Muestras aleatorias de Tipos Abstractos de Datos

Simón Ramírez Amaya

Departamento de Ingeniería de Sistemas y Computación
Universidad de los Andes

27 de abril de 2017

Agenda

Agenda

Objetivo

El propósito de este trabajo es diseñar e implementar una metodología para la generación de muestras aleatorias de objetos computacionales.

Motivación

La validación estadistica de software se basa en considerar las ejecuciones de un programa como experimentos estadísticos sobre los que se pueden plantear pruebas de hipótesis.

En el contexto de la Programación Orientada a Objetos (OOP) la generación aleatoria de objetos de prueba a partir de una distribución conocida es importante puesto que hace posible el razonamiento probabilístico sobre la corrección de un programa.

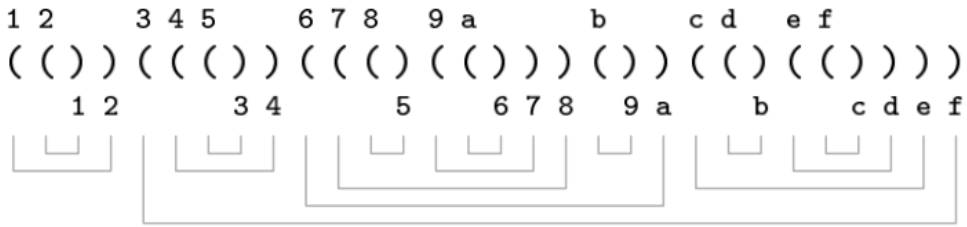
Agenda

Cadenas anidadas

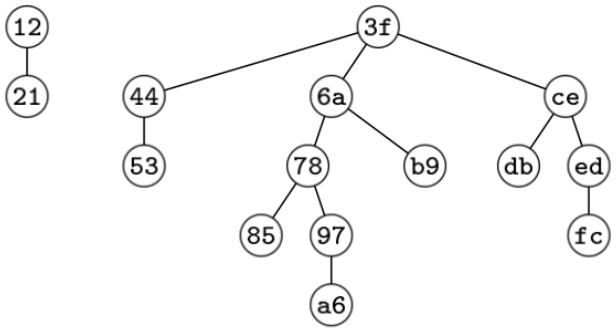
Knuth(2006) estudia la relación básica entre cadenas anidadas y árboles.

Se dice que una cadena $a_1 a_2 ... a_{2n}$ de tamaño n es anidada si y solo consta de n caracteres '(', n caracteres ')' y la k-ésima ocurrencia de '(' precede a la k-ésima ocurrencia de ')', $1 \leq k \leq n$.

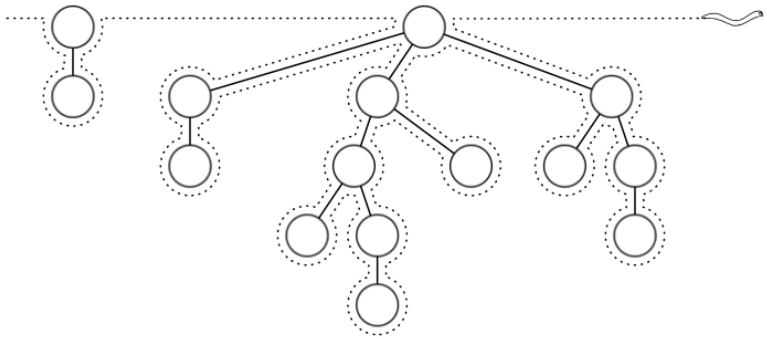
Equivalencia



(a) Cadena anidada

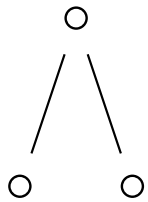


(b) Árbol

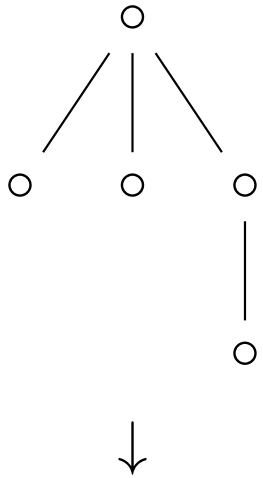


(c) Worm walk

Ejemplos

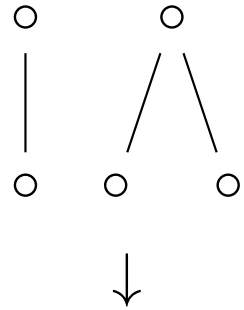


(a) Árbol con raíz (n=3)



(()()(()))

(b) Árbol con raíz (n=5)



(())(()())

(c) Árbol sín raíz (n=5)

Propósitos

Knuth (2006) estudia la generación de cadenas anidadas con dos propósitos:

Generar todas las cadenas anidadas de un tamaño arbitrario.

Generar una cadena anidada aleatoria de tamaño arbitrario.

Algorítmica

Sea A_{pq} la secuencia de cadenas α en orden lexicográfico tales que α contiene p parentésis izquierdos y q paréntesis derechos y $(^{q-p}\alpha$ es una cadena anidada, $0 \leq p \leq q$.

A_{pq} se puede expresar recursivamente como:

$$A_{00} = \epsilon$$

$$A_{pq} =)A_{p(q-1)}, (A_{(p-1)q} \qquad si \qquad 0 \leq p \leq q \neq 0$$

Por supuesto, A_{pq} es vacía si $p > q$ o $p < 0$. Note que A_{nn} es la secuencia de cadenas anidadas de tamaño n.

Algorítmica

Sea C_{pq} el número de elementos de la secuencia A_{pq}. C_{pq} se puede expresar recursivamente como:

$$C_{00} = 1$$

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

Por supuesto, $C_{pq} = 0$ si $p > q$ o $p < 0$. Note que C_{nn} es el número de cadenas anidadas de tamaño n.

Algorítmica

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. Esta secuencia, así como el número de árboles con n nodos, crece como $O(4^n)$.

q \ p	0	1	2	3	4	5	6
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

(a) C_{pq} y los Números de Catalán

Algorítmica

El arreglo triangular de C_{pq} puede pensarse como un grafo dirigido donde existe un arco desde cada nodo pq a su vecino inmediato a la izquierda $(p-1, q)$ y a su vecino inmediato hacia arriba $(p, q-1)$. En este contexto C_{pq} representa también el número de caminos desde el nodo (p, q) hasta el nodo $(0, 0)$.

