Testing Exhaustivo Acotado

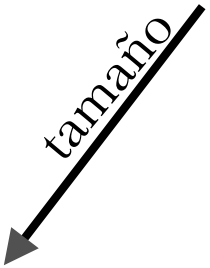
Testing Exhaustivo Acotado**:** Generación Exhaustiva Acotada| Generación de estructuras complejas| Problemas| Invariante de representación| KORAT

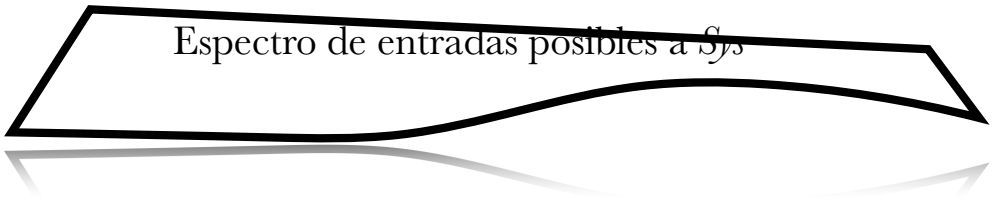
Testing

el triangulito , ejemplo de mínimo de una secuencia, triangulito finito que es el testing (los puntitos)

Idealmente, nos gustaría tener la garantía de que el sistema funciona correctamente para TODAS las posibles entradas.

La elección del conjunto de casos de test juega un rol fundamental en esta tarea.

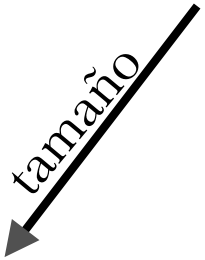




que decir, actividad compleja, muy cara

Selección de entradas

1) descripcion de las entradas, se reconoce una porción de aceptable de interés. aquellos que cumplen con esta descripción



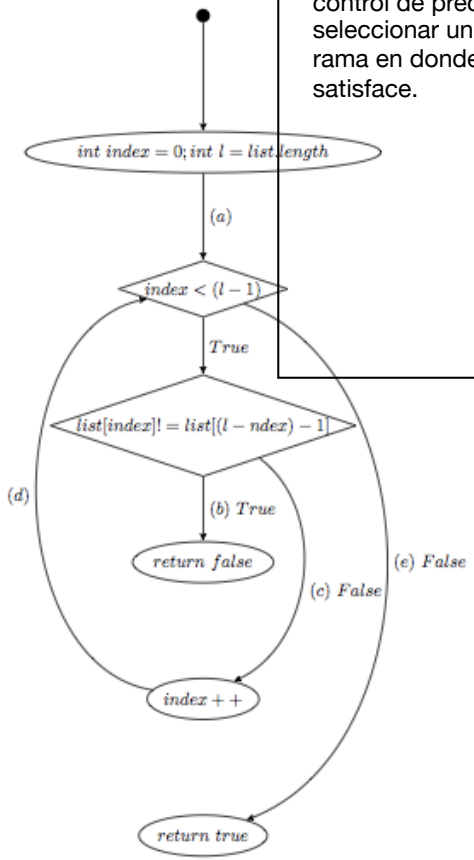
satisfacen *pre*

Selección de entradas

Por ejemplo, seleccionar entradas de tal manera de cubrir todos los arcos del grafo de flujo de control.
Suponiendo que mi programa hace control de precondicion, debería seleccionar una entrada que cubra la rama en donde la precondicion no se satisface.

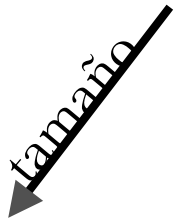
tamaño

satisfacen *pre*



Se utiliza el ***grafo de flujo de control*** para decidir q entradas elegir.

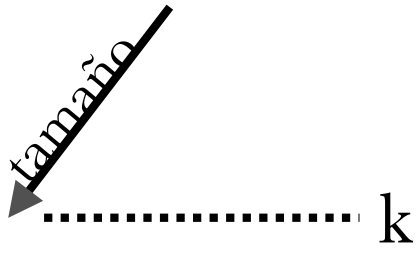
Selección de entradas



satisfacen *pre*

familia de criterios, no miro codigo sino que me concertó en la especificación
en este ejemplo una precondicion
ejemplo: todas las posibles formas de hacer verdadera la precondición.(y falsa?).
Un caso en ningún disjunto es verdadero.
alfa1 verdadero y los demás falsee, etc..

