Tesis de Licenciatura en Ciencias de la Computación

Acelerando ReMo a través de la construcción de permutaciones usando autómatas

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

Lorena Bourg

Directores

Dr. Marcelo Frías
Lic. Carlos López Pombo

Resumen

La presente tesis se realiza en el contexto del desarrollo de la herramienta ReMo. ReMo es una herramienta para el análisis de especificaciones relacionales. Como se describirá en el transcurso de esta tesis, dicho análisis requiere la generación de un conjunto de permutaciones. La contribución de esta tesis consiste en la propuesta y análisis experimental de una estructura de datos más eficiente que la utilizada en la actualidad para el almacenamiento y generación de permutaciones. Concluiremos que esta propuesta, basada en autómatas finitos, mejora a la actual (basada en tries) de la siguiente manera:

1. *El consumo de memoria se reduce a un tercio del actual utilizando tries.*

2. *El tiempo de generación de permutaciones válidas (dado un conjunto de permutaciones a evitar) se reduce a menos de la mitad.*

Se presentarán, también, distintas estrategias que pueden permitir reducir el tamaño del autómata finito.

Índice

1. Introducción

Las estadísticas muestran que, en el marco del proceso de desarrollo de software, los defectos se vuelven más graves y su corrección más costosa mientras más avanzado esté el proceso de construcción del mismo al momento de la detección ([GJM02]). De esto surge la necesidad, antes de pasar a futuras etapas del mismo, de comprobar que la especificación realizada del sistema cumpla con las expectativas que se tienen sobre la misma; sería conveniente garantizar que, a partir de una descripción formal planteada, se derive una serie de propiedades deseadas para la aplicación. Al plantear esta serie de propiedades se desea ver que con la especificación se está modelando el problema a resolver sin que ninguna propiedad deseada no sea consecuencia de las reglas básicas planteadas. Una descripción formal puede consistir en un conjunto de predicados o axiomas que imponen restricciones sobre el sistema que está siendo modelado. Analizar el comportamiento de dicho sistema consiste en la verificación de si una cierta propiedad que predica sobre este comportamiento se satisface o no. Luego, si estas propiedades son expresadas de manera formal, se puede representar el problema de verificación de propiedades de la siguiente forma:

Dados $\mathbf{E}$ una especificación, $\mathbf{P}$ un conjunto de propiedades y $\models$ la relación de *satisfacción*;

$$¿\, \mathbf{E} \models \mathbf{P}?$$

La mayor parte de los investigadores del área canalizan sus esfuerzos en el desarrollo de métodos algorítmicos para resolver este problema; este enfoque es denominado usualmente model checking ([McM94]) y consiste en la obtención de un algoritmo que responde *sí* o *no*, dependiendo de si la especificación del sistema que está siendo analizado satisface o no un conjunto de requerimientos dado.

Evidentemente, no es deseable que toda propiedad esperada sea planteada como un axioma; podríamos caer, muy probablemente, en especificaciones inconsistentes.

El primer paso para poder obtener algoritmos como los mencionados anteriormente es fijar el lenguaje que nos permitirá expresar de manera formal los requerimientos. El enfoque más utilizado para esto, y que ha demostrado muy buenos resultados, es el uso de lógicas para expresar las fórmulas, considerando la relación de satisfacción como la característica de "*ser modelo de*". Esto se ver por ejemplo en el análisis de sistemas reactivos [MP92], de sistemas concurrentes [S97], y también en la verificación de hardware [G92]. En particular, en el caso de sistemas reactivos y/o concurrentes, la especificación formal y el uso de métodos formales en general permiten obtener resultados confiables en áreas donde la complejidad de los problemas a tratar escapa lo analizable con métodos no formales.

En el marco del desarrollo de ReMo, una herramienta de model checking basada en el análisis de monotonía, ([GS05]) el lenguaje elegido fue el de las fork álgebras ([Fri02]). La elección de este lenguaje de especificación está fundada en que la mencionada herramienta se desarrolló con el objetivo de ser incorporada dentro de la herramienta Ar$_g$entum, actualmente en construcción.

Ar$_g$entum se basa en la unificación de distintas lógicas mediante la traducción de teorías expresadas en ellas a especificaciones relacionales, permitiendo así modelar distintos aspectos de un sistema en el lenguaje más conveniente para cada uno de ellos y dando la posibilidad de verificar propiedades sobre el sistema en su totalidad.

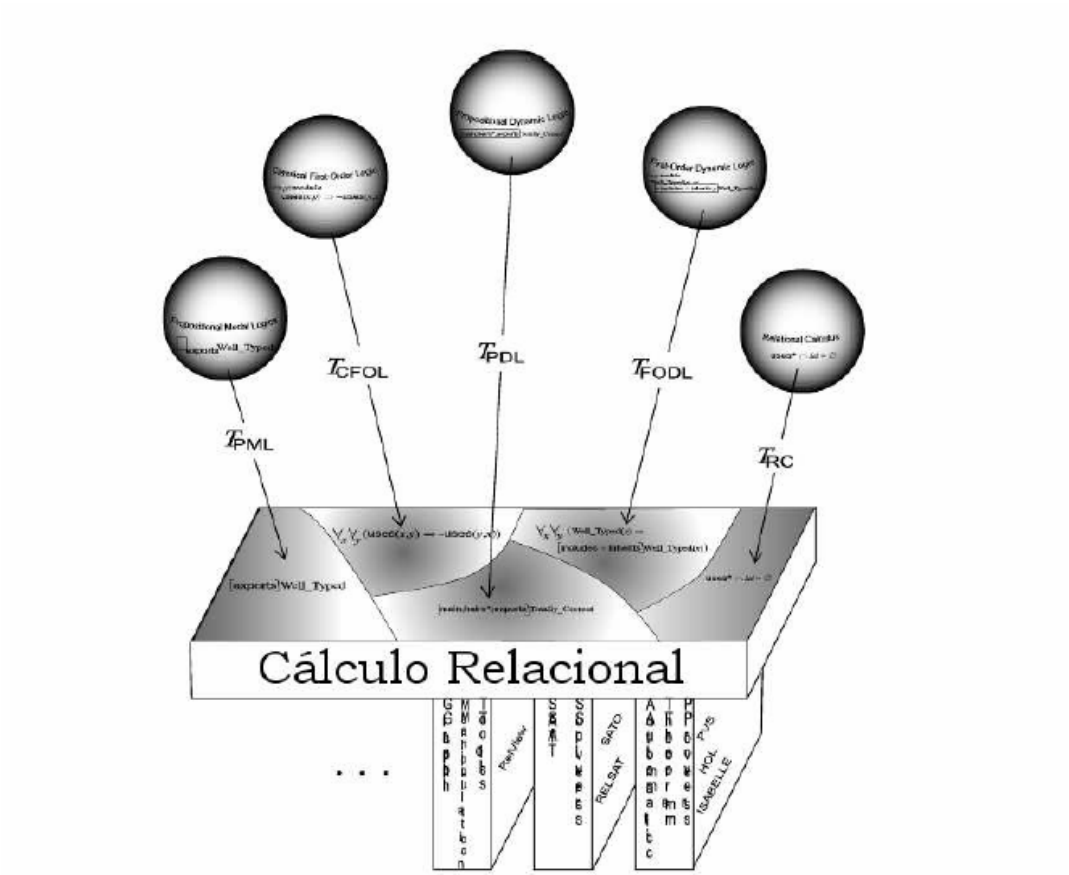


Figura 1.1: Un esquema de la herramienta Ar$_g$entum

Las esferas en la parte superior de la figura representan especificaciones de los distintos aspectos del sistema escritos en diferentes lenguajes, las flechas que se originan en las esferas mapean dichas especificaciones a una única especificación relacional (ubicadas en la caja horizontal). Esta especificación homogénea puede ser analizada utilizando herramientas (los prismas debajo de la caja mencionada) que pueden acoplarse a Ar$_g$entum. El sistema desarrollado en [GS05] constituye una de esas herramientas.

Tal como hemos enunciado el problema, éste resulta irresoluble en el caso general: dada una serie de axiomas encontrar una demostración o un contraejemplo para una propiedad planteada. Dada esta limitación lógico-matemática inherente surgen distintas alternativas de tipos de herramientas a aplicar a esta clase de problemas. Una de las opciones consiste en verificar las

propiedades deseadas dentro de un dominio acotado. Si efectivamente podemos acotar el dominio real de la aplicación, a todo efecto práctico da lo mismo tener una demostración lógica de una propiedad para el caso general que haber verificado el cumplimiento de la misma para todos los elementos de dicho dominio acotado.

Ahora bien, hallar dicha cota no resulta siempre posible, o dicha cota puede ser muy grande. Sin embargo tener la garantía del cumplimiento de la propiedad para dominios de un cierto rango habla a favor de la propiedad. O, visto de otra manera, estamos realizando una suerte de testing de nuestra especificación: podemos garantizar que para los casos probados (el dominio, en el caso de una especificación) no se encuentran fallas; aunque eso no impida que puedan encontrarse contraejemplos para otros dominios.

Como resultado de las investigaciones presentadas en [GS05] cuyo propósito es encontrar contraejemplos a propiedades, se construyó una aplicación (ReMo). Más precisamente, dada una especificación escrita en un lenguaje (basado en relaciones binarias), se intenta buscar modelos de la misma (restringidos a un cierto tamaño de los dominios definidos) que no satisfagan las propiedades buscadas.

Una especificación consiste de los siguientes componentes:

- Un conjunto de declaraciones de tipos elementales (o dominios relacionales) para los cuales se indicará un rango para su cardinalidad.
- Relaciones binarias definidas entre estos tipos.
- Un conjunto de axiomas.
- Una o más propiedades a verificar.

Se entiende como modelo a cualquier instanciación de las relaciones declaradas que satisfaga todos los axiomas; de la misma manera llamaremos contraejemplo de una propiedad a un modelo en el cual la misma es falsa.

Ahora bien, resolver el problema planteado (encontrar algún contraejemplo de una propiedad) en su forma más general es básicamente el problema de la satisfactibilidad (SAT) [GJ79] para el cual no se conocen algoritmos polinomiales. Dado que nos interesa encontrar contraejemplos (un modelo que no cumple alguna propiedad) una manera de lograrlo es recorrer el espacio de las relaciones binarias que queda definido de acuerdo a la especificación. Este espacio tiene un tamaño exponencial en función del tamaño de los dominios; como simple ejemplo pensemos en una especificación donde se define una sola relación $R \subseteq A \times A$, donde A es un conjunto arbitrario. Para ella existen $2^{|A|.|A|}$ instancias posibles ya que cada uno de los pares del producto cartesiano $A \times A$ puede intervenir o no en una instancia de R. Si introdujéramos otra relación más (S, también incluida en $A \times A$) la cantidad de posibles instanciaciones a considerar asciende a $2^{2.|A|.|A|}$, ya que han de explorarse todas las combinaciones posibles de instancias de R y S.

Evidentemente, el tamaño del espacio de relaciones así definido hace que resulte impráctico recorrerlo por completo, aún para dominios con cardinales no muy altos; por lo que se han desarrollado técnicas que permiten reducir drásticamente el espacio de búsqueda.

Una de esas estrategias, implementada en [GS05], aprovecha la información de monotonía de las variables relacionales en los casos en los cuales eso es posible para evitar generar asignaciones de las variables que se sabe que no producirán nuevos resultados.

Otra estrategia que se está implementando en [M??], se basa en el método de eliminación de isomorfismos descripto por Daniel Jackson, Somesh Jha y Craig Damon en [JJD98] como estrategia general de reducción del espacio de búsqueda que puede utilizarse incluso en los casos en los cuales no hay información de monotonía disponible. Es en el marco de esta estrategia que se desarrolla la presente tesis.

2. Álgebra relacional

Dada una relación binaria X sobre un conjunto A y dos elementos $a,b\in A$, denotaremos que a y b están relacionados a través de X como $(a,b)\in X$ o aXb, indistintamente.

Definición 2.1: Sea E una relación binaria sobre un conjunto A y R un conjunto de relaciones binarias tales que:

1. $\bigcup R\subseteq E$
2. Id (la relación identidad sobre el conjunto A), $\emptyset$ (la relación binaria vacía) y E pertenecen a R.
3. R es cerrado con respecto a la unión $(\cup)$, intersección $(\cap)$ y complemento relativo a E $(^{-})$.
4. R es cerrado ante la composición relacional (denotada como $;$) y la conversa o transpuesta (denotada como $^{\smile}$). Estas dos operaciones se definen como:

$$X;Y=\{(a,b):\exists c(aXc\wedge cYb)\}$$
$$\breve{X}=\{(a,b):bXa\}$$

Entonces, la estructura $\langle R,\cup,\cap,^{-},\emptyset,E,;,Id,^{\smile}\rangle$ se denomina un álgebra de relaciones binarias.

Definición 2.2: Notar que, según la *Definición 2.1* cada álgebra de relaciones binarias A contiene un conjunto A en el cual las relaciones binarias están definidas. Este conjunto será llamado la base de A y se denotará B_A.

Definición 2.3: Un álgebra de relaciones binarias es completa si su universo es de la forma $P(U\times U)$ para algún conjunto U y es cuadrada si la relación más grande es de la forma $U\times U$.

Se deduce de la *Definición 2.3* que toda álgebra de relaciones binarias completa es cuadrada. También, un álgebra de relaciones binarias cuadrada cuya relación más grande es $U\times U$, es una subálgebra del álgebra de relaciones binarias completa con universo $P(U\times U)$.

