:

Definición 6.3 *Conjunto de Máxima Cardinalidad de Arcos Débilmente Incomparables (Largest Weakly Incomparable Arc Set)*

Sea $G = (N, A)$ un ddgraph, $A' \subseteq A$. $LWI(A')$ es un largest weakly incomparable arc set para A' en G si es un subconjunto de A' que satisface:

- *para todo $e, e' \in LWI(A')$, $e \neq e'$, entonces e y e' son débilmente incomparables,*
- *$|LWI(A')| = max\{|A''|: A'' \subseteq A'$ y para todo $e, e' \in A''$, si $e \neq e'$ entonces e y e' son débilmente incomparables$\}$.*

Notamos que dado un ddgraph y un subconjunto no vacío de arcos A', hay por lo menos un largest weakly incomparable set of arcs para A', y posiblemente no es el único. Por ejemplo, sea S el spanning set para G_{GETOP} y el criterio All-Branches,

$S = \{e_1, e_3, e_6, e_7, e_{11}, e_{12}, e_{13}, e_{16}, e_{17}, e_{20}, e_{22}\}$.

Entonces, $LWI(S) = \{e_1, e_3, e_6, e_7, e_{11}, e_{12}, e_{13}, e_{16}, e_{17}, e_{22}\}$ es el único largest weakly incomparable arc set para S en el ddgraph G_{GETOP} de la Figura 6.5.

Ahora, juntando las nociones de spanning sets y de arcos débilmente incomparables, introducimos una definición formal de la cota $\beta_{All-Branches}$, como el máximo número de arcos unconstrained que es mutuamente débilmente incomparable. En otras palabras, si S es spanning set de arcos, esta cota considera un camino diferente por cada arco en $|LWI(S)|$.

Definición 6.4 $\beta_{All-Branches}$

Sea $G = (N, A)$ un ddgraph y S el spanning set of entities (arcos) para G y el criterio All-Branches. Entonces, $\beta_{All-Branches}$ se define como la cardinalidad de un largest weakly incomparable arc set para S en G, i.e.,

$$\beta_{All-Branches}(G) = |LWI(S)|.$$

Por ejemplo, $\beta_{All-Branches}(G_{GETOP}) = |\{e_1, e_3, e_6, e_7, e_{11}, e_{12}, e_{13}, e_{16}, e_{17}, e_{22}\}| = 10$.

6.2.5 Cálculo de la Cota $\beta_{All-Branches}$

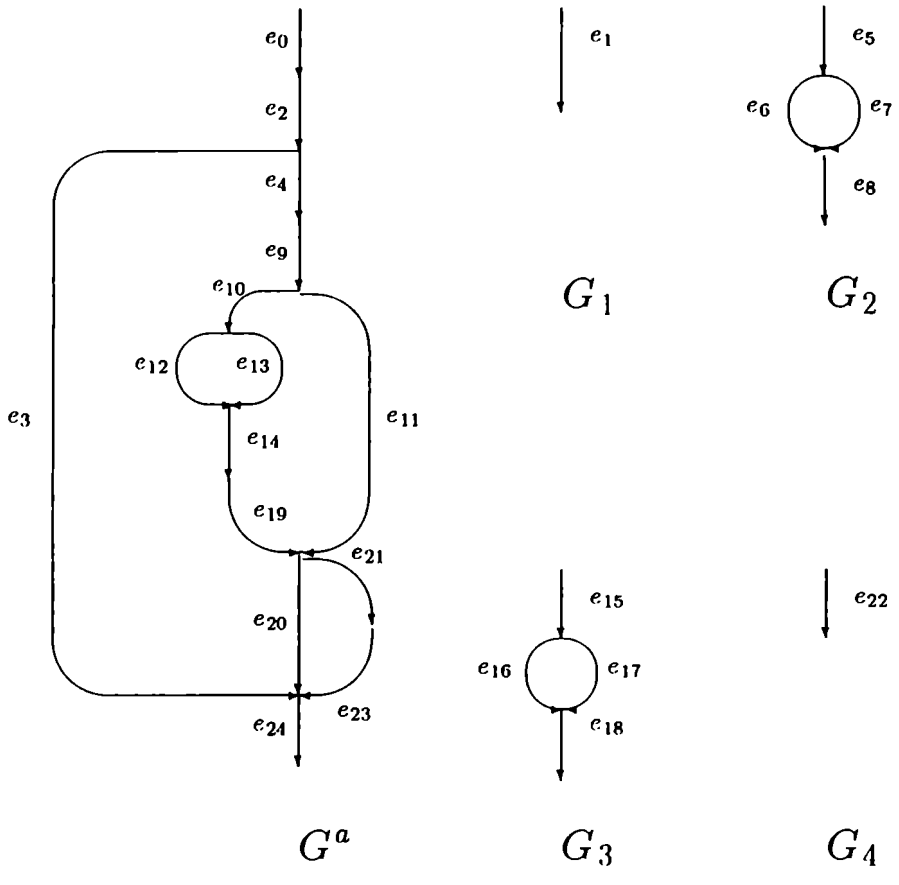
Ahora damos un esbozo de un método que permite calcular el valor de $\beta_{All-Branches}$ para un ddgraph G dado. Observamos lo siguiente:

1. Supongamos que existe un conjunto de caminos sin loops $p = \{p_1..., p_n\}$ tal que cubre cada arco en $G = (N, A)$. Entonces, el número mínimo de caminos necesario para cubrir todos los arcos en el spanning set S para G y el criterio All-Branches (y como consecuencia, todos los arcos en G) se puede calcular resolviendo un problema de flujo mínimo (MIN-FLOW)[2]. En particular, supongamos que asociamos a cada arco en el conjunto S una restricción de capacidad mínima de valor uno, y asociamos a los arcos en $A - S$ una restricción de capacidad mínima de valor cero. Entonces, el valor de una solución a MIN-FLOW con esas restricciones será el número de arcos en un largest weakly incomparable arc set para S.

2. Ahora, supongamos que no existe ningún conjunto de caminos sin loops que cubran todos los arcos en $G = (N, A)$. Entonces, hay un subconjunto de arcos que sólo puede cubrirse visitando un ciclo en G (i.e., cualquier camino completo en G que contiene un arco e en este subconjunto, contiene un loop). Podemos descomponer el ddgraph G en un digrafo conexo G^a que contiene sólo aquellos arcos que pueden ser cubiertos por caminos sin loops, y un conjunto $G_1..., G_r$ de subgrafos maximales conexos de G, que no contienen arcos en G^a.

Estos dos hechos nos permiten diseñar un procedimiento para calcular el valor de $\beta_{All-Branches}$, descomponiendo un ddgraph G en un subgrafo acíclico y un conjunto de componentes con ciclos. Luego, visitando el ddgraph, podemos identificar los arcos que no pueden ser cubiertos por un camino sin loops. Entonces, consideramos el subgrafo G^a de G que contiene sólo aquellos arcos que pueden ser cubiertos por caminos sin loops. Derivamos el conjunto $S \cap A^a$, i.e., el subconjunto de los arcos unconstrained de G que son arcos de G^a. Podemos calcular el máximo número de arcos débilmente incomparables en $S \cap A^a$, resolviendo un problema MIN-FLOW asociado. Por otro lado, derivamos las componentes maximales cíclicas $G_1..., G_r$ de G, que no están contenidas en G^a. Analogamente, para cada una de ellas, derivamos el conjunto $S \cap A_i$, i.e., el subconjunto de los arcos unconstrained de G que son arcos en G_i. Podemos calcular el máximo número de arcos débilmente incomparables para el conjunto $S \cap A_i$, para cada G_i, aplicando recursivamente

[2]Se puede resolver un MIN-FLOW problem en tiempo $O(|N||A|)$, usando una modificación de los algoritmos usados para resolver el problema de flujo máximo [28].



Figuras 6.6: Decomposición de G_{GETOP}

el mismo procedimiento. Finalmente, sumamos los valores obtenidos para cada componente de G y obtenemos el valor de $\beta_{All-Branches}$ correspondiente al ddgraph G, i.e.,

$|LWI(S)| = |LWI(S \cap A^a)| + |LWI(S \cap A_1)| + ... + |LWI(S \cap A_r)|$.

Como ejemplo, consideramos el ddgraph G_{GETOP} y su descomposición presentada en la Figura 6.6. Tenemos:

$S \cap A_1 = \{e_1\}$, entonces $|LWI(S \cap A_1)| = 1$;

$S \cap A_2 = \{e_6, e_7\}$, entonces $|LWI(S \cap A_2)| = 2$;

$S \cap A_3 = \{e_{16}, e_{17}\}$, entonces $|LWI(S \cap A_3)| = 2$;

$S \cap A_4 = \{e_{22}\}$, entonces $|LWI(S \cap A_4)| = 1$;

$S \cap A^a = \{e_3, e_{11}, e_{12}, e_{13}, e_{20}\}$, entonces $|LWI(A \cap A^a)| = 4$.

De hecho, ya sabíamos que en este caso $|LWI(S)| = 10$.

6.2.6 Discusión

En esta sección analizamos la relación entre las cotas nuevas y los números de testabilidad introducidos por Bache y Müllerburg [6].

Observamos que, por un lado, nuestra propuesta de seleccionar caminos de test que cubren los arcos unconstrained se puede considerar como una nueva estrategia de cubrimiento de test, presentada en el Capítulo 5 con el nombre de "spanning All-Branches criterion". Por otro lado, observamos que nuestra propuesta de seleccionar caminos de test "con significado" se puede considerar como otra estrategia de cubrimiento nueva, que referiremos con el nombre de "meaningful All-Branches criterion". Así, debe ser posible computar la testabilidad de estas estrategias nuevas a través del método general de Bache y Müllerburg. Para hacer esto, necesitamos saber cómo computar $\alpha_{All-Branches}$ y $\beta_{All-Branches}$, respectivamente, para los primos considerados así como para las funciones de secuencialidad y anidamiento.

En lo que sigue, P_1 se corresponde con el flowgraph *trivial*, i.e., el flowgraph que consiste en un único arco;

D_0 con el primo IF-THEN; D_1 con el primo IF-THEN-ELSE; C_n con el primo CASE-OF-n; D_2 con el primo WHILE-DO; D_3 con el primo REPEAT-UNTIL; D_4 con el primo EXIT-FROM-MIDDLE y L_2 con el primo 2-EXIT-LOOP, según Bache y Müllerburg [6].

Primero introducimos cómo computar $\alpha_{All-Branches}$. La Tabla 6.1 presenta los valores de $\alpha_{All-Branches}$ para el conjunto base de primos considerados por Bache y Müllerburg.

- Para el primo P_1 existe un único arco unconstrained, i.e, $\alpha_{All-Branches}(P_1) = 1$.
- Para el primo D_0 el arco que representa entrar al IF y el arco que representa no entrar al IF son unconstrained. Ya que no hay otro arco unconstrained en D_0, $\alpha_{All-Branches}(D_0) = 2$.
- Para el primo D_1 los arcos que representan entrar la parte THEN del IF y la parte ELSE del IF son ambos unconstrained. Ya que no hay otro arco unconstrained en D_1, $\alpha_{All-Branches}(D_1) = 2$.
- Para el primo C_n todos los arcos que representan entrar en una de las opciones del CASE son unconstrained. Ya que no hay otro arco unconstrained en C_n, $\alpha_{All-Branches}(C_n) = n$.

- Para el primo D_2 el único arco unconstrained corresponde a entrar al WHILE, i.e., $\alpha_{All-Branches}(D_2) = 1$.
- Para los primos D_3 y D_4 el único arco unconstrained corresponde a entrar al REPEAT al menos dos veces, i.e., $\alpha_{All-Branches}(D_3) = 1$.
- Para el primo L_2 los arcos que representan salidas son unconstrained en L_2. Además, el arco que representa el LOOP es también unconstrained. Entonces $\alpha_{All-Branches}(L_2) = 3$.

Primos	$\alpha_{All-Branches}$
P_1	1
D_0	2
D_1	2
C_n	n
D_2	1
D_3	1
D_4	1
L_2	3

Tablas 6.1: $\alpha_{All-Branches}$ para los primos

La Tabla 6.2 define el valor de $\alpha_{All-Branches}$ para las funciones de secuencialidad para los flowgraphs F_1, ..., F_n. Si construimos un flowgraph poniendo n flowgraphs en secuencia, los arcos unconstrained en el nuevo flowgraph son todos los arcos unconstrained en cada uno de los flowgraphs que lo componen. Entonces la función de secuencialidad es $\alpha_{All-Branches}(F_1; ...; F_n) = \alpha_{All-Branches}(F_1) + ... + \alpha_{All-Branches}(F_n)$.

Sequencing Function	$\alpha_{All-Branches}$
$F_1; ...; F_n$	$\alpha_{All-Branches}(F_1) + ... + \alpha_{All-Branches}(F_n)$

Tablas 6.2: $\alpha_{All-Branches}$ para la función de secuencialidad

La Tabla 6.3 define el valor de $\alpha_{All-Branches}$ para la función de anidamiento para cada primo considerado. La función de anidamiento depende del primo.

Nesting Function	$\alpha_{All-Branches}$
$D_0(F)$	$\alpha_{All-Branches}(F) + 1$
$D_1(F_1, F_2)$	$\alpha_{All-Branches}(F_1) + \alpha_{All-Branches}(F_2)$
$C_n(F_1, ..., F_n)$	$\alpha_{All-Branches}(F_1) + ... + \alpha_{All-Branches}(F_n)$
$D_2(F)$	$\alpha_{All-Branches}(F)$
$D_3(F)$	$\alpha_{All-Branches}(F) + 1$
$D_4(F_1, F_2)$	$\alpha_{All-Branches}(F_1) + \alpha_{All-Branches}(F_2)$
$L_2(F_1, F_2)$	$\alpha_{All-Branches}(F_1) + \alpha_{All-Branches}(F_2) + 1$

Tablas 6.3: $\alpha_{All-Branches}$ para la función de anidamiento

- Si anidamos un flowgraph F en un primo D_0, los arcos unconstrained en el nuevo flowgraph son aquéllos en el flowgraph F más el arco en D_0 que representa no entrar al IF; i.e., $\alpha_{All-Branches}(D_0(F)) = \alpha_{All-Branches}(F) + 1$.

- Si anidamos dos flowgraphs F_1 y F_2 en un primo D_1, los arcos unconstrained en el nuevo flowgraph son aquéllos en el flowgraphs F_1 más aquéllos en F_2; i.e., $\alpha_{All-Branches}(D_1(F))$ $= \alpha_{All-Branches}(F_1) + \alpha_{All-Branches}(F_2)$.

- De la misma manera, si anidamos los flowgraphs $F_1, ..., F_n$ en un primo C_n, $(\alpha_{All-Branches}C_n(F_1, ..., F_n))$ $= \alpha_{All-Branches}((F_1) + ... + \alpha_{All-Branches}(F_n)$.

- Si anidamos un flowgraph F en un primo D_2, los arcos unconstrained en el nuevo flowgraph son aquéllos en flowgraph F. De hecho, el único arco unconstrained en D_2 ha sido reemplazado por F, entonces $\alpha_{All-Branches}(D_2(F)) = \alpha_{All-Branches}(F)$.

- Si anidamos un flowgraph F en un primo D_3, los arcos unconstrained en el nuevo flowgraph son aquéllos en el flowgraph F más el arco de retorno en D_3, $\alpha_{All-Branches}(D_3(F)) =$ $\alpha_{All-Branches}(F) + 1$.

- Si anidamos dos flowgraphs F_1 y F_2 en un primo D_4, el número de arcos unconstrained en el nuevo flowgraph será la suma del número de arocs unconstrained en los dos flowgraphs anidados, $\alpha_{All-Branches}(D_4(F_1, F_2)) = \alpha_{All-Branches}(F_1) + \alpha_{All-Branches}(F_2)$.

- Si anidamos dos flowgraphs F_1 y F_2 en un primo L_2, el número de arcos unconstrained en el nuevo flowgraph será la suma del número de arcos unconstrained en los dos flowgraphs anidados más 1, $\alpha_{All-Branches}(L_2(F_1, F_2)) = \alpha_{All-Branches}(F_1) + \alpha_{All-Branches}(F_2) + 1$.

Ahora introducimos cómo computar $\beta_{All-Branches}$. Para cada primo, decimos que $\beta_{All-Branches}$ está dado como la suma de dos valores: $c + l$. Informalmente, en el largest weakly incomparable set of unconstrained arcos cuya cardinalidad está dada por $\beta_{All-Branches}$, c se corresponde con el número de arcos en el conjunto que pueden ser cubiertos por un camino sin loops, y l con el número de arcos en el conjunto que están dentro de un ciclo.

La Tabla 6.4 define c y l para el conjunto base de primos considerados por Bache y Müllerburg. Por ejemplo, para el primo IF-THEN-ELSE: $c = 2$ y $l = 0$, dando $\beta_{All-Branches} = 2 + 0 = 2$, y para el primo WHILE-DO: $c = 0$ y $l = 1$, dando $\beta_{All-Branches} = 0 + 1 = 1$.

Primes	c	l
P_1	1	0
D_0	2	0
D_1	2	0
C_n	n	0
D_2	0	1
D_3	0	1
D_4	0	1
L_2	2	1

Tablas 6.4: c, l para primos

- Para el primo P_1 el único arco compone el único LWI para P_1 y puede ser cubierto por un camino sin loops; i.e, $c(P_1) = 1$ y $l(P_1) = 0$.
- Para el primo D_0 el arco que representa entrar al IF y el arco que representa no entrar al IF componen el único LWI para D_0. Ambos arcos pueden ser cubiertos por caminos sin loops, i.e., $c(D_0) = 2$ y $l(D_0) = 0$.
- Para el primo D_1 los arcos que representan entrar a la parte THEN del IF y a la parte ELSE del IF componen el único LWI para D_0. Ambos arcos pueden ser cubiertos por caminos sin loops, i.e., $c(D_1) = 2$ y $l(D_1) = 0$.
- Para el primo C_n los arcos que representan entrar en cada una de las opciones del CASE componen el único LWI para C_n. Todos estos arcos pueden ser cubiertos por caminos sin loops, i.e., $c(C_n) = n$ y $l(C_n) = 0$.

- Para el primo D_2 el arco que representa entrar al WHILE compone el único LWI para D_2. Este arco está en un ciclo y entonces $c(D_2) = 0$ y $l(D_2) = 1$.

- Para los primos D_3 y D_4 el arco que representa entrar al REPEAT al menos dos veces forma el único LWI para ellos. Este arco está en un ciclo y entonces $c(D_3) = 0$ y $l(D_3) = 1$ y $c(D_4) = 0$ y $l(D_4) = 1$.

- Para el primo L_2 los arcos que representan salidas y el arco que representa el LOOP forman el único LWI para L_2. Los arcos que representan salidas pueden ser cubiertos por caminos sin loops, y el arco que representa el LOOP está en un ciclo. Por lo tanto, $c(L_2) = 2$ y $l(L_2) = 1$.

La Tabla 6.5 define la función de secuencialidad para los flowgraphs F_1, .., F_n. La computación se realiza en forma separada para los valores de c y l y luego se deriva $\beta_{All-Branches}$ sumando estos valores.

