Mejoras al *Testing* Exhaustivo Acotado

por **Valeria Bengolea**

Presentado ante la Facultad de Matemática, Astronomía y Física como parte de los requerimientos para la obtención del grado de Doctor en Ciencias de la Conputación de la

UNIVERSIDAD NACIONAL DE CORDOBA

Marzo, 2015

© FaMAF - UNC 2015

Director: Nazareno Matías Aguirre

Resumen

El *testing* consiste en ejecutar una pieza de código, cuya correctitud se desea evaluar, utilizando diversas entradas y examinar si el resultado obtenido en cada ejecución es el esperado. Esta actividad es muy importante en el desarrollo de *software*, la cual se estima que ocupa más de la mitad del costo -en tiempo y recursos humanos- total del desarrollo de *software* [19], por lo que es fundamental, siempre que sea posible, su automatización. En particular, la construcción automática de entradas para probar el *software* bajo análisis reduce significativamente los costos de hacer *testing*.

El *testing* exhaustivo acotado es una técnica muy efectiva para encontrar errores en el código [12, 36, 27] que consiste en generar automáticamente *todas* las entradas válidas dentro de ciertas cotas pre-establecidas, y posteriormente usar estas entradas para probar el código bajo análisis, con el objetivo de encontrar errores en él. La efectividad de los conjuntos de casos de *test* (*suites de test*) producidos mediante esta técnica, crece a medida que las cotas crecen, pero además, el tiempo consumido durante la generación y posterior ejecución de las *suites de test*, se incrementa de manera exponencial cuando las cotas crecen, y como consecuencia, muchas veces, esta tarea resulta inviable [19].

En este trabajo se presentan un grupo de técnicas destinadas a disminuir el tiempo empleado para realizar *testing exhaustivo acotado*, para lo cual, se propone reducir el tiempo de generación de las *suites de test*, así como también el tiempo de ejecución de las mismas mediante la reducción 'adecuada' de su tamaño. Para tal fin, se presentan tres enfoques. El primero de ellos consiste en incorporar un criterio de cobertura de código durante el proceso de búsqueda de instancias válidas, y utilizar la información que éste provee para evitar la exploración de porciones del espacio de búsqueda. Como consecuencia se obtienen -reduciendo el tiempo de generación- *suites de test* más pequeñas con la misma cobertura, con respecto al criterio seleccionado, que la *suite* exhaustiva acotada.

El segundo enfoque, consiste en explotar la información que brinda la especificación de la rutina bajo análisis, en particular la descripción de entradas válidas, la cual es una precondición implicita que las entradas deben satisfacer. Esta descripción de las entradas válidas, la cual en el contexto de la programación orientada a

objetos se corresponde con el invariante de representación de una clase, es utilizada en dos sentidos (lo cual da lugar a dos nuevas técnicas):

- Para obtener invariantes de representación separados caracterizando partes disjuntas de las entradas, lo cual es aprovechado en el proceso de generación para construir estas porciones de manera independiente y luego combinarlas para obtener la entrada de *test* completa. Esto permite reducir notablemente el tiempo de generación de *suites de test*, para los casos en los cuales las entradas están compuestas por porciones separadas del *heap*.

- Para definir un criterio de cobertura de caja negra, que permita particionar las entradas válidas en subdominios, de acuerdo a como estas entradas 'ejercitan' la implementación operativa del invariante de representación. Estas equivalencias entre casos de *test* son aprovechadas para descartar entradas de la *suite* y de esta manera reducir su tamaño sin perder clases de entradas viables.

El tercer y último enfoque, consiste en incorporar *cotas ajustadas* [14], las cuales son calculadas desde el invariante de representación, al proceso de generación exhaustiva acotada. Las *cotas ajustadas* brindan información útil que permite restringir el conjunto de valores que cada campo, perteneciente a la entrada, puede tomar. Esta información es utilizada en el proceso de generación para reducir el espacio de entradas posibles.

En los siguientes capítulos, estos enfoques se presentan en detalle junto con casos de estudio, en el contexto de estructuras de datos complejas alojadas en memoria dinámica, que muestran los beneficios de incorporar su uso en el *testing* exhaustivo acotado.

Palabras Claves: Generación Automática de Casos de Test, Testing Exhaustivo Acotado, Criterios de Cobertura, Invariante de Representación.

PACS: D.2.5

Resumen

Testing consists of executing a piece of software, whose correctness needs to be assessed, in a number of different test cases to check whether the obtained result corresponds with the expected one. This activity is very important in *Software Engineering* estimated to take more than half of the total software development cost [19], for this reason is fundamental, when possible, made it automatic. In Particular, the automatic generation of inputs to test the software under analysis reduce significantly the cost to do testing. Bounded exhaustive testing is a very effective technique to find bugs in code [12, 36, 27] that consists of generating automatically all valid inputs within certain given scope and subsequently using these inputs to test the code under analysis, with the goal of finding bugs on it. The effectiveness of the suites test produced by this technique increases as one increases the scope for the bounded exhaustive generation, also the time for test generation and the time for test execution grow exponentially with respect to the scope. As a consequence, in many cases, bounded exhaustive testing become impracticable [19].

In this thesis, a set of techniques are proposed, to overcome the above described problems. These techniques aim at reducing the time for bounded exhaustive testing, by either reducing the test generation time, or adequately reducing the obtained bounded exhaustive suites. For this purpose, three approaches are introduced. The first one consists of incorporating a coverage criterion during the search process of valid instance, and using the provided information for this criterion to avoid the exploration of portions of the search space. As a consequence we obtain- reducing generation time- test suites smaller with the same coverage, with respect to the selected criterion, that the bounded exhaustive suite.

The second approach, take advantage of the information given by the specification of the routine under test, in particular the description of valid inputs, which is a implicit precondition that input must satisfy. This description, which in the context of oriented object programming correspond to the representation invariant of a class, is used in two different senses:

- to factor out separate representation invariants for disjoint substructures of the inputs, which are used during the generation process to build these disjoint portions independently and then combine them to obtain the whole test input. This approach allow us to reduce consBÂžiderably the generation time of test suites, in the cases that inputs are composed by separate heap portions.

- to define a black box coverage criterion, that allow us to split the valid inputs into sub-domains, according to how these exercise the operative implementation of the representation invariant. These equivalences between test cases are used to discard inputs from the suite and, in this way, reduce its size without loosing viable inputs classes.

The third, and last approach, consist of incorporating *tight bounds* [14], which are computed from the representation invariant, into the bounded exhaustive generation process. The *tight bounds* provide some useful information that allow to constraint the set of value that each field, of the inputs, may take. This information is used in the generation process to reduce the possible inputs.

In the following chapters, these approaches are presented in details together with some case studies, in the context of complex heap data structures, that show the advantage of incorporating them in the use of bounded exhaustive testing.

Keywords: Automatic Generation of inputs, Bounded Exhaustive Testing, Coverage Criteria, Representation Invariant.

Agradecimientos

En primer lugar quiero agradecer a Nazareno Aguirre, mi director, su apoyo ha sido de fundamental importancia para culminar este trabajo. Nazareno ha contribuido de manera significativa en mi formación profesional. También quiero agradecer a Marcelo Frias, mi co-director.

Además, agradecer al Departamento de Computación de la Universidad Nacional de Río Cuarto, el cual, me brindó el lugar de trabajo para desarrollar este trabajo, sin dejar de mencionar a mis compañeros de trabajo/oficina con quienes tuve la oportunidad de trabajar en diferentes ámbitos.

Por último, quiero agradecer a Pablo Castro, sin su constante e incondicional apoyo, este trabajo no sería posible.

Dedicatoria

A Pablo, Bruno y Mateo.

Índice general

1. Introducción **2**

2. Conceptos Preliminares **10**
2.1. Testing 10
2.2. Testing Exhaustivo acotado 10
2.2.1. Generación (automática) exhaustiva acotada de entradas . . . 11
2.3. Invariante de Representación 15
2.4. El Algoritmo de `Korat` 16
2.5. Criterios de Test 22
2.5.1. Criterios de Cobertura de Caja Blanca 22
2.5.2. Criterios de Cobertura de Caja Negra 25
2.5.3. Testing de Mutación 26

3. Incorporación de Criterios de cobertura en la generación exhaustiva acotada **30**
3.1. Intuición del enfoque 30
3.2. Ejemplo motivador 32
3.3. Implementación de la técnica 34
3.4. Sobre la consistencia de esta implementación 39

4. Uso del invariante de representación para reducir el testing exhaustivo acotado **44**
4.1. Motivación e Intuición 45
4.2. Generación independiente de sub-estructuras disjuntas. 48
4.2.1. Un caso testigo: `NodeCaching LinkedList` 49
4.2.2. Descripción precisa del enfoque 50
4.2.3. Como determinar sub-estructuras diyuntas 52

4.2.4. Descomposición del `repOk()` 53
4.2.5. Sobre la correctitud de esta técnica 56
4.3. Reducción de *suites de test* exhaustivas acotadas usando criterios sobre `repOk` 57
4.3.1. Ejemplo guía: *Árboles Binarios* 57
4.3.2. Definición de un criterio de caja negra basado en `repOk()` . . 58
4.3.3. Mecanismo de Reducción 60
4.3.4. Sobre la correctitud de esta técnica 61

5. Uso de *cotas ajustadas* en la generación exhaustiva acotada 64
5.1. ¿Qué son las cotas Ajustadas? 64
5.2. Motivación 67
5.3. Incorporación *cotas ajustadas* en la generación exhaustiva acotada . . 69
5.3.1. Refinamiento de Cotas 72
5.3.2. Acerca de la correctitud de esta técnica 73
5.3.3. Limitaciones del Enfoque 75
5.4. Generación exhaustiva acotada desde especificaciones híbridas 75

6. Evaluación experimental 80
6.1. Incorporación de Criterios de cobertura en la generación exhaustiva acotada 81
6.1.1. Evaluación experimental utilizando criterios de cobertura de caja negra 83
6.1.2. Evaluación experimental utilizando criterios de cobertura de caja blanca 94
6.2. Reducciones al testing exhaustivo acotado basadas en el uso del invariante de representación 98
6.2.1. Generación independiente de sub-estructuras disjuntas 98
6.2.2. Reducción de *suites de tests* exhaustivas acotadas usando criterios sobre `repOk()` 105
6.3. Uso de *cotas ajustadas* en la generación exhaustiva acotada 121
6.4. Amenazas a la Válidez 125
6.5. Análisis de resultados obtenidos 127

7. Conclusiones y Trabajos Futuros 134

Índice de figuras

2.1. Grafo de Flujo de Control del ejemplo 2.5.3 28

4.1. Instancia mal formada de `NodeCachingLinkedList`. 50
4.2. Ejemplo de Árboles Binarios. El primero y el segundo ejercitan `repOK` de la misma manera, de acuerdo a cobertura de decisión. 59

5.1. Cotas ajustadas (BFS) para árboles binarios de búsqueda de hasta 4 nodos 70
5.2. Algoritmo de KoratCotas 77
5.3. Algoritmo de KoratCotas 78
5.4. Algoritmo de KoratCotas 79

6.1. Tiempos de corrida de `Korat` y `Korat+` para `listAsSet` 85
6.2. Tiempos de Corrida de `Korat` y `Korat+` para *Binomial Heaps (Merge)* 88
6.3. Tiempos de corrida de `Korat` y `Korat+` para *Grafos Dirigidos* 89
6.4. Tiempos de corrida de `Korat` y `Korat+` para *Grafos Dirigidos Etiquetados* 91
6.5. Tiempos de corrida de `Korat` y `Korat+` para *Árboles Binarios de Búsqueda (Delete)* 93
6.6. Tiempos de corrida de `Korat` y `Korat+` para *Árboles Rojos y Negros (rutina `Insert`)* 97
6.7. Tiempos de corrida de `Korat` y `Korat+` para *Árboles Rojos y Negros (rutina `find`)* 98
6.8. Tiempos de corrida de `Korat` y `Korat+` para *Árboles binarios de búsqueda (rutina `insert`)* 100
6.9. Tiempos de corrida de `Korat` y `Korat+` para *Árboles binarios de búsqueda (seach)* 101

6.10. Comparación generación exhaustiva acotada y generación separada acotada de sub-estructuras disjuntas para `NodeCachingLinkedList`, cuando el número de nodos se incrementa. (representatión gráfica). . 105
6.11. Comparación generación exhaustiva acotada y generación separada acotada de sub-estructuras disjuntas para `NodeCachingLinkedList`, cuando el número de claves permitidas se incrementa. (representatión gráfica). 106
6.12. Comparación generación exhaustiva acotada y generación separada acotada de sub-estructuras disjuntas para `Binomial Heap`(representación gráfica). 107
6.13. Comparación generación exhaustiva acotada y generación separada acotada de sub-estructuras disjuntas para pares compuestos por un árbol ordenado balanceado (AVL tree) y una lista simplemente encadenada, cuando el número de nodos en estas estructuras se incrementa. (representación gráfica). 107
6.14. Comparación generación exhaustiva acotada y generación separada acotada de sub-estructuras disjuntas para pares compuestos por un árbol ordenado balanceado (AVL tree) y una lista simplemente encadenada, cuando el número de claves permitidas en estas estructuras se incrementa. (representación gráfica). 108

Índice de cuadros

3.1. Evaluación `Korat (K)` y `Korat+ (K+)` para `ListAsSet`. Se muestra, en cada caso, espacio de búsqueda y entre paréntesis el tamaño de la *suite*. 39

6.1. Comparación `Korat` y `Korat+` para `listAsSet` 84
6.2. Tiempos de corrida de `Korat` y `Korat+` para `listAsSet` 84
6.3. Efectividad, con respecto a testing de mutación, de las suites de tests generadas por `Korat` y `Korat+` sobre la rutina `listToSet` (se reportan mutantes vivos). 86
6.4. Comparación `Korat` y `Korat+` para *Binomial Heap (Merge)* 87
6.5. Tiempo de Corrida de `Korat` y `Korat+` para *Binomial Heap (Merge)* . 87
6.6. Efectividad, con respecto a testing de mutación, de las suites de tests generadas por `Korat` y `Korat+` sobre la rutina `merge` de `binomilaHeap` (se reportan mutantes vivos). 88
6.7. Comparación `Korat` y `Korat+` para *Grafos Dirigidos* 89
6.8. Tiempos de corrida de `Korat` y `Korat+` para *Grafos Dirigidos* 89
6.9. Comparación `Korat` y `Korat+` para *Grafos Dirigido Etiquetados* . . . 91
6.10. Tiempos de corrida de `Korat` y `Korat+` para *Grafos Dirigidos Etiquetados* 91
6.11. Comparación `Korat` y `Korat+` para *Árboles Binarios de Búsqueda (Delete)* 92
6.12. Tiempos de corrida de `Korat` y `Korat+` para *Árboles Binarios de Búsqueda (Delete)* 93
6.13. Efectividad, con respecto a testing de mutación, de las suites de tests generadas por `Korat` y `Korat+` sobre la rutina `delete` de Árboles binarios de búsqueda (se reportan mutantes vivos). 94
6.14. Comparación `Korat` y `Korat+` para *listas simplemente encadenadas* . 95

6.15. Comparación `Korat` y `Korat+` para *Árboles Rojos y Negros (rutina* ***Insert****)* 96
6.16. Tiempos de corrida de `Korat` y `Korat+` para *Árboles Rojos y Negros (rutina* ***Insert****)* 96
6.17. Comparación `Korat` y `Korat+` para *Árboles Rojos y Negros (rutina* ***find****)* 97
6.18. Tiempos de corrida de `Korat` y `Korat+` para *Árboles Rojos y Negros (rutina* ***find****)* 98
6.19. Comparación `Korat` y `Korat+` para *Árboles binarios de búsqueda(rutina* ***insert****)* 99
6.20. Tiempos de corrida de `Korat` y `Korat+` para *Árboles binarios de búsqueda (rutina* ***insert****)* 99
6.21. Comparación `Korat` y `Korat+` para *Árboles binarios de búsqueda (rutina* ***search****)* 100
6.22. Tiempos de corrida de `Korat` y `Korat+` para *Árboles binarios de búsqueda (rutina* ***search****)* 101
6.23. Comparación generación exhaustiva acotada y generación separada acotada de sub-estructuras disjuntas para `NodeCachingLinkedList`, cuando el número de nodos se incrementa. 103
6.24. Comparación generación exhaustiva acotada y generación separada acotada de sub-estructuras disjuntas para `NodeCachingLinkedList`, cuando el número de claves permitidas se incrementa. 104
6.25. Comparación generación exhaustiva acotada y generación separada acotada de sub-estructuras disjuntas para `Binomial Heaps`. 109
6.26. Comparación generación exhaustiva acotada y generación separada acotada de sub-estructuras disjuntas para pares compuestos por un árbol ordenado balanceado (`AVL tree`) y una lista simplemente encadenada, cuando el número de nodos en estas estructuras se incrementa. 109
6.27. Comparación generación exhaustiva acotada y generación separada acotada de sub-estructuras disjuntas para pares compuestos por un árbol ordenado balanceado (`AVL tree`) y una lista simplemente encadenada, cuando el número de claves permitidas en estas estructuras se incrementa. 110
6.28. Tamaño de las *suites* exhaustivas acotadas y las reducciones basadas en `repOk()`, para probar la rutina `merge` de binomial heaps. 112

6.29. Efectividad, con respecto a *testing de mutación*, de las *suites de tests* exhaustivas y de las *suites de tests* reducidas sobre la rutina `merge` (se reportan mutantes vivos). 113
6.30. Tamaño de las *suites* exhaustivas acotadas y las reducciones basadas en `repOK` para probar las rutinas `insert`, `delete` y `find` de binomial heaps. 114
6.31. Efectividad, con respecto a *testing de mutación*, de las *suites de tests* exhaustivas y de las *suites de tests* reducidas sobre las rutinas `insert`, `delete` y `find` de binomial heaps (se reportan mutantes vivos para cada caso). 115
6.32. Tamaño de las *suites* exhaustivas acotadas y las reducciones basadas en `repOk()` para probar la rutina `isPalindromic` sobre listas doblemente encadenadas. 116
6.33. Efectividad, con respecto a *testing de mutación*, de las *suites de tests* exhaustivas y de las *suites de tests* reducidas sobre la rutina `isPalindromic` de Listas doblemente encadenadas (se reportan mutantes vivos para cada caso). 117
6.34. Tamaño de las *suites* exhaustivas acotadas y las reducciones basadas en `repOK` para probar las rutinas `delete`, `insert` y `search` sobre árboles binarios de búsqueda 117
6.35. Efectividad, con respecto a *testing de mutación*, de las *suites de tests* exhaustivas y de las *suites de tests* reducidas sobre la rutina `insert`, `delete` y `search` de árboles binarios de búsqueda (se reportan mutantes vivos para cada caso). 118
6.36. Tamaño de las *suites* exhaustivas acotadas y las reducciones basadas en `repOK` para probar las rutinas `add`, `remove` y `contains` sobre árboles rojos y negros. 119
6.37. Efectividad, con respecto a *testing de mutación*, de las *suites de tests* exhaustivas y de las *suites de tests* reducidas sobre las rutinas `add`, `remove` y `contains` de árboles rojos y negros (se reportan mutantes vivos para cada caso). 120
6.38. Comparación `Korat` vs `KoratCotas` para *árboles Rojos y Negros* utilizando `RepOk` de la distribución de `Korat` 122
6.39. Comparación `Korat` vs `KoratCotas` para *árboles Rojos y Negros* utilizando un `RepOk` alternativo 123
6.40. Comparación `Korat` vs `KoratCotas` para *Árboles binarios de búsqueda* utilizando `RepOk` de la distribución de `Korat` 124

6.41. Comparación `Korat` vs `KoratCotas` para *Árboles binarios de búsqueda* utilizando un `RepOk` alternativo 124
6.42. Comparación `Korat` vs `KoratCotas` para *Binomial Heap* utilizando `RepOk` de la distribución de `Korat` 125
6.43. Comparación `Korat` vs `KoratCotas` para *Binomial Heap* utilizando un `RepOk` alternativo 126

Capítulo 1

Introducción

En los inicios de la informática, las computadoras fueron utilizadas para resolver problemas precisos, principalmente ecuaciones, o problemas físico-matemáticos puntuales. Generalmente los mismos usuarios eran los responsables de codificar las soluciones. Con la disminución de los costos de *hardware*, y su crecimiento en poder computacional, las computadoras se volvieron más accesibles a la comunidad en general. La vida cotidiana se vio impregnada de sistemas de computación que realizaban tareas de las más variadas. En la actualidad, estos se involucran en la mayoría de los aspectos de la sociedad. Ejemplos de esto se pueden encontrar en diferentes ámbitos: telefonía celular, medicina, sistemas de aviónica, etc. En general, estos sistemas cuentan con cientos de líneas de código y realizan, en muchos casos, tareas consideradas críticas: de su buen funcionamiento dependen millones de dólares y/o vidas humanas. Son bien conocidos algunos errores de seriedad en *software* de sistemas críticos: *Therac-25* [22], *Arianne 5* [13], por nombrar sólo algunos. La *therac-25* es una máquina de radioterapia que, debido a un error de código, sobredosificó a por lo menos seis personas, de las cuales tres murieron al recibir cien veces la radiación esperada. En el caso del cohete espacial *Arianne 5*, un error en el *software* de control de navegación produjo la explosión del mismo, con costos que se estiman en millones de dólares.

En este contexto, la *Ingeniería de Software* surge desde la necesidad de contar con metodologías que permitan la construcción de *software* complejo de manera sistemática y garantizando altos niveles de calidad, donde calidad de software abarca una amplia variedad de atributos que son deseables en un sistema: mantenibilidad, reusabilidad, legibilidad, por nombrar algunos, son atributos deseables desde el punto de vista del desarrollador. Por otro lado, los atributos externos: correctitud,

confiabilidad, robustez, usabilidad, performance, etc, son aquellas propiedades que finalmente -aunque no más importantes que las anteriores- son visibles por los usuarios del producto. Los términos correctitud, confiabilidad y robustez caracterizan aquellas propiedad funcionales del sistema: *el software realiza su función de acuerdo lo esperado.*

Un programa es *funcionalmente* correcto si se comporta de acuerdo a su especificación, es decir, si éste cumple correctamente con el propósito para el cual fue construido [16].

Existen varios enfoques para asegurar calidad funcional de *software*. Dentro de estos enfoques se podrían distinguir dos ramas, o familias, de técnicas: aquellas basadas en el análisis estático del sistema, es decir, que no requieren de la ejecución del mismo durante el análisis. Por otro lado, se encuentran las técnicas de análisis dinámico, para las cuales si se requiere de la ejecución del sistema. Dentro de la primera clasificación caen técnicas tales como el *model checking* y la *ejecución simbólica*, mientras que técnicas como el análisis de rendimiento (*profiling* de recursos) y el *testing* pertenecen a las técnicas de análisis dinámico [16].

El *testing* juega actualemente, un rol muy importante a la hora de garantizar correctitud de un sistema, ya que a pesar de su incompletitud, es un mecanismo muy efectivo -y ampliamente usado- en verificación de software. Esencialmente, esta actividad consiste en ejecutar una pieza de código, cuya calidad se desea evaluar, utilizando diversos casos de *test* y examinar si el resultado obtenido en cada ejecución es el esperado. Estos casos de *test* instancian los parámetros del programa bajo análisis con diferentes entradas. Como es de esperar, en general, resulta imposible testear una pieza de *software* exhaustivamente (para toda posible entrada), ya que en muchos casos se admite un conjunto infinito, o finito pero muy grande, como posibles entradas válidas. La famosa frase de *Dijkstra "El testing puede ser usado para mostrar la presencia de errores pero nunca su ausencia"*, hace alusión a este hecho. Debido a esto, es necesario contar con un conjunto de casos de *test* que provea suficiente confianza de que el programa bajo análisis realiza lo esperado, incluso para las entradas que no están siendo testeadas. En otros palabras, estos casos de *test*, para ser considerados adecuados (útiles para detectar errores), deben evaluar el *software* bajo distintos escenarios. En la actualidad, existen varios criterios de *test*, los cuales fundamentalmente definen cuales son las diferentes circunstancias que un conjunto de casos de *test* debería ejercitar o contemplar [16]. La principal clasificación de criterios de *tests* los divide en dos grupos: criterios de caja blanca, o estructurales, y criterios de caja negra, o basados en especificación. Corresponden a la primera clasificación aquellos criterios que dependen de la estructura interna

del código evaluado, es decir, la evaluación del grado de cobertura de un conjunto de *test* depende de la lógica del programa. Por otro lado, los llamados criterios de caja negra no asumen conocimiento acerca de la estructura interna del *software* bajo análisis; en su lugar, hacen uso de la especificación del mismo para definir el grado de cobertura que un conjunto de *tests* tiene sobre el código evaluado [28].

La elección de un criterio de cobertura u otro, influye en los casos de *test* que se deben construir al hacer *testing*. Esta construcción de entradas, denominada generación de casos de *test*, es generalmente una actividad compleja, en la cual el ingeniero a cargo tiene que crear entradas que satisfagan, en muchos casos, diversas restricciones complejas, y que además ejerciten el *software* bajo diferentes circunstancias o escenarios. Estas entradas muchas veces son generadas de manera manual, requiriendo gran cantidad de tiempo y recursos humanos abocados a esta actividad, por lo que la automatización de esta tarea es fundamental para reducir los costos de hacer *testing* (y del desarrollo de *software* en general, ya que se estima que el *testing* ocupa más de la mitad de los costos del desarrollo total de un sistema [19]). En los últimos años, se han desarrollado varias técnicas y herramientas para automatizar el proceso de generación de entradas, lo cual es particularmente desafiante cuando se trata de entradas de *test* estructuralmente complejas, las cuales deben satisfacer propiedades no triviales para ser consideradas bien formadas o válidas. Ejemplo de esto son las estructuras de datos alojadas en el *heap* de memoria (por ejemplo, árboles binarios, listas,etc). Algunas herramientas [11, 39, 21, 26, 17] abordan el problema de la generación automática de *tests* exitosamente, utilizando lo que se conoce como *exploración exhaustiva acotada* de casos de *test*. Más precisamente, esta técnica consiste en, dada una descripción de las entradas válidas, generar todas las entradas dentro de ciertos límites (máximo número de objetos de cada clase involucrada en la entrada), que satisfacen algunas restricciones o reglas de buena formación dadas. Para ello se utilizan técnicas de *constraint solving*, tal como *model checking* y otros mecanismos relacionados con *búsqueda*.

El *testing exhaustivo acotado* involucra no sólo la generación exhaustiva acotada de entradas, sino también el posterior uso de esas entradas para probar una pieza de *software* dada. Esta técnica es, en la actualidad, muy usada no sólo en la academia sino también en la industria, para probar aplicaciones reales, con muy buenos resultados en ciertos escenarios [12, 27]. Algunos años atrás, esta actividad era casi impensable; varias razones argumentan los avances en este campo: en principio, el *hardware* ha experimentado un notable crecimiento en poder computacional y una baja en los costos, lo que hace mucho más accesible esta actividad. Además, los notables avances en otras técnicas tales como el *model checking*, análisis de satis-

facibilidad booleana, entre otras, la cuales dan soporte a la generación exhaustiva acotada, permiten un importante aumento en la escalabilidad de esta técnica. Por otro lado, probar el *software* para toda entrada, dentro de ciertas cotas, da la garantía que ningún caso límite se escapa [19]. Además, esta técnica se respalda en la denominada *hipótesis de la cota pequeña* [18], la cual conjetura que, si un programa tiene errores, la mayoría de esos errores pueden ser reproducidos usando entradas pequeñas.

Asimismo, la efectividad de esta técnica, con respecto a la capacidad de detectar errores en el código, crece a medida que las cotas utilizadas en la generación se incrementan. Es decir, si bien la mencionada *hipótesis de la cota pequeña* [18] da cuenta de la utilidad de esta técnica, hay situaciones en las cuales es necesario probar el programa con entradas relativamente grandes para obtener una buena cobertura y detectar errores. Por ejemplo, si el programa bajo prueba corresponde a alguna rutina de inserción/eliminación sobre árboles balanceados, se necesitarán como entrada estructuras lo suficientemente grandes para forzar rotaciones o ejercitar los mecanismos de re-balanceo, por lo que si se desea cubrir estos casos para encontrar posibles errores en la implementación, se deben considerar cotas grandes. En otras palabras, en muchos casos, podría ser necesario considerar alguna cota particular para el *testing exhaustivo acotado* con el objetivo de cubrir cierta clase de entradas de interés. Debido a que el tiempo de generación de entradas crece exponencialmente a medida que las cotas crecen, esto podría requerir un tiempo excesivo para la generación de las entradas; más aún, inclusive cuando el proceso de generación de esta *suite de tests* se completará, la *suite* obtenida podría ser tan grande que el tiempo requerido para ejecutarla, y examinar, para cada caso, si el resultado obtenido se corresponde con el esperado, sería prohibitivo. Por ejemplo, si se considera la rutina `merge` sobre *binomial heaps*, cuya entrada corresponde a un par de *binomial heaps*, la *suite de tests* exhaustiva acotada con cota 7 (cantidad máxima de nodos para cada *binomial heap*) toma más de 15 horas para ser generada en una computadora moderna, y tiene 11.538.197.056 casos de *tests*.

Con la intención de hacer del *testing exhaustivo acotado* una técnica viable, en los últimos años se han implementado diferentes mecanismos. En ciertos casos éstos evitan construir entradas redundantes y visitar parte del espacio de estados de búsqueda correspondiente a entradas inválidas. Un ejemplo de esto es la herramienta `Korat` que implementa un mecanismo de rotura de simetrías junto con un mecanismo para evitar la generación de entradas inválidas basada en una técnica de poda sofisticada; `TestEra` usa `Alloy` (junto con sus mecanismos de rotura de simetrías y optimización) para mejorar la generación; `UDITA` implementa un nove-

doso mecanismo de evaluación *lazy*, el cual en combinación con ejecución simbólica mejora sustancialmente el proceso de generación de casos *test*. Aún así, en muchos casos, el *testing exhaustivo acotado* resulta impracticable.

Con este escenario, la utilidad de contar con técnicas que permitan mejorar el tiempo en la generación exhaustiva acotada, así como también el tiempo para ejecutar cierta pieza de código con las *suites de tests* generadas de dicha manera, es evidente: obtener una mayor escalabilidad en el *testing exhaustivo acotado*, es decir, es fundamental investigar nuevos enfoques/técnicas que permitan reducir el tiempo de generación de *suites de tests* y el tiempo de ejecución de la mismas, siendo ésta la motivación central del presente trabajo. El primer objetivo planteado, abarca el estudio de técnicas que permitan achicar el espacio de búsqueda, en otras palabras, se desea evitar examinar candidatos del espacio de búsqueda que de antemano, se podría 'predecir' que son inválidos. El segundo de los objetivos tiene que ver con la obtención de *suites de tests* más pequeñas - pero igual de efectivas - que las *suites* exhaustivas acotadas, de tal manera de producir un impacto en el tiempo de ejecución de las mismas. Los siguientes supuestos guían la búsqueda de estas técnicas:

- Se cuenta con una especificación, la cual incluye una descripción de las entradas válidas, de la rutina bajo análisis,
- Se cuenta con criterios de coberturas de *testing* para ser utilizados sobre esta rutina.

De esta manera, las principales contribuciones de este trabajo consisten en tres enfoques para mejorar el *testing exhaustivo acotado* en los dos sentidos mencionados anteriormente. El primero de ellos, a grandes rasgos, se basa en guiar la búsqueda de entradas válidas considerando algún criterio de cobertura de código dado sobre la rutina bajo análisis. Este criterio es incorporado en el mecanismo de *constraint solving*, de tal manera que la exploración de posibles casos de *test* es reducida sin perder clases de entradas viables, correspondientes al criterio dado. En otras palabras, este enfoque apunta a emplear un criterio de *test* con el propósito de "podar" la generación de casos de *test*, obteniendo una cobertura exhaustiva acotada de clases de equivalencia, asociadas con el criterio dado. El segundo enfoque estudiado, tiene como propósito buscar técnicas que no requieran información extra del usuario - más que la que ya es requerida por la generación exhaustiva acotada-, como es el caso del enfoque mencionado anteriormente, siendo esta una de las motivaciones del uso de la información que brinda la especificación de la rutina bajo análisis. La idea es

explotar esta información, en particular la descripción de entradas válidas, la cual es una precondición implicita que las entradas deben satisfacer. Esta descripción de las entradas válidas, la cual en el contexto de la programación orientada a objetos se corresponde con el invariante de representación de una clase, es utilizada de dos formas diferentes (lo cual da lugar a dos nuevas técnicas):

- Para caracterizar de manera separada partes disjuntas de las entradas, lo cual es aprovechado en el proceso de generación para construir estas porciones de manera independiente y luego combinarlas para obtener la entrada de *test* completa. Esto permite reducir notablemente el tiempo de generación de *suites de tests*, para los casos en los cuales las entradas están compuestas por partes independientes alojadas en porciones disjuntas del *heap*. En otras palabras, el invariante de representación sobre la cual las entradas son generadas, es utilizada para obtener invariantes de representación separados que caractericen las porciones de la entrada que son disjuntas, y que por lo tanto podrían ser generadas por separado para luego ser combinadas. Este proceso, a diferencia del tradicional, tiene varias ventajas. En principio este enfoque da lugar a explotar la paralelización de la generación (partes disjuntas de la entrada se generan de forma paralela). En segundo lugar, incluso cuando estas entradas son generadas de manera secuencial, se evita el problema de considerar entradas inválidas compuestas por porciones bien formadas (satisfacen el invariante de representación) en combinación con partes mal formadas de la entrada.

- Para definir un criterio de cobertura de caja negra, que permita particionar las entradas válidas en clases de equivalencia, de acuerdo a como estas entradas 'ejercitan' la implementación imperativa del invariante de representación. Estas equivalencias entre casos de *test* son aprovechadas para descartar entradas de la *suite* que son equivalentes a alguna otra ya presente en la *suite*, y de esta manera reducir el tamaño de las *suites de tests* sin afectar su efectividad (en términos de su capacidad para detectar errores).

El tercer enfoque estudiado, es también basado en el uso de la información que brinda el invariante de representación, la cual, a diferencia del enfoque anterior, es usada indirectamente. Este enfoque está basado en el uso de *cotas ajustadas* [14], las cuales son calculadas desde el invariante de representación, para podar el espacio de búsqueda durante la generación exhaustiva acotada. Las cotas ajustadas, restringen las entradas excluyendo, del conjunto de valores que cada campo puede tomar, casos que son inviables debido a que producirían estructuras simétricas o no cumpliarían

con la especificación de las entradas válidas. Estas restricciones son aprovechadas en el proceso de generación para eludir instancias inválidas, y de esta manera, reducir el espacio explorado.

Los enfoques propuestos se evaluan en diferentes casos de estudios que involucran varias estructuras complejas alojadas en memoria dinámica en Java y para varios criterios de cobertura. Los resultados experimentales se muestran en el contexto de `Korat`, una herramienta de *testing* exhaustivo acotado basada en *constraint solving*.

Los resultados de la evaluación experimental realizada permiten concluir que:

- Los criterios de cobertura son de gran utilidad, para reducir *suites de tests* exhaustivas acotadas y obtener *suites* 'intermedias" (medidas en término de su efectividad), si se considera, por un lado las *suites de tests* exhaustivas acotadas, y por el otro las *suites de tests* con cobertura por clase de equivalencia "óptima", es decir, aquellas que contienen un caso de *test* por cada clase de equivalencia cubierta por el criterio de cobertura subyacente.

- La especificación de la rutina bajo análisis, en particular el invariante de representación, nos brinda información valuable, que es posible aprovechar tanto para filtrar entradas de una *suite* y producir *suites de tests* más pequeñas (con una efectividad comparable a la de las *suites* exhaustivas acotadas), como para conseguir aumentar la escalabilidad en esta técnica de generación.

Este trabajo está estructurado de la siguiente manera: en el Capítulo 2 se exponen algunos conceptos preliminares necesarios para el buen entendimiento de los subsiguientes capítulos; en los Capítulos 3, 4 y 5 se presentan las técnicas mencionadas, describiendolas en detalle. En el Capítulo 6 se presentan la evaluación experimental de cada técnica presentada, incluyendo una sección de análisis de resultados, donde se discuten los resultados experimentales obtenidos. Para finalizar, en los Capítulos **??** y 7 se presentan algunos trabajos relacionados y las conclusiones, respectivamente.

Cabe mencionar que los resultados presentados en esta tesis fueron publicados en [7], [9] y [10]. Además en el Capítulo 5 se introducen algunos resultados publicados en [32].

Capítulo 2

Conceptos Preliminares

En este capítulo, se exponen algunos conceptos necesarios para el buen entendimiento de los capítulos subsiguientes.

2.1. Testing

El *testing* es la forma más natural de verificar una pieza de código, la cual consiste en comprobar el comportamiento de los programas en un conjunto de situaciones particulares. Más precisamente, el programa bajo análisis se ejecuta con diferentes entradas, para luego examinar si el resultado obtenido es el esperado. Por supuesto, resulta imposible ejecutar el programa bajo todas las posibles entradas, ya que el conjunto de posibles entradas podría ser muy grande e incluso infinito (por ejemplo, una rutina que manipula un entero). Por ello, es necesario encontrar un conjunto de casos de prueba (*suite de tests*) adecuada, es decir que provea suficiente confianza de que el programa se comporta de la forma esperada, incluso para los casos en donde no ha sido probado. El *testing* es parte explícita de los procesos de desarrollo de software, los cuales en general incluyen esta etapa luego de la implementación, o en el caso de los procesos de desarrollo "agiles", el *testing* está presente en varias fases del proceso de desarrollo, principalmente como parte de la implementación.

2.2. Testing Exhaustivo acotado

El *Testing* exhaustivo acotado es una metodología de *testing* que consiste en probar exhaustivamente el programa bajo análisis para todas las entradas válidas

dentro de cierto rango preestablecido.

Ejemplo 2.2.1. Supongamos que se desea probar una rutina que toma como parámetro dos enteros. En principio, conforme a esta metodología, se deben acordar las cotas, en este caso particular: *cota inferior* y *cota superior*. Supongamos que se establecen las mismas en 1 y 10 respectivamente.