En 1941 Alfred Tarski introdujo la teoría elemental de relaciones binarias (ETBR) [T41] como una formalización lógica de las álgebras de relaciones binarias. La ETBR es una teoría formal en la que están presentes dos tipos de variables. El conjunto $IndVar = \{v_1, v_2, v_3, ...\}$ contiene las denominadas *variables individuales* y el conjunto $RelVar = \{R, S, T, ...\}$ contiene las *variables relacionales*. Si agregamos las *constantes relacionales* $0,1$ y $1'$ a las variables relacionales y clausuramos este conjunto por los operadores unarios $\bar{}$ y $\breve{}$ y los binarios $;, \cdot$ y $+$, obtenemos el conjunto de las *designaciones relacionales*. Ejemplos de estos objetos con $\breve{R}$ (léase 'la conversa de R') y $R;S$ (léase 'el producto relativo de R y S'). Las fórmulas atómicas son expresiones de la forma xRy (donde x, y son variables individuales arbitrarias y R es una variable relacional arbitraria) o $R = S$ (con R y S designaciones relacionales arbitrarias). De las fórmulas atómicas obtenemos fórmulas compuestas clausurando las primeras por los operadores lógicos unarios $\neg, \forall x, \forall y, ..., \exists y...$ ($x, y...$ variables individuales) y los operadores lógicos binarios $\vee, \wedge, \Rightarrow$ y $\Leftrightarrow$. Vamos a tomar un conjunto de axiomas lógicos y reglas de inferencia estándar para la teoría (ver [E72]). Se eligen como los axiomas que describen la semántica de los símbolos relacionales $0,1,1', \bar{}, \breve{}, +, \cdot$ y $;$ a las siguientes afirmaciones en las cuales x, y, z son variables individuales arbitrarias y R, S, T son designaciones relacionales arbitrarias.

$\forall x \forall y (x 1 y)$ (definición de universal)

$\forall x \forall y (\neg x 0 y)$ (definición de vacía)

$\forall x (x 1' x)$ (reflexividad de la identidad)

$\forall x \forall y \forall z ((x R y \wedge y 1' z) \Rightarrow xRz)$ (la identidad es una congruencia)

$\forall x \forall y (x \bar{R} y \Leftrightarrow \neg xRy)$ (definición del complemento)

$\forall x \forall y (x \breve{R} y \Leftrightarrow yRx)$ (definición de la conversa)

$\forall x \forall y (x R + S y \Leftrightarrow xRy \vee xSy)$ (definición de la unión)

$\forall x \forall y (x R \cdot S y \Leftrightarrow xRy \wedge xSy)$ (definición de la intersección)

$\forall x \forall y (x R ; S y \Leftrightarrow \exists z (xRz \wedge zSy))$ (definición del producto relativo o composición)

$R = S \Leftrightarrow \forall x \forall y (x R y \Leftrightarrow xSy)$ (definición de la igualdad)

A partir de la teoría elemental de relaciones binarias, Tarski introdujo el *cálculo relacional* (CR) [T41]. El mismo está definido como una restricción de la ETBR. Las fórmulas del cálculo relacional son aquellas fórmulas de la ETBR que no contienen ocurrencias de variables individuales. Como axiomas del cálculo relacional Tarski eligió un subconjunto de las fórmulas sin variables individuales válidas en la ETBR. Las fórmulas que Tarski eligió como axiomas (además de un conjunto de axiomas para los conectivos lógicos) son las siguientes:

1. $(R = S \wedge R = T) \Rightarrow S = T$
2. $R = S \Rightarrow (R + T = S + T \wedge R \cdot T = S \cdot T)$
3. $R + S = S + R \wedge S \cdot R = R \cdot S$
4. $(R + S) \cdot T = (R \cdot T) + (S \cdot T) \wedge (R \cdot S) + T = (R + T) \cdot (S + T)$
5. $R + 0 = R \wedge R \cdot 1 = R$

6. $R + \overline{R} = 1 \wedge R \cdot \overline{R} = 0$
7. $\overline{1} = 0$
8. $\breve{\breve{R}} = R$
9. $(R;S)^{\smile} = \breve{S};\breve{R}$
10. $(R;S);T = R;(S;T)$
11. $R;1' = R$
12. $(R;S)\cdot T = 0 \Rightarrow (S;T)\cdot \breve{R} = 0$
13. $R;1 = 1 \vee 1;\overline{R} = 1$

Los axiomas (1)-(7) son una axiomatización para álgebras Booleanas, los axiomas (8)-(12) describen el comportamiento de los operadores relacionales.

Definición 2.4: Un álgebra relacional es un álgebra $\langle A,+,\cdot,\overline{\ },0,1,;,1',{}^{\smile}\rangle$ donde $+,\cdot$ y $;$ son operaciones binarias, $\overline{\ }$ y ${}^{\smile}$ son unarias y $0,1$ y $1'$ son elementos distinguidos. Más aún, el reducto $\langle A,+,\cdot,\overline{\ },0,1\rangle$ es un álgebra booleana, y las siguientes identidades son satisfechas para todo $x,y,z \in A$.

$$x;(y;z) = (x;y);z \qquad (Ax.1)$$

$$(x+y);z = x;z + y;z \qquad (Ax.2)$$

$$(x+y)^{\smile} = \breve{x} + \breve{y} \qquad (Ax.3)$$

$$\breve{\breve{x}} = x \qquad (Ax.4)$$

$$x;1' = 1';x = x \qquad (Ax.5)$$

$$(x;y)^{\smile} = \breve{y};\breve{x} \qquad (Ax.6)$$

$$x;y\cdot z = 0 \Leftrightarrow z;\breve{y}\cdot x = 0 \Leftrightarrow \breve{x};z\cdot y = 0 \qquad (Ax.7)$$

La herramienta ReMo utiliza el lenguaje del álgebra relacional [T41] extendido con el operador *fork* (∇) [HV91], cuyo comportamiento se define cómo:

$$x\nabla y = (x;(1'\nabla 1))\cdot(y;(1\nabla 1'))$$
$$(x\nabla y);(p\nabla q) = (x;\breve{p})\cdot(y;\breve{q})$$
$$(1'\nabla 1)^{\smile};(1\nabla 1')^{\smile} \leq 1'$$

En la Figura 2.1 se presenta la gramática y semántica del lenguaje utilizado por la herramienta ReMo a efectos ilustrativos aunque no sea relevante para el desarrollo de la presente tesis.

$problem$ ::= decl* form
decl ::= var : $typexpr$
$typexpr$ ::=
$type \times type$
| $typexpr \times type$
| $type \times typexpr$

form ::=
expr <= expr (subset)
| !form (neg)
| form && form (conj)
| form || form (disj)

expr ::=
0_t (empty with type t)
| 1_t (universal with type t)
| id_t (identity with type t)
| expr + expr (union)
| expr & expr (intersection)
| [expr, expr] (fork)
| −expr (complement)
| ∼ expr(transpose)
| expr · expr (navigation)
| +expr (transitive closure)
| Var (variable)

M : form $\rightarrow$ $env \rightarrow Boolean$
X : expr $\rightarrow$ $env \rightarrow value$
$env = (var + type) \rightarrow value$
$value = atom +$
$atom \times value +$
$value \times atom$

$M[a <= b]e = X[a]e \subseteq X[b]e$
$M[!F]e = \neg M[F]e$
$M[F\&\&G]e = M[F]e \wedge M[G]e$
$M[F \mid\mid G]e = M[F]e \vee M[G]e$

$X[0_t] = \emptyset$
$X[1_t]$ = largest relation of type t
$X[a + b]e = X[a]e \cup X[b]e$
$X[a\&b]e = X[a]e \cap X[b]e$
$X[\,[a, b]\,]e = \{\, \langle x, \langle y, z\rangle\rangle : \langle x, y\rangle \in X[a]e \wedge \langle x, z\rangle \in X[b]e \,\}$
$X[-a] = X[1_t]e \setminus X[a]e$
$X[\sim a]e = \{\, \langle x, y\rangle : \langle y, x\rangle \in X[a]e \,\}$
$X[a \cdot b]e = X[a]e; X[b]e$
$X[+a]e$ = the smallest r such that $r;r \subseteq r$ and $X[a]e \subseteq r$
$X[v]e = e(v)$

Figura 2.1: Gramática y semántica del lenguaje de especificación de ReMo

3. El grupo de las permutaciones

1. Permutaciones

Definición 3.1: Una permutación de los elementos de un conjunto X es una función biyectiva de X en sí mismo.

Hay dos notaciones convencionales para describir una permutación:

Cartesiana

En una fila podemos describir el *orden natural* de los elementos a permutar y en otra el nuevo orden.
Por ejemplo:

1 2 3 4 5 6
5 1 4 6 2 3

nota que en la primera posición debe ubicarse el quinto elemento, en la segunda el primero, etc…

En la presente tesis supondremos que el *orden natural* de los elementos de un conjunto de cardinal n es 1 2 3 … n, y por lo tanto omitiremos la primera fila, describiendo sólo la segunda cuando nos refiramos a permutaciones en esta notación.

Cíclica

También es posible escribir una permutación en términos de cómo cambian los elementos cuando la permutación es aplicada. En el ejemplo anterior, el primer elemento se reemplaza por el quinto, el quinto por el segundo, el segundo por el primero, etc., lo que se nota con el ciclo (1 5 2). Con el resto de los elementos se sigue el mismo procedimiento y por lo tanto, la permutación 5 1 4 6 2 3 puede notarse en formato cíclico con los ciclos (1 5 2) (3 4 6).
La *forma canónica cíclica* de una permutación ubica en la primera posición de cada ciclo el número más chico del mismo y ordena los ciclos de menor a mayor según su primer elemento.
Esta notación permite omitir los puntos fijos, o sea los elementos que no cambian de posición. Por lo tanto la permutación 5 2 3 4 1 6, puede representarse con el ciclo (1 5).
En general, la notación cíclica es más compacta que la cartesiana.

Definición 3.2: Dadas dos permutaciones de n elementos π_1 y π_2, el producto de π_1 por π_2 es una permutación de n elementos que se obtiene de la aplicación sucesiva de las dos permutaciones. Por ejemplo:

Sean $\pi_1 = \begin{matrix} 1 & 2 & 3 \\ 1 & 3 & 2 \end{matrix}$ $\pi_2 = \begin{matrix} 1 & 2 & 3 \\ 2 & 3 & 1 \end{matrix}$

$$\pi_1 \pi_2 = \begin{matrix} 1 & 2 & 3 \\ 2 & 1 & 3 \end{matrix}$$

2. Grupos

Definición 3.3: Dados un conjunto no vacío G y una operación algebraica $*$, el conjunto G y la operación $*$ forman un grupo si:

- La operación $*$ en G es asociativa.
- Está definida, en G, la operación inversa a $*$.

Teorema 3.1: El conjunto de todas las permutaciones de n elementos junto con el operador *producto* (*Definición 3.2*) constituye un grupo.

Es fácil ver [K55] que el producto de permutaciones definido es asociativo, o sea, dadas las permutaciones π_1, π_2 y π_3,

$$(\pi_1\pi_2)\pi_3 = \pi_1(\pi_2\pi_3).$$

La permutación identidad (π_{id}) cumple con:

$$\pi_{id}\pi = \pi\pi_{id} = \pi$$

y la permutación inversa a

$$\begin{matrix} 1 & 2 & \dots & 3 \\ \alpha_1 & \alpha_2 & \dots & \alpha_n \end{matrix}$$

es

$$\begin{matrix} \alpha_1 & \alpha_2 & \dots & \alpha_n \\ 1 & 2 & \dots & 3 \end{matrix}$$

3. Permutando relaciones

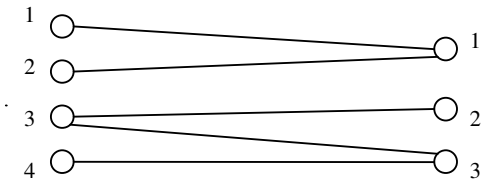
Definición 3.4: Consideremos una permutación π del conjunto U. Entonces podemos definir π'como:

$$\pi' R = \{(\pi u, \pi v) \mid (u,v) \in R\}$$

para cada valor de la relación R

Ejemplo:

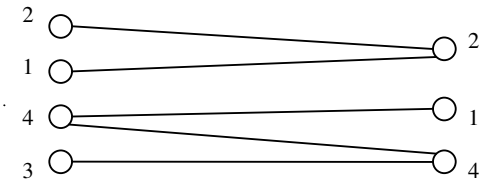
Sea R una relación binaria de tipo $T_1 \times T_1$, con $T_1 = \{1,2,3,4\}$ y la siguiente una asignación posible para R:



$R = \{(1,1)(2,1)(3,2)(3,3)(4,3)\}$

Y sea π = 2 1 4 3 una permutación del conjunto T_1. π define una permutación π' tal que $\pi' R = (2,2)(1,2)(4,1)(4,4)(3,4)$

$\pi' R =$



Definición 3.5: Dada una relación R y una permutación π, π es un automorfismo de R, si $\pi' R = R$.

Ejemplo:

π = 2 1 3 4 es un automorfismo de R ya que $\pi' R = R$.

.

4. Operaciones lógicas

Si consideramos el significado de los operadores relacionales como un conjunto de tuplas de relaciones, podemos definir π" como una permutación de los mismos operadores.

$$\pi'' O = \{(\pi' R_1, ..., \pi' R_n) \mid O(R_1, ..., R_{n-1}) = R_n\}$$

para cada operador O.

Definición 3.6: Un operador *lógico* es un operador tal que su conjunto característico de tuplas se mantiene invariante sobre cualquier permutación.

De forma equivalente, un *operador lógico* conmuta con la permutación de los valores de las relaciones; si O es un operador *lógico* que incluye a las relaciones $(R_1,...,R_n)$ y conmuta, entonces:

$$O=\{(\pi' R_1,...,\pi' R_{n-1}) \mid \pi' O(R_1,...,R_{n-1})=\pi' R_n\}$$

Y por lo tanto O debe incluir también a $(\pi' R_1,...,\pi' R_n)$ y mantenerse invariante sobre las permutaciones del dominio.