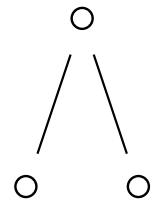
Al realizar un recorrido desde un nodo (p, q) con $p = q$ y representar cada transición al vecino izquierdo con '(' y cada transición al vecino superior con ')', se tiene un mapeo biyectivo entre caminos y cadenas anidadas.

Algorítmica

La figura (a) representa la caminata generadora del árbol y de la cadena anidada en la figura (b).

q \ p	0	1	2	3	4	5	6
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

(a) Caminata



(b) Árbol y cadena

Generando todas los cadenas

Para generar todos los cadenas se utiliza el algoritmo P para la generación lexicografica de cadenas anidadas. Llamaremos al conjunto de cadenas anidadas generado de esta manera P_n.

q \ p	0	1	2	3	4	5	6
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

(a) Primera caminata...

q \ p	0	1	2	3	4	5	6
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

(b) ...Última caminata

Generando una cadena aleatoria

Para generar una cadena anidada de tamaño n se realiza una caminata aleatoria desde el nodo (n, n) al nodo $(0, 0)$. A cada paso se decide si tomar a la izquierda o tomar hacia arriba en función de la proporción de caminos al que pasan por los nodos vecinos.

q \ p	0	1	2	3	4	5	6
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

(a) Caminata aleatoria

Definición

Cardoso (1991) 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 quiere definir un TAD $Y[X]$ donde X es un conjunto finito de TADS o tipos primitivos que hacen las veces de parámetros.

Funciones iniciales

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

Funciones constructoras

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

$$c \quad : \quad Y \quad \times \quad X_c \quad \rightarrow \quad Y$$

donde X_c es un producto cartesiano, posiblemente vacío, de algunos de los TADs parámetro.

Funciones generadoras

Para un TAD Y[X], el conjunto de funcionalidades $G = I \cup C$ se conoce como el conjunto de funciones *generadoras*.

TAD Cola[X]

El mundo de las colas es 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.

Las funcionalidades correspodientes que definen la construcción de nombres en el TAD $Cola[X]$ son:

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

TAD Cola[X]

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 $1, 3, 2$ tiene como denotación -o forma de ser consruida- la siguiente expresión pertenciente al tipo de interés:

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

TAD DCola[X]

Una doble cola (*deque* en inglés) es segundo tipo de estructura lineal en la que es posible insertar elementos por cualquiera de los dos extremos.

El conjunto de funciones *generadoras* del TAD $DCola[X]$ tiene una funcionalidad adicionalidad respecto al conjunto del TAD $Cola[X]$:

- $vac\_dcola : \rightarrow DCola$
- $ins\_der : DCola \times X \rightarrow DCola$
- $ins\_izq : X \times DCola \rightarrow DCola$

TAD DCola[X]

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

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

TAD DCola[X]

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:

$$ins\_der(ins\_der(ins\_der(vac\_dcola, 1), 3), 2)$$
$$ins\_izq(1, ins\_izq(3, ins\_izq(2, vac\_dcola)))$$

TAD Arbin[X]

La construcción inductiva de estructuras no lineales como los árboles binarios también puede abstraerse algebraicamente. 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 un elemento en parejas de árboles mas simples.

Las funcionalidades que definen la construcción de nombres en el TAD $Arbin[X]$ son:

- $vac\_arbin : \rightarrow Arbin$
- $cons : Arbin \times X \times Arbin \rightarrow Arbin$

TAD Arbin[X]

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

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

TAD Arbin[X]

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:

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

Tipos Primitivos

Cardoso (1991) 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.

a. Tipo *booleano*

   Dominio: **bool**

   Constantes: $\{true, false\}$

b. Tipo *natural*

   Dominio: **nat**

   Constantes: $\{0, 1, 2, ...\}$

c. Tipo *entero*

   Dominio: **int**

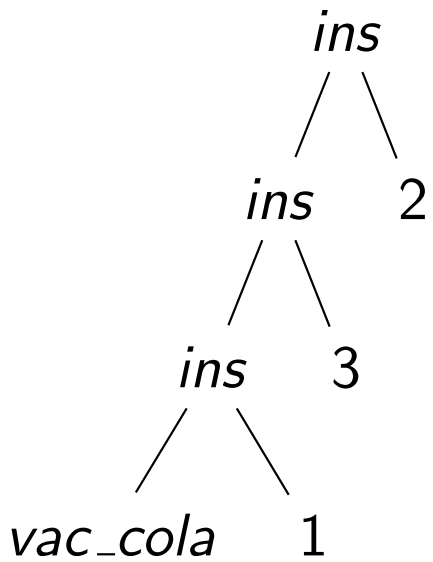
   Constantes: $\{... - 3, -2, -1, 0, 1, 2, 3, ...\}$

Nombres y árboles sintácticos

San Miguel (2013) señala que cualquier objeto computacional es denotable por la expresión que define su construcción inductiva. Dado que estas expresiones hacen parte de un lenguaje formal, para cualquier expresión debe existir un árbol sintáctico que la represente correctamente y que sea igualmente útil para denotar un objeto.

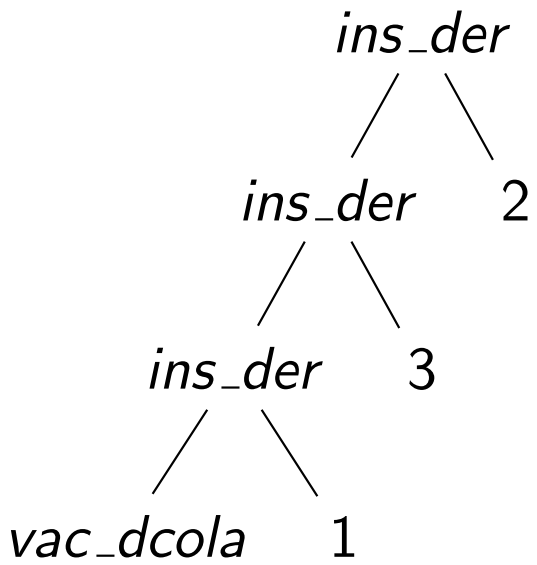
Más adelante se formaliza la equivalencia entre expresiones y árboles sintácticos y su relación con cadenas anidadas. Por el momento se presentan algunos árboles sintácticos que podrían representar los ejemplos de TADs vistos anteriormente.