El *Testing* exhaustivo acotado consta de dos etapas:

1. Construir todas las posibles entradas dentro de las límites establecidos: para este sencillo ejemplo, las entradas se construyen mediante la combinación de los posibles valores de cada parámetro, obteniendo pares de enteros, concretamente 100 pares de enteros.

2. Dado el programa bajo análisis, y las entradas generadas, ejecutar el programa con esas entradas como parámetro, para luego examinar si el resultado obtenido se corresponde con el esperado.

Esta técnica es muy usada en varios contextos, pero particularmente útil cuando se trata de rutinas parametrizadas con entradas estructuralmente complejas, las cuales deben satisfacer restricciones en general complejas para ser consideradas válidas. Un ejemplo de este tipo de entradas son las estructuras de datos alojadas en memoria dinámica.

El *testing* exhaustivo acotado, es una técnica que realizada de manera manual resulta inviable, por lo que su automatización es absolutamente indespensable. Si bien ciertas partes no son completamente automatizables, ya que son inherentemente humanas, en particular, la generación exhaustiva acotada de entradas podría ser realizada por un procedimiento automático, el cual generalmente involucra algún mecanismo de *constraint solving* tal como: búsqueda (*backtracking*), model checking, o combinaciones de estos.

2.2.1. Generación (automática) exhaustiva acotada de entradas

En lo que respecta a la generación automática de entradas, se pueden distinguir dos importantes enfoques [17]:

- *técnicas basadas en generación* (*generating approach*): Se basan en generar instancias de las entradas llamando a una rutina *generadora* que combina llamadas a constructores y rutinas de inserción de la estructura.

- *técnicas basadas en filtrado* (*filtering approach*): Estas técnicas se basan en construir entradas candidatas usando solo su definición estructural, para luego chequear si estos candidatos se corresponden a entradas válidas. Este enfoque requiere una especificación de las entradas válidas.

Con el objetivo de ilustrar el último enfoque, el cual es de interés para esta tesis, a continuación se introduce un ejemplo.

Ejemplo 2.2.2. Consideremos la siguiente implementación de árboles binarios en Java:

```
public class BinaryTree {
    private Node root;
    private int size;

    .....
}
```

```
public class Node {
    int key;
        Node left;
        Node right;

        .....
}
```

Supongamos que se desean generar, mediante un enfoque basado en filtrado, instancias de árboles binarios de hasta 3 nodos, con claves entre 1 y 3. Los posibles candidatos son todos aquellos que se consiguen como resultado de la combinación de todas las posibles asignaciones de valores a los campos de la estructura: la raíz del árbol podría tomar 1 de 4 valores: $Null$, N_0, N_1 o N_2, lo mismo sucede con los hijos izquierdo y derecho de la raíz y los restantes nodos del árbol; así como también el resto de los campos, `size` y `key`, deben variar su valor dentro de los rangos especificados (0 y 3 para `size`, 1 y 3 para las claves (`key`) de cada nodo).

En números:

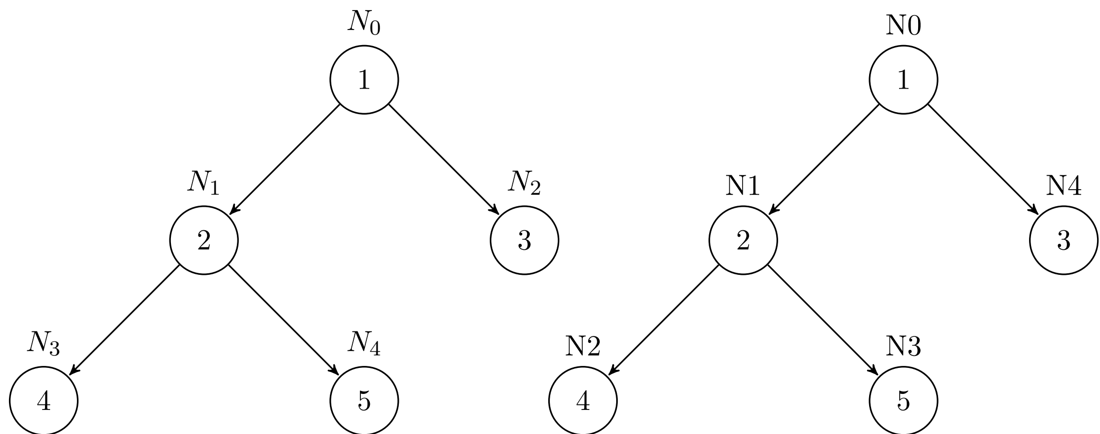
- 4 posibles asignaciones de valores para `root`.
- 3^3 asignaciones de claves a nodos (3 posibles valores de claves para cada uno de los 3 nodos del árbol)
- 4^3 asignaciones al campo `left` (4 posibles valores para cada uno de los 3 nodos del árbol)
- 4^3 asignaciones al campo `right` (4 posibles valores para cada uno de los 3 nodos del árbol)
- 4 posibles asignaciones de valores para el campo `size`.

De esta manera, se pueden determinar 1.769.472 candidatos posibles, de los cuales solo 9 de estos candidatos corresponden a árboles binarios bien formados, es decir, válidos. Conforme a este enfoque, luego de construir posibles candidatos se requiere, mediante alguna especificación de las entradas válidas, distinguir estos 9 candidatos válidos. La especificación requerida establece cuándo un objeto (que respeta la definición estructural dada) es considerado un árbol binario (es decir, acíclico con cantidad de nodos en el árbol compatible con el campo `size`.)

Como se puede notar, una pequeña porción de todos los posibles candidatos finalmente, resultan ser instancias válidas. Existen varias razones que colaboran en esta explosión de posibles estructuras: la construcción de entradas isomorfas o simétricas (es decir, iguales estructuralmente) y la iteración sobre elementos irrelevantes de la entrada son las más destacadas.

Entradas simétricas

Continuando con el ejemplo presentado arriba, y teniendo en cuenta árboles de hasta 5 nodos: N_0, N_1, N_2, N_3, N_4, para una única asignación de valores a claves (campo `key` de `Nodo`), todos los árboles cuyos nodos son una permutación (válida) de N_0, N_1, N_2, N_3, N_4. son instancias de árboles binarios que representan la misma estructura. Por ejemplo, las siguientes son considerados estructuras isomorfas o simétricas, es decir representan el mismo árbol:



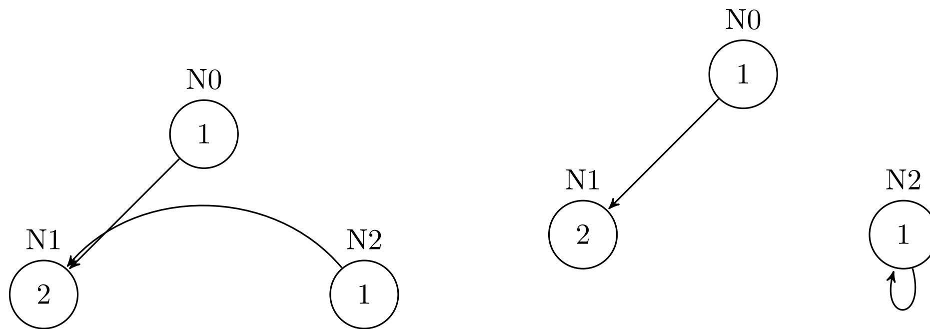
Se puede observar que, excepto por el nombre que se le ha asignado a cada nodo, ambas estructuras representan el mismo árbol. Se podría pensar en estos nombres de nodos como la dirección de memoria en el *heap* de cada uno de ellos. Si nos

abstraemos de las direcciones específicas de los nodos, un total de $n!$ estructuras diferentes representan el mismo árbol, siendo n la cantidad de nodos en el árbol.

La relación de *isomorfismo* entre instancias válidas, divide el espacio de entradas válidas en particiones isomorfas [11]. En general, es deseable construir una única instancia de cada clase de equivalencia de esta partición, lo cual, se puede lograr mediante la definición de algún orden único que debe ser utilizado para asignar de identificadores a los elementos en las instancias válidas. Por ejemplo, en el caso anterior, se podría definir un orden lexicográfico entre los nodos del árbol: $N_0 < N_1 < N_2 < ..$, y determinar la forma en que estos aparecen en la estructura; por ejemplo forzando la asignación de nombres a nodos, de menor a mayor y de manera consecutiva, desde la raíz y continuando en un recorrido en anchura, como es el caso del árbol que se muestra a la izquierda. De esta manera, el árbol de la derecha, no satisface el orden propuesto, y por lo tanto debería ser desechado.

Iteracion sobre elementos irrelevantes de la entrada

Ejemplo 2.2.3. Nuevamente considerando el ejemplo 2.2.2, las siguientes instancias del *heap* representan la misma estructura de clase `BinaryTree`:



Ambas estructuras representan el mismo árbol con 2 nodos: N_0 y N_1, siendo N_0 la raíz y N_1 el hijo izquierdo de N_0. N_2 pertenece a una porción del *heap* no alcanzable desde la raíz del mencionado árbol, por lo que no forma parte de la estructura. No solo es deseable no generar estas dos instancias, sino que además es deseable no examinar la validez de estos 2 candidatos, es decir, a sabiendas que N_2 no es parte de la estructura, no se debería iterar sobre N_2 para producir un nuevo candidato.

2.3. Invariante de Representación

Un invariante de representación es la propiedad que distingue instancias bien formadas de una clase, de aquellas mal formadas. Éste puede ser definido declarativamente, por ejemplo usando algún lenguaje de especificación de contratos tal como JML [30], o operativamente, es decir, vía una rutina que, cuando se aplica a un objeto retorna verdadero si y solo si el objeto es válido, es decir está bien formado. Esta rutina es generalmente llamada, por convención, `repOk()` [23].

Consideremos la implementación de árboles binarios en Java presentada en el Ejemplo 2.2.2.

Las instancias válidas de esta clase son aquellas que cumplen con:

- Ser un árbol acíclico,
- la cantidad de nodos en el árbol debe coincidir con `size`.

Podría definirse el invariante de representación que caracteriza a estas instancias, de manera operativa (`repOK()`), de la siguiente manera:

```
public boolean repOK() {
        if (root == null)
            return size == 0;
        Set visited = new HashSet();
        visited.add(root);
        LinkedList workList = new LinkedList();
        workList.add(root);
        while (!workList.isEmpty()) {
             Node current = (Node) workList.removeFirst();
             if (current.left != null) {
                  if (!visited.add(current.left))
                      return false;
                  workList.add(current.left);
             }
             if (current.right != null) {
                 if (!visited.add(current.right))
                     return false;
                 workList.add(current.right);
             }
        }
        return (visited.size() == size);
}
```

Esta rutina sin parámetros, retorna *verdadero* si y solo si el objeto sobre el cual se invoca es una instancia bien formada de la clase `BinaryTree`. En primera instancia, se recorre la estructura para, mediante estructuras auxiliares, examinar si corresponde a un árbol sin ciclos; y por último se chequea si el tamaño es consistente, es decir, si la cantidad de nodos se corresponde con el valor del campo `size`.

Los nuevos objetos creados de clase `BinaryTree` deben satsifacer este invariante de representación, es decir, los constructores públicos de esta clase deben asegurar que el invariante de representación se mantiene cuando terminan. Además, los métodos públicos que modifican objetos (rutinas de inserción y eliminación de elementos en el árbol) de clase `BinaryTree` deben preservar el invariante de representación, esto significa que, asumiendo que el invariante de representación se cumple antes de la llamada a un método, este método debe asegurar que el invariante de representación también se mantiene cuando termina. Muchas veces a estos requerimientos se los asocia con la idea de contrato, un método asume la responsabilidad de cumplir con él, siempre y cuando éste se haya cumplido antes.

Como se postula en [23],es una buena práctica de desarrollo equipar las implementaciones de estructuras complejas con rutinas `repOk`, debido a que estas rutinas son de gran ayuda en el *debugging* de sus implementaciones. Las rutinas `repOk` pueden ser llamadas en *test* de pruebas, para chequear si ciertos métodos establecen o preservan el invariante, o directamente dentro de métodos y constructores de la clase, antes de que estos retornen, de la manera descripta arriba.

2.4. El Algoritmo de `Korat`

`Korat` es una herramienta que implementa un algoritmo para la generación (basada en filtrado) exhaustiva acotada de casos de *test* compuestos por datos estructuralmente complejos [11].

Ejemplo 2.4.1. Supongamos, por ejemplo, que se necesita probar una rutina que toma como parámetro una lista simplemente encadenada ordenada. La siguiente definición de esta estructura (una variante de `SinglyLinkedList`, es la que acompaña la distribución de `Korat`):

```
class SortedSinglyLinkedList{
    Node header;
    int size;
    ...
}
```

```
class Node{
    Integer elem;
    Node next;
    ...
}
```

Una instancia de esta lista está compuesta de una referencia a la cabeza de la lista (nodo `header`), y un valor entero indicando la longitud de la lista (`size`). La lista simplemente encadenada comienza con un nodo `header` que no tiene elementos (nodo centinela); el contenido de la lista (sus valores) comienza desde el segundo nodo. La lista debería ser acíclica, ordenada (sin tener en cuenta el nodo centinela), y el número de nodos en la lista (menos el `header`) debería coincidir con el valor del campo `size`.

`Korat` puede ser usada para generar las listas del ejemplo 2.4.1 automáticamente, para luego poder probar la rutina bajo análisis con las mismas. `Korat` requiere dos rutinas extras en la clase asociada con la entrada (en el ejemplo, `SortedSinglyLinkedList`). Una de ellas es la rutina lógica sin parámetros, `repOk()` [23], que chequea si una estructura satisface su invariante de representación, detallado en la sección anterior. Para el Ejemplo 2.4.1, `repOk()` debería chequear que:

- el nodo centinela no es `null` y que no hay ningún elemento almacenado en él,
- la lista es acíclica,
- la lista está ordenada y
- el número de nodos en la lista menos uno coincide con el valor del campo `size`.

Siguiendo estos puntos, `repOk()` podría codificarse de la siguiente manera:

```
public boolean repOk() {
    if (header == null)
        return false;
    if (header.element != null)
        return false;
    Set visited = new HashSet();
    visited.add(header);
    Node current = header;
    while (true) {
```

```
            Node next = current.next;
            if (next == null)
                break;
            if (next.element == null)
                return false;
            if (!visited.add(next))
                return false;
            current = next;
        }
        if (!this.isSorted())
            return false;
        if (visited.size() - 1 != size)
            return false;
        return true;
    }
```

Esta rutina `repOk` está codificada de una manera muy particular, la cual más adelante se va claramante que es conveniente para explotar el mecanismo de poda que `Korat` incorpora en la búsqueda de intancias válidas. Ésta chequea, en principio, que `header` sea un nodo centinella (es decir, no nulo y sin clave), luego mediante el uso de una estructura auxiliar se chequea aciclicidad. Por último, se examina (mediante una rutina `isSorted()`) que la lista este ordenada. Notese que la lista se visita dos veces: en el primer recorrido se chequea aciclidad y en el segundo se examina si la misma está ordenada.

La otra rutina que `Korat` requiere es una rutina que provee las cotas para los dominios involucrados en la estructura (por ejemplo, 1 lista, 0 a 4 nodos, 1 a 3 valores enteros), la cual es llamada *finitization*.

`Korat` genera todas las posibles estructuras válidas dentro de las cotas dadas. Es decir, todas aquellas que satisfacen la rutina `repOk()`. Siguiendo con el Ejemplo 2.4.1 , esto significa que `Korat` generará todas las listas simplemente encadenadas con nodo centinela, acíclicas y ordenadas, donde el campo `size` coincide con la cantidad de nodos menos uno, de tamaño como máximo 3, conteniendo valores entre 1 y 3. Para lograr esto, `Korat` construye un vector, donde cada entrada corresponde a un valor de un campo de los objetos involucrados, de manera más específica, un conjunto de objetos de una clase forma un *dominio de clase* (*class domain*). El conjunto de valores que cada campo puede tomar forma un *dominio de campo* (*field domain*). De esta manera los índice dentro de un vector corresponden a identificadores únicos en el dominio correspondiente. Es importante notar que un *dominio de campo* es

la unión de algunos *dominios de clase*. Continuando con el ejemplo de las listas, el vector debería tener longitud 10:

- 1 valores para la cabeza de la lista (`header`),
- 1 valor para el `size` de la lista,
- 8 campos para los correspondientes campos (`elem` y `next`) para cada uno de los 4 nodos que la lista podría tener.

Por ejemplo, el vector:

$$\langle N0, 0, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL\rangle$$

representa la lista vacía. `Korat` utiliza estos vectores, llamados *vectores candidatos*, para representar las estructuras candidatas, pero, como se especifico arriba, las entradas corrsponden al índice en el dominio correspondiente. Por ejemplo, el vector candidato:

$$\langle 1, 0, 0, 0, 0, 0, 0, 0, 0, 0\rangle$$

se corresponde con el vector mostrado previamente (cada entrada en el vector tiene el primer valor de su dominio, es decir, el valor con índice 0 (excepto por el primer valor) la cabeza de la lista, la cual apunta al primer node).

Generalmente, la mayoría de los vectores candidatos corresponde a estructuras inválidas, es decir a estructuras que no satisfacen el `repOk()`. De hecho, el espacio de candidatos posibles es, en el ejemplo de las listas, 3200000 ($5^5 \times 4^5$), pero solo existen 8 listas simplemente encadenadas ordenadas válidas (sin tener en cuenta estructuras isomorfas), dentro de las cotas establecidas anteriormente.

`Korat` explora de manera exhaustiva el espacio de los vectores candidatos, usando búsqueda (*backtracking*) con un mecanismo sofisticado de poda. Más en detalle, `Korat` hace lo siguiente: comienza con el vector inicial con todos los índices en cero. Luego ejecuta `repOk()` sobre este vector, monitoreando los campos accedidos en su ejecución, los cuales almacena en una pila. `Korat` utiliza está pila para realizar la búsqueda sobre los vectores candidatos, de la siguiente forma: si el vector candidato corriente satisface el `repOk()`, entonces se considera un candidato válido (en este caso, todos los campos alcanzables deben estar en la pila de campos accedidos). Si `repOk()` falla, entonces el candidato es descartado. Para construir el próximo

candidato, Korat incrementa, a su siguiente valor, el último campo accedido. Si uno, o más campos, accedidos ya llegaron a su valor máximo posible, su valor es reseteado a 0, y el campo accedido anterior a ese/esos es incrementado. Cuando todos los campos alcanzaron sus valores máximos dentro de sus correspondientes dominios, entonces el espacio de estados de vectores candidatos ha sido explorado de manera exhaustiva y Korat termina.

Como se puede notar, si repOk() falla, puede que no todos los campos alcanzables estén en la pila de campos accedidos, debido a que repOk() podría fallar antes de explorar todos los campos alcanzables, por ejemplo, en el caso de las listas del ejemplo 2.4.1, si los dos primeros elementos de la lista no están en orden, repOk() devuelve falso sin la necesidad de mirar el resto de la lista. Realizar la búsqueda solo sobre los campos accedidos es lo que le permite a Korat podar grandes partes del espacio de vectores candidatos. Debido a que si el último campo accedido para determinar la falla no es modificado, la salida de repOk() no cambiará debido a su determinismo. En otras palabras, las partes de la estructura visitadas por repOk() no cambiarían y por lo tanto repOk() fallaría nuevamente. Es importante notar que la forma en que repOk está escrito afecta sustancialmente a la eficiencia de Korat: repOk debería fallar lo antes posible, es decir, visitando la menor cantidad de elementos de la estructura, y de esta manera obtener una pila de campos accedidos menor, lo cual lleva a Korat a explotar su mecanismo de poda.

Es importante notar que el procedimiento llevado a cabo por Korat nos garantiza que ninguna estructura es pasada por alto, es decir, este procedimiento tiene la propiedad de *completeness*. Con el propósito de clarificar este concepto, se introduce una descripción (en pseudo-código) del Algoritmo de Korat:

```
function korat() {
    Vector current = initVector;
    Stack accessedFields = new Stack();
    boolean ok;
    do{
        (ok, accessedFields) = current.repOk();
        if(ok){
            reportValid(current);
            accessedFields.push(current.reachableFields −
               accessedFields);
        }
```

```
        field = accessedFields.pop();
        while (!accessedFields.isEmpty() &&
                current[field] >= nonIsoMax(current,
                    accessedFields, field)) {
            current[field] = 0;
            field = accessedFields.pop();
        }
        if (!accessedFields.isEmpty()) current[field]++;
    } while (current != lastVector && !accessedFields.
        isEmpty())
}
```

Es importante notar que en esta descripción abstracta del algoritmo, se hacen abusos de notación y se supone que `repOk()` se aplica sobre vectores candidatos; y retorna, no solo el resultado de la ejecución de esta función sobre el vector candidato correspondiente (un valor lógico, indicando si el candidato es válido o no), sino también una pila con los campos visitados durante su ejecución (`accessedFields`). Se puede notar como la búsqueda se realiza sobre los campos accedidos por `repOk()`; además, cuando el vector corriente es válido, todos los campos alcanzables son insertados en la pila de campos accedidos, de esta manera estos también son considerados durante la búsqueda y ningún candidato válido será omitido.

Mediante el mecanismo de búsqueda descripto, además, de su técnica de poda, `Korat` también evita generar estructuras isomorfas [11]. Básicamente, dos candidatos son isomorfos si solo difieren en la identidad de sus objetos, es decir, si uno de los candidatos puede ser obtenido desde el otro cambiando la identidad de sus objetos. La mayoría de las aplicaciones no depende de la identidad de los objetos (la cual representa la dirección de memoria o la referencia a los objetos en el *heap*), y por lo tanto cuando se generan estructuras es deseable evitar la generación de estructuras isomorfas, cuyo tratamiento terminará siendo redundante. `Korat` evita generar candidatos isomorfos definiendo un orden lexicográfico entre los vectores candidatos, y generando solo el más pequeño en ese orden, entre todos los candidatos isomorfos. Básicamente, cuando se considera el rango (es decir, posibles valores) de los campos de clases (*class-typed*) en la construcción de candidatos, durante la búsqueda, éste es restringido a tomar solo uno más de los valores ya referenciados en la estructura dentro de su correspondiente dominio. Con este fin, dentro de la descripción del algoritmo de `Korat` presentado, se usa una función auxiliar, llamada `nonIsoMax`, la cual retorna el máximo índice posible para un campo dado. Por ejemplo, supongamos que en la construcción de candidatos es necesario considerar diferentes valores para una

posición i en el vector candidato. Además, supongamos que la posición i corresponde a un dominio de clase D, y ningún campo de ese dominio a sido accedido antes de i en la última invocación de `repOk()`, por lo tanto el único valor posible para la posición i es 0. Dicho de otra manera, si k objetos diferentes de dominio D han sido accedidos en la última invocación de `repOk()`, estos deben ser indexados desde 0 a $k-1$, y por ende la posición i irá desde 0 hasta k, pero nunca será mayor a k. Los mecanismos de poda y eliminación de candidatos isomorfos de `Korat` permiten, en muchos casos, reducir de manera significativa el espacio de búsqueda. Volviendo a las listas del Ejemplo 2.4.1, `Korat` explora explora solo 319 candidatos de 3200000 casos posibles para listas de tamaño 0 a 3, hasta 4 nodos, y valores enteros entre 1 y 3 [11, 26].

2.5. Criterios de Test

Probar una pieza de código de manera exhaustiva es generalmente poco factible, debido al gran tamaño (muchas veces infinito) del espacio de posibles entradas para el código bajo análisis. Debido a esto, generalmente se considera solo un subconjunto significativo de las posibles entradas para testear el código. A fin de probar el software bajo diferentes circunstancias, y de esta manera incrementar las chances de encontrar errores, se proponen diferentes criterios de *test* [16]. Un criterio de cobertura de código es una forma de medir cuán adecuada es una *suite de tests*, es decir, cuán bien sus entradas "ejercitan" el programa bajo prueba. Los criterios de cobertura se clasifican principalmente en criterios de *Caja Blanca* y criterios de *Caja Negra* [28].

2.5.1. Criterios de Cobertura de Caja Blanca

Los criterios de *Caja Blanca*, se caracterizan por utilizar la estructura del código del programa bajo prueba para medir cuán adecuada es una *suite de tests*.

Entre los criterios de caja blanca existentes se pueden mencionar: *cobertura de sentencias*, *cobertura de decisión* y *cobertura de caminos*.

Una *suite de tests* satisface el criterio de *cobertura de sentencias*, si cada sentencia del código bajo análisis se ejecuta al menos una vez por algún caso de *test* en la *suite*.

Ejemplo 2.5.1. Considere el siguiente ejemplo, tomado de [28]:

```
public void foo(int a, int b, int x){
```

```
        if(a >1 && b=0){
                x=x/a;
        }
        if(a=2 || x>1){
                x=x+1;
        }
}
```

Una *suite de tests* adecuada para esta rutina, según *cobertura de sentencia*, podría ser aquella conteniendo un único *test*, por ejemplo: $(a = 2, b = 0, x = 2)$, el cual, debido a que hace verdaderas las dos decisiones de la rutina (condiciones de ambos `if-then`), alcanza para atravezar cada sentencia del código.

Este criterio es considerado muy débil, ya que cualquier error en alguna decisión (condiciones de estructuras `if-then-else`, `while`, etc.) del programa no sería detectado por este criterio [41]. En el Ejemplo 2.5.1, la condición del primer `if-then` podría contener un *or*, en lugar de un *and*, y esto no sería detectado por la *suite de tests*: $(a = 2, b = 0, x = 2)$.

Un criterio más fuerte es el criterio de *cobertura de decisión*, el cual se satisface sobre una *suite de tests*, si cada punto de decisión del código se evalúa por `true` y por `false` por algún *test* en la *suite*.

Ejemplo 2.5.2. Dada la siguiente rutina JAVA, la cual chequea si un año es, o no, bisiesto:

```
public boolean bisiesto(int a){
        if((a %4 == 0) &&(a %100!=0)) || (a%400 ==0)){
                return true;
        }else return false;

}
```

Una *suite de tests* adecuada para esta rutina, según *cobertura de decisión*, podría ser: $\{(a = 2001), (a = 2008)\}$, la cual, con el primer caso de *test* hace falsa la condición del `if-then` y con el segundo, hace verdadera dicha condición.

Se puede notar que, *cobertura de decisión* incluye a *cobertura de sentencia*, es decir, toda *suite de tests* que satisface *cobertura de decisión* también satisface *cobertura de sentencia*.

Sin embargo, éste no es adecuado cuando se está en presencia de deciciones complejas, como lo es la decisión del Ejemplo 2.5.1. En este caso, la condición del

`if-then` podría, por ejemplo, contener la condición: ($a\%400 \; != \; 0$), en lugar de ($a\%400 \; == \; 0$) y esto pasaría completamente inadvertido si se utiliza la *suite de tests* dada. Un criterio más apropiado para estos casos, es el criterio de *cobertura de condición*; el cual, expresa que una *suite de tests* es adecuada para probar cierta pieza de código, si cada condición dentro de cada decisión del programa bajo análisis, se evalúa al menos una vez, por *true* y por $false$. Siguiendo el mismo Ejemplo (2.5.2) *bisiesto* posee un punto de decisión (`if-then`), el cual está compuesto por 3 condiciones: $a\%4 == 0$, $a\%100! = 0$ y $a\%400 == 0$.

Otros criterios de test más fuertes surgen de la combinación de estos dos últimos: *cobertura de decisión/condición* y *cobertura de decisión/condición múltiple*.

Por último, una *suite de tests* satisface el criterio de *cobertura de camino*, si todos los posibles caminos de ejecución son ejercitados por algún *test* presente en la *suite*, donde un camino de ejecución es un camino, desde el nodo de inicio al nodo final, en el *grafo de flujo de control* del programa bajo análisis. Un *grafo de flujo de control* es la representación, usando grafos dirigidos, de todos lo caminos que pueden ser atravezados durante la ejecución del programa. Los nodos del grafo representan segmentos de sentencias que se ejecutan secuencialmente y los arcos representan transferencias de control entre nodos.

Ejemplo 2.5.3. Supongamos el siguiente método en JAVA:

```
public static  boolean capicua(char[]  list) {
        int index = 0;
        int l = list.length;
        while (index<(l-1)) {
             if (list[index] != list[(l-index)-1]) {
                  return false;
             }
             index++;
        }
        return true;
   }
```

El grafo de flujo de control correspondiente se muestra en la Figura 2.1 (los puntos de decisión en el programa son representados con diamantes). Un posible camino en este grafo es *"acde"*, es decir una ejecución en donde el ciclo es atravesado una única vez. Este ejemplo hace evidente que este criterio resulta impracticable para la mayoría de los programas, particularmente para aquellos que contienen ciclos, ya

que resulta en una cantidad infinita de caminos.

Con el propósito de hacer de este criterio un criterio viable, se imponen restricciones sobre los caminos, para obtener un subconjunto "representativo" de ellos; dando lugar a nuevos criterios de cobertura [41], entre ellos se destacan:

- *cobertura de caminos simples*: el cual requiere que una *suite de tests*, para ser adecuada, cubra todos los *caminos simples* del grafo de flujo de control. Un *camino simple* es aquel que no contiene ocurrencias repetidas de arcos [41].

- *cobertura de caminos elemental*: el cual requiere que una *suite de tests*, para ser adecuada, cubra todos los *caminos elementales* del grafo de flujo de control. Un *camino elemental* es aquel que no contiene ocurrencias repetidas de nodos [41].

2.5.2. Criterios de Cobertura de Caja Negra

Los criterios de *caja negra*, ven el código del programa bajo prueba como una "caja negra", es decir, unicamente la especificación del mismo es usada para medir cuál adecuada es la *suite de tests*.

Uno de los criterios más importante dentro de esta clasificación, es el criterio de *partición en clases de equivalencia*, el cual consiste en dividir las entradas en subdominios según la especificación del programa bajo prueba, es decir, los subdominios deberían corresponder a casos similares con respecto a consitencia en las fallas (es decir, si un caso de *test* en un subdominio detecta un error, se espera que el resto de los casos del mismo subdominio también lo detecten). De esta manera, si el programa funciona correctamente para un caso de *test* en un subdominio, se supone que lo hará para los otros casos pertenecientes al mismo subdominio [28]. Para lograr una adecuada división de las entradas, es necesario identificar los casos de *test* para los cuales se especifican diferentes comportamientos. Una técnica para lograr este particionado consiste en considerar cada condición especificada sobre las entradas como una clase de equivalencia; si las entradas correspondientes a una clase no se tratan uniformemente (por ej: distintas salidas), se debe particionar aún más las clases teniendo en cuenta los diferentes tratamientos. Además, se deben incluir las clases correspondientes a entradas inválidas.

Otro criterio de caja negra, que técnicamente, corresponde a un caso especial del anterior [8], es el criterio de cobertura de *Predicados lógicos sin parámetros* (PBP por su siglas en inglés)[24]. Este criterio de *test* se aplica a rutinas cuya

entrada es un objeto, en el sentido de la programación orientada a objetos. Éste consiste en observar la clase correspondiente a la entrada, y las operaciones lógicas sin parámetros con visibilidad pública que ésta provee.

Ejemplo 2.5.4. Una pila podría tener operaciones lógicas `isEmpty()` y `isFull()`, las cuales retornan verdadero si la pila está vacía y si la pila está llena respectivamente; y falso en otro caso. Un conjunto de casos de *test* es adecuado con respecto a PBP, si cada combinación factible de valores lógicos para los predicados se satisface al menos por un caso de *test* en el conjunto. Para este caso, se necesita, en principio, cuatro casos de *test* que cumplan con los siguientes predicados:

- `isEmpty()` y `isFull()`
- `isEmpty()` y no `isFull()`
- no `isEmpty()` y `isFull()`
- no `isEmpty()` y no `isFull()`

Como se puede observar, de éstos, el primero de ellos es el único inviable, los restantes deberían ser cubiertos por algún caso de *test* en el conjunto, es decir, al menos un *test* debería hacer verdadero estos predicados.

2.5.3. Testing de Mutación

Testing de Mutación es un criterio de *test*, también considerado de *caja blanca*, útil para medir la efectividad de una *suite de tests*, según cuán buena es ésta para detectar errores en el programa. Para ello, lo que se hace es insertar cambios, llamados mutaciones, al código del programa bajo análisis, obteniendo programas distintos al original. A cada uno de estos programas se lo denomina *mutante*. Si un mutante y el programa original se comportan de manera diferente (para algún caso de *test* en la *suite*) el error es detectado por la *suite* (el mutante se considera muerto), en otro caso el mutante queda vivo. Un mutante puede mantenerse vivo basicamente por dos razones [41]:

- La *suite de tests* es inadecuada, es decir no es útil para detectar los errores insertados en el código, por ejemplo, porque no ejercita la parte del código mutada.

- El mutante es equivalente al programa original. La mutación nos deja un programa que se comporta exactamente igual al original. Esto sucede, en general, con pocos mutantes [41].

Tomando un conjunto de mutantes y una *suite de tests* se mide el porcentaje de mutantes que ésta pudo matar, lo que es usado como un indicador de que tan buena es la *suite* para detectar errores.

Varias herramientas dan soporte al *testing* de mutación para Java: `Mujava` [4], `Jumble` [3], `Judy` [2], etc. En general, las mutaciones de código que éstas realizan se clasifican en: mutaciones a *nivel de métodos* y mutaciones a *nivel de clases*. Los operadores de mutación a *nivel de métodos*, producen cambios en el programa mediante inserción, reemplazo o eliminación de operadores primitivos del lenguaje: $<, >, ==$, etc. Por otro lado, los operadores de mutación a *nivel de clases*, producen mutaciones haciendo uso de características específicas de los lenguajes orientados a objetos, como lo son la herencia, el polimorfismo, y la ligadura dinámica [25].

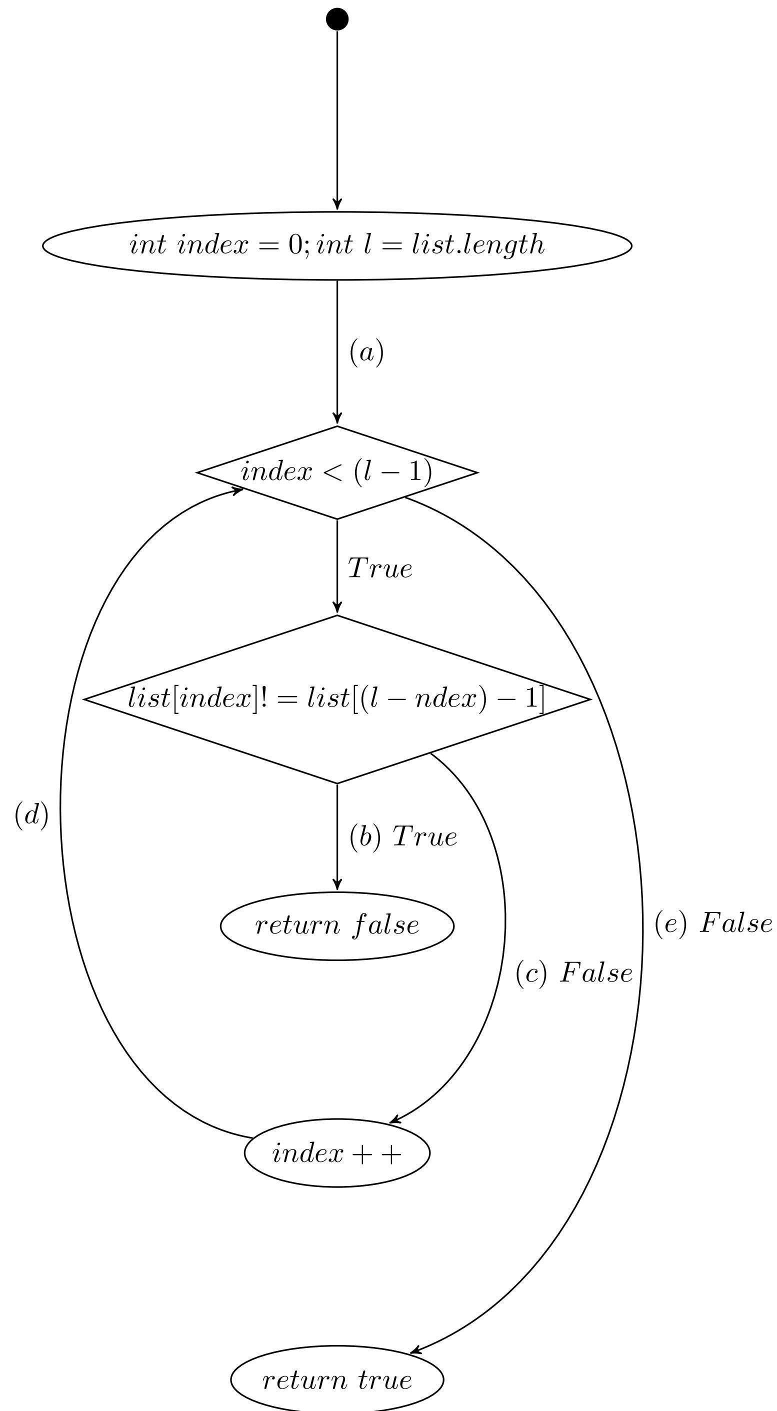


Figura 2.1: Grafo de Flujo de Control del ejemplo 2.5.3

Capítulo 3

Incorporación de Criterios de cobertura en la generación exhaustiva acotada

En adelante se presenta la primera tećnica propuesta para mejorar el *testing exhaustivo acotado*, basado en *constraint solving*, particularmente para mecanismos basados en filtrado. Este Capítulo se organiza de la siguiente manera: en la Sección 3.1 se presenta la idea del enfoque, en la Sección 3.2 se introduce un ejemplo que ilustra la aplicación de la técnica, y que además sirve como guía para explicar cómo incorporar el criterio de cobertura en el proceso de generación exhaustivo acotado, lo cual se describe en la Sección 3.3. El enfoque es presentado en el contexto de la herramienta `Korat`.

3.1. Intuición del enfoque

A grandes rasgos, la técnica consiste en guiar la búsqueda considerando algún criterio de cobertura de *testing* dado. Este criterio es incorporado en el mecanismo de *constraint solving* de tal manera que la exploración de posibles casos de *test* es reducida, sin perder clases de entradas viables correspondientes al criterio dado.