Sequencing Function	c	l
$F_1;...;F_n$	$\max(c(F_1),...,c(F_n))$	$l(F_1) + ... + l(F_n)$

Tablas 6.5: c y l para la función de secuencialidad

Si construimos un flowgraph F poniendo n flowgraphs $F_1...,F_n$ en secuencia, podemos construir un LWI L para F como sigue. Primero consideramos los arcos dentro de un ciclo. De la definición de LWI, todo arco dentro de un ciclo en $F_1...,F_n$ debe estar en todo LWI para F. Entonces, $l(F_1;...;F_n) = l(F_1) + ... + l(F_n)$. Ahora consideramos los arcos que puedens er cubiertos por caminos sin loops. Sea el flowgraph F_i tal que $c(F_i) = \max(c(F_1),...,c(F_n))$. Ya que $F_1,...,F_n$ están en secuencia, podemos considerar que los arcos que pueden ser cubiertos por caminos sin loops en F_i están en L, y L es un LWI para F. Entonces la función de secuencialidad es $c(F_1;...;F_n) = \max(c(F_1),...,c(F_n))$.

Por ejemplo, si un primo IF-THEN-ELSE y un primo WHILE-DO se combinan en secuencia, entonces $c = max(2,0) = 2$, $l = 0 + 1 = 1$, dando $\beta_{All-Branches} = 2 + 1 = 3$.

La Tabla 6.6 define la función de anidamiento para cada primo considerado, que permite obtener los valores de c y l. Por ejemplo, si anidamos dos flowgraphs F_1 y F_2 en un primo IF-THEN-ELSE, el valor de c es la suma de los valores c_1 y c_2 para los dos flowgraphs anidados, y el valor de l es la suma de los valores l_1 y l_2 para los dos flowgraphs anidados. Si anidamos un

flowgraph F en un primo WHILE-DO, el valor de c es cero (que es igual al valor de c para el primo WHILE-DO) y el valor de l es igual al valor $c_1 + l_1$ para el flowgraph anidado.

- Si anidamos un flowgraph F en un primo D_0, un LWI para D_0 se puede construir agregando el arco que representa no entrar al IF a un LWI para F. Entonces, el número de arcos en el nuevo LWI que puede ser cubierto por un camino sin loops es $c(D_0(F)) = c(F) + 1$; y aquéllos en un ciclo $l(D_0(F)) = l(F)$.

- Si anidamos dos flowgraphs F_1 y F_2 en un primo D_1, un LWI para D_0 puede construirse uniendo un LWI para F_1 con un LWI para F_2. Entonces, el número de arcos en el nuevo LWI que pueden ser cubiertos por un camino sin loops es $c(D_1(F)) = c(F_1) + c(F_2)$; y aquéllos en un ciclo $l(D_0(F)) = l(F_1) + l(F_2)$.

- De la misma manera, si anidamos los flowgraphs $F_1, ..., F_n$ en un primo C_n, $c(C_n(F_1, ..., F_n))$ $c(F_1) + ... + c(F_n)$ y $l(C_n(F_1, ..., F_n)) = l(F_1) + ... + l(F_n)$.

- Si anidamos un flowgraph F en un primo D_2 un LWI para F es también un LWI para D_2, y todos los arcos en LWI estarán dentro un ciclo en $D_2(F)$. Entonces, $c(D_2) = 0$ y $l(D_2)$ $= l(F) + c(F)$.

- Si anidamos un flowgraph F en un primo D_3, un LWI L para D_3 puede construirse agregando el arco que representa el LOOP a un LWI L_F para F. Entonces, los arcos en LWI que pueden ser cubiertos por un camino sin loops son aquéllos en L_F que pueden ser cubiertos por un camino sin loops en F. Los arcos en LWI dentro un ciclo en L son aquéllos en L_F dentro un ciclo en F más el arco del LOOP en D_3. Por lo tanto, $c(D_3) = c(F)$ y $l(D_3) = l(F) + 1$.

- Si anidamos dos flowgraphs F_1 y F_2 en un primo D_4, un LWI L para D_4 puede construirse como sigue. Sean L_{F_1} y L_{F_2} LWI para F_1 y F_2, respectivamente.

  - los arcos en L_{F_1} que pueden ser cubiertos por un camino sin loops en F_1 están en L. Estos arcos también pueden ser cubiertos por un camino sin loops en D_4.
  - los arcos en L_{F_2} que pueden ser cubiertos por un camino sin loops en F_2 están en L. Estos arcos están dentro un ciclo en D_4.
  - los arcos en L_{F_1} que están dentro un ciclo en F_1, están en L. Estos arcos están dentro un ciclo en D_4.
  - los arcos en L_{F_2} que están dentro un ciclo en F_2, están en L. Estos arcos están dentro un ciclo en D_4.

Therefore, $c(D_4) = c(F_1)$ y $l(D_4) = l(F_1) + l(F_2) + c(F_2)$.

- Si anidamos dos flowgraphs F_1 y F_2 en un primo L_2, tenemos el mismo caso anterior, y además tenemos que considerar la segunda salida del LOOP. Esta salida estará en cualquier LWI para $L_2(F_1, F_2)$, y puede ser cubierto por un camino sin loops. Por lo tanto, $c(L_2) = c(F_1) + 1$ y $l(L_2) = l(F_1) + l(F_2) + c(F_2)$.

Nesting Function	c	l
$D_0(F)$	$c(F) + 1$	$l(F)$
$D_1(F_1, F_2)$	$c(F_1) + c(F_2)$	$l(F_1) + l(F_2)$
$C_n(F_1, ..., F_n)$	$c(F_1) + ... + c(F_n)$	$l(F_1) + ... + l(F_n)$
$D_2(F)$	0	$l(F) + c(F)$
$D_3(F)$	$c(F)$	$l(F) + 1$
$D_4(F_1, F_2)$	$c(F_1)$	$l(F_1) + l(F_2) + c(F_2)$
$L_2(F_1, F_2)$	$c(F_1) + 1$	$l(F_1) + l(F_2) + c(F_2)$

Tablas 6.6: c y l para la función de anidamiento

Así, aplicando el método de Bache y Müllerburg, podemos computar recursivamente las cotas $\alpha_{All-Branches}$ y $\beta_{All-Branches}$ a partir del árbol de descomposición de cualquier flowgraph[3].

6.3 Validación de la Cota $\beta_{All-Branches}$

Creemos que es muy importante que la utilidad de las cotas propuestas sea evaluada mediante experimentación en el campo.

Algunas validaciones empíricas de la cota $\beta_{All-Branches}$ han sido realizadas [15]. En esta sección resumimos esta experimentación.

En el caso de estudio presentado, se ha realizado la validación del comportamiento de $\beta_{All-Branches}$ sobre varios programas que ya habían sido testeados usando el criterio All-Branches y para los cuales el número real de casos de test ejecutados estaba disponible. Además, ya que el número

[3]Notamos que las funciones de secuencialidad y anidamiento dadas en la Tabla 6.2 y Tabla 6.3, y Tabla 6.5 y Tabla 6.6, respectivamente, no se aplican al flowgraph trivial. En cambio, las Tablas 6.1 y 6.4 se debe considerar para él, i.e., el secuenciamiento o anidamiento del flowgraph trivial en cualquier flowgraph G no cambia el valor de $\alpha_{All-Branches}$, c y l del flowgraph G.

ciclomático del flowgraph del programa es a menudo usado como cota, el mismo análisis se ha realizado para este número.

El caso de estudio descripto involucró otro resultado de investigación, relacionado a la generación de caminos de test [12]. Dicha parte se presenta en el Capítulo 7.

6.3.1 Descripción del Experimento

Después de analizar los costos de la fase de testing de varios proyectos desarrollados usando tecnología avanzada, con procesos iterativos y desarrollo incremental, la División de Software de Ericsson Telecomunicazioni R&D decidió, hace algún tiempo, comenzar a invertir en investigación sobre testing de software. Considerando que un proceso de desarrollo consta de las etapas de análisis, diseño, implementación y test de unidad, se evaluó que el costo del test de unidad puede ser hasta 2/3 del costo total del desarrollo. En particular para el proyecto considerado, el cliente pidió un cubrimiento de al menos el 80% de All-Branches, como criterio de terminación de la fase de test de unidad. Por consiguiente, cualquier método que podría reducir los costos del test de unidad, tal como el método propuesto aquí, sería útil en términos de reducción del costo total. Además, para proyectos con estrategias de test complicadas y dependencias entre las diferentes fases, es una meta básica pasar *milestones* (hitos) a tiempo. La cota $\beta_{All-Branches}$ podría ser de ayuda en la estimación del esfuerzo esperado para la actividad de test de unidad.

Proyecto y Metodología El proyecto elegido como caso de estudio fue llevado a cabo por Ericsson Telecomunicazioni R&D en cooperación con el oficina principal en Estocolmo y otras subsidiarias de Ericsson. Fue uno de los proyectos más grandes desarrollados en Europa usando tecnología orientada a objetos, y la primera experiencia para Ericsson Telecomunicazioni. El proyecto comenzó en 1992, después de una fase de prototipación, y continuó con fases de desarrollo solapadas, agregando funcionalidades en cada paso incremental. Se usaron milestones para coordinar las fases. La aplicación considerada aquí se diseñó siguiendo una metodología orientada a objetos y se implementó en el lenguaje C++ usando la plataforma SUN OS. Para este proyecto, se usó un proceso de desarrollo de software específico, consistente en Análisis, Diseño, Implementación y Test, basado en la methodología ObjectOry [48]. Debido al ajustado plan de entregas del cliente, el software modelado en una fase temprana no se puede restructurar y las unidades de software fueron creciendo en tamao y complejidad. La actividad de testing que fue elegida para el caso de estudio fue uno de las iteraciones en el desarrollo incremental de este proyecto.

Estrategia de Testing Básica La estrategia de test para este proyecto incluyó cuatro tipos distintos de fases de testing, todas ellas dentro de la metodología ObjectOry:

Basic Test que es la fase de test de unidad, testeando los módulos más chicos del sistema ("test object").

Integration Test que consiste en testear un área funcional. Todos los módulos involucrados en esa área se integran juntos.

Function Test que consiste en verificar las funcionalidades del sistema.

System Test que verifica la performance y los requerimientos arquitectónicos del sistemas.

La fase monitoreada en el caso de estudio fue la de "Basic Test". Esta es una parte esencial del proceso de testing: un fase de test de unidad pobre nunca puede ser compensada por otras actividades de testing.

El objetivo del test de unidad era chequear el código con reglas de codificación, reglas de diseño y especificación. El criterio de terminación de esta etapa, como dijimos, era alcanzar al menos un cubrimiento de tipo All-Branches para el módulo (que puede estar compuesto de varios "métodos") de al menos el 80%. La herramienta TCAT [2] fue usada para medir el cubrimiento.

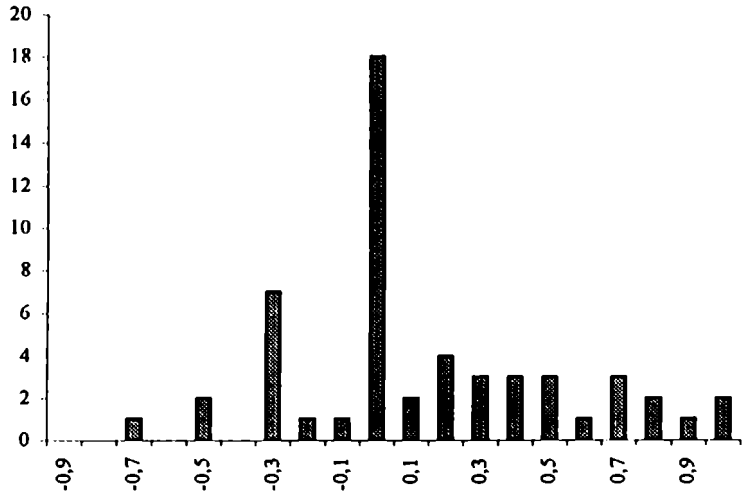
El procedimiento usado en el momento en que comenzó el proyecto era completamente manual. Se hacía un plan de test, basado en las especificaciones; para cada caso de test, se seleccionaba datos de test y se controlaba el resultado obtenido con respecto a lo esperado. Se guardaba el resutlado de los test en archivos "log". Se usaban "stubs" para simular el ambiente.

6.3.2 Evaluación de la Cota $\beta_{All-Branches}$

El objetivo del caso de estudio era evaluar la cota $\beta_{All-Branches}$ como una métrica para la predicción del número de casos de test necesarios para obtener 100% de cubrimiento All-Branches. La conclusión esperada era que la cota daba una buena estimación del esfuerzo de test para ser usada en el plan del proyecto.

Comparamos la cota teórica con el número real de casos de test usados para testear 55 unidades de programa. Se hizo la comparación considerando el error absoluto y el error relativo entre el número esperado y el número real de casos de test NT, y estudiando sus distribuciones empíricas. Específicamente, el error absoluto está dado por:

$$\Delta\beta_{All-Branches} = \beta_{All-Branches} - NT$$



Figuras 6.7: Observaciones del error relativo ϵ_β

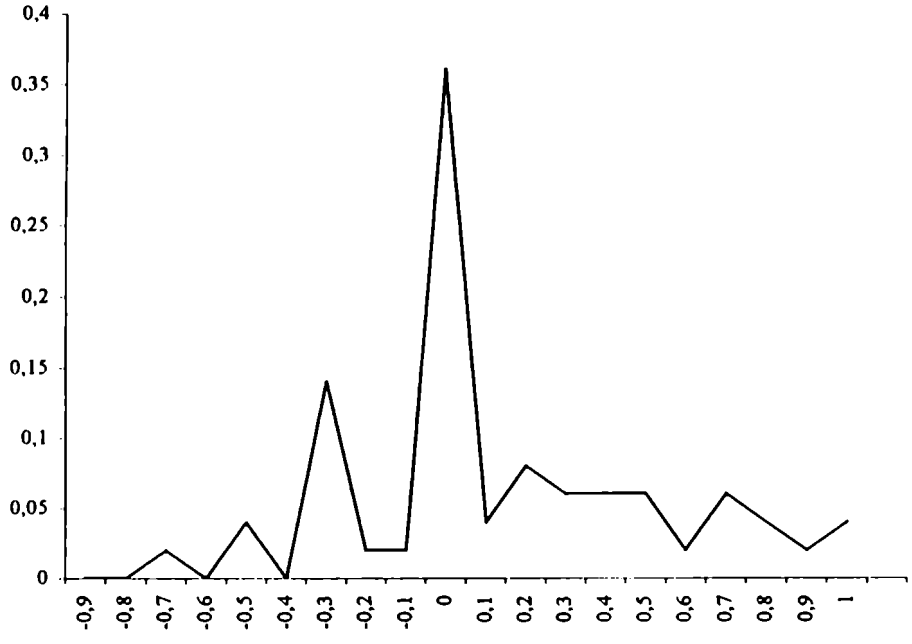
y el error relativo por:

$$\epsilon_\beta = \Delta\beta_{All-Branches}/NT$$

Para evaluar la efectividad de $\beta_{All-Branches}$, consideramos el error relativo. De las 55 unidades consideradas, sólo 1 observación dió un error relativo mayor que 1. Pensamos que este único caso anómalo podría desecharse para el análisis, sin distorsionar la evaluación de los resultados, ya que para todos los otros valores la conducta del error relativo era bastante estable. A partir de los datos juntados para las 54 unidades analizadas, se derivó el histograma de la Figura 6.7. Dividimos el intervalo [1, 1] en el que estaba el error relativo, en 20 sub-intervalos de 0.1, reportados en el eje de las X. En el eje Y reportamos el número de observaciones dentro de cada sub-intervalo. Como muestra la Figura 6.8, también se derivó la distribución empírica de los errores relativos; i.e., el eje Y en la Figura 6.8, da la proporción: (número de errores relativos observados dentro del sub-intervalo)/(número del total de observaciones), para cada sub-intervalo [54]. Luego de las primeras 35 observaciones, observamos que la estimación para la distribución de probabilidad era estable.

Hemos computado la media y la varianza para las $N = 54$ observaciones, que dieron:

$$\overline{\epsilon_\beta} = \sum_N \frac{\epsilon_\beta}{N} = 0.12$$



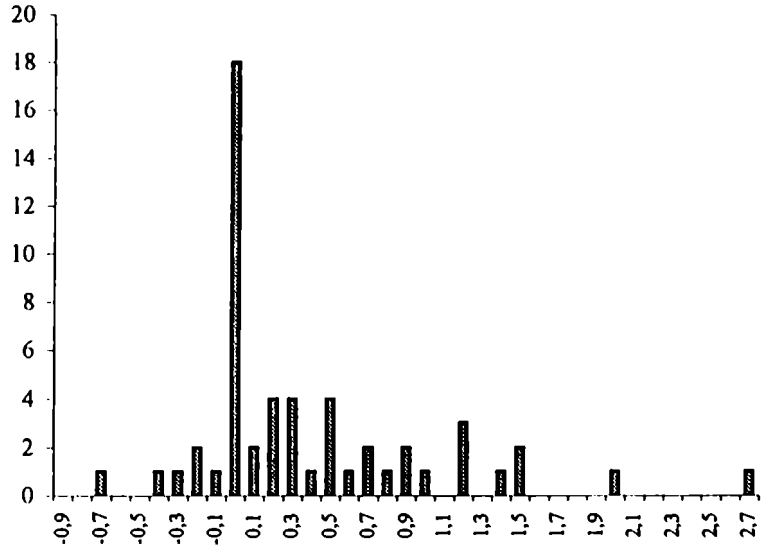
Figuras 6.8: Distribución Empírica de ϵ_β

$$s^2(\epsilon_\beta) = \sum_N \frac{(\epsilon_\beta - \overline{\epsilon_\beta})^2}{N-1} = 0.15$$

Los resultados confirman que la métrica proporciona una buena estimación del esfuerzo de test: 20 de las 54 observaciones (i.e., 37%) dieron un error relativo menor que el 10%. Observamos que la cota $\beta_{All-Branches}$ predice el número de casos de test necesarios para alcanzar el 100% de cubrimiento All-Branches, mientras que algunas de las unidades testeadas alcanzaron un cubrimiento de entre el 80% y hasta el 100%. Presumible, si todas las unidades hubieran alcanzado un cubrimmiento total, los valores observados de ϵ_β serían aún más cercanos a 0.

Para generalizar estos resultados, tratamos de identificar una distribución teórica subyacente para la distribución empírica. De las Figuras 6.7 y 6.8, observamos que la conducta de ϵ_β parece similar a la de una distribución teórica normal. Para validar esta hipótesis, i.e., para verificar que las diferencias entre los resultados observados y aquellos relativos a una distribución normal con media $\lambda = \overline{\epsilon_\beta}$ y con desviación standard $\sigma = \sqrt{s^2(\epsilon_\beta)}$ son insignificantes, aplicamos los test estadísticos clásicos χ^2 y Kolmogorov-Smirnov [51]. Ambos tests confirmaron que la hipótesis era aceptable con un nivel de confianza del 94%.

Comparamos estos resultados con las predicciones obtenidas al usar la complejidad ciclomática



Figuras 6.9: Observaciones del error relativo $\varepsilon_{v(G)}$

[58] del ddgraph del programa. Como hemos dicho, la complejidad ciclomática $v(G)$ a menudo se usa como una cota del número de casos de test necesarios para alcanzar cubrimiento All-Branches.