Ejemplo Cola[X]



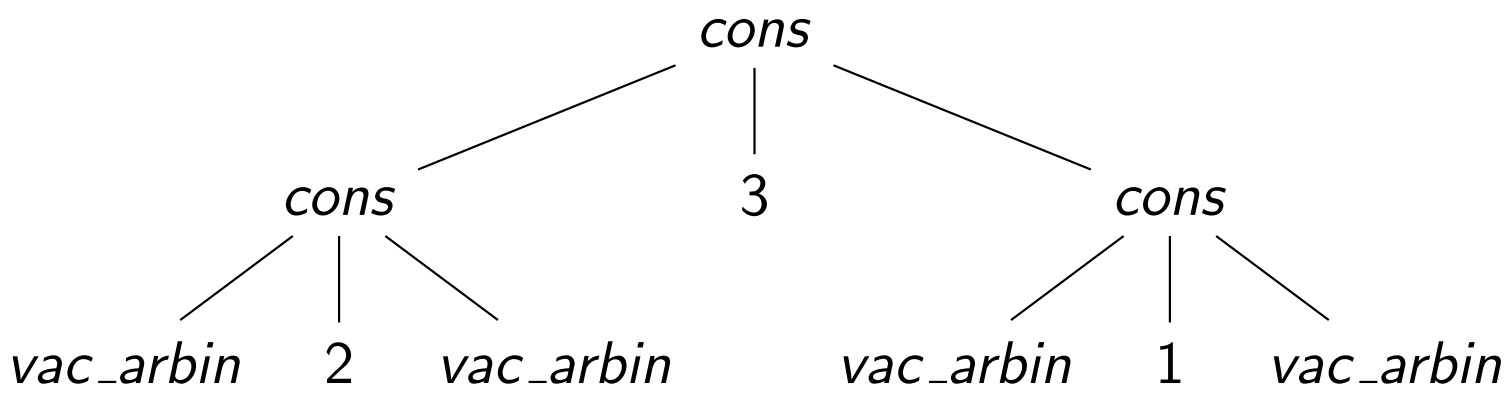
$$ins(ins(ins(vac\_cola, 1), 3), 2)$$

Ejemplo DCola[X]



$$ins\_der(ins\_der(ins\_der(vac\_dcola, 1), 3), 2)$$

Ejemplo Arbin[X]



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

Agenda

Definición

Se pretende dar solución a tres problemas:

1. Dado un TAD $Y[X]$ y un $n > 0$, encontrar todos los árboles sintácticos de n nodos que denotan nombres en Y.
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.
3. Implementar algoritmos para (1) y (2), estimando sus complejidades computacionales.

Alcance

Este trabajo es un estudio exploratorio que se restringe a los tipos de datos descritos anteriormente. En principio, los métodos desarrollados serán generales aunque para las implementaciones solución no se va más allá de manejar estos ejemplos sencillos.

Así mismo, siempre se trabaja 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 mal por memoria para cualquier especificación de un intervalo finito arbitrariamente grande.

Muestra Aleatoria

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 $x \in \Omega$, $P(x) = 1/N$.

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.

Agenda

Árboles k-arios

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

(1) V es la lista vacía de etiquetas de E y se denota $\varnothing$.

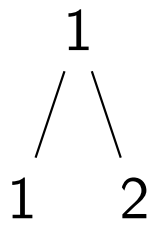
(2) $V = \langle r \rangle \quad \& \quad (\quad \&i \quad | \quad 0 < i \leq k \quad : \quad V_i \quad)$, $r \in E$, V_i es un árbol k-ario sobre E, $0 < i \leq k$.

Árboles k-arios

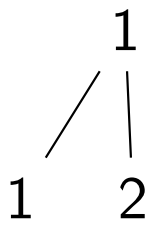
En el caso (1) se dice que el árbol es vacío.

En el caso (2) se define r como la raíz del árbol y se dice que $raiz(V) = r$. También se dice que el *número de hijos* de r es el número de V_i's en la definición que no son vacíos y se denota $grado(r)$. 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, $0 < i \leq k$. Finalmente, el *j-esimo* elemento de $hijos(r)$ se denota como $hijos(r, j)$, $0 < j \leq grado(r)$.

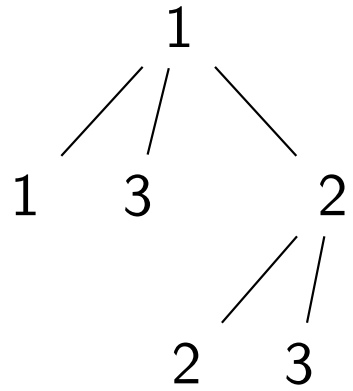
Árboles k-arios - Ejemplos



(a) *Árbol 2-ario*



(b) *Árbol 3-ario*



(c) *Árbol 4-ario*

Árboles k-arios - Ejemplos

Para el ejemplo c:

$$r = 1$$

$$grado(r) = 3$$

$$V_1 = \langle 1 \rangle$$

$$V_2 = \langle 3 \rangle$$

$$V_3 = \varnothing$$

$$V_4 = \langle 2 \rangle \& (\langle 2 \rangle \& \langle 3 \rangle \& \varnothing \& \varnothing)$$

$$hijos(r) = \langle \langle 1 \rangle, \langle \langle 3 \rangle, \langle 2 \rangle \& (\langle 2 \rangle \& \langle 3 \rangle \& \varnothing \& \varnothing) \rangle$$

Árboles k-arios - Notación

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

(1) Si V es el árbol vacío entonces $d(V) =$ "".

(2) 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-esimo subárbol de r.

Árboles k-arios - Notación

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

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

El *árbol 3-ario* en el ejemplo (b) se denota mediante la expresión:

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

El *árbol 4-ario* en el ejemplo (c) se denota mediante la expresión:

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

Árboles k-arios - 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.