Ejemplo:

Consideremos el operador + . Es lógico porque:

$$\begin{aligned}
&\pi'(R)+\pi'(S)= \\
&(\breve{\pi};R;\pi)+(\breve{\pi};S;\pi)= \\
&(\breve{\pi};R+\breve{\pi};S);\pi= \\
&\breve{\pi};(R+S);\pi= \\
&\pi'(R+S)
\end{aligned}$$

4. Reducciones del espacio de Búsqueda

1. Monotonía

Una estrategia utilizada en [GS05] para podar el espacio de búsqueda a recorrer es el análisis de la monotonía de las variables relacionales.

Definición 2.1: Sea $T(x)$ un término relacional y x una variable relacional, se dice que T es monótono creciente con respecto a x si vale:

$\forall R \forall S (R \subseteq S \Rightarrow T(R) \subseteq T(S))$

para R y S asignaciones de la variable relacional x

Análogamente, T es monótono decreciente con respecto a x si vale:

$\forall R \forall S (S \subseteq R \Rightarrow T(S) \subseteq T(R))$

Contar con esta información puede ser muy útil a la hora de buscar valores para las relaciones que satisfagan los axiomas y no las propiedades. Supongamos que tenemos una fórmula de la forma $T = 1$, siendo 1 la relación universal correspondiente a la aridad de T. Si T es monótono creciente con respecto a una relación R, basta con hallar una instancia I de R que haga que $T(I) = 1$ se satisfaga para saber que, para cualquier otra instancia I' de R tal que $I \subseteq I'$ seguirá valiendo $T(I') = 1$. Y, por la contrarrecíproca, si J es una instancia de R para la cual $T(J) = 1$ es falso, sabemos que $\forall J' (J' \subseteq J \Rightarrow \neg(T(J') = 1))$.

Resultados análogos (e inversos) se obtienen en el caso de que T sea monótono decreciente con respecto a R.

Si todas las fórmulas de una especificación tuvieran la forma $T = 1$, y además se pudiera conocer la monotonía de cada uno de estos términos T para todas las relaciones que intervienen en ellos, no sería necesario generar todas las instancias posibles para cada relación: si una instancia de la relación R no satisficiera un axioma, podríamos saber que, o bien todas las instancias contenidas en ella, o bien todas las que la contienen (según fuese T creciente o decreciente con respecto a R), seguirán haciendo falso dicho axioma; por lo tanto, podríamos descartar todas esas instancias de antemano, ya que ninguna de ellas formará parte de un modelo de la especificación. De la misma forma, se podría predecir que un conjunto de instancias va a satisfacer todas las propiedades, y descartarlas ya que no se hallarán contraejemplos para ellas.

Si bien no existe un método automático para conocer la monotonía de un término con respecto a todas sus relaciones en el caso general, sí es posible obtenerla en algunos casos a partir de los signos con los que aparece cada relación en cada fórmula.

Definición 2.2.: Una variable relacional R aparece positiva en una fórmula relacional F, siendo F de la forma $T = 1$ para algún término relacional T, si R aparece afectada por una cantidad par de complementos en T. Una variable relacional R aparece negativa en una fórmula relacional F, siendo F de la forma $T = 1$ para algún término relacional T, si R aparece afectada por una cantidad impar de complementos en T. Como antes, llamamos 1 a la relación universal correspondiente a la aridad de T.

Una variable relacional puede aparecer a la vez positiva y negativa en una misma fórmula. Por ejemplo, R aparece positiva y negativa en $R + \overline{R} = 1$

Distinguimos, entonces, 4 valores posibles para el signo de R en F:

Positivo: R aparece únicamente positiva en F.
Negativo: R aparece únicamente negativa en F.
Ambos: R aparece positiva y negativa en F.
Ninguno: R no aparece en F.

Se puede ver que, si el signo de R en $T = 1$ es Positivo, entonces T es monótono creciente respecto de R. A la vez, si el signo de R en $T = 1$ es Negativo, entonces T es monótono decreciente respecto de R. Ambas propiedades pueden demostrarse haciendo inducción en la estructura del término T. Obviamente, si el signo es Ninguno, el resultado obtenido al evaluar $T = 1$ no dependerá de R. Si el signo es Ambos, no podemos afirmar nada sobre la monotonía de T con respecto a R. Puede ser monótono (ejemplo: $R + (R.\overline{R}) = 1$) o no serlo (ejemplo: $\breve{R} + \overline{R} = 1$).

Si bien no todas las fórmulas son de la forma $T = 1$ para algún término relacional T, existe un procedimiento efectivo para traducir una fórmula sin variables a otra equivalente con esta forma [TG87].

2. Recorridos

Como elemento fundamental para el algoritmo de búsqueda de contraejemplos se necesita un método para listar las relaciones involucradas de acuerdo a la especificación; más aun, ha de ser posible generar todas las combinaciones admisibles de instancias de las relaciones. El primer requisito para estos procedimientos es que no generen durante un recorrido de una variable más de una vez una relación (o que sea posible detectar esta repetición) y, luego, que no se repitan combinaciones de instancias.

Lo que se desea, entonces, es algún procedimiento efectivo para recorrer el reticulado[1] asociado al espacio de búsqueda de una relación. Para estos reticulados relacionales la relación vacía constituye el mínimo y la universal el

[1] *Definición*:: Un retículado es un conjunto parcialmente ordenado en el cual todo subconjunto finito no vacío tiene un supremo y un ínfimo.

máximo, y la relación de inclusión entre relaciones define el orden entre los elementos.

En [GS05] se definen dos posibles formas generales de recorrerlos:

1. **BFS (Breadth First Search):**

   El reticulado se recorre por niveles. Un nivel está definido por la cantidad de pares que tiene la relación (en una relación de $n \times n$ hay $n^2 + 1$ niveles). Dada una instancia de una relación R, la siguiente en el recorrido se obtiene sacándole un par y agregándole otro que no haya sido usado junto a los pares que permanecen en R. Cuando ya se visitaron todas las combinaciones de pares posibles en un nivel, se pasa al siguiente.

2. **DFS (Depth First Search):**

   El reticulado se recorre por ramas. Dada una instancia de una relación R, la rama de R en el reticulado se define como $\{S / R \subseteq S \vee S \subseteq R\}$. Entonces, dada R, la siguiente relación a visitar es la siguiente en la rama (agregando o quitando un par según se esté ascendiendo o descendiendo en el reticulado). Al llegar al extremo de una rama, se pasa a otra todavía no visitada. Así enunciado, este recorrido estará visitando varias veces una misma relación (una por cada permutación de sus pares), lo cual no es deseado (Ver figura 4.1). Por lo tanto, al momento de avanzar, si la que será la próxima relación a visitar ya fue generada, se pasa a otra rama no visitada.

   Se llama avanzar a la acción de obtener la siguiente instancia (o valuación) para una variable relacional según su recorrido. Si se tiene información de monotonía se puede conocer algunos resultados a partir de otros y por lo tanto evitar el recorrido completo del reticulado ya que no es necesario pasar por aquellas instancias de relaciones para las cuales el resultado de las fórmulas se infiere, y que no pueden formar parte de un modelo o un contraejemplo.

   Si se utiliza un recorrido DFS y efectuamos una poda (interrupción del recorrido de una rama, pasando a la siguiente) se estarán descartando todas las relaciones incluidas o que incluyan (según el recorrido sea descendente o ascendente) a la última visitada (y que no hayan sido consideradas antes). Entonces, combinando esto con la información de monotonía se puede lograr una reducción del espacio de búsqueda:

   Si una constante relacional aparece positiva en una fórmula, un recorrido DFS ascendente nos permite, una vez conocida una instancia para la cual la fórmula fue verdadera, saber que para todas las relaciones que la sucederán en su rama la fórmula seguirá siendo verdadera.

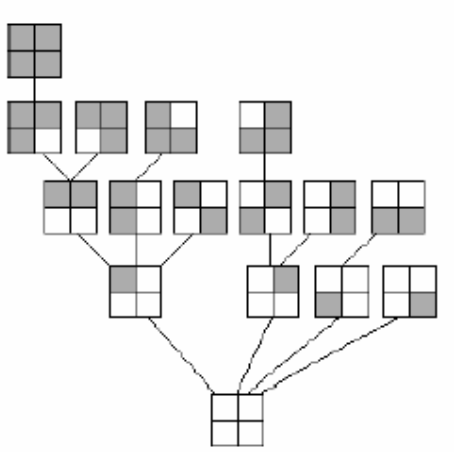


Figura 4.1: Un ejemplo de un recorrido DFS ascendente

Análogamente, un recorrido DFS descendente nos permitirá saber que a partir de una cierta instancia en que falle la fórmula, ésta seguirá siendo falsa durante el resto del recorrido de la rama. Si una constante aparece negativa en una fórmula los resultados son exactamente opuestos a los descriptos.

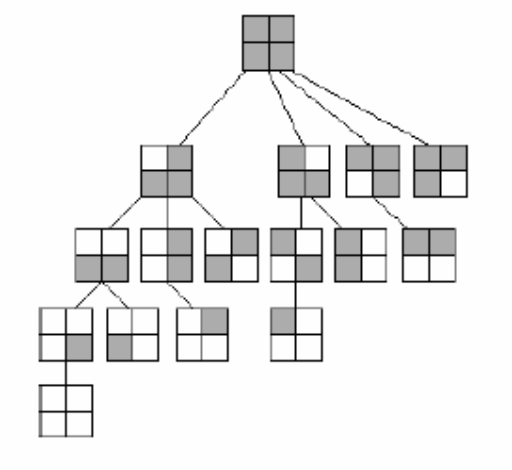


Figura 4.2: Un ejemplo de un recorrido DFS descendente

Sabiendo esto, se pueden elegir los recorridos a utilizar de manera tal que las podas permitan descartar partes del reticulado para las cuales la búsqueda de contraejemplos no dará frutos (ya sea porque algún axioma es siempre falso o las propiedades son siempre verdaderas). Por otro lado, el recorrido BFS es mejor utilizarlo para las situaciones en las cuales el análisis de monotonía no resulta útil, ya que se aprovecha el aumento de eficiencia en la generación de instancias provisto por ese recorrido.

Como se deduce de la sección 1, no siempre podrá adoptarse un recorrido del reticulado que aproveche la información de monotonía. En particular porque podemos tener especificaciones que no contengan dicho tipo de información. En

una tesis actualmente en desarrollo (ver [M??]), se está implementando una nueva estrategia para la reducción del espacio de búsqueda que es *general*. En este contexto, la generalidad de la estrategia deviene de su aplicabilidad a especificaciones arbitrarias. La estrategia desarrollada consiste en la eliminación de isomorfismos durante el proceso de enumeración de modelos, como se suscribe a continuación.

3. Eliminación de isomorfismos

Un modelo relacional consiste de:

1. Conjuntos no vacíos para cada dominio de datos, y
2. Relaciones de la aridad que corresponda para cada una de las variables relacionales de la especificación.

Recordemos (capítulo 3) que las operaciones del álgebra relacional son *lógicas*. Luego, es claro que una estrategia de generación de modelos que tome en cuenta los elementos de los dominios corre el riesgo de generar modelos equivalentes para la especificación, es decir, modelos que se obtienen unos de otros por permutaciones de los elementos de los dominios, y para los cuales la evaluación de las fórmulas de la especificación no varía.

Luego, si se eliminaran estos modelos se mejoraría el tiempo de análisis de una especificación, mientras que al mismo tiempo se garantizaría la no pérdida de contraejemplos en caso de que estos existan.

La eliminación de modelos isomorfos (así se llaman estos modelos que se diferencian sólo por una permutación de los elementos de su conjunto base), puede ser computacionalmente compleja. Por ello es necesario encontrar un compromiso entre la cantidad de modelos eliminados y el tiempo necesario para determinar que deben eliminarse.

k	con isomorfismos	sin isomorfismos
1	2	2
2	16	7
3	512	36
4	65.536	317
5	$3{,}4.10^{7}$	5.624
6	$6{,}9.10^{10}$	251.610
7	$5{,}6.10^{14}$	33.642.660

Figura 4.3: Cantidad de relaciones binarias de $k \times k$ con y sin la inclusión de isomorfismos

Un primer aporte en esta dirección fue realizado por Daniel Jackson, Somesh Jha y Craig Damon [JJD98], al presentar una estrategia de generación de modelos relacionales que evita la generación de una importante cantidad de modelos isomorfos. La estrategia consiste en la generación incremental de valores para las variables relacionales del modelo, y la eliminación de permutaciones que se puede garantizar que producirán modelos isomorfos.

Durante la generación incremental de los valores para las variables relacionales, consideremos que tenemos una interpretación que ha asignado valores $v_1, v_2, ..., v_i$ a las variables relacionales $r_1, r_2, ..., r_i$ y nos encontramos en el momento de generar un valor para la variable r_{i+1}. Si tenemos valores a_1 y a_2 para r_{i+1} y una permutación π tal que $\pi(v_j) = v_j \quad \forall j \ 1 \le j \le i$ y $\pi(a_1) = a_2$, entonces no es necesario generar ambos valores a_1 y a_2, dado que $v_1, v_2, ..., v_i, a_1$ será extensible a un contraejemplo sí y sólo sí $v_1, v_2, ..., v_i, a_2$ lo es. La generación de uno sólo de los valores alcanza. Para determinar de forma eficiente si una permutación producirá un modelo isomorfo a uno previamente generado, Jackson, Jha y Damon mantienen una partición de los dominios que no es óptima pero que se calcula de forma eficiente.