En la Figura 6.9, se muestra el histograma obtenido para $v(G)$. Hemos considerado las mismas 54 observaciones usadas para validar la cota $\beta_{All-Branches}$ (la misma observación anómala para el caso de $\beta_{All-Branches}$ resultó anómala para $v(G)$, dando un valor muy alto de error relativo, y fue descartada). También esta vez, 20 de las 54 observaciones (37%) dieron un error relativo menor del 10%. Sin embargo, de las restantes 34 observaciones, 8 cayeron fuera del intervalo [-1, 1], entre 1 y 2.67. La media y varianza dieron respectivamente 0.37 y 0.39. Por lo tanto, podemos concluir que $v(G)$ no es tan confiable como es $\beta_{All-Branches}$ para estimar el número de casos de test necesarios para lograr un cubrimiento All-Branches.

Finalmente, hemos analizado la proporción entre la cota $\beta_{All-Branches}$ y el número total de branches en las unidades estudiadas, de manera de tener una estimación somera de qué porcentaje del número de branches se corresponde con el número de tests necesarios. La media para esta observación fue 0.48 con una varianza de 0.016.

En conclusión, hemos descripto un caso real donde hemos evaluado la efectividad de la cota $\beta_{All-Branches}$ para predecir el número de casos de test necesarios para alcanzar 100% de cubrimiento All-Branches. Los resultados observados confirman que la cota $\beta_{All-Branches}$ proporciona una estimación buena del número de tests necesarios para alcanzar cubrimiento All-Branches.

Aplicando clásicos tests estadísticos, pudimos concluir también que la distribución empírica del error relativo entre el número esperado de casos de test y el número observado, se puede considerar una distribución normal con $\lambda = 0.12$ y $\sigma^2 = 0.15$ con el 94% de nivel de confianza. El comportamiento de la complejidad ciclomática, a menudo usado como estimación, se analizó también por comparación y los resultados fueron inferiores.

Por consiguiente, el caso de estudio descripto confirma la utilidad del resultado investigado y sugiere que su introducción en la fase de test de unidad hace al testing de tipo All-Branch más predecible.

6.4 Estimación del Costo del Testing: Caso General

Claramente, el número de tests necesarios para satisfacer un criterio de cubrimiento variará dependiendo del criterio usado, y de la misma manera variará el costo. Como fue señalado en [36], a medida que los requerimientos de cubrimiento se vuelven más exigentes, la efectividad del descubrimiento de defectos generalmente se incrementa, pero también lo hace el número de casos de test.

En esta sección generalizamos los resultados presentados para el caso de cubrimiento All-Branches a otros criterios de cubrimiento. Esta generalización es directa, pero la incluimos aquí por completitud.

6.4.1 Uso de Spanning Sets para Estimar el Costo del Testing

Como hemos establecido para el caso de testing All-Branches, los spanning sets of entities son útiles para estimar el costo del testing de cubrimiento en el caso general. Sea $G = (N, A)$ un ddgraph, c un criterio de cubrimiento, S un spanning set of entities para G y c. Ya que el cubrimiento de S garantiza la satisfacción de c para G, se puede tomar como una estimación del número de casos de test necesario para satisfacer c, a la cardinalidad de tal conjunto. Para ser precisos, éste es el número de casos de test requerido en la peor hipótesis que se elige un camino diferente para cubrir cada entidad en S. En este sentido el número de entidades unconstrained en un spanning set se puede ver como una cota "segura" en el número de casos de test necesarios para satisfacer la estrategia elegida.

Definición 6.5 α_c

Sea $G = (N, A)$ un ddgraph, c un criterio de cubrimiento, S un spanning set of entities para G y c. Entonces, $\alpha_c(G)$ se define como la cardinalidad de $|S|$, i.e.,

$$\alpha_c(G) = |S|.$$

Para el criterio All-Branches hemos introducido la cota $\beta_{All-Branches}$ como un refinamiento de la cota $\alpha_{All-Branches}$. Se ha hecho esto ya que se pueden combinar más de un arco unconstrained en un camino, y esto se debe considerar cuando se estima el número de casos necesario para satisfacer el criterio All-Branches. Sin embargo, en el caso de otros criterios de cubrimiento, pensamos que la cota α es una buena estimación del número de casos necesario para satisfacer el criterio, y la cota β no proporciona mejoras sobre ella. Creemos que, en estos casos, el valor de las cotas α y β será probablemente muy cercano uno al otro. Así, calcular β no tiene sentido. Sin embargo, un análisis más amplio y una mayor experimentación son necesarios para validar esta hipótesis. De todas maneras, incluimos esta generalización aquí por completitud.

6.4.2 Definición de la Cota β

En esta sección generalizamos las nociones de incompaarabilidad débil, largest weakly incomparable arc set, y cota $\beta_{All-Branches}$ introducidas más arriba.

En el caso general, buscamos un conjunto de caminos que cubren un genérico spanning set of entities; de hecho, la propiedad fundamental de los spanning set of entities garantiza que estos conjuntos de caminos cubren todas las entidades en el ddgraph. Más precisamente, buscamos un conjunto de caminos $\wp$ que agrupe las entidades que pertencen a un spanning set de una manera apropiada; i.e., requerimos que un camino en $\wp$ cubra más de una entidad en un spanning set, sólo si se cumplen las mismas condiciones precisas introducidas en la Sección 6.2.4.

Ahora definiremos cuándo dos entidades en G no pueden ser cubiertas por el mismo camino. En este caso las llamaremos dos *entidades débilmente incomparables.* Ahora definimos formalmente esta relación con referencia a un subconjunto genérico E de $E_c(G)$.

Definición 6.6 *Entidades Débilmente Incomparables*

Sea $G = (N, A)$ un ddgraph, y c un criterio de cubrimiento. Sean $E, E' \in S \subseteq E_c(G)$ y $E \neq E'$. Las entidades E y E' son entidades débilmente incomparables en S si para cualquier camino completo en G que contiene a ambas E y E' es siempre posible derivar un pseudo-camino que contenga sólo a una de ellas y que sea un camino completo en G.

Presentamos ahora la definición formal de un largest weakly incomparable entity set:

Definición 6.7 *Largest Weakly Incomparable Entity Set*

Sea $G = (N, A)$ un ddgraph, c un criterio de cubrimiento, $S \subseteq E_c(G)$. $LWI(S)$ es un largest weakly incomparable entity set para S en G si es un subconjunto de S que satisface:

- *para todo $E, E' \in LWI(S)$, si $E \neq E'$ entonces E y E' son débilmente incomparables,*
- *$|LWI(S)| = max\{|S'|: S' \subseteq S$ y para todo $E, E' \in S'$, si $E \neq E'$ entonces E y E' son débilmente incomparables$\}$.*

Notamos que dado un ddgraph y un subconjunto no vacío de entidades S, hay por lo menos un largest weakly incomparable set of entities para S, y posiblemente no es el único.

Ahora, juntando las nociones de spanning set of entities y de entidades débilmente incomparables, introducimos una definición formal de la cota β_c para un ddgraph G y un criterio c, como el máximo número de entidades en un spanning set of entities para G y c tal que sean mutuamente débilmente incomparables. En otras palabras, si S es spanning set of entities para G y c, esta cota considera un camino diferente para cada entidad en $|LWI(S)|$.

Definición 6.8 β_c

Sea $G = (N, A)$ un ddgraph, c un criterio de cubrimiento, S un spanning set of entities para G y c. Entonces β_c se define como la cardinalidad de un largest weakly incomparable entity set para S en G, i.e.,

$$\beta_c(G) = |LWI(S)|.$$

6.5 Conclusiones y Trabajo Futuro

En este capítulo hemos analizado el problema de establecer una cota inferior en el número de casos de test requerido para alcanzar un cubrimiento particular. Éste es un problema importante por varias razones. Por un lado, a menudo se usan medidas de cubrimiento estructural para evaluar la completitud del testing. Además, una cota inferior en el número de casos requerido para alcanzar un cubrimiento particular puede ser útil para predecir cuánto esfuerzo sería necesario para testear un programa dado.

Hemos mostrado que las cotas existentes no son adecuadas, ya que durante el desarrollo de las mismas se ha ignorado el uso de este número en la práctica. Hemos presentado dos cotas nuevas. Primero, la cardinalidad de un spanning set of entities para el ddgraph y el criterio de cubrimiento considerado se puede tomar como una cota inferior "segura" en el número de casos de test necesario para satisfacer el criterio elegido. Luego, combinando las nociones de spanning

set of entities y de entidades débilmente incomparables, hemos introducido una nueva cota "con significado" del número de casos de test a planear para un cubrimiento particular. La noción de incomparabilidad débil captura la idea intuitiva de combinar entidades unconstrained para crear caminos "con significado". Esta cota aborda el verdadero problema en testing estructural: encontrar un conjunto de caminos ejecutable. Hemos notado que esta cota inferior debería ser útil en el caso que se requiere cubrimiento de tipo All-Branches. En cambio, la cardinalidad de un spanning set of entities se puede considerar una cota útil en el caso general.

Hemos descrito algunos resultados preliminares experimentales en la evaluación de la cota $\beta_{All\text{-}Branches}$ como una métrica para la predicción del esfuerzo de testing necesario para alcanzar cubrimiento All-Branches. Hemos validado el comportamiento de $\beta_{All-Branches}$ en varias unidades de programa, testeadas según la estrategia de testing All-Branches, y para el que el número real de casos de test ejecutados ha sido recolectado. Además, ya que el número ciclomático $v(G)$ del flowgraph del programa G a menudo se usa como tal cota, hemos ejecutado el mismo análisis en este número para realizar una comparación entre ambos. Los resultados confirman que $\beta_{All-Branches}$ proporciona una estimación buena del esfuerzo de test. También concluimos que $v(G)$ no es tan confiable como $beta_{All-Branches}$ para estimar los casos de test necesarios para alcanzar cubrimiento All-Branches.

Esperamos que haya pioneros que incluyan nuestras cotas α y β en una herramienta de ayuda al testing, y que proporcionen evidencia empírica de la utilidad de las mismas para estimar el costo de testear un programa.

Capítulo 7

Generación de Caminos de Test

El proceso de testing puede describirse como:

1. seleccionar un conjunto de caminos de test tratando de satisfacer un criterio de cubrimiento e identificar datos de test para ejercer los caminos seleccionados;

2. ejecutar el programa usando los datos de test elegidos y supervisar qué entidades realmente se ejercieron;

3. evaluar la proporción entre el número de entidades ejercidas y el número total de entidades en el programa. Si esta proporción alcanza un valor predefinido, se detiene el testing; de lo contrario, se deben generar más casos de test: se repite el proceso desde el paso 1.

En este procedimiento, los pasos 2 y 3 pueden ser automatizados. De hecho, existen varios analizadores dinámicos de cubrimiento (dynamic coverage analyzers) (por ejemplo, [20, 22, 49, 67, 70, 74, 88], entre otros) que ejecutan el programa y pueden evaluar qué entidades se ejercieron a medida que los tests son ejecutados. El primer paso, que involucra la mayor parte del esfuerzo del testing, queda librado a la habilidad y creatividad del tester.

En este capítulo mostramos cómo usar la información proporcionada por un spanning set of entities para construir, para un ddgraph dado G y un criterio de cubrimiento dado c, un conjunto de caminos que cubra toda entidad en E_c, i.e., un conjunto de casos de test que satisface c. Presentamos un algoritmo general llamado CONSTRUCT-A-TEST-PATH-SET, que construye tal conjunto de caminos, tratando de cubrir las entidades en un spanning set of entities. La definición de un spanning set of entities presentada en el Capítulo 4, garantiza que el conjunto de caminos encontrados implica el cubrimiento de todas las entidades. La generación de datos de

test que ejecuten los casos de test elegidos está fuera del alcance de este trabajo.

En la próxima sección, analizamos los algoritmos existentes para construir un conjunto de caminos.

Luego, presentamos información necesaria para introducir el algoritmo. Ésto incluye la relación de dominación entre arcos de un ddgraph (ver Capítulo 4), y la relación simétrica, post-dominación.

Luego, describimos el algoritmo ALL-BRANCHES-TEST-PATH-SET, que encuentra un conjunto de caminos de test que satisface el criterio All-Branches para un programa dado, representado por un ddgraph. Este algoritmo ha sido publicado previamente en [12]. También presentamos experimentación preliminar.

Luego, introducimos el algoritmo general CONSTRUCT-A-TEST-PATH-SET, que construye un conjunto de caminos que cubre las entidades en un spanning set, para un criterio de cubrimiento elegido.

Finalmente, presentamos las conclusiones y el trabajo futuro.

7.1 Trabajo Previo

Para un ddgraph dado, se puede derivar muchos conjuntos diferentes de caminos que satisfacen un criterio dado. Así, se debe especificar a priori cuál es el criterio seguido en la construcción de los caminos. De hecho, en la literatura existen varios trabajos que abordan el problema de hallar un conjunto de caminos que satisface un criterio dado, cada uno desde un punto de vista específico. En esta sección, presentamos los algoritmos existentes para la generación de caminos.

La mayor parte del trabajo existente se ha hecho para generar conjuntos de caminos que satisfacen el criterio All-Branches.

Dado el alto costo del testing, los primeros trabajos trataron de encontrar *conjuntos mínimos de caminos*, i.e., conjuntos de caminos con el mínimo número posible de caminos que satisfacen un criterio dado, minimizando el número de tests requeridos. En [66] se discuten varios problemas usando un acercamiento de teoría de redes; se demuestra que el tamaño del conjunto mínimo de caminos para un digrafo G se puede determinar resolviendo un problema de flujo mínimo o un problema de matching máximo. En [61], se propuso un procedimiento heurístico para minimizar el número de caminos: los caminos entre una decisión y otra decisión en el que se descompone el programa se organizan en niveles crecientes. El cubrimiento All-Branches se aseguraría entonces cubriendo los caminos de los niveles más altos. Finalmente, en [77], el problema de seleccionar

caminos de test fue formulado como un problema de programación entera y se presentó un modelo generalizado optimal para la selección de caminos. Para cualquier criterio de selección, incluso el criterio All-Branches, un conjunto mínimo de caminos que lo satisface es seleccionado a partir del conjunto de todos los caminos con loops iterados a lo sumo una vez (notamos que, en cualquier caso, se debe derivar primero este conjunto de caminos usando alguna otra metodología).

Sin embargo, es un problema bien conocido [42] que los caminos derivados usando sólo información sintáctica pueden ser caminos *no factibles*, i.e., caminos del flujo de control que no pueden ser ejecutados usando ningún dato de input. Desgraciadamente, nunca se puede garantizar que un conjunto cualquiera de caminos generado mirando solamente la estructura de un programa, contiene sólo caminos factibles, ya que el problema de determinar si un input de entrada ejecuta un camino dado es indecidible [81]. No obstante, se puede idear una estrategia de selección de caminos que apunte a reducir los efectos de caminos no factibles.

En particular, como se sugirió en [93], uno podría seleccionar un conjunto de caminos cuyos caminos constitutivos involucran un número mínimo de predicados, ya que es más probable que éstos sean factibles. Esta última aserción no es sólo intuitivamente verdadera, sino que también ha sido validada de manera estadística [93].

Otro autores han propuesto que el problema de los caminos no factibles sea enfrentado por el usuario. Se le pide al usuario dar al algoritmo que genera el conjunto de caminos información sobre la factibilidad de los mismos.

Por ejemplo, en [52] se propone un método automático, en el que el próximo camino seleccionado es aquel que cubre la mayor cantidad de segmentos del programa no cubiertos hasta ahora. El camino resultante se muestra entonces al usuario, que puede aceptarlo o pedir la generación de un nuevo camino. En [92] se presenta una jerarquía de métricas de test estructural para dirigir la selección de caminos. La información necesaria para prevenir la generación de caminos no factibles la suministra manualmente el usuario.

A diferencia de los métodos propuestos hasta ahora, el método introducido aquí no está ligado a una política en particular. Primero presentamos un algoritmo que construye un conjunto de caminos que satisface el criterio All-Branches. Este algoritmo puede usar diferentes políticas para guiar la generación de caminos, de manera que pueden construirse diferentes conjuntos de caminos con propiedades diferentes. Además, presentamos un método generalizado para la generación automática de conjuntos de caminos que satisfacen un criterio de cubrimiento seleccionado de la familia de criterios del Capítulo 3.

7.2 Definiciones Básicas

En esta sección presentamos algunas nociones que serán útiles para introducir los algoritmos de generación de caminos de test. En particular, usamos las definiciones de dominación y post-dominación introducidas en la Sección 4.3.1.

Sea $G = (N, A)$ un ddgraph con arco de entrada e_0 y arco de salida e_k.

Un *camino de dominación* p_{DT} en $DT(G)$ es una secuencia de arcos $p_{DT} = e_1, ..., e_q$, tal que $e_i = PARENT(e_{i+1})$ en $DT(G)$ para $i = 1, ..., q-1$. En el caso general, un camino de dominación en $DT(G)$ no es un camino en G. Por ejemplo, las secuencias de arcos e_1, e_2, e_7 y e_2, e_3, e_6 son caminos de dominación en $DT(G_{SORT})$ presentado en la Figura 4.4. En particular, e_2, e_3, e_6 no es un camino en G_{SORT}.

Un *camino de post-dominación* p_{IT} en $IT(G)$ es una secuencia de arcos $p_{IT} = e_1, ..., e_q$, tal que $e_{j+1} = PARENT(e_j)$ en $IT(G)$ para $j = 1, ..., q-1$. Notamos que un camino de post-dominación se deriva a partir de una hoja y hacia la raíz de $IT(G)$. En general, un camino de post-dominación en $IT(G)$ no es un camino en G. La secuencia p_{IT}=e_4, e_6, e_7, e_8 es un camino de post-dominación en $IT(G_{SORT})$ presentado en la Figura 4.4.

Los algoritmos descriptos en este capítulo usan caminos de dominación y post-dominación para construir caminos en el ddgraph. Aprovechan el hecho de que un camino en $DT(G)$ (o en $IT(G)$) siempre corresponde a un conjunto de caminos en el ddgraph G, y, precisamente, el primero es un *pseudo-camino* de cada camino en el conjunto (según la definición de pseudo-camino dada en el Capítulo 2).

Recordamos que un pseudo-camino no es necesariamente un camino; i.e., dos arcos consecutivos en un pseudo-camino pueden no ser adyacentes en el ddgraph. Como dijimos más arriba, los algoritmos construirán un camino en el ddgraph usando caminos de dominación y de post-dominación, que son pseudo-caminos en el ddgraph. Decimos que existe una *discontinuidad* entre dos arcos consecutivos en un camino de dominación o en un camino de post-dominación si no son adyacentes en el ddgraph, como se define más abajo.

Definición 7.1 *Discontinuidad*

Sea G un ddgraph, $DT(G)$ su árbol de dominación y $IT(G)$ su árbol de post-dominación. Dados dos arcos e y e' que representan nodos adyacentes en $DT(G)$ [en $IT(G)$], con $e = PARENT(e')$ [$e' = PARENT(e)$], decimos que existe una discontinuidad entre ellos si no son adyacentes en G, i.e., si $HEAD(e) \neq TAIL(e')$.

Por ejemplo, existe una discontinuidad entre e_3 y e_6, ya que son nodos adyacentes en $DT(G_{SORT})$

(ver la Figura 4.4 A), pero no son adyacentes en G_{SORT} (ver la Figura 2.1).