Árboles k-arios - Tamaño

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

(1) $tam(\varnothing) = 0$.

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

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

Árboles k-arios - 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:

(1) $preorden(\varnothing) = \langle\rangle$.

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

Árboles k-arios - Preorden

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

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

TPFs

Un tipo primitivo finito (TPF) es un tipo primitivo cuyo dominio es un subconjunto finito de los tipos primitivos definidos anteriormente.

Se puede pensar en definir un TPF a partir de la restricción de un tipo primitivo no finito. Son de interés para este proyecto los TPFs definidos a partir de la consideración de intervalos cerrados de constantes de los tipos *nat* e *int*. Los dominios de estos tipos finitos se denotan con las expresiones

$$nat@a\#b@ \text{ donde } a, b \in nat, a \leq b \text{ y } |nat@a\#b@| = b - a + 1$$

$$int@c\#d@ \text{ donde } c, d \in int, c \leq d \text{ y } |nat@a\#b@| = d - c + 1$$

.

TPFs - Funciones generadoras

Es conveniente pensar en como describir un tipo primitivo finito utilizando la notación para la construcción inductiva de nombres de TADs.

La "construccion" 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:

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

TPFs - Funciones generadoras

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:

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

TADs - Funciones generadoras

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

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

TADs - Funciones generadoras

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.

Por ejemplo, la función generadora *cons* del TAD $Arbin[X]$ tiene $aridad(cons) = 3$, $Z(cons, 1) = Z(cons, 3) = Arbin[X]$ y $Z(cons, 2) = X$.

Conformidad

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:

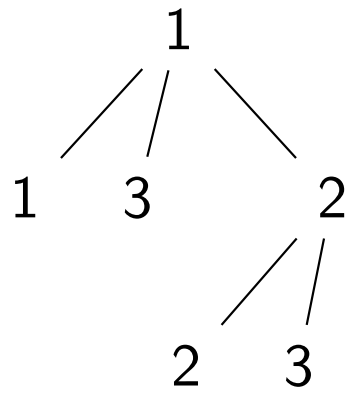
$$con(r, f) \equiv grado(r) = aridad(f)$$

Conformidad - Ejemplo

La raíz r del árbol 4-ario ejemplo 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.



Listas de etiquetas sintácticas

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 $L(V, Y[X])$.

Listas de etiquetas sintácticas

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

Listas de etiquetas sintácticas

Este conjunto de tuplas anidadas se define recursivamente:

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

Listas de etiquetas sintácticas

$L(V, Y[X])$ es simplemente el conjunto resultante de linealizar cada una de las tuplas en $T(V, Y[X])$.

Si $T(r, Y[X]) = \{u_1, ..., u_p\}$, $p \geq 0$, entonces:

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

Aptitud

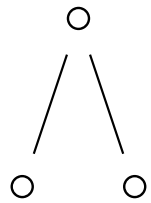
Finalmente, el conjunto de *árboles sintácticos representativos de nombres* resultante de recorrer V dejando las etiquetas de las listas en $L(V, Y[X])$ se denota $A(V, Y[X])$.

Se dice que V es un *árbol apto respecto a* $Y[X]$ si

$$A(V, Y[X]) \neq \varnothing$$

Ejemplo

Sea V el siguiente árbol ejemplo:



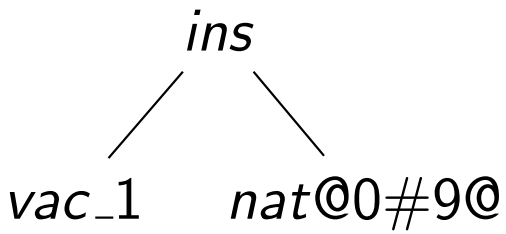
Sea también $ejemplo[nat@0\#9@]$ un TAD con las siguientes funciones generadoras:

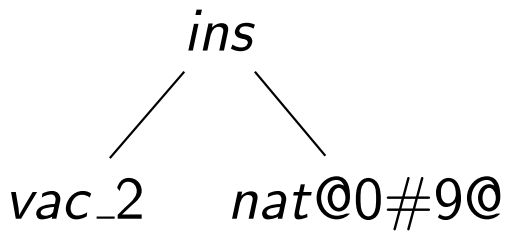
- $vac\_1 : \rightarrow ejemplo$
- $vac\_2 : \rightarrow ejemplo$
- $ins : ejemplo \times nat@0\#9@ \rightarrow ejemplo$

En este caso:

$$T(\cdot) = \{(ins, (vac\_1, nat@0\#9@)), (ins, (vac\_2, nat@0\#9@))\}$$

$$L(\cdot) = \{\langle ins, vac\_1, nat@0\#9@\rangle, \langle ins, vac\_2, nat@0\#9@\rangle\}$$





Representatividad

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

En particular se puede calcular para cada uno de los árboles generados por $P_n, n \geq 0$. La unión sobre todo dos los conjuntos de árboles generados de esta manera se denotara $A(n, Y[X])$. Este conjunto contiene todos los árboles sintácticos de n nodos representativos de nombres en $Y[X]$. Formalmente:

$$A(n, Y[X]) = \left(\bigcup V \mid V \in P(n) : A(V, Y[X]) \right)$$

Representatividad

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 denota $A(n, Y[X_0])$.

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:

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

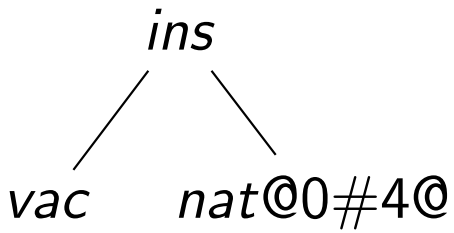
Distribución representativa

La distribución de probabilidad en la que la probabilidad de un árbol $V \in A(n, Y[X])$ es la ponderación de su representatividad es la distribución de probabilidad representativa y se denota $\Phi(t)$.
Formalmente: Formalmente:

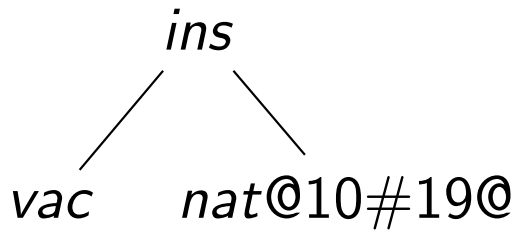
$$\phi(V) = \frac{rep(V)}{\left(\sum V \mid V \in A(n, Y[X]) : rep(V) \right)}$$