En este método, que fue implementado en ReMo, las interpretaciones son construidas incrementalmente en un árbol. Las variables se ordenan y en cada nivel del árbol se enumeran los valores de una variable. Cada hoja del árbol se corresponde con una interpretación. La Figura 4.4 muestra una parte del árbol. La hoja marcada como *I* da la interpretación que se obtiene al asignarles a las variables los valores de los nodos marcados: en este caso la primera variable R_1 tiene el valor a, la segunda el valor b, etc, hasta la última que tiene el valor g.

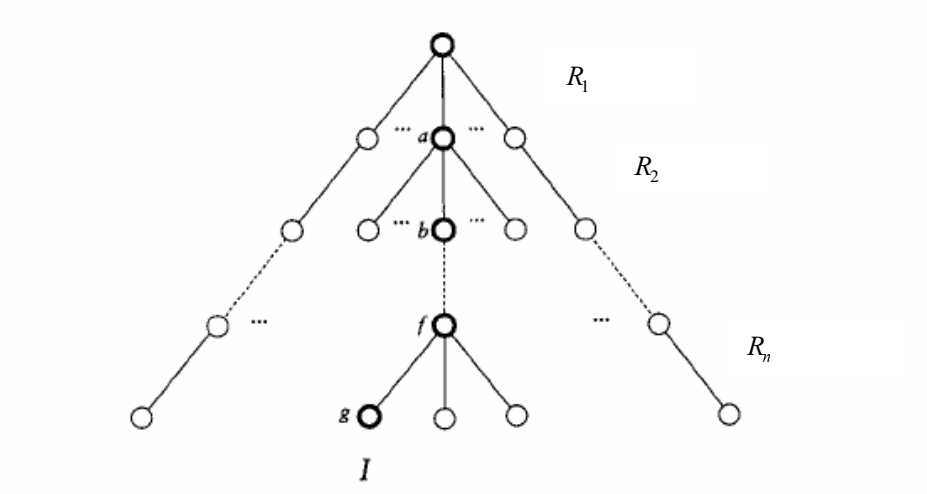


Figura 4.4: Parte de un árbol de enumeración

Las interpretaciones isomorfas no se eliminan en las hojas, ya que esto requeriría la construcción del árbol completo, sino que son podadas durante la enumeración. En lugar de generar todos los valores para una variable, se generan los valores suficientes como para garantizar que al menos una interpretación perteneciente a cada clase de equivalencia estará presente. El conjunto de los valores generados como hijos de un nodo en el árbol, varía entre los nodos de un mismo nivel porque la noción de equivalencia depende de la interpretación que se está construyendo.

Dada una interpretación *I* parcialmente construida, el método evita la generación de πv para cualquier permutación π que sea automorfismo de *I*. La Figura 4.5 ilustra esta situación. El nodo interno marcado como *I* se corresponde con la interpretación parcial; v_1 y v_2 son valores enumerados tales que:

$$\pi v_1 = v_2$$

donde π es un automorfismo de I, o sea, para todos los valores de una relación en *I,*

$$\pi v = v$$

En esta situación, sólo uno de los valores v_1 y v_2 es necesario (ya que v_2 es isomorfo a v_1), entonces si se incluye v_1, v_2 puede evitarse y por lo tanto se puede podar el subárbol que le corresponde.

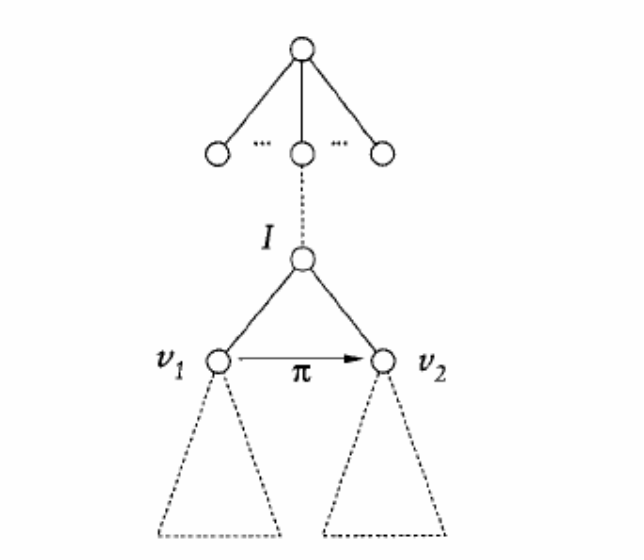


Figura 4.5: Eliminación de isomorfismos en un nodo interno.

Por lo tanto, si queremos encontrar valores para una variable relacional R, basta con generar las asignaciones no isomorfas y aplicarles las permutaciones que no son automorfismos de la asignación parcial *I.*

El método de Jackson, Jha y Damon aprovecha sólo un subconjunto de los automorfismos. Para cada relación considera permutaciones del dominio o del codominio de la relación.

ReMo incorpora la técnica de eliminación de isomorfismos. Para hacerlo, almacena el conjunto de permutaciones que puede ser evitado al generar valores para una determinada variable. A diferencia de Jackson, Jha y Damon que consideran permutaciones del dominio o del codominio de una relación, en ReMo se consideran ambos, con lo que es posible eliminar una mayor cantidad de permutaciones.

Nótese que dado un conjunto de permutaciones a almacenar, es esencial que dicho almacenamiento sea eficiente en cuanto al espacio utilizado y en cuanto al tiempo requerido para generar las permutaciones no evitables.

5. Generación de permutaciones usando tries

En el marco de la estrategia de eliminación de isomorfismos se asignan valores a las variables relacionales evitando un conjunto de automorfismos de las asignaciones previas que se sabe no darán resultados adicionales. Para determinar las asignaciones que deben realizarse, en la tesis en curso que implementa la estrategia de eliminación de isomorfismos ([M??]) se almacenan en un trie las permutaciones que deben evitarse, y mediante la generación exhaustiva de todas las permutaciones del dominio determina cuáles deben ser generadas en función de su pertenencia al trie.

Las permutaciones que deben evitarse se almacenan en formato cíclico para optimizar la cantidad de espacio necesario para su almacenamiento, pero se generan en formato cartesiano, por lo tanto es necesaria la conversión entre las dos notaciones tanto en el momento de su almacenamiento, como luego de determinar que una permutación debe ser efectivamente generada.

Algoritmo de Generación:

```
INPUT: conjunto de permutaciones a no generar.
OUTPUT: el complemento del conjunto de permutaciones dadas como
        INPUT.

       Por cada permutación, convertirla al formato cíclico e
       insertarla en el trie
       Por cada permutación generada por algún algoritmo exhaustivo
              (se generan de esta manera todas las permutaciones de ese dominio)
              Si está almacenada en el trie, descartarla
              Sino, convertirla al formato cartesiano y devolverla.
```

Dado que es necesario realizar esta generación cada vez que debe generarse un nuevo conjunto de asignaciones de valores para una variable relacional, toda optimización que se pueda hacer en la generación redundará en una sustancial disminución del tiempo total necesario para chequear una especificación.

Hay algunos aspectos de esta estrategia que sería deseable optimizar:

1. Independientemente del tamaño del conjunto de permutaciones que haya que generar, esta estrategia debe generar todas las permutaciones del dominio especificado. Sería interesante poder generar sólo aquellas permutaciones que se requieran, sin que sea necesaria la obtención de todas aquellas del mismo dominio para poder determinarlas.

2. Por cada permutación que se genera es necesario determinar si pertenece o no al conjunto de las que no es necesario generar, con un costo que si bien usando un trie es lineal en la cantidad de elementos de la permutación, se realiza para la totalidad de las permutaciones del dominio (factorial en el tamaño del mismo).

3. La estructura de almacenamiento de permutaciones. Hay estructuras basadas en árboles que son más eficientes que otras respecto del espacio que utilizan. Por ejemplo, los tries llamados *patricia* permiten albergar más de un caracter entre dos nodos en los casos en los cuales un nodo tenga un único hijo. Aún así, estas estrategias aprovechan los prefijos comunes de las cadenas que albergan, pero no los sufijos, por lo cual en algunas situaciones puede suceder que se esté almacenando información redundante.
   Por otro lado, aún si se consideraran optimizaciones de estructuras de este tipo respecto al espacio de almacenamiento, no permitirían un cambio sustancial en la estrategia de generación del complemento, ya que de todas formas sería necesario generar todas las permutaciones del dominio especificado.

Estos factores indican que una estructura que pueda aprovechar tanto los prefijos como los sufijos de las cadenas que almacena y que permita obtener fácilmente el complemento del conjunto de permutaciones sería una optimización importante para el problema de la generación de valores para las variables relacionales.

En la Figura 5.1 se muestra la estructura de un trie al cual se le insertaron las permutaciones 3 1 4 2, 1 4 3 2 y 2 1 4 3 ((2 1 3 4), (2 4) y (4 3)(2 1) en formato cíclico respectivamente).

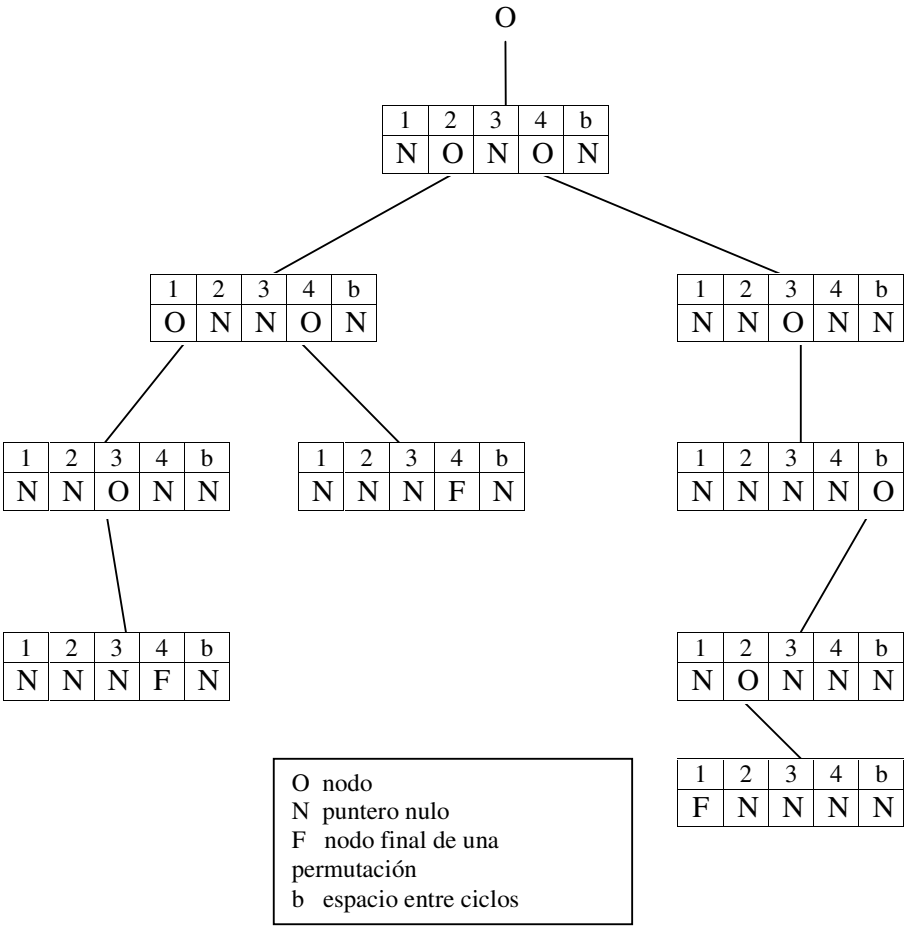


Figura 5.1: Un trie con las permutaciones (2 1 3 4), (2 4) y (4 3)(2 1)

6. Autómatas finitos determinísticos

En la presente tesis se propone a los autómatas finitos como estructura para una estrategia de generación que permita obtener sólo aquellas permutaciones necesarias; se analizan las distintas variantes de implementación de la nueva estrategia y se presentan los resultados obtenidos.

La estructura de almacenamiento de permutaciones que utilizaremos será un autómata finito determinístico. Esta estructura permite obtener el complemento de un lenguaje regular con un costo lineal en la cantidad de estados del mismo y puede ser minimizada.

Un autómata finito determinístico es una quíntupla $(\Sigma, Q, q_0, \partial, F)$, en donde:

- Σ es el alfabeto de entrada (un conjunto finito y no vacío de símbolos). En nuestro caso, serán los naturales del intervalo [1,..,d] donde d es el tamaño del dominio a considerar.
- Q es un conjunto finito y no vacío de estados.
- q_0 es un estado inicial, elemento de Q
- ∂ es la función de transición de estados: $\partial : Q \times \Sigma \to Q$
- F es el conjunto de estados finales, subconjunto de Q.

Definición 6.1: Un autómata finito es determinístico si

$$\forall e \in \Sigma, \forall q, q_1, q_2 \in Q, \quad \partial(q,e) = q_1 \wedge \partial(q,e) = q_2 \Rightarrow q_1 = q_2$$

Definición 6.2 : Dos estados q_1 y q_2 son indistinguibles si

$$\forall x \in \Sigma^*, \tilde{\partial}(q_1, x) \subset F \Leftrightarrow \tilde{\partial}(q_2, x) \subset F$$

en donde $\tilde{\partial} : Q \times \Sigma^* \to Q$ es una extensión de la función de transición de estados ∂ para cadenas definida por:

$$\tilde{\partial}(q_1, e) = \partial(q_1, e)$$
$$\tilde{\partial}(q_1, ex) = \tilde{\partial}(\partial(q_1, e), x)$$

con $e \in \Sigma$ y $x \in \Sigma^*$

Definición 6.3: Un autómata finito es mínimo si no contiene dos estados indistinguibles.

Los autómatas finitos se pueden utilizar para representar un lenguaje sobre un alfabeto Σ. El lenguaje que nos interesará representar es el de las permutaciones de determinado dominio.