Ahora podemos ser más precisos al describir cómo el algoritmo ALL-BRANCHES-TEST-PATH-SET introducido en la próxima sección construye caminos del ddgraph: primero deriva un camino de dominación o uno de post-dominación, y luego "llena" las discontinuidades, si existen, con un camino adecuado en G. Por ejemplo, en el camino $p_{DT} = e_1, e_2, e_3, e_6$ en $DT(G_{SORT})$, la discontinuidad entre e_3 y e_6 puede llenarse con el camino $p' = e_4$, obteniendo el camino $p = e_1, e_2, e_3, e_4, e_6$ del ddgraph.

Observamos que, en general, hay más de un camino que se puede usar para llenar una discontinuidad. Por ejemplo, se puede llenar el discontinuidad en el ejemplo precedente también con el camino $p'' = e_5$. La relación entre caminos de dominación o post-dominación y caminos en el ddgraph se establece formalmente en la proposición que sigue.

Proposición 7.1 *Sea G un ddgraph con arco de entrada e_0 y arco de salida e_k, $DT(G)$ su árbol de dominación y $IT(G)$ su árbol de post-dominación.*

1. *Si existe un camino p_{DT} en $DT(G)$ desde e_0 hasta e, $p_{DT} = e_0, e_1, ..., e_q, e$, entonces existe un camino p desde e_0 hasta e en el ddgraph G y p tiene la siguiente forma:*

   $p = e_0, p_0, e_1, p_1, ..., e_q, p_q, e_{q+1}(= e)$,

   donde para $j = 0, ..., q$, p_j es un camino (que puede ser vacío) desde $HEAD(e_j)$ hasta $TAIL(e_{j+1})$,

2. *Si existe un camino p_{IT} en $IT(G)$ desde e hasta e_k, $p_{DT} = e, e_2, ..., e_q, e_k$, entonces existe un camino p desde e hasta e_k en el ddgraph G y p tiene la siguiente forma:*

   $p = (e_1 =)e, p_1, e_2, p_2, ..., e_q, p_q, e_{q+1}(= e_k)$,

   donde para $j = 1, ..., q$, p_j es un camino (que puede ser vacío) desde $HEAD(e_j)$ hasta $TAIL(e_{j+1})$,

La demostración es directa a partir de las definiciones de dominación y post-dominación.

7.3 Generación de Caminos para Cubrimiento All-Branches

7.3.1 Generalidades

En esta sección describimos el algoritmo ALL-BRANCHES-TEST-PATH-SET, que halla un conjunto de caminos de test que satisface el criterio All-Branches para un programa dado representado

por un ddgraph. Los resultados presentados en esta sección han sido publicados en [12][1].

El algoritmo ALL-BRANCHES-TEST-PATH-SET construye un conjunto de caminos que cubre todos los arcos en un spanning set de arcos para un ddgraph y el criterio de cubrimiento All-Branches. Por la definición de spanning sets presentada en el Capítulo 4, el conjunto de caminos encontrado cubrirá todos los arcos en el ddgraph.

El algoritmo ALL-BRANCHES-TEST-PATH-SET construye un conjunto de caminos completos $\wp = \{p_1, ..., p_n\}$ para un ddgraph dado $G = (N, A)$ con arco de entrada e_0 y arco de salida e_k. Primero se deriva un spanning set de arcos S para G y el criterio All-Branches. El algoritmo ALL-BRANCHES-TEST-PATH-SET entonces construye $\wp$. Los caminos en $\wp$ se derivan de a uno. Para construir un camino, ALL-BRANCHES-TEST-PATH-SET selecciona un arco e_u en S todavía no cubierto. Luego, llama a la función recursiva FIND-A-PATH; esta función encuentra p_u desde el arco de entrada e_0 hasta el arco de salida e_k, pasando por el arco e_u. Este procedimiento se repite hasta que se cubren todos los arcos en S.

Para ser más explícitos, FIND-A-PATH construye el camino desde e_0 hasta e_k, usando el arco e_u, concatenando el único camino de dominación p_{DT} en el árbol de dominación $DT(G)$ que va desde la raíz e_0 hasta la hoja e_u, con el único camino de post-dominación p_{IT} en el árbol de post-dominación $IT(G)$ que va desde la hoja e_u hasta la raíz e_k. Sean $p_{DT} = e_0, e_1, ..., e_{r-1}, e_u(= e_r)$, y $p_{IT} = (e_r =)e_u, e_{r+1}, ..., e_{k-1}, e_k$. Por la Proposición 7.1, la sucesión de arcos obtenida concatenando p_{DT} con p_{IT} es un pseudo-camino en G, i.e., existe un camino en G desde e_0 hasta e_k de la siguiente forma:

$$p_u = e_0, p_{0\,1}, e_1, p_{1\,2}, ...e_{r-1}, p_{r-1\,u}, e_u, p_{u\,r+1}, e_{r+1}, p_{r+1\,r+2}, e_{r+2}, ..., e_{k-1}, p_{k-1\,k}, e_k$$

donde cada p_{ij} es un camino (no vacío) desde e_i hasta e_j en G, si hay una discontinuidad entre e_i y e_j en el árbol correspondiente, y es un camino vacío si e_i y e_j son arcos adyacentes en G. Así, cuando se descubre una discontinuidad entre dos arcos e_i y e_j en un camino de dominación o de post-dominación, el algoritmo debe construir un camino p_{ij} en G. Para hacer esto, se hace una llamada recursiva a FIND-A-PATH y se construye el sub-ddgraph G^* de G con arco de entrada e_i y arco de salida e_j. De nuevo, se selecciona un arco e^*_u del spanning set correspondiente (notar que esto siempre será posible, porque había una discontinuidad entre e_i y e_j en G), y se encuentra un camino p^*_{ij} en G^* desde el arco de entrada e_i hasta el arco e_j, usando el arco e^*_u. Una vez más, se construye este camino p^*_{ij} concatenando el camino de dominación p_{DT} en $DT(G^*)$ desde

[1] En [12] este algoritmo tiene el nombre FIND-A-TEST-PATH-SET o FTPS.

e_i hasta e^*_u, con el camino de post-dominación p_{IT} en $IT(G^*)$ desde e^*_u hasta e_j. Notar que el camino construido en G^* siempre corresponde a un camino en G.

Los caminos encontrados usando el procedimiento descripto más arriba pueden contener ciclos (cada vez que se selecciona un arco dentro de un ciclo) pero, por construcción, cada ciclo en un camino será iterado al máximo una vez.

Notamos que, de la estructura del algoritmo, se puede ver fácilmente que el número n de caminos generados no es más grande que $|S|$.

En la descripción precedente del algoritmo, la política de selección del próximo arco a cubrir en cada llamada recursiva de FIND-A-PATH ha sido dejada implícita intencionalmente. De hecho, el próximo arco a ejercer se puede escoger entre los arcos en un spanning set de acuerdo a diferentes criterios, y el criterio seguido afectará las propiedades del conjunto de caminos que se construirá automáticamente. El algoritmo deja implícita la política de selección, llamando a un procedimiento SELECT-AN-ARC genérico.

En la próxima sección, se analizan algunas aplicaciones posibles y útiles de este procedimiento.

Aquí abajo, introducimos notación que se empleará en la presentación del algoritmo y presentamos funciones que se usarán en él. El algoritmo se presenta en las Figuras 7.1 y 7.2.

- El procedimiento FIND-A-SPANNING-SET-OF-ENTITIES has sido presentado en la Figura 4.3.
- Se usa el conjunto de arcos auxiliar NOT_USED, que contiene los arcos aún no cubiertos en el spanning set S. Por lo tanto, cuando este conjunto está vacío, el algoritmo ALL-BRANCHES-TEST-PATH-SET termina.
- $DT(G)$ y $IT(G)$ son funciones que, dado un ddgraph G, construyen los árboles de dominación y post-dominación de G, respectivamente.
- El procedimiento SUB-DDGRAPH(G, e_a, e_b) (descripto en el Capítulo 2 y presentado en la Figura 2.4) deriva el sub-ddgraph de G con arcos distinguidos e_a y e_b.
- Se usan las siguientes operaciones sobre caminos:
  - $\langle\ e\ \rangle$ devuelve el camino formado por el arco e;
  - $p_1 + p_2$ devuelve un camino que consiste en el camino p_1 seguido por el camino p_2, si la HEAD del último arco en p_1 es la TAIL del primer arco en p_2;
  - $p_1 - p_2$ devuelve un camino obtenido eliminando en p_1 todos los arcos de p_2, si los arcos en p_2 son los últimos arcos de p_1.

Algorithm ALL-BRANCHES-TEST-PATH-SET($G = (N, A)$: ddgraph; e_0, e_k: arcs): set of paths;

begin

S = FIND-A-SPANNING-SET-OF-ENTITIES(G, A)

$NOT\_USED = S$

$\wp$=$\emptyset$; {$\wp$, *initially empty, will return the test paths found*}

while $|NOT\_USED| > 0$ **do**

{*finding a new path p*}

e_u=SELECT-AN-ARC(NOT_USED)

NOT_USED=NOT_USED - $\{e_u\}$

p = FIND-A-PATH($G, e_u, e_0, e_k, S, NOT\_USED$)

$\wp = \wp \cup \{p\}$

return($\wp$)

end algorithm

Figuras 7.1: ALL-BRANCHES-TEST-PATH-SET Algorithm

- Se usan las siguientes funciones sobre árboles:
  - PARENT(e,T), que devuelve el padre del arco e en el árbol T;
  - DISC?(e,e',T), que devuelve un valor Booleano, igual a true si existe una discontinuidad entre los arcos e y e', e igual a false, en otro caso.

7.3.2 Selección del Próximo Arco

Como dijimos más arriba, el algoritmo ALL-BRANCHES-TEST-PATH-SET deja la política de selección del próximo arco como un parámetro abierto, haciendo una llamada a una función genérica SELECT-AN-ARC. En esta sección mostramos cómo el algoritmo trabaja para distintas políticas de selección. En particular, discutimos una política que apunta a reducir el número de caminos y otra que apunta a prevenir la generación de caminos no factibles.

Debido a su costo alto, un tema importante en el área de testing es minimizar el número de casos de test para satisfacer un criterio de test dado. En testing de tipo All-Branches, esto corresponde a hallar un conjunto mínimo de caminos que cubren todos los arcos del ddgraph. En nuestro algoritmo, para reducir el número de caminos, como cada (pseudo-)camino se construye usando un arco unconstrained, cada camino debería cubrir tantos arcos como sea posibles en el

```
Function FIND-A-PATH (G: ddgraph; e_u, e_a, e_b: arc; S, NOT_USED: set of arcs;
out NOT_USED: set of arcs): path;
begin
{finding a path using e_u}
    p = ( e_u )
    {finding a dominator path from e_a to e_u}
    DT = DT(G)
    e_i = e_u
    while (e_i ≠ e_a)
        do begin
            e_p=PARENT(e_i, DT)
            if DISC?(e_p, e_i, DT) then begin
                G' = SUB-DDGRAPH(G, e_p, e_i)
                S' = FIND-A-SPANNING-SET-OF-ENTITIES(G', ARCS(G'))
                {selecting an unconstrained arc e'_u}
                e'_u = SELECT-AN-ARC(S')
                NOT_USED=NOT_USED - {e'_u}
                p' = FIND-A-PATH(G', e'_u, e_p, e_i, S', NOT_USED)
                p = p' - ( e_i ) + p
            end
            else p = ( e_p ) + p
            e_i=e_p
            end
    {finding an implied path from e_u to e_b}
    IT = IT(G)
    e_i=e_u
    while (e_i ≠ e_b)
        do begin
            e_p = PARENT(e_i, IT)
            if DISC?(e_p, e_i, IT) then begin
                G' = SUB-DDGRAPH(G, e_i, e_p)
                S' = FIND-A-SPANNING-SET-OF-ENTITIES(G', ARCS(G'))
                {selecting an unconstrained arc e'_u}
                e'_u = SELECT-AN-ARC(S')
                NOT_USED=NOT_USED - {e'_u}
                p' = FIND-A-PATH(G', e'_u, e_i, e_p, S', NOT_USED)
                p = p - ( e_i ) + p'
            end
            else p = p + ( e_p )
            e_i = e_p
            end
    return(p)
end function
```

Figuras 7.2: FIND-A-PATH Function

spanning set S que no estén ya cubiertos por otros caminos. Por consiguiente, el criterio *mínimo-número-de-caminos* requiere que un "arco aún no cubierto" en S sea elegido cuando sea posible. Esta selección se hace antes de derivar un camino nuevo, y también cada vez que se hace una iteración recursiva para llenar una discontinuidad. Esto corresponde a implementar una función SELECT-AN-ARC que elige un arco en S, en particular, uno del conjunto NOT_USED, cuando es posible.

Sin embargo, al buscar un bajo número de caminos, es probable que se construyan caminos no factibles. Es imposible que el algoritmo ALL-BRANCHES-TEST-PATH-SET cumpla completamente el objetivo específico de derivar caminos ejecutables, ya que está basado sólo en el análisis de flujo de control del programa. Sin embargo, el número de caminos no factibles puede ser reducido usando criterios diferentes al criterio *mínimo-número-de-caminos*. En particular, trabajos experimentales recientes [93] han proporcionado evidencia estadística que apoya la idea muy intuitiva de que caminos que involucran un número reducido de predicados es más probable que sean factibles. De acuerdo con esto, la heurística que siga el algoritmo para la selección del próximo arco debe construir un conjunto de caminos que cubra todos los arcos en un ddgraph, cuyos caminos constitutivos involucren un número mínimo de predicados (*nodos de decisión*).

Por consiguiente, necesitamos desarrollar un método para escoger un arco tal que el camino construido por el algoritmo usando este arco tenga un número mínimo de predicados. Desgraciadamente, no es posible dar una caracterización general de tal arco usando sólo la información en los árboles de dominación y post-dominación de G. En cambio, en general, necesitamos la información de los árboles de dominación y post-dominación del SUB-DDGRAPH (G,e,e'), cada vez que un arco (e,e') tiene una discontinuidad en $DT(G)$ o $IT(G)$.

Sin embargo, usando sólo la información en $DT(G)$ y $IT(G)$, es posible dar un método heurístico que se puede usar para elegir un arco tal que el camino correspondiente construido por el algoritmo tenga un número de predicados cercano al mínimo. Remarcamos que el criterio propuesto en [93] es de por si una heurística que apunta a reducir los efectos en caminos no factibles al usar el criterio All-Branches.

Observamos que:

- Si e es un nodo en $DT(G)$ tal que tiene por lo menos dos hijos en $DT(G)$, entonces $HEAD(e)$ es un nodo de decisión en G.
- Si e es un nodo en $IT(G)$ tal que el arco (PARENT(e),e) tiene una discontinuidad en $IT(G)$, entonces $HEAD(e)$ es un nodo de decisión en G (ya que hay más de una ruta para alcanzar a PARENT(e) desde e).

Por consiguiente, proponemos el método siguiente para elegir el próximo arco e_u: contamos el número de nodos de decisión en el camino de dominación que va desde el arco de entrada hasta e_u, más el número de discontinuidades en el camino de post-dominación desde e_u hasta el arco de salida. El número resultante es una cota inferior del número de nodos de decisión en todos los caminos completos que contienen al arco e_u. En el caso en el cual tenemos dos arcos con el mismo número de nodos de decisión, preferimos un arco no usado. Llamamos a esta política *menos-pred* y al camino construido usándola un camino *menos-pred.*

A pesar de lo que hemos dicho sobre la implementación de la función SELECT-AN-ARC, cualquier política posible usada en la generación de caminos puede elegir caminos no factibles [81]. Ahora presentamos una técnica que se puede usar para enfrentar este problema, si permitimos interacción entre el algoritmo y el tester, perdiendo un poco de automatización. La idea en la que se basa la técnica es la siguiente: al reconocer un camino como no factible, el tester puede darle información al algoritmo de que una combinación particular de arcos es imposible o indeseable. El algoritmo entonces reasume la fase de generación de caminos, tomando en cuenta esta información. Se han usado técnicas similares previamente, e.g. en [92]. En ese trabajo la información que es difícil o imposible de obtener automáticamente es suministrada manualmente por el tester para prevenir la generación estática de caminos no factibles. Esta información es llamada *allegations.*

Ya que nuestro algoritmo construye los caminos uno a la vez, es posible tratar este problema durante la construcción de caminos. Supongamos que el algoritmo encuentra que p_u es un camino no factible, que cubre al arco e_u. Entonces:

1. Podemos almacenar esta información. Por ejemplo, se puede extender el algoritmo de manera de construir y mantener una tabla de composiciones no deseadas de branches, que crece progresivamente. Se puede usar esta información en la fase de generación de caminos para prevenir la construcción futura de caminos que contienen estas combinaciones de arcos no deseadas.

2. Se generará un camino nuevo para cubrir al arco e_u. Sea S el pseudo-camino en G, $s_p = e_0, ..., e_u, ..., e_k$, donde $e_0..., e_u$ es el camino de dominación en $DT(G)$ entre e_0 y e_u, y $e_u..., e_k$ es el camino de post-dominación en $IT(G)$ entre e_u y e_k. Como mostramos en la Proposición 7.1, podemos obtener el conjunto de todos los caminos desde e_0 hasta e_k que contienen al arco e_u, llenando las discontinuidades en S: en general, se pueden usar distintos caminos para hacer esto. Entonces, cuando el tester reconoce el camino p_u generado por el algortimo como no factible, se requerirá al algoritmo que construya un nuevo camino que cubra e_u.

El algoritmo llenará las discontinuidades en s_p con caminos nuevos, teniendo en cuenta la información sobre la no factibilidad de p_u, almacenada como se describió en la parte 1.

El algoritmo puede acceder a la tabla para verificar la no factibilidad de un camino mientras se realiza la construcción. Por ejemplo, la función SELECT-AN-ARC podría seleccionar el próximo arco de un subconjunto más pequeño, obtenido eliminando del spanning set S todos los arcos guardados en la tabla como imposibles de combinar con los arcos ya elegidos para el camino en construcción. Sin embargo, en esta selección, el algoritmo todavía puede seguir cualquier política, tal como *mínimo-número-de caminos* o *menos-pred*.

Observamos que se puede usar un procedimiento similar para "pedirle" al algoritmo que construya un camino que cubra un par de branches específicamente requerido.

En la siguiente sección presentaremos ejemplos de la aplicación de estas políticas a un ddgraph.

7.3.3 Uso del Algoritmo ALL-BRANCHES-TEST-PATH-SET

Ahora discutimos cómo el algoritmo ALL-BRANCHES-TEST-PATH-SET se puede usar en All-Branches testing. En particular presentamos unos ejemplos para mostrar cómo ALL-BRANCHES-TEST-PATH-SET puede ayudar a manejar el problema de no factibilidad variando la implementación de SELECT-AN-ARC. Un ddgraph dado G puede corresponder a muchos programas distintos (en realidad, un número infinito de programas), i.e., todo aquellos programas cuyos ddgraph asociado es G. Los resultados del experimento en [93] aseguran que un conjunto de caminos que cubre todos los arcos en un spanning set, derivado según la política *less-pred*, tiene una oportunidad muy buena de ser factible. Entonces, como una política general, cuando los testers no tienen a mano información sobre la semántica del programa, entonces deben primero tratar con el conjunto de caminos derivado por ALL-BRANCHES-TEST-PATH-SET según la política *less-pred*.

