No. 5
OCTUBRE de 2017

quantil

Documentos de Trabajo

Edición electrónica

Muestras Aleatorias a partir de Tipos Abstractos de Datos

Simón Ramírez Amaya
Rodrigo Cardoso Rodríguez

quantil | matemáticas aplicadas

Serie Documentos de Trabajo Quantil, 2017-5
Edición electrónica.

OCTUBRE de 2017

Comité editorial:
Álvaro J. Riascos, CoDirector General y Director Modelos Económicos e I&D
Diego Jara, CoDirector General y Director Matemáticas Financieras
Juan David Martin, Investigador Senior
Juan Pablo Lozano, CoDirector Matemáticas Financieras
Mateo Dulce, Investigador
Natalia Iregui, Directora Administrativa
Simón Ramírez, Director Tecnologías de Información

© 2017, Quantil S.A.S., Estudios Económicos,
Carrera 7 # 77 - 07. Oficina 901, Bogotá, D. C., Colombia
Teléfonos: +57(1) 805 1814
E-mail: info@quantil.com.co
http://www.quantil.com.co

Impreso en Colombia – Printed in Colombia

La serie de Documentos de Trabajo Quantil se circula con propósitos de discusión y divulgación. Los artículos no han sido evaluados por pares ni sujetos a ningún tipo de evaluación formal por parte del equipo de trabajo de Quantil.

Publicado bajo licencia:

Atribución – Compartir igual

Creative Commons: https://co.creativecommons.org

Muestras Aleatorias a partir de Tipos Abstractos de Datos

Simón Ramírez Amaya[1]

Rodrigo Cardoso Rodríguez[2]

[1]Quantil. correo electrónico: simon.ramirez@quantil.com.co
[2]Universidad de los Andes. correo electrónico: rcardoso@uniandes.edu.co

Resumen

El propósito de este trabajo es generar muestras de tipos abstractos de datos (TADS) para la validación estadística de software. Para este fin se diseñó e implementó una metodología que permite muestrear nombres de objetos a partir de la generación de árboles sintácticos que los representen. El trabajo desarrollado permite generar muestras de objetos simples y ofrece herramientas para hacer factible la generación de objetos más complejos.

Índice general

Índice de figuras

Capítulo 1

Introducción

El propósito de este trabajo es diseñar e implementar una metodología para la generación de muestras aleatorias de objetos computacionales. La generación aleatoria de objetos es importante puesto que hace posible el razonamiento probabilístico sobre la corrección de un programa. La solución desarrollada ofrece la posibilidad de muestrear aleatoriamente instancias simples de las estructuras de datos más comunes y ofrece herramientas para hacer factible la generación de objetos más complejos.

La metodología desarrollada se fundamenta en el trabajo existente en dos líneas de investigación. La primera es la Teoría de Tipos Abtractos de Datos (TADS) cuya principal referencia para este proyecto es [1]. La segunda es la Validación Estadística de Software cuyas principales referencias para este proyecto son [7] y [5]. La metodología también se fundamenta en el trabajo seminal en generación y algorítmica de árboles de [4].

El resto del documento tiene la siguiente estructura. El Capítulo 2 presenta una descripción general del proyecto que incluye sus objetivos y los antecedentes relevantes. En el Capítulo 3 se especifican formalmente los problemas a resolver. En el Capítulo 4 se expone el desarrollo del proyecto. En este capítulo se comienza por introducir el marco teórico relevante y se procede a proponer un esquema de solución en los términos introducidos en el marco teórico. En el Capítulo 5 se presenta la implementación de la metodología y se detallan sus principales características. En el capítulo 6 se validan los resultados obtenidos por la implementación para cada uno de los problemas planteados en el Capítulo 3. Finalmente se concluye y se sugieren recomendaciones para trabajos futuros.

Capítulo 2

Descripción General

2.1. Objetivos

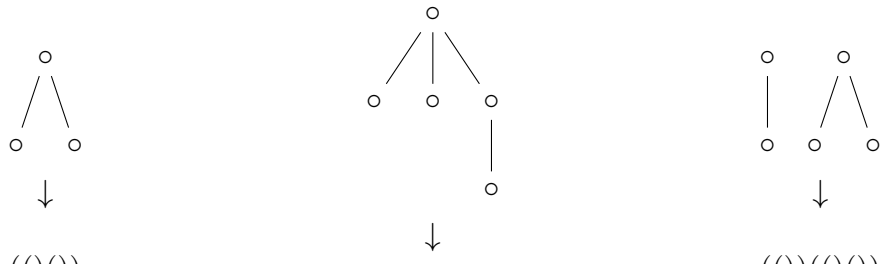
Este proyecto tiene tres objetivos. El primero de ellos es diseñar una metodología para la generación de árboles sintácticos que sirvan para denotar nombres de objetos computacionales arbitrarios. El segundo objetivo es diseñar una metodología de muestreo aleatorio sobre el espacio de árboles sintácticos. El tercer y último objetivo es implementar las metodologías diseñadas para las estructuras de datos más comunes e investigar la complejidad y los límites muestrales de dicha implementación.

2.2. Antecedentes

2.2.1. Árboles y Cadenas Anidadas

[4] estudia la relación existentre entre árboles y cadenas anidadas. El autor explica un mapeo biyectivo entre los dos. Se dice que una cadena $a1a2...a2n$ es anidada si y solo si consta de n caracteres "(", n caracteres ")" y la k-ésima ocurrencia de "(" precede a la k-ésima ocurrencia de ")", $0 \leq k \leq n$. La existencia de un mapeo biyectivo entre árboles y cadenas anidadas hace al problema de generar árboles equivalente al problema de generar cadenas anidadas.

La figura 2.1 presenta algunos ejemplos básicos de equivalencias entre árboles y cadenas anidadas. [5] detalla los procediemintos de traducción de árboles a cadenas y de cadenas a árboles, así como el algoritmo P de [4] para la generación de todas los cadenas anidadas en orden lexicográfico.



(()())

(a) Árbol con raíz (n=3)

(()()(()))

(b) Árbol con raíz (n=5)

(())(()())

(c) Árbol sín raíz (n=5)

Figura 2.1: Ejemplos de árboles como cadenas anidadas

2.2.2. Cadenas Anidadas y Caminatas Aleatorias

[4] estudia la generación aleatoria de cadenas anidadas. El autor aprovecha la estructura recursiva de la generación lexicográfica de cadenas para muestrear de manera uniforme sobre el espacio de todas las cadenas anidadas a partir de caminatas aleatorias.

Sea C_{pq} la cantidad de cadenas anidadas de la forma $(^{q-p}\alpha$ donde $(^{q-p}$ es la cadena formada por $q-p$ paréntesis derechos y α es una cadena con p paréntesis izquierdos y q paréntesis derechos y $0 \leq p \leq q \neq 0$. C_{pq} se puede calcular recursivamente como

$$C_{00} = 1$$

$$C_{pq} = C_{p(q-1)} + C_{(p-1)q} \qquad si \qquad 0 \leq p \leq q \neq 0$$
$$C_{pq} = 0 \qquad si \qquad p > q$$

La estructura recursiva C_{pq} puede visualizarse mediante el arreglo triangular descrito en la figura 2.2. La secuencia de números enteros C_n con $p = q = n \geq 0$ describe la cantidad de árboles con n nodos y corresponde a la secuencia de los Números de Catalán (figura 2.2.(a)). Esta secuencia, así como el número de árboles con n nodos, crece como $O(4^n)$.

El arreglo triangular de la figura 2.2 puede pensarse como un grafo dirigido donde existe un arco desde cada nodo pq a su vecino inmediato a la izquierda $(p, q-1)$ y a su vecino inmediato hacia arriba $(p-1, q)$. En este caso C_{pq} representa el número de caminos desde el nodo (p, q) hasta el nodo $(0, 0)$. Si al realizar un recorrido desde un nodo (p, q) con $p = q$ se representa cada transición al vecino izquierdo con '(' y cada transición al vecino superior con ')', se obtiene tiene un mapeo biyectivo entre caminos y cadenas anidadas. La figura 2.2.(b) muestra la caminata generadora de la cadena anidada del ejemplo 2.1.(a).

q							
0	1						
1	1	1					
2	1	2	2				
3	1	3	5	5			
4	1	4	9	14	14		
5	1	5	14	28	42	42	
6	1	6	20	48	90	132	132
	0	1	2	3	4	5	6

p

(a) C_{pq} y los Números de Catalán

q							
0	1						
1	1	1					
2	1	2	2				
3	1	3	5	5			
4	1	4	9	14	14		
5	1	5	14	28	42	42	
6	1	6	20	48	90	132	132
	0	1	2	3	4	5	6

p

(b) Recorrido ejemplo 2.1.(a)

Figura 2.2: Cadenas anidadas y caminatas sobre arreglos triangulares

Para generar cadenas anidadas de tamaño $2n$ el autor propone realizar caminatas aleatorias desde el nodo (n, n) al nodo $(0, 0)$. A cada paso se decide si tomar a la izquierda (concatenar un paréntesis derecho) o tomar hacia arriba (concatenar un paréntesis izquierdo) en función de la proporción de caminos al nodo $(0, 0)$ desde el nodo fuente que pasan por el nodo destino. La distribución de probabilidad sobre el espacio de caminos desde el nodo (n, n) al nodo $(0, 0)$ que emerge de la algorítmica anteriormente descrita es uniforme.