Este enfoque puede considerarse un complemento al mecanismo basado en "filtrado" [17] en generación exhaustiva acotada de casos de *test*, es decir, el proceso de generar exhaustivamente todas las posibles entradas (dentro de límites preestablecidos) y filtrar para conservar solo aquellas válidas (que satisfacen la especificación). Esta técnica tiene en cuenta un criterio de *test* como parte del proceso de

"generación y filtrado". Básicamente, uno de los principales objetivos de la técnica es evitar la exploración de porciones del espacio de búsqueda, cuando se tiene la garantía que los candidatos en esas porciones corresponden a clases ya cubiertas por otras entradas de *test* previamente generadas. El resultado es un conjunto de *tests* que se encuentran entre exhaustivo acotado y cobertura de clases de equivalencia óptima (un representante por cada clase).

Como se mencionó en capítulos anteriores, la motivación de este trabajo radica en el hecho de que, en muchos casos, utilizar generación exhaustiva acotada de casos de *test* para probar una pieza de código para todas las entradas válidas hasta cierta cota, incluso para cotas pequeñas se vuelve algo costoso e impracticable, debido a la gran cantidad de entradas obtenidas. Por consiguiente, un criterio de *test* podría ser empleado con el objetivo de "podar" la generación de casos de *test*, obteniendo una cobertura exhaustiva acotada de clases de equivalencia, asociadas con el criterio dado. Por ejemplo, supongamos que se desea probar un programa y que se cuenta con un criterio de *test* sobre él. Si se está interesado en cobertura de clases de equivalencia, debería ser suficiente generar un único caso de *test* para cada clase (viable) de equivalencia. Por otro lado, mediante generación exhaustiva acotada se deberían construir todas las entradas válidas dentro de las cotas dadas. En su lugar, se propone hacer una especie de generación exhaustiva, pero aprovechando la posibilidad de podar parte del espacio de búsqueda cuando se tiene la certeza que todos los candidatos correspondientes a esa parte podada, pertenecen a clases de equivalencia ya cubiertas.

Más detalladamente, la técnica propuesta se basa en las siguientes observaciones. El mecanismo de generación de casos de *test*, en el cual está sentada la generación exhaustiva acotada en técnicas basadas en filtrado, generalmente contiene algún proceso para evitar la generación de entradas redundantes. Independientemente de cómo ese proceso está implementado, éste se ajusta, en la mayoría de los casos, a alguna operación de poda. Por ejemplo, las fórmulas de rotura de simetría que incorpora `TestEra` en el modelo `Alloy` del programa bajo prueba, orientan al *SAT solver* a evitar parte del espacio de búsqueda (en este contexto asignaciones a variables proposicionales) [21]. De manera similar, `UDITA` poda el espacio de búsqueda con el objetivo de evitar la generación de estructuras isomorfas, mediante la incorporación de un control de isomorfismo dentro del grupo de objetos manipulados, y las operaciones para obtener nuevos objetos de él, durante la construcción de las estructuras alojadas en memoria dinámica [17]; `Korat` realiza una poda similar, imponiendo un orden en los objetos del mismo tipo en el proceso de construcción de entradas válidas [11]. La técnica que se propone en este capítulo, explota tales procesos de poda,

y los usa para evitar porciones del espacio de búsqueda que llevarían a producir entradas que caen en clases de equivalencia ya cubiertas por entradas construidas previamente.

3.2. Ejemplo motivador

Un caso de estudio que ilustra los beneficios de esta técnica es la rutina `listAsSet` que, dada una lista `l` y un conjunto `s`, ambos implementadas sobre listas simplemente encadenadas, determina si `s` es el resultado de convertir `l` a un conjunto, es decir, sin tener en cuenta las repeticiones y el orden de los elementos en la lista. Desde el punto de vista de la implementación, y teniendo en cuenta el invariante de representación de los conjuntos sobre listas simplemente enlazadas, `s` debería ser el resultado de remover repeticiones y ordenar la lista `l`. No es difícil encontrar un contexto en donde rutinas como `listAsSet` resultan relevantes. Una aplicación obvia de tal función sería el chequeo de que una rutina `list-to-set` (transforma una lista en un conjunto) realiza lo correcto. Si se desea generar casos de *test* para probar esta rutina, es necesario proveer dos objetos, una lista simplemente encadenada de enteros (la lista `l`), y una lista simplemente encadenada estrictamente ordenada, (el conjunto `s`):

```
public class ListToSet{
    public StrictlySortedSinglyLinkedList set;
    public SinglyLinkedList list;
    ...

}
```

```
public class
    StrictlySortedSinglyLinkedList{

    Node header;
    int size;
    .....
}
```

```
public  class Node{

     Integer element;
     Node next;
   ....
}
```

```
public class SinglyLinkedList{

    Entry header;
    int size;
   ...
}
```

```
public  class Entry{

    Integer element;
    Entry next;
    ...
}
```

Son instancias válidas de `SinglyLinkedList` aquellos objetos que cumplen con ser listas con nodo centinela, acíclicas, y cuyo valor del campo `size` coincide con la cantidad de nodos en la lista menos 1(el nodo centinela). Por otro lado, son instancias de clase `StrictlySortedSinglyLinkedList` aquellos objetos que cumplen con todo lo antedicho y además sus elementos están estrictamente ordenados.

Si se establecen como cotas para la generación exhaustiva acotada hasta 5 nodos y valores enteros entre 1 y 4 para ambas listas, entonces tenemos 5456 objetos `listToSet` válidos, es decir, que cumplen con la especificación informal dada arriba. La cota establecida es relativamente pequeña, sin embargo la cantidad de objetos es lo suficientemente grande para hacer del *testing* una tarea inviable.

Una alternativa posible es considerar un criterio de cobertura que permita, de manera sistemática, seleccionar algunas de estas entradas. Si se considera, por ejemplo, el criterio de *test* "*predicados lógicos sin parámetro*", introducido en la Sección 2.5.2 del Capítulo 2, y se tienen en cuenta los siguientes predicados:

- `list` está vacía
- `list` tiene elementos repetidos
- `list` está ordenada, y
- `set` está vacío.

entonces, mediante la combinación de los valores de verdad de los mismos se consiguen 16 clases de equivalencia, de las cuales sólo 10 son satisfacibles, por ejemplo, las clases que surgen de considerar *verdadero* al primer predicado y *falso* al tercer predicado, no son clases de equivalencia satisfacibles debido a que no es posible encontrar una instancia de `ListToSet` donde `list` (instancia de `SinglyLinkedList`) sea vacía y desordenada.

Teniendo en cuenta que este criterio se satisface si como mínimo se tiene un caso de *test* por cada clase de equivalencia satisfacible, solo son necesarios 10 casos de *test* para satisfacer este criterio, es decir, un representante por cada clase de equivalencia viable.

El enfoque que se presenta en este capítulo puede ser considerado algo en el medio entre generación exhaustiva acotada, mediante la cual se obtienen *suites* demasiado grandes, y cobertura de clases de equivalencias (un representante por cada clase de equivalencia viable), lo cual produce *suites de tests* pequeñas, pero en general poco efectivas para encontrar errores.

Además, en general, el espacio de candidatos explorados durante la generación exhaustiva acotada suele ser muy grande, por ejemplo, si en este caso se utiliza `Korat` como herramienta de generación automática (imponiendo cota para el campo `size` de ambas listas entre 0 y 5), se exploran 1.274.977 candidatos, incluso cuando esta herramienta incorpora en su algoritmo de búsqueda potentes mecanismos de poda.

Si el criterio de cobertura es utilizado durante el proceso de generación y filtrado para producir "saltos" dentro del espacio de estados válidos con la intención de forzar cambios de clases de equivalencia, es decir, descartar entradas válidas que pertenecen a clases de equivalencia ya cubiertas por otro caso de *test* en la *suite*, se logran reducir entradas válidas (sin eliminar clases de equivalencia viables) evitando mirar parte del espacio de búsqueda.

En la sección siguiente se explica en detalle como incorporar tal poda en el mecanismo de generación, particularmente de `Korat`, guiando la presentación con el ejemplo introducido en esta sección.

3.3. Implementación de la técnica

En adelante, se describe detalladamente la técnica propuesta, y su implementación como una variante de `Korat`. Fundamentalmente, esta técnica se basa en la observación de que, en muchos casos, el número de casos de *test* válidos, acotados con un valor k puede ser demasiado largo, incluso para valores de k pequeños, y por lo tanto evaluar el *software* bajo todos estos casos es generalmente inviable. De esta observación, surge la intención de evitar la generación de algunos casos de *test*; la idea principal es evitar la generación de casos de *test* cuya clase de equivalencia asociada, teniendo en consideración algún criterio de *test*, ya haya sido cubierta por otro caso de *test* presente en la *suite de tests*. La idea es obtener una *suite de tests* que este entre cobertura de clase de equivalencia óptima (uno por clase de equivalencia) y generación exhaustiva acotada. Es decir, realizar generación exhaustiva acotada, pero con un mecanismo de poda basado en la información provista por un criterio de *test* dado. Por ejemplo, si una determinada decisión en el código ya fue cubierta por algún caso de *test*, entonces podemos evitar la generación de aquellos casos de *test* que ejercitan la misma decisión, sin dejar de satisfacer el criterio de cobertura de decisión.

En adelante, se presenta una implementación de esta técnica como una variante, llamada `Korat+`, del algoritmo de `Korat`, descripto en detalle en el Capítulo 2.

Retomando el ejemplo de la rutina `listAsSet` presentado en 3.2, una precondición implicita para este método es que las entradas satisfagan el invariante de representación, es decir, las entradas deben estar compuestas por pares de listas bien formadas (según la especificación (informal) dada en la sección anterior). `Korat` requiere de un predicado imperativo `repOk()` (implementación imperativa del invariante de representación) el cual, cuando se aplica a un objeto retorna verdadero si y solo si el candidato es válido, es decir está bien formado. Una explicación detallada del concepto de invariante de representación y su uso, es dada en el Capítulo 2.

La siguiente es una implementación de `repOK()` para `ListToSet`:

```
// ListToSet
public boolean repOK() {
        if (!list.repOK())
            return false;
        return set.repOK();
}

//SinglyLinkedList
public boolean repOk() {
        if (header == null)
            return false;
        if (header.getElement()!= null)
            return false;
        if(isAcyclicAndSizeConsistent())
            return true;
}
```

```
// StrictlySortedSinglyLinkedList
public boolean repOk() {
        if (header == null)
            return false;
        if (header.getElement()!= null)
            return false;
        if(isAcyclicAndSizeConsistent())
            return isSorted();
}
```

```
public boolean isSorted() {
        if (size>1) {
            for (Node current = header.next; current.next != null; current =
                current.next) {
                if (current.element.compareTo(current.next.element) >= 0)
                    return false;
            }
        }
        return true;
}
```

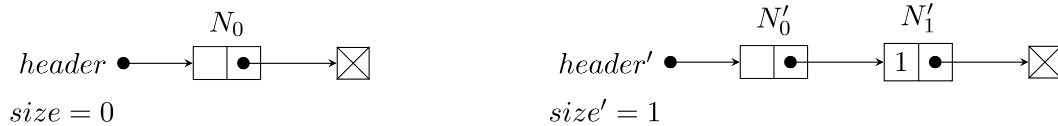
La rutina `repOk()` para `ListToSet`, en primer lugar, chequea si `l` es una lista con nodo centinela acíclica y con cantidad de nodos consistente con el campo `size` de la estructura (de lo cual se encarga la rutina `repOk()` de `SinglyLinkedList`), y si ésto se cumple entonces chequea, si `s` es una lista con elemento centinela acíclica, con cantidad de nodos consistente con el campo `size` de la estructura y estrictamente ordenada (de lo cual se encarga la rutina `repOk()` de `StrictlySortedSinglyLinkedList`).

De la misma manera que `Korat` requiere un predicado imperativo `repOk()`, esta variante (`Korat+`) requiere una rutina llamada `eqClass()`. Esta rutina retorna, dado un candidato válido (es decir, un candidato que satisface el invariante de represen-

tación), la clase de equivalencia a la cual éste pertenece según algún criterio de *test* escogido. De la misma manera que `repOk()`, esta rutina debe ser determinista (por la misma razón que `repOk()` debe serlo).

En lugar de podar (avanzar varios candidatos de una vez) solo cuando `repOk()` falla como lo hace `Korat`, `Korat+` poda el espacio de búsqueda en ambos casos, cuando `repOk()` falla y cuando no. Cuando `repOk()` falla, se avanzan varios candidatos de una vez, basados en el hecho de que si ninguno de los campos accedidos por `repOk()` se cambia, entonces `repOk()` fallará nuevamente. Cuando `repOk()` no falla (devuelve *true*), se procede de la siguiente manera: se ejecuta `eqClass()` y se memorizan los campos accedidos por esta rutina; luego se avanzan varios candidatos de una vez para forzar un cambio en el último campo accedido, ya que si ninguno de los campos accedidos por `eqClass()` cambia, la clase de equivalencia tampoco lo hará y el caso de *test* caerá en una clase de equivalencia que se sabe que ya fue cubierta.

Volviendo a `ListToSet`, para realizar la búsqueda de candidatos válidos, `Korat`, armará los vectores candidatos compuestos de valores para los campos de todos los objetos de las dos listas. En tal búsqueda, `Korat` construye el siguiente par de listas (para la lista y el conjunto respectivamente):



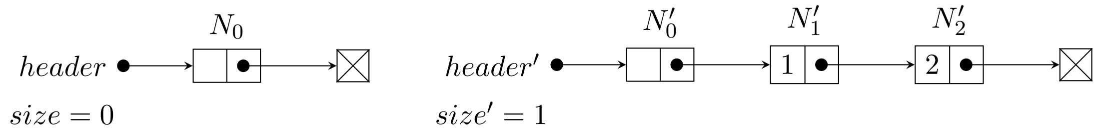
Con el fin de aportar claridad a la explicación, en adelante se llamará a este par de listas (l, s). Claramente (l, s) satisface `repOk()`: l es una lista vacía acíclica en combinación con s, una lista de tamaño 1, acíclica y ordenada. Analicemos como procede `Korat` cuando se encuentra con este candidato.

Debido a que `repOk()` se satisface, todas las partes (alcanzables) de la estructura serán examinadas (en otras palabras, fue necesario mirar toda la estructura para asegurar su validez) cuando éste es ejecutado. `repOk()` inspecciona la estructura en el siguiente orden:

$size'(1)$
$N'_1.next(null)$
$N'_1.element(1)$
...

Es decir, $size'$ (`size` de s) es el último campo accedido, anteriormente $N'_1.next$ y así sucesivamente.

`Korat` construirá el siguiente candidato válido luego de la construcción de varios candidatos inválidos: los primeros candidatos (inválidos) se obtienen luego de avanzar $size'$ reiteradas veces hasta llegar a su valor máximo (cada avance de $size'$ da como resultado un candidato inválido, cuya validez es chequeda mediante `repOk()` y $size'$ es el último campo accedido). Cuando $size'$ llega a su valor máximo (todavía un candidato inválido, cuyos últimos dos campos accedidos por `repOk()` son $N'_1.next$ y $size'$ en ese orden) éste se reseateado a cero y se avanza $N'_1.next$. En este punto se considera N'_0 como posible valor para $N'_1.next$, lo que resulta en una lista cíclica. Lo mismo sucede con $N'_1.next = N'_1$. Por último, se considera N'_2 como valor de $N'_1.next$, lo que luego de "arreglar" $size$ y el orden de los elementos terminará en un candidato válido, en particular el candidato compuesto por la lista l (la cual es vacía) en combinación con el conjunto que contiene dos elementos:



Es decir, debido a la manera en que `repOk()` está escrito, `Korat` producirá todos los conjuntos válidos más grande (teniendo en cuenta el orden en el cual `Korat` los produce) que s en combinación con la lista vacía l antes de avanzar l, es decir antes de producir listas no vacías.

Ahora analicemos como `Korat+` se comporta. El criterio descripto como parte del Ejemplo 3.2 podría ser implementado mediante una rutina, llamada `eqClass()`, de la siguiente manera:

```
public BigInteger eqClass() {
    boolean[] classes = new boolean[4];
```

```
    classes[0] = list.isEmpty();
    classes[1] = list.noReps();
    classes[2] = list.sorted();
    classes[3] = set.isEmpty();
    return getClass(classes);
}
```

Donde `getClass()` traslada un arreglo de valores lógicos a un entero único. De acuerdo al criterio de *test* descripto arriba, este par de listas corresponde a la clase de equivalencia $\langle T, F, T, F \rangle$, es decir, la primera lista (*list*) es vacía, sin elementos repetidos y ordenada, mientras la segunda (*set*) es no vacía. A fin de determinar la clase de equivalencia para este caso de *test*, las partes de la estructura que son examinadas cuando `eqClass()` es ejecutado son:

$N'_0.next$
$header'(N'_0)$
$N_0.next$
$header(N_0)$

siendo $N'_0.next$ el último campo accedido, $header'$ el anterior, y así sucesivamente. Por lo tanto si ninguno de estos campos es modificado, el candidato producido corresponderá a la misma clase de equivalencia que el candidato corriente (l, s), la cual ya está cubierta por este candidato válido.

Por ende, `Korat+` poda la búsqueda avanzando el último campo accedido, $N'_0.next$, el cual ya ha llegado a su máximo valor dentro de su dominio (debido a la regla "como máximo un objeto no tocado"). Luego `Korat+` intenta avanzar $header'$, el cual también está en su máximo valor, entonces continua considerando valores más grandes para $N_0.next$. Como se puede notar `Korat+` evita generar muchos conjuntos no vacíos, los cuales en combinación con la lista vacía caerían en una clase equivalencia ya cubierta. Por ejemplo, si nuevamente se establece que ambas listas tienen entre 0 y 5 nodos, y los valores enteros van desde 1 hasta 4, entonces `Korat+` construye 6798 candidatos (de los cuales 682 son válidos), evitando la construcción 1.268.179 candidatos (de los cuales 5456 son válidos, pero cubren clases ya cubiertas) que `Korat` hubiera mirado.

En la siguiente tabla, se pueden apreciar algunos de los resultados obtenidos para este ejemplo, los cuales exhiben que, mediante la poda descripta, se reduce notablemente el tamaño de las *suites* generadas, el espacio de búsqueda y consecuentemente

el tiempo utilizado durante la búsqueda. Este caso de estudio se retoma en el Capítulo 6, donde se muestran los resultados experimentales completos para el mismo.

Cotas	***K***	***Tiempo K***	***K+***	***Tiempo K+***	***CC***
0-4,1-3	14.679 (320)	0,422s	679 (80)	0,269s	10
0-5,1-4	1.274.977 (5456)	2,39s	6798 (682)	0,395s	10
0-6,1-5	197.651.224 (124992)	329,809s	89.650 (7812)	0,817s	10

Cuadro 3.1: Evaluación `Korat (K)` y `Korat+ (K+)` para `ListAsSet`. Se muestra, en cada caso, espacio de búsqueda y entre paréntesis el tamaño de la *suite*.

3.4. Sobre la consistencia de esta implementación

En esta sección se pretende discutir acerca de la completitud de este enfoque (implementado como una variante de `Korat`) con respecto a cobertura de clases de equivalencia, es decir:

Sean $S+$ y S las *suites de test* generadas por `Korat+` y `Korat` respectivamente para cierta clase de objetos C y cierta cota k, sea $Crit$ el criterio de cobertura aplicado sobre la rutina bajo prueba, toda entrada e tal que $e \in S$ y $e \notin S+$, pertenece a alguna de las clases de entradas viables de $S+$ con respecto a $Crit$. Es decir, toda entrada válida dentro del espacio de estados "podados" por `Korat+` corresponde a clases de equivalencia previamente cubiertas (no se eliminan clases de equivalencia viables).

Con la intención de comparar el algoritmo de `Korat+` con el algoritmo estándar de `Korat`, se recuerda el *pseudocódigo* del algoritmo de `Korat` explicado detalladamente en el Capítulo 2:

```
function korat() {
    Vector current = initVector;
    Stack accessedFields = new Stack();
    boolean ok;
    do{
```

```
        (ok, accessedFields) = current.repOk();
        if(ok){
            reportValid(current);
            accessedFields.push(current.reachableFields -
                accessedFields);
        }
        field = accessedFields.pop();
        while (!accessedFields.isEmpty() &&
                current[field] >= nonIsoMax(current,
                    accessedFields, field)) {
            current[field] = 0;
            field = accessedFields.pop();
        }
        if (!accessedFields.isEmpty()) current[field]++;
    } while (current != lastVector && !accessedFields.
        isEmpty())
}
```

Recordar que `repOk()`, se aplica (haciendo abuso de notación) a vectores candidatos y retorna dos cosas: el resultado de la función (un valor lógico indicando si el candidato es, o no, válido) y la pila conteniendo los campos accedidos en su ejecución (`accessedFields`). La búsqueda (*backtracking*) se realiza sobre esta pila (`accessedFields`) de campos accedidos.

`Korat+` también realiza la búsqueda sobre `accessedFields` extrayendo más elementos de esta pila (poda extra), pero no modifica los campos accedidos.

El pseudocodigo de korat+ es el siguiente:

```
function korat+() {
    Vector curr = initVector;
    Stack accessedFields = new Stack();
    boolean ok;
    do{
        (ok, accessedFields) = curr.repOk();
        if(ok){
            reportValid(curr);
            fields.push(curr.Fields - accessedFields);
            (eqClass, eqFields) = curr.eqClass();
            reportEqClass(eqClass);
        }
```

```
        List modified = new List();
        field = accessedFields.pop();
        while (!accessedFields.isEmpty() &&
             curr[field] >= nonIsoMax(curr, accessedFields,
                field)){
            curr[field] = 0;
            modified.add(field);
            field = accessedFields.pop();
        }
        if (!accessedFields.isEmpty()) {
            curr[field]++;
            modified.add(field);
        }
        // extra pruning
        if (ok && (eqFields - modified == eqFields)){
            for each field in modified
                curr[field] = 0
            boolean found = false;
            while (!accessedFields.isEmpty() && !found) {
                field = accessedFields.pop();
                if (eqFields.contains(field))
                    found = true;
                else
                    curr[field] = 0;
            }
            if(found){
                while (!accessedFields.isEmpty() &&
                     curr[field] >= nonIsoMax(curr,
                        accessedFields, field)) {
                    curr[field] = 0;
                    field = accessedFields.pop();
                }
                if (!accessedFields.isEmpty())
                    curr[field]++;
            }
         }
    } while (!accessedFields.isEmpty())
}
```

Como se puede observar en esta descripción en *pseudo-código* de `Korat+`, el monitoreo de los campos accedidos por `eqClass()` es usado para extraer más elementos de la pila de los campos accedidos por `repOk()`, para forzar un cambio en los campos accedidos por `eqClass()` y, por lo tanto causar una poda más profunda en la búsqueda. Para realizar esta poda, el algoritmo necesita calcular la clase de equivalencia para cada candidato válido, monitoreando los campos accedidos en este cálculo (almacenados en `eqFields`). Luego se chequea si el algoritmo estándar del cálculo del "siguiente candidato" ya ha avanzado algunos de los campos accedidos por la rutina `eqClass()` y sino se fuerza tal avance.

Garantizar que esta variante de `Korat` es *sound* es relativamente directo: como ya fue expresado, `Korat+` realiza la búsqueda sobre `accessedFields`, solo que extrae más elementos de esta pila, pero no modifica los campos accedidos (y por lo tanto no altera el orden de búsqueda)

Esta poda más profunda solo omite candidatos válidos que pertenecen a clases de equivalencia ya cubiertas. El siguiente escenario argumenta este hecho: la etapa de poda extra está activa, entonces el candidato previo, referido como v_p, es un candidato válido, debido a que `ok` es `true`; además, el cálculo del siguiente candidato (mediante el algoritmo estándar de `Korat`) no modificó ninguno de los campos accedidos por `eqClass()`. La última etapa de poda modificó el último campo, de acuerdo a `accessedFields`, que aparece en `eqFields`. Sea v un vector candidato válido descartado durante este proceso. Debido a que v fue descartado mediante este proceso de poda "extra", éste cae dentro del espacio de búqueda podado, el cual coincide en sus valores de `eqFields` con v_p y por ende comparte con él la misma clase de equivalencia (debido al determinismo de `eqClass()`). Por lo tanto los candidatos descartados en el proceso de poda extra corresponden a la misma clase de equivalencia que v_p, la cual ya fue cubierta por este caso de *test*.

Capítulo 4

Uso del invariante de representación para reducir el testing exhaustivo acotado

La técnica presentada en el capítulo anterior, incorpora un criterio de *test* en el mecanismo de generación exhaustiva acotada basada en filtrado, con el objetivo de podar la búsqueda de candidatos válidos y obtener *suites de tests* más pequeñas que las exhaustivas acotadas que sastisfagan el criterio de *test* dado. Para lograr esto, dado un método bajo análisis, el usuario (*tester*) es el encargado de proveer (mediante la rutina `eqClass`) un criterio de cobertura de *test* sobre este método.

En este capítulo se presentan un par de técnicas para mejorar el *testing* exhaustivo acotado:

- *generación disjunta*
- *reducción basada en el invariante de representación de suites de tests*

las cuales no requieren información extra del usuario, más que la especificación de las entradas válidas, la cual, en el caso de las técnicas de generación exhaustivas acotadas basadas en filtrado es un requisito. La principal motivación de estas técnicas es explotar la información que brinda la especificación de la rutina bajo análisis, particularmente la precondición, para reducir el tiempo de generación de *suites de tests* exhautivas acotadas (este es el caso de la primera técnica mencionada arriba) y el tiempo necesario para usar *suites* exhaustivas acotadas durante la prueba, es decir, reducir adecuadamente las *suites* generadas (este es el caso de la segunda técnica mencionada).

Como ya se vio en capítulos anteriores, las técnicas de generación exhaustivas acotadas basadas en filtrado utilizan las restricciones que indican si una estructura esta bien formada o no (invariante de representación), para producir posibles candidatos y filtrar aquellos que no cumplen tales restricciones, es decir, los mal formados. Ambas técnicas mencionadas, aprovechan el invariante de representación que caracteriza las entradas válidas del programa bajo análisis, en dos sentidos:

- para obtener invariantes de representación separados caracterizando sub-estructuras disjuntas de las entradas.
- para definir un *criterio de cobertura de caja negra*, que permita particionar las entradas válidas en clases de equivalencia, de acuerdo a como estas entradas 'ejercitan' el código de la implementación imperativa del invariante de representación.

Es importante hacer notar que estas técnicas requieren tener el invariante de representación en forma de una rutina imperativa (`repOk()`) [23]. De todas maneras, éstas también son relevantes en contextos donde se emplean invariantes de representación declarativas, tales como las que se especifican en `Eiffel` o vía lenguajes como `JML` [30] y `Code Contracts` [1]. Los lenguajes declarativos para especificar contratos están, en general, acompañados de ambientes de chequeo de contratos (*run-time checking*), los cuales convierten estos contratos declarativos en ejecutables; el código correspondiente a su evaluación en tiempo de ejecución correspondería a lo que en este trabajo se refiere como `repOk()`.

Este capítulo se organiza como se detalla a continuación: en la Sección 4.1 se presenta la motivación detrás de estos enfoque así como una intución de los mismos, en la Sección 4.2 se presenta en detalle la primera técnica mencionada: *generación disjunta*, mientras que en la Sección 4.3 se describe la técnica de reducción de *suites de tests* propuesta. En ambos casos, se utilizan ejemplos como eje de la presentación.

4.1. Motivación e Intuición

La primera técnica, la cual es llamada *generación disjunta*, tiene como objetivo reducir el tiempo de generación de las *suites de tests*, para los casos en los cuales las entradas del programa bajo análisis están compuestas por porciones alojadas de manera disjunta en el *heap*. Esta técnica consiste en generar las partes que sabemos que son disjuntas de la estructura de manera independiente, para luego combinarlas

obteniendo la estructura original. Para ello, juega un rol importante el invariante de representación, ya que éste es usado para obtener invariantes de representación más pequeños y locales a cada sub-estructura disjunta de la estructura principal.

Esta técnica surge de la siguiente observación: las herramientas que realizan generación exhaustiva acotada en general tienen su eficiencia ligada a la forma en la cual `repOk()` esta implementado. En particular, si las entradas están compuestas de dos sub-estructuras separadas s_1 y s_2, chequear si la estructura está bien formada, chequeando en primer lugar si s_1 está bien formada y luego si s_2 también lo está, puede ser completamente diferente desde el punto de vista de la eficiencia, que chequear primero s_2 y luego s_1. Como ejemplo, suponga que se desea probar la rutina `AddAll(List l)` sobre *Árboles binarios de búsqueda balanceados (AVLTree)*, la cual inserta todos los elementos de l en el árbol sobre el cual es aplicada. Para probar esta rutina es necesario proveer pares de objetos conformados por una lista y un árbol:

```
public Class AddAllAvlTree{
    AvlTree tree;
    SinglyLinkedList list;

    .....
}
```

si se generan objetos de este tipo utilizando la herramienta `Korat`, se podría utilizar algunas de las siguientes implementaciones del invariante de representación (repOK()):

```
public boolean repOK() {
    if(!tree.repOK())
        return false;
    return list.repOK();
}
```

```
public boolean repOK() {
    if(!list.repOK())
        return false;
    return tree.repOK();
}
```

Ambos realizan lo mismo, pero en distinto orden. El `repOk` que se presenta a la izquierda, en primer lugar chequea que el árbol sea válido (de lo cual se encarga la rutina `repOk()` de `AvlTree`), y si esto se cumple, chequea que `list` sea una lista válida (de lo cual se encarga la rutina `repOk()` de `singlyLinkedList`). Por el contrario, la rutina que se presenta a la derecha examina, en primer lugar, la validez de la lista y luego la validez del árbol. Si se generan *suites de tests* exhaustivas acotadas (árboles de hasta 6 nodos, con 2 valores posibles para las claves y listas de tamaño entre 0 y 6, con 2 valores posibles para las claves), utilizando `Korat` y estas dos rutinas `repOk()`, los resultados arrojados difieren abismalmente en el tamaño del espacio de candidatos explorados: en ambos casos, hay 635 objetos válidos (los

`repOk()` son equivalentes), aunque utilizando el primer `repOk()` se exploran 6884 candidatos, mientras que con el `repOk()` de la derecha se exploran 671.896 candidatos. Esto se debe particularmente a las siguientes caracteristicas de las estructuras involucradas en este ejemplo: `AvlTree` tiene restricciones fuertes, lo cual tiene una incidencia en la relación espacio de búsqueda/entradas válidas: el espacio de búsqueda es grande en comparación con la cantidad de entradas válidas encontradas (pocas instancias satisfacen `repOk()`). Por otro lado, en las listas pasa lo contrario: el espacio de búsqueda es pequeño en relación con la cantidad de entradas encontradas válidas. En el `repOk()` de la izquierda se chequea, en primer lugar, que el árbol sea válido. Debido a que `Korat` realiza el *bactracking* sobre los campos accedidos por `repOk()`, en esta configuración si el `repOk()` de `Avltree` resulta en falso se realiza el *backtracking* sólo sobre los campos del árbol (note que los campos de la lista, en este caso, no estan en la pila de campos accedidos), lo cual permite luego de encontrar un árbol válido, (lo cual requirió probablemente recorrer un espacio de búsqueda grande), ponerlo en combinación con posibles listas, y realizar *backtracking* sobre listas hasta encontrar una estructura que satisfaga `repOk()`. Note que en este caso, las listas tienen restricciones más débiles, con lo cual el árbol válido será puesto en combinación con listas que son mayormente válidas. Por otro lado, el `repOk()` de la derecha, requiere en primer lugar encontrar una lista válida, para luego ponerla en combinación con posibles árboles, los cuales tienen la particularidad de ser pocos estructuras válidas en comparación con el tamaño del espacio de búqueda, es decir, la lista válida será combinada con muchos árboles inválidos hasta dar con uno válido, la cual resulta en un espacio de búsqueda más grande que en el caso de la izquierda.

El enfoque que se propone tiene varias ventajas sobre el estándar: si la generación de las sub-estructuras es independiente, el impacto en la eficiencia de la generación de como está escrito `repOk()` es reducido. Además las sub-estructuras disjuntas podrían ser generadas de manera paralela, e incluso cuando éstas son generadas secuencialmente se evita el problema de reconsiderar cada estructura válida de la segunda sub-estructura por cada estructura inválida de la primera sub-estructura.

Por otro lado, la segunda técnica, *reducción basada en repOk() de suites de tests*, ha sido pensada con el propósito de reducir el tamaño de las *suites de tests* exhaustivas acotadas, y como consecuencia el tiempo consumido durante la prueba, usando estas *suites* como entradas. El siguiente análisis motiva el desarrollo de esta técnica: el código del `repOk()` provee información acerca de la "variabilidad" de las entradas, si se considera un criterio de cobertura de caja blanca sobre el código `repOk()`, es posible definir una relación de equivalencia entre las entradas válidas. Dos entradas de *test* válidas serán consideradas equivalentes si éstas ejercitan el

código de `repOk()` de manera similar, de acuerdo a algún criterio de cobertura de caja blanca. Estas equivalencias entre casos de *test* son aprovechadas para filtrar entradas de la *suite de tests*, eliminando esos *tests* que son equivalentes a algún otro presente en la *suite*. Básicamente, esta técnica involucra la definición de un *criterio de cobertura de caja negra*, definido en términos de un *criterio de cobertura de caja blanca con respecto al invariante de representación de las entradas del programa bajo prueba*. Concretamente, el nuevo criterio definido especifica cuando dos entradas diferentes pueden ser consideradas equivalentes, sin tener en cuenta el código de la estructura bajo prueba (por ellos es caja negra), considerando solo la estructura de la rutina `repOk()`. Mantener las clases de equivalencia independientes del código bajo análisis tiene una ventaja importante de subrayar: la *suite de tests* obtenida podría ser usada para probar cualquier rutina donde las entradas estén caracterizadas por el mismo invariante de representación, contrapuesto al *testing* de caja blanca en donde las entradas generadas son adecuadas para probar una rutina particular.

En resumen, estos enfoques contribuyen al *testing* exhaustivo acotado de la siguiente manera: en primer lugar, mediante la generación separada de sub-estructuras disjuntas se reduce el espacio de candidatos a ser considerados durante la generación, y por lo tanto el tiempo invertido en el proceso de generación de entradas. Esta generación separada es aplicable en casos en los cuales las entradas están compuestas de sub-estructuras disjuntas. Cabe destacar que esta técnica produce *suites de tests* exhaustivas acotadas, es decir, no se afecta las *suites* producidas en relación a la generación exhaustiva acotada estándar, sino que se las genera de manera más eficiente. Por otro lado, la otra técnica introducida apunta a reducir el tiempo consumido durante el *testing*, reduciendo el tamaño de las *suites* exhaustivas acotadas ya generadas, de acuerdo con un criterio de cobertura de caja blanca sobre el código del `repOk()`, que caracteriza las entradas de la rutina bajo análisis.

4.2. Generación independiente de sub-estructuras disjuntas.

En esta sección se presenta la técnica, mencionada en las secciones iniciales de este capítulo, para construir *suites de tests* exhaustivas acotadas eficientemente, reduciendo el espacio de estados explorados durante la generación, la cual es aplicable en casos en los cuales se desean generar instancias de una estructura compleja, compuesta de partes alojadas en porciones disjuntas del *heap*. En primer instancia, se presenta un caso de estudio que ilustra la motivación principal de este enfoque.

El mismo es utilizado más adelante para explicar en detalle la implementación de la técnica.

4.2.1. Un caso testigo: `NodeCaching LinkedList`

`NodeCaching LinkedList` es una estructura de datos que acompaña la colección de *Apache Commons*, y corresponde a una implementación de listas que intenta reducir la creación de objetos, y el posterior uso del *garbage collector*, manteniendo una lista de los nodos borrados en una *cache*. Básicamente, la estructura consiste de una *lista circular doblemente encadenada*, conteniendo los elementos reales de la lista, y una *lista simplemente encadenada* representando la *cache*. La siguiente es la definición estructural de `NodeCachingLinkedList`:

```
public class NodeCachingLinkedList {
    private LinkedListNode header;
    private int size;

    private LinkedListNode firstCachedNode;
    private int cacheSize;
    private int maximumCacheSize;
    ...
}

public class LinkedListNode {
    Object value;
    LinkedListNode previous;
    LinkedListNode next;
    ...
}
```

Suponiendo que se requiere probar una rutina que manipula instancias de esta clase, y que se utilizará *testing* exhaustivo acotado para hacerlo. Es decir, se desea probar la rutina para todas las entradas válidas dentro de ciertas cotas. Básicamente, estos objetos están compuestas de la cabeza de la lista circular (`header`) y la cabeza de la lista *cache* (`firstCachedNode`). Estas dos estructuras son *disjuntas*, es decir, para cualquier instancia de la clase `NodeCachingLinkedList`, ningún nodo está en ambas listas.

Considere el siguiente escenario de generación: se generaron todas las instancias de `NodeCachingLinkedList` conteniendo hasta 8 nodos, con tamaño hasta 3 para

la lista circular y hasta 4 para la lista *cache*, usando una herramienta de generación exhaustiva acotada. Se puede notar que el nodo adicional, con respecto a la suma de los tamaños de las dos listas, tiene que ver con el hecho de que la lista circular doblemente encadenada contiene un elemento centinela (la cabeza de la lista), el cual no es considerado en el correspondiente campo `size`. En esta generación, usando la herramienta `Korat`, 13.164 estructuras candidatas son exploradas, de las cuales 450 son válidas (esto es considerando un único valor posible para los campo claves de ambas listas). Muchas de las estructuras visitadas inválidas están compuestas de una lista circular doblemente encadenada válida y una lista *cache* inválida, o viceversa. La Figura 4.1 muestra un candidato visitado durante la generación exhaustiva acotada con estas caraterísticas, una lista circular bien formada (de tamaño 3) en combinación con una cache cíclica (ciclo entre los nodos N_5 y N_6). Esta clase de candidatos inválidos, es muy común en estructuras compuestas por sub-estructuras disjuntas. Además, la cantidad de estas estructuras se incrementa cuando las cotas lo hacen, siendo una de las causas de la ineficiencia en la generación exhaustiva acotada.