Sin embargo, como sucede con cualquier solución obtenida trabajando en forma estadística, para un programa específico el conjunto de caminos generado puede contener varios caminos no factibles, y un conjunto de caminos derivado usando una estrategia diferente podría ser más útil. Por ejemplo, consideramos el ddgraph G_{GETOP} de la Figura 6.5. En él hemos numerado los nodos de decisión, de acuerdo con los indicadores a los predicados del programa de la Figura 6.4. Un spanning set de entidades es:

$$S = \{e_1, e_3, e_6, e_7, e_{11}, e_{12}, e_{13}, e_{16}, e_{17}, e_{20}, e_{22}\}$$

Usando la política *less-pred*, para cada arco en S el procedimiento SELECT-AN-ARC cuenta

el número N_d de nodos decisión en el (pseudo-)camino obtenido concatenando el camino de dominación y el camino de post-dominación. Entonces selecciona el arco con el N_d más bajo, posiblemente prefiere uno no usado; cuando hay varias opciones, elige el que tiene el " índice asociado más alto". Así, con Thus, with

arc	e_1	e_3	e_6	e_7	e_{11}	e_{12}	e_{13}	e_{16}	e_{17}	e_{20}	e_{22}
N_d	3	2	6	6	5	7	7	9	9	4	6

el algoritmo elige los siguientes 10 caminos:

$p_1 = e_0e_2e_3e_{24}$
$p_2 = e_0e_1e_2e_3e_{24}$
$p_3 = e_0e_2e_4e_9e_{11}e_{20}e_{24}$
$p_4 = e_0e_2e_4e_9e_{11}e_{21}e_{22}e_{23}e_{24}$
$p_5 = e_0e_2e_4e_5e_7e_8e_9e_{11}e_{20}e_{24}$
$p_6 = e_0e_2e_4e_5e_6e_8e_9e_{11}e_{20}e_{24}$
$p_7 = e_0e_2e_4e_9e_{10}e_{13}e_{14}e_{19}e_{20}e_{24}$
$p_8 = e_0e_2e_4e_9e_{10}e_{12}e_{14}e_{19}e_{20}e_{24}$
$p_9 = e_0e_2e_4e_9e_{10}e_{13}e_{14}e_{15}e_{17}e_{18}e_{19}e_{20}e_{24}$
$p_{10} = e_0e_2e_4e_9e_{10}e_{13}e_{14}e_{16}e_{16}e_{18}e_{19}e_{20}e_{24}$

Sin embargo, para este programa podemos ver fácilmente que los predicados en los nodos 4, 6, 8 y 9 corresponden a la misma condición, i.e., $i < lim$, donde lim es un parámetro fijo y i es una varibale no decreciente. Por consiguiente, una vez que esta condición se vuelve FALSA, i.e., una vez que $i \geq lim$, en cada ocurrencia siguiente de la misma condición, será siempre FALSA. Más precisamente, tenemos lo siguiente:

- e_7 es incomparable con e_{12}, e_{16}, e_{20};
- e_{13} es incomparable con e_{16}, e_{20};
- e_{17} es incomparable con e_{20}.

Por consiguiente, los caminos precedentes p_5, p_7, p_9, p_{10} no son factibles. En cambio, usando la política bajo *mínimo-número-de-caminos*, por ejemplo, SELECT-AN-ARC recogería el "índice asociado más alto", en preferencia para un arco en S no cubierto, y entonces selecciona los siguientes 7 caminos:

Path	input string	*lim*
P_1'	3.4A\n	1
P_2'	3.4A	10
P_3'	3A\n	1
P_4'	34.5A\n	1
P_5'	34.5A\n	2
P_6'	+	10
P_7'	' '3.4A\n	1

Tablas 7.1: A set of test inputs for branch testing function GETOP

$p_1' = e_0e_2e_4e_9e_{10}e_{13}e_{14}e_{15}e_{17}e_{18}e_{19}e_{21}e_{22}e_{23}e_{24}$
$p_2' = e_0e_2e_4e_9e_{10}e_{12}e_{14}e_{15}e_{16}e_{18}e_{19}e_{20}e_{24}$
$p_3' = e_0e_2e_4e_9e_{11}e_{21}e_{22}e_{23}e_{24}$
$p_4' = e_0e_2e_4e_5e_7e_8e_9e_{10}e_{13}e_{14}e_{15}e_{17}e_{18}e_{19}e_{21}e_{22}e_{23}e_{24}$
$p_5' = e_0e_2e_4e_5e_6e_8e_9e_{10}e_{13}e_{14}e_{15}e_{17}e_{18}e_{19}e_{21}e_{22}e_{23}e_{24}$
$p_6' = e_0e_2e_3e_{24}$
$p_7' = e_0e_1e_2e_4e_9e_{10}e_{13}e_{14}e_{15}e_{17}e_{18}e_{19}e_{21}e_{22}e_{23}e_{24}$

Todos estos caminos son factibles. Por ejemplo, los inputs en la Tabla 7.1 los ejecutan.

Entonces, en este caso la política *mínimo-número-de-caminos* sería mucho más útil que la política *less-pred*. Por supuesto, no intentamos generalizar esta conclusión. Sin embargo, este ejemplo simple muestra que puede ser útil en práctica poder derivar caminos de acuerdo a políticas diferentes.

Otro ejemplo en el que la política *less-pred* generaría muchos caminos no factibles es el caso común de un programa con varios loops en secuencia que se tienen que iterar el mismo número de veces. Muchas funciones de manipulación de matrices presentan este tipo de estructura, e.g.,

for i from 1 to n do begin ... end;
for j from 1 to n do begin ... end;
for k from 1 to n do begin ... end;

En este caso la política *less-pred* es más probable que genere muchos caminos, cada uno entrando sólo unos pocos loops (el conjunto real de caminos depende del resto de la estructura

del flujo de control). En cambio, una estrategia ad hoc, que requiera que cada camino entre en cada loop el mismo número de veces, sería más útil y se puede incorporar fácilmente a nuestro algoritmo.

Ahora queremos volver al ddgraph en la Figura 6.5, para ilustrar cómo nuestro algoritmo se puede aplicar para combinar una política de construcción de caminos con prácticas indirectas derivadas del análisis de caminos de ejecución. En este ejemplo ya hemos señalado algunas incompatibilidades claras entre arcos en S. Se pueden tener en cuenta estas incompatibilidades fácilmente, estableciendo restricciones apropiadas en el conjunto del que SELECT-AN-ARC recoge el próximo arco. Por ejemplo, mientras construye un camino por e_7, SELECT-AN-ARC no debe recoger e_{12}, e_{16}, e_{20}. Siguiendo todavía la política *less-pred*, pero teniendo en cuenta las incompatibilidades precedentes, se obtienen los caminos siguientes:

$p''_1 = e_0e_2e_3e_{24}$
$p''_2 = e_0e_1e_2e_3e_{24}$
$p''_3 = e_0e_2e_4e_9e_{11}e_{20}e_{24}$
$p''_4 = e_0e_2e_4e_9e_{11}e_{21}e_{22}e_{23}e_{24}$
$p''_5 = e_0e_2e_4e_5e_7e_8e_9e_{11}e_{21}e_{22}e_{23}e_{24}$
$p''_6 = e_0e_2e_4e_6e_7e_8e_9e_{11}e_{20}e_{24}$
$p''_7 = e_0e_2e_4e_9e_{10}e_{13}e_{14}e_{19}e_{21}e_{22}e_{23}e_{24}$
$p''_8 = e_0e_2e_4e_9e_{10}e_{12}e_{14}e_{19}e_{20}e_{24}$
$p''_9 = e_0e_2e_4e_9e_{10}e_{13}e_{14}e_{15}e_{17}e_{18}e_{19}e_{21}e_{22}e_{23}e_{24}$
$p''_{10} = e_0e_2e_4e_9e_{10}e_{12}e_{14}e_{15}e_{16}e_{18}e_{19}e_{20}e_{24}$

Todos ellos son factibles.

Como hemos dicho, hay dos posibles usos de un criterio del testing: se puede usar para generar casos de test o para determinar si el proceso de testing se puede terminar. Más comúnmente, en la práctica, se usan medidas de cubrimiento estructural para verificar la adecuación. Así, en un contexto del testing "real", podemos suponer que un conjunto de datos de entrada de test ya existe, por ejemplo, derivados de la especificación funcional del programa, y el cubrimiento All-Branches se usa para medir cuán completos son estos datos de prueba. A menudo, los datos de test derivados de especificaciones funcionales no alcanzan para satisfacer cubrimiento All-Branches. En tales casos tienen que encontrarse y ejecutarse otros caminos de test para alcanzar cubrimiento All-Branches. Nuestro algoritmo puede ayudar a hallar estos caminos adicionales. Se debe derivar primero el subconjunto de S de arcos no ejecutados, por ejemplo usando un analizador de cubrimiento [60]. Nuestro algoritmo puede entonces ser ejecutado para derivar no un conjunto entero de caminos que satisface el criterio del testing All-Branches, pero sí un

subconjunto de caminos del ddgraph que cubre los arcos todavía no cubiertos, i.e., $NOT-USED$ se inicializa como el conjunto de estos arcos no cubiertos.

En conclusión sugerimos primero testear los caminos más simples a través de los arcos aún no cubiertos en S, i.e., implementar SELECT-AN-ARC para que siga la política *less-pred*, basándonos en los resultados experimentales de [93]. Sin embargo, el tester podría saber, o aprender a medida que trata de ejecutar estos caminos, que algunas ramas del programa son incompatibles, o viceversa, que algunas ramas se deben ejecutar en el mismo camino. En cualquiera de estos casos, se puede incorporar esta información fácilmente en el algoritmo ajustando adecuadamente el procedimiento SELECT-AN-ARC, en una de las maneras sugeridas más arriba. El algoritmo entonces automáticamente sugiere un conjunto de caminos que ejercen todo los arcos hasta ahora no cubiertos en el ddgraph del programa. Estos caminos, siendo cortos y/o de acuerdo con las restricciones " semánticas" suministradas por el tester, tienen una probabilidad buena de ser ejecutables.

Según nuestro conocimiento, ALL-BRANCHES-TEST-PATH-SET es el único procedimiento de generación de un conjunto de caminos que satisface el criterio de testing All-Branches, que ofrece esta capacidad de manipulación de estrategias diferentes en la construcción de un camino. En [66] y [77] se deriva el conjunto de caminos con una cantidad mínima de caminos. En [93] se usa un método que deriva los caminos más cortos, y si algunas ramas quedan no cubiertas, entonces se identifican los segundos, terceros,...,n-ésimos caminos más cortos. En [92], se proporciona una descripción detallada de cómo usar información suministrada por el tester para recortar caminos no factibles de un conjunto de caminos cortos. Todos estos algoritmos dependen de una política particular.

7.3.4 Análisis Teórico

El teorema 7.1 presentado aquí prueba que el algoritmo ALL-BRANCHES-TEST-PATH-SET es correcto y termina siempre.

Teorema 7.1 *Terminación y Corrección del Algoritmo ALL-BRANCHES-TEST-PATH-SET*

Sea $G = (N, A)$ un ddgraph con arcos distinguidos e_0 y e_k. Entonces el algoritmo ALL-BRANCHES-TEST-PATH-SET aplicado a G termina y devuelve un conjunto de caminos desde e_0 hasta e_k, que cubren a todos los arcos en G.

La demostración puede verse en el Apéndice A.

Ahora realizamos un análisis teórico de la complejidad en tiempo del algoritmo. Ya que en cada llamada recursiva ejecutada por el procedimiento FIND-A-PATH, se cubre un arco (no cubierto) en un spanning set S, y ya que se puede ejecutar cada ciclo como máximo una vez, el número de llamadas recursivas está acotado por $O(|S|)$. Con cada recursión, se ejecuta una llamada a SUB-DDGRAPH, que requiere tiempo $O(|A|)$. Se debe construir los árboles de dominación y post-dominación del sub-ddgraph, y cada uno requiere tiempo $O(|A|\alpha(|A|,|N|))$.

Además, se llama al procedimiento FIND-A-SPANNING-SET-OF-ENTITIES, que requiere tiempo $O(|A|^3\alpha(|A|,|N|))$. (En realidad, en cada iteración del algoritmo, el número de arcos en el sub-ddgraph disminuye, así que $|A|$ es una cota superior). Supongamos que la función SELECT-AN-ARC es $O(s)$. Entonces la construcción de cada camino requiere tiempo $O(|S| * (|A|^3\alpha(|A|,|N|) + s))$. El algoritmo ALL-BRANCHES-TEST-PATH-SET llama a la función FIND-A-PATH como máximo $|S|$ veces, así que el algoritmo toma tiempo $O(|S|^2*(|A|^3\alpha(|A|,|N|)+ s))$.

Sin embargo, en [12] presentamos una versión distinta de FIND-A-SPANNING-SET-OF-ENTITIES para el caso de cubrimiento All-Branches, que requiere tiempo $O(|A|\alpha(|A|,|N|))$. Por lo tanto, el algorithm ALL-BRANCHES-TEST-PATH-SET puede ser implementado en tiempo $O(|S|^2 * (|A|\alpha(|A|,|N|) + s))$.

El orden del procedimiento SELECT-AN-ARC depende de la implementación elegida. Si se usa la política *mínimo-número-de-caminos*, el procedimiento será $O(1)$; si se usa la política *menos-caminos*, será $O(|A|)$. En ambos casos, ALL-BRANCHES-TEST-PATH-SET tomará tiempo $O(|S|^2 * |A|\alpha(|A|,|N|))$.

7.4 Experimentación

Aún cuando en la última sección hemos demostrado que el algoritmo ALL-BRANCHES-TEST-PATH-SET es en teoría correcto y eficiente para el testing All-Branches, su utilidad práctica tiene todavía que ser confirmada. En principio, la tarea del tester se debería hacer más sencilla por tener a disposición el "conjunto de caminos correcto". En la práctica, los parámetros involucrados en un proceso de test son tantos y tan complejos que sólo una validación empírica del método puede ser confiada. En esta sección, describimos un caso de estudio real para validar nuestro método de generación de caminos de test dentro de un ambiente de test del mundo real. Un informe extensivo en este caso de estudio se puede encontrar en [15].

El método para derivar conjuntos de caminos de test se ha implementado como prototipo en

una herramienta llamada BAT (Branch Analysis and Testing). La validación se realizó dentro de la fase de test de unidad del proceso de testing del software, en Ericsson Telecomunicazioni. El software fue desarrollado para controlar una nueva generación de sistemas de la telecomunicaciones (ver Capítulo 6 donde se presenta una descripción completa del ambiente de test). El caso de estudio involucró a la fase de test básica, que es apropiada para la aplicación de la teoría propuesta, ya que incluye el testing All-Branches de unidades de programa hasta un cubrimiento predeterminado. Comparamos el comportamiento del método contra el procedimiento de test realizado hasta el momento, que es esencialmente manual para la parte considerada. Se supervisaron dos testers con habilidades diferentes, y se les asignaron dos conjuntos de programas para testear, un conjunto usando nuestro método y el otro siguiendo el procedimiento normal. Informamos aquí los resultados observados.

7.4.1 Evaluación de la Efectividad de BAT

El objetivo del caso de estudio era evaluar la efectividad de BAT como una herramienta para mejorar el proceso de testing All-Branches. En particular, nos interesamos en validar dos aspectos específicos:

- proporcionar al tester con un conjunto de caminos que garantizan 100% de cubrimiento All-Branches puede reducir el esfuerzo del testing;
- los caminos derivados por BAT es muy probable que sean factibles.

Mientras se planificaba el caso de estudio, encontramos que la evaluación de efectividad de BAT era difícil, debido a los siguientes dos problemas:

1. Para poder comparar el testing con y sin la generación automática de caminos de test, necesitaríamos correr dos sesiones de testing independientes, con y sin BAT, usando el mismo conjunto de unidades de programa. Sin embargo, si se usa una misma persona para correr ambas sesiones, claramente la segunda sesión se beneficiaría de la experiencia ganada en la sesión previa, no importa qué caso se toma primero. Por otro lado, si usamos a dos personas diferentes para correr los dos sesiones de testing separadamente, el experimento sería afectado por la diferencia posible entre la habilidad de cada tester.
2. Aún cuando podríamos superar la primera dificultad, por ejemplo hallando dos testeres absolutamente equivalentes, nos enfrentamos con el problema que las unidades de programa que se testearon eran parte de un proyecto real, y así los recursos de testing y tiempo eran

limitados. Esta segunda dificultad nos convenció más allá de cualquier duda de que no podíamos duplicar el test de cada programa.

Antes de comenzar la fase de testing, se seleccionaron los módulos más grandes y más complejos con lo cuales validar el uso de la herramienta BAT. Luego se definió la estrategia de testing. Se eligió un tester experto y uno principiante para el caso de estudio. Ninguno de ellos había codificado los módulos que iban a testear.

Se estructuró la fase de test como sigue:

1. ejecutar test básico en la manera tradicional basándose en la especificación del test;
2. evaluar el cubrimiento obtenido;
3. dividir al azar el conjunto de unidades de programa que no han cumplido los requerimientos de cubrimiento del testing en dos grupos: el grupo STANDARD, para ser testeado siguiendo los procedimientos internos standard, i.e., sin el apoyo de la herramienta BAT, y el grupo BAT, para ser testeado usando la herramienta BAT. En particular, para el grupo STANDARD el tester trató de seleccionar manualmente tests adicionales que aumentarían el cubrimiento. Para los procedimientos de BAT, el tester trató de probar los caminos sugeridos por la herramienta. En ambos casos, el testing fue terminado cuando se alcanzó un umbral dado de promedio del cubrimiento para todas las unidades del programa en un módulo.

Los módulos a ser testeados se seleccionaron según la prioridad del proyecto en ese momento. Para evitar una distorsión de los resultados, las unidades de los dos grupos fueron probadas en forma alternada, i.e., una del grupo STANDARD y luego una del grupo BAT.

Se siguieron criterios idénticos en la recolección de datos para los tests ejecutados con y sin la herramienta; para cada unidad: el número de branches en la unidad, los cubrimientos inicial y final, y el tiempo (expresado en minutos) necesario para ejecutar el testing.

Los resultados no permitieron realizar evaluaciones estadísticas significativas (como habíamos esperado dado los problemas 1 y 2 de más arriba). Sin embargo, podemos derivar algunas conclusiones interesantes.

- La experiencia con BAT confirmó la asunción de que los caminos derivados por la herramienta son factibles con una probabilidad muy buena. Todos los caminos sugeridos por la herramienta para las unidades del programa testeado (acumulativamente, aproximadamente 80 caminos para el grupo BAT) eran factibles.

- La efectividad de BAT está fuertemente relacionada con la experiencia del tester. En particular el tester experto no encontró a BAT demasiado útil, ya que podía derivar el "camino correcto" inmediatamente, así que haberlo forzado a usar la herramienta realmente atrasó su trabajo. Sólo encontró útil a BAT para secuencias de programa complicadas, cuando los casos del test no eran obvios y su experiencia no era suficiente para alzanzar el cubrimiento requerido. En el caso de estudio, en cambio, lo forzamos a usar la herramienta para todas las unidades de programa en el grupo BAT.

  Sin embargo, en estas evaluaciones se debe considerar que el tester experto elegido para este caso de estudio fue el mejor experto en testing en el Departamento, un tester realmente muy bueno, y esto puede haber desviado la evaluación.