Por ejemplo, el autómata de la Figura 6.1 acepta las cadenas 2 0 3 1, 2 3 0 1 y 1 2 3 0 que representan a las correspondientes permutaciones de cuatro elementos.

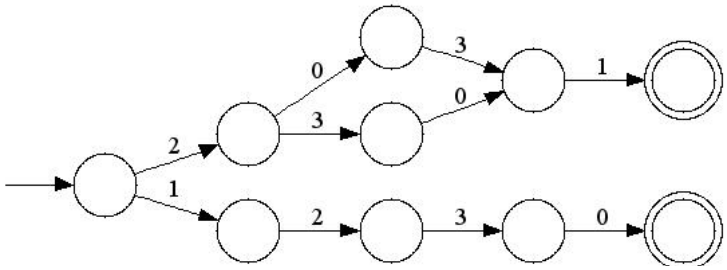


Figura 6.1: Un autómata que acepta las cadenas 2 0 3 1, 2 3 0 1 y 1 2 3 0

1. Algoritmos sobre autómatas

Existen distintas operaciones que se pueden realizar sobre autómatas finitos y sobre el lenguaje que aceptan. A continuación enumeramos las operaciones que serán de nuestro interés y la complejidad de los algoritmos conocidos para implementar dichas operaciones. Se asume que n es la cantidad de estados del autómata.

- Unión: Dados dos autómatas a_1 y a_2 , genera un autómata que acepta la unión de los lenguajes aceptados por a_1 y a_2.
  Por ejemplo, si:

a_1 =

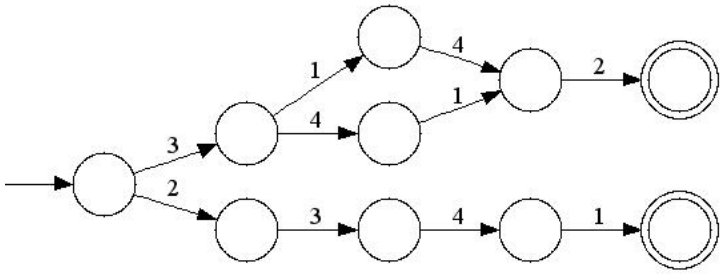


Figura 6.2: a_1 acepta las cadenas 3 1 4 2, 3 4 1 2 y 2 3 4 1

a_2 =

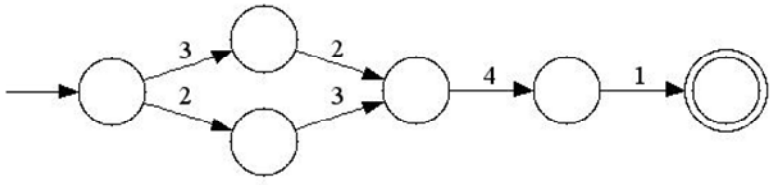


Figura 6.3: a_2 acepta las cadenas 3 2 4 1 y 2 3 4 1

$a_1 \cup a_2 =$

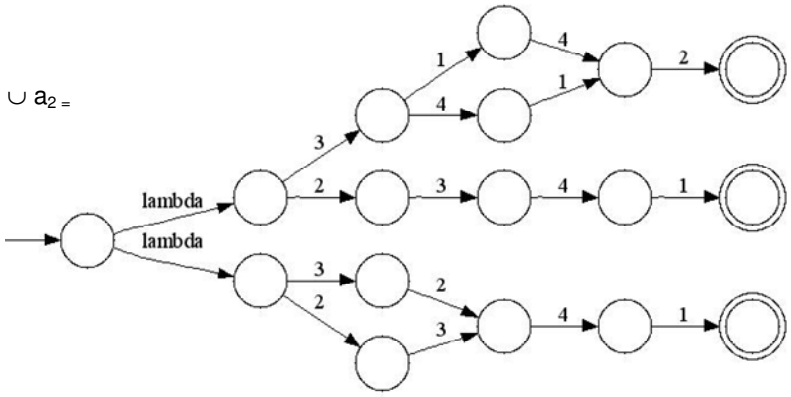


Figura 6.4: $a_1 \cup a_2$ acepta las cadenas 3 1 4 2, 2 3 4 1, 2 3 4 1, 3 2 4 1 y 2 3 4 1

$a_1 \cup a_2$ acepta tanto las permutaciones que acepta a_1 como las que acepta a_2. O sea: 3 1 4 2, 2 3 4 1, 2 3 4 1, 3 2 4 1 y 2 3 4 1.

Las transiciones etiquetadas lambda permiten el pasaje de un estado a otro sin la consumición de ningún elemento de la cadena de entrada.

El autómata $a_1 \cup a_2$ no es un autómata ni determinístico ni mínimo. Si bien la operación de unión tiene un costo constante, para obtener un autómata determinístico es necesario un costo exponencial en la cantidad de estados [HU79].

- Complemento: Esta operación construye, a partir de un autómata a_1, un autómata a_3 que acepta el complemento del lenguaje que acepta a_1 con respecto a un alfabeto.

  Para realizar esta operación, es necesario que el autómata a_1 sea total, o sea que $\forall e \in \Sigma, \forall q \in Q \quad \partial(q,e)$ esté definida. Para ello, se le añade al autómata un estado "trampa" t, de no aceptación, tal que $\forall e \in \Sigma, \forall q \in Q /$ $\partial(q,e)$ no está definida en a_1, $\partial(q,e) = t$ en a_2. Al ser t un estado de no aceptación, el lenguaje aceptado por los dos autómatas no varía.

  Por ejemplo:
  Si

$a_1 =$

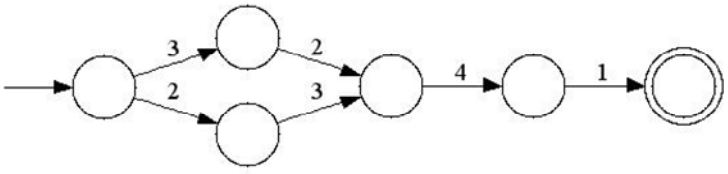


Figura 6.5: a_1 acepta las cadenas 3 2 4 1 y 2 3 4 1

El autómata a_2 que reconoce el mismo lenguaje que a_1, tal que su función de transición de estados es total sería:

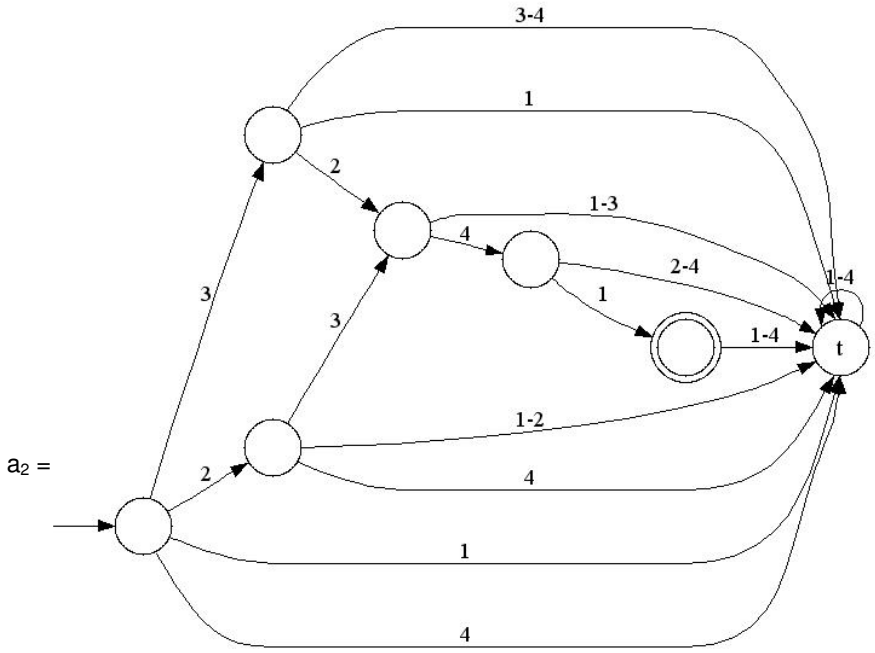


Figura 6.6: a_2 es total y acepta las cadenas 3 2 4 1 y 2 3 4 1

Dado un autómata a_2 tal que su función de transición de estados es total, para obtener el autómata a_3 que acepta el complemento del lenguaje aceptado por a_2 , basta con cambiar la propiedad de aceptación de los estados, de forma tal que $\forall q \in Q, (q \in F$ en $a_2 \Leftrightarrow q \notin F$ en $a_3)$

Continuando con nuestro ejemplo, el autómata a_3 que reconoce el complemento del lenguaje aceptado por a_2 sería:

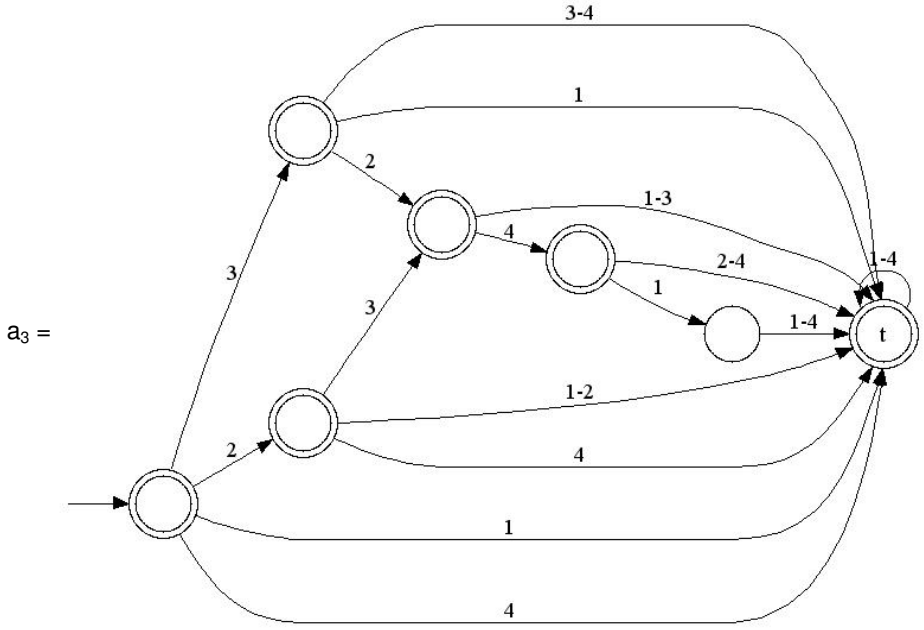


Figura 6.7: a_3 es total y acepta el complemento con respecto a $\Sigma = \{1,2,3,4\}$ de las cadenas 3 2 4 1 y 2 3 4 1

El proceso de totalización tiene un costo cuadrático en la cantidad de estados, mientras que se puede realizar la complementación en tiempo constante.

- Minimización

El proceso de minimización consiste en individualizar los estados que son equivalentes entre sí y que suponen una redundancia en la estructura y eliminarlos. El resultado es un autómata finito que acepta el mismo lenguaje que el original pero con la mínima cantidad de estados.

Existen distintos algoritmos de minimización. Mencionamos algunos de ellos junto con su complejidad en donde n es la cantidad de estados del autómata.

-Algoritmo de Huffman O(n^2) [HU79]
-Algoritmo de Hopcroft O($n \log n$) [G73]

2. Expresiones Regulares

Los lenguajes aceptados por los autómatas finitos pueden ser fácilmente descriptos por expresiones simples llamadas *expresiones regulares*.

Sea Σ un conjunto finito de símbolos y sean L, L_1 y L_2 conjuntos de cadenas pertenecientes a Σ^*. La concatenación de L_1 y L_2, denotada $L_1 L_2$ es el conjunto $\{xy / \ x \in L_1 \wedge y \in L_2\}$. Definimos $L^0 = \{\lambda\}$ y $L^i = LL^{i-1}$ para $i \geq 1$. La clausura de Kleene (o clausura) de L denotada L^*, es el conjunto:

$$L^* = \bigcup_{i=0}^{\infty} L^i$$

y la clausura positiva de L denotada L^+**,** es el conjunto:

$$L+ = \bigcup_{i=1}^{\infty} L^i$$

Sea Σ un alfabeto. Las expresiones regulares sobre Σ y los conjuntos que denotan están definidos recursivamente de la siguiente manera:

1. $\varnothing$ es una expresión regular y denota al conjunto vacío.
2. λ es una expresión regular y denota al conjunto $\{\lambda\}$
3. Por cada $a \in \Sigma$, a es una expresión regular y denota el conjunto $\{a\}$.
4. Si r y s son expresiones regulares que denotan los lenguajes R y S respectivamente, entonces $(r \mid s)$, (rs) y $(r*)$ son expresiones regulares que denotan a los conjuntos $R \cup S$, RS y $R*$ respectivamente.

Se puede ver que los lenguajes aceptados por los autómatas finitos son exactamente aquellos denotados por expresiones regulares [HU79].

.

7. Estrategias con Autómatas

Es posible ver a las permutaciones en formato cartesiano como cadenas de Σ^*, siendo Σ el alfabeto definido por $\{1,2,...,n\}$ con n el tamaño del dominio a considerar. Por lo tanto es posible la construcción de un autómata finito que acepte el lenguaje de las permutaciones que es necesario evitar y luego complementar este autómata para obtener las permutaciones que es necesario generar.

Dentro del lenguaje aceptado por el autómata complementado existen cadenas que no son permutaciones válidas en el dominio que estamos considerando. Por ejemplo, cadenas con elementos repetidos o de longitud distinta al tamaño del dominio,

Sin embargo, es posible realizar un recorrido del autómata en la búsqueda de las cadenas que acepta, de forma tal de sólo quedarnos con aquellas que representan a permutaciones válidas del dominio que estamos considerando, podando aquellas ramas ni bien sabemos que la cadena que estamos considerando viola alguna condición necesaria.