Selección de entradas



satisfacen *pre*

* No se corre el riesgo de ignorar casos bordes
* *Hipótesis de la cota pequeña*: "Si un programa tiene errores, la mayoría de estos errores pueden ser detectados usando entradas pequeñas"

Testing Exhaustivo Acotado

Testing Exaustivo Acotado

Un enfoque para hacer testing en general, y para la generación de entradas.

Generar TODAS las posibles entradas dentro de alguna cota dada.

Muy efectivo cuando se trata de rutinas parametrizadas con estructuras complejas alojadas en memoria dinámica.

Generación Exhaustiva Acotada

mostrar la diferencia con los generadores de teorías

Es viable solo si es automático

Fácil de automatizar para rutinas parametrizadas con tipos de datos básicos.

Muy difícil para rutinas parametrizadas con tipos de datos estructuralmente complejos (e.g., AVLs, árboles de búsqueda, árboles rojos y negros, grafos, ...)

Generación Exhaustiva Acotada

en lugar de usarla API pública, ya que para eso necesitaríamos conocer exactaen contarle que esta sería otra manera- no hay tool - todavía.

Una manera de generar todas las estructuras posibles en cierto rango es construir posibles candidatos y chequear si estos candidatos se corresponde con una estructura válida.

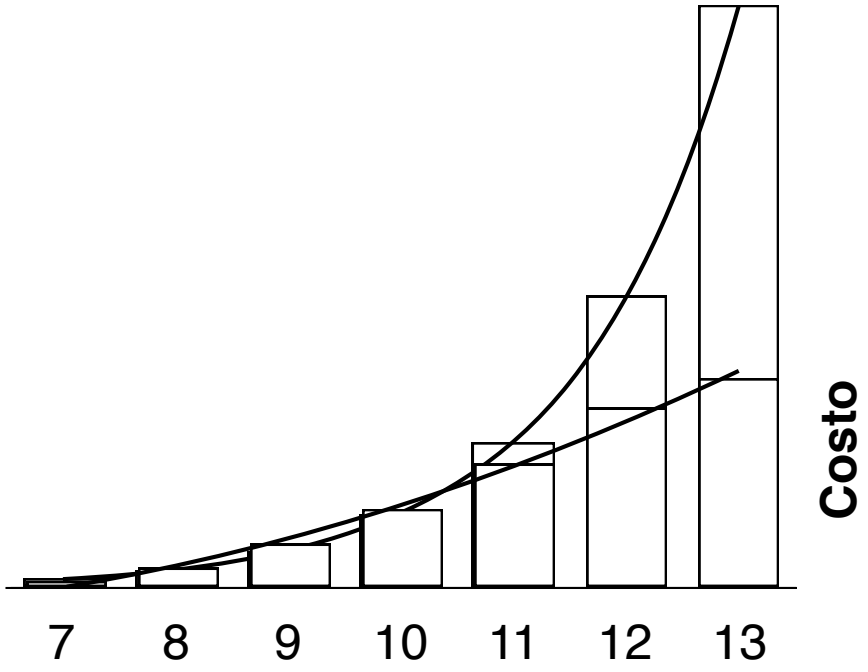
Se necesita como entrada el rangos para los dominios, una descripción de las entradas válidas (precondición, Invariante de representación) y el código bajo prueba.

TESTING EXHAUSTIVO ACOTADO

Problemas

- ***Costo de Generación***: espacio de búsqueda es extremadamente amplio.

- ***Costo de Testing***: las *suites de test* producidas son muy grandes



Enumerar estructuras. Cuán difícil resulta?

Todo nodo es o bien rojo o bien negro.
La raíz es negra.
Todas las hojas (NULL) son negras.
Todo nodo rojo debe tener dos nodos hijos negros.
Cada camino desde un nodo dado a sus hojas descendientes contiene el mismo número de nodos negros.