- Una observación interesante es que, analizando después los caminos escogidos a mano por el tester experto para las unidades del programa en el grupo STANDARD, vimos que eligió exactamente el conjunto de caminos que habría sido sugerido por BAT. Dada la experiencia y habilidad del tester, consideramos que es una confirmación importante (aunque anecdótica) de que BAT funciona bien.

- Un tester que no es especialista, en cambio, puede encontrar bastante útil la herramienta: el principiante sintió que podía trabajar en una manera más productiva y que podía derivar un número más bajo de casos de test para alzanzar los requerimientos del cliente, para las unidades del programa testeadas usando BAT. El uso de la herramienta BAT aumentó la productividad del principiante (medida como el número de branches cubiertas por minuto), acercándola a la productividad del experto. De hecho, comparando los resultados obtenidos por los dos testers, uno puede observar que, sin usar BAT, el experto obtuvo resultados que eran casi 5 veces superiores a los que obtuvo el principiante. Sin embargo, después de introducir BAT, las ejecuciones de los dos testers se volvieron, hasta cierto punto, comparables.

  Notamos que en esta comparación, los dos grupos de unidades, con y sin BAT, eran homogéneas con respecto a un único tester, mientras que eran heterogéneas considerando los dos testers. Para el tester principiante los estudios presentan un cubrimiento inicial igual a 0 en la mayoría de casos, mientras que para el experto, el cubrimiento inicial a menudo es alto. Así, hasta un cierto punto, el trabajo del principiante era más complicado que el del experto.

```
Algorithm CONSTRUCT-A-PATH-COVERING-A-DUA (G: ddgraph; T: dua):
path;
begin
    G' = COVERING-PATHS(G, T);
    p = CONSTRUCT-A-COMPLETE-PATH(G');
    return(p);
end
```

Figuras 7.3: CONSTRUCT-A-PATH-COVERING-A-DUA Algorithm

7.5 Un Algoritmo General de Generación de Caminos

7.5.1 Generalidades

En esta sección mostramos cómo usar la información proporcionada por un spanning set of entities para construir, para un ddgraph dado G y un criterio de cubrimiento dado c, un conjunto de caminos que cubre toda entidad en E_c. Presentamos un algoritmo general llamado CONSTRUCT-A-TEST-PATH-SET, que construye tal conjunto de caminos, tratando de cubrir todas las entidades en un spanning set. La definición de spanning set of entities garantiza que el conjunto de caminos encontrado implica el cubrimiento de todas las entidades en E_c.

Primero, presentamos varios algoritmos que serán útiles para introducir el algoritmo general CONSTRUCT-A-TEST-PATH-SET.

Usamos el procedimiento CONSTRUCT-A-COMPLETE-PATH, que dado un ddgraph G devuelve un camino completo en G, i.e., devuelve un camino desde el arco de entrada hasta el arco de salida de G. Este camino se puede construir usando alguna política particular[2]. Para implementar este procedimiento, podemos usar una versión modificada del algoritmo FIND-A-PATH presentado en la Sección 7.3, que genera un camino completo usando la información proporcionada por los árboles de dominación y de post-dominación, y un spanning set of entities. Sin embargo, también se puede implementar este procedimiento en tiempo $O(|A|^2)$, usando un algoritmo depth first search [75].

Usamos el algoritmo CONSTRUCT-A-PATH-COVERING-AN-ARC, que se puede implementar usando una versión modificada del algoritmo FIND-A-PATH presentado en la Sección 7.3 que genera un camino completo que cubre un arco seleccionado.

El algoritmo CONSTRUCT-A-PATH-COVERING-A-DUA, presentado en la Figura 7.3, con-

[2] En particular, esta política se puede elegir arbitrariamente.

```
Algorithm CONSTRUCT-A-COVERING-PATH (G: ddgraph; E: entity):
path;
begin
    case type(E) do
    arc ⟹
        p = CONSTRUCT-A-PATH-COVERING-AN-ARC(G, E);
    dua ⟹
        p = CONSTRUCT-A-PATH-COVERING-A-DUA (G, E);
    set of duas ⟹
        T = SELECT-A-DUA(E);
        p = CONSTRUCT-A-PATH-COVERING-A-DUA (G, T);
    path ⟹
        p = E;
    return(p); end
```

Figuras 7.4: CONSTRUCT-A-COVERING-PATH Algorithm

struye un camino completo en un ddgraph G que cubre una dua dada T. Para hacer esto, usamos el procedimiento COVERING-PATHS presentado en la Figura 4.7, para construir un grafo que representa todos los caminos completos en G que cubren la dua T. Entonces, usando CONSTRUCT-A-COMPLETE-PATH, selecciona y devuelve un camino completo en tal grafo.

El algoritmo CONSTRUCT-A-COVERING-PATH presentado en la Figura 7.4 construye un camino completo en un ddgraph que cubre una entidad particular en un spanning set of entities. De acuerdo al tipo de la entidad (arco, dua, clase de duas o camino) este algoritmo procede en maneras diferentes. Si el tipo de entidad es arco, entonces construye un camino llamando a CONSTRUCT-A-PATH-COVERING-AN-ARC. Si el tipo de entidad es dua, construye un camino llamando a CONSTRUCT-A-PATH-COVERING-A-DUA. Si el tipo de entidad es clase de duas, entonces el algoritmo selecciona una dua en la clase, y hace una llamada a CONSTRUCT-A-PATH-COVERING-A-DUA. Si el tipo de entidad es camino, entonces el algoritmo devuelve ese mismo camino.

Ahora podemos introducir el algoritmo general. En la Figura 7.5 presentamos el algoritmo CONSTRUCT-A-TEST-PATH-SET, que dado un ddgraph G y un criterio de cubrimiento c, construye un conjunto de caminos cubriendo toda entidad según ese criterio de cubrimiento en ese ddgraph.

```
Algorithm CONSTRUCT-A-TEST-PATH-SET (G: ddgraph; c: coverage
criterion): set of paths;
begin
    S = FIND-A-SPANNING-SET-OF-ENTITIES(G, E_c(G));
    ℘ = ∅;
    while U ≠ ∅ do
        e = SELECT-AN-ENTITY(S);
        p = CONSTRUCT-A-COVERING-PATH(G, e);
        for each e ∈ S do
            if COVERS?(p, e) then S = S − {e};
        ℘ = ℘ ∪ {p};
    return(℘);
end
```

Figuras 7.5: CONSTRUCT-A-TEST-PATH-SET Algorithm

El algoritmo primero construye un spanning set of entities (usando el procedimiento FIND-A-SPANNING-SET-OF-ENTITIES presentado en el Capítulo 4.1). Luego, selecciona una entidad unconstrained E hasta ahora no cubierta, en el spanning set S elegido, y construye un camino p que cubre E, llamando al procedimiento CONSTRUCT-A-COVERING-PATH presentado en la Figura 7.4. Entonces, elimina de S todas las entidades cubiertas por p. Se repite este procedimiento hasta que se cubren todas las entidades en S.

El algoritmo CONSTRUCT-A-TEST-PATH-SET deja como un parámetro abierto la política de selección de la próxima entidad unconstrained, haciendo una llamada a una función genérica SELECT-AN-ENTITY. Una vez que se ha escogido una entidad, el algoritmo CONSTRUCT-A-TEST-PATH-SET construye un camino que cubre esa entidad.

La política de selección del próximo arco unconstrained usada para construir ese camino es también un parámetro abierto. Se pueden hacer estas selecciones al azar, o bien se pueden implementar otras políticas para garantizar alguna propiedad especial del conjunto de caminos generado.

7.5.2 Análisis Teórico

En esta sección mostramos que el algoritmo CONSTRUCT-A-TEST-PATH-SET es correcto y siempre termina. También presentamos un análisis de la complejidad temporal del algoritmo.

El primer resultado presentado se concluye a partir del Teorema 7.1.

Teorema 7.2 *Terminación y Corrección de CONSTRUCT-A-PATH-COVERING-AN-ARC*

Sea $G=(N, A)$ un ddgraph con arco de entrada e_0 y arco de salida e_k, c un criterio de cubrimiento, sea e un arco en G.

Entonces, CONSTRUCT-A-PATH-COVERING-AN-ARC(G, e) termina y devuelve un camino completo en G que cubre al arco e.

Teorema 7.3 *Terminación y Corrección de CONSTRUCT-A-PATH-COVERING-A-DUA*

Sea $G=(N, A)$ un ddgraph con arco de entrada e_0 y arco de salida e_k, c un criterio de cubrimiento, sea $T=[d, u, X]$ una dua en G.

Entonces, CONSTRUCT-A-PATH-COVERING-A-DUA(G, T) termina y devuelve un camino completo en G que cubre T.

La demostración puede verse en el Apéndice A.

Teorema 7.4 *Terminación y Corrección de CONSTRUCT-A-COVERING-PATH*

Sea $G=(N, A)$ un ddgraph con arco de entrada e_0 y arco de salida e_k, c un criterio de cubrimiento, sea e una entidad en G.

Entonces, CONSTRUCT-A-COVERING-PATH(G, e) termina y devuelve un camino completo en G que cubre e.

La demostración es directa a partir de los teoremas presentados más arriba.

El teorema 7.5 prueba que el algoritmo CONSTRUCT-A-TEST-PATH-SET es correcto y termina siempre.

Teorema 7.5 *Terminación y Corrección del Algoritmo CONSTRUCT-A-TEST-PATH-SET*

Sea $G=(N, A)$ un ddgraph con arco de entrada e_0 y arco de salida e_k, c un criterio de cubrimiento.

Entonces, CONSTRUCT-A-TEST-PATH-SET(G, c) termina y devuelve un conjunto de caminos completos en G que cubren a todas las entidades en $E_c(G)$.

La demostración puede verse en el Apéndice A.

Discutimos ahora brevemente el número de caminos generado por el algoritmo. De la estructura del algoritmo, se puede ver fácilmente que el número de caminos generado no es más grande que la cardinalidad de un spanning set of entities.

Ahora realizamos un análisis teórico de la complejidad temporal del algoritmo para un dd-graph $G = (N, A)$. La complejidad temporal de FIND-A-SPANNING-SET-OF-ENTITIES es $O(|E_c(G)|^2 * t)$ (see Chapter 4), donde t es

$$\begin{cases} O(|D|^2) & \text{si las entidades son clases de duas} \\ O(|A|) & \text{en otro caso} \end{cases}$$

Podemos suponer que SELECT-AN-ENTITY es una función de tiempo constante. Supongamos que S es el spanning set of entities construido por el algoritmo, i.e., S= FIND-A-SPANNING-SET-OF-ENTITIES$(G, E_c(G))$. La complejidad temporal de CONSTRUCT-A-COVERING-PATH es el máximo entre las complejidades temporales de CONSTRUCT-A-PATH-COVERING-AN-ARC y CONSTRUCT-A-PATH-COVERING-A-DUA, i.e., i.e., $\max\{O(|S|*|A|^3*\alpha(|A|,|N|)), O(|A|^2)\}$. Entonces, CONSTRUCT-A-COVERING-PATH toma tiempo $O(|S|*|A|^3*\alpha(|A|,|N|))$. Eliminar de S todas las entidades cubiertas por el camino construido toma tiempo $O(|S| * |A|)$.

Ya que por lo menos una entidad todavía no cubierta en S se cubre con cada iteración ejecutada por CONSTRUCT-A-TEST-PATH-SET, el número de iteraciones está acotado por $|S|$. En cada iteración, seleccionamos una entidad nueva en tiempo constante; ejecutamos CONSTRUCT-A-COVERING-PATH, en $O(|S|*|A|^3*\alpha(|A|,|N|))$; y eliminamos de S todas las entidades cubiertas por el camino construido en $O(|S| * |A|)$. Por consiguiente, cada iteración toma tiempo $O(|S| * |A|^3 * \alpha(|A|,|N|))$.

En conclusión CONSTRUCT-A-TEST-PATH-SET toma tiempo $O(|E_c(G)|^2 * t + |S|^2 * |A|^3 * \alpha(|A|,|N|))$.

7.6 Conclusiones y Trabajo Futuro

En este capítulo hemos mostrado cómo usar la información proporcionada por un spanning set of entities para construir, para un ddgraph dado G y un criterio del cubrimiento dado c, un conjunto de caminos cubriendo toda entidad en E_c.

Primero introducimos el algoritmo ALL-BRANCHES-TEST-PATH-SET, que halla un conjunto de caminos que satisface el criterio de cubrimiento All-Branches para un programa dado representado por un ddgraph. Hemos demostrado que este algoritmo es correcto. También hemos reportado un caso de estudio real. Hemos obtenido sólo evidencia cualitativa de que el método es muy útil, aunque principalmente para testeres no expertos.

En particular, observamos que la productividad de un tester no especialista usando el método propuesto se vuelve comparable a la de un tester experto. Pudimos confirmar la efectividad

del método usado por la herramienta para generar caminos de test, ya que todos los caminos propuestos eran factibles. Notamos que pudimos verificar, después de terminado el experimento, que para las unidades testeadas a mano por el experto, los caminos sugeridos por la herramienta fueron idénticos a los que se escogieron.

Entonces, hemos presentado un algoritmo general llamado CONSTRUCT-A-TEST-PATH-SET que construye un conjunto de caminos enfocándose en cubrir todas las entidades en un spanning set. La definición de un spanning set of entities garantiza que el conjunto de caminos encontrado implica el cubrimiento de todas las entidades para ese programa y ese criterio de cubrimiento.

Capítulo 8

Conclusiones

En esta tesis hemos presentado una teoría general que apunta a mejorar la relación costo/efectividad del proceso de testing. Proponemos reducir el costo de testear un programa reduciendo el conjunto de entidades a ser cubiertas para satisfacer un criterio de test. Proponemos estimar el costo de testear un programa limitando el número de tests a ser cubiertos para satisfacer un criterio de test.

El método que hemos presentado para reducir y estimar el número de tests necesarios para satisfacer criterios de cubrimiento estructurales se basa en la observación de que un test generalmente cubre más de una entidad. Tradicionalmente, no se hace ningún esfuerzo para generar a priori datos de test que satisfacen varios requerimientos a la vez. En este trabajo hemos identificado mediante análisis estático un subconjunto de entidades de *mínima* cardinalidad con la propiedad que cualquier conjunto de caminos que cubre a este subconjunto de entidades, cubre toda entidad en el programa. Hemos llamado a este subconjunto mínimo un "spanning set of entities".

La información proporcionada por un spanning set of entities se pueden usar para:

- *Reducir el costo del testing*: la generación de casos de test puede ser dirigida para cubrir un (mínimo) spanning set of entities, en lugar de todas las entidades en un programa, reduciendo el número de test y previniendo la selección de caminos redundantes.

  En esta tesis hemos estudiado el uso de un spanning set of entities para verificar si el testing de cubrimiento es completo (y así, evaluar la completitud del testing más efectivamente); para prevenir la generación de caminos redundantes, enfocando la generación de casos de test en el cubrimiento de un spanning set of entities; para guiar la selección de entidades

para aumentar el cubrimiento; para tratar el problema de no-factibilidad. Además, hemos introducido una familia nueva de criterios de cubrimiento de test, basados en el concepto de spanning sets of entities.

- *Estimar el costo del testing*: el número de entidades en un spanning sets of entities se puede usar como una estimación del número de tests necesarios para satisfacer un criterio de cubrimiento elegido.

  En esta tesis hemos analizado el problema de establecer cuántos casos de test se deben pensar para alcanzar un cubrimiento particular durante el desarrollo del software. Hemos presentado dos medidas diferentes para ayudar a estimar el costo de aplicar una estrategia de test. Hemos presentado algunos resultados experimentales sobre estas cotas.

- *Generar test suites*: se pueden seleccionar caminos de test que cubran las entidades en un spanning set of entities, aún no cubiertas.

  En esta tesis hemos presentado el algoritmo de generación de caminos CONSTRUCT-A-TEST-PATH-SET, que usa la información proporcionada por un spanning set of entities para construir un conjunto de caminos de test que satisface un criterio de cubrimiento dado[1]. La definición de un spanning set of entities garantiza que el conjunto de caminos encontrado implica el cubrimiento de todas las entidades. Hemos presentado experimentación.

Como hemos dicho, esta teoría se ha desarrollado apuntando a mejorar la relación costo/efectividad del proceso de testing. Creemos que el uso de esta teoría en la práctica reducirá realmente el costo del testear un programa y dará realmente una estimación buena del costo del proceso de testing.

Sin embargo, como con muchas técnicas de reducción del costo en testing, el costo de aplicar la técnica se debe comparar con el ahorro real que proporciona. En esta tesis, hemos presentado experimentación preliminar. Sin embargo, para confirmar la utilidad de la nueva teoría se necesitan más experimentos. Nuestro trabajo futuro en el corto término será investigar empiricamente la proporción entre el número total de entidades y el número más pequeño de entidades en un spanning set. De esta manera, queremos estimar en cuánto se reduce el costo del testing al usar un spanning set of entities. Ya que la identificación de un spanning set toma tiempo polinómico y se puede automatizar, es de esperar que sea más eficiente que seleccionar tests al azar hasta que se alcanza cubrimiento total.

[1] La generación de datos de test para ejecutar los caminos seleccionados está fuera del alcance de este trabajo.

Además de las extensiones propuestas en cada capítulo, algunos otros trabajos podrían hacerse basados en los resultados de esta tesis. En particular, la aplicación de la nueva teoría para tratar el problema de *aliasing* [5]. Además, estudiaremos su aplicación al testing de regresión.

Además, los conceptos de spanning set y subsumption presentados aquí se puede aplicar a otros criterios diferentes. Dado un criterio nuevo, primero se debe identificar el conjunto de requerimientos. Luego, se debe implementar la relación de subsumption. De esta manera, obtendremos una jerarquía de requerimientos que nos permitirá identificar un spanning set, i.e., un conjunto mínimo de requerimientos que se deben ejecutar para satisfacer el criterio nuevo. Pensamos extender estos conceptos a diferentes criterios de cubrimiento.

Apéndice A

Demostraciones

Teorema 4.1

Sea G un ddgraph, c un criterio de cubrimiento, y $R_c(G)$ el reduced c-subsumption digraph de G. Sea U un conjunto de c-entities de $E_c(G)$ representando las hojas de $R_c(G)$, i.e., cada entidad en U representa una hoja en $R_c(G)$ y todas las hojas en $R_c(G)$ tienen un representante en U. Entonces U es un spanning set of entities para G y c.

Demostración:

Probaremos que se cumplen los dos puntos de la Definición 4.1.

1. *Un conjunto de caminos $\wp$ que cubre toda entidad en U cubre todas las entidades en $E_c(G)$.*

   Por construcción, $\wp$ cubre todas las entidades en todas las hojas de $R_c(G)$. Sea E una entidad en el nodo n en $R_c(G)$, que no es una hoja. Entonces, existe al menos un arco en $R_c(G)$ desde n hasta algún nodo n'. Si n' no es una hoja, entonces existe al menos un arco en $R_c(G)$ desde n' hasta algún nodo n''. Ya que $R_c(G)$ es acíclico y finito, existe un camino desde n hasta algún n_l, que es hoja en $R_c(G)$. Un camino que cubre n_l debe cubrir n; y por construcción, hay un camino en $\wp$ que cubre n_l.