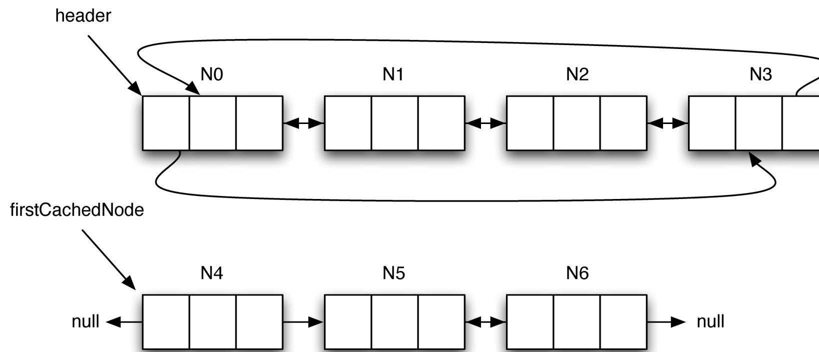


Figura 4.1: Instancia mal formada de `NodeCachingLinkedList`.

4.2.2. Descripción precisa del enfoque

La técnica que se propone esta basada en la observación de que, durante el proceso de generación, muchas estructuras mal formadas son producidas combinando partes bien formadas con otras mal formadas. La técnica propuesta reduce el espacio de estados de candidatos, evitando esta clase de estructuras mal formadas,

mediante la construcción individual de sub-estructuras que son disjuntas en el *heap*, y la posterior combinación de ellas, para obtener una posible estructura bien formada. De esta manera solo se combinan sub-estructuras bien formadas. La mayoria de los enfoques de generación, en casos en donde la estructura esta compuesta por sub-estructuras separadas, la búsqueda de instancias válidas se realiza sobre una representación lineal de la estructura, sin distinción de sub-partes, es decir mirando la estructura como un todo, lo cual resulta en muchas instancias inválidas que podrían ser evitadas si se distinguieran las porciones (o sub-estructuras) disjuntas.

Con el fin de describir más precisamente la técnica, sea C una clase para la cual deseamos construir una *suite de tests* exhaustiva acotada con cota k, sea `repOk()` la implementación imperativa del invariante de representación de C. La técnica propuesta consiste de los siguientes pasos:

- Identificar sub-estructuras $s_1, s_2, ..., s_n$ de las instancias de clase C las cuales son disjuntas, en el sentido que están siempre alojadas en partes independientes del *heap*, y las cuales, juntas conforman C (es decir, $s_1, s_2, ..., s_n$ son una partición de C). En donde si se entienden a las estructuras como grafos descriptos por un predicado (`repOk()`), las sub-estructuras pueden ser vistas como sub-grafos descriptos por un predicado (`repOk'()`), tal que si una sub-estructura s' no satisface `repOk'()`, toda estructura s que contenga a s' no satisface `repOk()`.

- Definir estructuras de datos auxiliares $S_1, S_2, ..., S_n$ que representan las sub-estructuras disjuntas previamente definidas.

- Utilizando el `repOk`, obtener invariantes de representación $repOk_1, repOk_2, ..., repOk_n$, tal que, si una instancia c de C satisface `repOk`, entonces las sub-instancias c_{s_i} de c, correspondientes a la sub-estructura s_i satisfacen $repOk_i$.

- Generar *suites* exhaustivas acotadas de manera independiente para $S_1, S_2, ..., S_n$ con cota k.

- Combinar instancias de las *suites* exhaustivas acotadas previamente generadas, para construir instancias de C.

- Filtrar esas instancias de C que no satisfacen `repOk`.

Si se tiene la certeza de que algunas partes diferentes de la estructura son siempre disjuntas, como es el caso delEejemplo presentado en 4.2.1, es posible tomar porciones del código del `repOk()` de la estructura completa, para obtener invariantes de

representación más pequeños e independientes que caractericen las partes disjuntas de la estructura. Estas rutinas `repOk()` de las estructuras más pequeñas son usadas para generar de manera independiente las sub-estructuras, las cuales, luego son combinadas para formar la estructura completa.

4.2.3. Como determinar sub-estructuras diyuntas

De la descripción dada en la sección anterior surge la siguiente cuestión técnica: ¿Cómo se puede saber si toda instancia de la estructura está siempre compuesta de sub-estructuras disjuntas?. Esta información se podría obtener de diferentes fuentes, la más obvia es desde el desarrollador, es decir, el desarrollador está a cargo de detectar cuáles partes de la estructura podrían ser generadas de manera independiente, y proveer esta información como entrada del proceso de generación disjunta. Por ejemplo, para el caso de `NodeCachingLinkedList`, la información requerida es que la lista *cache* referenciada por el campo `firstCachedNode` is disjunta de la estructura referenciada por el campo `header`, es decir, la *lista circular doblemente encadenada*. Otra forma de obtener esta información es desde la *definición de las cotas*. Básicamente, proveer las cotas para realizar la generación exhaustiva acotada consiste en construir *dominios* para los diferentes campos de la estructura a ser generada. Si se proveen dominios disjuntos para campos del mismo tipo, entonces se puede utilizar esa información para procesar el `repOk()` de la manera descripta. A modo de ejemplo, supongamos que las cotas sobre el número de objetos, y los posibles valores en los campos, para ser usados en la construcción de instancias de la estructura, están dadas usando una notación programática. Siguiendo con el ejemplo de `NodeCachingLinkedList`, se podrían definir las cotas como sigue:

```
...
Domain EntriesList = createDomain(LinkedListNode , numEntry)
EntriesList.setNullAllowed();
Domain EntriesCache = createDomain(LinkedListNode , numEntry)
EntriesCache.setNullAllowed();
setScope(header, EntriesList);
setScope(header.*next, EntriesList);
setScope(header.*previous, EntriesList);
setScope(firstCachedNode, EntriesCache);
setScope(firstCachedNode.*next, EntriesCache);
setScope(firstCachedNode.*previous, EntriesCache);
...
```

En este caso, se definen dos dominios: `EntriesList` y `EntriesCache`, ambos compuestos de `numEntry` instancias de `LinkedListNode` (más `null`). De esta manera, se definen cotas para diferentes campos, lo cual, expone claramente que los campos de las listas referenciadas por `header` y `firstCachedNode` no comparten objetos. Esta información podría ser explotada para detectar porciones disjuntas de la estructura.

Finalmente, esta información podría estar presente en el `repOk()`, explícita o implícitamente. Por ejemplo, para el caso de `NodeCachingLinkedList`, el invariante de representación podría incluir el chequeo de que la *cache* y la lista circular no comparten nodos.

Este problema, aunque es muy importante, está fuera del alcance de este trabajo. En la evaluación de la técnica (presentada en el capítulo 6), se asume que la información acerca de sub-estructuras disjuntas es provista por el desarrollador.

4.2.4. Descomposición del `repOk()`

Otro punto importante que aparece en la descripción de este enfoque, es el proceso automático sobre el `repOk()`, para obtener rutinas `repOk()` más pequeñas e independientes que caractericen las sub-estructuras disjuntas. Esto no es un problema trivial, y es claro que en muchos casos el `repOk()` original podría no estar completamente modularizado, e incluso cuando, tomando porciones del `repOk()` se pueden obtener invariantes de representación 'locales' a cada sub-estructura, podría haber condiciones de buena formación 'globales' a la estructura original que tienen

que ver con la relación entre las sub-estructuras disjuntas (note que este invariante de representación 'global' está incluida en los pasos descriptos de la técnica). Por ejemplo, si `NodeCachingLinkedList` es usada para implementar un conjunto, y en la *cache* se chequea pertenencia antes de la inserción (de esta manera se evita crear un nodo nuevo), entonces parte del `repOk()` chequeará que la *cache* y la lista circular no comparten valores almacenados (a parte de que no comparten nodos), es decir, en este ejemplo, de la descomposición se obtienen no sólo `repOk()` locales a cada sub-estructura sino también un `repOk()` "global" (el cual chequea que las dos listas no comparten valores). En muchos casos este proceso de descomposición es directo, por ejemplo, si se considera una rutina que recibe como parámetro más de una estructura. Algunos ejemplos de esta situación son la rutina `merge` sobre binomial heaps (la cual toma dos binomial heaps), operaciones de intersección/unión sobre conjuntos, e incluso rutinas de consulta, inserción y eliminación, las cuales usualmente toman dos parámetros, la colección a ser modificada o consultada y el valor a insertar, borrar o buscar. En todos estos casos, la entrada es compuesta, y el `repOk()` para las entradas es la conjunción de los `repOks` para las estructuras parámetros.

Nuevamente, este problema está fuera del alcance de este trabajo. Con el objetivo de realizar los experimentos, se procesa `repOk()` tomando porciones (*slicing*) de él con respecto a la parte de la estructura visitada por cada sentencia en la invariante de representación. De esta manera se obtiene, además de los `repOks` 'locales', un `repOk()` 'global' compuesto de sentencias en las cuales más de una sub-estructura disjunta podría estar involucrada.

Retomando el ejemplo de `NodeCachingLinkedList`, una implementación del invariante de representación es la siguiente:

```
public boolean repOK() {
   if (header == null) return false;
   if (header.next == null) return false;
   if (header.previous == null) return false;
   if (cacheSize > maxCacheSize) return false;
   if (MAXIMUM_CACHE_SIZE != 20) return false;
   if (size < 0) return false;
   int cyclicSize = 0;
   Node n = this.header;
   do{
       cyclicSize++;
       if (n.previous == null) return false;
       if (n.previous.next != n) return false;
       if (n.next == null) return false;
       if (n.next.previous != n) return false;
       if (n != null)  n = n.next;
   } while (n != header && n != null);
```

```
    if (n == null) return false;
    if (size != cyclicSize - 1) return false;
    int acyclicSize = 0;
    Node m = firstCachedNode;
    Set<Node> visited = new HashSet<Node>();
    visited.add(firstCachedNode);
    while (m != null){
        acyclicSize++;
        if (m.previous != null) return false;
        if (m.value == null) return false;
        m = m.next;
        if (!visited.add(m)) return false;
    }
    if (cacheSize != acyclicSize) return false;
        return true;
}
```

Este `repOk()` chequea si una estructura es una instancia de `NodeCachingLinkedList` bien formada, chequeando si se cumplen las siguientes condiciones:

- la lista es una lista circular doblemente encadenada bien formada,
- el tamaño de la *cache* no excede el tamaño máximo,
- el tamaño de la lista es consistente con el número de nodos en ella,
- la lista *cache* es acíclica, y
- el tamaño de la *cache* es consistente con la cantidad de nodos en ella.

Este `repOk()` podría ser dividido en dos: una parte que se refiere sólo a la lista circular (y su correspondiente campo `size`), y otra que se refiere s 'olo a la lista *cache* (y su correspondiente campo *size*). Estos son los dos `repOk()` locales resultantes:

```java
public boolean repOK() {
    if (header == null)
        return false;
    if (header.next == null)
        return false;
    if (header.previous == null)
        return false;
    if (size < 0)
        return false;
    int cyclicSize = 0;
    Node n = header;
    do{
        cyclicSize++;
        if (n.previous == null)
            return false;
        if (n.previous.next != n)
            return false;
        if (n.next == null)
            return false;
        if (n.next.previous != n)
            return false;
        if (n != null)
            n = n.next;
     }while(n!=header && n!=null);
     if (n == null)
        return false;
     if (size != cyclicSize -1)
        return false;
     return true;
}
```

```java
public boolean repOK() {
    if (cacheSize > maxCacheSize)
       return false;
    if (MAXIMUM_CACHE_SIZE != 20)
        return false;
    int acyclicSize = 0;
    Node m = firstCachedNode;
    Set<Node> visited = new HashSet<Node
        >();
    visited.add(firstCachedNode);
    while (m != null){
        acyclicSize++;
        if (m.previous != null)
            return false;
        if (m.value == null)
            return false;
        m = m.next;
        if (!visited.add(m))
            return false;
    }
    if (cacheSize!= acyclicSize)
        return false;
    return true;
}
```

4.2.5. Sobre la correctitud de esta técnica

En esta sección se discute la correctitud de este enfoque, el cual es *sound* y completo con respecto a generación exhaustiva acotada, en el siguiente sentido:

"Una instancia es producida por generación exhaustiva acotada para cota k si y sólo si, esta instancia es producida por la técnica de generación separada, también para cota k"

La prueba de esto es relativamente directa. Se puede notar que la prueba de *soundness* es trivial debido al último paso en la técnica de generación disjunta, el cual, filtra de la *suite de test* exhaustiva acotada producida por generación disjunta esas instancias que no satisfacen `repOk()`. Acerca de la prueba de completitud, el argumento es el siguiente: Sea c una instancia de clase C que satisface `repOk()`, y está dentro de la cota k. Para cada sub-estructura S_i de C, hay una instancia c_{s_i} incluida en c, también dentro de la cota k (debido a que ésta es una sub-instancia de

c, la cual está acotada por k). Además, debido a que c satisface `repOk()`, c_{s_i} satisface $repOk_i$ (ver paso 3 en la técnica de generación disjunta). Entonces, la instancia c_{s_i} es producida como parte de la generación exhaustiva acotada para la sub-estructura S_i. Debido a que esto se cumple para cada sub-estructura S_i, $c_{s_i}, ..., c_{s_n}$ se construyen mediante la correspondiente generación exhaustiva acotada 'local'. La combinación de $c_{s_i}, ..., c_{s_n}$ se corresponde con c ($S_1, ..., S_n$ son una partición de C), la cual, satisface `repOk()`, y por lo tanto es producida mediante la técnica de generación disjunta.

4.3. Reducción de *suites de test* exhaustivas acotadas usando criterios sobre `repOk`

El *testing* exhaustivo acotado muchas veces obliga a usar cotas muy pequeñas, debido a que el tamaño de las *suites* exhaustivas acotadas crece rápidamente cuando éstas se incrementan, y por lo tanto el uso de las mismas durante la prueba, se vuelve inviable. En esta sección, se explica la técnica mencionada en los inicios de este capítulo, para reducir *suites de tests* exhaustivas acotadas. Esta técnica asume que se tiene una implementación imperativa del invariante de representación de las entradas a ser producidas; es por ello que el enfoque es más apropiado en combinación con técnicas basadas en filtrado para la generación de *test* (para los cuales la invariante de representación es un requisito).

4.3.1. Ejemplo guía: *Árboles Binarios*

Sea la siguiente clase Java, una definición estructural de árboles binarios de enteros:

```
public class BinaryTree {
    private Node root;
    private int size;

    ...
}
```

```
public class Node {
   private int key;
   private Node left;
   private Node right;

   ...
   // setters and getters
   // of the above fields
   ...
}
```

El invariante de representación de esta clase debería chequear los siguientes puntos:

- la estructura que comienza en `root` es un árbol, es decir es acíclico y con un único padre para cada nodo (excepto la raíz) alcanzable desde la raíz,
- su tamaño es consistente, es decir, el valor del campo `size` se corresponde con la cantidad de nodos en el árbol.

La siguiente es una implementación del mismo (tomada de la distribución de `Korat` [11]):

```
public boolean repOK() {
    if (root == null) return size == 0;
    Set visited = new HashSet();
    visited.add(root);
    LinkedList workList = new LinkedList();
    workList.add(root);
    while (!workList.isEmpty()) {
        Node current = (Node) workList.removeFirst();
        if (current.getLeft() != null) {
            if (!visited.add(current.getLeft()))
                return false;
            workList.add(current.getLeft());
        }
        if (current.getRight() != null) {
            if (!visited.add(current.getRight()))
                return false;
            workList.add(current.getRight());
        }
    }
    return (visited.size() == size);
}
```

Se plantea el siguiente escenario (el cual se retoma en las secciones siguientes): Dada varias rutinas que reciben como parametro un árbol binario, se requiere generar objetos de esta clase para probar tales rutinas.

4.3.2. Definición de un criterio de caja negra basado en `repOk()`

Con el objetivo final de describir la técnica, aquí se explica como definir un criterio de cobertura de *test* (o una familia de criterios) usando el código de `repOk()`. Continuando con el escenario de *testing* enunciado en la sección anterior, como un criterio de *caja negra* para probar estas rutinas, podemos definir una partición de todas los árboles binarios válidos de acuerdo a la forma en que las diferentes

estructuras ejercitan la rutina `repOk()`. La motivación es básicamente que los *test* que "ejercitan" el código del `repOk()` de la misma forma pueden ser considerados similares, y por lo tanto estarían en la misma clase de equivalencia.

En primer lugar, es necesario definir que significa "ejercitan en forma similar". Esto puede ser realizado, en principio, eligiendo un criterio de cobertura de *caja blanca*, para ser aplicado al código de `repOk()`. Por ejemplo, podría considerarse el criterio de *cobertura de decisión* sobre `repOk()`; en este caso, 2 entradas del código bajo prueba son consideradas equivalentes si evalúan todos los puntos de decisión de `repOk()` en los mismos valores. Como ejemplo, supongamos que se ha generado, con algún mecanismo, una *suite de tests* exhaustiva acotada de tamaño k, que contiene las siguientes árboles:

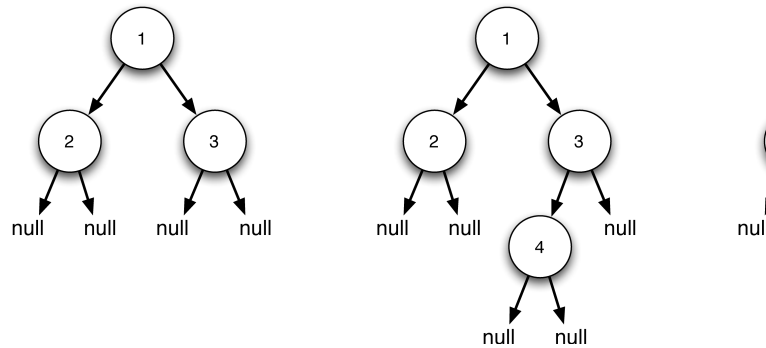


Figura 4.2: Ejemplo de Árboles Binarios. El primero y el segundo ejercitan `repOK` de la misma manera, de acuerdo a cobertura de decisión.

el primero y el segundo deberían ser considerados equivalentes, pero ninguno de ellos debería ser considerado equivalente al tercero. Esto se debe a que el tercer árbol nunca evalúa en verdadero el predicado `current.getRight() != null`, mientras que si lo hacen los dos primeros. Es de importancia subrayar que los puntos de decisión del código bajo análisis (rutinas que toman como parámetro árboles binarios), no se toman en cuenta a la hora de definir este criterio.

En este ejemplo, se eligió uno de los criterio de cobertura de caja blanca más simples para ser aplicado a `repOk()`; si se elige un criterio más sofisticado, se obtiene una relación de equivalencia de granularidad más fina sobre el espacio de entradas válidas. Por ejemplo, el criterio de cobertura de *decisión con conteo (CDC)*, el cual tiene en cuenta el número de veces que cada decisión en el programa evalúa en verdadero y en falso, se define de manera precisa como sigue:

Dado un programa bajo prueba P y dos entradas c_1 y c_2 para P, c_1 y c_2 son equivalentes de acuerdo a P bajo el criterio CDC si y sólo si, para cada punto de decisión cond *en P, el número de veces que* cond *se evalúa en verdadero (respectivamente en falso), cuando se ejecuta P con c_1 como entrada, es igual al número de veces que* cond *se evalúa en verdadero (respectivamente en falso), cuando P se ejecuta con c_2 como entrada.*

Bajo este criterio, los dos primeros árboles (considerados equivalentes bajo el criterio de cobertura de decisión) son considerados diferentes, es decir, *CDC* los distingue y por lo tanto éstos pertenecen a distintas clases de equivalencias de acuerdo a *CDC*.

En general, se puede notar que cualquier criterio de cobertura de caja blanca induce una partición del espacio de entradas del programa bajo prueba, donde cada clase en la partición captura algún camino o decisión expresada como una restricciones sobre la entrada. Dado un programa bajo prueba P, un criterio *Crit*, y una entrada c, se denota con $[\![c]\!]^P_{Crit}$ la partición a la que c pertenece, es decir, el conjunto de todas las entradas que ejercitan el código de P en la misma forma que lo hace c, de acuerdo con el criterio *Crit*. Sea C una clase equipada con su correspondiente `repOk()`, y sea *Crit* el criterio de cobertura de caja blanca seleccionado. Dado dos objetos válidos c_1 y c_2 de C, es decir, dos objetos que satisfacen el invariante de representación de C, se dice que c_1 es equivalente a c_2 (de acuerdo a `repOk()` bajo *Crit*), *si y sólo si* se cumple:

$$[\![c_1]\!]^{\texttt{repOK}}_{Crit} = [\![c_2]\!]^{\texttt{repOK}}_{Crit}$$

4.3.3. Mecanismo de Reducción

Luego de decidir que criterio de caja blanca será aplicado a `repOk()`, éste se puede utilizar para reducir *suites de tests* exhaustivas acotadas. Teniendo generada una *suite de tests* exhaustiva acotada de tamaño K para usar en la etapa de *testing*, y bajo la suposición que no se poseen los recursos suficientes para analizar el programa bajo prueba con todas estas K entradas, en su lugar sólo se cuenta con recursos para probar el programa con una porción de esta *suite*. Sea n al tamaño esperado de la *suite de tests* reducida (por ejemplo, n podría ser $K/10$ si se espera una reducción de un orden de magnitud). Una reducción de esta *suite de tests* puede ser conseguida siguiendo los pasos detallados a continuación:

- Determinar el número de clases de equivalencia cubiertas por las entradas de la *suite de tests* exhaustiva acotada de acuerdo al criterio de caja blanca

seleccionado (aplicado sobre `repOk()`), llamemoslo c.

- Determinar un límite para la cantidad deseada de casos de *test* que se quiere obtener por cada clase de equivalencia q, llamemoslo max_q. Por ejemplo, podríamos tomar $max_{eq} = n/c$, es decir, dividir el tamaño esperado de la *suite de tests* reducida (n) por el número de clases de equivalencia (c).

- Procesar la *suite de tests* exhaustiva para dejar sólo max_q entradas por cada clase de equivalencia q.

Como se mencionó más arriba, el resultado de aplicar el proceso descripto, depende fuertemente del criterio de cobertura de caja balnca seleccionado. Además, este proceso también depende fuertemente de la estructura del `repOk()`, es decir de su implementación.Por ejemplo, una sentencia `if-then-else` con una condición compuesta podría ser escrita con sentencias `if-then-else` anidadas con condiciones atómicas; estos programas estructuralmente diferentes pero semánticamente equivalentes, producirían, para el mismo criterio de caja blanca, diferentes clases de equivalencia, y por lo tanto se obtendrían diferentes *suite de tests* reducidas.

4.3.4. Sobre la correctitud de esta técnica

De manera opuesta a la técnica presentada en la Sección 4.2, la técnica de reducción de *suites de tests* exhaustivas acotadas basada en `repOk` es *sound* pero no *completa*, con respecto a la generación exhaustiva acotada. Nuevamente *soundness* es relativamente trivial, debido a que las reducciones basadas en `repOk` se realizan sobre *suites de tests* exhaustivas acotadas previamente generadas, filtrando algunos casos de *test* de acuerdo a algún criterio de cobertura de caja blanca sobre el código de `repOk()`, y a un número máximo de entradas de *test* consideradas para cada clase de equivalencia. La técnica es también *completa* con respecto a cobertura de clases de equivalencia (cubre todas las clases de entradas viables dentro de la cota establecida):

Cada clase de equivalencia q cubrible dentro de una cota k será cubierta por las suites reducidas, siempre y cuando el valor max_q (máximo número de tests para la clase de equivalencia q) sea mayor que cero para todo clase de equivalencia q.

Con respecto a esto, el razonamiento es el que sigue: Sea B una *suite* exhaustiva acotada para la clase C con cota k. Sea $Crit$ un criterio de cobertura de caja blanca a

ser aplicado sobre `repOk()`, el invariante de representación para la clase C. Se asume que existe una clase de equivalencia q para la cual su máximo es mayor a cero, la cual es cubierta por B, pero no cubierta por la *suite* reducida B_{Crit}. Entonces existe una instancia c de C la cual esta presente en B, la cual cubre la clase q y obviamente no está en B_{Crit}. Además, debido a que q no es cubierta por B_{Crit}, no hay ninguna entrada en B_{Crit} correspondiente a la clase de equivalencia q. Debido a que c fue dejado afuera de B_{Crit}, éste fue eliminado durante el proceso de reducción de la *suite* sobre B. Esto sólo puede suceder si el límite max_q de instancias fue alcanzado en la reducción, lo que significa que ya tenemos entradas en B_{Crit} que cubren q (debido a que max_q es positivo), de esta manera se llega a una contradicción. Por lo tanto, las reducciones basadas en `repOk()` son completas con respecto a cobertura de clases de equivalencia.

Capítulo 5

Uso de *cotas ajustadas* en la generación exhaustiva acotada

Este capítulo pretende estudiar si las *cotas ajustadas* [14], las cuales han sido muy útiles en el contexto de análisis de programas Java, podrían contribuir a aumentar la escalabilidad en la generación exhaustiva acotada. En primera instancia, en la Sección 5.1 se explica en detalle este concepto, para luego en las siguientes secciones presentar un enfoque para aprovechar las *cotas ajustadas* en la generación exhaustiva acotada, el cual se presenta como una variante de `Korat`. Por último, en la Sección 5.4 se discute una alternativa basada en el uso de invariantes híbridos, es decir, especificaciones que son expresadas, declarativamente, imperativamente o como combinación de predicados imperativos y declarativos. Este último enfoque, es parte de los resultados presentado en [32]

5.1. ¿Qué son las cotas Ajustadas?

En los capítulos anteriores se utilizó el término *Cotas* para referirse al tamaño de cada campo de la entrada involucrada en la generación exhaustiva *acotada*, es decir, el tamaño de los dominios de los campos implicados en la entrada. En este capítulo, este término es utilizado en otro sentido, y por ello, en adelante se habla de *cotas ajustadas* [14]. Esta sección está destinada a describir en detalle este concepto, para lo que en principio se introduce un ejemplo. Considere la siguiente implementación de árboles binarios, utilizada en varias oportunidades en capítulos anteriores:

```
public class BinaryTree {
    private Node root;
    ...
}
```

```
public class Node {
   private int key;
   private Node left;
   private Node right;
  ...
}
```

Se consideran instancias válidas de esta clase aquellas estructuras que cumplan con ser un árbol acíclico y con un único padre para cada nodo (excepto la raíz) alcanzable desde la raíz.

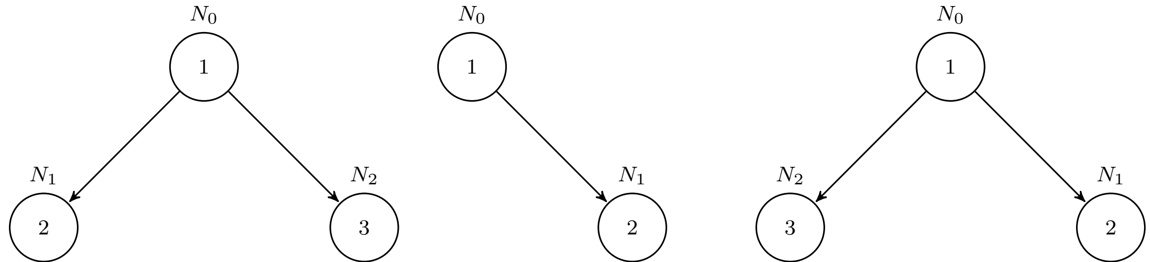
Asumiendo que los nodos del árbol (de clase `Node`) tienen identificadores N_0, N_1, N_2, los campos `left` y `right` podrían interpretarse como relaciones binarias que van de identificadores de nodos a identificadores de nodos:

$$left : \{N_0, N_1, N_2\} \rightarrow \{N_0, N_1, N_2, null\}$$
$$right : \{N_0, N_1, N_2\} \rightarrow \{N_0, N_1, N_2, null\}$$

si existe alguna instancia de árbol binario tal que $N_i.left = N_j$ entonces $(N_i, N_j) \in left$

Para evitar instancias simétricas o isomorfas (ver Sección 2.2.1, Capítulo 2), se asume que se cuenta con un *predicado de rotura de simetrías* el cual establece un orden (lexicográfico) entre los nombres de nodos: $N_0 < N_1 < N_2$ y la forma en que estos aparecen en la estructura, por ejemplo, forzando la asignación de nombre a nodos, de menor a mayor y de manera consecutiva, comenzando desde la raíz y continuando en un recorrido en anchura de izquierda a derecha (*BFS*).

Para clarificar este concepto consideremos las siguientes instancias de árbol binario de tamaño máximo 3:



por el orden impuesto en los identificadores de los nodos, sólo las dos primeras instancias son posibles, mientras que la tercera no se corresponde con el orden en recorrido en anchura (de izquierda a derecha) establecido.

El orden impuesto como parte del mecanismo de rotura de simetría, permite acotar superiormente el co-dominio de estas relaciones utilizando la información que dicho orden brinda: por ejemplo, para el caso del campo `left`, se sabe que:

$$(N_0, N_2) \notin left,\ (N_1, N_0) \notin left,\ (N_2, N_0) \notin left\ y\ (N_2, N_1) \notin left$$

es decir, no existen instancias válidas de árboles binarios que respeten el orden impuesto donde $N_0.left = N_2$ ó $N_1.left = N_0$ ó $N_2.left = N_0$ ó $N_2.left = N_1$.

Por lo tanto se puede reescribir la relación `left`, acotando su co-dominio:

$$\begin{aligned} \text{left} = \quad & \{(N_0, N_0), (N_0, N_1), \cancel{(N_0, N_2)}, (N_0, null), \\ & \cancel{(N_1, N_0)}, (N_1, N_1), (N_1, N_2), (N_1, null), \\ & \cancel{(N_2, N_0)}, \cancel{(N_2, N_1)}, (N_2, N_2), (N_2, null)\} \end{aligned}$$

La especificación de las entradas válidas (invariante de representación) también provee información útil que puede ser usada para acotar el co-dominio de `left` (igualmente para `right`) aún más. Para el caso de los árboles binarios, esta especificación establece que las instancias válidas satisfacen la propiedad de *aciclicidad*, por ende:

$$\begin{aligned} \text{left} = \quad & \{\cancel{(N_0, N_0)}, (N_0, N_1), (N_0, null), \\ & \cancel{(N_1, N_1)}, (N_1, N_2), (N_1, null), \\ & \cancel{(N_2, N_2)}, (N_2, null)\} \end{aligned} \quad \equiv \quad \begin{aligned} \text{left} = \quad & \{(N_0, N_1), (N_0, null), \\ & (N_1, N_2), (N_1, null), \\ & (N_2, null)\} \end{aligned}$$

De una manera mas general y formal, las cotas se definen como sigue [14]:

Sean $f_1, .., f_n$, campos de una estructura C alojada en *heap*, Sea I una especificación que describe las instancias válidas de C, una cota ajustada de cierto tamaño s para un campo $f_i : A- > B$ es una relación binaria $Cota_{f_i} \subseteq (AxB)$ tal que: si existe una instancia de C del tamaño s que satisface I, en la que $a.f_i = b$ entonces $(a, b) \in Cota_{f_i}$

Retomando el ejemplo de árboles binarios, estas *cotas ajustadas* sobre el co-dominio de `left` (igualmente para `right`), eliminan posibles asignación de nombres de nodos a `left`, que llevan a construir árboles inválidos.

Las *cotas ajustadas* tienen su origen en el contexto de la lógica relacional: `KodKod` [37] es un procedimiento de decisión para la lógica relacional (utilizado por `Alloy`[18] como lenguaje intermedio), el cual transforma fórmulas de la lógica relacional en una fórmula proposicional equivalente que luego es analizada por un *SAT solver*.

Su principal propósito es implementar dicha tradución de manera eficiente, es decir, utilizando la menor cantidad de variables proposicionales. Además, las cotas ajustadas son utilizadas en otros trabajos: en [14] se presenta una técnica de análisis basado en SAT para programas Java con anotaciones JML que hace uso (además de computarlas) de estas cotas para lidiar con problemas de *satifacibilidad booleana* más pequeños. En [15] estas cotas se utilizan para mejorar la *performance* de la ejecución simbólica en los casos en donde se utilizan algoritmos perezosos. En [6] estas cotas son utilizadas para, en combinacion con SAT incremental, producir eficientemente *suites de tests* que se ajustan a un criterio de cobertura dado.

5.2. Motivación

Dada la siguiente clase Java, la cual implementa árboles binarios de búsqueda:

```
public class SearchTree {
    private Node root;
    ...
}
```

```
public class Node {
   private int key;
   private Node left;
   private Node right;
  ...
}
```

El invariante de representación de la misma, establece que las instancias válidas de esta clase son aquellas que cumplen con ser un árbol acíclico, con un único padre para cada nodo (excepto la raíz) alcanzable desde la raíz y ordenado.

Se pretende generar exhaustivamente (bajo cotas preestablecidas) instancias de árboles binarios de búsqueda (`SearchTree`) utilizando la herramienta `Korat`, la cual, como se describió en el Capítulo 2, introduce un mecanismo muy efectivo para eludir entradas simétricas y evitar la iteración sobre porciones no alcanzables del *heap*. El espacio de estados explorados durante la generación es muy grande en comparación con la cantidad de candidatos válidos, por ejemplo, para árboles de tamaño entre 0 y 4, con hasta 4 nodos, y con claves en sus nodos entre 0 y 4, `Korat` explora 2657 candidatos, de los cuales solo 146 son válidos (salvo isomorfismo), es decir, más del 90 % de los candidatos explorados representan estructuras inválidas (no satisfacen el invariante de representación). Para tamaños de dominios más grandes, por ejemplo, 8, el número de candidatos explorados se eleva a 8.278.266, de los cuales solo un porcentaje menor al 1 % es válido (lo cual corresponde a 46.960 entradas válidas, salvo isomorfismo).

Por otro lado, la mayoría de los mecanismos de generación de *test*, donde `Korat` no es una excepción, son sensibles a la forma en la cual la especificación es expresada, es decir, tienen su eficiencia ligada a como la especificación está escrita sintácticamente. Por ejemplo, cuando se chequean diferentes restricciones sobre las entradas, hacerlo en diferente orden conducen a resultados (en términos de número de candidatos explorados) radicalmente distintos. Recorrer la estructura en un determinada orden (por ejemplo: recorrido en anchuro o en profundidad) también arroja resultados diferentes. Volviendo al ejemplo presentado anteriormente, `Korat` requiere de un invariante de representación escrito de manera imperativa (`repOk`), el cual no solo es utilizado para filtrar estructuras inválidas sino también es utilizado para definir un orden canónico entre los nombres de los objetos que forman parte de la estructura de acuerdo al orden en que `repOk` los visita, con el propósito de evitar estructuras simétricas. De esta manera la eficiencia de `Korat` está altamente ligada a como `repOk` está implementado (esto conlleva a escribir implementaciones poco naturales como por ejemplo: recorrer la estructura varias veces para visitar diferentes campos, además de requerir que el usuario conozca la mejor estrategia para hacerlo). Los resultados presentados en el párrafo anterior fueron calculados utilizando el `repOk` de árboles binarios de búsqueda que forma parte de la distribución de `Korat`. El uso de otra implementación, por ejemplo implementando un recorrido en profundidad produce resultados abismalmente diferentes: para tamaño 4 se exploran 776.335 (en lugar de 2657 como en el caso anterior) y para tamaño 8 el tiempo de generación supera las 5 horas, en lugar de los 1.6 segundos consumidos utilizando el `repOk` proveniente de la distribución de `Korat`.

Las *cotas ajustadas*, restringen las entradas excluyendo, del conjunto de valores que cada campo puede tomar, casos que son inviables debido a que producirían estructuras simétricas o no cumplirían con la especificación de las entradas válidas. Como se mencionó en la sección anterior, éstas han sido utilizadas con éxito en el contexto de especificaciones declarativas [14, 6, 15]. De estos antecedentes surge la intención de usar las *cotas ajustadas* en el contexto de la generación exhaustiva acotada, en donde se cree, podrían resultar beneficiosas para evitar del espacio de estructuras posibles aquellas que no son compatibles con las *cotas ajustadas*, y lograr una escalabilidad mayor en los casos en donde se cuenta con un `repOk` que no está escrito de la manera que la herramienta de generación (este caso `Korat`) espera.

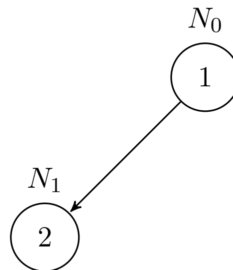
5.3. Incorporación *cotas ajustadas* en la generación exhaustiva acotada

En esta sección se describe el enfoque estudiando para aprovechar las *cotas ajustadas* durante el proceso de generación exhaustivo acotado. El enfoque es presentado en el contexto de la herramienta `Korat`, con lo cual, se requiere que el invariante sea dado de manera imperativo.

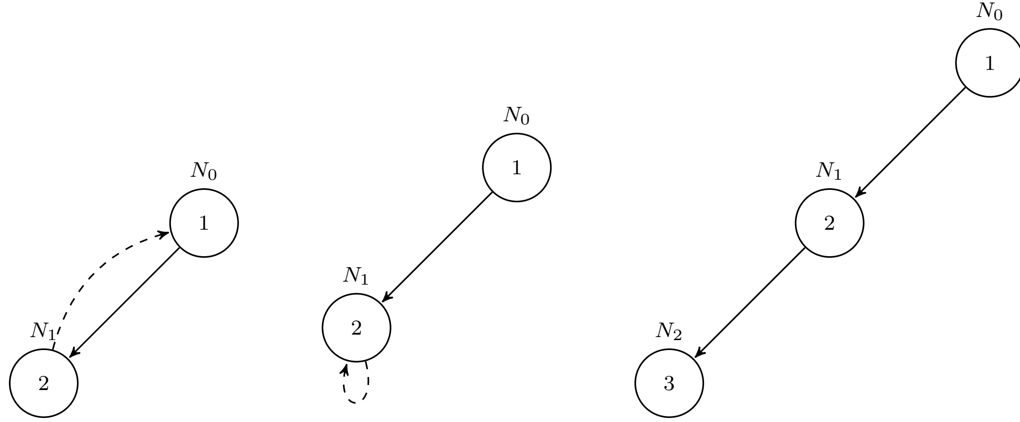
Esencialmente, este enfoque se basa en la siguiente observación: suponiendo que se cuenta con las *cotas ajustadas* previamente calculadas para cierta estructura, estas brindan información acerca de estructuras en el espacio de búsqueda que no es necesario examinar, ya que con certezas son inválidas (por rotura de simetría o por invariante de representación).