2.2.3. Tipos Primitivos de Datos

[1] define los tipos primitivos como tipos de datos que se consideran conocidos *a priori* de modo que se puede hablar de variables que toman valores en ellos sin tener que decir nada más. Un tipo primitivo consta de un dominio y de un conjunto de constantes. Los tipos relevantes que se consideran elementales se listan a continuación:

a. Tipo *booleano*
   - Dominio: **bool**
   - Constantes: $\{true, false\}$

b. Tipo *natural*
   - Dominio: **nat**
   - Constantes: $\{0, 1, 2, ...\}$

c. Tipo *entero*
   - Dominio: **int**
   - Constantes: $\{... - 3, -2, -1, 0, 1, 2, 3, ...\}$

2.2.4. Tipos Abstractos de Datos

[1] estudia abstracciones algebraicas de modelos computacionales de la realidad conocidas como tipos abstractos de datos (TADS). Un TAD consta de un conjunto subyacente llamado tipo de interés (notado Y) y de un conjunto de funcionalidades sobre Y y sobre otros TADs conocidos. En el caso general se busca definir un TAD $Y[X]$ donde X es un conjunto finito de TADS o tipos primitivos que hacen las veces de parámetros.
Para comenzar se modela con un conjunto finito I de funciones llamadas *iniciales*, los objetos del tipo de interés considerados más simples. Cada operación inicial i tiene una funcionalidad de la forma

$$i \quad : \quad X_i \quad \rightarrow \quad Y,$$

donde X_i es un producto cartesiano, posiblemente vacío, de algunos de los TADs parámetro. Se continúa definiendo inductivamente el tipo de interés a partir de un conjunto finito C de funciones llamadas *constructoras*, que simulan el mecanismo de construcción de objetos complejos a partir de objetos más simples. Cada operación c tiene una funcionalidad

$$c \quad : \quad Y^m \quad \times \quad X_c \quad \rightarrow \quad Y,$$

donde X_c es un producto cartesiano, posiblemente vacío, de algunos de los TADs parámetro y $m > 0$ es la multiplicidad de c, denotada $m(c)$. El conjunto de funcionalidades $G = I \cup C$ se conoce como conjunto de funciones *generadoras*.

2.2.4.1. TAD $Cola[X]$

[1] presenta el mundo de las colas como un ejemplo simple de construcción inductiva. En primer lugar se considera la cola más simple posible, a saber, la cola que no tiene elementos. Todas las demás colas pueden imaginarse como el resultado de insertar un elemento a una cola más simple. Son suficientes entonces dos operaciones, una que nombre una cola vacía y otra que nombre la inserción de elementos en colas ya construidas, para construir cualquier cola. Las funcionalidades correspodientes que definen la construcción de nombres en el TAD $Cola[X]$ son:

$$\begin{array}{lllllll} * & vac\_cola & : & & & \rightarrow & Cola \\ * & ins & : & Cola & \times \quad X & \rightarrow & Cola \end{array}$$

El tipo de interés del TAD $Cola[X]$ es el conjunto de nombres de la forma:

$$\begin{aligned} Y = \{ & vac\_cola, ins(vac\_cola, x), \\ & ins(ins(vac\_cola, x), x'), \\ & ins(ins(ins(vac\_cola, x), x'), x''), ...\}. \end{aligned}$$

A manera de ejemplo, se considera el TAD $Cola[nat]$. La cola de los 3 números naturales $1, 3, 2$ en la que el 1 es el primer elemento de la cola, el 3 es el segundo y el 2 es el tercero tiene como denotación -o forma de ser consruida- la siguiente expresión pertenciente al tipo de interés:

$$ins(ins(ins(vac\_cola, 1), 3), 2).$$

2.2.4.2. TAD $DCola[X]$

La definición del TAD $Cola[X]$ propuesta por [1] permite la inserción de elementos por un solo extremo de la estructura lineal. [2] propone un segundo tipo de cola en la que es posible insertar elementos por cualquiera de los dos extremos llamada doble cola (*deque* en inglés). El conjunto de funciones *generadoras* del TAD $DCola[X]$ corresponde al conjunto de funciones *generadoras* del TAD $Cola[X]$ junto con una funcionalidad adicional:

$$\begin{array}{lllllll} * & vac\_dcola & : & & & \rightarrow & Cola \\ * & ins\_der & : & Cola & \times\ X & \rightarrow & Cola \\ * & ins\_izq & : & X & \times\ Cola & \rightarrow & Cola \end{array}$$

El tipo de interés del TAD $DCola[X]$ es el conjunto de nombres de la forma:

$$\begin{aligned} Y = \{&vac\_dcola, ins\_der(vac\_dcola, x), ins\_izq(x, vac\_dcola), \\ &ins\_der(ins\_der(vac\_dcola, x), x'), \\ &ins\_der(ins\_izq(x, vac\_dcola), x'), \\ &ins\_izq(x', ins\_der(vac\_dcola, x)), \\ &ins\_izq(x', ins\_izq(x, vac\_dcola)), ...\}. \end{aligned}$$

A manera de ejemplo se considera el TAD $DCola[nat]$. La doble cola de los 3 números naturales $1, 3, 2$ en la que el 1 es el primer elemento de la doble cola, el 3 es el segundo y el 2 es el tercero tendría como denotación -o forma de ser construida- expresiones pertenecientes al tipo de interés como las siguientes:

$$\begin{aligned} &ins\_der(ins\_der(ins\_der(vac\_dcola, 1), 3), 2) \\ &ins\_izq(1, ins\_izq(3, ins\_izq(2, vac\_dcola))). \end{aligned}$$

2.2.4.3. TAD $Arbin[X]$

La construcción inductiva de estructuras no lineales también puede abstraerse algebraicamente. Para el caso de los árboles binarios [1], en primer lugar se considera el árbol binario más simple de todos, a saber, el árbol que no tiene elementos. Todos los demás árboles pueden imaginarse como el resultado de insertar en un árbol binario más simple un elemento y especificar un primer árbol binario como hijo izquierdo y un segundo árbol binario como hijo derecho del elemento insertado. Las funcionalidades que definen la construcción de nombres en el TAD $Arbin[X]$ son:

$$
\begin{array}{lllllllll}
* & vac\_arbin & : & & & & & \rightarrow & Arbin \\
* & cons & : & Arbin & \times & X & \times & Arbin & \rightarrow & Arbin
\end{array}
$$

El tipo de interés del TAD $Arbin[X]$ es el conjunto de nombres de la forma:

$$
\begin{aligned}
Y = \{ & vac\_arbin, cons(vac\_arbin, x, vac\_arbin) \\
& cons(cons(vac\_arbin, x', vac\_arbin), x, vac\_arbin) \\
& cons(vac\_arbin, x, cons(vac\_arbin, x', vac\_arbin)) \\
& cons(cons(vac\_arbin, x', vac\_arbin), x, cons(vac\_arbin, x', vac\_arbin)), ...\}.
\end{aligned}
$$

A manera de ejemplo se considera el TAD $Arbin[nat]$. El árbol binario de los 3 números naturales $1, 3, 2$ en el que el 3 es la raíz, el 1 es el hijo izquierdo y el 2 es el hijo derecho tendría como denotación -o forma de ser construida- la siguiente expresion perteneciente al tipo de interés:

$$cons(cons(vac\_arbin, 1, vac\_arbin), 3, cons(vac\_arbin, 2, vac\_arbin)).$$

2.2.5. Nombres de Objetos y Árboles Sintácticos

[7] propone la interpretación de las clases en el contexto de la Programación Orientada a Objetos (OOP por sus siglas en inglés) como tipos primitivos o abstractos en los que los parámetros se han instanciado con tipos concretos. En este orden de ideas, los objetos son instancias de tipos concretos y por lo tanto son denotables por la expresión que define su construcción inductiva. Dado que estas expresiones hacen parte de un lenguaje formal, para cualquier expresión debe haber un árbol sintáctico que la represente correctamente y que sea igualmente útil para denotar un objeto.

En particular, los ejemplos presentados en las secciones 2.2.4.1 a 2.2.4.3 para los TADS de interés son reresentables a partir de árboles sintácticos. Las figuras 2.4 a 2.7 presentan los árboles sintácticos correspondientes.

2.3. Relevancia del Proyecto

La verificación del hecho de que una solución informática propuesta es en efecto una solución correcta debería ser un proceso efectivo y de costo mínimo. En el contexto de OOP, la generación aleatoria de objetos es importante puesto que hace posible el razonamiento probabilístico sobre la corrección de un programa. Contrario a otras alternativas como el desarrollo de pruebas ad hoc o la verificación formal, la validación estadística mediante la generación aleatoria de objetos de prueba es tanto robusta como eficiente y permite concluir acerca de la correción de un programa con algún grado de confianza [7].

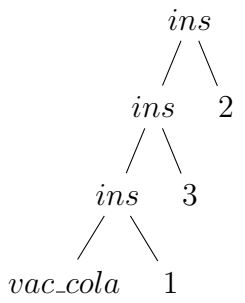


Figura 2.3: Árbol sintáctico para el ejemplo 2.2.4.1

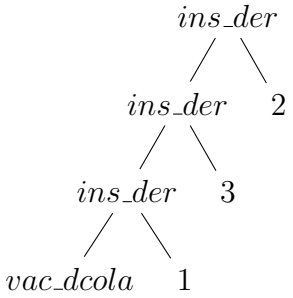


Figura 2.4: Árbol sintáctico para el ejemplo 2.2.4.2 (1)

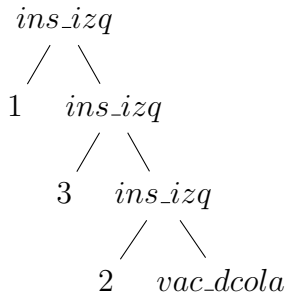


Figura 2.5: Árbol sintáctico para el ejemplo 2.2.4.2 (2)

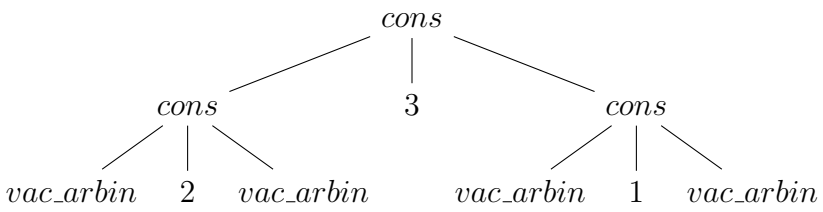


Figura 2.6: Árbol sintáctico para el ejemplo 2.2.4.3

Capítulo 3

Diseño y Especificaciones

3.1. Definición del Problema

Este proyecto ofrece solución a tres problemas correspondientes a los objetivos planteados en 2.1:

(1) Dado un TAD $Y[X]$ y un $n > 0$, encontrar todos los árboles sintácticos de n nodos que denotan nombres en Y.
El problema de generar árboles que denoten nombres en Y se plantea de esta manera puesto que, en general, Y es un conjunto infinito (incluso sin instanciar los parámetros en X a tipos concretos). Ahora bien, mientras no se instancien los tipos parámetros en X a tipos concretos no se tendrán nombres de objetos como tales, aunque sí lo que podría denominarse nombres abstractos. Estos son constructos cuya estructura tiene variables que pueden reemplazarse por elementos de tipos concretos dando lugar a nombres de objetos.