2. *Para cualquier $U' \subseteq E_c(G)$, tal que cualquier conjunto de caminos que cubre toda entidad en U' cubre todas las entidades en $E_c(G)$, vale que $|U_c(G)| \leq |U'|$.*

   En particular, tal conjunto de caminos debe cubrir todas las entidades en U. Supongamos que $|U'| < |U|$. Entonces, existe al menos una hoja n en U tal que para todo E en n, E no está en U'. Pero ninguna entidad en U' subsume a $rep(n)$. Por lo tanto, existe al menos un conjunto de caminos que cubre a U' pero no a $rep(n)$.

Proposición 4.1

AL(a, b, c) vale si y sólo si b pertenece al camino característico del sub-ddgraph(G, a, c).

Demostración: b pertenece al camino característico de sub-ddgraph (G, a, c)

$\Longleftrightarrow$ $\forall$ p camino completo en sub-ddgraph(G, a, c), p contiene a b

$\Longleftrightarrow$ $\forall$ p camino de a a c que no pasa por a ni por c más de una vez en G, p contiene a b

$\Longleftrightarrow$ $\forall$ p camino de a a c, p contiene a b (de hecho, si p pasa por a o c más de una vez, siempre contiene un subpath de a a c que no pasa por a ni por c más de una vez)

$\Longleftrightarrow$ AL(a, b, c).

Teorema 4.2

Sean e y e' dos arcos en un ddgraph G con arco de entrada e_0 y arco de salida e_k. Entonces, e subsume a e' si y solo si e' domina o post-domina a e.

Demostración:

$\Rightarrow$) Supongamos que e subsume a e'. Y supongamos que no es verdad que e' domina a e o e' post-domina a e. Entonces, existe un camino p_1 de e_0 a e que no contiene a e', y también existe un camino p_2 de e a e_k que no contiene a e'. Sea p el camino de e_0 a e_k obtenido atravesando los arcos en p_1 de e_0 a e y luego continuando con los arcos en p_2 hasta e_k:

$$p = \overbrace{e_0, \ldots, e}^{p_1}, \underbrace{\ldots, e_k}_{p_2}$$

Así, p es un camino completo que contiene a e pero no contiene a e'; por lo tanto, e no subsume a e'. Y ésta es una contradicción con la hipótesis.

$\Leftarrow$) Si e' domina a e, entonces todo camino de e_0 a e contiene a e'. Por otro lado, si e' post-domina a e, entonces cada camino de e a e_k contiene a e'. Por lo tanto, en cualquiera de los dos casos todo camino completo que contiene a e contiene también a e', i.e., e subsume a e'.

Teorema 4.3

Sean e y e' dos arcos en el ddgraph G con arco de entrada e_0 y arco de salida e_k. Entonces, e subsume e' si y solo si AL(e_0,e',e) o AL(e,e',e_k).

Demostración: e subsume a e'

$\overset{Theorem 4.2}{\Longleftrightarrow}$ e' domina a e o e' post-domina a e

$\overset{Proposition 4.2}{\Longleftrightarrow}$ AL(e_0,e',e) o AL(e,e',e_k).

Proposición 4.3

Sea $G=(N, A)$ un digraph. Sean $n_1, n_2 \in N$, y X una variable en G.

Entonces, $\{p$: p es un camino desde n_1 hasta n_2 en ALL-DEF-CLEAR-PATHS$(n_1, n_2, X, G)\}$ = $\{p$: p es un camino desde n_1 hasta n_2 en G que no contiene una definición de $X\}$.

Demostración:

⇒) Sea p un camino desde n_1 hasta n_2 en ALL-DEF-CLEAR-PATHS(n_1, n_2, X, G), donde ALL-DEF-CLEAR-PATHS(n_1, n_2, X, G) = ALL-PATHS(n_1, n_2, G_1), $G_1=(N_1, A_1)$, $A_1 = \{e \in A$: e no contiene ninguna definición de $X\}$; $N_1 = \{n$: $\exists e \in A_1$, n=TAIL(e) o n=HEAD$(e)\}$.

Supongamos que e está en p. Ya que $e \in A$, p es un camino en G. Por otro lado, e no contiene ninguna definición de X. Por lo tanto, p es un camino desde n_1 hasta n_2 que es def-clear con respecto a X en G.

⇐) Sea p un camino desde n_1 hasta n_2 en G que no contiene ninguna definición de X. Sea e un arco en p. Entonces, e no contiene ninguna definición de X. Sea $G_1= (N_1, A_1)$, donde $A_1 = \{e \in A$: e no contiene ninguna definición de $X\}$ y $N_1 = \{n$: $\exists e \in A_1$, n=TAIL(e) o n=HEAD$(e)\}$. Por definición, $e \in A_1$. Sea ALL-DEF-CLEAR-PATHS (n_1, n_2, X, G)= (N_p, A_p). Ya que existe un camino desde n_1 hasta n_2 en G_1 que contiene a e, $e \in A_p$.

Entonces, p es un camino desde n_1 hasta n_2 en ALL-DEF-CLEAR-PATHS(n_1, n_2, X, G).

Proposición 4.4

Sea $G=(N, A)$ un ddgraph, e_0 el arco de entrada de G, e_k el arco de salida de G. Sea $T=|d, u, X|$ una dua en G. Sean G_1, G_2 y G_3 los digraphs definidos en la Figura 4.7.

Entonces, $\{p$: p es un camino desde e_0 hasta e_k en COVERING-PATHS$(G, T)\}$ = $\{p$: $\exists p_1, p_2, p_3$, tal que $p = p_1 + \langle\ d\ \rangle + p_2 + \langle\ u\ \rangle + p_3$; p_1 es un camino en G_1 desde e_0 hasta TAIL(d); p_2 es un camino en G_2 desde HEAD(d) hasta TAIL(u); and p_3 es un camino en G_3 desde HEAD(u) hasta $e_k\}$.

Demostración:

⇒) Sea p un camino de e_0 a e_k en COVERING-PATHS (G, T), $p = e_0 \ldots, e_k$.

Por construcción, G_1, G_2 y G_3 son digrafos distintos, sin conexión entre cualquiera dos de ellos. Entonces, ningún arco en G_i alcanza un arco en G_j, para $i, j = 1, 2, 3$ y $i \neq j$. El arco d se debe usar para ir de G_1 a G_2, y el arco u se debe usar para ir de G_2 a G_3. Ya que $p = e_0 \ldots, e_k$ y e_0 es un arco en G_1, y e_k está en G_3, entonces d y u están en p. Entonces,

$p = \underbrace{e_0, \ldots,}_{p_1} d \overbrace{\ldots,}^{p_2} u \underbrace{, \ldots, e_k}_{p_3}$, p_1 es un camino en G_1, p_2 es un camino en G_2 y p_3 es un camino en G_3.

$\Leftarrow$) Sea p un camino tal que exiten tres caminos p_1, p_2, p_3, tales que

1. $p = p_1 + \langle\, d \,\rangle + p_2 + \langle\, u \,\rangle + p_3$;
2. p_1 es un camino en G_1 desde e_0 hasta TAIL(d);
3. p_2 es un camino en G_2 desde HEAD(d) hasta TAIL(u); y
4. p_3 es un camino en G_3 desde HEAD(u) hasta e_k.

Ya que p_1 es un camino en G_1, entonces es un camino en COVERING-PATHS(G, T). De la misma manera, p_2, p_3, $\langle\, d \,\rangle$ y $\langle\, u \,\rangle$ son caminos en COVERING-PATHS(G, T). Por lo tanto, p es un camino en COVERING-PATHS(G, T).

Corolario 4.1

Sea $G=(N, A)$ un ddgraph, e_0 el arco de entrada de G, e_k el arco de salida de G. Sea $T=[d, u, X]$ una dua en G. Entonces, todo camino desde e_0 hasta e_k en COVERING-PATHS(G, T) cubre a T.

Demostración: Un camino de e_0 a e_k en COVERING-PATHS(G, T) tiene la siguiente forma: $p = \underbrace{e_0, \ldots,}_{p_1} d \overbrace{\ldots,}^{p_2} u \underbrace{, \ldots, e_k}_{p_3}$.

Ya que el camino p_2 es un camino en G_2, no contiene ninguna definición de X. Entonces, p contiene un camino def-clear con respecto a X de d a u, i.e., p cubre a T.

Proposición 4.5

Sea $G=(N, A)$ un ddgraph, e_0 el arco de entrada de G, e_k el arco de salida de G. Sea $T=[d, u, X]$ una dua en G. Entonces, $G^* = (N^*, A^*)$ = REDUCE(COVERING-PATHS(G, T)) es un ddgraph.

Demostración: Notamos que, por construcción, G^* es un digrafo.

Luego, notamos que e_0 y e_k están en G^*. De hecho, por definición, e_0 es un arco en ALL-PATHS(e_0, TAIL(d), G), y e_k es un arco en ALL-PATHS(HEAD(u), e_k, G). Entonces, por construcción, pertenecen a A^*. Notamos también que $indegree(TAIL(e_0)) = 0$ y $outdegree(TAIL(e_0)) = 1$, $indegree(HEAD(e_k)) = 1$ y $outdegree(HEAD(e_k)) = 0$, porque no agregamos ningún arco durante esta construcción.

Ahora, sea $e \in A^*$. Así, $e \in A_1 \cup A_2 \cup A_3 \cup \{d, u\}$, por el procedimiento COVERING-PATHS en la Figura 4.7. Por un lado, si $e \in A_1$, entonces existe un camino en G_1 de e_0 a e; este camino está también en G^*; entonces, e_0 alcanza a e. De la misma manera, podemos mostrar que e_0 alcanza a cualquier arco en $A_2 \cup A_3 \cup \{d, u\}$. En cambio, si $e \in A_3$, entonces existe un camino en G_3 de e a e_k; este camino está también en G^*; entonces, e alcanza a e_k. De la misma manera, podemos mostrar que e alcanza a e_k, para cualquier arco $e \in A_2 \cup A_3 \cup \{d, u\}$. Entonces, cualquier arco en A^* es alcanzado por e_0 y alcanza a e_k.

Ya que G^* = REDUCE(COVERING-PATHS(G, T)), cada nodo $n \in N^*$, $n \neq TAIL(e_0)$, $n \neq HEAD(e_k)$, $(indegree(n) + outdegree(n)) > 2$.

Por consiguiente, REDUCE(COVERING-PATHS(G, T)) es un ddgraph.

Teorema 4.4

Sea G un ddgraph, e_0 el arco de entrada de G, e_k el arco de salida de G. Sea T=$[d, u, X]$ una dua en G. Sea G' = REDUCE(COVERING-PATHS(G, T)).

Entonces, $\{p$: p es un camino completo en $G'\}$ = $\{p$: p es un camino completo en G que cubre a $T\}$.

Demostración:

⇒) Por el Corolario 4.1, cada camino completo en G' cubre a T. Y, por construcción, si p es un camino en G', entonces es un camino en G.

⇐) Sea p un camino de e_0 a e_k en G tal ese p cubre a T:

$p = \underbrace{e_0, ..., d}_{p_1} \overbrace{, ...,}^{p_2} \underbrace{u, ..., e_k}_{p_3}$; i.e., $p_1 = e_0, ..., TAIL(d)$, $p_2 = HEAD(d), ..., TAIL(u)$, $p_3 = HEAD(u), ..., e_k$, y $p = p_1 + \langle\, d \,\rangle + p_2 + \langle\, u \,\rangle + p_3$. Ya que p cubre a T, contiene un subcamino def-clear con respecto a X desde d a u. Entonces, por construcción y Proposición 4.3, p_1 es un camino en G_1, p_2 es un camino en G_2 y p_3 es un camino en G_3. Entonces, por Proposición 4.4, p es un camino en G'.

Teorema 4.5

Sea G un ddgraph, e_0 el arco de entrada de G, e_k el arco de salida de G. Sean T_1=$[d_1, u_1, X_1]$ y T_2=$[d_2, u_2, X_2]$ dos duas en G. Sea G' = REDUCE(COVERING-PATHS(G, T_1)). Sea G''=SUB-DDGRAPH(G', d_2, u_2).

Entonces, T_1 subsume a T_2 si y solo si

1. d_2 pertence al camino característico de G', y

2. u_2 pertence al camino característico de G', y

3. si ningún arco distinto de d_2 y u_2 en G'' contiene una definición de X_2.

Demostración:

$\Rightarrow$) Supongamos que T_1 subsume a T_2, entonces, cada camino completo p en G que cubre a T_1 cubre a T_2 también.

Supongamos que la condición 1 no se cumple. Entonces, existe por lo menos un camino $p = e_0 ... d_1 ..., u_1 ..., e_k$ en G' que no contiene al arco d_2. Así, p no cubre a T_2. Sin embargo, por el Teorema 4.4, p es un camino en G, y p cubre a T_1. Por consiguiente, T_1 no subsume a T_2. Ésta es una contradicción con la hipótesis, y así, la condición 1 debe cumplirse.

De la misma manera, podemos demostrar que la condición 2 debe cumplirse.

Ahora, supongamos que existe un arco e en G'', $e \neq d_2, u_2$, y e contiene una definición de X_2. Ya que G'' es un ddgraph con arco de entrada d_2 y arco de salida u_2, existe por lo menos un camino $p = d_2 ..., e ..., u_2$ en G''. Por construcción, p es también un camino en G'. Ya que G' es un ddgraph, p es subcamino de por lo menos un camino completo p' en G': $p' = e_0, ..., \underbrace{d_2, ..., e, ..., u_2}_{p}, ..., e_k$.[1] Ya que p' es un camino en G', por el Teorema 4.4, p' es un camino en G que cubre a T_1. Sin embargo, p' no cubre a T_2. Por consiguiente, T_1 no subsume a T_2. Ésta es una contradicción con la hipótesis, y así, la condición 3 debe cumplirse.

$\Leftarrow$) Suponemos que las condiciones 1, 2 y 3 se cumplen.

Sea p un camino completo en G que cubre a T_1. Por el Teorema 4.4, p es un camino de e_0 a e_k en G'. Por hipótesis, d_2, u_2 son arcos en p, i.e., $p = e_0, ..., \underbrace{d_2, ..., u_2}_{p'}, ..., e_k$. Llamamos p' al subpath de p desde d_2 a u_2. Por construcción, p' es un camino completo en G'' (o si no, contiene un camino completo en G''). Por consiguiente, p' no contiene una definición de X_2; es decir, p' es un camino def-clear con respecto a X_2 desde d_2 a u_2. Y por lo tanto, p' cubre a T_2.

[1] Podemos suponer que d_2 y u_2 en p son las únicas ocurrencias de d_2 y u_2 en p'. Si no fuera así, podríamos eliminar las otras ocurrencias.

Teorema 4.6

Sea G un ddgraph, X una variable en G definida en el arco d, X' una variable en G definida en el arco d'. S_d^X subsume $S_{d'}^{X'}$ si y solo si $S_{d'}^{X'} \in \text{SUB}(S_d^X)$.

Demostración:

$S_{d'}^{X'} \in \text{SUB}(S_d^X) \overset{\text{SUB}}{\Longleftrightarrow} \forall T \in S_d^X,\ S_{d'}^{X'} \in \text{TOP}(T) \overset{\text{TOP}}{\Longleftrightarrow} \forall T \in S_d^X,\ \exists T'=[d', u', X]$ tal que T subsume a T'.

Por otro lado, S_d^X subsume a $S_{d'}^{X'} \overset{\text{Subsumption}}{\Longleftrightarrow} \forall\ p$ camino completo, p cubre a S_d^X, entonces p cubre a $S_{d'}^{X'}$

Por lo tanto, necesitamos mostrar que $\forall T \in S_d^X,\ \exists T'=[d', u', X']$ tal que T subsume a T' si y sólo si $\forall\ p$ camino completo, si p cubre a S_d^X, entonces p cubre a $S_{d'}^{X'}$.

$\Rightarrow$) Sea p un camino completo que cubre alguna dua T en S_d^X. Por hipótesis, existe $T'=[d', u', X']$ $\in S_{d'}^{X'}$ tal que T subsume a T'. Entonces, p cubre a $T' \in S_{d'}^{X'}$, i.e., p cubre a $S_{d'}^{X'}$.

$\Leftarrow$) Supongamos que $\exists T \in S_d^X,\ \forall T'=[d', u', X']$: T no subsume a T'. Entonces, $\forall T' \in S_{d'}^{X'}$ existe un camino completo p que cubre a T y no cubre a T'. Es decir, p no cubre a $S_{d'}^{X'}$.

Por otro lado, ya que el camino completo p cubre a T, entonces p cubre a S_d^X. Y esto es una contradicción.

Proposición 6.1

Sea $G = (N, A)$ un ddgraph. Entonces, existe un único spanning set of entities S para G y el criterio All-Branches.

Demostración: Supongamos que existen dos spanning sets of entities S_1, S_2 para G y el criterio All-Branches. Entonces, por la Definición 4.1,

1. $S_1, S_2 \subseteq A$.
2. Un conjunto de caminos $\wp$ que cubre todos los arcos en S_i (for $i = 1, 2$) cubre todos los arcos en A.
3. $|S_1| \leq |S_2|$ y $|S_2| \leq |S_1|$. Entonces, $|S_1| = |S_2|$.

Supongamos que $S_1 \neq S_2$. Por el Teorema 4.1, existe por lo menos una hoja en $R_{All-Branches}(G)$ que contiene por lo menos dos arcos e_1 y e_2. Supongamos que $e_1 \in S_1$, $e_1 \notin S_2$; y $e_2 \in S_2$, $e_2 \notin S_1$. Por construcción e_1 subsume a e_2 y e_2 subsume a e_1. Así, por la Sección 4.3.2:

- e_1 domina a e_2 o e_1 post-domina a e_2, y

- e_2 domina a e_1 o e_2 post-domina a e_1.

Tenemos cuatro posibilidades. Supongamos que e_1 domina a e_2 y e_2 domina a e_1. Entonces (ver Definición 4.3), cada camino p desde el arco de entrada hasta e_2 contiene a e_1; cada camino p desde el arco de entrada hasta e_1 contiene a e_2. En un caso finito, ésto es posible solamente si $e_1 = e_2$. Se pueden analizar las otras tres posibilidades en la misma manera.

Por consiguiente, $S_1 = S_2$.

Teorema 7.1

Terminación y Corrección del Algoritmo ALL-BRANCHES-TEST-PATH-SET

Sea $G = (N, A)$ un ddgraph con arcos distinguidos e_0 y e_k. Entonces el algoritmo ALL-BRANCHES-TEST-PATH-SET aplicado a G termina y devuelve un conjunto de caminos desde e_0 hasta e_k, que cubren a todos los arcos en G.

Demostración:

- *Terminación*: El número de arcos en un spanning set of entities S para G y el criterio All-Branches está limitado por el número de arcos en A. El algoritmo ALL-BRANCHES-TEST-PATH-SET invoca a la función FIND-A-PATH para cada arco en S no cubierto. En cada iteración de cada *while*, e_i será PARENT(e_i, T), siendo entonces imposible no alcanzar la raíz del árbol correspondiente T. Para cada e_i se puede hacer una llamada recursiva a FIND-A-PATH. La función FIND-A-PATH terminará en un número finito de pasos, ya que el sub-ddgraph construido en una llamada recursiva tiene un número de arcos estrictamente menor que el ddgraph. Por consiguiente, el algoritmo ALL-BRANCHES-TEST-PATH-SET termina en un número finito de pasos.