-260-

este es uno de los "pocos" que satisface el invariante de RN

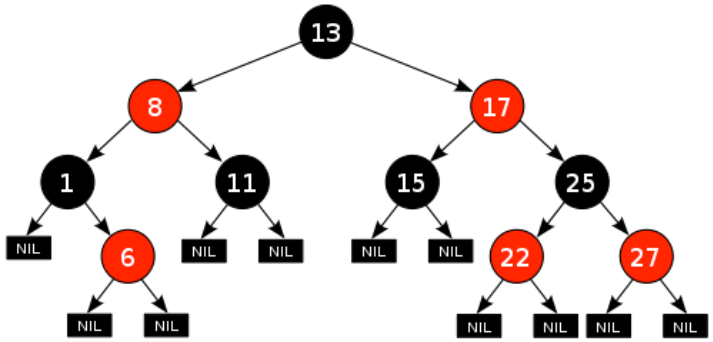
Consideremos árboles rojos y negros (n nodos, m claves)

Número de árboles binarios:

Número de asignaciones de claves a nodos:

Número de asignaciones de colores a nodos:

10 n, 10 m -> 171.991.040.000.000.000



Datos Estructuralmente Complejos en Aplicaciones

Las estructuras complejas no forman parte solamente de librerías de estructuras de datos. Se encuentran en:

- Aplicaciones que manipulan archivos XML (o similares), los cuales deben respetar ciertas reglas de buena formación
- Aplicaciones para el análisis de sitios web, donde los sitios pueden interpretarse como grafos con ciertas características (acíclicos?, fuertemente conexos?)

Problemas en la Explosión de Estructuras Posibles

Algunos problemas en el manejo de la explosión de estructuras posibles son los siguientes:

- Iteración sobre elementos irrelevantes del espacio de estados de la estructura (e.g., sobre elementos inalcanzables del heap)
- Estructuras simétricas (i.e., instancias redundantes)

Un Ejemplo: Iteración sobre Elementos no Alcanzables

Supongamos que queremos testear una rutina que manipula árboles binarios de búsqueda.

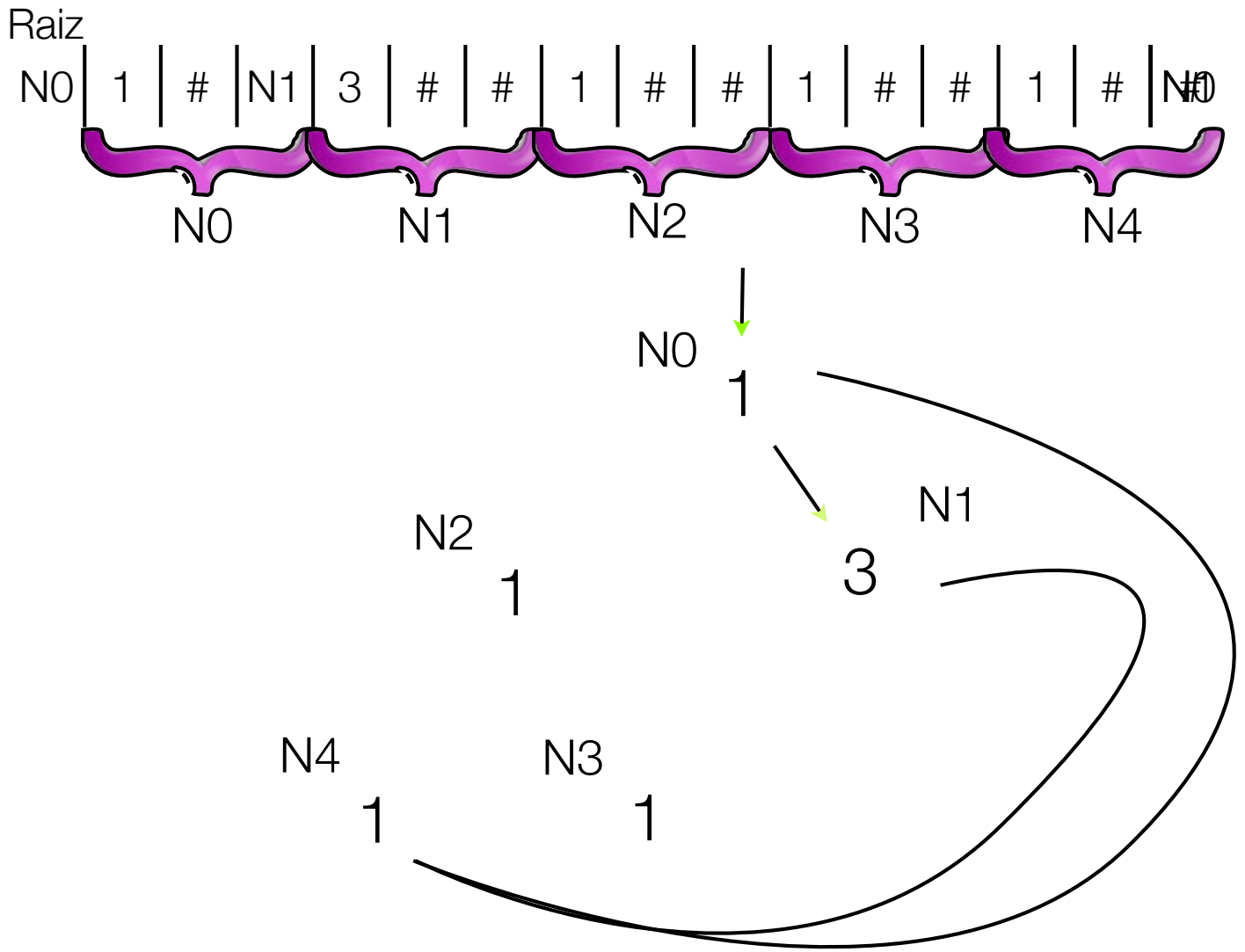
Nos interesa generar TODOS los árboles binarios de búsqueda de (hasta) 5 nodos (conteniendo valores del 1 a 5).