(2) Dados un TAD $Y[X]$, $n > 0$ y $m > 0$ encontrar una muestra aleatoria con reemplazo de tamaño m sobre el conjunto de nombres en Y denotados por árboles sintácticos de n nodos.
Una distribución uniforme discreta es una función de probabilidad P sobre un espacio de resultados Ω de cardinalidad $N \in nat$, $N > 0$, donde para todos los posibles resultados $x \in \Omega$, $P(x) = 1/N$ [6]. En este proyecto se entiende por *muestra aleatoria sobre* Ω un subconjunto $S \subset \Omega$ donde todo $x \in S$ fue elegido bajo una distribución uniforme.

(3) Implementar algoritmos para (1) y (2), estimando sus complejidades computacionales.

3.2. Especificación

El presente es un estudio exploratorio que se restringe a los tipos de datos descritos en 2.2.3 y 2.2.4. En principio, los métodos desarrollados serán generales, aunque para las implementaciones solución no se va a ir a más allá de manejar estos ejemplos sencillos. Así mismo, siempre se trabajará con dominios finitos. Esto implica, por ejemplo, no trabajar directamente con *nat* sino con intervalos finitos arbitrariamente grandes en *nat*. Se espera que las implementaciones solución sean viables en el sentido en que no terminen en estados de excepción por memoria para cualquier especificación de un intervalo finito arbitrariamente grande.

Capítulo 4

Desarrollo

4.1. Marco Teórico

4.1.1. Árboles k-arios

4.1.1.1. Definición

Para $k \geq 0$, un *árbol k-ario* V sobre una bolsa[1] de etiquetas E es una lista[2] de vértices o nodos que cumple una de las siguientes condiciones:

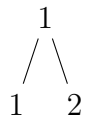
(1) V es la lista vacía de etiquetas de E y se denota $\varnothing$.
(2) $V = \langle r \rangle \quad \& \quad (\quad \&i \quad | \quad 0 < i \leq k \quad : \quad V_i \quad)$, $r \in E$, V_i es un árbol k-ario sobre E, $0 < i \leq k$.

En el caso (1) se dice que el árbol es vacío. En el caso (2) se define r como la raíz del árbol y se dice que $raiz(V) = r$. La lista de *hijos* de r se denota $hijos(r)$ y corresponde a la enumeración de V_i donde V_i no es un árbol vacío, $0 < i \leq k$. También se dice que el *número de hijos* de r es el número de V_i's no vacíos y se denota $grado(r)$. Finalmente, el *j-ésimo* elemento de $hijos(r)$ se denota como $hijos(r, j)$, $0 < j \leq grado(r)$.

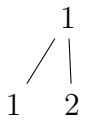
La figura 4.1 presenta múltiples *árboles k-arios* a manera de ejemplos. En la figura 4.1.a se observa un *árbol 2-ario* sobre la bolsa $B = \{1 : 2, 2 : 1\}$. En este caso, $r = 1$, $V_1 = \langle 1 \rangle$, $V_2 = \langle 2 \rangle$, $grado(r) = 2$ e $hijos(r) = \langle \langle 1 \rangle, \langle 2 \rangle \rangle$. En la figura 4.1.b se observa un *árbol 3-ario* sobre la bolsa $B = \{1 : 2, 2 : 1\}$. En este caso, $r = 1$, $V_1 = \langle 1 \rangle$, $V_2 = \langle 2 \rangle$, $V_3 = \varnothing$,

[1] Una *bolsa* o *multiconjunto* (*bag* en inglés) es una agregación de elementos que, contrario a lo usual en conjuntos, permite repeticiones. La notación utilizada sigue a [3]
[2] La definición de listas y las principales operaciones sobre las mismas se detallan en el Glosario de Notaciones - Apéndice C

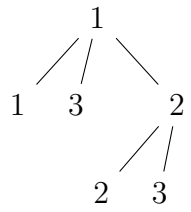
$grado(r) = 2$ e $hijos(r) = \langle\langle 1\rangle, \langle 2\rangle\rangle$. En la figura 4.1.c se observa un *árbol 4-ario* sobre la bolsa $B = \{1 : 2, 2 : 2, 3 : 2\}$. En este caso, $r = 1$, $V_1 = \langle 1\rangle$, $V_2 = \langle 3\rangle$, $V_3 = \varnothing$, $V_4 = \langle 2\rangle\&(\langle 2\rangle\&\langle 3\rangle\&\varnothing\&\varnothing)$, $grado(r) = 3$ e $hijos(r) = \langle\langle 1\rangle, \langle 3\rangle, \langle 2\rangle\&(\langle 2\rangle\&\langle 3\rangle\&\varnothing\&\varnothing)\rangle$.



(a) *Árbol 2-ario*



(b) *Árbol 3-ario*



(c) *Árbol 4-ario*

Figura 4.1: Ejemplos de *árboles k-arios*

4.1.1.2. Notación

Dado un árbol k-ario V, se define la notación de V por la cadena $d(V)$, donde:

- Si V es el árbol vacío entonces $d(V) =$ "".
- De lo contario, $d(V) = r(d(V_1)...d(V_k))$ donde $r = raiz(V)$ y $d(V_j)$, $0 < j \leq k$, es la denotación del j-ésimo subárbol de r.

De acuerdo a esta notación, el *árbol 2-ario* del ejemplo en 4.1.a se denota mediante la expresión:

$$V = 1(1()2()).$$

El *árbol 3-ario* ejemplo en 4.1.b se denota mediante la expresión:

$$V = 1(1()2()).$$

El *árbol 4-ario* ejemplo en 4.1.c se denota mediante la expresión:

$$V = 1(1()3()2(2()3())).$$

4.1.1.3. Equivalencia

Se considera que dos árboles k-arios son equivalentes si tienen la misma notación. Es decir, si V_1 y V_2 son árboles k-arios entonces:

$$V_1 = V_2 \equiv d(V_1) = d(V_2)$$

Por ejemplo, los árboles ejemplo 4.1.a y 4.1.b son árboles equivalentes. Por el contrario, los árboles ejemplo 4.1.b y 4.1.c no lo son.

4.1.1.4. Tamaño

El tamaño de un árbol k-ario V es su cantidad de nodos y se denota $tam(V)$, donde:

- $tam(\varnothing) = 0$.
- $tam(\quad \langle r \rangle \quad \& \quad (\&i|0 < i \leq k : V_i) \quad) = 1 + (+i|0 < i \leq k : tam(V_i))$

Los tamaños de los árboles ejemplo 4.1.a, 4.1.b y 4.1.c son 3, 3 y 6 respectivamente.

4.1.1.5. Recorrido en preorden

El *recorrido en preorden* de un *árbol k-ario* V es una lista de etiquetas que se denota $preorden(V)$ y se define como:

- $preorden(\varnothing) = \langle\rangle$.
- $preorden(\quad \langle r \rangle \quad \& \quad (\&i|0 < i \leq k : V_i) \quad) = \langle r \rangle \quad \& \quad (\&i|0 < i \leq k : preorden(V_i))$

El *recorrido en preorden* del ejemplo 4.1.a es

$$preorden(V) = \langle 1, 1, 2 \rangle$$

El *recorrido en preorden* del ejemplo 4.1.b es

$$preorden(V) = \langle 1, 1, 2 \rangle$$

Finalmente, El *recorrido en preorden* del ejemplo 4.1.c es

$$preorden(V) = \langle 1, 1, 3, 2, 2, 3 \rangle$$

4.1.2. Tipos Primitivos y Funciones Generadoras

4.1.2.1. Tipos Primitivos Finitos

Un tipo primitivo finito es un tipo primitivo cuyo dominio es un subconjunto finito de los tipos primitivos definidos en 2.2.4. Por ejemplo, el tipo primitivo *bool* es también un tipo primitivo finito. Por el contrario, el tipo primitivo *nat* y el tipo primitivo *int* son tipos primitivos no finitos.

Se puede pensar en definir un tipo primitivo finito a partir de la restricción de un tipo primitivo no finito. Son de interés para este proyecto los tipos primitivos finitos definidos a partir de la consideración de intervalos cerrados de constantes de los tipos *nat* e *int*. Para $a \in nat$ y $b \in nat$ con $a \leq b$, se denota el tipo primitivo finito correspondiente a $[a, b]$ como $nat[a, b]$, con dominio $nat@a\#b@$ y constantes $\{a, ..., b\}$. Así mismo, para

$c \in int$ y $d \in int$ con $c \leq d$, se denota el tipo primitivo finito correspondiente a $[c, d]$ como $int[c, d]$, con dominio $int@c\#d@$ y constantes $\{c, ..., d\}$. Como resulta evidente, el tamaño de los tipos primitivos considerados es:

$$|bool| = 2$$

$$|nat@a\#b@| = b - a + 1$$

$$|int@c\#d@| = d - c + 1$$

4.1.2.2. Funciones Generadoras

Un tipo primitivo finito también puede describirse mediante la notación introducida en 2.2.4 para denotar TADS. La construcción de nombres en un tipo primitivo finito $tipoprimitivofinito$ con dominio tpf y constantes $\{c_1...c_n\}$ puede representarse mediante n funciones iniciales de la forma:

$$* \quad c_i \quad : \qquad \rightarrow \quad tpf \qquad 1 \leq i \leq n$$

Sin embargo, dado que el dominio de un tipo primitivo finito puede ser arbitrariamente grande y muestrear aleatoriamente sobre el dominio es un problema trivial, esta representación es poco económica. Se prefiere representar $tipoprimitivofinito$ con una única función inicial de la forma:

$$* \quad tpf \quad : \qquad \rightarrow \quad tpf$$

donde el nombre de la función generadora corresponde al nombre del dominio del tipo primitivo finito y por lo tanto aporta la información necesaria para conocer de manera inequívoca los nombres del tipo primitivo. Esta representación supone que se sigue una notación para tipos primitivos finitos como la sugerida en 4.1.2.1, donde el nombre del dominio del tipo permite determinar sin ambiguedades cuál es el conjunto finito de nombres incluido en el tipo.