- *Corrección*: Sea $\wp = \{p_1, ..., p_k\}$ el conjunto de caminos que devuelve el algoritmo ALL-BRANCHES-TEST-PATH-SET.

  Para demostrar que el algoritmo es correcto, demostraremos que $\wp$ es un conjunto de caminos completos en G que cubre todos los arcos en G.

  Sea $p_i \in \wp$. Para construir p_i, el algoritmo llama a la función FIND-A-PATH. FIND-A-PATH tiene dos *whiles* diferentes: uno para la construcción de un camino de e_0 al arco seleccionado e_u, y otro para la construcción de un camino de e_u a e_k. El primer *while* encuentra un camino de dominación en el árbol de dominación. El caso base es cuando dos arcos adyacentes en el camino de dominación son arcos adyacentes en el ddgraph,

y el camino en el ddgraph es así un camino de dominación. En otro caso, i.e., dos arcos adyacentes en el camino de dominación no son arcos adyacentes en el ddgraph (i.e., DISC?(PARENT(e_i, $DT(G)$), e_i, $DT(G)$) es verdadero), el algoritmo llama a la función recursiva FIND-A-PATH para encontrar un camino de PARENT(e_i, $DT(G)$) a e_i. Notamos que SUB-DDGRAPH(G, PARENT(e_i, $DT(G)$), e_i) siempre existe y contiene por lo menos un arco unconstrained, ya que es un ddgraph. Así, se construye un camino p' de PARENT(e_u, $DT(G)$) a e_u en G'. Ya que G' se obtiene de G eligiendo algunos caminos en G, cada camino en G' es un camino (o la reducción de un camino) en G. Así, p' es un camino desde PARENT(e_u, $DT(G)$) a e_u en G. Se puede hacer un razonamiento similar para el segundo *while*, que encuentra un camino del post-dominación en el árbol de post-dominación. Por consiguiente, p es un conjunto de caminos completos en G.

Cuando el algoritmo ALL-BRANCHES-TEST-PATH-SET termina, cada arco en el spanning set S está cubierto por lo menos por un camino en p (condición de terminación). De la Definición 4.1, un conjunto de caminos que cubre un conjunto spanning de entidades, cubre todas las entidades para el ddgraph G y el criterio del cubrimiento considerado. Por consiguiente, p cubre todo los arcos en G.

Teorema 7.3

Terminación y Corrección de CONSTRUCT-A-PATH-COVERING-A-DUA

Sea G=(N, A) un ddgraph con arco de entrada e_0 y arco de salida e_k, c un criterio de cubrimiento, sea T=$[d, u, X]$ uns dua en G.

Entonces, CONSTRUCT-A-PATH-COVERING-A-DUA(G, T) termina y devuelve un camino completo en G que cubre T.

Demostración:

Ya sabemos que el procedimiento COVERING-PATHS presentado en la Figura 4.7, construye, para un ddgraph dado G y una dua dada T, un grafo que representa todos los caminos que cubren a T en G (ver Teorema 4.4). El procedimiento CONSTRUCT-A-PATH-COVERING-A-DUA(G, T) selecciona un camino completo en ese grafo usando CONSTRUCT-A-COMPLETE-PATH (ver la Figura 7.3). Entonces, el camino p que devuelve CONSTRUCT-A-PATH-COVERING-A-DUA (G, T) es un camino completo en G que cubre T.

Teorema 7.5

Terminación y Corrección del Algoritmo CONSTRUCT-A-TEST-PATH-SET

Sea $G=(N,\ A)$ un ddgraph con arco de entrada e_0 y arco de salida e_k, c un criterio de cubrimiento.

Entonces, CONSTRUCT-A-TEST-PATH-SET(G, c) termina y devuelve un conjunto de caminos completos en G que cubren a todas las entidades en $E_c(G)$.

Demostración:

- *Terminación*: El número de entidades en un spanning set $|S|$ está acotado por el número de entidades en el ddgraph. El algoritmo CONSTRUCT-A-TEST-PATH-SET invoca a la función CONSTRUCT-A-COVERING-PATH para cada entidad en S no cubierta. Sabemos que el algoritmo CONSTRUCT-A-COVERING-PATH siempre termina (ver Teorema 7.4). Cada camino nuevo cubre por lo menos una entidad todavía no cubierta en S. Así, se reduce por lo menos en uno en cada iteración. Por consiguiente, el algoritmo CONSTRUCT-A-TEST-PATH-SET termina en un número finito de pasos.

- *Corrección*: Sea p el conjunto de caminos que devolvió el algoritmo CONSTRUCT-A-TEST-PATH-SET. Ya sabemos que CONSTRUCT-A-COVERING-PATH realmente construye un camino que cubre una entidad. Cuando el algoritmo CONSTRUCT-A-TEST-PATH-SET termina, cada entidad en S será cubierta por lo menos por un camino en p (condición de terminación). De la Definición 4.1, un conjunto de caminos que cubre un conjunto spanning de entidades cubre todas las entidades en G.

Bibliografía

[1] In *Proceeding of International Quality Week in San Francisco*, May 1993-97.

[2] *STW, Software TestWorks, Test Coverage Tools: TCAT, S-TCAT, TCAT-PATH, T-SCOPE.* SR Software Research, Inc., San Francisco, 1991.

[3] W. R. Adrion, M. A. Branstad, and J. C. Cherniavsky. Validation, verification and testing of computer software. *ACM Comp. Surveys*, 14(2):159–192, June 1982.

[4] H. Agrawal. Dominators, super blocks, and programa coverage. In *Proc. Principles of Programming Languages (POPL'94)*, pages 25–34, January 1994.

[5] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques and Tools.* Addison-Wesley Pub. Co., Reading, Mass., 1986.

[6] R. Bache and M. Müllerburg. Measures of testability as a basis for quality assurance. *Software Engineering Journal*, 86–92, March 1990.

[7] A. L. Baker and S. H. Zweben. A comparison of measures of control flow complexity. *IEEE Transactions on Software Engineering*, SE-6(6):506–512, November 1980.

[8] B. Beizer. *Balck-Box Testing, Techniques for Functional Testing of Software and Systems.* J. Wiley & Sons, 1995.

[9] B. Beizer. *Software Testing Techniques, Second Edition.* Van Nostrand Reinhold, New York, 1990.

[10] C. Berge. *Graphs and Hypergraphs.* North-Holland, 1973.

[11] A. Bertolino. Unconstrained edges and their application to branch analysis and testing of programs. *The Journal of Systems and Software*, 20(2):125–133, February 1993.

[12] A. Bertolino and M. Marré. Automatic generation of path covers based on the control flow analysis of computer programs. *IEEE Transactions on Software Engineering*, SE-20(12):885–899, December 1994.

[13] A. Bertolino and M. Marré. A general path generation algorithm for coverage testing. In *Proceeding of International Quality Week in San Francisco*, May 1997.

[14] A. Bertolino and M. Marré. How many paths are needed for branch testing? *The Journal of Systems and Software*, 35(2):95–106, November 1996.

[15] A. Bertolino, R. Mirandola, and E. Peciola. A case study in branch testing automation. In *Proc. of the IFIP 3rd Int. Conf. on Achieving Quality in Software (AQuIS '96), Florence, Italy*, pages 369–380, 1996.

[16] J. M. Bieman and J. L. Schultz. Estimating the number of test cases required to satisfy the all-du-paths testing criterion. *Proc. ACM SIGSOFT Third Symposium on Software Testing, Analysis, and Verification (TAV3), Key West, Florida*, 179–186, 1989.

[17] B. Boehm. Software engineering. *IEEE Transactions on Computer*, December 1976.

[18] K. Steiglitz C. H. Papadimitriou. *Combinatorial Optimization Algorithms and Complexity.* Prentice-Hall, 1982.

[19] T. Chusho. Test data selection and quality estimation based on the concept of essential branches for path testing. *IEEE Transactions on Software Engineering*, SE-13(5):509–517, May 1987.

[20] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, SE-2():215–222, 1976.

[21] L. A. Clarke, A. Podgurski, D. J. Richardson, and S. J. Zeil. A formal evaluation of data flow path selection criteria. *IEEE Transactions on Software Engineering*, 15(11):, Nov. 1989.

[22] L. A. Clarke and D. J. Richardson. *Symbolic evaluation methods - implementation and applications, in Computer Program Testing (B. Chandrasekaran and S. Radicchi, editors.* Elsevier North-Holland, NY, 1981.

[23] E. W. Dijkstra. The humble programmer. *Communications of ACM*, 15:859–866, October 1972.

[24] R. P. Dilworth. A decomposition theorem for partially ordered sets. *Annals Math.*, 51(1):161–166, Jan. 1950.

[25] J. W. Duran and S. C. Ntafos. An evaluation of random testing. *IEEE Transactions on Software Engineering*, SE-10(7):438–444, July 1984.

[26] J. C. Laprie (Ed.). *Dependability: basic concepts and associated terminology, dependable computing and fault-tolerant systems.* Springer-Verlag, 1991.

[27] N.E. Fenton and R.W. Whitty. Axiomatic approach to software metrication through program decomposition. *The Computer Journal*, 4(29):329–339, 1986.

[28] L.R. Ford and D.R. Fulkerson. *Flows in Networks.* Princeton University Press, NJ, 1962.

[29] P. G. Frankl and S. N. Weiss. An experimental comparison of the effectiveness of branch testing and data flow testing. *IEEE Transactions on Software Engineering*, 19(8):774–787, August 1993.

[30] P. G. Frankl and S. N. Weiss. An experimental comparison of the effectiveness of the all-uses and all-edges adequacy criteria. *Proc. Fourth Int. Symposium on Soft. Testing and Analysis*, 154–164, 1991.

[31] P. G. Frankl and E. J. Weyuker. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, 14(10):1483–1498, October 1988.

[32] P. G. Frankl and E. J. Weyuker. A formal analysis of the fault-detecting ability of testing methods. *IEEE Transactions on Software Engineering*, SE-19(3):202–213, March 1993.

[33] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering.* Prentice Hall, NJ, 1991.

[34] J. B. Goodenough and S. L. Gerhart. Towards a theory of test data selection. *IEEE Transactions on Software Engineering*, SE-1(2):156–173, June 1975.

[35] J. B. Goodenough and S. L. Gerhart. Towards a theory of testing: data selection criteria. In *Current trends in programming methodology, vol. 2, R. T. Yeh Ed. Prentice-Hall, Englewood Cliffs, N. J.*, pages 44–79, 1977.

[36] R. Gupta and M. L. Soffa. Employing static information in the generation of test cases. *Software Testing, Verification and Reliability*, 3(1):29–48, 1993.

[37] D. Hamlet. Partition testing does not inspire confidence. *IEEE Transactions on Software Engineering*, 15(11):1402–1411, Dec. 1990.

[38] D. Hamlet. Testing for probable correctness. *Proc. ACM SIGSOFT/IEEE Workshop on Software Testing, Banff, Canada*, 92–97, 1986.

[39] D. Hamlet. Theoretical comparison of testing methods. *Proc. ACM SIGSOFT Third Symposium on Software Testing, Analysis, and Verification (TAV3), Key West, Florida*, 28–37, 1989.

[40] M.J. Harrold, R. Gupta, and M. L. Soffa. A methodology for controlling the size of a test suite. *ACM Transactions on Software Engineering and Methodology*, 2(3):270–285, July 1993.

[41] M. S. Hecht. *Flow Analysis of Computer Programs.* North Holland, 1977.

[42] D. Hedley and M. A. Hennell. The causes and effects of infeasible paths in computer programs. In *Proc. of the 8th Int. Conf. on Software Engineering, Imperial College, London, UK*, pages 259–266, Aug. 1985.

[43] J. R. Horgan and S. A. London. Atac: a data flow coverage testing tool for c. In *Proc. of Symposium on Assessment of Quality Software Development Tools, New Orleans, LA*, pages 2–10, May 1992.

[44] W. E. Howden. *Functional Program Testing and Analysis.* McGraw-Hill, 1987.

[45] W. E. Howden. Methodology for the generation of program test data. *IEEE Transactions on Computers*, c-24(5), May 1975.

[46] W. E. Howden. Reliability of the path analysis testing strategy. *IEEE Transactions on Software Engineering*, 2():208–215, 1976.

[47] IEEE/ANSI. *IEEE Standard Glossary of Software Engineering Terminology.* IEEE Std 610.12-1990, 1990.

[48] I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard. *Object Oriented Software Engineering - A use case driven approach.* Addison-Wesley, NY, 1992.

[49] R. A. Kemmerer. Testing formal specifications to detect design errors. *IEEE Transactions on Software Engineering*, 11():32–43, 1985.

[50] B. W. Kernighan and D. M. Ritchie. *The C Programming Language.* Prentice-Hall, 1978.

[51] E.D. Knuth. *The Art of Computer Programming, Vol. 2.* Addison-Wesley, Reading, Massachusetts, 1981.

[52] K. W. Krause, R. W. Smith, and M. A. Goodwin. Optimal software test planning through automated network analysis. *IEEE Symposium on Computer Software Reliability*, 1973.

[53] J. W. Laski and B. Korel. A data flow oriented program testing strategy. *IEEE Transactions on Software Engineering*, SE-9(3):347–354, May 1983.

[54] S.S. Lavenberg. *Computer Performance Modelling Handbook.* Academic Press, New York, 1983.

[55] T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–141, 1979.

[56] M. Marré and A. Bertolino. Reducing and estimating the cost of test coverage criteria. *Proc. of the 18th International Conference on Software Engineering (ICSE 18), Berlin*, 486–494, March 1996.

[57] M. Marré and A. Bertolino. Unconstrained duas and their use in achieving all-uses coverage. *ACM Proc. of the Int. Symp. on Software Testing and Analysis (ISSTA 96), San Diego, CA, USA*, 147–157, January 1996.

[58] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE 2(4):308–320, December 1976.

[59] T. J. McCabe, editor. *Structured Testing.* Silver Spring, MD: IEEE Comp. Soc. Press, 1982.

[60] E. F. Miller. Software testing technology: an overview. *Handbook of Software Engineering (C. R. Vick and C. V. Ramamoorthy, eds.), Van Nostrand Reinhold Company*, 1984.

[61] E. F. Miller, M. R. Paige, J. P. Benson, and W. R. Wisehart. Structural techniques of program validation. In *Digest COMPCON74*, pages 161–164, 1974.

[62] J. D. Musa and A. F. Ackerman. Quantifying software validation: when to stop testing? *IEEE Software*, 19–27, May 1989.

[63] G. J. Myers. *The Art of Software Testing.* Wiley, New York, 1979.

[64] S. C. Ntafos. A comparison of some structural testing techniques. *IEEE Transactions on Software Engineering*, SE-14(6):868–874, June 1988.

[65] S. C. Ntafos. On required element testing. *IEEE Transactions on Software Engineering*, SE-10(6):795–803, Nov. 1984.

[66] S. C. Ntafos and S. L. Hakimi. On path cover problems in digraphs and applications to program testing. *IEEE Transactions on Software Engineering*, SE-5(5):520–529, Sep. 1979.

[67] L. J. Osterweil, L. D. Fosdick, and R. N. Taylor. *Error and anomaly diagnosis through data flow analysis, in Computer Program Testing (B. Chandrasekaran and S. Radicchi, editors.* Elsevier North-Holland, NY, 1981.

[68] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating functional tests. *Communications of ACM*, 31(3):676–686, June 1988.

[69] W. Perry. *Effective methods for software testing.* J. Wiley & Sons, 1995.

[70] E. Ploedereder. Pragmatic techniques for program analysis and verification. In *Proceedings of Fourth International Conference on Software Engineering, Munich, Germany*, September 1979.

[71] Plowowarski, Ohba, and Caruse. Coverage mesurement experience during function test. In *Proceedings of Fifteen International Conference on Software Engineering*, 1993.

[72] R. E. Prather. On hierarchical metrics. *Soft. Eng. J.*, 2(2):42–45, 1987.

[73] S. Rapps and E.J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, SE-11(4):367–375, April 1985.

[74] B. G. Ryder. Constructing the call graph of a program. *IEEE Transactions on Software Engineering*, SE-5:216–226, 1979.

[75] S. Sahni. *Concepts in Discrete Mathematics.* The Camelot Pu. Co., Second Edition, 1985.

[76] S. R. Schach. *Software Engineering.* Richard D. Irwin and Asken Associates, Second Edition, 1993.

[77] H. S. Wang, S.R. Hsu, and J.C. Lin. A generalized optimal path selection model for structural program testing. *The Journal of Systems and Software*, 10:55–63, 1989.

[78] E. Weyuker. Assessing test data adequacy through program inference. *ACM Transactions on programming languages and systems*, 5(4):641–655, October 1983.

[79] E. Weyuker. The cost of data flow testing: an empirical study. *IEEE Transactions on Software Engineering*, 16(2):121–128, 1990.

[80] E. Weyuker. More experience with data flow testing. *IEEE Transactions on Software Engineering*, 19(9):912–919, Sept. 1993.

[81] E. Weyuker. Translatability and decidability questions for restricted classes of program schemas. *SIAM Journal on Computers*, 8(4):587–598, 1979.

[82] E. Weyuker and B. Jeng. Analizing partition testing strategies. *IEEE Transactions on Software Engineering*, 17(7):703–711, July 1991.

[83] E. J. Weyuker. The complexity of data flow criteria for test data selection. *Information Processing Letters*, 19(2):103–109, August 1984.

[84] E. J. Weyuker and T. J. Ostrand. Theories of program testing and application of revealing subdomains. *IEEE Transactions on Software Engineering*, 6(3):236–246, May 1980.

[85] E. J. Weyuker, S. N. Weiss, and D. Hamlet. Comparison of program testing strategies. In *Proc. ACM SIGSOFT Fourth Symposium on Software Testing, Analysis, and Verification (TAV4)*, pages 1–10, 1991.

[86] L. J. White. General overview of software testing. In *Software Testing: Metodi e Tecniche, Scuola Estiva, Taranto, Italy*, pages 5–28, June 1993.

[87] L. J. White. Regression testing. In *Software Testing: Metodi e Tecniche, Scuola Estiva, Taranto, Italy*, pages 91–102, June 1993.

[88] C. Wilson and L. J. Osterweil. A data flow analysis tool for the c programming language. In *Proc. of Sixth Computer Software and Applications Conference*, Nov. 1982.

[89] W. E. Wong. Effect of test size and block coverage on fault detection effectiveness. In *Proc. of the 5th Int. Symp. on Software Reliability Engineering*, pages 230–238, 1994.

[90] W. E. Wong. *On Mutation and Data Flow A Thesis.* PhD thesis, SERC-TR-149-P, Software Eng. Research Center, Purdue University, December 1993.

[91] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur. Effect of test set minimization on fault detection effectiveness. In *Proc. of the 17th Int. Conf. on Software Engineering (ICSE 17), Seattle, WA, USA*, pages 41–50, April 1995.

[92] M. R. Woodward, D. Hedley, and M. A. Hennell. Experience with path analysis and testing of programs. *IEEE Transactions on Soft. Engineering*, SE-6(3), May 1980.

[93] D. F. Yates and N. Malevris. Reducing the effects of infeasible paths in branch testing. *ACM SIGSOFT Software Engineering Notes*, 14(8):48–54, Dec. 1989.