Cada estructura puede representarse por:

1 valor r (nodo raíz)

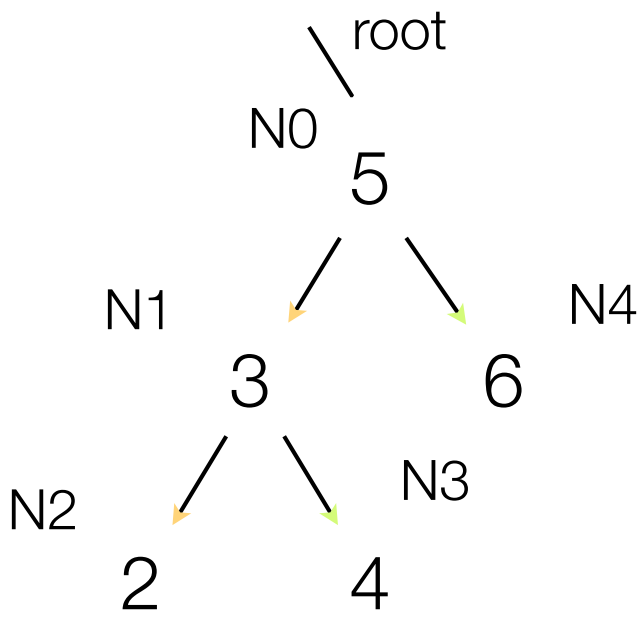
5 tuplas (v_i, izq_i, der_i), que representan los nodos

Un Ejemplo: Iteración sobre Elementos no Alcanzables



Estructuras Simétricas

Consideremos el siguiente árbol:



Si nos abstraemos de las direcciones específicas de los nodos, un total de 5! estructuras diferentes representan el mismo árbol.

Descripción de las Entradas Válidas

la especifi- cacio´n de la rutina bajo an´alisis, en particular la descripci´on de entradas va´lidas, la cual es una precondici´on implicita que las entradas deben satisface. Esta descripcio´n de las entradas validas, la cual en el contexto de la programacio´n orientada a

objetos se corresponde con el invariante de representacio´n de una clase

Descripción de las Entradas válidas

Operativa

Declarativa

Ejemplo de Invariante de Representación Declarativo

```
public class BinaryTree {

    private Node root;
    private int size;
    ...
}
```

```
@Invariant
(this.root==null => this.size = 0)&&

all n : Node | n in this.root.*(left @+ right ) =>
                (
                   n !in n.^(left @+ right)
                ) ;
```

```
        public class Node {

                Node left;
                Node right;

        }


        }
```