4.1.3. TADS y Funciones Generadoras

4.1.3.1. Caracterización

Sea $Y[X]$ un TAD cualquiera con un conjunto finito de funciones iniciales $I = \{i_1, ..., i_l\}$, un conjunto finito de funciones constructoras $C = \{c_1, ..., c_m\}$, un conjunto de funciones generadores $G(Y) = I \cup C$ y un conjunto finito de TADs parámetro $X = \{X_1, ..., X_n\}$.

Por construcción sabemos que cualquier función $f \in G$ es de la forma

$$f \quad : \quad \prod_{j=1}^{aridad(f)} Z_j \quad \rightarrow \quad Y$$

donde debe valer que

$$(\forall j \quad | \quad 0 < j \leq aridad(f) \quad : \quad Z_j \in X \cup \{Y\})$$

La expresión $Z(f, j)$ con $0 < j \leq aridad(f)$ denota el j-ésimo TAD que compone el producto cartesiano que define el dominio de f.

Para ilustrar los conceptos introducidos en esta sección se considera el caso de la función generadora *cons* del TAD $Arbin[X]$. En este caso, $aridad(cons) = 3$, $Z(cons, 1) = Z(cons, 3) = Arbin[X]$ y $Z(cons, 2) = X$.

4.1.3.2. Nombre

Dada una función generadora f de la forma descrita en 4.1.3.1, la expresión $nom(f)$ denota el nombre de la función. Por ejemplo, para las funciones generadoras *vac_arbin* y *cons* del TAD $Arbin[X]$, $nom(vac_arbin)$ = "*vac_arbin*" $nom(cons)$ = "*cons*".

4.1.4. Conformidad

4.1.4.1. Definición

Para $k \geq 0$ sea t un árbol k-ario no vacío con raíz r sobre una bolsa de etiquetas E. Sea $Y[X]$ un TAD y f una función tal que $f \in G(Y)$. Decimos que r es *conforme respecto a* f si el grado de r es igual a la aridad de f. Formalmente:

$$con(r, f) \equiv grado(r) = aridad(f)$$

4.1.4.2. Ejemplo

La raíz r del árbol en la figura 2.1.b es conforme respecto a la función generadora *cons* del TAD $Arbin[X]$ puesto que $grado(r) = aridad(cons) = 3$. La raíz no es conforme respecto a ninguna otra función generadora en los TADS conocidos.

4.1.5. Listas de Etiquetas Sintácticas y Aptitud

4.1.5.1. Definición

Dado un TAD $Y[X]$ y un *árbol k-ario* sobre una bolsa de etiquetas E, se pretende encontrar el conjunto de listas de etiquetas sintácticas (nombres de funciones generadoras) tales que, al recorrer V en preorden dejando las etiquetas, el árbol sintáctico resultante represente correctamente nombres en $Y[X]$. Este conjunto se denotará $L(V, Y[X])$.

Sea $r = raiz(V)$ y G el conjunto de funciones generadoras de $Y[X]$. Se comienza por construir un conjunto de tuplas anidadas [3] $T(V, Y[X])$ que represente el producto cartesiano de los conjuntos de etiquetas conformes respecto a los TADS correspondientes en $Y[X]$ para cada nodo de V. Este conjunto de tuplas anidadas se define recursivamente utilizando el concepto de conformidad introducido en 4.1.4:

$$T(V, Y[X]) \equiv \begin{cases} \bigcup\limits_{\{f \in G | con(r,f)\}} \{nom(f)\} & grado(r) = 0 \\ \bigcup\limits_{\{f \in G | con(r,f)\}} \{nom(f)\} \quad \times & \\ \qquad \prod\limits_{0 < j \leq grado(r)} T(hijos(r,j), Z(f,j)) & grado(r) \neq 0 \end{cases}$$

$L(V, Y[X])$ es simplemente el conjunto resultante de linealizar cada una de las tuplas en $T(V, Y[X])$ de acuerdo a lo mencionado en el Apéndice C (Glosario de Notaciones). Si $T(r, Y[X]) = \{u_1, ..., u_p\}$, $p \geq 0$, entonces:

$$L(V, Y[X]) = (\quad \bigcup i \quad | \quad 0 < i \leq p \quad : \quad lin(u_i) \quad)$$

Finalmente, el conjunto de *árboles sintácticos representativos de nombres* resultante de recorrer V dejando las etiquetas de las listas en $L(V, Y[X])$ se denota $A(V, Y[X])$. Se dice que V es un *árbol apto respecto a* $Y[X]$ si $A(V, Y[X]) \neq \varnothing$.

[3] La definición de tuplas anidadas y las principales operaciones sobre las mismas se detallan en el Glosario de Notaciones - Apéndice C.

4.1.5.2. Ejemplo

Sea V el árbol en la figura 2.1.a. Sea también $ejemplo[nat@0\#9@]$ un TAD con las siguientes funciones generadoras:

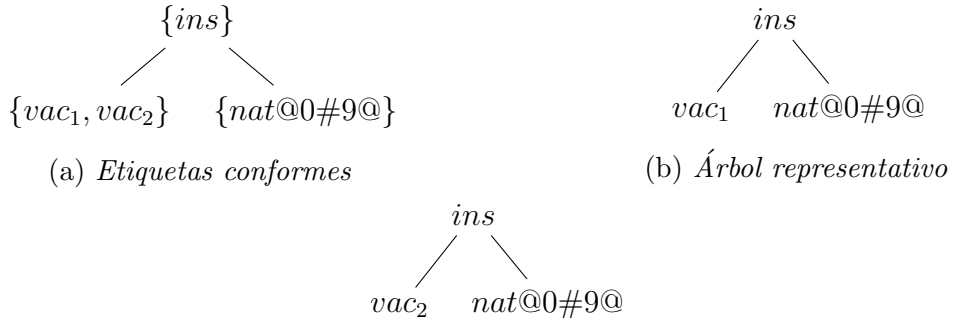
$$
\begin{array}{llllll}
* & vac\_1 & : & & \rightarrow & ejemplo \\
* & vac\_2 & : & & \rightarrow & ejemplo \\
* & ins & : & ejemplo \times nat@0\#9@ & \rightarrow & ejemplo
\end{array}
$$

La figura 4.2.a presenta las etiquetas de las funciones generadoras conformes para cada nodo de V. En este caso:

$$L(raiz(V), Y[X]) = \{(ins, (vac\_1, nat@0\#9@)), (ins, (vac\_2, nat@0\#9@))\}$$

$$S(raiz(V), Y[X]) = \{\langle ins, vac\_1, nat@0\#9@\rangle, \langle ins, vac\_2, nat@0\#9@\rangle\}$$

Las figuras 4.2.b y 4.2.c presentan los árboles resultantes de recorrer V en preorden dejando las etiquetas de cada una de las listas en $L(V, Y[X])$. Cada uno de estos árboles síntacticos representa nombres en Y y por lo tanto se puede decir que V es *apto respecto a* $ejemplo[nat@0\#9@]$.



(a) *Etiquetas conformes*

(b) *Árbol representativo*

(c) *Árbol representativo*

Figura 4.2: Ejemplo 4.1.5.2

4.1.6. Árboles Sintácticos Representativos

4.1.6.1. Definición

El conjunto de *árboles sintácticos representativos de nombres* $A(V, Y[X])$ se puede construir para cualquier *árbol k-ario* V y TAD $Y[X]$. En particular, se puede calcular

para todos los diferentes *árboles* de tamaño n sobre la bolsa se etiquetas $N = \{\text{“”} : n\}$ que contiene el string vacío con multiplicidad n. Este cojunto de *árboles k-arios* sobre N se puede encontrar utilizando el algoritmo P para la generación de cadenas anidadas mencionado en la sección 2.2 y se denotará P_3. Por ejemplo, para $n = 3$:

$$P_n = \{()()(), ()(()), (())(), (()()), ((()))\}$$

Dados $n > 0$ y un TAD $Y[X]$, el conjunto de todos los *árboles sintácticos representativos de nombres* en $Y[X]$ de tamaño n se denota $A(n, Y[X])$ y es el resultado de unir $A(V, Y[X])$ sobre todo $V \in P(n)$. Formalmente:

$$A(n, Y[X]) = (\quad \bigcup V \quad | \quad V \in P(n) \quad : \quad A(V, Y[X]) \quad)$$

4.1.6.2. Ejemplos

(1) Para $n = 3$ y $Y = Cola[int@0.,9@]$ se tiene que:

$$A(3, Cola[int@0.,9@]) = \{ins\_der(vac\_cola()nat@0\#9@())\}$$

(2) Para $n = 3$ y $Y = DCola[int@0.,9@]$ se tiene que:

$$A(3, DCola[int@0.,9@]) = \{ins\_izq(nat@0\#9@()vac\_dcola()),\\ ins\_der(vac\_dcola()nat@0\#9@())\}$$

(3) Para $n = 3$ y $Y = Arbin[int@0.,9@]$ se tiene que:

$$A(3, Arbin[int@0\#9@]) = \{\}$$

(4) Para $n = 4$ y $Y = Arbin[int@0.,9@]$ se tiene que:

$$A(4, Arbin[int@0.,9@]) = \{cons(vac\_arbin()int@0\#9@()vac\_arbin())\}$$

(5) Para $n = 4$ y $Y = Arbin[bool]$ se tiene que:

$$A(4, Arbin[bool]) = \{cons(vac\_arbin()bool()vac\_arbin())\}$$

(6) Para $n = 4$ y $Y = Arbin[nat@0\#9@, bool]$ se tiene que:

$$A(4, Arbin[int@0\#9@, bool]) = \{cons(vac\_arbin()int@0\#9@()vac\_arbin()),\\ cons(vac\_arbin()bool()vac\_arbin())\}$$

4.1.7. Representatividad de Árboles Sintácticos

Para $n > 0$ y un TAD $Y[X]$, sea $A(n, Y[X_0])$ el conjunto de árboles sintácticos resultante de reemplazar las etiquetas de tipos primitivos finitos en $A(n, Y[X])$ por elementos del dominio del tipo primitivo finito. Se dice que un *árbol* $V \in A(n, Y[X])$ con etiquetas en las generadoras de X_0 *representa* al subconjunto de $A(n, Y[X_0])$ resultante de reemplazar las generadoras por elementos del tipo primitivo finito. La unión sobre estos subconjuntos es justamente $A(n, Y[X_0])$. Además dos *árboles* $V, V' \in A(n, Y[X])$ tales que $V \neq V'$ representan conjuntos disyuntos y por lo tanto los conjuntos representados por todos los árboles en $A(n, Y[X])$ son una partición de $A(n, Y[X_0])$.