Ejemplos

Para $n = 3$ y $Y[X] = Cola[nat@0\#4@, nat@10\#19@]$



(a) $rep(V) = 5,\ \Phi(t) = 1/3$



(b) $rep(V) = 5,\ \Phi(t) = 2/3$

Agenda

Algorítmica (1)

Se proponen los siguientes esquemas de solución para los problemas (1) y (2):

1. Generar P_n
   Para cada árbol $V \in P_n$ generado:
   1) Generar el conjunto de tuplas anidadas de etiquetas $T(raiz(V), Y[X])$.
   2) Para cada tupla anidada *tup* no vacía generada:
      1) Linealizar *tup* a una secuencia s.
      2) Etiquetar V con s a partir de un recorrido en preorden.
      3) Persistir $d(V)$.

Algorítmica (2)

2. 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.

Algorítmica

La solución propuesta permite desacoplar la generación de árboles síntácticos representativos del muestreo aleatorio de nombres.

En términos prácticos esto significa no es necesario recalcular el conjunto de árboles sintácticos representativos para generar múltiples muestras aleatorias de nombres.

Complejidad (1)

Para dar solución al problema (1) se deben generar C_n cadenas anidadas utilizando el algoritmo P. La traducción de cada cadena anidada a un árbol cuesta una iteración sobre la cadena anidada. Es decir, esta traducción cuesta exactamente $2n$ operaciones.

Adicionalmente, si existen $q > 0$ funciones generadoras conformes por nodo se deben generar q^n árboles adicionales por cada árbol en P_n. La complejidad temporal resultante es

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

.

Complejidad (2)

Sea n, el tamaño de los nombres a generar, a el tamaño del conjunto*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 requierea lo sumo n reemplazos de etiquetas de tipos primitivos finitos por elementos en su dominio. Es decir que:

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

GNom

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

GNom

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

La aplicación hace uso de las librerías *matplotlib*, *numpy* y del framework para pruebas *pytest*.

Validación (1)

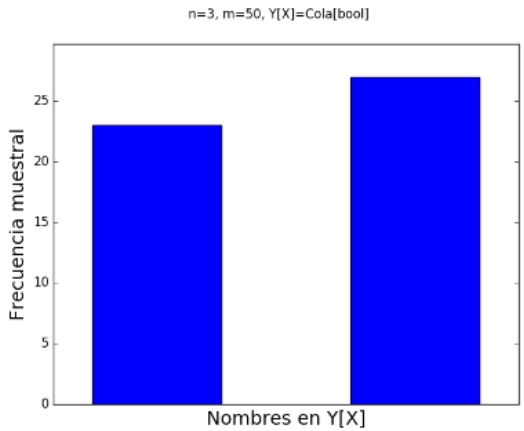
Para validar la corrección de la solución implementada para el problema de generar árboles representativos se le pidió a un experto que calculara manualmente la solución a 15 instancias del problema de su elección. En todos los casos se verificó positivamente que la solución retornada coincidiera con la solución calculada.

Validación (2)

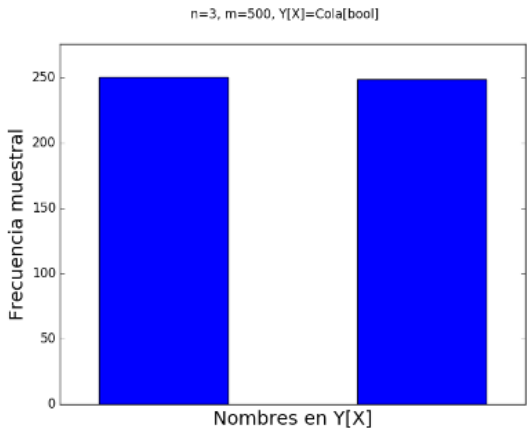
Para validar la correción de la solución implementada para el problema de generar muestras aleatorias de nombres se verifica gráficamente que la distribución empírica de las muestras converja a la distribución uniforme sobre el espacio de nombres a medida que el tamaño muestral aumenta.

Se considera un conjunto de instancias heterogeneas del problema:

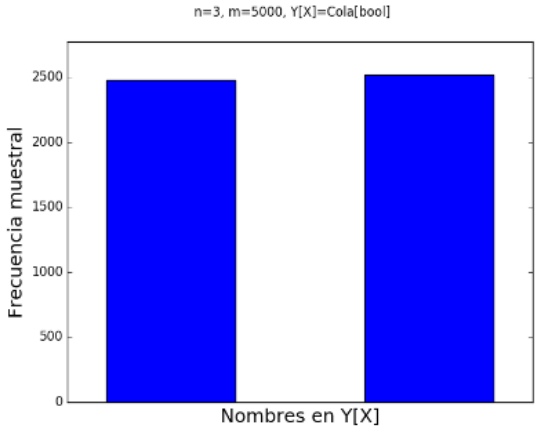
1. $n = 3,\ Y[X] = Cola[bool],\ m = \{5e1, 5e2, 5e3, 5e4\}$
2. $n = 3,\ Y[X] = Cola[nat@0\#9@],\ m = \{1e2, 1e3, 1e4, 1e5\}$
3. $n = 5,\ Y[X] = DCola[bool, int@-3\#3@]\ m = \{2e3, ..., 1e6\}$
4. $n = 7,\ Y[X] = Arbin[nat@0\#19@],\ m = \{1e4, 1e5, 1e5, 3e6\}$



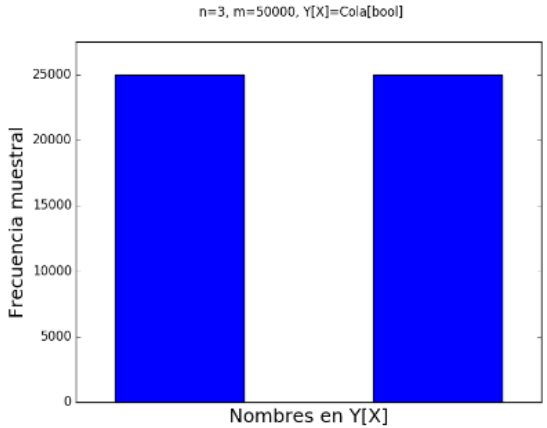
(a) Distribución muestral ($m = 5e1$)