Presentaremos los distintos enfoques que se utilizaron para construir el autómata que aceptara un conjunto de permutaciones, describiremos los problemas que se presentaron, las soluciones que propusimos y finalmente los valores que muestran que efectivamente es posible mejorar, con este nuevo enfoque, los tiempos de generación del complemento de un conjunto de automorfismos.

1. El problema de la construcción del autómata.

Dado que a partir de una expresión regular es posible construir el autómata que acepta el mismo lenguaje, es posible construir el autómata que acepta el lenguaje de todas las permutaciones que queremos evitar a partir de una expresión regular que las exprese.

Por ejemplo, dado el lenguaje compuesto por el siguiente conjunto de permutaciones para un dominio de tamaño cuatro:

4 1 2 3
1 2 4 3
2 1 4 3

la expresión regular que lo expresa es 4 1 2 3 | 1 2 4 3 | 2 1 4 3.

Dada una expresión regular de la forma $p_1 \mid p_2 \mid p_3 \mid \ldots \mid p_n$, donde p_i es una cadena perteneciente a Σ^* el algoritmo de construcción de autómatas

determinísticos construye trivialmente el autómata que acepta la cadena p_1 y utiliza el algoritmo de unión de autómatas presentado en la Figura 6.4 para unir éste con el resultado de la llamada recursiva sobre el resto de la expresión regular. Este algoritmo consume un tiempo constante, pero es necesario determinizarlo para complementarlo lo cual tiene un costo exponencial.

En la Figura 7.1 se muestra el autómata construido mediante uniones a partir de la expresión regular 4 1 2 3 | 1 2 4 3 | 2 1 4 3.

Ejemplo:

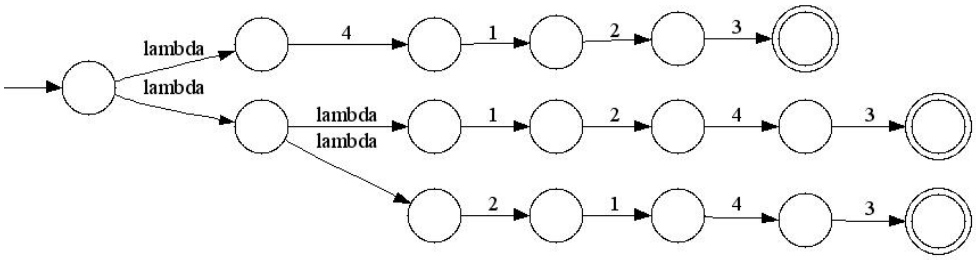


Figura 7.1: Autómata que acepta las cadenas 4 1 2 3, 1 2 4 3 y 2 1 4 3

Autómata determinístico y mínimo:

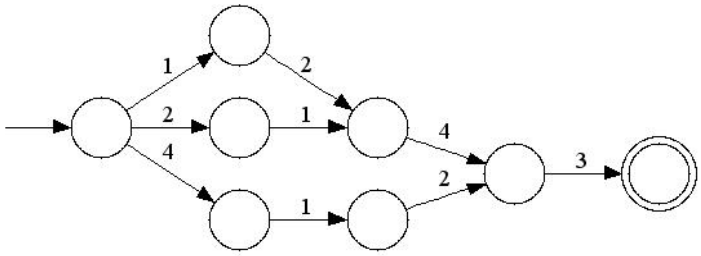


Figura 7.2: Autómata determinístico y mínimo que acepta las cadenas 4 1 2 3, 1 2 4 3 y 2 1 4 3

La primera aproximación utilizada en esta estrategia construía, a partir del conjunto de permutaciones a no generar, una cadena representando a la expresión regular que expresaba el lenguaje de ese conjunto de permutaciones y construía, a partir de la cadena, una estructura que representaba a la expresión regular.

A partir de la expresión regular, y utilizando el algoritmo mencionado en la Figura 6.4 se construía el autómata.

Algoritmo de generación

```
INPUT: conjunto de permutaciones a no generar.
OUTPUT: el complemento del conjunto de permutaciones dadas como
        INPUT.

       Lectura del conjunto de permutaciones de entrada y
       construcción de la cadena con la sintaxis de una expresión
```

```
regular que expresa el lenguaje del conjunto de permutaciones
leído.

Construcción, a partir de la cadena, de la expresión regular.

Construcción del autómata determinístico, a partir de la
expresión regular.

Totalización y complementación del autómata.

Lectura del autómata complementado para generar las
permutaciones válidas que acepta el autómata.
```

Los primeros tiempos que se obtuvieron con esta aproximación, no tenían comparación a los tiempos obtenidos con la estrategia del trie. Se tardaba muchísimo en la construcción de la cadena, de la expresión regular y en la construcción del autómata a partir de esta última.

Diversos factores influían en estos tiempos. La construcción de la expresión regular era costosa y la determinización del autómata es exponencial en la cantidad de estados (y la cantidad de estados es exponencial en función del tamaño del dominio) y necesaria al construirlo a partir de una expresión regular basada en uniones.

Una opción era optimizar la expresión regular. En vez de tener en cuenta una expresión de la forma $p_1|p_2|p_3|..|p_n$, la alternativa era construirla tratando de miniminizarla, de forma que la determinización del autómata sea a su vez menos costosa.

Pero de todas formas, el costo que se ahorra en algoritmos sobre autómatas, se ganaría en estrategias sobre expresiones regulares, con el problema de que la expresión regular como estructura o como cadena, no es fácilmente manipulable.

Como conclusión se decidió evitar la construcción de autómatas a partir de expresiones regulares y se probaron distintas aproximaciones al problema de la construcción de forma manual.

2. El gran autómata

Uno de los intentos consistió en la construcción manual de un gran autómata que aceptara en principio todas las permutaciones del dominio especificado.

Si construyéramos esta estructura, podríamos, por cada permutación perteneciente al conjunto que queremos evitar, recorrer hasta la hoja correspondiente del autómata y cambiar el atributo de aceptación del estado.

El autómata construido es determinístico, no necesitaríamos totalizarlo, y el complemento se realiza trivialmente cuando “cargamos” la información de las permutaciones que no queremos evitar modificando los estados finales.

Además, el autómata que resulta, acepta como lenguaje exactamente aquellas permutaciones que queremos generar, no es necesario correr un algoritmo especial que sólo obtenga las permutaciones válidas del lenguaje aceptado por el autómata.

Algoritmo de generación

```
INPUT:  conjunto de permutaciones a no generar.
```

```
OUTPUT: el complemento del conjunto de permutaciones dadas como
        INPUT.


        Construcción del autómata que acepta todas las permutaciones
        válidas del dominio.

        Lectura del conjunto de permutaciones de entrada, y por cada
        permutación, recorrido del autómata hasta el estado final en
        el cual se acepta. Cambio del atributo de aceptación de ese
        estado.

        Lectura del lenguaje aceptado por el autómata.
```

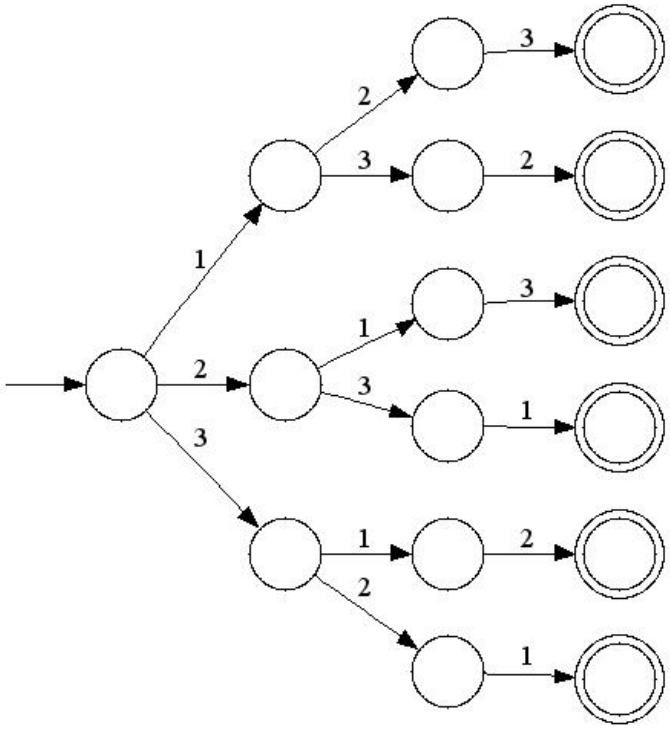


Figura 7.3: Autómata que acepta todas las permutaciones de tres elementos

Si bien esta estrategia de construcción tiene varias ventajas, tiene un importante problema: El tamaño del autómata. Es costoso construirlo, tanto en tiempo como en memoria. No se estaría optimizando uno de los aspectos que aspirábamos a mejorar: no generar la totalidad de permutaciones del dominio. Ya para un dominio de diez elementos, el tamaño del autómata requiere un uso de memoria mayor a 1500 Mb, y dado que la generación del complemento de automorfismos debe realizarse durante el proceso de asignación de valores a las variables relacionales y que probablemente deba almacenarse más de un autómata durante este proceso, ese consumo de memoria no resulta conveniente ni razonable. Por más que se decidiera pagar el costo que implicaría minimizarlo, es necesario construir primero toda la estructura y por lo tanto, no evitaríamos el consumo de memoria necesaria para albergarla durante el algoritmo de

generación. En el Capítulo 8 (Resultados), puede verse una comparación de las dos estrategias respecto de de los tiempos y memoria utilizada.

3. Creaciones inteligentes

Una mejora deseable al enfoque anterior, sería construir el autómata que sólo albergara a las permutaciones que queremos evitar, aunque tengamos nuevamente el costo de la totalización y del complemento que explicamos anteriormente.

Dado el conjunto de permutaciones a complementar, podemos ir insertando "inteligentemente" cada una de ellas en el autómata, de forma que el mismo sea determinístico.

Algoritmo Insert

```
INPUT: p, permutación a insertar en el autómata; a, autómata
OUTPUT: el autómata a luego de insertar la permutación p

      Mientras se pueda "avanzar" en el autómata consumiendo
      elementos de la permutación de entrada, avanzar.

      A partir del estado desde el cual no se pudo avanzar, crear
      los estados necesarios para albergar a lo que queda por
      consumir de la permutación.
```

Los autómatas que la repetición del algoritmo insert genera, tienen la siguiente forma:

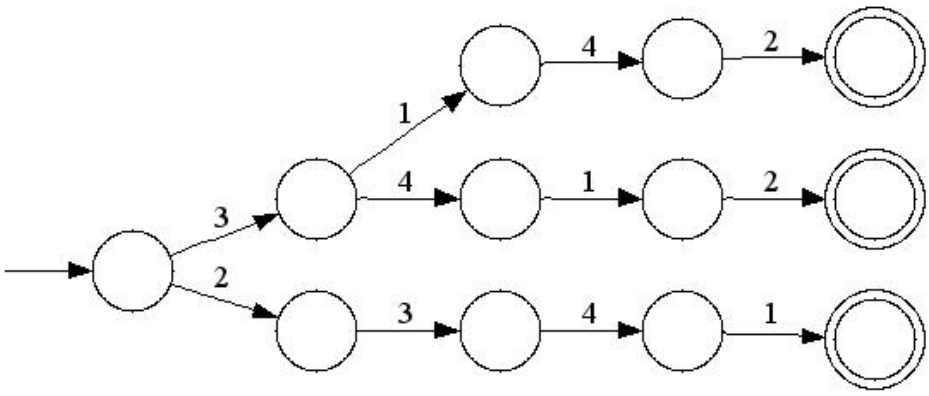


Figura 7.4: Autómata construido con el algoritmo *Insert* que acepta las cadenas 2 3 4 1, 3 4 1 2 y 3 1 4 2

Claramente este autómata no es mínimo. Es posible representar el mismo lenguaje con el siguiente autómata:

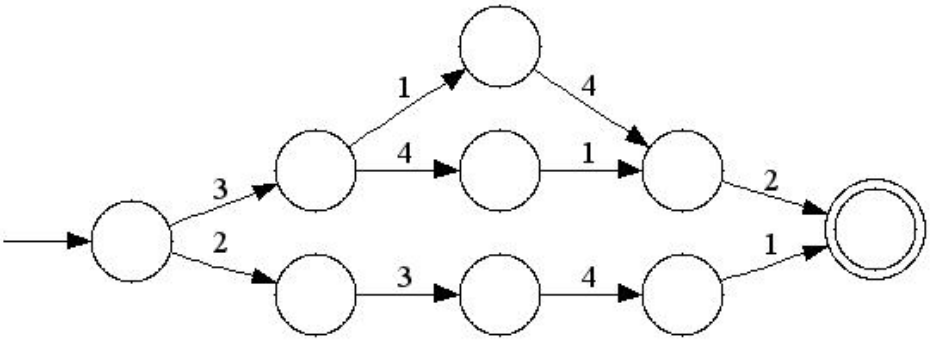


Figura 7.5: Autómata mínimo que acepta las cadenas 2 3 4 1, 3 4 1 2 y 3 1 4 2

Es posible, durante la construcción del autómata, totalizar y complementarlo a medida que se insertan las permutaciones que se quieren evitar. Esto redunda en un mayor costo de construcción de la estructura pero disminuye los tiempos totales ya que no es necesario volver a recorrerla más que durante el proceso de generación de las permutaciones.

Como podemos ver en la *Tabla 8.2* del Capítulo 8 (Resultados), con este enfoque se logra una sustancial disminución de los tiempos de generación respecto de los tiempos necesarios en la estrategia que utiliza tries, y una gran disminución del consumo de memoria tanto en comparación con la estrategia descripta en 7.2 como con respecto a la que utiliza tries.