La representatividad de un árbol V se nota $rep(V)$ y es el tamaño del subconjunto de árboles en $A(n, Y[X_0])$ que representa. Sea $E(V) = \{\text{“}e_1^V\text{”} : m_1, ..., \text{“}e_p^V\text{”} : m\_p\}$ con $0 \leq p \leq n$ la bolsa de etiquetas sintácticas de funciones generadoras de tipos primitivos finitos en V notadas según las convenciones descritas en 4.1.2. Entonces

$$rep(V) = \prod_i |e_i^V|^{m_i}$$

Se consideran algunos de los ejemplos presentados en 4.1.6.2:

$$rep(4.1.6.2.(1)) = 10$$

$$rep(4.1.6.2.(4)) = 10$$

$$rep(4.1.6.2.(5)) = 2$$

4.1.8. Distribución Representativa

Para $n > 0$ y $Y[X]$ un TAD cualquiera, la *distribución de probabilidad representativa* sobre el espacio de árboles $A(n, Y[X])$ es la distribución de probabilidad en la que la probabilidad de un árbol $V \in A(n, Y[X])$ es igual a la ponderación de su representatividad. Denotamos a esta distribución como $\Phi(V)$. Formalmente:

$$\phi(V) = \frac{rep(V)}{\left(\quad \sum V \quad | \quad V \in A(n, Y[X]) \quad : \quad rep(V) \quad \right)}$$

Es ilustrativo notar que $rep(V)$ es la cantidad de nombres en $Y[X]$ que representa V y que la sumatoria en el denominador es la cantidad total de nombres en $Y[X]$ representables a partir de árboles de tamaño n.

Se consideran los ejemplos 4.1.6.2.(3) y 4.1.6.2.(6). En el caso del ejemplo 4.1.6.2.(3):

$$\phi(ins\_izq...) = \frac{10}{20}$$

$$\phi(ins\_izq...) = \frac{10}{20}$$

En el caso del ejemplo 4.1.6.2.(6):

$$\phi(cons(vac\_arbin(int...) = \frac{10}{12}$$

$$\phi(cons(vac\_arbin(bool...) = \frac{2}{12}$$

4.2. Diseño de la Solución

4.2.1. Diseño

Se propone el siguiente esquema de solución para los problemas (1) y (2) planteados en la sección 3.1:

1. Generar todos los árboles n-arios diferentes de tamaño n sobre la bolsa de etiquetas N. Para este propósito se utiliza el algoritmo P para la generación lexicográfica de paréntesis anidados.
   Para cada árbol V generado:
   a) Generar el conjunto de tuplas anidadas de etiquetas $T(raiz(V), Y[X])$.
   b) Para cada tupla anidada *tup* no vacía generada en (1.a):
      1) Linealizar *tup* a una secuencia s.
      2) Etiquetar V con s a partir de un recorrido en preorden.
2. Para generar una muestra aleatoria de tamaño m:
   a) Para $1...m$:
      1) Escoger un árbol V del conjunto de árboles etiquetados generado en (1) utilizando la distribución de probabilidad representativa $\Phi(t)$.
      2) Reemplazar todas las etiquetas de tipos primitivos finitos en V por etiquetas de elementos utilizando la distribución uniforme sobre el dominio del tipo primitivo finito.

La solución propuesta permite desacoplar la generación de árboles sintácticos representativos del muestreo aleatorio de nombres. En términos prácticos esto significa que no es necesario recalcular el conjunto de árboles sintácticos representativos para generar múltiples muestras aleatorias de nombres.

4.2.2. Complejidad en Tiempo y Espacio

4.2.2.1. Generación de Árboles Sintácticos

Para calcular la compejidad temporal de la solución para el problema (1) se estima el costo de cada uno de los pasos. Para comenzar, se debe generar C_n cadenas anidadas utilizando el algoritmo P. Como se mencionó en la sección 2.2.2, C_n crece como $O(4^n)$. La traducción de cada cadena anidada a un árbol cuesta una iteración sobre la cadena anidada. Es decir, esta traducción cuesta exactamente $2n$ operaciones adicionales. Ahora bien, para cada uno de estos árboles hay que generar un número adicional de *árboles sintácticos representativos de nombres* que, como se puede observar en la sección 4.1.5.1 y en el ejemplo 4.1.5.2, depende fundamentalmente de la cantidad de funciones generadoras en el TAD $Y[X]$ que sean conformes para cada uno de los n nodos del árbol generado a partir de la cadena anidada. Dado que el conjunto de *árboles sntácticos representativos de nombres* se construye a partir del producto generalizado de los conjuntos de funciones generadoras conformes sobre todos los nodos, si la cardinalidad de los conjuntos de funciones generadoras conformes para todos los nodos es q, $q > 0$, entonces se deben generar q^n árboles adicionales. Es decir que

$$T(n, q) = O(4^n * (2n + q^n))$$

En el caso de los TADS $Cola[X]$ y $Arbin[X]$, cada función generadora tiene una aridad diferente y por lo tanto la complejidad temporal es:

$$T(n, 1) = O(4^n * 2n)$$

En el caso del TAD $DCola[X]$, dos funciones generadoras tienen la misma aridad y por lo tanto su complejidad en el peor de los casos es:

$$T(n, 2) = O(4^n * (2n + 2^n))$$

Para calcular la complejidad espacial se tiene en cuenta el número de árboles que deben coexistir simultaneamente en memoria. En el peor de los casos:

$$S(n, q) = O(q^n + 1)$$

4.2.2.2. Generación de Nombres

Para calcular la complejidad de la solución para el problema(2) se considera como operación básica la generación de un árbol. Sea n, el tamaño de los nombres a generar, a el tamaño del conjunto de cadenas anidadas que denotan *árboles sintácticos representativos de nombres* y m el tamaño muestral. Para poder muestrear nombres se requiere traducir a cadenas anidadas con un costo de $2n$ por cadena anidada. Adicionalmente, se requiere

generar m árboles adicionales para construir la muestra. Cada árbol en la muestra requiere a lo sumo n reemplazos de etiquetas de tipos primitivos finitos por elementos en su dominio. Es decir que:

$$T(a, n, m) = O(a * 2n + m * n)$$

$$S(a, n, m) = O(a + m)$$

Capítulo 5

Implementación

Generador de Nombres (GNom) es una aplicación para la generación de muestras de TADS. La aplicación ofrece dos funcionalidades:

1. Dado un TAD y un tamaño de árbol como parámetros, generar el conjunto de árboles sintácticos representativos en un archivo de texto.
2. Dado un TAD, un tamaño de árbol y un tamaño muestral como parámetros, generar una muestra aleatoria de nombres en un archivo de texto.

5.1. Descripción

GNom es una aplicación desarrollada en Python con una arquitectura de dos niveles. La aplicación consta de una interfaz de línea de comando (CLI por sus siglas en inglés) que consume los servicios ofrecidos por dos módulos independientes que contienen la lógica funcional.

5.1.1. Dependencias

La aplicación principal hace uso de las dependencias *NumPy* y *matplotlib*. *NumPy* es el principal paquete de análisis estadístico para Python y *matplotlib* es una librería para la generación de gráficas en 2D. Para ejecutar las pruebas descritas en el capítulo 6 se utiliza el framework de pruebas unitarias para Python *pytest*.

5.1.2. Disponibilidad y Ejecución

Las instrucciones de instalación y uso de GNom se detallan en el manual de uso (Apéndice A de este documento).

5.2. Tipos de Datos Soportados

GNom soporta los TADS y tipos primitivos finitos mencionados en este documento:

- TADS
  1. $Cola[X]$
  2. $DCola[X]$
  3. $Arbin[X]$
- Tipos Primitivos Finitos
  1. $bool$
  2. $nat@a\#b@$
  3. $int@c\#d@$

Capítulo 6

Validación

6.1. Métodos

6.1.1. Generación de Árboles Sintácticos

Se propone como método de validación para la generación de árboles sintácticos verificar que el conjunto de árboles generado por GNom corresponde al conjunto de árboles que describen correctamente los objetos en $Y[X]$ de acuerdo a la notación introducida en 4.1.1 y 4.1.2. Para instancias pequeñas del problema el cálculo manual de la solución es factible y por lo tanto la validación empírica de los resultados es posible. En el Apéndice A se detalla la ubicación de las soluciones calculadas manualmente.

Para este proposito se implementaron 15 pruebas en las que se verifica que el conjunto de árboles generado por GNom corresponda al conjunto de árboles solución calculado manualmente. Con el fin de hacer las pruebas lo más amplias posibles se consideraron instancias heterogéneas del problema. Las instancias se listan a continuación:

1. $n = 3$, $Y[X] = Cola[bool]$
2. $n = 3$, $Y[X] = Cola[nat@0\#9@]$
3. $n = 3$, $Y[X] = Cola[bool, nat@0\#9@]$
4. $n = 3$, $Y[X] = DCola[bool]$
5. $n = 4$, $Y[X] = Arbin[bool]$
6. $n = 4$, $Y[X] = Arbin[Cola[nat@0\#9@]]$
7. $n = 5$, $Y[X] = Cola[bool]$
8. $n = 5$, $Y[X] = DCola[bool]$
9. $n = 5$, $Y[X] = DCola[bool, int@-3\#3@]$
10. $n = 6$, $Y[X] = Arbin[Cola[nat@0\#9@]$
11. $n = 7$, $Y[X] = Arbin[bool]$
12. $n = 7$, $Y[X] = Cola[bool]$

13. $n = 7$, $Y[X] = DCola[bool]$
14. $n = 9$, $Y[X] = Arbin[Cola[nat@0\#9@]]$
15. $n = 10$, $Y[X] = Arbin[bool]$

6.1.2. Muestreo Aleatorio de Nombres

Se propone como método de validación para el muestreo aleatorio de nombres verificar gráficamente que la distribución empírica de las muestras converge a la distribución uniforme sobre el espacio de nombres cuando el tamaño muestral aumenta. Para este propósito GNom ofrece la funcionalidad de graficar un histograma muestral a petición del usuario.