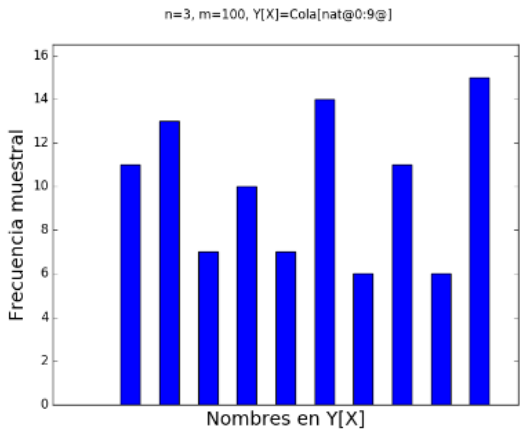
(b) Distribución muestral ($m = 5e2$)



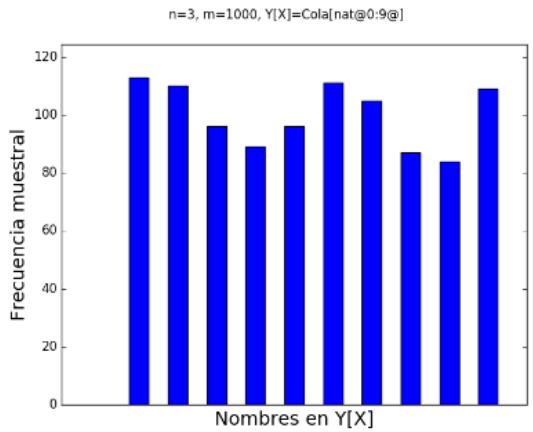
(c) Distribución muestral ($m = 5e3$)



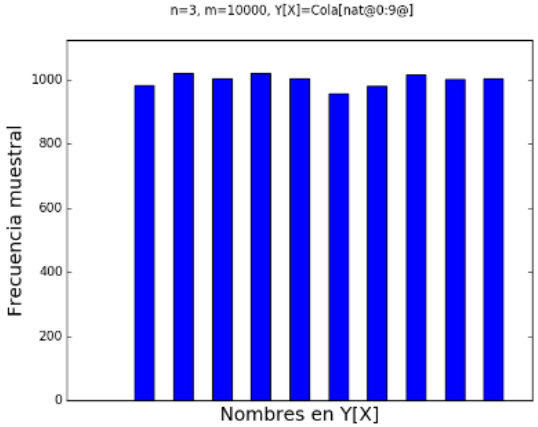
(d) Distribución muestral (m=$m = 5e3$)



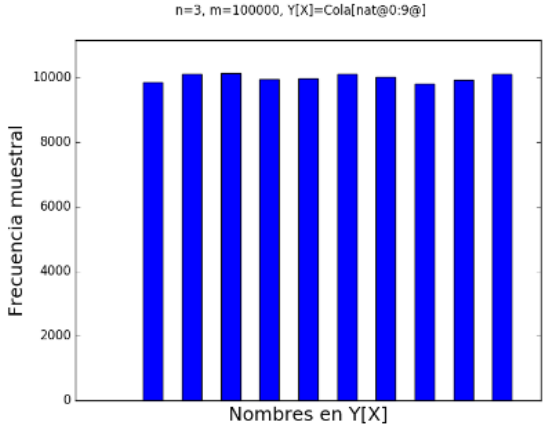
(a) Distribución muestral ($m = 1e2$)



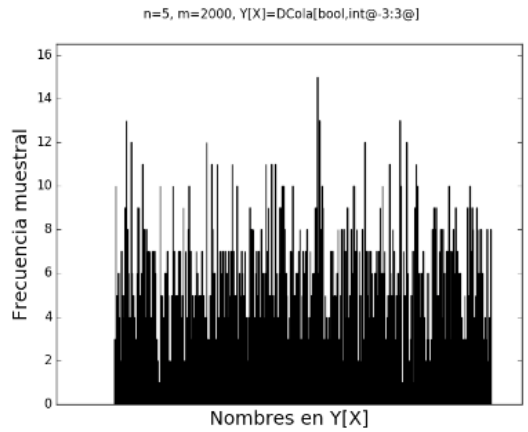
(b) Distribución muestral ($m = 1e3$)



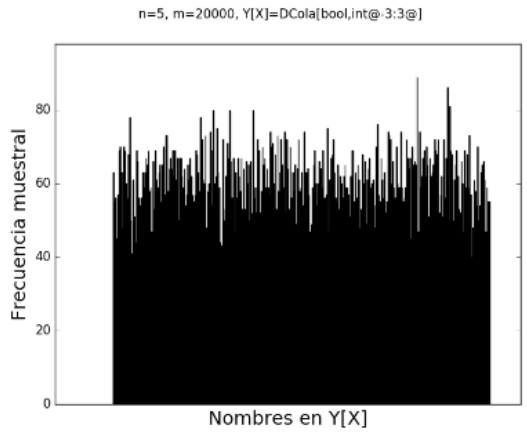
(c) Distribución muestral ($m = 1e4$)



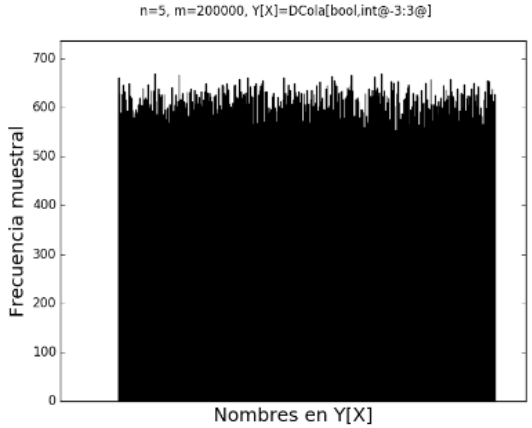
(d) Distribución muestral ($m = 1e5$)