Cabe destacar que la técnica para incorporar las *cotas ajustadas* al proceso de generación exhaustiva acotada que se presenta a continuación, está basada en el trabajo presentado en [29].

Retomando el ejemplo presentado en la sección anterior, árboles binarios de búsqueda, en la búsqueda de instancias válidas (para árboles de tamaño entre 0 y 3), `Korat` construye el siguiente candidato:



Claramente, esta instancia es válida (satisface el invariante de representación). `Korat` utiliza el orden de visita de `repOk` para realizar el *backtracking*. Debido a que `repOk` se satisface, todas las partes (alcanzables) de la estructura serán examinadas. Suponiendo que el último campo visitado por `repOk` corresponde a $N_1.next$ (se ignoran las claves de los nodos por simplicidad), `Korat` intentará construir el siguiente candidato avanzando este campo, el cual actualemente apunta a $Null$, es decir los siguientes candidatos construidos (en el orden presentados de izquierda a derecha) corresponden a:



Los primeros dos candidatos, corresponden a estructuras cíclicas, por ende inválidas, es decir `Korat` visita estos dos candidatos inválidos antes de dar con la siguiente estructura válida.

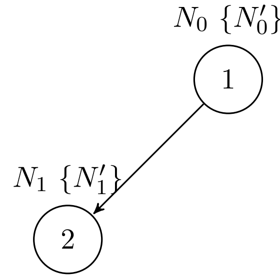
Supongamos ahora que se cuenta con *cotas ajustadas*, las cuales imponen una asignación de identificadores ($N'_0 < N'_1 < N'_2 < N'_3$) a nodos utilizando un recorrido del árbol en anchura (*Breadth First Search* (BFS)), para esta clase de objetos de tamaño 4, las cuales se presentan en la Figura 5.1.

$$\begin{aligned} \text{left} = \; & \{(N'_0, N'_1), (N'_0, null), \\ & (N'_1, N'_2), (N'_1, N'_3)(N'_1, null), \\ & (N'_2, null), (N'_2, N'_3), (N'_3, null)\} \end{aligned} \qquad \begin{aligned} \text{right} = \; & \{(N'_0, N'_1), (N'_0, N'_2), (N'_0, null), \\ & (N'_1, N'_2), (N'_1, N'_3), (N'_1, null), \\ & (N'_2, N'_3), (N'_2, null)\} \end{aligned}$$

$$\text{root} = \{null, N'_0\}$$

Figura 5.1: Cotas ajustadas (BFS) para árboles binarios de búsqueda de hasta 4 nodos

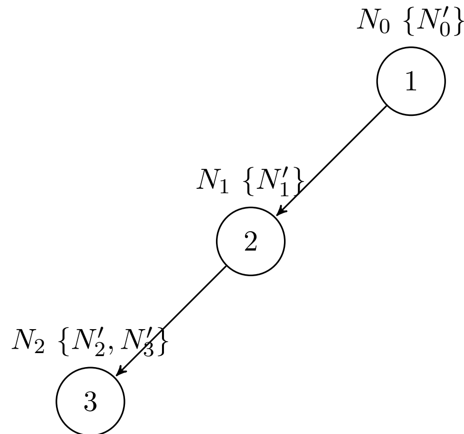
Basicamente, esta variante de `Korat`, a la cual se la llama `KoratCotas`, consiste en etiquetar los nodos con un conjunto de nombres compatibles con las *cotas ajustadas*, es decir cada nodo se etiqueta con el conjunto de identificadores que pueden, de acuerdo a las cotas, ser asignados a él (un nodo puede recibir diferentes identificadores en diferentes estructuras). Continuando con el ejemplo, y bajo los mismos supuestos: el último campo accedido por `repOk` corresponde a $N_1.next$. Cada nodo del candidato construido anteriormente será (durante la búqueda) etiquetado con el conjunto de nodos compatibles con las cotas (sin tener en cuenta el `null`). Estas etiquetas serán utilizadas para examinar, en la búsqueda del siguiente candidato, si cierto avance de un campo es, o no, viable de acuerdo a las cotas:



Según el cálculo estándar del "siguiente candidato" , la próxima asignación para $N_1.next$ corresponde a N_0 (actualemente $N_1.next = Null$). `KoratCotas` antes de construir este candidato examina la compatibilidad con la *cotas ajustada* mediante el siguiente chequeo:

$$\{N'_0\} \cap left(\{N'_1\}) \neq \emptyset$$

es decir, el conjunto de etiquetas de $left(\{N'_1\}) = \{N'_2, N'_3\}$ no interseca con el conjunto de etiquetas asociadas a N_0 ($\{N'_0\}$) entonces no es posible que $N_1.left = N_0$ (debido a las restricciones que imponen las cotas). En el siguiente avance corresponde asignarle a $N_1.next$ el nodo N_1, el cual nuevamente viola las restricciones inducidas por las cotas, por lo que el siguiente candidato que `KoratCotas` construye será:

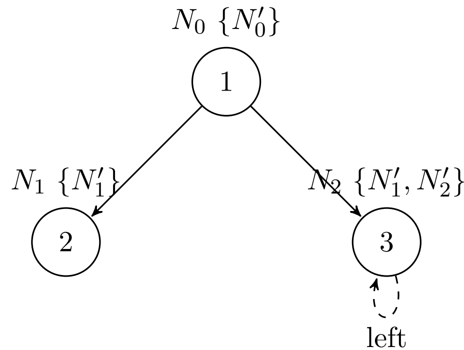


Debido a que la asignación $N_1.left = N_2$, en el cálculo del siguiente candidato, involucra un nuevo nodo (N_2), es decir un nodo que no fue previamente usado en la estructura, este nodo debe ser etiquetado con el conjunto de nodos asociados a $left(\{N'_1\})$

Vale la pena aclarar dos puntos, el primero de ellos es que se hace abuso de notación al referirse a $left(\{N_i, N_j\})$, lo cual, en realidad expresa $left(N_i) \cup left(N_j)$. En segundo lugar, es importante mencionar que el etiquetado que `KoratCotas` realiza, es independiente de la forma que `Korat` etiqueta los nodos (por ello se hace la diferencia entre identificadores primados y no primados). Esto acarrea una ventaja muy importante: el orden en que las *cotas ajustadas* se calculan no interfiere en el orden en que `Korat` rompe simetrías, es decir el orden de recorrido de `repOk`. En otras palabras, las cotas se calculan suponiendo un orden canónico, el cual es impuesto por el predicado de rotura de simetría utilizado para el cálculo de las mismas [14], el cual en muchos casos no coincide con el orden de recorrido de `repOk`, con lo cual mantener éstos de manera independiente asegura que la técnica es *sound*.

5.3.1. Refinamiento de Cotas

Como se puede observar, las etiquetas que `KoratCotas` le asigna a cada nodo en la estructura son conjuntos (la imagen de la relación $Cota_{f_i}$, con f_i un campo, vía el identificador N_j). Más pequeño es este conjunto, más chances hay de encontrar una violación de las *cotas* que lleve a evitar alguna estructura inválida. Retomando el ejemplo de los árboles binarios de búsqueda y las *cotas ajustadas* presentadas en 5.1, considere el siguiente árbol, el cual es cíclico, y por ende inválido.



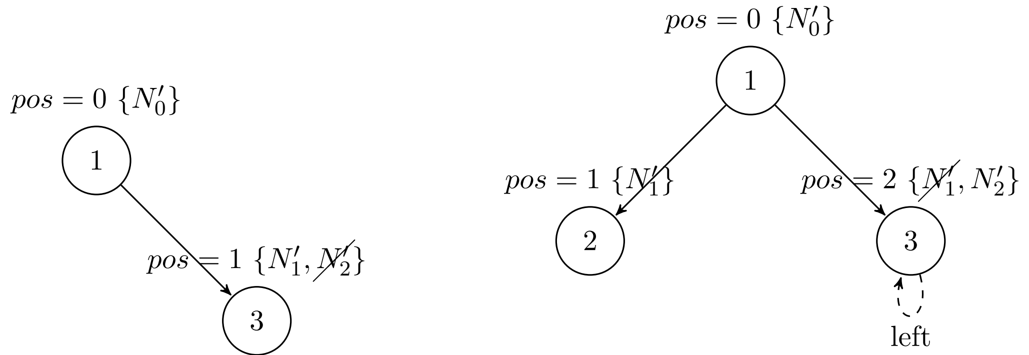
Este árbol, si bien es inválido, es compatible con las *cotas* presentadas, es decir, cada nodo fue etiquetado de acuerdo a las cotas. Si se analiza la razón por la cual este árbol fue generado por `KoratCotas`:

$$left(\{N'_1, N'_2\}) = \{N'_2, N'_3\} \cap \{N'_1, N'_2\} = \{N'_2\} \neq \emptyset$$

Esto se debe a la presencia de N_1' en la etiqueta asociada N_2, es decir si no se tiene en cuenta N_1' en la etiqueta de N_2 entonces sucede que:

$$left(\{N_2'\}) = \{N_3'\} \cap \{N_2'\} = \emptyset$$

y por lo tanto $N_2.left = N_2$ viola las *cotas*, por ende la estructura no es generada. Las *cotas* restringen posibles valores pero, dada una instancia, ciertas decisiones ya se tomaron, por ejemplo, la raíz podía ser *null* o N_0, para ya ha tomado el valor N_0. De la misma manera podríamos eliminar N_1' del conjunto de identificadores asociados a N_2, debido a que éste ya fue asignado a $root.left$. La técnica de refinamiento que se propone (también basada en la técnica de refinamiento que se presenta en [29]) incorporar a `KoratCotas` consiste en recorrer la estructura en *breadth first search* (debido a que este es el orden con el cual fueron calculadas las cotas), enumerando los nodos (distintos de *null*) en el orden en que aparecen. Esa posición (*pos*) es utilizada para eliminar nodos del conjunto de cotas que no coinciden con ella. Como ejemplo considere los siguientes árboles:



A la derecha se vuelve a la estructura anterior donde, como se puede notar, el refinamiento logra evitar la generación de esta estructura ya que: $left(N2) = N3 \cap N2 = \emptyset$.

Es importante destacar que debido a que el refinamiento se realiza mediante un recorrido en *breadth first search*, esto restringe el orden de cálculo de las cotas, las cuales también deben ser calculadas en ese orden.

5.3.2. Acerca de la correctitud de esta técnica

En esta sección se pretende discutir acerca de la correctitud de este enfoque (implementado como una variante de `Korat`) es decir:

`KoratCotas` y `Korat` producen el mismo conjunto de instancias válidas

Con el propósito de clarificar este concepto, en la Figura 5.2 se introduce una descripción abstracta del algoritmo de `KoratCotas`, la cual está basada en el algoritmo (también abstracto) de `Korat`, presentado en el Capítulo 2.

En este algoritmo, se resaltan aquellas partes que fueron agregadas al algoritmo estándar de `Korat`. Como se puede observar, luego de cada avance (línea 22), se chequea si este avance es compatible con las *cotas ajustadas* (rutina `boundsOk`). Si este avance no es compatible con las *cotas*, entonces se fuerza un nuevo avance (haciendo uso de la variable `nextCandidateFound`). Es importante notar que, si el avance realizado no es compatible con las cotas ajustadas, `lastField` (índice del último campo accedido) es nuevamente insertado (línea 29) en la pila de campos accedidos (`accessedFields`), lo cual asegura que no se cambia el orden de *backtracking* de `Korat`.

Por otro lado, la rutina `refineLabelSets` (línea 25) es la encargada de realizar el proceso de refinamiento de etiquetas descripto en la Sección 5.3.1. Este procedimiento recorre la estructura en orden *BFS* calculando la posición de cada nodo (sin tener en cuento los `nulls` en la estructura y los campos que no están en la pila de campos accedidos).

Es importante destacar que `updateReferenceCount` se encaraga de llevar la cuenta de la cantidad de veces que un nodo está siendo referenciado en la estructura. Esto es importante ya que los nodos frescos deben ser etiquetados (con el conjunto de identificadores compatibles con las cotas) al momento de incorporarse en la estructura.

Con respecto a la correctitud de esta técnica, asumiendo que las cotas ajustadas provistas son correctas, y que el mecanismo de etiquetado de nodos con conjunto de identificadores compatibles con las cotas también lo es ([29]), supongamos que tenemos un vector candidato válido (v) que fue generado por `Korat` pero no por `KoratCotas`. La única manera de que `KoratCotas` salte este vector candidato, es que el anterior (v_0) haya violado las *cotas*, y el avance que se hace (mediante las cotas) lleve a un vector v_1 mayor que v. Si tal violación se produce, ésta se produce en el índice `lastField` del vector candidato v_0, el cual es apilado nuevamente en la pila y avanzado de la manera estándar. Debido a que la búsqueda se realiza sobre `accessedFields` y que al momento de forzar un avance, esta pila no se modifica (sumado al hecho de que `repOk` es determinístico) se está seguro que tal vector v no existe.

5.3.3. Limitaciones del Enfoque

Como se explicó más arriba, este enfoque requiere de las *cotas ajustadas* previamente calculadas. Existen varias técnicas para el cómputo de las mismas; por ejemplo: en [14] se presenta `TACO`, esta herramienta implementa un algoritmo paralelo para el cómputo de *cotas ajustadas* (lo cual requiere de un cluster para lograr dicho cómputo) desde una especificación en `JML` [30]. Las *cotas ajustadas* que esta herramienta devuelve respetan el orden *Breadth First Search*. Por otro lado, en [31] se presentan diferentes enfoques secuenciales para el cómputo eficiente de las mismas, uno de ellos (al igual que `TACO`) requiere de una especificación en `JML`, el otro está basado en una especificación dada en términos de un predicado inductivo expresado por medio de la lógica de separación.

Si bien actualmente existen enfoques para el cómputo de *cotas* que resuelven este problema de manera eficiente, todos ellos lo hacen desde una invariante declarativa. Este hecho, desde el punto de vista de la técnica presentada anteriormente, es una limitación significativa, ya que ésta requiere de un invariante imperativo para la generación de entradas. Hasta el momento, no se conoce ningún enfoque (eficiente) para el cálculo de *cotas* a partir de invariantes imperativos, por ende, la técnica presentada no solo demanda el invariante de representación escrito de manera imperativa sino también declarativamente, lo cual adiciona un esfuerzo no menor en la traducción del mismo de un lenguaje a otro, en muchos casos, desconocido.

5.4. Generación exhaustiva acotada desde especificaciones híbridas

Esta sección está dedicada a introducir el enfoque para generación exhaustiva acotada, basado en invariantes híbridos, al cual se hace mención en la introducción del capítulo. Las ideas introducidas más abajo fueron publicadas en [32] (quien escribe es co-autora del mismo). Esta técnica se encuentra basada en el uso de invariantes expresados imperativamente, declarativamente (es decir usando un lenguaje de especificación de contratos) o como combinación de ambas formas, para la generación de *suites de tests* exhaustivas acotadas. Dichas especificaciones *híbridas* se procesan usando mecanismos ya conocidos para las partes imperativas y declarativas, pero la combinación de ambos permite explotar información del lado declarativo, tales como las *cotas ajustadas* computadas a partir de la parte declarativa, que permiten mejorar la búsqueda de instancias válidas. Además, debido a que como se explicó más

arriba en este capítulo, la mayoría de los mecanismos de generación son sensibles a cómo la especificaćion está (sintácticamente) escrita, como es el caso particular de `Korat` (y en general de los enfoques basados en invariantes imperativos), la técnica propuesta automáticamente evalúa diferentes formas de procesar el lado imperativo y diferentes alternativas de combinaciones para las partes declarativas e imperativas, para decidir que configuración del invariante de representación produce *suites de tests* de manera más eficiente.

```
1  function koratCotas() {
2      Vector current = initVector;
3      Stack accessedFields = new Stack();
4      boolean ok;
5      do{
6          (ok, accessedFields) = current.repOk();
7          if(ok){
8              reportValid(current);
9              accessedFields.push(current.reachableFields -
                   accessedFields);
10         }
11
12       boolean nextCandidateFound = false;
13           while (!nextCandidateFound) {
14             lastField = accessedFields.pop();
15             while (!accessedFields.isEmpty() &&
16                   current[lastField] >= nonIsoMax(current,
                        accessedFields, lastField)) {
17                 current[lastField] = 0;
18                 updateReferenceCount(lastField);
19                 lastField = accessedFields.pop();
20             }
21             if (!accessedFields.isEmpty()){
22                   current[lastField]++;
23                   updateReferenceCount(lastField);
24                   if (correspondsToBoundedField(lastField)) {
25             refineLabelSets(lastField,current);
26                nextCandidateFound = boundsOk(lastField, current);
27                restoreLabelSets();
28                if (!nextCandidateFound)
29                     accessedFields.add(lastField);
30          }else nextCandidateFound = true;
31        }
32
33     }
34
35     } while (current != lastVector && !accessedFields.isEmpty())
36 }
```

Figura 5.2: Algoritmo de KoratCotas

```
function boundsOk(int currentIndex, Vector currentVector){
    boolean brokenBounds = false;
    String fieldName = fieldName(currentIndex);
    LabeSet source = labelSet(currentVector[currentIndex]);
    LabeSet target = targeOfSet(fieldName, labelSet(currentIndex
        ));
    if (!target.intersects(source)) {
      brokenBounds = true;
    }
    return brokenBounds;
  }
```

Figura 5.3: Algoritmo de KoratCotas

```
procedure refineLabelSets(int currentIndex, Vector currentVector
  ) {
   Queue  workList = new List();
   Set visited = new Set();
   if (accessedFields.contains(rootPosicion))
     workList.add(rootPosicion);
   List bfsTraversal = new List();
   while (!workList.isEmpty()) {
     int current = workList.remove();
     if (current == -1) break
     if (visited.add(current)) {
       bfsTraversal.add(current);
       for (int i=0; i<currentVector.length; i++) {
         if (accessedFields.contains(i) || i==currentIndex) {
           if (notNull(currentVector[i]) && !visited.contains(
              currentVector[i]))
             workList.add(currentVector[i]);
           }else workList.add(-1);
       }
     }
   }
   int i = 0;
   while (i<bfsTraversal.size()) {
     int currentNode = bfsTraversal.get(i);
     labelSet(currentNode) = new LabelSet(i);
     i++;
   }
 }
```

Figura 5.4: Algoritmo de KoratCotas

Capítulo 6

Evaluación experimental

En los capítulos 3, 4 y 5 se presentaron 4 enfoques para mejorar la generación exhaustiva acotada:

- Generación exhaustiva acotada de clases de equivalencia asociadas a un criterio dado,
- Reducciones al *testing* exhaustivo acotado basadas en el uso del invariante de representación:
  - Generación independiente de sub-estructuras disjuntas,
  - Reducción de *suites de tests* exhaustivas acotadas usando criterios sobre `repOk()`
- Cotas ajustadas en la generación exhaustiva acotada.

En este capítulo se evaluan experimental estas técnicas con el objetivo de analizar la *eficiencia* y *efectividad* de las *suites* producidas en comparación con generación exhaustiva acotada. En otras palabras, las siguientes preguntas de investigación guian esta experimentación:

- *RQ1*: ¿Los enfoques presentados son más eficientes que la generación exhaustiva acotada?
- *RQ2*: ¿Cuán efectivas son las *suites de test* reducidas con respecto a generación exhaustiva acotada?

La eficiencia de las técnicas fue medida con respecto al tiempo consumido durante la generación (excepto por la técnica de reducción de *suites de tests* usando criterios sobre `repOk()`, la cual se realiza a posteriori, es decir, a partir de una *suite* exhaustiva acotada previamente generada). Además, en los casos de las técnicas de reducción de *suites de test* es deseable medir la eficiencia con respecto al tiempo de *testing*, es decir, el tiempo consumido para realizar *testing* utilizando las *suites* reducidas, lo cual, en este caso se realiza de manera indirecta: se midió tamaño de las *suites* reducidas con respecto a generación exhaustiva acotada, y se espera que esto contribuya, en la mayoría de los casos, a disminuir el tiempo de *testing* de manera proporcional a la reducción en tamaño de las *suites*.

En lo que respecta a la efectividad de las *suites de test* reducidas, es de interés analizar si las clases de equivalencia usadas no sólo son útiles para achicar la *suites de tests* sino también para producir suites útiles a la hora de encontrar errores en el código. En otras palabras: ¿Las suites de tests reducidas utilizando estas clases de equivalencia, conservan buenas propiedades a la hora de encontrar errores en el código bajo prueba?. Con la intención de responder a esta pregunta se utilizó *Testing de Mutación*.

6.1. Incorporación de Criterios de cobertura en la generación exhaustiva acotada

La efectividad de esta técnica fue evaluada para diversos casos de estudio, los cuales involucran el uso de varias estructuras. En el caso de la evaluación para criterios de caja negra se utilizó el criterio de cobertura de clases de equivalencia presentado en el Capítulo 2, y las estructuras de datos que se listan a continuación:

- *ListToSet: listAsSet* (descripta detalladamente en la Sección 4.3.1 del Capítulo 3),
- *Binomial heap: merge (dado dos binomial heaps los combina para obtener un tercero como salida.)*,
- *Grafos Dirigidos*,
- *Grafos dirigidos etiquetados*,
- *Árboles binarios de búsqueda: delete* y

- *lista simplemente encadenada: algoritmo de ordenamiento estable* .

Para el caso de la evaluación considerando un criterio de cobertura de caja blanca, se utilizó el criterio de cobertura de decisión sobre las rutinas de inserción y búsqueda de las siguientes estructuras:

- *Árboles Rojos y Negros* y
- *Árboles binarios de búsqueda*

Con el objetivo de medir que tan buenas son las *suites de tests* producidas utilizando la técnica presentada, se decidió tomar algunos de los casos de estudio de caja negra, particularmente aquellos en los que la poda fue más efectiva, y hacer un análisis de cuan buena es la *suite de tests* obtenida para encontrar errores, en comparación con las *suites* obtenidas utilizando `Korat`. Los casos de estudio seleccionados son:

- *list as set* con la implementación de la operación *listToSet*
- *binomial heaps* con la implementación de la operación *merge*, y
- *árboles binarios de búsqueda* con la implementación de la operación *delete*

Para realizar esta medición se utilizó *testing de mutación*. Se generaron mutantes de las rutinas bajo análisis y luego se emplearon las *suites de tests* generadas por `Korat`, `Korat+` y clases de equivalencia óptimas (UPC) (es decir, uno por cada clase equivalencia) para ver cuantos mutantes, cada *suite de tests* es capaz de matar (detectar).

Las *suites* UPC, son aquellas producidas tomando una entrada válida de la *suite* exhaustiva acotada producida por `Korat`, por cada clase de equivalencia correspondiente al criterio seleccionado, en particular la primera entrada válida generada/encontrada por cada clase de equivalencia.

Para cada caso de estudio, se eligió reportar el número de candidatos explorados, acompañado por el número de candidatos válidos encontrados. Esto es la medida más razonable para emplear, si se está interesado en evaluar el nivel de contribución de la técnica, en el proceso de filtrado estándar. En este caso, estos números se reflejan directamente en los tiempos de corrida, debido a que las rutinas `eqClass()` empleadas (la parte con más influencia (con respecto al tiempo de corrida) de la poda extra que realiza `Korat+`) no incrementan, en una forma notable, el tiempo de

corrida de `Korat`, para las cotas incluidas en estos casos de estudio. Esto se confirma mediante las tablas presentadas que muestran tiempos de corrida. Sin embargo, sólo se reportan los tiempos de generación. Se espera que esto, también se vea reflejado en el tiempo necesario para realizar *testing* utilizando las entradas producidas. Para correr todos los experimentos se utilizó un procesador *Intel Core 2 Duo 3.06GHz con 4GB de RAM*, y los datos reportados corresponden a experimentos que terminan en un tiempo límite de 5 horas.

Por otro lado, cabe aclarar que las rutinas `eqClass()` utilizadas como parte de estos experimentos tienen en cuenta, en el caso de cobertura de caja negra, las clases de equivalencia resultantes de la combinación de los predicados listados como parte de cada caso de estudio. Para criterios de cobertura de caja blanca, la rutina `eqClass()` se consigue mediante la correcta instrumentación del código bajo análisis, de manera tal que dado una estructura válida, la clase de equivalencia a la cual pertenece se calcula según las decisiones (para cobertura de decisión) que ésta ejercita en el código.

6.1.1. Evaluación experimental utilizando criterios de cobertura de caja negra

listAsSet. El primer caso de estudio corresponde a la rutina `listAsSet`, presentada en detalle en el Capítulo 3, la cual requiere como entrada objetos de la siguiente clase:

```
public class ListToSet{
    public StrictlySortedSinglyLinkedList set;
    public SinglyLinkedList list;
    ...

}
```

y el criterio de caja negra *BPLP*, el cual requiere cubrir todas las combinaciones de los predicados:

- "`list` esta vacía"
- "`list` tiene elementos repetidos"
- "`list` esta ordenada"

- “`set` esta vacía”

Los métodos `repOk()` y `eqClass()` fueron implementados como se describió en la Sección 4.3.1 del mencionado capítulo. Este es un caso de estudio muy simple, con pocas clases de equivalencia, pero sirve como ejemplo para mostrar los beneficios de esta técnica. Los resultados experimentales se muestran en la Tabla 6.1. Las cotas indican rango de valores posibles para el campo `size` de ambas listas (como cotas separadas), máximo número de nodos en cada lista (como cotas separadas), y el número de valores diferentes permitidos para las claves, respectivamente. Para `Korat` y `Korat` con poda basada en cobertura (`Korat+`), la Tabla 6.1 muestra el número de vectores explorados, junto con la cantidad de casos de *test* válidos (es decir que satisfacen el `repOk()`). Además, indicamos el número de clases de equivalencia cubiertas (CC), para cada cota (las clases cubiertas son las mismas para ambos algoritmos debido a que la técnica es *sounded*). La Tabla 6.2 y el Gráfico de líneas 6.1 comparan los tiempos consumidos por `Korat` y `Korat+` para este caso de estudio.

Cotas	***Korat***	***Korat+***	***CC***
0-2,0-2,3,3,3	1121 (91)	185 (26)	8
0-4,0-4,3,3,3	1485 (91)	211 (26)	8
0-4,0-4,4,4,3	14679 (320)	679 (80)	10
0-5,0-5,5,5,4	1.274.977 (5456)	6798 (682)	10
0-5,0-5,5,5,5	6.692.357 (24211)	16369 (1562)	10
0-6,0-6,6,6,5	1.97.651.224 (124.992)	89.650 (7812)	10
0-7,0-7,7,7,6	TIMEOUT	1.453.804 (111.974)	10

Cuadro 6.1: Comparación `Korat` y `Korat+` para `listAsSet`

Cota	Korat	Korat+
0-4,0-4,4,4,3	0,422s	0,269s
0-5,0-5,5,5,4	2,39s	0,395s
0-6,0-6,6,6,5	329,809s	0,817s
0-7,0-7,7,7,6	TIMEOUT	3,36s

Cuadro 6.2: Tiempos de corrida de `Korat` y `Korat+` para `listAsSet`

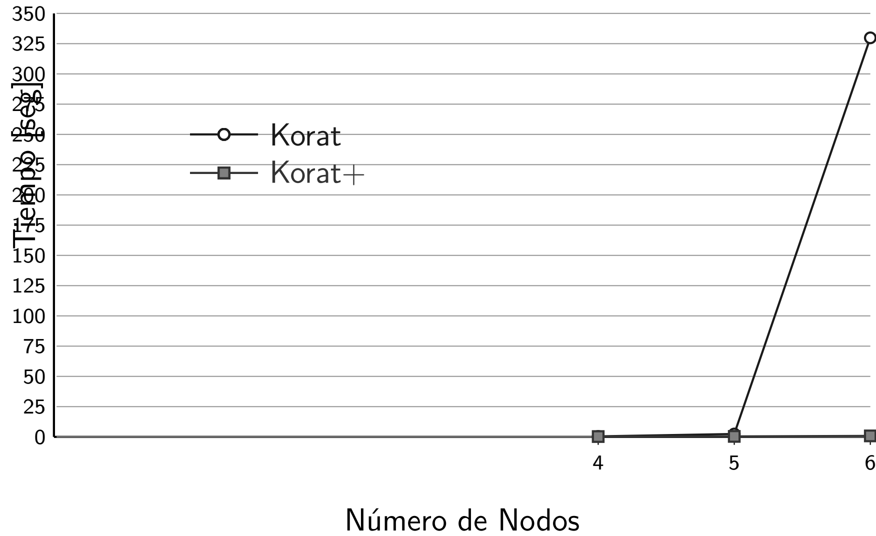


Figura 6.1: Tiempos de corrida de `Korat` y `Korat+` para `listAsSet`

Para evaluar la efectividad de las diferentes suites (exhaustiva acotada, suites de tests reducida producida por `Korat+` y la suite generada tomando exactamente una entrada por cada clase de equivalencia cubierta (UPC)) se mutó la rutina `ListToSet` (el número de mutantes obtenidos fue 49). La tabla 6.3 muestra los resultados para este caso de estudio. Se reportan mutantes vivos (los cuales corresponden basicamente a errores no detectados) para cada una de las *suites* de test.

Es importante recordar que las *suites* UPC, son aquellas producidas tomando una entrada válida de la *suite* exhaustiva acotada producida por `Korat`, por cada clase de equivalencia correspondiente al criterio seleccionado, en particular la primera entrada válida generada/encontrada por cada clase de equivalencia.

Binomial Heap (`Merge`). El segundo caso de estudio se refiere a *Binomial Heaps*. Una operación fundamental sobre *Binomial heaps* es la operación `merge` de 2 *heaps*, la cual puede ser realizada muy eficientemente. Suponiendo que es de interés probar esta rutina, es necesario proveer pares de *binomial heaps* como entradas. Esta operación depende como éstos se componen y del grado de los *binomial heaps* a componer. Considerando particionado en clases de equivalencia como el criterio de cobertura de caja negra, los siguientes predicados deberían proveer una cobertura adecuada:

- el primer *heap* es vacío,

List as Set (49 mutantes)			
Scope	Korat	Korat+	UPC
0-2,0-2,3,3,3	3	9	15
0-4,0-4,3,3,3	3	9	15
0-4,0-4,4,4,3	3	9	11
0-5,0-5,5,5,4	3	9	10
0-5,0-5,5,5,5	3	9	10

Cuadro 6.3: Efectividad, con respecto a testing de mutación, de las suites de tests generadas por `Korat` y `Korat+` sobre la rutina `listToSet` (se reportan mutantes vivos).

- el segundo *heap* es vacío,
- el primer *heap* tiene más elementos que el segundo,
- ambos *heaps* tienen la misma cantidad de elementos,
- el primer *heap* tiene grado más grande que el segundo,
- ambos *heaps* tienen el mismo grado, y
- ambos *heaps* contienen un árbol con el mismo grado.

Para estos experimentos se utilizó la implementación de *binomial heaps* con su correspondiente `repOk()`, extraída de la distribución de `Korat`, replicada para los dos *binomial heaps*. El dominio para cada uno de estos dos *binomial heaps* fue definido de manera disjunta en el correspondiente método *finitization*. Los resultados experimentales se muestran en la Tabla 6.4. En este caso, la cota indica el máximo número de elementos que ambos *heaps* pueden tener. Las claves en los nodos están definidas en entre 0 y este valor (considerando repeticiones en las mismas). La Tabla 6.4 muestra el número de vectores candidatos explorados junto con la cantidad de estos vectores que fueron encontrados válidos. Además, se exhibe la cantidad de clases de equivalencia cubiertas (CC). Por otro lado, la Tabla 6.5 y el Gráfico de líneas 6.2 comparan los tiempos de corrida de `Korat` y `Korat+` para este caso de estudio.

Cota	Korat	Korat+	CC
2	348 (36)	147 (15)	6
3	5389 (784)	1315 (56)	10
4	150.448 (14.400)	46.786 (435)	10
5	3.125.314 (876.096)	647.410 (1872)	10
6	274.808.123 (57.790.404)	55.745.855 (43.134)	10

Cuadro 6.4: Comparación `Korat` y `Korat+` para *Binomial Heap (Merge)*

Cota	Korat	Korat+
2	0,42s	0,394s
3	0,656s	0,454s
4	1,436s	0,86s
5	16,347s	2,492s
6	1360,323s	178,072s

Cuadro 6.5: Tiempo de Corrida de `Korat` y `Korat+` para *Binomial Heap (Merge)*

Grafos Dirigidos. El tercer caso de estudio corresponde a la generación de casos de *test* para una rutina manipulando grafos dirigidos. La implementación de grafos dirigidos es una implementación orientada a objetos estándar, consistiendo de un vector de vértices, donde cada uno de estos vértices tiene su correspondiente lista simplemente encadenada y estrictamente ordenada, representando su lista de adyacencia. Si se esta interesado en generar entradas cuyos arcos tenga densidad variada y que cubran casos bordes, se podrían tener en cuenta la combinación de las siguientes características sobre grafos dirigidos, para considerar particionado en clases de equivalencia:

- vacuidad,
- densidad, y
- completitud.

Los resultados experimentales para este caso de estudio se muestran en las Tablas 6.7 y 6.8. En este caso, las cotas indican el número exacto de nodos en el grafo dirigido. Como en los casos previos, la Tabla 6.7 muestra el número de vectores

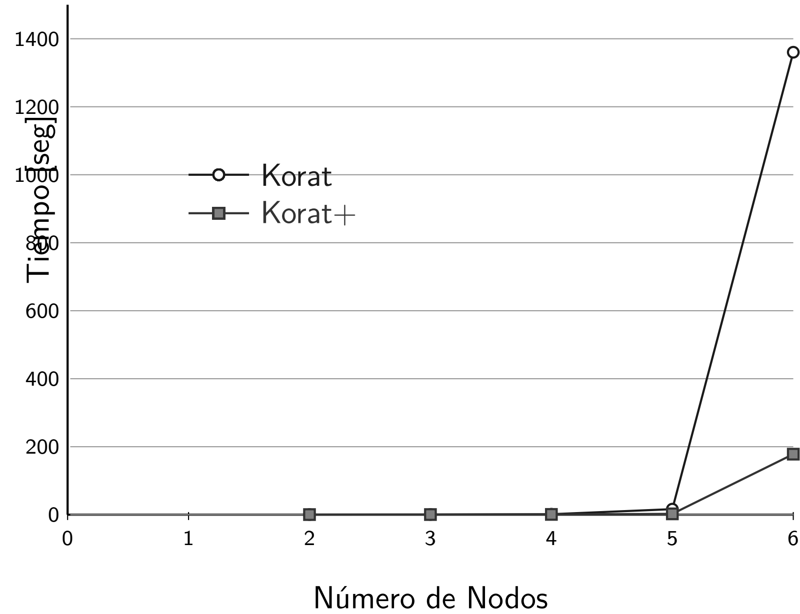


Figura 6.2: Tiempos de Corrida de `Korat` y `Korat+` para *Binomial Heaps (Merge)*

!ht

Binomial Heaps (117 mutantes)			
Scope	Korat	Korat+	UPC
2	38	39	44
3	8	8	17
4	7	7	17
5	7	7	17
6	7	7	17

Cuadro 6.6: Efectividad, con respecto a testing de mutación, de las suites de tests generadas por `Korat` y `Korat+` sobre la rutina `merge` de `binomilaHeap` (se reportan mutantes vivos).

candidatos explorados junto con la cantidad de éstos que son válidos, y la cantidad de clases cubiertas (CC). Como podemos notar en esta tabla, el número de casos válidos crece muy rápido, impidiendo que se reportan resultados con cotas mayor a 3. La Tabla 6.8 y el Gráfico de líneas 6.3 muestran una comparación de los tiempos de corrida de `Korat` y `Korat+` para este caso de estudio.

Cota	Korat	Korat+	CC
2	1624 (382)	518 (126)	4
3	372.861.255 (47.672.840)	11.670.154 (899.852)	4
4	TIMEOUT	TIMEOUT	

Cuadro 6.7: Comparación `Korat` y `Korat+` para *Grafos Dirigidos*

Cota	Korat	Korat+
2	0,343s	0,265s
3	1145,341s	37,854s

Cuadro 6.8: Tiempos de corrida de `Korat` y `Korat+` para *Grafos Dirigidos*

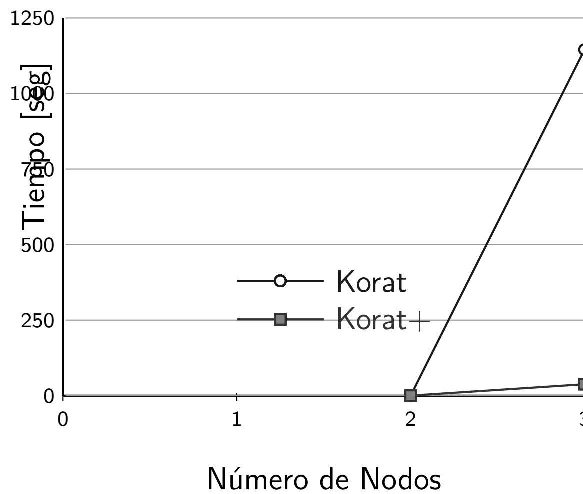


Figura 6.3: Tiempos de corrida de `Korat` y `Korat+` para *Grafos Dirigidos*

Grafos Dirigidos Etiquetados. El siguiente caso de estudio es una extensión del anterior. En este caso se tiene como objetivo generar casos de *test* para Grafos Dirigidos *Etiquetados*. La implementación de estos grafos es una extensión del anterior, en la cual, cada entrada en la lista de adyacencia de un vértice contiene su correspondiente etiqueta, llamado también *peso*. Algunos algoritmos muy comunes sobre grafos dirigidos etiquetados son aquellos para calcular clausura transitiva o camino más corto, por ejemplo usando el algoritmo de *floyd*. Usando la definición de camino mínimo se pueden definir algunas clases de equivalencia representativas, basadas en las siguientes propiedades de este tipo de grafos:

- aciclicidad,
- presencia de pesos (etiquetas) negativas, y
- conectitud del grafo (es fuertemente conexo?).