4. Creaciones más inteligentes

Como vimos, el algoritmo de construcción de autómatas descripto en la sección anterior no genera autómatas mínimos.

La minimización del autómata generado permitiría reducir el espacio necesario para el mismo, pero no evitaría el tiempo de construcción redundante del autómata y añadiría el tiempo de minimización ($n \log n$ en el mejor de los casos [G73]).

Sería deseable intentar construir desde el inicio un autómata que se asemejara al autómata mínimo que no acepta el lenguaje que queremos evitar.

Si bien no hemos conseguido tal resultado, es posible plantear mejoras que nos acerquen a este ideal. Así como antes la inserción nos permitió colapsar prefijos comunes de las permutaciones, ahora intentaremos colapsar sufijos para así reducir el tamaño de la estructura y consecuentemente el espacio en memoria que esta ocupa.

El algoritmo de inserción de permutaciones en el autómata, recorre la estructura hasta que encuentra el primer elemento en el cual la permutación a insertar y el camino que está recorriendo, difieren. A partir de ese momento crea los estados y transiciones necesarias para albergar a la permutación que se está insertando.

Por lo tanto, en el caso en que dos permutaciones coincidan en sus últimos k elementos se construirán estados repetidos que contendrán la misma información. Una posible solución a este problema es no sólo recorrer la estructura desde el inicio hasta el momento en el cual los elementos difieren, sino también hacerlo desde el estado final (sólo es necesario tener un único estado final en el

autómata). De esta forma, sólo se insertarán los estados necesarios para representar la información que permite diferenciar la permutación que se está insertando de aquellas que ya se encuentran en el autómata.

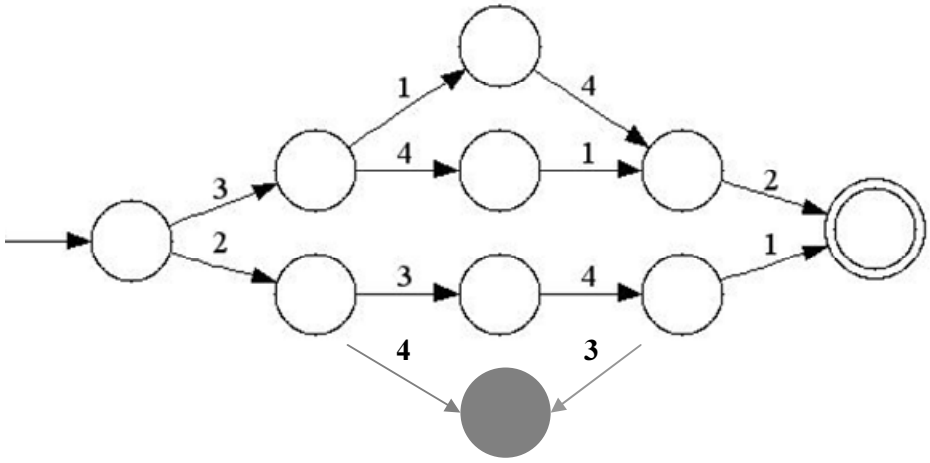
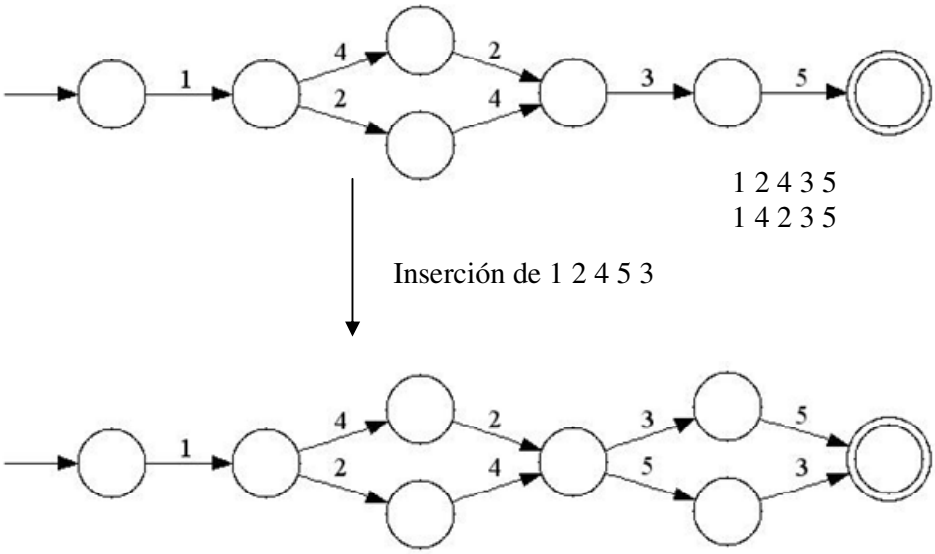


Figura 7.6: Algoritmo ingenuo con colapsado de sufijos
(Inserción de la permutación 2 4 3 1)

Desarrollar esta idea en forma ingenua puede provocar que cometamos errores que nos lleven a construir autómatas que no representen el conjunto de permutaciones deseado. Esto puede observarse en la Figura 7.7.



El autómata ahora acepta las permutaciones 1 2 4 3 5, 1 4 2 3 5, 1 2 4 5 3 y 1 4 2 5 3 (esta última nunca se quiso insertar)

Figura 7.7: Generación de permutaciones espúreas a través del uso de la implementación ingenua del algoritmo de inserción.

Puede observarse que el hecho de colapsar sufijos en forma maximal en cada inserción nos forzó la aparición de permutaciones espúreas.

Este problema surge del hecho de haber colapsado sufijos en forma prematura y con una observación local, siendo que la forma en la cual se comparten los sufijos sólo puede ser fruto de una observación global del conjunto de permutaciones que será insertado.

Discutiremos a continuación dos enfoques que pueden ser de utilidad en la resolución de este problema. El primero se apoya en la determinación *a priori* de la longitud de los sufijos que se compartirán; esto es, una vez que la inserción desde el estado inicial de una permutación produce una bifurcación, ésta sólo podrá compartir su sufijo con otra si este tiene exactamente la longitud prefijada. Se observa con relativa facilidad que la situación presentada en la Figura 7.7 ya no es posible, pues una vez que dos permutaciones difieren en un símbolo estas serán disjuntas hasta el eventual colapso de sus prefijos.

Por supuesto que esta estrategia de inserción no es óptima pues la correcta elección de la longitud de sufijos a compartir requiere el análisis de una cantidad factorial de permutaciones y no estamos dispuestos a invertir esa cantidad de tiempo, aun cuando esto implique que podamos obtener un autómata mínimo. La pregunta que cabe destacar es: de todos los valores que puede tomar la longitud de los sufijos a colapsar, ¿cuál de ellos es un buen candidato? La respuesta a esta pregunta cae por fuera del foco de esta tesis así que a los efectos prácticos de someter a prueba empírica la generación de permutaciones usando autómatas tomaremos una longitud de un carácter. Esto quiere decir que una vez producida una bifurcación entre dos permutaciones estas sólo podrán reunirse a lo sumo un estado antes del estado final del autómata. El argumento central de esta elección es que sólo por el hecho de compartir el estado final, el tamaño de la estructura resultante se reduce en promedio un 25% respecto de la que resulta del enfoque presentado anteriormente pues se elimina una cantidad (potencialmente) factorial de estados del autómata. Al mismo tiempo, compartir una arista permite colapsar los sufijos de todas aquellas permutaciones que terminan con la misma etiqueta. Hacer crecer la longitud del sufijo implicaría que se estarían generando n^2 sufijos diferentes y ya no es evidente si esto produce una mejora o un deterioro en los tiempos pues esto depende estrictamente del conjunto de permutaciones con las que estemos trabajando.

El segundo enfoque pretende explotar la posibilidad de despegar sufijos pegados previamente mientras se inserta una permutación a medida que esto sea necesario. Si observamos, en la Figura 7.6, el estado previo en la inserción de la permutación 1 2 4 5 3 podemos ver que la aparición de la permutación espúrea ocurre como consecuencia de avanzar por un prefijo pasando por sobre un estado de tiene más de un predecesor (una situación análoga se obtiene si nos encontramos retrocediendo desde el estado final y pasamos por sobre un estado que tiene más de un sucesor). Luego, podríamos pensar en "avanzar" siempre y cuando no ocurra esto, es decir, siempre que no "avancemos" por sobre un estado con más de un predecesor ("retrocedamos" por sobre un estado con más de un sucesor). Esto produce otro tipo de situaciones no deseadas, un ejemplo puede encontrarse en la Figura 7.8

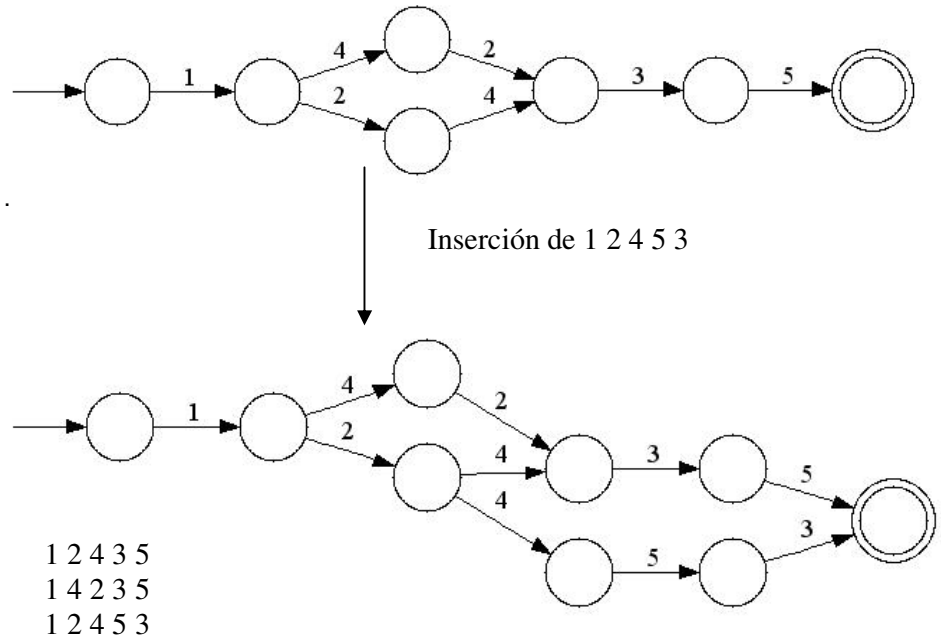


Figura 7.8: Introducción de no-determinismo a través de la inserción previa a un estado con más de un predecesor.

La introducción de no-determinismo no es una situación fácil de salvar, la opción obvia es aplicar el algoritmo de determinización de autómatas finitos, pero como ya hemos visto, este algoritmo tiene costo temporal exponencial en la cantidad de estados que conforman el autómata, que ya es un conjunto de cardinalidad exponencial. Una salida no tan obvia es la determinización constructiva del autómata. Esta estrategia implica la posibilidad de, a medida que se avanza por el prefijo maximal que deseamos compartir, proceder a "despegar" estados que habían sido determinados como equivalentes para aprovechar un mismo sufijo. Gráficamente sobre el caso anterior, sería proceder a insertar la permutación parcial 5 6 a partir del estado distinguido, luego de haber avanzado el estado en el cuál se produce el colapso del sufijo (ver Figura 7.9)

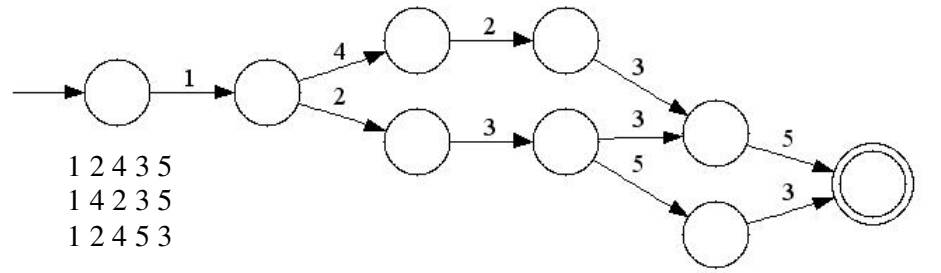


Figura 7.9: Avance del punto de colapso de sufijos frente a la aparición de no-determinismo e inserción de permutaciones parciales.

Se observa que esta operatoria subsana la aparición de no-determinismo a través de la determinización en línea del autómata. El problema que esta estrategia presenta es el hecho de que cada inserción obliga la eventual replicación de todo el autómata restante, provocando, en consecuencia, un crecimiento en su tamaño que depende estrictamente del conjunto de permutaciones a insertar y un aumento sensible del tiempo necesario para su construcción. Si se observa el comportamiento del algoritmo y cómo se produce la replicación de estados para eliminar el no-determinismo introducido por esta forma de inserción es posible vislumbrar un paulatino acercamiento a la estructura resultante de utilizar el enfoque presentado anteriormente pero pagando en tiempo de ejecución la eventual optimización en la utilización del espacio necesario para almacenar el autómata.

A continuación presentamos el algoritmo de inserción implementado, que aprovecha los prefijos maximales de las permutaciones a insertar y los sufijos de longitud uno.