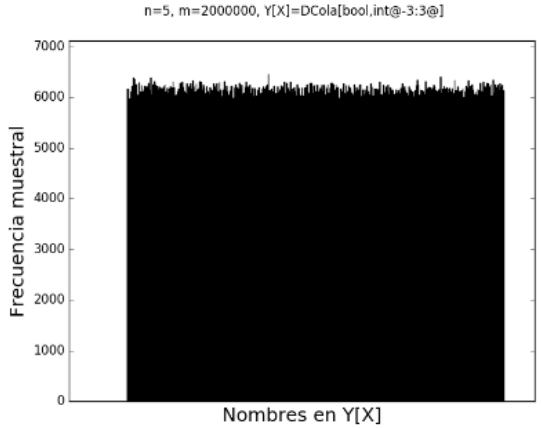
(a) Distribución muestral ($m = 2e3$)



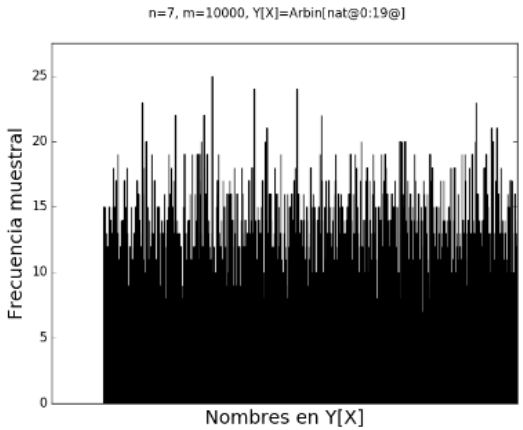
(b) Distribución muestral ($m = 2e4$)



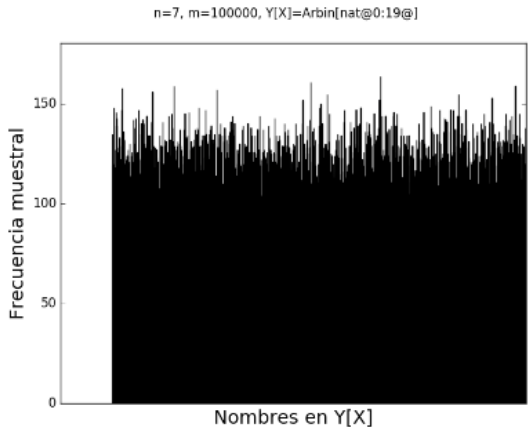
(c) Distribución muestral (m=$m = 2e5$)



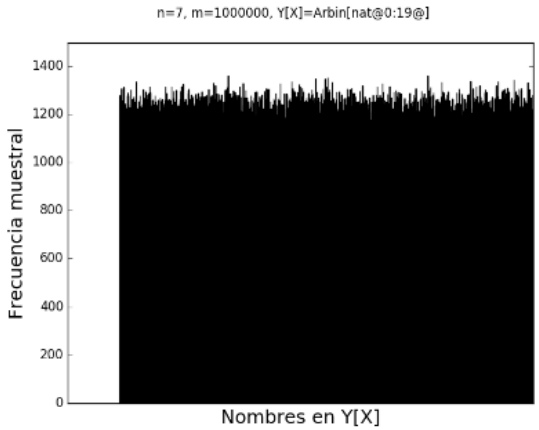
(d) Distribución muestral ($m = 2e6$)



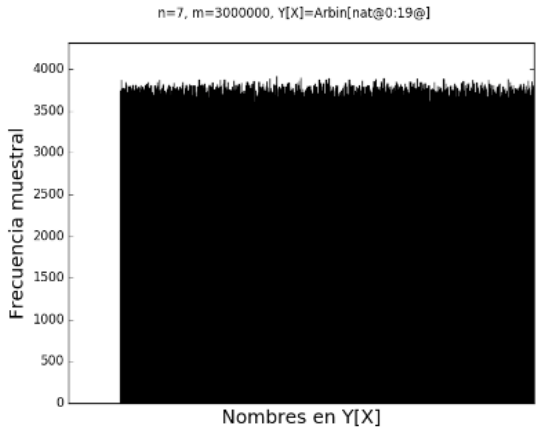
(a) Distribución muestral ($m = 1e4$)



(b) Distribución muestral ($m = 1e5$)