Como en el caso de la validación para la generación de árboles, se considera un conjunto de instancias heterogéneas del problema. Estas instancias se listan a continuación:

1. $n = 3$, $Y[X] = Cola[bool]$, $m = \{5e1, 5e2, 5e3, 5e4\}$
2. $n = 3$, $Y[X] = Cola[nat@0\#9@]$, $m = \{1e2, 1e3, 1e4, 1e5\}$
3. $n = 5$, $Y[X] = DCola[bool, int@ - 3\#3@]$, $m = \{2e3, 2e4, 2e5, 1e6\}$
4. $n = 7$, $Y[X] = Arbin[nat@0\#19@]$, $m = \{1e4, 1e5, 1e5, 3e6\}$

6.1.3. Complejidad y Límites Muestrales

6.1.3.1. Generación de Árboles Sintácticos

Para validar los resultados presentados en 4.2.2 y estimar los límites muestrales de la solución para generar árboles sintácticos se propone monitorear el tiempo que demora GNom en solucionar varias instancias del problema variando el tamaño de entrada n. Se incluyen tres TADS diferentes con el fin de validar el planteamiento de la generación de un árbol como operación básica para estimar la complejidad temporal. Las instancias consideradas son:

- $n = \{1, ..., 17\}$, $Y[X] = Cola[nat@0\#9@]$
- $n = \{1, ..., 17\}$, $Y[X] = Cola[nat@0\#9@]$
- $n = \{1, ..., 17\}$, $Y[X] = Arbin[nat@0\#9@]$

6.1.3.2. Generación de Nombres

Para validar los resultados presentados en 4.2.2 y estimar los límites muestrales de la solución para muestrear nombres se propone monitorear el tiempo que demora GNom

en solucionar varias instancias del problema variando el tamaño de entrada m. Las instancias consideras son:

- $m = \{1e1, ..., 1e5\}$, $n = 15$, $Y[X] = Cola[nat@0\#9@]$
- $m = \{1e1, ..., 1e6\}$, $n = 15$, $Y[X] = DCola[nat@0\#9@]$
- $m = \{1e1, ..., 1e6\}$, $n = 16$, $Y[X] = Arbin[nat@0\#9@]$

6.2. Resultados

6.2.1. Generación de Árboles Sintácticos

El resultado de las pruebas planteadas en 6.1.1 es satisfactorio. Es decir, en cada uno de los casos el conjunto de árboles sintácticos generado por GNom corresponde al conjunto solución. En el manual de uso (Apéndice A) se detalla el proceso para reproducir las pruebas utilizando el framework de pruebas para Python *pytest*, así como la localización de los archivos de salida y de validación de las pruebas.

6.2.2. Muestreo Aleatorio de Nombres

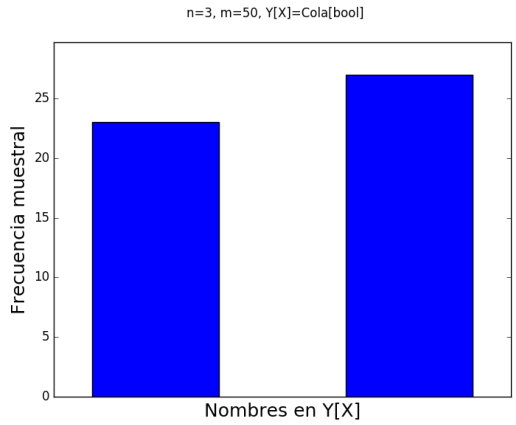
El resultado de las pruebas planteadas en 6.1.2 es satisfactorio. En cada uno de los casos se observa que la distribución empírica de las muestras converge a la distribución uniforme a medida que aumenta el tamaño muestral. Las secuencia de histogramas correspondientes para el resto de las pruebas planteadas en 6.1.1 se pueden generar siguiendo las instrucciones incluidas en el manual de uso (Apéndice A). A continuación se incluyen algunos de los resultados de las pruebas a manera de ejemplo.

6.2.2.1. $n = 3$, $Y[X] = Cola[bool]$

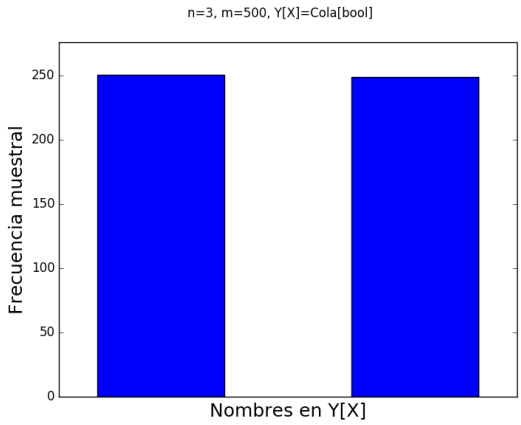
La figura 6.1 muestra la convergencia de la distribución muestral sobre el espacio de 2 nombres.

6.2.2.2. $n = 3$, $Y[X] = Cola[nat@0\#9@]$

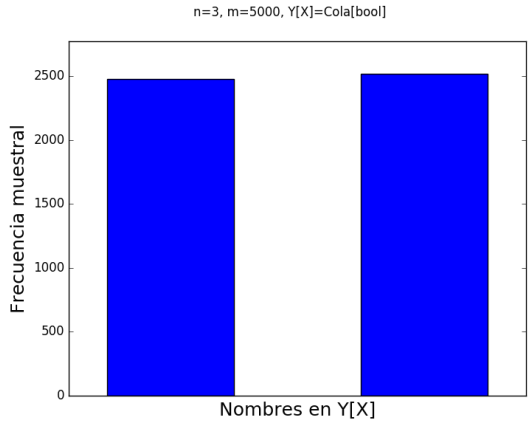
La figura 6.2 muestra la convergencia de la distribución muestral sobre el espacio de 10 nombres.



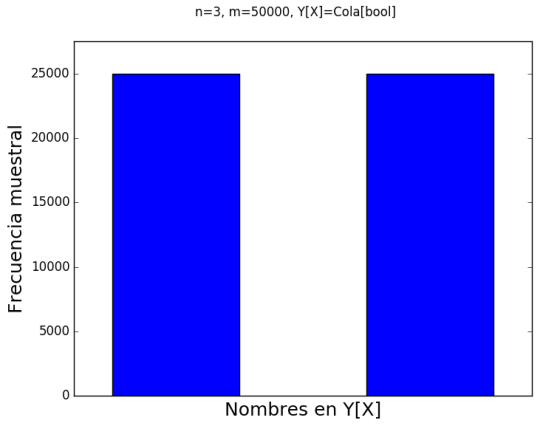
(a) Distribución muestral ($m = 5e1$)



(b) Distribución muestral ($m = 5e2$)



(c) Distribución muestral ($m = 5e3$)



(d) Distribución muestral (m=$m = 5e3$)

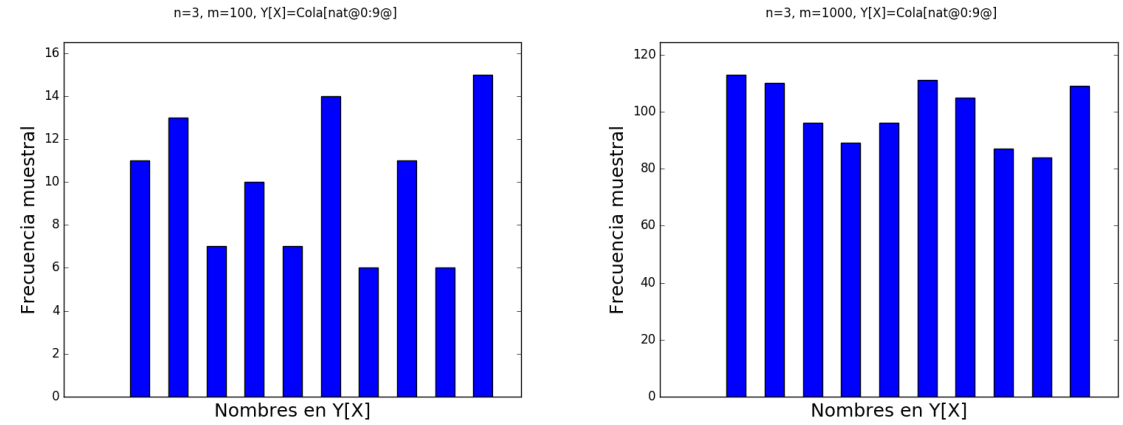
Figura 6.1: Validación para la prueba 6.1.2.1

6.2.2.3. $n = 5$, $Y[X] = DCola[bool, int@ - 3\#3@]$

La figura 6.3 muestra la convergencia de la distribución muestral sobre el espacio de 324 nombres.

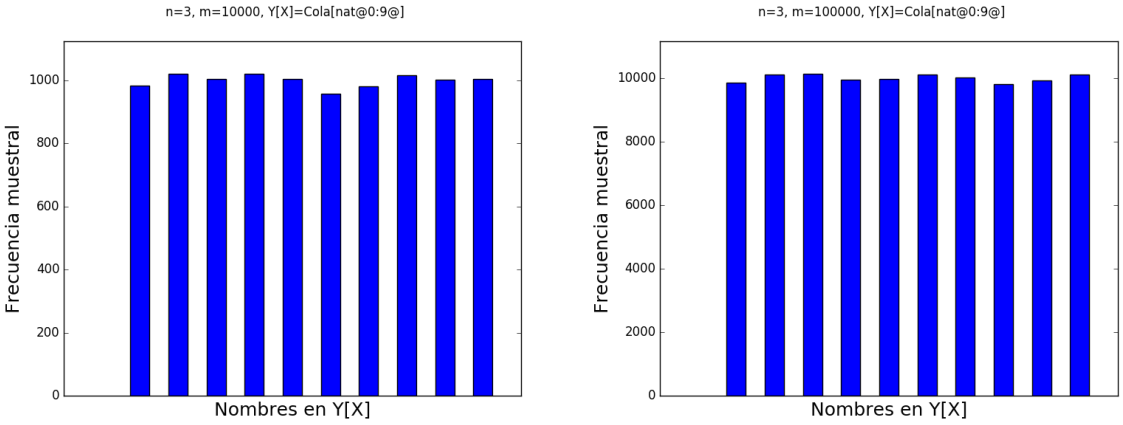
6.2.2.4. $n = 7$, $Y[X] = Arbin[nat@0\#19@]$

La figura 6.4 muestra la convergencia de la distribución muestral sobre el espacio de 800 nombres.