Todos estos predicados juegan un importante rol en el cálculo de la clausura transitiva o del camino más corto. Por esto es que, se considera que estos predicados son adecuados para ser utilizados en cobertura de clases de equivalencia. Como en el caso anterior, para conseguir casos con arcos de densidad variada y cubrir casos bordes, también se tiene en cuenta *vacuidad*, *densidad* y *completitud* de la estructura, como en el caso de estudio anterior. Los resultados experimentales para este caso de estudio se muestran en las Tablas 6.9 y 6.10. Las cotas indican la cantidad exacta de nodos en el grafo, y el rango de valores válidos para las etiquetas. Nuevamente, la Tabla 6.9 reporta la cantidad de vectores candidatos explorados junto con la cantidad de estos que son válidos, y el número de clases cubiertas. La Tabla 6.10 y el Gráfico de líneas 6.4 comparan los tiempos de corrida de `Korat` y `Korat+` para este caso de estudio.

Árboles Binarios de Búsqueda. El quinto caso de estudio para cobertura de caja negra, es la operación `delete` sobre *árboles binarios de búsqueda*, la cual, dado un valor y un árbol, elimina ese valor del árbol dado. Para probar esta rutina es necesario proveer pares de entradas compuestas por combinaciones de un árbol y un valor a borrar en él. La implementación de árboles binarios de búsqueda utilizada para este caso, es aquella provista por la distribución de `Korat` junto con su respectivo `repOk()`. Las clases de equivalencia en este caso corresponden a un análisis de la posición del valor a ser eliminado dentro del árbol; para ellos se eligieron los siguientes casos:

Cota	Korat	Korat+	CC
2,-1-1	1062 (332)	984 (256)	11
2,-2-2	2272 (1542)	1,750 (1022)	11
3,-1-1	18.003.420 (493.232)	17.815.155 (304.982)	13
3,-2-2	33.122.848 (15.612.660)	25.486.513 (7.976.340)	13
3,-3-3	205.397.228 (187.887.040)	103.127.315 (85.617.142)	13
4,-4-4	TIMEOUT	TIMEOUT	

Cuadro 6.9: Comparación `Korat` y `Korat+` para *Grafos Dirigido Etiquetados*

Cota	Korat	Korat+
2,-2,2	0,632s	0,534s
3,-3,3	2812,665s	1333,942s

Cuadro 6.10: Tiempos de corrida de `Korat` y `Korat+` para *Grafos Dirigidos Etiquetados*

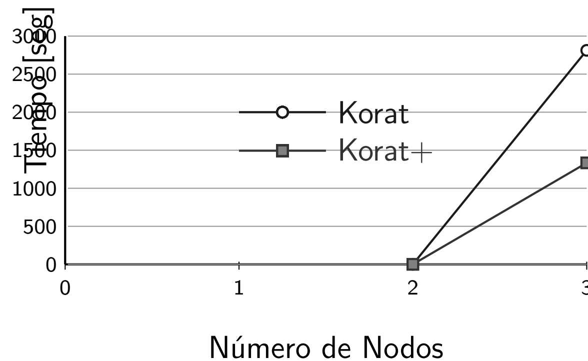


Figura 6.4: Tiempos de corrida de `Korat` y `Korat+` para *Grafos Dirigidos Etiquetados*

- el valor no se encuentra en el árbol,
- el valor se encuentra en la raíz,
- el valor se encuentra en una hoja,
- el valor se encuentra en un nodo con sub-árboles izquierdo y derecho no vacíos,

- el valor se encuentra en un nodo con sub-árbol izquierdo no vacío y sub-árbol derecho vacío, y
- el valor se encuentra en un nodo con sub-árbol derecho no vacío y sub-árbol izquierdo vacío.

Los resultados experimentales se pueden observan en las Tablas 6.11 y 6.12. La Tabla 6.11 reporta cantidad de candidatos explorados junto con la cantidad de ellos que corresponden a estructuras bien formadas y la cantidad de clases de equivalencia cubiertas (CC). Las cotas en este caso, indican el número máximo de nodos en el árbol, el rango de valores permitidos para el campo *size* del árbol, y el número de valores posibles para las claves en los nodos del árbol. En particular la Tabla 6.12 y el Gráfico de líneas 6.5 comparan los tiempos de corrida de ambos algoritmos: `Korat` y `Korat+` para este caso de estudio.

Cuadro 6.11: Comparación `Korat` y `Korat+` para *Árboles Binarios de Búsqueda (Delete)*

Cota	Korat	Korat+	CC
3,0-3,3	534 (45)	500 (43)	8
3,0-3,4	1152 (148)	1011 (125)	8
3,0-3,6	4290 (822)	3331 (586)	8
3,0-3,8	12.144 (2760)	8675 (1793)	8
4,0-4,7	46.795 (5005)	34.240 (3089)	9
5,0-5,8	477.888 (29416)	338.292 (16.137)	9
6,0-6,9	4.597.299 (167814)	3.213.270 (83511)	9
10,0-10,13	TIMEOUT	TIMEOUT	

Para evaluar la efectividad de las diferentes suites (exhaustiva acotada, suites de tests reducida producida por `Korat+` y la suite generada tomando exactamente una entrada por cada clase de equivalencia cubierta (UPC)) se mutó la rutina `delete` (el número de mutantes obtenidos fue 24). La tabla 6.1.1 muestra los resultados para este caso de estudio. Se reportan mutantes vivos (los cuales corresponden basicamente a errores no detectados) para cada una de las *suites* de test.

Listas Simplemente Encadenadas. El último de los casos de estudio para caja negra, es en relación a listas simplemente encadenadas de enteros. Dado un

Cuadro 6.12: Tiempos de corrida de `Korat` y `Korat+` para *Árboles Binarios de Búsqueda (Delete)*

Cota	Korat	Korat+
3,0-3,6	0,423s	0,359s
4,0-4,7	0,88s	0,748s
5,0-5,8	1,607s	1,333s
6,0-6,9	7,529s	5,724s

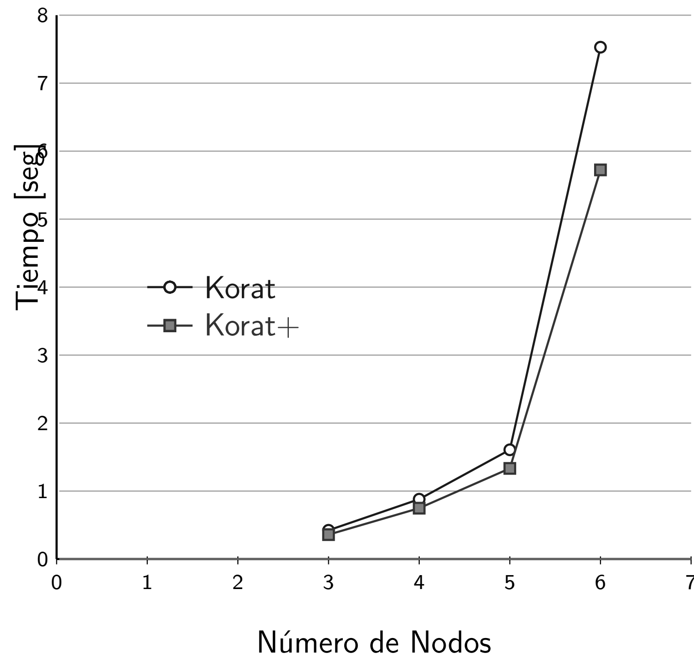


Figura 6.5: Tiempos de corrida de `Korat` y `Korat+` para *Árboles Binarios de Búsqueda (Delete)*

algoritmo de ordenamiento estable, se considera cobertura de clases de equivalencia para probar la misma. Algunas clases representativas pueden ser definidas teniendo en cuenta los siguientes predicados:

- la lista es vacía

Árboles binarios de búsqueda (`delete`) (24 mutantes)			
Scope	Korat	Korat+	UPC
3,0-3,3	2	2	2
3,0-3,4	2	2	2
3,0-3,6	2	2	2
3,0-3,8	2	2	2
5,0-5,8	0	0	2
6,0-6,9	0	0	2

Cuadro 6.13: Efectividad, con respecto a testing de mutación, de las suites de tests generadas por `Korat` y `Korat+` sobre la rutina `delete` de Árboles binarios de búsqueda (se reportan mutantes vivos).

- la lista tiene repeticiones
- los elementos están desordenados.

Los resultados experimentales se muestran la Tabla 6.14. Las cotas indican el rango de valores válidos para el campo *size*, el número máximo de nodos en la lista y el número de valores enteros permitidos en cada nodo de la lista. Esta tabla muestra el número de candidatos explorados junto con el número de listas válidas encontradas y el número de de clases cubiertas. Como se puede notar, tanto `Korat` como `Korat+` producen la misma salida (vectores explorados y casos de *test* válidos), es decir, `Korat+` no evita mirar ningún candidato dentro del espacio de búsqueda. Esto se debe a que para determinar a que clase de equivalencia pertenece una entrada es necesario examinar la estructura completa. Más precisamente, en el peor caso, es necesario examinar toda la lista para poder decidir si ésta, está, o no, ordenada y si tiene, o no, repeticiones.

6.1.2. Evaluación experimental utilizando criterios de cobertura de caja blanca

Como se menciona más arriba, esta técnica también fue analizada para algunos casos de estudio utilizando un criterio de caja blanca de complejidad media: *cobertura de decisión*. Este criterio es apropiado en los casos de estudio seleccionados debido a que los procedimientos bajo análisis cuentan con muchas decisiones, pero en su gran mayoría simples. Los resultados que se muestran a continuación muestran que

Cotas	Korat/Korat+	CC
0,3,4,3	319 (40)	4
0,4,5,4	3388 (341)	4
0,5,6,5	46.684 (3906)	4
0,6,7,6	781.960 (55.987)	4
0,7,8,7	15.349.933 (960.804)	4

Cuadro 6.14: Comparación `Korat` y `Korat+` para *listas simplemente encadenadas*

esta técnica parece ser más efectiva cuando se trata de criterios de cobertura de caja negra.

Árboles Rojos y Negros. El primero de los casos de estudio que involucra cobertura de caja blanca corresponde a las operaciones de inserción (`insert`) y búsqueda (`find`) en *árboles rojos y negros*. La implementación de árboles rojos y negros que se uso en estos experimentos es aquella que provee la distribución de `Korat`. El criterio de cobertura de *test* empleado en este caso es el criterio de *cobertura de decisión*. Los resultados experimentales se muestran en las Tablas 6.15, 6.16, 6.17 y 6.18. Las dos primeras tablas muestran los resultados obtenidos para la rutina `insert`, mientras que las dos últimas lo hacen para la rutina `find`. En estas tablas se reportan, la cantidad de vectores candidatos explorados junto con la cantidad de estos que son válidos, y el número de clases cubiertas, para diferentes cotas. Por otro lado, también se reportan los tiempos de corrida de `Korat+` en comparación con `Korat` estándar. Además, estos resultados se exhiben en los Gráficos de líneas: 6.6 y 6.7, lo cual permite una lectura más visual de los resultados. Las cotas indican el máximo número de nodos, el rango de valores para el campo `size` del árbol, y el rango de valores permitidos para las claves almacenadas en los nodos del árbol.

Árboles binarios de búsqueda. En este caso se analiza una estructura que ya fue utilizada previamente, *árboles binarios de búsqueda*. Al igual que en el caso de estudio anterior, se consideran bajo análisis las rutinas de inserción (`insert`) y búsqueda (`search`); y el criterio de cobertura de decisión sobre las mismas. La implementación utilizada en estos experimentos es aquella que provee la distribución de `Korat`. Es importante resaltar que todas las decisiones de estas rutinas son atómicas, por lo que cobertura de decisión sería el criterio más adecuado. Los resultados experimentales se muestran en las Tablas 6.19, 6.20, 6.21 y 6.22. En éstas se

Cota	Korat	Korat+	CC
3,0-3,3	552 (36)	532 (34)	10
3,0-3,6	2598 (462)	2333 (350)	18
4,0-4,4	3300 (116)	3205 (102)	19
4,0-4,8	30.408 (3656)	25.202 (2272)	27
5,0-5,5	19.035 (370)	18.461 (300)	27
6,0-6,6	106.650 (1206)	102.962 (892)	29
7,0-7,7	588203 (4011)	566075 (2,695)	31
5,0-5,10	375.620 (31.970)	282.067 (16.378)	29
8,0-8,8	3.027.008 (13.384)	2.912.167 (8143)	47
8,0-8,11	21.026.038 (362.890)	16.714.960 (160.483)	47
9,0-9,9	14.944.140 (43.992)	14.422.677 (24.257)	47
9,0-9,11	48.226.046 (417.340)	41.675.314 (187.511)	49
10,0-10,10	75.236.610 (141.010)	73.048.869 (70.877)	49
11,0-11,11	394.857.826 (440.814)	385.920.181 (203.605)	49

Cuadro 6.15: Comparación `Korat` y `Korat+` para *Árboles Rojos y Negros (rutina `Insert`)*

Cota	Korat	Korat+
3,0-3,3	0,388s	0,366s
4,0-4,4	0,517s	0,442s
5,0-5,5	0,701s	0,736s
6,0-6,6	1,275s	1,143s
7,0-7,7	2,58s	2,474s
8,0-8,8	10,316s	9,709s
9,0-9,9	52,887s	49,007s
10,0-10,10	281,931s	271,16s

Cuadro 6.16: Tiempos de corrida de `Korat` y `Korat+` para *Árboles Rojos y Negros (rutina `Insert`)*

reportan, para varias cotas, tamaño de espacio explorado acompañado de la cantidad de estructuras válidas, y número de clases de equivalencia cubiertas. Además, se reportan tiempos de corrida de `Korat` y `Korat+`, graficando los mismos en los

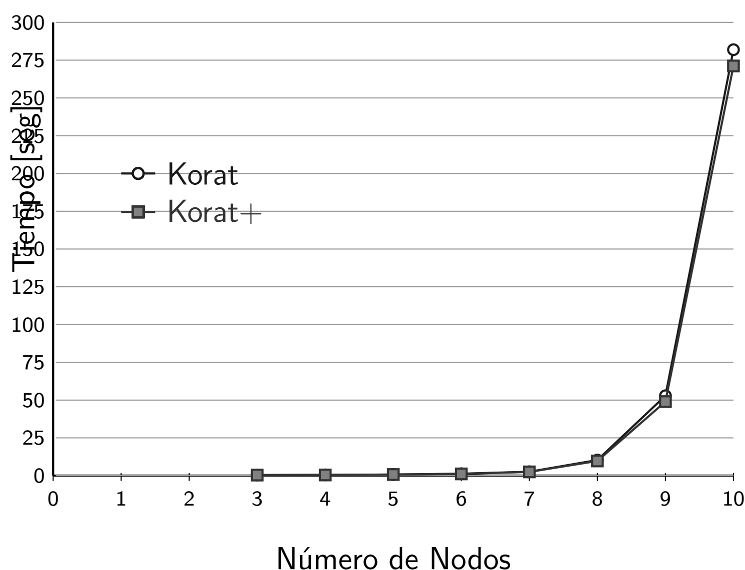


Figura 6.6: Tiempos de corrida de `Korat` y `Korat+` para *Árboles Rojos y Negros (rutina* ***`Insert`****)*

Cota	Korat	Korat+	CC
3,0-3,3	552 (36)	532 (34)	7
3,0-3,6	2698 (462)	2333 (350)	7
4,0-4,4	3300 (116)	3205 (102)	8
4,0-4,8	30.408 (3656)	25.202 (2272)	8
5,0-5,5	19.035 (370)	18.461 (300)	8
5,0-5,10	375.620 (31.970)	282.067 (16.378)	8
6,0-6,12	4.482.228 (284.220)	3.071.411 (123.490)	8
7,0-7,14	52.105.144 (2.551.486)	32.878.324 (956.280)	8
8,0-8,8	3.027.008 (13.384)	2.912.167 (8143)	8

Cuadro 6.17: Comparación `Korat` y `Korat+` para *Árboles Rojos y Negros (rutina* ***`find`****)*

Gráfico de líneas 6.8 y 6.9. Las cotas, en este caso indican el máximo número de nodos, el rango para el campo *size* del árbol, y el rango de valores posibles para las claves almacenadas en los nodos del árbol.

Cota	Korat	Korat+
3,0-3,6 6	0,462s	0,452s
4,0-4,8	1,082s	1,098s
5,0-5,10	2,167s	1,876s
6,0-6,12	15,674s	10,135s
7,0-7,14	180,46s	109,814s

Cuadro 6.18: Tiempos de corrida de `Korat` y `Korat+` para *Árboles Rojos y Negros (rutina* ***`find`****)*

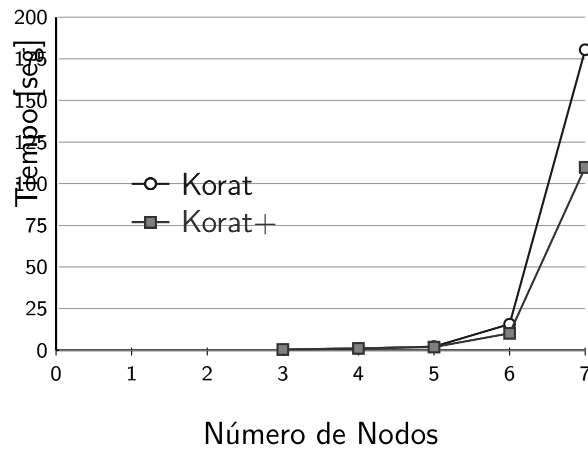


Figura 6.7: Tiempos de corrida de `Korat` y `Korat+` para *Árboles Rojos y Negros (rutina* ***`find`****)*

6.2. Reducciones al testing exhaustivo acotado basadas en el uso del invariante de representación

6.2.1. Generación independiente de sub-estructuras disjuntas

En el caso de la técnica de generación disjunta, la evaluación se basa sobre tres casos de estudio, correspondientes al análisis de rutinas sobre estructuras alojadas

Cota	Korat	Korat+	CC
5,0-5,5	41.540 (940)	38.040 (767)	9
5,0-5,10	1.720.830 (142.250)	1.076.259 (68.258)	9
6,0-6,6	371.700 (4386)	338.985 (3290)	9
6,0-6,12	32.702.676 (1.960.884)	18.830.994 (801.733)	9
7,0-7,7	3.301.956 (20.650)	3.015.006 (14.280)	9
7,0-7,14	601.871.102 (27.563.746)	323.780.633 (9.803.030)	9
8,0-8,8	29.007.816 (97.880)	26.601.485 (62.624)	9

Cuadro 6.19: Comparación `Korat` y `Korat+` para *Árboles binarios de búsqueda(rutina `insert`)*

Cota	Korat	Korat+
5,0-5,10	3,589s	2,636s
6,0-6,12	60,802s	32,653s
7,0-7,14	1175,809s	564,917s

Cuadro 6.20: Tiempos de corrida de `Korat` y `Korat+` para *Árboles binarios de búsqueda (rutina `insert`)*

en memoria dinámica:

- Binomial Heaps y la rutina `merge`, la cual dado dos *binomial heaps* los combina para obtener un tercero como salida.
- AvlTree y la rutina `AddAll`, la cual adiciona todos los elementos de una colección (en este caso una lista) dada al árbol.
- Node Caching Linked List y la rutina `getFirst`, la cual retorna el primer elemento de la lista.

Esta técnica es relevante para cualquier herrramienta de generación exhaustiva acotada, en particular aquellas basadas en filtrado. Sin embargo, es importante mencionar que los experimentos fueron realizados utilizando `Korat` [11].

Para cada uno de los tres casos de estudio seleccionados, se obtienen diferentes `repOks` predicando sobre porciones separadas de la estructura original o sobre parámetros disjuntos (diferentes parámetros de una rútina) como en el caso de la

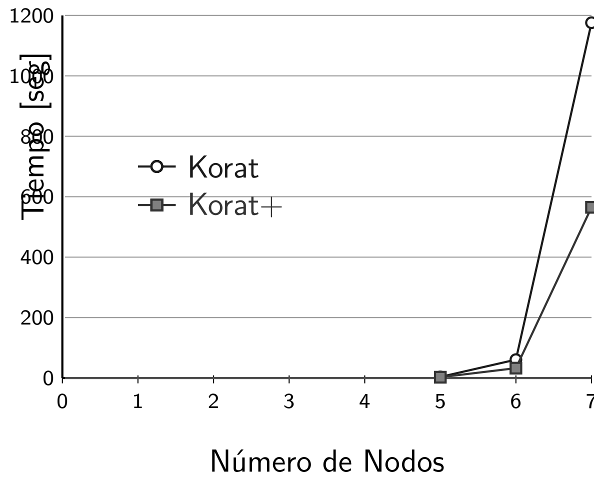


Figura 6.8: Tiempos de corrida de `Korat` y `Korat+` para *Árboles binarios de búsqueda (rutina* ***`insert`****)*

Cota	Korat	Korat+	CC
2,0-2,2	62 (10)	57 (10)	6
2,0-2,4	228 (68)	198 (60)	7
3,0-3,3	534 (45)	500 (43)	8
3,0-3,6	4290 (822)	3331 (586)	8
4,0-4,4	4660 (204)	4314 (181)	8
4,0-4,8	87.032 (10.600)	60.102 (6121)	8
5,0-5,5	41.540 (940)	38.040 (767)	8
5,0-5,10	1.720.830 (142.250)	1.076.259 (68.258)	8
6,0-6,6	371.700 (4386)	338.985 (3290)	8
6,0-6,12	32.702.676 (1.960.884)	18.830.994 (801.733)	8
7,0-7,7	3.301.956 (20.650)	3.015.006 (14.280)	8

Cuadro 6.21: Comparación `Korat` y `Korat+` para *Árboles binarios de búsqueda (rutina* ***`search`****)*

rutina `merge` de *binomial heaps* y la rutina `AddAll` de *AvlTree*. Estos `repOks` son utilizados para generar entradas automáticamente usando `Korat`, de manera exhaustiva acotada para diferentes cotas con el objetivo de analizar escalabilidad.

Cota	Korat	Korat+
2,0-2,4	0,262s	0,264s
3,0-3,6	0,425s	0,352s
4,0-4,8	0,993s	1,019s
5,0-5,10	3,427s	2,643s
6,0-6,12	56,518s	31,852s

Cuadro 6.22: Tiempos de corrida de `Korat` y `Korat+` para *Árboles binarios de búsqueda (rutina `search`)*

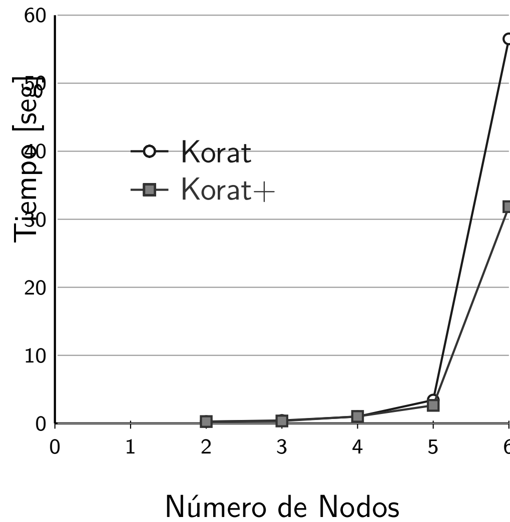


Figura 6.9: Tiempos de corrida de `Korat` y `Korat+` para *Árboles binarios de búsqueda (seach)*

Para correr todos los experimentos se utilizó un procesador Intel Core i7 3.2GHz con 16GB de RAM.

Para estos experimentos, los `repOks` son procesados utilizando *slicing* directo con respecto a las partes de la estructura visitadas por cada sentencia en el código del `repOk()` procesado. Más precisamente, el *slicing* realizado esta basado sobre el *grafo de definición y uso* [20] construido sobre la implementación del `repOk()`. Los *grafos de definición y uso* son construido desde el *grafo de flujo de control* del código bajo análisis (en este caso `repOk()`) incorporando información acerca de las variables que

son usadas (y definidas) en cada nodo y arista del grafo. Luego, estos grafos son usados para computar todas las sentencias en `repOk()` que pueden afectar el valor de algunas variables, relacionadas a alguna parte de la estructura.

Además de los `repOks` "locales", un `repOk` "global" es obtenido tomando esas sentencias en las cuales más de una sub-estructura disjunta podría estar involucrada.

Debido a que se utiliza la herramienta `Korat`, en primer lugar se obtienen vectores candidatos representando sub-estructuras. Estos vectores candidatos válidos retornados por `Korat` sobre diferentes sub-estructuras se memorizan, para luego combinarlos con el propósito de construir vectores candidatos (o objetos sobre ellos) correspondientes a la estructura original, y chequear, cuando sea necesario, el `repOk` "global" (si hay restricciones extras, a parte de las restricciones propias de cada sub-estructura). Por cada caso de estudio, se mide el tiempo necesario para realizar este proceso, lo cual incluye el tiempo requerido para construir todas las sub-estructuras, más el tiempo requerido para combinarlas.

Finalmente, la información acerca de las partes de la estructura que son disjuntas en la estructura es requerida a los desarrolladores. En otras palabras, el proceso usado para generar estructuras mediante generación separada asume, en estos experimentos, que esta información es provista como una entrada.

Listas circulares doblemente encadenadas con memoria cache (`Node Caching Linked Lists`). El primer caso de estudio, involucra una estructura llamada NodeCachingLinkedList, la cual consiste en una lista circular doblemente encadenada con memoria cache. Esta estructura ya fue presentada en la Sección 4.2.1 del Capítulo 4 . Las Tablas 6.23 y 6.24 muestran, para diferentes cotas (las cotas especifican el número máximo de nodos en la estructura, el tamaño máximo de la lista circular, el tamaño máximo de la *cache* y el número de claves permitidas en la estructura), el número de candidatos explorados, el tamaño de la *suite* exhaustiva acotada (EA) y el tiempo empleado durante la generación. Esta información se presenta para ambos, generación exhaustiva acotada y *generación disyunta*, es decir, generación independiente (y posterior combinación) de las sub-estructuras de `NodeCachingLinkedList`. Como se puede notar el número de entradas válidas, es decir, el tamaño de las *suites* exhaustivas (EA), se reporta sólo una vez, debido a que es el mismo para ambos casos (la técnica es *sound* y completa). Estas dos tablas muestran, como el número de estructuras visitadas, y el tiempo consumido en esta visita, crece a medida que las cotas también lo hacen en dos dimensiones diferentes. La Tabla 6.23 muestra, como estos números cambian cuando el número de nodos crece y el número de claves permitidas se mantiene constante. La Tabla 6.24 muestra lo contrario, es

decir, como estos números cambian cuando el número de nodos es constante y el número de claves permitidas es variable. Las filas resaltadas muestras los casos en los cuales generación disjunta supera, con respecto al tiempo, a la generación exhaustiva acotada tradicional. En este caso de estudio, son evidentes las diferencias, con respecto a cantidad de candidatos explorados, de ambas técnicas de generación. Por ejemplo, para cota 16,7,8,2, se reducen los 152.594.160 candidatos explorados por la generación exhaustiva acotada, a 292.988 candidatos visitados, lo cual se ve significativamente reflejado en el tiempo consumido (274,444s vs. 9,534s). Las Figuras 6.10 y 6.11 proveen una representación gráfica de la información presentada en las Tablas 6.23 y 6.24 respectivamente. Aquí se ve claramente como se logra un importante incremente en la escalabilidad de la generación.

Cota	EA	Korat		Generación Disj.	
		Explorados	Tiempo	Explorados	Tiempo
4,1,2,2	132	1485	0,207s	183	0,42s
6,2,3,2	1014	13.610	0,309s	859	0,45s
8,3,4,2	6840	102.426	0,654s	3254	0,524s
10,4,5,2	43.560	698.155	1,598s	11.024	0,65s
12,5,6,2	269.724	4.433.071	6,872s	34.649	1,021s
14,6,7,2	1.646.058	26.602.064	43,54s	102.974	2,317s
16,7,8,2	9.967.920	152.594.160	274,444s	292.988	9,534s
18,8,9,2	60.108.828	844.607.873	1768,611s	805.192	54,092s

Cuadro 6.23: Comparación generación exhaustiva acotada y generación separada acotada de sub-estructuras disjuntas para `NodeCachingLinkedList`, cuando el número de nodos se incrementa.

Binomial Heaps. Este caso de estudio, corresponde a la rutina `merge` sobre binomial heaps, utilizada también en la sección anterior. Como ya se mencionó previamente, esta clase de generación exhaustiva acotada es típica de situaciones que involucran probar rutinas que reciben más de una estructura como parámetro (por ejemplo, unión e intersección de operaciones sobre conjuntos). Para el caso particular de los binomial heaps, `merge` es una de esas rutinas, ya que manipula un par de binomial heaps. Esta rutina toma como parámetro dos binomial heap, y produce un tercer binomial heap correspondiente a la unión de los parámetros. En casos como este, el `repOk()` viene dado de manera modularizada, ya que simplemente corres-

Cota	EA	Korat		Generación Disj	
		Explorados	Tiempo	Explorados	Tiempo
8,3,4,2	6840	102.426	0,654s	3254	0,525s
8,3,4,3	60.860	902.178	1,634s	11.604	0,955s
8,3,4,4	353.340	5.101.182	6,216s	31.094	0,952s
8,3,4,5	1.515.150	20.787.504	22,392s	68.948	1,754s
8,3,4,6	5.222.000	67.420.914	70,544s	134.214	3,604s
8,3,4,7	15.289.560	185.298.006	192,722s	237.764	8,339s
8,3,4,8	39.475.620	449.407.158	477,14s	392.294	19,799s
8,3,4,9	92.246.330	988.610.772	1018,966s	612.324	48,154s

Cuadro 6.24: Comparación generación exhaustiva acotada y generación separada acotada de sub-estructuras disjuntas para `NodeCachingLinkedList`, cuando el número de claves permitidas se incrementa.

ponde a la conjunción de los `repOk()` de ambos parámetros. La Tabla 6.25 muestra, para diferentes cotas, el número de candidatos explorados, el tamaño de las *suites* generadas (EA) y el tiempo consumido por la generación para ambas técnicas: generación exhaustiva tradicional y generación disjunta. Como en el caso anterior, la cantidad de estructuras encontradas válidas es reportada sólo una vez ya que coincide en ambos casos. Las cotas en este caso, corresponden al número máximo de elementos para ambos *heaps*, y el rango para las claves de los nodos, que va desde cero al valor especificado. Nuevamente, la técnica de *generación disjunta* muestra un buen desempeño, con respecto a estructuras visitadas y consecuentemente en tiempo consumido, en comparación con generación exhaustiva acotada. Por ejemplo, para cota 6 se exploran 42.815 candidatos, comparado con 274.808.123 candidatos visitados por la generación exhaustiva acotada tradicional. La Figura 6.12 refleja gráficamente estos resultados.

Árboles binarios de búsqueda balanceados (`AVL Trees`). El último caso de estudio presentado para la evaluación de la técnica de generación exhaustiva disjunta, corresponde a `AVLTree` y `SinglyLinkedList`. La idea es generar entradas para probar la rutina `addAll`, la cual recibe como parámetro un par compuesto por estas dos estructuras. Esta operación consiste en agregar todos los elementos de la colección (una lista en este caso) al árbol ordenado dado. Las Tabla 6.26 y 6.27 muestran, para diferentes cotas (donde cada cota especifica el número máximo de nodos en el

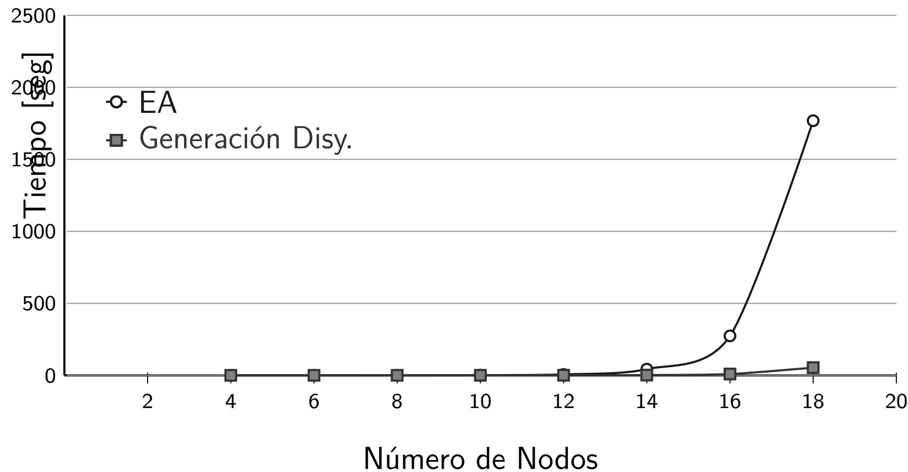


Figura 6.10: Comparación generación exhaustiva acotada y generación separada acotada de sub-estructuras disjuntas para `NodeCachingLinkedList`, cuando el número de nodos se incrementa. (representatión gráfica).

árbol, el rango del campo `size` del árbol, el número máximo de nodos en la lista, el rango del campo `size` de la lista y el número de claves en el árbol y en la lista), el número de candidatos explorados, el tamaño de las *suites de tests* exhaustivas acotadas (EA), y el tiempo consumido durante la generación. Esta información se presenta tanto para generación exhaustiva acotada como para generación separada de los árboles y las listas. Los resultados experimentales nuevamente señalan los beneficios de la técnica presentada. Las Figuras 6.13 y 6.14 resaltan gráficamente la información presentada en las Tablas 6.26 y 6.27 respectivamente.

6.2.2. Reducción de *suites de tests* exhaustivas acotadas usando criterios sobre `repOk()`

Para el caso de *reducción de suites de tests basada en* ***`repOk()`***, la evaluación se basó en varios casos de estudio, correspondientes al análisis de varias rutinas sobre algunas estructuras de datos alojadas en memoria dinámica seleccionadas:

- *Binomial heaps*,
- *Listas doblemente encadenadas*,
- *Árboles binarios de búsqueda* y

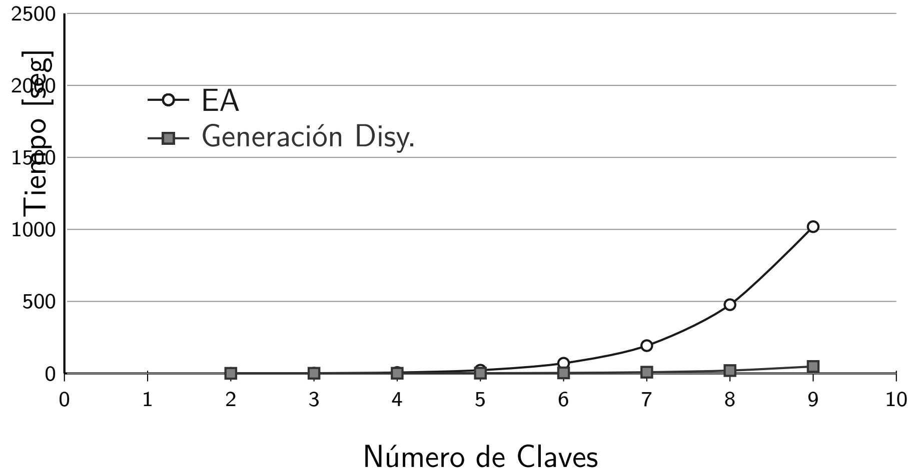


Figura 6.11: Comparación generación exhaustiva acotada y generación separada acotada de sub-estructuras disjuntas para `NodeCachingLinkedList`, cuando el número de claves permitidas se incrementa. (representatión gráfica).

- *Árboles rojos y negros*

Los experimentos fueron realizados utilizando 3 diferentes criterios de cobertura de caja blanca sobre `repOk()`: *cobertura de decisión*, *cobertura de caminos* y una variante de cobertura de decisión: *cobertura de decisión con conteo* (CDC). Es importante subrayar que como se compara con *suites* exhaustivas acotadas, es posible determinar de manera precisa cuales son las clases de equivalencia viables para cada criterio de cobertura, por ejemplo, se puede determinar cuales son los caminos que la *suite* exhaustiva acotada cubre, lo cual es necesario durante el proceso de reducción.

Como se mencionó arriba, se utilizó el criterio de cobertura de *decisión con conteo (CDC)*. Este criterio tiene en cuenta el número de veces que cada decisión en el programa bajo análisis, evalúa en verdadero y en falso, y es útil en este contexto debido a que, en general, hay una relación entre el tamaño de la estructura y el número de veces que un punto de decisión particular en el `repOk()`, se evalúa en verdadero o en falso. Esto se puede ver claramente si se piensa en condiciones de estructuras iterativas. Como consecuencia, cuando el tamaño de la estructura crece, el número de clases de equivalencia también se incrementa, y por lo tanto también la variedad de casos de *test* en las *suites* reducidas.

Para realizar los experimentos presentados, se tomaron los `repOk()` para cada una de las estructuras, y automáticamente fueron instrumentados para obtener, des-

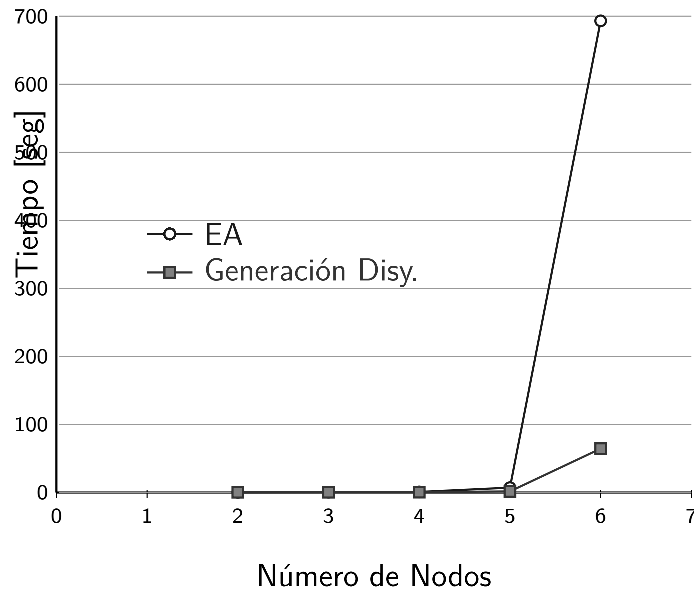


Figura 6.12: Comparación generación exhaustiva acotada y generación separada acotada de sub-estructuras disjuntas para `Binomial Heap`(representación gráfica).