Descripción de las Entradas Válidas

la especifi- cación de la rutina bajo an´alisis, en particular la descripci´on de entradas va´lidas, la cual es una precondici´on implícita que las entradas deben satisfacer. Esta descrip- ci´on de las entradas válidas, la cual en el contexto de la programaci´on orientada a

objetos se corresponde con el invariante de representacio´n de una clase

Descripción de las Entradas válidas

Operativa

Declarativa

Ejemplo de Invariante de Representación Operacional

```
public class BinaryTree {

    private Node root;
    private int size;
    ...
}
```

```
public boolean repOK() {
        if (root == null)
            return size == 0;
        // checks that tree has no cycle
        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);
    }
```

```
        public class Node {

               Node left;
               Node right;

        }


        }
```

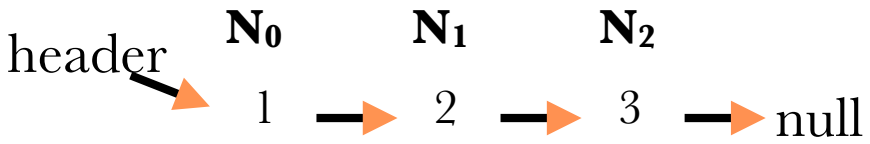
Estado del arte

Hay herramientas que lo hacen muy eficientemente aunque muy sensibles a la forma en que repOk esta escrito.

(importa el orden de las restricción por ejemplo) y si las restricciones son chequedas en una solo pasada

Si la *Pre* es **operacional** (repOk), hay herramientas que descartan:

* entradas inválidas. (evitan mirarlas a todas)
* entradas redundantes (simétricas)



Korat (Búsqueda)

la poda para que es : no mirar todo el espacio de búsqueda
evitar estructuras simétricas-isomorfas

Realiza generación exhaustiva acotada (bounded exhaustive) de casos de test para código Java

Requiere cotas para dominios

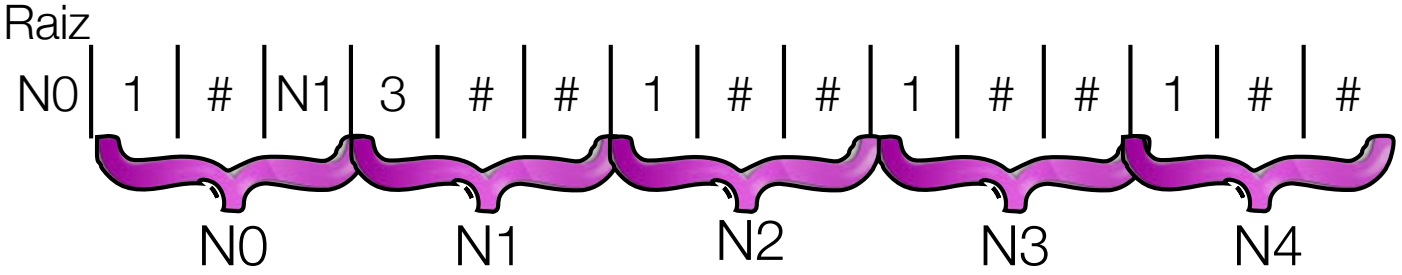
Requiere la especificación operativa del invariante de representación de la estructura (rutina repOK)

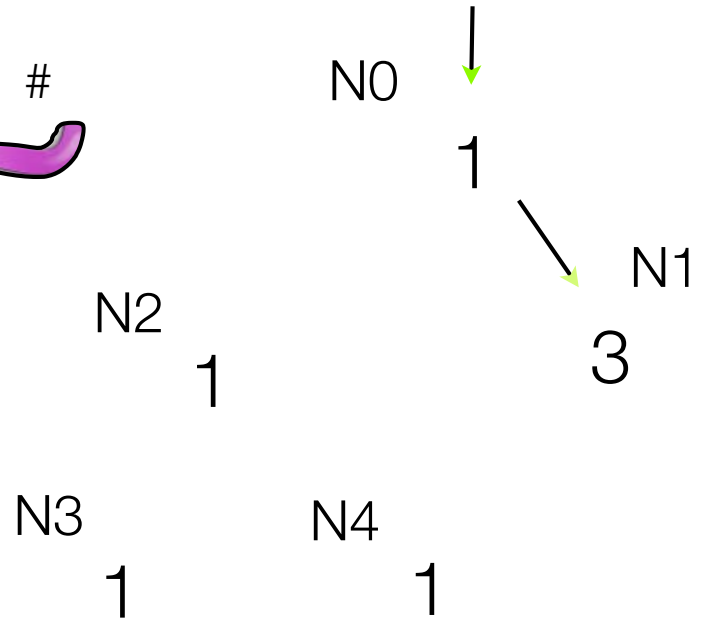
Basado en búsqueda *depth first search* (backtracking) con potentes mecanismos de poda