(a) Distribución muestral ($m = 1e2$)

(b) Distribución muestral ($m = 1e3$)



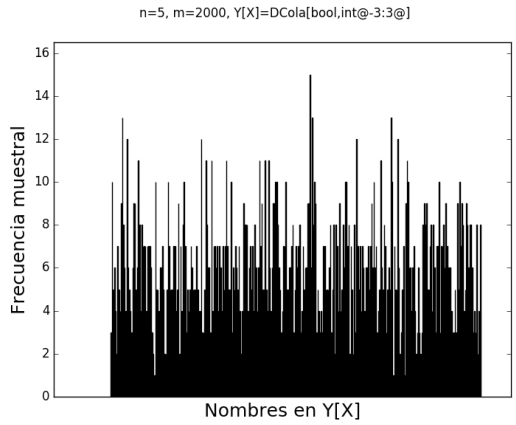
(c) Distribución muestral ($m = 1e4$)

(d) Distribución muestral ($m = 1e5$)

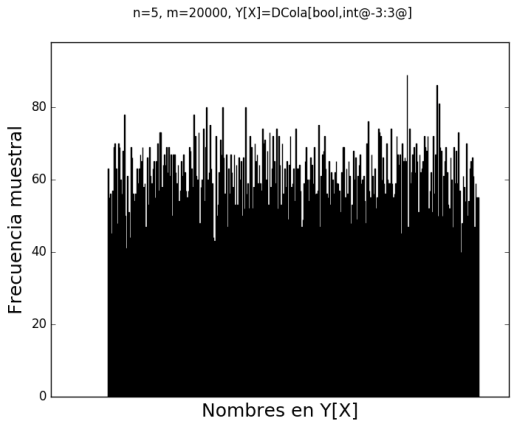
Figura 6.2: Validación para la prueba 6.1.2.2

6.2.3. Complejidad y Límites Muestrales

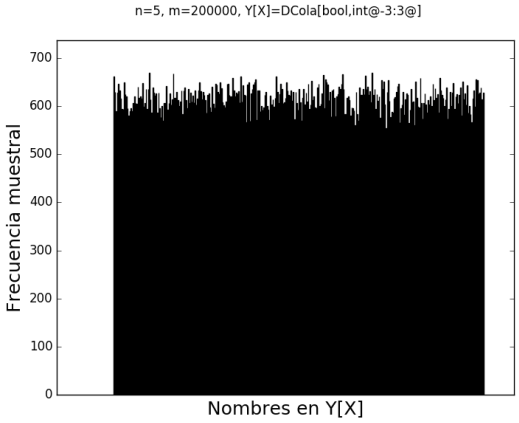
Los resultados experimentales sugieren que el análisis realizado en 4.2.2 es correcto. En la figura 6.5 se observa que el tiempo de ejecución de la solución para generar árboles crece de manera exponencial respecto al tamaño n. Así mismo, también se observa que la generación para el TAD $DCola[X]$ tiende a crecer más rápidamente que los otros TADS incluidos en la muestra. En la figura 6.6 se observa que el tiempo de ejecución para la generar muestras de nombres crece linealmente respecto al tamaño muestral m.



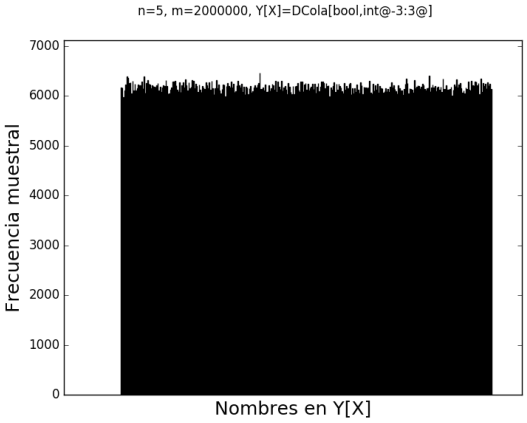
(a) Distribución muestral ($m = 2e3$)



(b) Distribución muestral ($m = 2e4$)

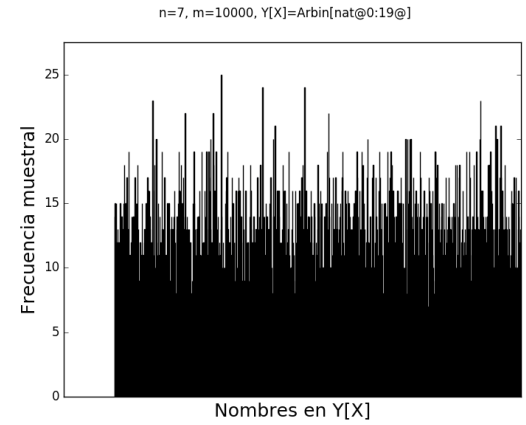


(c) Distribución muestral (m=$m = 2e5$)

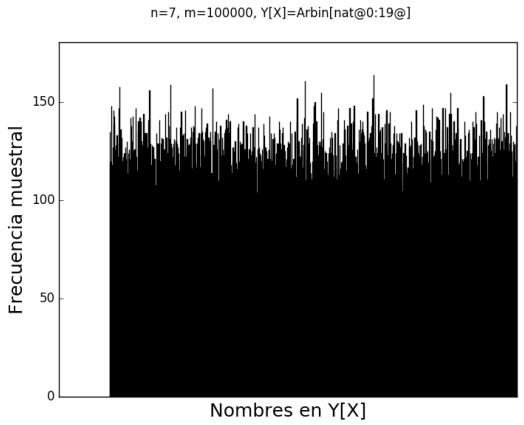


(d) Distribución muestral ($m = 2e6$)

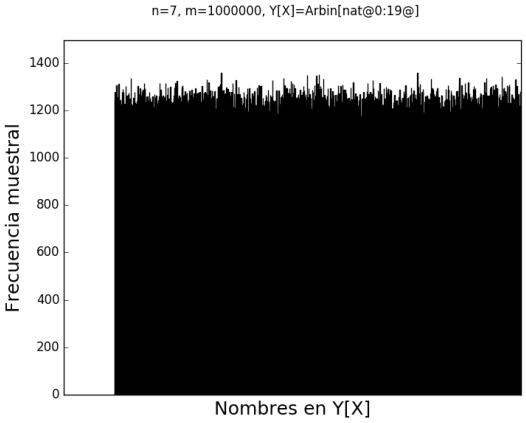
Figura 6.3: Validación para la prueba 6.1.2.3



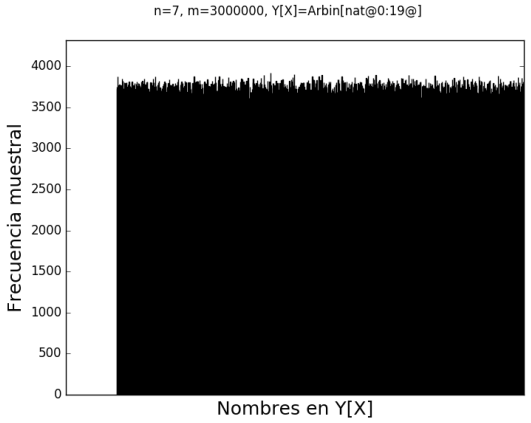
(a) Distribución muestral ($m = 1e4$)



(b) Distribución muestral ($m = 1e5$)



(c) Distribución muestral (m=$m = 1e6$)



(d) Distribución muestral ($m = 3e6$)