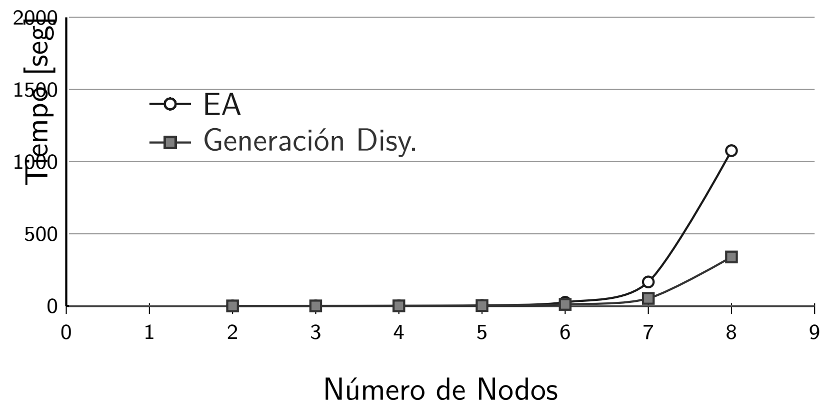


Figura 6.13: Comparación generación exhaustiva acotada y generación separada acotada de sub-estructuras disjuntas para pares compuestos por un árbol ordenado balanceado (AVL tree) y una lista simplemente encadenada, cuando el número de nodos en estas estructuras se incrementa. (representación gráfica).

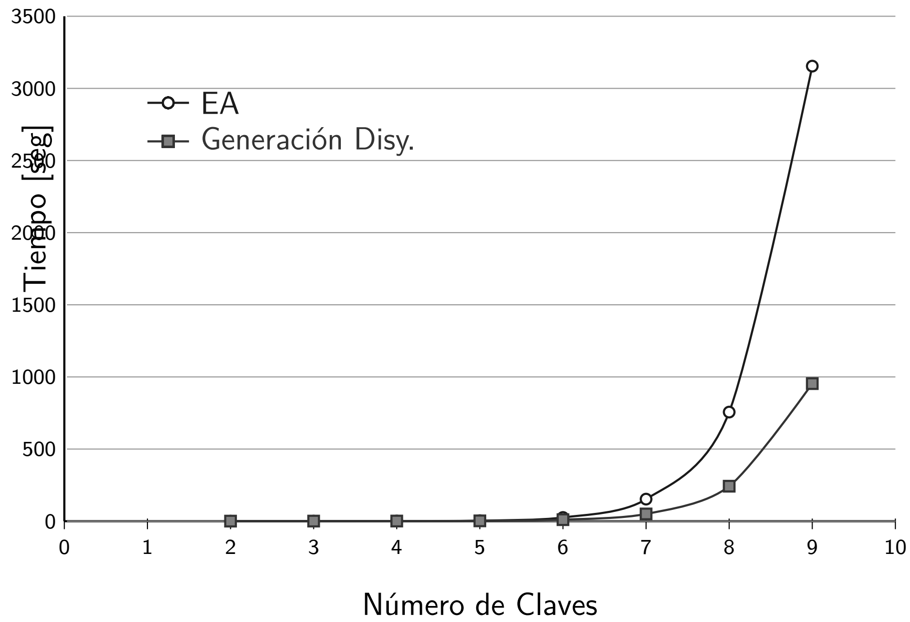


Figura 6.14: Comparación generación exhaustiva acotada y generación separada acotada de sub-estructuras disjuntas para pares compuestos por un árbol ordenado balanceado (AVL tree) y una lista simplemente encadenada, cuando el número de claves permitidas en estas estructuras se incrementa. (representación gráfica).

Cota	EA	Korat		Generación Disj.	
		Explorados	Tiempo	Explorados	Tiempo
2,2	36	348	0,26s	58	0,427s
3,3	784	5389	0,499s	235	0,45s
4,4	14.400	150.448	0,866s	1666	0,559s
5,5	876.096	3.125.314	7,187s	8122	1,641s
6,6	57.790.404	274.808.123	693,116s	42.815	64,499s
7,7		TO	TO	261.788	13470,976s

Cuadro 6.25: Comparación generación exhaustiva acotada y generación separada acotada de sub-estructuras disjuntas para `Binomial Heaps`.

Cota	EA	Korat		Generación Disj.	
		Explorados	Tiempo	Explorados	Tiempo
2,0,2,3,0,2,6	1591	2925	0,303s	442	0,546s
3,0,3,4,0,3,6	14.763	22.878	0,533s	3951	0,669s
4,0,4,5,0,4,6	181.935	243.217	1,253s	22.122	1,052s
5,0,5,6,0,5,6	1.427.643	1.790.743	4,255s	80.288	2,62s
6,0,6,7,0,6,6	8.789.959	10.724.428	25,836s	232.025	10,719s
7,0,7,8,0,7,6	52.739.911	63.672.425	166,652s	772.602	52,568s
8,0,8,9,0,8,6	316.439.623	380.739.460	1076,359s	2.511.653	339,56s

Cuadro 6.26: Comparación generación exhaustiva acotada y generación separada acotada de sub-estructuras disjuntas para pares compuestos por un árbol ordenado balanceado (`AVL tree`) y una lista simplemente encadenada, cuando el número de nodos en estas estructuras se incrementa.

de una llamada a `repOk()` sobre una estructura válida dada, la clase de equivalencia a la cual la estructura pertenece, para cada criterio seleccionado. Los `repOks` instrumentados se ejecutaron sobre las entradas de la *suite de tests* exhaustiva acotada para obtener información a acerca de la clase de equivalencia a la cual cada entrada pertenece, luego, utilizando esta información se construyeron *suites de tests* reducidas seleccionando, de la *suite* exhaustiva acotada, alguno de los casos de *test* por cada clase de equivalencia viable. En particular, se redujeron las *suites de tests* exhaustivas acotadas a uno y dos órdenes de magnitud, es decir, al 10 % y 1 % del tamaño original de las *suites*. Para correr todos los experimentos se utilizó un

Cota	EA	Korat		Generación Disj.	
		Explorados	Tiempo	Explorados	Tiempo
6,0,6,7,0,6,2	635	6884	0,339s	5609	0,628s
6,0,6,7,0,6,3	12.023	27.594	0,511s	10.505	0,829s
6,0,6,7,0,6,4	136.525	202.784	1,175s	26.313	1,124s
6,0,6,7,0,6,5	1.210.922	1.569.144	3,882s	75.438	2,384s
6,0,6,7,0,6,6	8.789.959	10.724.428	25,836s	232.025	10,719s
6,0,6,7,0,6,7	52.020.403	61.244.650	152,977s	685.649	50,262s
6,0,6,7,0,6,8	254.354.457	292.245.608	756,38s	1.831.897	242,524s
6,0,6,7,0,6,9	1.053.448.702	1.188.984.604	3154,268s	4.389.842	953,695s

Cuadro 6.27: Comparación generación exhaustiva acotada y generación separada acotada de sub-estructuras disjuntas para pares compuestos por un árbol ordenado balanceado (`AVL tree`) y una lista simplemente encadenada, cuando el número de claves permitidas en estas estructuras se incrementa.

procesador Intel Core i7 3.2GHz con 16GB de RAM.

La selección de casos de *test* fue realizada tomando como máximo N_r/M entradas por cada clase de equivalencia, donde N_r es el tamaño de la *suite* reducida (por ejemplo 10 % de la *suite* exhaustiva acotada) y M es el número de clases de equivalencia. En todos los casos (reducción al 10 % y al 1 %), cuando la *suite de tests* exhaustiva acotada fue demasiado pequeña para reducirla al 10 % (o 1 %) de su tamaño original, se tomó como mínimo una entrada por cada clase de equivalencia viable. Además, el número de clases de equivalencia cubiertas (CC), es decir la cantidad de clases de equivalencia diferentes viables por cada criterio seleccionado, también se reporta como parte de los resultados experimentales

Para medir efectividad del enfoque, se tomaron algunos rutinas manipulando las estructuras seleccionadas para el análisis: `merge`, `insert`, `delete` y `find`, sobre binomal heaps, `isPalindromic` sobre listas doblemente encadenadas, `insert`, `delete` y `search` sobre árboles binarios de búsqueda y `add`, `remove` y `contains` sobre árboles rojos y negros, y se generaron mutantes. Los resultados reportan la comparación, en cuanto a la habilidad de matar mutantes, de las diferentes *suites*: exhaustiva acotada, *suites de tests* reducidas y *suite de tests* “uno por cada clase de equivalencia (UPC)”, la cual consisten de exactamente una entrada por cada clase de equivalencia cubrible (es decir, la *suite* minimal con la misma cobertura que la *suite* exhaustiva acotada correspondiente).

Es valioso destacar la relación entre clases cubiertas (CC) y *suites* "uno por clase de equivalencia" (UPC). CC es el número de clases de equivalencia cubiertas por cada criterio seleccionado. Las *suites* UPC son aquellas que se construyen tomando un caso de *test* por cada clase de equivalencia cubierta. De manera más precisa, los casos de *test* seleccionados para construir las *suites* "uno por clase de equivalencia" son los primeros casos de *test* generados por cada clase de equivalencia viable.

Por último, es importante mencionar que este enfoque se basa en reducir *suites de tests* exhaustivas acotadas ya generadas, es decir no ataca el problema de la generación exhaustiva acotada en sí.

Binomial Heap (`merge`) Este caso de estudio, envuelve probar la rutina `merge` sobre *Binomial Heap*. Como se vio en las secciones anteriores, esta rutina toma como parámetro un par de *binomial heaps*, y produce un tercer *binomial heap*, el cual corresponde a la unión de los dos parámetros. Este es un ejemplo en el cual las *suites* exhaustivas acotadas crecen rápidamente, haciendo que el *testing* utilizando estas *suite*, resulte impracticable. La Tabla 6.28 muestra, para varios cotas, el tamaño de las *suites* exhaustivas acotadas (EA) y las *suites* con reducción al 10 % y al 1 % del tamaño original de la *suite*, para los tres criterios de caja blanca mencionados aplicados al código de `repOk()`. Para cada criterio, se reporta el número de clases de equivalencia que fueron cubiertas por la *suite* (CC). Las cotas especifican el número máximo de elementos para ambos *heaps*, y el rango permitido para las claves de los nodos (desde cero al número especificado).

Como se mencionó en los inicio de esta sección, se mide la efectividad de las *suites* usando *testing de mutación*. Se mutó la rutina `merge` y se obtuvo un total de 113 mutantes no equivalentes al programa original. Luego, se utilizaron estos mutantes para evaluar la habilidad de matar mutantes de las *suites* exhaustivas acotadas, las *suites* reducidas y las *suites* óptimas: "uno por cada clase de equivalencia" (UPC), es decir, la *suites* generadas tomando exactamente una entrada por cada clase de equivalencia (CC). La Tabla 6.29 reporta los resultados obtenidos. En esta tabla, se indican la cantidad de mutantes vivos y se resaltan los casos en los cuales la efectividad (en términos de mutantes muertos) de las *suites* reducidas coincide con el de las *suites* exhaustivas acotada. Como se puede notar, en este caso las *suites* reducidas, para todos los criterios bajo análisis fueron, en la mayoría de los casos, tan efectivas como las *suites* exhaustivas acotadas, con respecto a *testing de mutación*, incluso para las reducciones al 1 % del tamaño original.

Cota	EA	Cob. Decisión		
		10 %	1 %	CC
2,2	36	3	3	3
3,3	784	76	4	4
4,4	14.400	1200	144	4
5,5	876.096	49.420	7506	4
6,6	57.790.404	2.455.826	342.166	4
Cota	EA	Cob. Dec c/Conteo		
		10 %	1 %	CC
2,2	36	9	9	9
3,3	784	59	16	16
4,4	14.400	1060	119	25
5,5	876.096	42.500	6460	36
6,6	57.790.404	1.993.860	315.698	49
Cota	EA	Cob. Caminos		
		10 %	1 %	CC
2,2	36	9	9	9
3,3	784	59	16	16
4,4	14.400	1060	119	25
5,5	876.096	42.500	6460	36
6,6	57.790.404	1.993.860	315.698	49

Cuadro 6.28: Tamaño de las *suites* exhaustivas acotadas y las reducciones basadas en `repOk()`, para probar la rutina `merge` de binomial heaps.

Binomial Heaps (`insert`, `delete` y `find`) El segundo caso de estudio corresponde a las rutinas `insert`, `delete` y `find` que manipulan un *binomial heap*. Estas rutinas corresponden a las operaciones de inserción, eliminación y búsqueda, respectivamente, sobre *binomial heaps*. La Tabla 6.30 muestra, para varios cotas, el tamaño de las *suites* exhaustivas acotadas (EA) y las *suites* con reducciones al 10 % y 1 % del tamaño original, para los tres criterios de caja blanca seleccionados aplicados a `repOk()`; nuevamente indicamos la cantidad de clases de equivalencia cubiertas por cada criterio (CC). En este caso, la cota simplemente indica el tamaño de los *binomial heap* correspondientes.

Para evaluar la efectividad de las diferentes *suites* (exhaustiva acotada, *suites de tests* reducidas al 10 % y al 1 % del tamaño de la *suite* orginal y la *suite* generada

Cota	EA	Cob. Decisión			Cob. Dec c/Conteo			Cob. Caminos		
		10 %	1 %	UPC	10 %	1 %	UPC	10 %	1 %	UPC
2,2	34	96	96	96	38	38	38	38	38	38
3,3	4	9	82	82	4	10	10	4	10	10
4,4	3	3	9	82	3	3	8	3	3	8
5,5	3	3	3	82	3	3	8	3	3	8
6,6	3	3	3	82	3	3	8	3	3	8

Cuadro 6.29: Efectividad, con respecto a *testing de mutación*, de las *suites de tests* exhaustivas y de las *suites de tests* reducidas sobre la rutina `merge` (se reportan mutantes vivos).

tomando exactamente una entrada por cada clase de equivalencia cubierta (UPC)), se mutaron las rutinas `insert`, `delete` y `find` (el número de mutantes no equivalentes al programa original obtenidos fue 97, 170 y 28 respectivamente). La Tabla 6.31 reporta los resultados para este caso de estudio. Para cada rutina, se reporta la cantidad de mutantes sobre la cual se realizó la medición (M). En este caso, las *suites de tests* reducidas no son tan efectivas como en el caso anterior, especialmente para el caso de `delete`. Sin embargo, los resultados continúan siendo buenos, teniendo en cuenta el tamaño de las *suites* reducidas. Por ejemplo, para cota 8 y *cobertura de decisión con conteo*, la *suite* reducida al 10 % falla en detectar un único mutante (17 vs 18 de 170 mutantes totales) comparado con la *suite* exhaustiva acotada. El mutante que no es detectado en este caso corresponde a la inserción de un "operador de post-incremento" en el código bajo prueba. El problema es que capturar este mutante depende exactamente de cual valor es eliminado y donde este valor esta ubicado en el *heap*, es decir en que nodo esta almacenado. Debido a que el criterio de cobertura sobre `repOk` sólo tiene en cuenta la estructura del *binomial heap*, pero no donde el valor a ser eliminado esta ubicado, se consiguen clases de equivalencia demasiado gruesas y las entradas de *test* que ejercitaban el código mutado fueron filtradas de su clase.

Listas doblemente encadenadas (`isPalindromic`) El siguiente caso de estudio corresponde a la rutina `isPalindromic`, la cual chequea si una secuencia de números enteros dada (implementada sobre listas doblemente encadenada) es palíndroma. La Tabla 6.32 muestra, para varias cotas, el tamaño de la *suite* exhaustiva acotada (EA), y de las *suites* reducuidas al 10 % y al 1 %, para los tres criterios de caja

Cota	EA	Cob. Decisión			Cob. Dec c/Conteo			Cob. Caminos		
		10 %	1 %	CC	10 %	1 %	CC	10 %	1 %	CC
2	12	3	3	3	3	3	3	3	3	3
3	84	8	4	4	8	4	4	8	4	4
4	480	40	4	4	40	5	5	40	5	5
5	4680	264	38	4	339	40	6	339	40	6
6	45.612	1938	270	4	2772	367	7	2772	367	7
7	751.912	37.650	3814	4	33.052	4947	8	33.052	4947	8
8	4.829.952	241.568	24.220	4	217.662	29.494	9	217.662	29.494	9

Cuadro 6.30: Tamaño de las *suites* exhaustivas acotadas y las reducciones basadas en `repOK` para probar las rutinas `insert`, `delete` y `find` de binomial heaps.

blanca aplicados al `repOk()`. Además, se indica el número de clases de equivalencia cubiertas en cada caso (columna CC). Las cotas en este caso corresponde al número de entradas en la lista, el rango para el campo `size`, y el número de valores enteros permitidos en la lista.

Como se hizo en el resto de las casos, se mutó la rutina `isPalindromic`, obteniendo 21 mutantes no equivalentes al programa original. La Tabla 6.33 muestra los resultados de el análisis para este caso de estudio, indicando los mutantes que quedan vivos, y resaltando aquellos casos en los cuales las *suites* reducidas coinciden en efectividad con las *suites de tests* exhaustivas acotadas. Como se puede observar, las *suites de tests* reducidas, en la mayoría de los casos, son tan efectivas como las exhaustivas acotadas, incluso para las *suites* reducidas al 1 % del tamaño original.

Árboles binarios de búsqueda (`delete`, `insert` y `search`) En el siguiente caso de estudio se consideran árboles binarios de búsqueda y las rutinas principales de esta estructura: `insert`, `delete` y `search`, las cuales insertan, eliminan y buscan, respectivamente, un valor entero dentro del árbol. La Tabla 6.34 muestra, para varias cotas, el tamaño de varias *suites*, y el número de clases cubiertas. Las cotas indican el número máximo de nodos en el árbol, el rango para el campo `size` del árbol, y el número de claves permitidas en cada nodo.

Nuevamente, se mutaron las rutinas `insert`, `delete` y `search` (el número de mutantes obtenidos fue 9, 24 y 4 respectivamente) y se utilizaron estos mutantes para evaluar la efectividad de las *suites*. La Tabla 6.35 reporta los resultados obtenidos. Para cada rutina en la tabla, se reporta la cantidad de mutantes sobre la cual se

Cota	Op (M)	EA	Cob. Decisión			Cob. Dec c/conteo			Cob. Caminos		
			10 %	1 %	UPC	10 %	1 %	UPC	10 %	1 %	UPC
2	insert(97)	34	42	42	42	42	42	42	42	42	42
	delete(170)	110	138	138	138	138	138	138	138	138	138
	find(28)	0	12	12	12	12	12	12	12	12	12
3	insert(97)	23	24	32	32	24	32	32	24	32	32
	delete(170)	67	92	135	135	92	135	135	92	135	135
	find(28)	0	8	12	12	8	12	12	8	12	12
4	insert(97)	23	23	32	32	23	32	32	23	32	32
	delete(170)	63	85	135	135	87	135	135	87	135	135
	find(28)	0	6	12	12	6	12	12	6	12	12
5	insert(97)	23	23	23	32	23	23	32	23	23	32
	delete(170)	47	76	87	135	51	71	135	51	71	135
	find(28)	0	2	6	12	2	6	12	2	6	12
6	insert(97)	23	23	23	32	23	23	32	23	23	32
	delete(170)	19	51	85	135	34	39	101	34	39	101
	find(28)	0	2	6	12	0	3	12	0	3	12
7	insert(97)	23	23	23	32	23	23	32	23	23	32
	delete(170)	17	23	68	135	21	35	101	21	35	101
	find(28)	0	0	5	12	0	0	12	0	0	12
8	insert(97)	23	23	23	32	23	23	32	23	23	32
	delete(170)	17	40	56	135	18	34	101	18	34	101
	find(28)	0	2	5	12	0	0	12	0	0	12

Cuadro 6.31: Efectividad, con respecto a *testing de mutación*, de las *suites de tests* exhaustivas y de las *suites de tests* reducidas sobre las rutinas `insert`, `delete` y `find` de binomial heaps (se reportan mutantes vivos para cada caso).

realizó la medición (M). En este caso de estudio, las *suites de tests* reducidas son, en muchos de los casos, tan efectivas como las *suites* exhaustivas acotadas, con menos efectividad en la rutina `delete`. Sin embargo, la habilidad para matar mutantes en esta rutina aún es muy buena en las *suites* reducidas al 10 %, con cobertura de decisión con conteo, donde casi se logra los mismos resultados que con las *suites* exhaustivas acotadas con cota 6,0,6,9 (2 vs. 0 de 24 mutantes). Estos dos mutantes corresponden a la inserción de un operador condicional y al reemplazo de de un operador. En este caso estos dos mutantes resultaron ser programas equivalentes entre sí. Estos mutantes no son detectados por las *suites* reducidas por razones

Cota	EA	Cob. Decisión		
		10 %	1 %	CC
4,0,4,4	156	8	2	2
4,0,4,8	820	42	5	2
5,0,5,5	1555	78	8	2
5,0,5,10	16.105	806	81	2
6,0,6,6	19.608	981	99	2
6,0,6,12	402.234	20.112	2012	2
7,0,7,7	299.593	14.980	1498	2
7,0,7,14	12.204.241	610.213	61.022	2
Cota	EA	Cob. Dec c/conteo		
		10 %	1 %	CC
4,0,4,4	156	10	4	4
4,0,4,8	820	50	7	4
5,0,5,5	1555	100	13	5
5,0,5,10	16.105	777	108	5
6,0,6,6	19.608	1035	136	6
6,0,6,12	402.234	15.786	2193	6
7,0,7,7	299.593	13.239	1781	7
7,0,7,14	12.204.241	402.933	55.918	7
Cota	EA	Cob. Caminos		
		10 %	1 %	CC
4,0,4,4	156	10	4	4
4,0,4,8	820	50	7	4
5,0,5,5	1555	100	13	5
5,0,5,10	16.105	777	108	5
6,0,6,6	19.608	1035	136	6
6,0,6,12	402.234	15.786	2193	6
7,0,7,7	299.593	13.239	1781	7
7,0,7,14	12.204.241	402.933	55.918	7

Cuadro 6.32: Tamaño de las *suites* exhaustivas acotadas y las reducciones basadas en `repOk()` para probar la rutina `isPalindromic` sobre listas doblemente encadenadas.

similares al caso de *binomial heaps*. Las entradas son distinguidas por condiciones estructurales del los árboles, número de hijos izquierdos y derechos no nulos. Cuando

Cota	EA	Cob. Decisión			Cob. Dec c/conteo			Cob. Caminos		
		10 %	1 %	OPC	10 %	1 %	OPC	10 %	1 %	OPC
4,0,4,4	13	13	21	21	13	20	20	13	20	20
4,0,4,8	13	13	21	21	13	13	20	13	13	20
5,0,5,5	11	13	20	21	11	13	20	11	13	20
5,0,5,10	11	13	13	21	11	11	20	11	11	20
6,0,6,6	11	11	13	21	11	11	20	11	11	20
6,0,6,12	11	11	13	21	11	11	20	11	11	20
7,0,7,7	11	11	11	21	11	11	20	11	11	20
7,0,7,14	11	11	11	21	11	11	20	11	11	20

Cuadro 6.33: Efectividad, con respecto a *testing de mutación*, de las *suites de tests* exhaustivas y de las *suites de tests* reducidas sobre la rutina `isPalindromic` de Listas doblemente encadenadas (se reportan mutantes vivos para cada caso).

se prueba la operación `delete`, las chances de ejercitar la parte mutada del código no dependen sólo de la estructura del árbol sino también del valor que se quiere borrar, y donde éste esta ubicado dentro del árbol.

Cota	EA	Cob. Decisión			Cob. Dec c/conteo			Cob. Caminos		
		10 %	1 %	CC	10 %	1 %	CC	10 %	1 %	CC
3,0,3,3	45	5	5	5	7	7	7	9	9	9
3,0,3,4	148	10	5	5	14	7	7	9	9	9
3,0,3,6	822	70	5	5	72	7	7	78	9	9
3,0,3,8	2760	228	25	5	242	21	7	248	27	9
5,0,5,8	29.416	1836	240	5	2634	278	16	2888	260	65
6,0,6,9	167.814	10.158	1095	5	14.430	1605	22	16.665	1576	197

Cuadro 6.34: Tamaño de las *suites* exhaustivas acotadas y las reducciones basadas en `repOK` para probar las rutinas `delete`, `insert` y `search` sobre árboles binarios de búsqueda

Árboles Rojos y Negros (`remove`, `add` y `contains`) El último caso de estudio que presentamos involucra rutinas manipulando árboles rojos y negros. Estas rutinas, al igual que en caso de los árboles binarios de búsqueda, son las rutinas de inserción,

Cota	Op (M)	EA	Cob. Decisión			Cob. Dec c/conteo			Cob. Caminos		
			10 %	1 %	UPC	10 %	1 %	UPC	10 %	1 %	UPC
3,0,3,3	delete(24)	2	12	12	12	12	12	12	12	12	12
	insert(9)	0	0	0	0	0	0	0	0	0	0
	search(4)	0	0	0	0	0	0	0	0	0	0
3,0,3,4	delete(24)	2	12	12	12	12	12	12	12	12	12
	insert(9)	0	0	0	0	0	0	0	0	0	0
	search(4)	0	0	0	0	0	0	0	0	0	0
3,0,3,6	delete(24)	2	12	12	12	12	12	12	12	12	12
	insert(9)	0	0	0	0	0	0	0	0	0	0
	search(4)	0	0	0	0	0	0	0	0	0	0
3,0,3,8	delete(24)	2	9	12	12	9	12	12	9	12	12
	insert(9)	0	0	0	0	0	0	0	0	0	0
	search(4)	0	0	0	0	0	0	0	0	0	0
5,0,5,8	delete(24)	0	9	16	16	9	12	12	9	12	12
	insert(9)	0	0	0	0	0	0	0	0	0	0
	search(4)	0	0	0	0	0	0	0	0	0	0
6,0,6,9	delete(24)	0	9	16	16	2	9	12	0	12	12
	insert(9)	0	0	0	0	0	0	0	0	0	0
	search(4)	0	0	0	0	0	0	0	0	0	0

Cuadro 6.35: Efectividad, con respecto a *testing de mutación*, de las *suites de tests* exhaustivas y de las *suites de tests* reducidas sobre la rutina `insert`, `delete` y `search` de árboles binarios de búsqueda (se reportan mutantes vivos para cada caso).

eliminación y búsqueda de un elemento en el árbol. La Tabla 6.36 muestra, para distintas cotas, el tamaño de las correspondientes *suites* y el número de clases de equivalencia cubiertas en cada caso. Las cotas indican el número máximo de nodos en el árbol, el rango válido para el campo `size` del árbol, y el número de claves permitidas en el árbol. En este caso, los caminos y tamaños de las *suites de tests* fueron, para algunas cotas, demasiado grandes para permitir hacer el análisis. Por ello, se cosnisderaron para este caso de estudio una versión acotada de cobertura de caminos (conocida como *cobertura de caminos simples*) [41], en la cual no se tienen en cuenta repeticiones de nodos.

Nuevamente, se mutaron las rutinas `remove`, `add` y `contains` (el número de mutantes obtenidos, no equivalentes al programa original, fue 138, 115 y 32, respec-

Cota	EA	Cob. Decisión			Cob. Dec c/conteo			Cob. Cam. Sim.		
		10 %	1 %	CC	10 %	1 %	CC	10 %	1 %	CC
4,0,4,4	164	14	7	7	16	16	16	108	108	108
4,0,4,8	6408	500	62	7	608	64	16	169	157	157
5,0,5,5	575	53	7	7	30	30	30	97	97	97
5,0,5,10	56.790	2732	496	7	5313	532	30	245	165	157
6,0,6,6	1962	174	14	7	184	16	46	113	113	113
6,0,6,12	412.140	10.411	2652	7	38.579	4017	46	505	229	157
7,0,7,7	6377	469	61	7	570	66	66	154	142	142
7,0,7,14	3.045.266	89.960	11.654	7	284.408	29.449	66	2211	465	157

Cuadro 6.36: Tamaño de las *suites* exhaustivas acotadas y las reducciones basadas en `repOK` para probar las rutinas `add`, `remove` y `contains` sobre árboles rojos y negros.

tivamente). Los resultados del análisis son reportados en la Tabla 6.37. Para cada rutina en la tabla, se reporta la cantidad de mutantes sobre la cual se realizó la medición (M).

En este caso de estudio, las *suites* reducidas muestran ser más efectivas en el caso de la rutina `contains` coincidiendo, en la mayoría de los casos, con la habilidad de matar mutantes de las *suites* exhaustivas acotadas. En las otras dos rutinas, si bien no se obtienen los mismos resultados que las *suites* exhaustivas acotadas, éstos son bastante buenos, por ejemplo para la rutina `add` con criterio de cobertura de decisión con conteo para cota 7,0,7,7 sólo 4 mutantes de 115 restan vivos en comparación con la *suite* exhaustiva acotada y en la rutina `remove` para cota 7,0,7,14 sólo 3 mutantes de 138 restan vivos comparado con la *suite* exhaustiva acotada. Acerca de los mutantes que son pasados por alto por las *suites* reducidas, este caso no difiere demasiado de los casos de estudio anteriores, las partes mutada del código están ubicadas de manea profunda en él y la habilidad de capturar estas mutaciones, depende exactamente de donde esta localizado el nodo ha ser borrado/insertado, pero esta propiedad no se tiene en cuenta dentro del criterio de cobertura aplicado al `repOk`.

Cota	Op (M)	EA	Cob. Decisión			Cob. Dec c/Conteo			Cob. Cam. Simp.		
			10 %	1 %	OPC	10 %	1 %	OPC	10 %	1 %	OPC
4,0,4,4	remove(138)	43	78	78	77	66	66	66	43	43	43
	add(115)	27	47	79	79	79	79	79	33	33	33
	contains(32)	2	4	5	5	5	5	5	2	2	2
4,0,4,8	remove(138)	43	62	77	77	61	66	66	78	78	78
	add(115)	27	29	32	79	29	47	79	42	51	51
	contains(32)	2	2	3	5	2	4	5	4	4	4
5,0,5,5	remove(138)	39	77	78	77	64	64	64	64	64	64
	add(115)	27	32	79	79	79	79	79	45	45	45
	contains(32)	2	3	5	5	5	5	5	4	4	4
5,0,5,10	remove(138)	39	51	73	77	48	63	64	78	78	78
	add(115)	25	29	31	79	28	32	79	38	47	67
	contains(32)	2	2	2	5	2	3	5	4	4	4
6,0,6,6	remove(138)	37	62	78	77	42	62	62	67	67	67
	add(115)	25	31	47	79	44	79	79	51	51	51
	contains(32)	2	2	4	5	3	5	5	4	4	4
6,0,6,12	remove(138)	37	49	56	77	46	57	62	78	78	78
	add(115)	25	29	29	79	26	29	79	36	38	67
	contains(32)	2	2	2	5	2	2	5	4	4	4
7,0,7,7	remove(138)	37	56	77	77	37	62	62	72	72	72
	add(115)	25	29	32	79	29	79	79	42	51	51
	contains(32)	2	2	3	5	2	5	5	4	4	4
7,0,7,14	remove(138)	37	46	51	77	40	46	62	72	78	78
	add(115)	25	31	47	79	25	29	79	36	38	68
	contains(32)	2	2	2	5	2	2	5	4	4	4

Cuadro 6.37: Efectividad, con respecto a *testing de mutación*, de las *suites de tests* exhaustivas y de las *suites de tests* reducidas sobre las rutinas `add`, `remove` y `contains` de árboles rojos y negros (se reportan mutantes vivos para cada caso).

6.3. Uso de *cotas ajustadas* en la generación exhaustiva acotada

En esta sección se evalúa la técnica basada en el uso de *cotas ajustadas* en la generación exhaustiva acotada. La experimentación se sienta sobre tres casos de estudio, correspondientes a las siguientes estructuras alojadas en memoria dinámica:

- *Árboles Rojos y Negros*
- *Árboles Binarios de Búsqueda*
- *Binomial Heap*

En todos los casos, se asumen dadas las cotas en orden *Breadth First Search*, las cuales fueron obtenidas del repositorio de cotas generadas por `TACO` [14].

Para cada uno de los tres casos de estudio seleccionados, se comparan los resultados arrojados por `KoratCotas` con `Korat` estándar para dos diferentes invariantes de representación (`repOks`): El primero de ellos es el que provee la distribución de `Korat` para cada una de las estructuras seleccionadas. Por otro lado, se evalua la técnica para otro `repOk`, el cual, si bien es una codificación completamente aceptable del invariante de representación, no respeta la configuración que `Korat` espera para explotar su mecanismo de poda. Con esta comparación se pretende mostrar como la técnica presentada es útil cuando no se ha escrito (sintácticamente) `repOk` de una manera especifica, la cual en general requiere conocimiento de como la herramienta funciona internamente.

Para cada caso de estudio, se eligió reportar, para diferentes *scopes* y diferentes `repOks`, el número de candidatos explorados, acompañado por el número de candidatos válidos encontrados (el cual se reporta una única vez para ambos algoritmos, ya que `KoratCotas` es *sound* con respecto a la cantidad de estructuras válidas generadas). Además, se reporta el tiempo consumido por la generación en cada caso. Esta es la medida más razonable para emplear, si se esta interesado en evaluar el nivel de contribución de la técnica en el proceso de filtrado estándar.

Para correr todos los experimentos se utilizó un procesador Intel Core i5 1.4GHz con 4GB de RAM, y los datos reportados corresponden a experimentos que terminan en un tiempo límite de 3 horas.

Árboles Rojos y Negros El primero de los casos de estudio involucra la generación de instancias de Árboles Rojos y Negros. La implementación de árboles rojos

y negros que se uso en estos experimentos es aquella que provee la distribución de Korat. Los resultados experimentales para este caso de estudio se muestran en las Tablas 6.38 y 6.39. La primera tabla muestra los resultados obtenidos utilizando el repOk proveniente de la distribución de Korat. Por otro lado, la Tabla 6.39 muestra los resultados utilizando un repOk alternativo el cual fue tomado de TACO [14]. En estas tablas se reportan, por un lado, la cantidad de vectores candidatos explorados, la cantidad de estos que son válidos, y el tiempo de corrida de KoratCotas en comparación con Korat estándar, para diferentes *scopes*. Las cotas indican el máximo número de nodos, el rango de valores para el campo size del árbol, y el rango de valores permitidos para los valores (claves) almacenados en los nodos del árbol. Los casos en donde KoratCotas supera (en tiempo de generación) a Korat fueron resaltados.

En este caso de estudio, son evidentes las diferencias para el caso en el que se usa un repOk alternativo al que ofrece la distribución de Korat. Por ejemplo, para tamaño 9,0,9,9 se reduce el tiempo de generación a 392,284 segundos, donde Korat tradicional toma más de 3 horas. Esto impacta positivamente en el tamaño máximo de árboles alcanzado, antes de llegar al tiempo límite impuesto. Por el contrario, no hay mejora significativa para el caso en el que se usa el repOK proveniente de Korat.

Scope	Encontrados	Korat		KoratCotas	
		Explorados	Tiempo	Explorados	Tiempo
2,0,2,2	7	45	0,371s	31	0,393s
3,0,3,3	16	248	0,381s	136	0,41s
4,0,4,4	41	1603	0,432s	1112	0,493s
5,0,5,5	115	8609	0,562s	6344	0,601s
6,0,6,6	327	37.872	0,64s	28.250	0.829s
7,0,7,7	911	163.728	1,034s	122.053	1,11s
8,0,8,8	2489	715.883	1,828s	527.426	1.768s
9,0,9,9	6753	3.209.989	5,628s	2.193.948	4,693s
10,0,10,10	18.439	15.219.176	26,191s	10.201.953	19,981s
11,0,11,11	51.242	75.814.869	146,7s	50.439.326	103,402s
12,0,12,12	146.073	385.422.689	734,474s	252.257.215	505,271s

Cuadro 6.38: Comparación Korat vs KoratCotas para *árboles Rojos y Negros* utilizando RepOk de la distribución de Korat

Scope	Encontrados	Korat		KoratCotas	
		Explorados	Tiempo	Explorados	Tiempo
2,0,2,2	7	78	0,365s	48	0,482s
3,0,3,3	16	738	0,382s	184	0,476s
4,0,4,4	41	7966	0,495s	1974	0,51s
5,0,5,5	115	94.471	0,757s	14641	0,775s
6,0,6,6	327	1.163.047	1,978s	225.687	1,96s
7,0,7,7	911	14.554.849	13,376s	3.458.960	5,135s
8,0,8,8	2489	183.490.261	173,775s	49.524.800	48,992s
9,0,9,9	6753	TO	TO	372.227.010	392,284s
10,0,10,10	18439	TO	TO	5.598.066.175	7210,058s
11,0,11,11	TO	TO	TO	TO	TO
12,0,12,12	TO	TO	TO	TO	TO

Cuadro 6.39: Comparación `Korat` vs `KoratCotas` para *árboles Rojos y Negros* utilizando un `RepOk` alternativo

Árboles Binarios de Búsqueda El segundo caso de estudio corresponde a la generación de instancias de árboles binarios de búsqueda. Ccomo en caso anterior se reportan las resultados con dos `repOks` diferentes: Uno de ellos tomado de la distribución de `Korat`, el otro fue tomado de `Fajita` [6].

Los resultados experimentales para este caso de estudio se muestran en las Tablas 6.40 y 6.41. En cada caso utilizando uno de los dos `repOks` mencionados arriba. En estas tablas se reportan, la cantidad de vectores candidatos explorados, la cantidad de estos que son válidos, y el tiempo de corrida de `KoratCotas` en comparación con `Korat` estándar, para diferentes *scopes*. Las cotas indican el máximo número de nodos, el rango de valores para el campo `size` del árbol, y el rango de valores permitidos para los valores (claves) almacenados en los nodos del árbol. Los casos en donde `KoratCotas` supera (en tiempo de generación) a `Korat` fueron resaltados. E

En este caso de estudio, nuevamente la técnica muestra buenos resultados sólo en el caso en donde se usa un `repOk` alternativo al que provee `Korat`. En este caso, se reduce el tiempo de generación en al menos 2 ordenes de magnitud (107,586 seg vs más de 3 horas (10.800 segundos)).