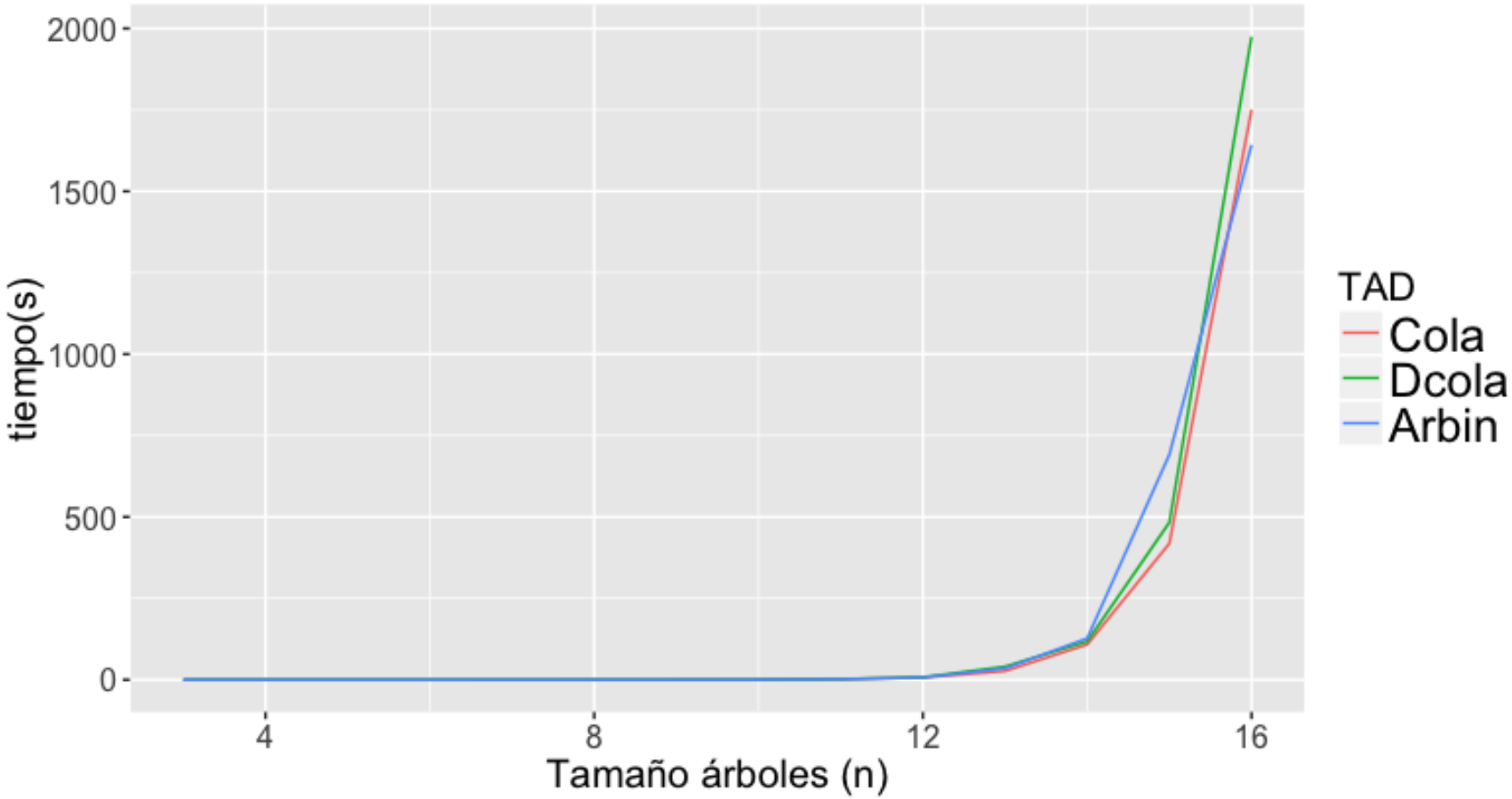


(c) Distribución muestral (m=$m = 1e6$)

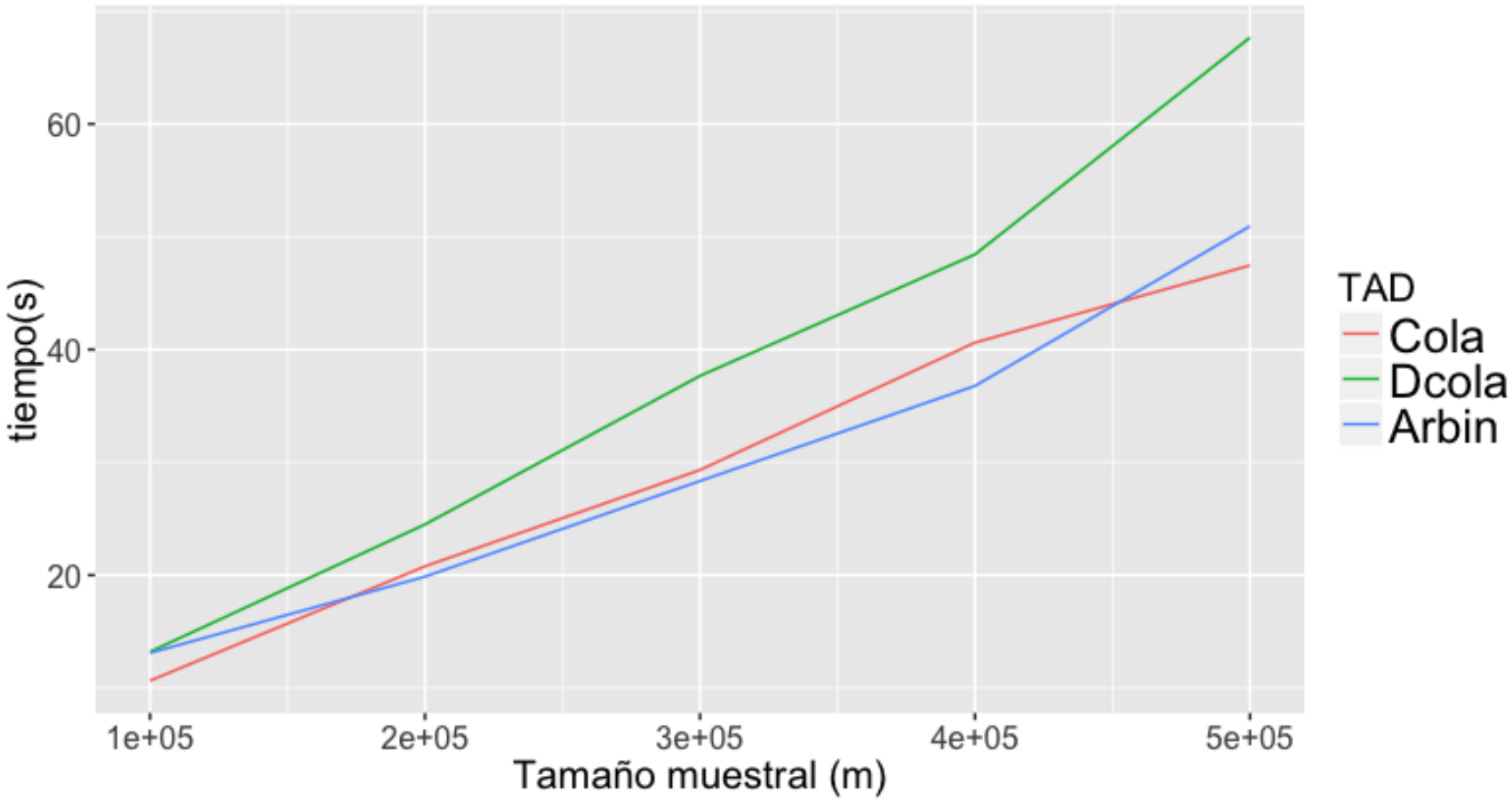


(d) Distribución muestral ($m = 3e6$)

Complejidad 1



Complejidad 2



Agenda

A pesar de que este trabajo constituye un avance en el estado de la investigación, tiene una aplicación práctica limitada.

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.

Por último, es importante considerar que este proyecto se ha limitado a trabajar con un subconjunto arbitrario de TADs.