Korat: Estrategia de Búsqueda

Korat realiza una búsqueda de instancias válidas sobre el espacio de instancias posibles

Representa las instancias con vectores candidatos

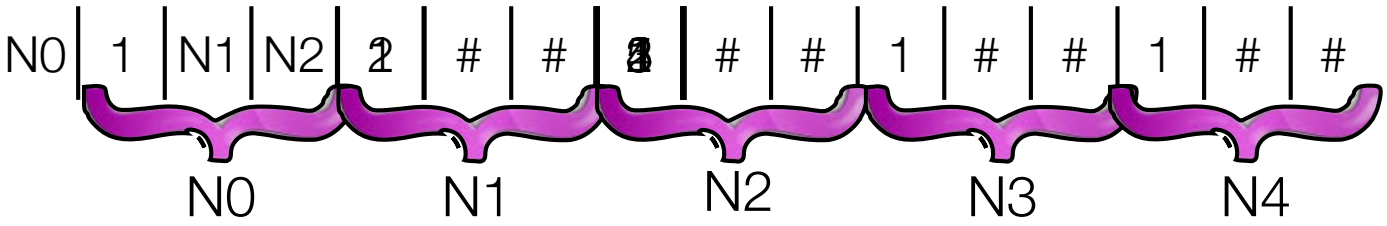




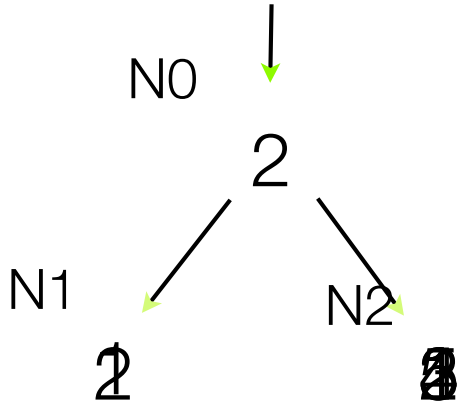
El backtracking se realiza de acuerdo al orden de visita de los elementos de la estructura, según repOK()

Backtracking en Vectores Candidatos

Consideremos el siguiente ejemplo (ABB):



repOK(): isBinaryTree(); isSorted()



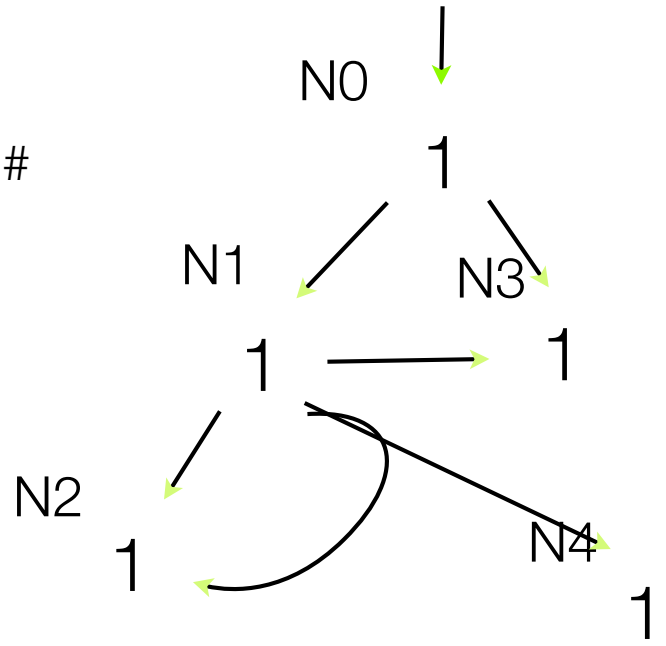
El backtracking evita iterar sobre porciones no alcanzables de la estructura