Binomial Heap Este caso de estudio, envuelve la generación de instancias de clase *Binomial Heap*. Esta estructura fue utilizada en reiteradas oportunidades en

Scope	Encontrados	Korat		KoratCotas	
		Explorados	Tiempo	Explorados	Tiempo
2,0,2,2	10	54	0.25s	43	0,256s
3,0,3,3	37	372	0.265s	323	0,276s
4,0,4,4	146	2.657	0.337s	2455	0,377s
5,0,5,5	599	19.477	0.544s	18.669	0,61s
6,0,6,6	2521	145.115	0.89s	141.926	0,999s
7,0,7,7	10.805	1.092.288	1.683s	1.079.779	1.599s
8,0,8,8	46.960	8.278.266	6.797s	8.229.337	6.807s
9,0,9,9	206.395	63.055.896	50.928s	62.864.731	50,001s
10,0,10,10	915.641	482.221.830	473.951s	481.475.163	419,253s
11,0,11,11			TO	TO	TO
12,0,12,12			TO	TO	TO

Cuadro 6.40: Comparación `Korat` vs `KoratCotas` para *Árboles binarios de búsqueda* utilizando `RepOk` de la distribución de `Korat`

Scope	Encontrados	Korat		KoratCotas	
		Explorados	Tiempo	Explorados	Tiempo
2,0,2,2	10	264	0,259s	51	0,267s
3,0,3,3	37	11408	0,396 s	684	0,33s
4,0,4,4	146	776335	0,989s	14.870	0,578s
5,0,5,5	599	74.592.776	67,027s	449.996	2,834s
6,0,6,6	2521	TO	TO	17.073.168	107,586s
7,0,7,7	10805	TO	TO	TO	TO

Cuadro 6.41: Comparación `Korat` vs `KoratCotas` para *Árboles binarios de búsqueda* utilizando un `RepOk` alternativo

esta tesis. En las Tablas 6.42 y 6.43 se muestran, para varios cotas y dos diferentes `repOks` (uno de ellos tomado de la distribución de `Korat` y el otro del `benchmark de Roops` [5]), el tamaño de las *suites* exhaustivas acotadas generadas (se reportan una única vez debido a que `KoratCotas` produce los mismas entradas que `Korat`), el tamaño del espacio explorado en cada caso y el tiempo consumido durante la generación. Las cotas especifican el número máximo de elementos en el *heaps*, y el rango permitido para las claves de los nodos (desde cero al número especificado). Nuevamente, los casos en donde `KoratCotas` supera (en tiempo de generación) a

Korat fueron resaltados.

Como en el resto de los casos de estudio, utilizando el repOk alternativo, KoratCotas reduce sustancialmente el espacio de estados explorados, por ejemplo, para binomial heaps de tamaño 8, se exploran 122.679.453 candidatos, en lugar de 1.483.194.820, lo cual impacta considerablemente en el tiempo consumido en la generación (1091,961 seg vs 177,832 seg).

Scope	Encontrados	Korat		KoratCotas	
		Explorados	Tiempo	Explorados	Tiempo
2	6	50	0,266s	41	0,281s
3	28	269	0,271s	240	0,288s
4	120	1273	0,282s	1162	0,315s
5	936	6100	0,409s	5777	0,486s
6	7602	32.980	0,542s	30.180	0,621s
7	107.416	215.860	0,787s	205.426	0,863s
8	603.744	1.113.028	1,967s	1.019.088	1,963s
9	8.746.120	10.840.751	18,345s	10.199.755	17,936s
10	117.157.172	146.608.835	254,348s	144.055.509	246,758s
11		TO	TO	TO	TO
12		TO	TO	TO	TO

Cuadro 6.42: Comparación Korat vs KoratCotas para *Binomial Heap* utilizando RepOk de la distribución de Korat

6.4. Amenazas a la Válidez

Esta sección se presentan algunas amenazas a la válidez de los experimentos que se realizaron. En principio, los casos de estudios seleccionados representan situaciones naturales de *testing* en el contexto de las estructuras de datos complejas alojadas en memoria dinámica (el principal blanco del *testing* exhaustivo acotado). Se eligieron estructuras de diferentes complejidades, incluyendo estructuras de datos con restricciones simples, intermedias y complejas (por ejemplo, listas encadenadas, árboles binarios de búsqueda y binomial heaps respectivamente). Debido a que, como se vió en capítulos anteriores, la forma en que repOk esta sintácticamente escrito afecta a la eficiencia de la técnica base, siempre que fue posible, las implementaciones fueron tomadas de la distribución de Korat. En aquellos casos en donde no

Scope	Encontrados	Korat		KoratCotas	
		Explorados	Tiempo	Explorados	Tiempo
2	6	110	0,258s	59	0,28s
3	28	1036	0,266s	639	0,291s
4	120	12.935	0,492s	5352	0,51s
5	936	183.780	2,502s	73.147	1,754s
6	7602	3.202.245	6,975s	419.167	4,274s
7	107.416	64.592.184	52,005s	8.526.322	13,588s
8	603.744	1.483.194.820	1091,961s	122.679.453	177,832s
9	8746.120	TO	TO	1.397.714.674	2851,767s
10		TO	TO	TO	TO

Cuadro 6.43: Comparación `Korat` vs `KoratCotas` para *Binomial Heap* utilizando un `RepOk` alternativo

se contaba con dicha implementación se tomó de otros *benchmarks* (Roops [5], por ejemplo).

Para la evaluación de la técnica basada en la incorporación de criterios de cobertura en la generación, se utilizaron dos criterios de cobertura, uno de caja blanca: criterio de cobertura de decisión y un criterio de cobertura de caja negra: cobertura de clases de equivalencia.

Con respecto a los criterios de cobertura utilizados para la evaluación de la técnica de reducción de *suites de test* basada en `repOk`, también se seleccionaron criterios de complejidad variada: un criterio simple como lo es cobertura de decisión, cobertura de decisión con conteo (definido en capítulos anteriores) como un criterio de complejidad media, y cobertura de camninos. Continuando con la técnica de reducćción basada en `repOk`, los casos de *test* seleccionados para cada *suite de tests* reducida son los primeros casos de *test* generados/encontrados de cada clase de equivalencia viable. Se podría aplicar algún otro mecanismo de selección de casos de *test*, por ejemplo, seleccionar una cantidad de casos de *test* para cada clase de equivalencia de manera aleatoria. Además, las *suites de tests* sobre las cual se aplican las reducciones fueron generadas usando `Korat` [11], Así como también para la evaluación de todas las técnicas, las *suites de test* exhaustivas acotadas contra la cuales se realizan las comparaciones, fueron generadas uitlizando esta herramienta. Debido a esto, estas *suites* excluyen estructuras isomorfas. Las estructuras isomorfas y el mecanismo de rotura de simetría de `Korat` se explican en detalle en el Capítulo 2

Para los casos en donde se realiza *testing* de mutación, la herramienta usada para generar mutantes de las rutinas seleccionadas, fue `muJava` [25]: los mutantes son aquellos obtenidos de la aplicación de 12 diferentes operadores de mutación a nivel de métodos.

Por último, las cotas ajustadas utilizadas en la evaluación de la técnica subyacente, fueron automáticamente generadas con herramientas para tal fin [14, 31], y por consiguiente se asumen correctas.

6.5. Análisis de resultados obtenidos

Esta sección se resumen y discuten los resultados experimentales exhibidos en este capítulo. En la primera parte de la experimentación, se evalúa la técnica de generación exhaustiva acotada de clases de equivalencia asociadas a un criterio dado, implementada como una variante de `Korat` (`Korat+`). Para ello, se compara este nuevo enfoque con la generación exhaustiva acotada tradicional. Los resultados obtenidos muestran que la eficiencia del enfoque depende en gran medida de la calidad de `repOk()` y `eqClass()`, y como éstos se relacionan entre sí. Por ejemplo, en los casos en los que `eqClass()` necesita visitar la estructura completa para lograr determinar la clase de equivalencia del caso de *test*, no hay ninguna poda extra a la realizada por `Korat`; esto se debe al hecho de que el cálculo del "siguiente candidato" de `Korat` ya estaría avanzando alguno de los campos observados por `eqClass()`, ya que éste observo todos los campos. El último de las casos de estudio para caja negra presentados, *listas simplemente encadenadas*, evidencia esta situación: determinar a que clase de equivalencia pertenece una lista dada requiere mirar toda la lista, por lo que, como se puede observar, `Korat+` examina igual cantidad de candidatos que `Korat`. La técnica provee mejores resultados cuando el criterio de *test* bajo consideración es tal que con sólo examinar una porción pequeña de la estructura se puede determinar a que clase de equivalencia, el caso de *test* pertenece. Esto es exactamente lo que sucede en los dos primeros casos de estudio de caja negra, en los cuales, la técnica muestra mejores ganancias. En el caso de *binomial heaps*, los predicados utilizados para determinar las clases de equivalencia fueron:

- el primer *heap* es vacío,
- el segundo *heap* es vacío,
- el primer *heap* tiene más elementos que el segundo,

- ambos *heaps* tienen la misma cantidad de elementos,
- el primer *heap* tiene grado más grande que el segundo,
- ambos *heaps* tienen el mismo grado, y
- ambos *heaps* contienen un árbol con el mismo grado.

Como se puede notar sólo es necesario examinar el tamaño de ambos *binomial heaps* para determinar la clase de equivalencia a la cual pertenece una estructura, incluso determinar el grado de un *binomial heap* depende de la cantidad de nodos en él (el grado de cada árbol binomial dentro de un *binomail heap* corresponde a un dígito en la representación binaria de su tamaño).

Otro factor importante en la eficiencia de esta técnica, implementada en `Korat+`, comparado con `Korat` es el tamaño del espacio de candidatos ́validos sobre el espacio de búsqueda. De manera más precisa, cuando `repOk()` falla de manera frecuente, es decir, cuando las restricciones para ser una estructura válida son fuertes, `Korat` explota su mecanismo de poda asociado. Es decir, cuando `repOk()` se satisface mayor cantidad de veces de las que éste falla, es cuando `Korat+` contribuye con más fuerza a la poda, ya que mientras que `Korat` debería avanzar al siguiente candidato sin poda, `Korat+` trataría de podar candidatos pertenecientes a la clase de equivalencia ya cubierta. Esto se puede observar, por ejemplo, en los casos de estudio de caja blanca que se presentaron. Se puede notar que, cuando para `Korat` la cantidad de casos de *test* válidos es grande en comparación con el número de candidatos explorados (`repOk()` se satisface frecuentemente) esta técnica tiende a contribuir de manera más sustancial a la poda.

Acerca de la elección de los casos de estudio, aunque éstos corresponden a piezas de código relativamente pequeñas, representan, situaciones naturales de *testing* en el contexto de datos estructuralmente complejos. Alguno de los casos de estudio elegidos son también empleados por `Korat`, de los cuales hemos utilizado el código provisto por la distribución de `Korat` en donde, las rutinas `repOK()` están escritas de manera tal que explotan el proceso de poda de esta herramienta.

Con respecto a la efectividad de las *suites* producidas, `Korat+` obtiene mejores resultados en comparación a clases de equivalencia óptimas (UPC), y a medida que las cotas crecen, `Korat+` consigue resultados similares a los de las *suites de tests* exhaustivas acotadas. Estos resultados confirman la intuición de que la técnica presentada se encuentra en el medio entre clases de equivalencia óptimas y *testing* exhaustivo acotado.

En la segunda parte de este capítulo, se evaluaron experimentalmente las técnicas para reducir el tiempo empleado en el *testing* exhaustivo acotado, basadas en el uso del invariante de representación:

- *Generación independiente de sub-estructuras disjuntas*
- *Reducción de suites de tests exhaustivas acotadas usando criterios sobre* `repOk()`

La primera de ellas pretende reducir el tiempo de generación de *tests*, mediante la generación independiente de sub-estructuras disjuntas de las entradas, y la segunda tiene como objetivo reducir el tiempo de *testing*, por medio de la reducción de las *suites de tests* generadas exhaustivamente filtrando casos redundantes, equivalentes a otros presentes en la *suite*, de acuerdo a algún criterio de caja blanca aplicado al `repOk()`. Ambas técnicas hacen uso del invariante de representación del código bajo prueba, usualmente implementado como una rutina `repOk()`.

La técnica *generación disjunta*, fue evaluada para varios casos de estudio que involucran algunas estructuras de datos alojadas en memoria dinámica: *NodeCachinglinkedList (getFirst)*, *binomialheaps (merge)*, *Árboles binarios de búsqueda balanceados y listas simplemente encadenadas (AddAll)*.

Los resultados son contundentes: en los tres casos evaluados se observa un importante incremento en la escalabilidad de la generación. La *generación disjunta* evita la exploración de gran cantidad de candidatos inválidos, por ejemplo para el caso de `merge` sobre *binomial heaps* se redujo la cantidad de candidatos visitados de 274.808.123 a 42.815, lo cual impactó fuertemente en el tiempo consumido, el cual se redujo en un orden de magnitud (693,116 vs 64,499).

Para el caso de *reducción de suites de tests basada en* `repOk()`, la evaluación se basó en estructuras de datos alojadas en memoria dinámica seleccionadas: *binomial heaps*, *árboles binarios de búsqueda*, *listAsSet*, *grafos dirigidos*, *listas doblemente encadenadas*, *árboles rojos y negros* y *árboles binarios de búsqueda balanceados*. Los experimentos fueron realizados utilizando 3 diferentes criterios de cobertura de caja blanca sobre `repOk()`: *cobertura de decisión*, *cobertura de caminos* y una variante de cobertura de decisión, llamada *cobertura de decisión con conteo*, todos de diferente complejidad.

Las *suites de tests* exhaustivas acotadas fueron reducidas a uno y dos ordenes de magnitud, es decir, al 10 % y 1 % del tamaño original de las *suites*.

En la mayoría de los casos de estudio, *cobertura de decisión con conteo* y *cobertura de caminos*, no muestran casi diferencia en el número de clases de equivalencia de casos de *test*, y por lo tanto, tampoco hay diferencia en los tamaños de las *suites*

reducidas correspondientes. Esto se debe al hecho de que, en algunos casos, especialmente para estructuras lineales, cuando `repOk()` retorna verdadero, el número de veces que cada decisión se evalúa en verdadero o en falso, generalmente, determina el *camino* que sigue la ejecución de `repOk()`. En otras estructuras, en particular árboles binarios de búsqueda, el número de veces que cada decisión se evalúa en verdadero o en falso generalmente determina varios caminos, correspondientes a alguna permutación en el orden en el cual las decisiones son evaluadas.

Además, se midió la efectividad de la técnica de acuerdo a *testing de mutación*, tomando algunas rutinas que manipulan las estructuras seleccionadas para el análisis:

- `merge`, `insert`, `delete` y `find` sobre binomial heap
- `insert`, `delete` y `search` sobre árboles binarios de búsqueda
- `isPalindromic` sobre listas doblemente encadenadas,
- `add`, `remove` y `contains` sobre árboles rojos y negros

Para estas rutinas, se generaron mutantes y se midió la efectividad de las diferentes *suites* exhaustivas acotadas, reducidas y "uno por clase de equivalencia" (UPC).

El número de mutantes obtenidos fue 113 mutantes para `merge`, 97 para `insert`, 170 para `delete` y 28 para `find` de binomial heaps, 24 mutantes para `delete`, 9 para `insert` y 4 para `search` de árboles binarios de búsqueda, 138 mutantes para `remove`, 115 para `add`, 32 para `contains` de árboles rojos y negros, y 21 mutantes para `isPalindromic` sobre listas doblemente encadenadas.

En el caso de árboles binarios de búsqueda, el filtrado basado en cobertura de decisión sobre `repOk()` tiene un número de mutantes muertos que decrece cuando las cotas se incrementan. Esto tiene que ver con el hecho de que todas las clases viables se cubren en la cota mas pequeña (es decir, ninguna clase nueva se cubre cuando se incrementa la cota), y el incremento de las cotas altera el orden de generación de los vectores candidatos; como resultado, cuando la cota se incrementa, la *suite de tests* reducida puede cambiar, seleccionando diferentes casos de *test* para alguna de las clases de equivalencia cubiertas.

El caso de estudio en el cual la técnica muestra mejor desempeño es en *binomial heaps* (`merge`), por ejemplo con sólo el 1 % del tamaño de la *suite* exhaustiva acotada, para cota 6, y cualquiera de los tres criterios de cobertura utilizados, se obtienen resultados idénticos que con la *suite* exhaustiva acotada. En el resto de los casos de estudio, si bien los resultados no son tan buenos como en la rutina `merge`, aún son

muy buenos si se tiene en cuenta el tamaño de las *suites de tests* reducidas: en la mayoría de los casos con el 10 % de las entradas se matan casi todos los mutantes, restando sólo unos pocos para lograr los resultados óptimos.

Con respecto a los casos de estudio seleccionados (en ambas técnicas) representan, situaciones típicas de *testing* en el contexto de la implementación de estructuras complejas alojadas en memoria dinámica. Se eligieron casos de estudio de complejidad variada, incluyendo estructuras de datos con restricciones simples, intermedias y complejas (por ejemplo listas simplemente encadenadas, árboles binarios de búsqueda y binomial heap, respectivamente). Debido a que esta técnica depende de la estructura del `repOk()`, se tomaron estas rutinas de la distribución de `Korat` en lugar de escribir implementaciones de `repOk()` propias. Además, para el caso de las reducciones de *suite de tests*, se seleccionaron criterios de cobertura de complejidad variada: un criterio de cobertura simple como lo es el criterio de *cobertura de decisión*, uno más complejo: *cobertura de caminos* y uno de complejidad intermedia :*cobertura de decisión con conteo*.

En la última sección, se evaluó experimentalmente la incorporación de *cotas ajustadas* en la generación exhaustiva acotada, en particular se incorporaron *cotas ajustadas* al proceso de generación de `Korat`, de lo cual surge una variante de esta herramienta denominada `KoratCotas`. La evaluación se basó en estructuras de datos alojadas en memoria dinámica seleccionadas: *binomial heaps*, *árboles binarios de búsqueda* y *árboles rojos y negros*. Los experimentos fueron realizados utilizando 2 diferentes implementaciones del invariante de representación (`repOk`). Para cada una de las estructuras seleccionadas, uno de los `repOk` fue tomado de la distribución de `Korat`, el otro fue adquirido de fuentes tales como TACO [14] y el *benchmark de roops* [5]. Los resultados experimentales obtenidos cuando se compara `KoratCotas` con `Korat` utilizando el primer `repOk`, muestran que `KoratCotas` no obtiene ventajas relevantes sobre `Korat`. Más precisamente, `KoratCotas` evita generar algunos candidatos, pero no los suficientes para obtener ganancias significativas en tiempo. Esto se debe a que el `repOk` está escrito de tal manera que `Korat` explota al máximo su mecanismo de poda, y que `KoratCotas` en la búsqueda de candidatos válidos evita generar candidatos inválidos, que no son compatibles con las *cotas ajustadas*, cuya generación aportaría a podar la búsqueda. Para clarificar esto último, recordemos algunos detalles de implementación de `Korat`. Esta herramienta explota su mecanismo de poda cuando `repOk` falla, es decir, cuando éste retorna falso, `Korat`, y por ende también `KoratCotas`, avanza las partes de la estructura visitadas por `repOk()`, ya que si éstas no cambian, `repOk` fallará nuevamente. `KoratCotas` en estos casos, evita generar candidatos (inválidos y no compatibles con las cotas) que

debido a este mecanismo de poda, y a que `repOk` esta escrito de manera tal que falla lo antes posible (es decir visitando una porción pequeña de la estructura) obligarían a podar el espacio de candidatos explorados.

Por otro lado, cuando el `repOK` alternativo es utilizado, `KoratCotas` ayuda a reducir significativamente el espacio de candidatos explorados, por ejemplo, para el caso de árboles rojos y negros, de tamaño 8, se reduce el espacio de búsqueda en un 75 %, lo cual produce un importante aumento en la escalabildad de la técnica, que permite pasar de tamaño 8 a tamaño 10, antes de alcanzar el tiempo límite (3 horas).

Con respecto a la elección de casos de estudio, se eligieron estructuras con restricciones de complejiad variada. En todos los casos se utilizaron estructuras de datos que acompañan la distribución de `Korat`, las cuales estan equipadas con su correspondiente `repOk`. Además, en el caso de los `repOk` alternativos utilizados, ninguno de ellos fue implementado *ad-hoc*. Como se mencionó mas arriba, Éstos fueron tomados del *benchmark de roops* y de las casos de estudio provistos por TACO.

Capítulo 7

Conclusiones y Trabajos Futuros

La generación exhaustiva acotada de casos de *tests* es una técnica de *testing* muy efectiva en varios contextos, particularmente cuando se está en presencia de entradas estructuralmente complejas. Sin embargo, en muchos casos, tanto el tiempo de generación, como el tiempo de ejecución de los *tests*, crece exponencialmente con respecto a las cotas (*scopes*) empleadas durante la generación exhaustiva acotada. Este hecho, combinado con la necesidad de los desarrolladores de usar cotas relativamente grandes para cubrir algunos casos de interés o lograr cierto nivel de cobertura (o en general incrementar las posibilidades de encontrar errores) hacen, en muchos casos, de ambas -la generación y el uso de las *suites* generadas- algo impracticable. En esta tesis se investigó el uso de criterios de cobertura y de la especificación de las entradas válidas, para de revertir esta situación. La idea principal de este trabajo se basa en la búsqueda de nuevas técnicas que permitan reducir el tiempo de generación de *suites de tests* exhaustivas acotadas, así como también el tiempo de ejecución de las mismas. Con este propósito, se estudiaron diferentes mecanismos para podar el espacio de búsqueda de entradas y para reducir adecuadamente *suites* exhaustivas acotadas. Las principales contribuciones de este trabajo se presentan en los Capítulos 3, 4 y 5.

En el Capítulo 3 se presentó una técnica para mejorar la generación exhaustiva acotada basada en filtrado, incorporando criterios de cobertura, y empleando estos criterios para podar la búsqueda de entradas de *test* válidas. La técnica se enfoca en entradas estructuralmente complejas, y esencialmente consiste en incorporar en el proceso usual de poda, presente en las técnicas de generación de *tests*, un mecanismo de poda extra, que evite visitar partes del espacio de búsqueda cuando se tiene la certeza que los candidatos en esa parte del espacio corresponden a clases de

entradas ya cubiertas. Se implementó esta técnica como una variante de `Korat`, una herramienta que automáticamente genera casos de *test* mediante un mecanismo basado en "filtrado"[11], y se desarrollaron algunos casos de estudio, en el contexto de estructuras de datos alojadas en memoria dinámica, cuyos resultados experimentales asociados proveen suficiente evidencia de los beneficios de la misma. Estos resultados confirman la intuición de que este enfoque se encuentra en el medio de clases de equivalencia óptimas y *testing* exhaustivo acotado, es decir, las *suites de tests* obtenidas son significativamente más pequeñas que las *suites* exhaustivas acotadas con una efectividad comparable con estas últimas. Además de la experimentación se puede inferir que la técnica provee mejores resultados cuando el criterio de *testing* bajo consideración es tal que con sólo examinar una porción pequeña de la estructura se puede determinar a que clase de equivalencia, el caso de *test* pertenece. Por otro lado, cuando las restricciones para ser una estructura válida son fuertes, es decir `repOk` se satisface menor cantidad de veces de las que éste falla (lo cual, se refleja en un espacio de estados explorados grande en comparación con la cantidad de entradas válidas), `Korat` explota su mecanismo de poda asociado. Por el contrario, cuando las restricciones son débiles, el enfoque presentado contribuye con más fuerza a la poda.

En el Capítulo 4 se presentaron dos enfoques para aprovechar la especificación de las entradas válidas de la rutina bajo análisis que permiten reducir el tiempo empleado en el *testing* exhaustivo acotado.

El invariante de representación, se usa de dos formas diferentes:

- para obtener invariantes de representación separadas para cada sub-estructura disjunta de la entrada, e independizar la generación de estas partes disjuntas.

- para definir un criterio de caja negra para el programa bajo prueba, basado en la definición de relaciones de equivalencias de las entradas, definido en términos de un criterio de caja blanca sobre la implementación imperativa del invariante de representación (`repOk()`).

Estas técnicas basadas en el invariante de representación producen un impacto significativo en el *testing* exhaustivo acotado. Para la primera técnica, ésto se debe a que, cuando las entradas están compuestas de sub-estructuras disjuntas, éstas pueden ser generadas independientemente, evitando considerar estructuras inválidas, compuestas de sub-estructuras válidas en combinación con sub-estructuras inválidas. La evaluación experimental realizada muestra que, incluso cuando la generación independiente se realiza de forma secuencial, se reduce el espacio de búsqueda hasta

en un 99 %, lo cual produce un impacto sustancial en el tiempo de generación. Por otro lado, la técnica para filtrar casos de *test* presentada, se basa en la observación de que si dos entradas ejercitan el código del invariante de representación "de la misma manera" ellas podrían ser consideradas equivalentes, motivando la definición de un criterio de cobertura de caja negra sobre el programa bajo análisis. Nuevamente, la evaluación experimental muestra clara evidencia de que este criterio es de ayuda en el contexto del *testing* exhaustivo acotado, éste permite reducir las *suites* mientras se mantiene su efectividad; sin dejar de mencionar la ventaja que provee el hecho de mantener las clases de equivalencia independientes del código bajo análisis: la *suite de test* mantiene la misma propiedad que la *suite* exhaustiva acotada, ésta es adecuada para probar cualquier rutina que respete la misma especificación de entradas válidas.

Aunque estas técnicas, presentadas en el capítulo 4, requieren un invariante de representación imperativo, pueden ser relevantes y adaptables a contextos donde se cuenta con invariantes de representación declarativos. Los lenguajes declarativos cada vez son más populares en el ámbito de la programación por contratos (por ejemplo, invariantes en `Eiffel` o vía lenguajes declarativos tales como `JML` y `Code Contracts`), donde estos contratos declarativos son además ejecutables; el código correspondiente a su evaluación en tiempo de ejecución debería corresponder a lo que en este trabajo se refiere como `repOk()`.

En el Capítulo 5, se propuso incoporporar *cotas ajustadas* [14] al proceso de generación exhaustiva acotada, y de esta manera evitar la generación de estructuras (inválidas) que no son compatibles con las *cotas ajustadas*. Las *cotas ajustadas* restringen las entradas excluyendo, del conjunto de valores que cada campo puede tomar, casos que son inviables debido a que producirían estructuras simétricas o mal formadas. Éstas están siendo usadas con éxito en el ámbito de análisis de programas [14, 29, 6], lo cual motivó el estudio de las mismas en el contexto de la generación exhaustiva acotada. Particularmente, se incorporaron las *cotas ajustadas* (previamente calculadas a partir de la especificación de las entradas válidas) en el mecanismo de generación de `Korat`, y se realizó una evaluación experimental considerando diferentes `repOk`. En todos los casos evaluados, no es posible superar los resultados obtenidos por `Korat` (al menos de manera significativa) cuando se utiliza el `repOk` que provee la distribución de esta herramienta, el cual, esta escrito de manera tal que `Korat` explota su mecanismo de poda al máximo. Por el contrario, cuando se utiliza un `repOk` alternativo (implementado de manera *ad-hoc* para cada caso, pero intentando que los mismos sean implementaciones naturales de los invariantes), las cotas mejoran de manera significativa el tiempo de generación (en algunos casos

éste se reduce hasta 2 ordenes de magnitud, por ejemplo, para el caso de árboles binarios de búsqueda se pasa de una espera mayor a 3 horas a 107,586 seg). Esto se debe a que, tanto `Korat` como muchas de las herramientas de generación, tienen su eficiencia ligada a cómo la especificación está sintácticamente expresada, lo cual lleva a escribir invariantes de representación de una manera poco natural; `Korat` es un ejemplo de esto. La evidencia muestra que las *cotas ajustadas* pueden contribuir en mejorar la generación exhaustiva acotada en aquellos casos en donde se cuenta con un invariante de representación que no fue pensado para ser utilizado por un enfoque particular.

En general, los resultados de la evaluación experimental obtenidos, ponen en evidencia la utilidad que provee el uso de criterios de cobertura y sobre todo de la información que provee la especificación de las entradas de la rutina bajo análisis, para mejorar el *testing* exhaustivo acotado. Esto último, en el caso de las técnicas de generación basadas en “filtrado” es un requisito, por lo que utilizar esta información de alguna de las formas propuestas no demanda un esfuerzo extra de parte del usuario.

Como trabajos futuros, hay varias líneas en relación a algunos de los enfoques presentados en esta tesis. En particular, la efectividad de la paralelización de la generación disjunta no ha sido evaluada, lo cual se espera que mejore la generación, comparado con el enfoque secuencial presentado. Este enfoque paralelo debería ser comparado con otros enfoques de generación exhaustiva acotada paralela, tal como el presentado en [27]. Algunos problemas técnicos discutidos en este trabajo son fuentes interesantes de trabajos futuros. Uno de ellos tiene que ver con el análisis de diferentes fuentes de información que ayuden a determinar cuándo se está en presencia de sub-estructuras disjuntas. El análisis de fuentes tales como el método “finitization”, otra forma de que el usuario provea dicha información, o incluso mediante el uso de la información que brinda el invariante de representación; es un problema importante que se debe analizar, en particular si es deseable automatizar completamente este enfoque. El otro problema, también en relación al mismo enfoque, es el procesamiento de `repOk`, para obtener versiones locales del invariante de representación que caractericen partes disjuntas de una estructura. Con respecto a esto, en este trabajo la experimentación fue realizada mediante un mecanismo de *slicing* simple basado en el grafo de *definición y uso* sobre `repOk`, pero podría ser usado algún otro mecanismo más sofisticado para ello. Por otro lado, la incorporación de *cotas ajustadas* en el proceso de generación exhaustiva acotada de entradas, también deja algunas líneas abiertas de investigación, en particular, en lo que respecta al cómputo de cotas ajustadas desde invariantes imperativos, dado que la necesi-

dad de expresar el invariante de representación de manera imperativa y declarativa (usado para el cómputo), es decir vía algún lenguaje de especificación de contratos, impone una limitación no menor a este enfoque y hasta el momento no se conoce ningún mecanismo eficiente de cálculo de cotas desde invariantes imperativos. Por último, en esta tesis también se presenta un enfoque de generación basado en el uso de invariantes híbridos. Los invariantes son descriptos imperativamente, declarativamente o como combinación de ambas formas, pero no se cuenta, actualmente, con un lenguaje que permita escribir dichas especificaciones de manera amigable, de donde surge la necesidad de definir/implementar un lenguaje para tal fin.

Bibliografía

[1] *Code contracts, http://research.microsoft.com/en-us/projects/contracts/.* 4

[2] *Judy home page, http://mutationtest.com/.* 2.5.3

[3] *Jumble home page, http://jumble.sourceforge.net/.* 2.5.3

[4] *Mujava home page, http://www.cs.gmu.edu/ offutt/mujava/.* 2.5.3

[5] *Roops : http://code.google.com/p/roops/.* 6.3, 6.4, 6.5

[6] Pablo Abad, Nazareno Aguirre, Valeria S. Bengolea, Daniel Ciolek, Marcelo F. Frias, Juan Galeotti, Tom Maibaum, Mariano M. Moscato, Nicolás Rosner, and Ignacio Vissani, *Improving test generation under rich contracts by tight bounds and incremental sat solving.*, ICST, IEEE, 2013, pp. 21–30. 5.1, 5.2, 6.3, 7

[7] Nazareno Aguirre, Valeria S. Bengolea, Marcelo F. Frias, and Juan P. Galeotti, *Incorporating coverage criteria in bounded exhaustive black box test generation of structural inputs*, TAP, 2011, pp. 15–32. 1

[8] Boris Beizer, *Black-box testing: techniques for functional testing of software and systems*, John Wiley & Sons, Inc., New York, NY, USA, 1995. 2.5.2

[9] Valeria S. Bengolea, Nazareno Aguirre, Darko Marinov, and Marcelo F. Frias, *Using coverage criteria on repok to reduce bounded-exhaustive test suites*, TAP, 2012, pp. 19–34. 1

[10] ______, *Repok-based reduction of bounded exhaustive testing*, Softw. Test., Verif. Reliab. **24** (2014), no. 8, 629–655. 1

[11] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov, *Korat: automated testing based on java predicates*, ISSTA, 2002, pp. 123–133. 1, 2.2.1, 2.4, 2.4, 3.1, 4.3.1, 6.2.1, 6.4, 7

[12] Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov, *Automated testing of refactoring engines*, Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (New York, NY, USA), ESEC-FSE '07, ACM, 2007, pp. 185–194. (document), 1

[13] Mark Dowson, *The ariane 5 software failure*, SIGSOFT Softw. Eng. Notes **22** (1997), no. 2, 84–. 1

[14] Juan P. Galeotti, Nicolás Rosner, Carlos López Pombo, and Marcelo F. Frias, *Analysis of invariants for efficient bounded verification*, ISSTA, 2010, pp. 25–36. (document), 1, 5, 5.1, 5.1, 5.2, 5.3, 5.3.3, 6.3, 6.3, 6.4, 6.5, 7

[15] Jaco Geldenhuys, Nazareno Aguirre, Marcelo F. Frias, and Willem Visser, *Bounded lazy initialization*, NASA Formal Methods, 2013, pp. 229–243. 5.1, 5.2

[16] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli, *Fundamentals of software engineering*, (2002). 1, 2.5

[17] Milos Gligoric, Tihomir Gvero, Vilas Jagannath, Sarfraz Khurshid, Viktor Kuncak, and Darko Marinov, *Test generation through programming in udita*, ICSE (1), 2010, pp. 225–234. 1, 2.2.1, 3.1

[18] Daniel Jackson, *Software abstractions - logic, language, and analysis.*, MIT Press, 2006. 1, 5.1

[19] Vilas Jagannath, Yun Young Lee, Brett Daniel, and Darko Marinov, *Reducing the costs of bounded-exhaustive testing*, In FASE, 2009. (document), 1

[20] P. Jalote, *An integrated approach to software engineering*, Texts in Computer Science, Springer, 2005. 6.2.1

[21] Sarfraz Khurshid and Darko Marinov, *Testera: Specification-based testing of java programs using sat*, Autom. Softw. Eng. **11** (2004), no. 4, 403–434. 1, 3.1

[22] N. G. Leveson and C. S. Turner, *An investigation of the therac-25 accidents*, Computer **26** (1993), no. 7, 18–41. 1

[23] Barbara Liskov and John Guttag, *Program development in java: Abstraction, specification, and object-oriented design*, 1st ed., Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000. 2.3, 2.3, 2.4, 4

[24] Lisa Ling Liu, Bertrand Meyer, and Bernd Schoeller, *Using contracts and boolean queries to improve the quality of automatic test generation*, Proceedings of the 1st international conference on Tests and proofs (Berlin, Heidelberg), TAP'07, Springer-Verlag, 2007, pp. 114–130. 2.5.2

[25] Yu-Seung Ma, Jeff Offutt, and Yong Rae Kwon, *Mujava: an automated class mutation system: Research articles*, Softw. Test. Verif. Reliab. **15** (2005), no. 2, 97–133. 2.5.3, 6.4

[26] Aleksandar Milicevic, Sasa Misailovic, Darko Marinov, and Sarfraz Khurshid, *Korat: A tool for generating structurally complex test inputs*, ICSE, 2007, pp. 771–774. 1, 2.4

[27] Sasa Misailovic, Aleksandar Milicevic, Nemanja Petrovic, Sarfraz Khurshid, and Darko Marinov, *Parallel test generation and execution with korat*, Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering (New York, NY, USA), ESEC-FSE '07, ACM, 2007, pp. 135–144. (document), 1, 7

[28] Glenford J. Myers and Corey Sandler, *The art of software testing*, John Wiley & Sons, 2004. 1, 2.5, 2.5.1, 2.5.2

[29] Rosner N., J. Geldenhuys, N. Aguirre, W. Visser, and M.F. Frias, *Bliss: Improved symbolic execution by bounded lazy initialization with sat support*, Software Engineering, IEEE Transactions on **PP** (2015), no. 99. 5.3, 5.3.1, 5.3.2, 7

[30] Erik Poll, Patrice Chalin, David Cok, Joe Kiniry, and Gary T. Leavens, *Beyond assertions: Advanced specification and verification with jml and esc/java2*, In Formal Methods for Components and Objects (FMCO) 2005, Revised Lectures, volume 4111 of LNCS, Springer, 2006, pp. 342–363. 2.3, 4, 5.3.3

[31] Pablo Ponzio, *Cómputo secuencial eficiente de cotas ajustadas, y su impacto en la performance de los análisis de programas basados en sat*, Ph.D. thesis, Universidad de Buenos Aires, 2013. 5.3.3, 6.4

[32] Nicolás Rosner, Valeria S. Bengolea, Pablo Ponzio, Shadi Abdul Khalek, Nazareno Aguirre, Marcelo F. Frias, and Sarfraz Khurshid, *Bounded exhaustive test input generation from hybrid invariants*, Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages

& Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014, 2014, pp. 655–674. 1, 5, 5.4

[33] Gregg Rothermel, Sebastian Elbaum, Alexey Malishevsky, Praveen Kallakuri, and Brian Davia, *The impact of test suite granularity on the cost-effectiveness of regression testing*, 2001.

[34] Koushik Sen, Darko Marinov, and Gul Agha, *Cute: a concolic unit testing engine for c*, ESEC/SIGSOFT FSE, 2005, pp. 263–272.

[35] Junaid Haroon Siddiqui and Sarfraz Khurshid, *An empirical study of structural constraint solving techniques*, ICFEM, 2009, pp. 88–106.

[36] Kevin Sullivan, Jinlin Yang, David Coppit, Sarfraz Khurshid, and Daniel Jackson, *Software assurance by bounded exhaustive testing*, SIGSOFT Softw. Eng. Notes **29** (2004), no. 4, 133–142. (document)

[37] Emina Torlak and Daniel Jackson, *Kodkod: A relational model finder*, In Tools and Algorithms for Construction and Analysis of Systems (TACAS, Wiley, 2007, pp. 632–647. 5.1

[38] Willem Visser, Corina S. Pasareanu, and Sarfraz Khurshid, *Test input generation with java pathfinder*, ISSTA, 2004, pp. 97–107.

[39] Tao Xie, Darko Marinov, and David Notkin, *Rostra: A framework for detecting redundant object-oriented unit tests*, ASE, 2004, pp. 196–205. 1

[40] Yanbing Yu, James A. Jones, and Mary Jean Harrold, *An empirical study of the effects of test-suite reduction on fault localization*, Proceedings of the 30th international conference on Software engineering (New York, NY, USA), ICSE '08, ACM, 2008, pp. 201–210.

[41] Hong Zhu, Patrick A. V. Hall, and John H. R. May, *Software unit test coverage and adequacy*, ACM Comput. Surv. **29** (1997), no. 4, 366–427. 2.5.1, 2.5.1, 2.5.3, 6.2.2