Figura 6.4: Validación para la prueba 6.1.2.4

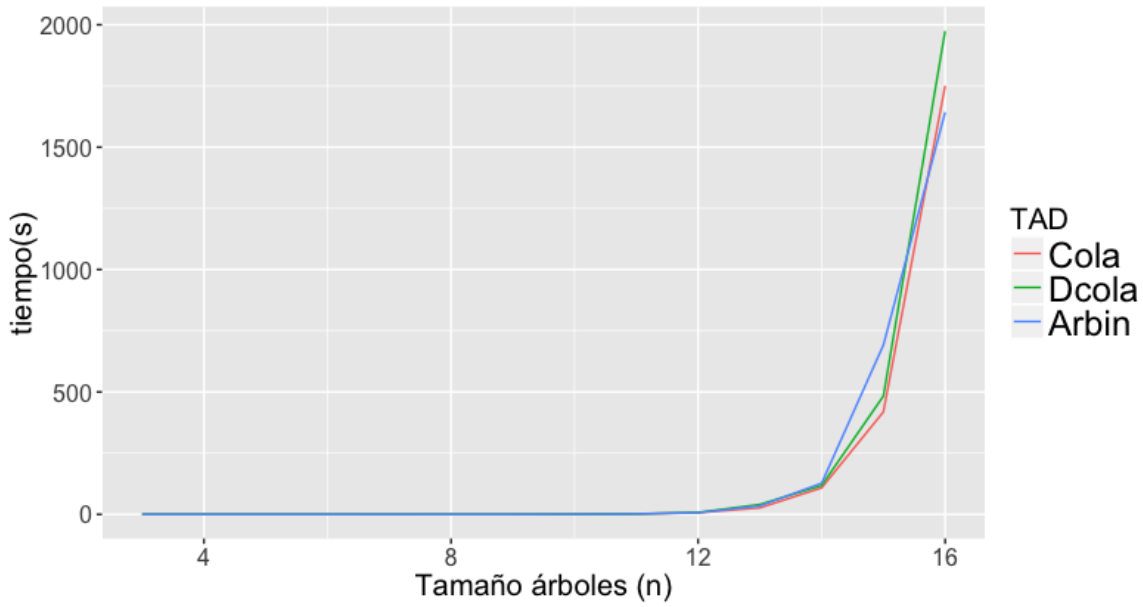


Figura 6.5: Tiempo de ejecución para la generación de árboles

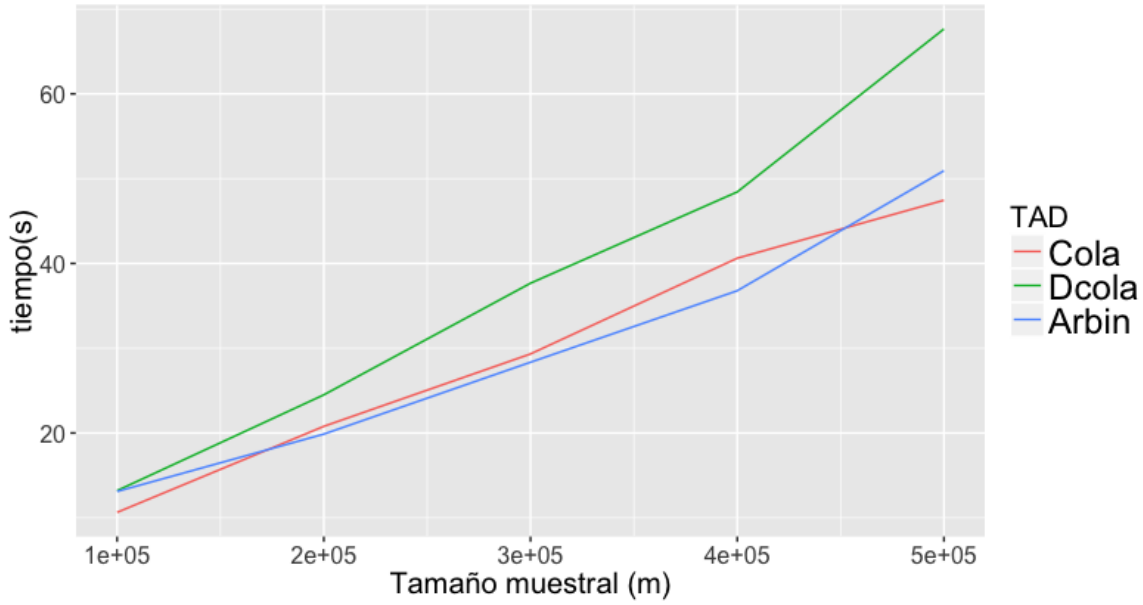


Figura 6.6: Tiempo de ejecución para la generación de muestras

Capítulo 7

Conclusiones

7.1. Discusión

El propósito de esta sección es discutir el trabajo realizado en relación con los objetivos planteados inicialmente. Estos objetivos corresponden primero, a diseñar una metodología para la generación de árboles síntacticos que sirvan para denotar nombres de objetos computacionales arbitrarios. Segundo, diseñar una métodología de muestreo aleatorio sobre el espacio de árboles sintácticos. Por último implementar las metodologías diseñadas para las estructuras de datos más comunes e investigar la complejidad y los límites muestrales de dicha implementación.

Para diseñar la metodología para generar árboles sintácticos fue necesario desarrollar una notación propia que permitiera hablar de árboles y TADS con precisión. Una de las principales contribuciones de este trabajo es haber enriquecido el acervo de nociones alrededor de problema a partir de la introducción o profundización de conceptos como *arbol* $k-ario$, $función\ generadora$, $conformidad$ o *aptitud*. A pesar de haber sido demandante, esta tarea fue indispensable y provechosa en el largo plazo puesto que permitió proponer una solución clara de implementación relativamente sencilla. En la medida en que el diseño de la metodología fue instrumental para desarrollar la solución propuesta, y eventualmente podría serlo para otras soluciones, se considera que el primer objetivo de este trabajo se cumplió.

Para cumplir el segundo objetivo también fue necesario introducir nociones como *representatividad* o *tipo primitivo* $finito$. Estas nociones permitieron desacoplar el proceso de generación de la información necesaria para muestrear del proceso mismo de muestreo. Como en el caso del primer objetivo, en la medida en que el diseño de la metodología fue instrumental para desarrollar la solución propuesta, y eventualmente podría serlo para otras soluciones, se considera que el segundo objetivo de este trabajo se cumplió.

Finalmente, la evidencia experimental recogida durante el proceso de validación sugiere que GNom cumple con los requerimientos funcionales con garantías de calidad. Adicionalmente, tanto la evidencia experimental como el análisis realizado permiten aproximar los requerimientos de tiempo y espacio de GNom en función del tamaño del problema. La generación de objetos simples ($n \leq 15$) en tiempos razonables es posible. Se considera que el tercer y último objetivo de este trabajo también se cumplió.

7.2. Trabajo Futuro

A pesar de que este trabajo constituye un avance en el estado de la investigación, tiene una aplicación práctica limitada. Es probable que una fracción importante del software por validar requiera objetos más complejos de los que se está en capacidad de producir. Considere, por ejemplo, el caso de un software por validar que requiere como objetos de prueba colas de 25 elementos. En este caso habría que generar el conjunto de árboles sintácticos de tamaño 51. Con la solución propuesta, generar este conjunto de árboles para poder muestrear tomaría una cantidad de tiempo prohibitiva.

Reducir la complejidad temporal de la generación de árboles sintácticos debería ser una prioridad del trabajo futuro en esta dirección. Por ejemplo, se podría pensar en recorrer el espacio de paréntesis anidados de manera inteligente de tal forma que se evite visitar los C_n paréntesis con n nodos. También se podría sacar provecho de la naturaleza paralelizable del problema de generación de árboles sintácticos y pensar en adoptar algún esquema que permita construir el conjunto de árboles de manera paralela.

Por último, es importante considerar que este proyecto se ha limitado a trabajar con un subconjunto arbitrario de TADs y por lo tanto la consideración de otras posibilidades constituye una dirección de trabajo promisoria. Por ejemplo, se podría incorporar otros elementos de la Teoría de Tipos Abstractos de Datos como la definición de TADs producto o la definición de TADs por restricción. Otra posibilidad relevante sería limitar el análisis de los problemas planteados a un TAD de interés particular.

Bibliografía

[1] Rodrigo Cardoso. *Verificación y Desarrollo de Programas*. Uniandes, Bogotá, Colombia, 1991.

[2] Donald E. Knuth. *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1997.

[3] Donald E. Knuth. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[4] Donald E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 4: Generating All Trees–History of Combinatorial Generation (Art of Computer Programming)*. Addison-Wesley Professional, 2006.

[5] Diva Martinez. Random generation of tree data structures. Universidad de los Andes, 7 2015.

[6] W. Mendenhall, R.J. Beaver, and B.M. Beaver. *Introduction to Probability and Statistics*. Cengage Learning, 2012.

[7] J. Sanmiguel. *Validación de software mediante métodos estadísticos*. PhD thesis, Universidad de los Andes, Bogota, Colombia, 2013.

Apéndice A

Manual de Uso GNom

A.1. Disponibilidad

GNom se encuentra disponible como un paquete de Python en un repositorio público de GitHub. Para clonar localmente el repositorio se debe ejecutar el comando:

```
$ git clone https://github.com/RamirezAmayaS/GNom.git
```

A.2. Instalación

A.2.1. Dependencias

GNom soporta Python 3.0+. Se recomienda utilizar el manejador de paquetes *pip*8,1 que viene incluido en la distribución estandar de Python 3.5.2. Para instalar las dependencias requeridas se debe ejecutar el siguiente comando desde el directorio raíz de la aplicación:

```
$ pip install -r requirements.txt
```

A.2.2. GNom

GNom no requiere instalación.

A.3. Ejecución

Para iniciar GNom se debe ejecutar el siguiente comando desde el directorio raíz de la aplicación:

```
$ python3 -m generador
```

Las soluciones se pueden encontrar en archivos de texto en la ruta relativa al directorio raíz de la aplicación $\backslash salidas$.

A.4. Pruebas

Para ejecutar las pruebas mencionadas en la sección 6.2.1 se debe ejecutar el siguiente comando desde el directorio raíz de la aplicación:

```
$ pytest tests
```

La solución calculada manualmente para cada una de las pruebas se pueden encontrar en archivos de texto en la ruta relativa al directorio raíz de la aplicación $\backslash test\backslash soluciones\backslash generador\_árboles$,

A.5. Notación

Ocasionalmente GNom le pedirá al usuario que especifique un TAD vía línea de texto. Se deben respetar las convenciones utilizadas a lo largo de este documento para nombrar TADS. Estas convenciones se resumen a continuación:

$$TAD \rightarrow nomtad[parametros]$$

$$nomtad \rightarrow \text{“}Cola\text{”}|\text{“}DCola\text{”}|\text{“}Arbin\text{”}$$

$$parametros \rightarrow parametro, ..., parametro$$

$$parametro \rightarrow TAD|tipoprimitivofinito$$

$$tipoprimitivofinito \rightarrow \text{“}bool\text{”}|\text{“}nat@a\#b@\text{”}|\text{“}int@a\#b@\text{”}$$

Apéndice B

Especificaciones Relevantes

B.1. Máquinas

Las pruebas descritas en la sección 6 se llevaron a cabo en una máquina con las siguientes especificaciones:

- Sistema Operativo : macOs Sierra (10.12)
- Processor : 2.6 GHz Intel Core 15
- Memoria Principal : 8GB 1600 MHz DDR3

Apéndice C

Glosario de Notaciones

C.1. Listas

Una lista sobre un conjunto A es una función de un subconjunto de los números naturales en A. Formalmente:

$$l \quad : D \quad \rightarrow \quad A \quad \text{donde} \quad D \subseteq nat$$

En este trabajo una lista l de tamaño $n \geq 0$ sobre un conjunto A se denota como:

$$l = \langle a_1, ..., a_n \rangle \quad \text{donde} \quad (\quad \forall i \quad | \quad 1 \leq i \leq n \quad : \quad l(i) = a_i \wedge a_i \in A)$$

Sea $l_a = \langle a_1, ..., a_n \rangle$ una lista sobre A, $n \geq 0$ y $l_b = \langle b_1, ..., b_m \rangle$ una lista sobre B, $m \geq 0$. La operación & se define como:

$$l_a \& l_b = \langle a_1, ..., a_n, b_1, ..., b_m \rangle$$

C.2. Tuplas Anidadas

Para $n \geq 0$ una n-tupla anidada tup sobre un conjunto Q es una secuencia finita de tamaño n que cumple alguna de las siguientes condiciones:

- $tup = \varnothing$
- $tup = (tup_1, ..., tup_n)$ donde tup_i es una tupla anidada sobre Q o $tup_i \in Q$, $0 < i \leq n$.

C.2.0.1. Linealización

Dada una tupla anidada *tup*, se define la linealización de *tup* por la lista $lin(tup)$, donde:

- $lin(tup) =<>$ si $tup = \varnothing$.
- $lin(tup) =< tup >$ si $tup \in Q$
- $lin(tup) =< lin(tup_1) > \&, ..., \& < lin(tup_n) >$ si tup $= (tup_1, ..., tup_n)$.