Korat: Estrategia de Poda

Korat realiza podas del espacio de estados en casos en los cuales repOK() falla

Evita en muchos casos visitar espacios grandes de vectores candidatos inválidos

N0	1	#	N1	1	N2	N3	1	#	#	1	#	#	1	#	#



Ejemplo de Invariante de Representación Operacional

```
public class BinaryTree {

    private Node root;
    private int size;
    ...
}
```

```
public boolean repOK() {
        if (root == null)
            return size == 0;
        // checks that tree has no cycle
        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);
    }
```

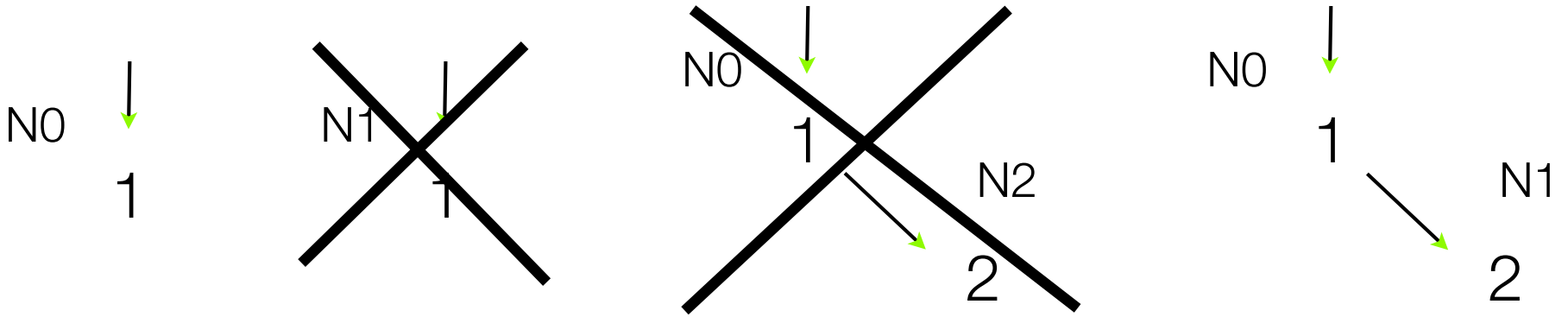
```
        public class Node {

               Node left;
               Node right;

        }
```

```
                                    if !isSorted()
                                          return false;
```

Korat
Rotura de Simetrías

Korat realiza rotura de simetrías mediante el uso de una regla muy simple:

> "Durante la visita, se puede observar a lo sumo un objeto 'no tocado' previamente"
>
> El índice de un nodo no puede ser mayor a k+1, con k el mayor índice de los objetos del mismo tipo ya visitados



Korat
Rango para los dominios

```
public static IFinitization finStrictlySortedSinglyLinkedList(int minSize, int maxSize,
            int numEntries, int numElems) {
        IFinitization f = FinitizationFactory.create(StrictlySortedSinglyLinkedList.class);

        IObjSet entries = f.createObjSet(practico10.Node.class, true);
        entries.addClassDomain(f.createClassDomain(practico10.Node.class, numEntries));

        IIntSet sizes = f.createIntSet(minSize, maxSize);

        IObjSet elems = f.createObjSet(Integer.class);
        IClassDomain elemsClassDomain = f.createClassDomain(Integer.class);
        elemsClassDomain.includeInIsomorphismCheck(false);
        for (int i = 1; i <= numElems; i++)
            elemsClassDomain.addObject(new Integer(i));
        elems.addClassDomain(elemsClassDomain);
        elems.setNullAllowed(true);

        f.set("header", entries);
        f.set("size", sizes);
        f.set(practico10.Node.class, "element", elems);
        f.set(practico10.Node.class, "next", entries);
        return f;

    }
```

Para analizar…

En muchos casos el espacio de búsqueda es extremadamente amplio, incluso para cotas de tamaño pequeño.

Korat funciona mejor cuando repOK() “falla mucho”

No funciona bien cuando repOK() triunfa con frecuencia

Material Bibliografico

Korat: Automated Testing Based on Java Predicates, Boyopati et al, 2002.

Korat: A Tool for Generating Structurally Complex Test Inputs, Milicevic et al, 2007.

DEMO