Algoritmo Insert

```
INPUT: p, permutación a insertar en el autómata; a, autómata
OUTPUT: el autómata a luego de insertar la permutación p

       Mientras se pueda "avanzar" en el autómata desde el estado
       inicial consumiendo elementos de la permutación de entrada,
       avanzar.

       Si se puede "retroceder" en el autómata desde el estado final
       consumiendo el último elemento de la permutación de entrada,
       retroceder.

       Crear los estados y transiciones necesarios para representar
       la permutación a insertar entre los estados a partir de los
       cuales no se pudo avanzar ni retroceder.
```

Algoritmo de generación

```
INPUT: conjunto de permutaciones a no generar.
OUTPUT: el complemento del conjunto de permutaciones dadas como
        INPUT.

       Lectura del conjunto de permutaciones de entrada.

       Construcción de un estado inicial, uno final y un estado
       trampa

       Por cada permutación insertarla en el autómata con el
       algoritmo Insert. Totalizar y complementar cada estado que se
       construye.

       Lectura del autómata complementado para generar las
       permutaciones válidas que acepta el autómata.
```

A pesar de que el algoritmo de inserción con colapsado de sufijos presentado en la Figura 7.6 tiene los inconvenientes previamente discutidos, de cualquier forma es importante destacar que el mismo efectivamente reduce de forma significativa el tamaño del autómata. Tanto así, que el autómata resultante es mínimo, como se demostrará en el Teorema 7.1 para el cual introducimos la siguiente definición.

Definición 7.1 (Algoritmo de inserción con colapsado de sufijos)

Sea T un autómata determinístico y mínimo, $(\Sigma, Q, q_0, \delta, \{q_f\})$.

Sea q_0 el estado inicial del autómata, q_f su estado final y $p = xaAa_k x'$ con $x, A, x' \in \Sigma^*$ y $a, a_k \in \Sigma$ la permutación a insertar. Suponemos que p tiene como mínimo dos elementos (el caso de permutaciones de dominios de cardinal uno no tiene demasiado interés)

El algoritmo de inserción crea estados $q_1, q_2, ..., q_k$ tales que:

a) $\delta(q_i, a) = q_1$

b) $\tilde{\delta}(q_1, A) = q_k$

c) $\delta(q_k, a_k) = q_j$

si $\exists\, q_i, q_j \in Q$ tales que:

1. $\tilde{\delta}(q_0, x) = q_i$ y $\neg\exists q' \in Q / \tilde{\delta}(q_0, xa) = q'$
2. $\tilde{\delta}(q_j, x') = q_f$ y $\neg\exists q'' \in Q / \tilde{\delta}(q'', a_k x') = q_f$

Lema 7.1

Sea T el autómata $(\Sigma, Q, q_0, \delta, \{q_f\})$ obtenido con las sucesivas aplicaciones del algoritmo *insert*.

$$\forall y \in \Sigma^*\ \forall s_1, s_2 \in Q \left(\tilde{\delta}(s_1, y) = q_f \wedge \tilde{\delta}(s_2, y) = q_f \Rightarrow s_1 = s_2\right)$$

Dem:

Por el absurdo. Supongamos que no.

$$\exists\, s_1, s_2 \in Q\, \exists y \in \Sigma^* \left(\tilde{\delta}(s_1, y) = q_f \wedge \tilde{\delta}(s_2, y) = q_f \wedge s_1 \neq s_2\right)$$

Caso 1: s_1 y s_2 se construyeron en ejecuciones distintas del algoritmo de inserción con colapsado de sufijos, o sea, insertando permutaciones distintas.

Sea $p_1 = x_1 a_1 A_1 a_{k1} x'_1$ la permutación que se insertó en la ejecución del algoritmo que creó a s_1 y $p_2 = x_2 a_2 A_2 a_{k2} x'_2$ la permutación que se insertó en la ejecución del algoritmo que creó a s_2.

Sin pérdida de generalidad, asumiremos que s_1 se construyó antes que s_2.

Por hipótesis $\exists y \in \Sigma^* / \tilde{\delta}(s_2, y) = q_f$. Entonces $y = zx'_2$ con z sufijo de $a_2 A_2 a_{k2}$ tal que:

$$\exists q_j / \tilde{\delta}(q_j, x'_2) = q_f \wedge \neg\exists q'' / \tilde{\delta}(q'', a_{k2} x'_2) = q_f \quad (1)$$

(por la *Definición 7.1)*

s_2 se está creando para almacenar una parte de la cadena $a_2 A_2 a_{k2}$ Por cada permutación que se inserta, es necesario crear como mínimo un estado

(como se deduce de la *Definición 7.1*), necesario para albergar a dos caracteres (que es en lo mínimo que pueden diferir dos permutaciones. z representa la parte de la permutación nueva que se está insertando a partir de s_2 y por lo tanto z no puede ser nulo.

pero por hipótesis también $\tilde{\delta}(s_1, y) = q_f$, o sea $\tilde{\delta}(s_1, z'a_{k2}x'_2) = q_f$ (ya que $y = zx'_2$ con z sufijo de $a_2A_2a_{k2}$) $\Rightarrow \exists q_m / \tilde{\delta}(s_1, z') = q_m \quad \wedge \quad \tilde{\delta}(q_m, a_{k2}x'_2) = q_f$ lo que contradice a (1). ♦

Caso 2: s_1 y s_2 se construyeron en la misma ejecución del algoritmo de inserción con colapsado de sufijos

Sea $p = xaAa_kx'$ la permutación que se insertó en la ejecución algoritmo de inserción con colapsado de sufijos que creó a s_1 y a s_2.

Por la especificación del algoritmo de inserción con colapsado de sufijos, s_1 y s_2 son algunos de los estados $q_1, q_2, ..., q_k$ tales que $\delta(q_i, a) = q_1$, $\tilde{\delta}(q_1, A) = q_k$ y $\delta(q_k, a_k) = q_j$ con $q_i, q_j \in Q$ tales que:

$\tilde{\delta}(q_0, x) = q_i$ y $\neg\exists q' \in Q / \tilde{\delta}(q_0, xa) = q'$

$\tilde{\delta}(q_j, x') = q_f$ y $\neg\exists q'' \in Q / \tilde{\delta}(q'', a_kx') = q_f$

Por lo tanto $\tilde{\delta}(s_1, zx') = q_f$ y $\tilde{\delta}(s_2, z'x') = q_f$ con z, z' sufijos de aAa_k. Dado que el algoritmo de inserción del Caso 1 mantiene el determinismo, $z \neq z'$. Pero por hipótesis $\tilde{\partial}(s_1, y) = q_f$ y $\tilde{\partial}(s_2, y) = q_f \Rightarrow \quad z = y \quad \wedge \quad z' = y$ (por ser determinístico) $\Rightarrow z = z'$. ♦

Como se vio en la *Definición 6.3* un autómata finito es mínimo si no contiene dos estados indistinguibles. O sea,

$$\forall q_1, q_2 \in Q, \left(\forall x \in \Sigma^*, \left(\tilde{\delta}(q_1, x) \subset F \Leftrightarrow \tilde{\delta}(q_2, x) \subset F\right)\right) \Rightarrow q_1 = q_2$$

Teorema 7.1

El autómata que se obtiene por las sucesivas aplicaciones del Algoritmo de inserción con colapsado de sufijos es mínimo.

Demostración:

Supongamos que no, por lo tanto existen dos estados indistinguibles.

$$\exists q_1, q_2 \in Q, \left(\forall x \in \Sigma^*, \left(\tilde{\delta}(q_1, x) \subset F \Leftrightarrow \tilde{\delta}(q_2, x) \subset F\right)\right) \wedge q_1 \neq q_2$$

O sea, $\exists q_1, q_2 \in Q$,

$$\left(\forall x \in \Sigma^* \left(\left(\tilde{\delta}(q_1, x) = q_f \wedge \tilde{\delta}(q_2, x) = q_f\right) \vee \left(\tilde{\delta}(q_2, x) \neq q_f \wedge \tilde{\delta}(q_1, x) \neq q_f\right)\right)\right) \wedge q_1 \neq q_2$$

o sea, $\exists q_1, q_2 \in Q$,

$$\left(\forall x \in \Sigma^* \left(\left(\tilde{\delta}(q_1, x) = q_f \wedge \tilde{\delta}(q_2, x) = q_f \wedge q_1 \neq q_2\right) \vee \left(\tilde{\delta}(q_2, x) \neq q_f \wedge \tilde{\delta}(q_1, x) \neq q_f \wedge q_1 \neq q_2\right)\right)\right) \quad (2)$$

Pero por construcción del autómata q_f es alcanzable desde cualquier estado del autómata,

$$\forall q_1 \in Q \ \ \exists x \in \Sigma^* / \tilde{\delta}(q_1, x) = q_f \quad (3)$$

Entonces, de (2) y de (3) se obtiene

$$\exists q_1, q_2 \in Q, \exists x \in \Sigma^* \left(\tilde{\delta}(q_1, x) = q_f \wedge \tilde{\delta}(q_2, x) = q_f \wedge q_1 \neq q_2 \right)$$

lo que es absurdo por *Lema 7.1.* ♦

Si bien el teorema 7.1 no resulta de aplicación inmediata en el proceso de inserción de permutaciones en el autómata (por lo previamente discutido), el mismo abre una nueva línea de trabajo que consiste en la búsqueda de modificaciones del algoritmo que permitan, cuando es posible, el colapsado de sufijos. En ese sentido, ya hay algunos resultados preliminares de algunas situaciones en las que tal colapsado puede ser realizado, y son el objeto de trabajo futuro.

8. Resultados

A continuación mostramos los resultados que se obtuvieron comparando las dos estrategias: la generación usando tries y el autómata mínimo descripto en la última sección del capítulo anterior.

También describimos el algoritmo de generación de los casos de test usados para la comparación.

1. Generación de los casos de tests

Los casos de test usados para la comparación consisten en un conjunto de permutaciones generadas aleatoriamente.

Para cada tamaño de dominio se generaron cuatro conjuntos de permutaciones de distinto tamaño:

Tamaño del dominio	Cantidad de permutaciones (10%)	Cantidad de permutaciones (25%)	Cantidad de permutaciones (40%)	Cantidad de permutaciones (65%)	Cantidad de permutaciones (80%)
8	4032	10080	16128	26208	32256
9	36288	90720	145152	235872	290304
10	362880	907200	1451520	2358720	3628800
11	3991680	9979200	--	--	--

No se generaron casos de test de un dominio menor a ocho ya que los tiempos de generación eran despreciables para esos casos. Tampoco para dominios mayores a once ni para un porcentaje mayor al 25% de las permutaciones de tamaño once, ya que en esos casos la cantidad de memoria necesaria la generación era mayor a la que se contaba y por lo tanto los tiempos no eran confiables debido a los accesos a disco.

Dado que todavía no se tiene información respecto de la presencia o ausencia de patrones en los automorfismos de las asignaciones de las variables relacionales contenidas en casos de estudio típicos para la herramienta, se decidió generar aleatoriamente las permutaciones usadas en los casos de test. De esta forma se evita la degeneración de la estructura de almacenamiento de las mismas que podría inducir patrones no deseados en los tiempos de generación del complemento.

Algoritmo de Generación

```
INPUT: d (tamaño del dominio),
```

```
         qty (cantidad de permutaciones a generar)
OUTPUT:  conjunto de qty permutaciones de dominio d generadas
         aleatoriamente

       Mientras el caso de test tenga menos de qty permutaciones
       Para cada permutación  p generada
             Determinar incluir p en el caso de test con
             una probabilidad de qty/d!
       finMientras
```

De esta manera se generan todas las permutaciones de determinado dominio y se eligen al azar la cantidad de permutaciones que es necesario almacenar.

Una vez generados los casos de tests, se usaron los mismos conjuntos de permutaciones como entrada de las dos estrategias.

Exceptuando la comparación final de tiempos totales, y de estrategias anteriores se compararon los tiempos de las dos estrategias en los casos en los cuales podían obtenerse datos para ambas. Por lo tanto en varias de las comparaciones que se muestran a continuación no se contemplan aquellos casos de test que tienen más del 40% de las permutaciones de tamaño 10 ya que la estrategia que utiliza tries comienza a utilizar más memoria de la disponible, produciendo un incremento sustancial del tiempo de generación debido al acceso a disco.

En todos los casos se usó un procesador AMD Athlon 3200+ de 64 bits con 2 Gb de memoria RAM dual channel, con Mandriva Linux para procesadores de 64 bits. Versión del kerner: Linux 2.6.11-6mdk

1. Comparaciones de las estrategias anteriores

Comparaciones de tiempo y memoria usando la estrategia descripta en la Sección 7.2

	Tiempo (seg)		Memoria (Mb)	
	Trie	Autómata	Trie	Autómata
10%	0.1	0.08	6.6	13
25%	0.11	0.09	12	13
40%	0.14	0.09	18	13
65%	0.18	0.1	27	13
80%	0.2	0.1	32	13

Tabla 8.11: Dominio de tamaño 8

	Tiempo (seg)		Memoria (Mb)	
	Trie	Autómata	Trie	Autómata
10%	0.89	0.89	50	130
25%	1.14	0.96	123	132
40%	1.39	1.02	176	132
65%	1.75	1.12	256	133
80%	1.98	1.18	287	138

Tabla 8.12: Dominio de tamaño 9

	Tiempo (seg)		Memoria (Mb)	
	Trie	Autómata	Trie	Autómata
10%	9.67	10.02	489	1400
25%	10.96	10.04	1020	1400
40%	13.03	10.26	1400	1500
65%	trash	10.86	trash	1500
80%	trash	11.16	trash	1500

Tabla 8.13: Dominio de tamaño 10

Comparaciones de tiempo y memoria usando la estrategia descripta en la Sección 7.3

	Tiempo (seg)		Memoria (Mb)	
	Trie	Autómata	Trie	Autómata
10%	0.1	0.04	6.6	2.96
25%	0.11	0.05	12	5.14
40%	0.14	0.06	18	7.06
65%	0.18	0.08	27	9.95
80%	0.2	0.09	32	11

Tabla 8.21: Dominio de tamaño 8

	Tiempo (seg)		Memoria (Mb)	
	Trie	Autómata	Trie	Autómata
10%	0.89	0.35	50	19
25%	1.14	0.52	123	41
40%	1.39	0.68	176	61
65%	1.75	0.92	256	90
80%	1.98	1.04	287	106

Tabla 8.22: Dominio de tamaño 9

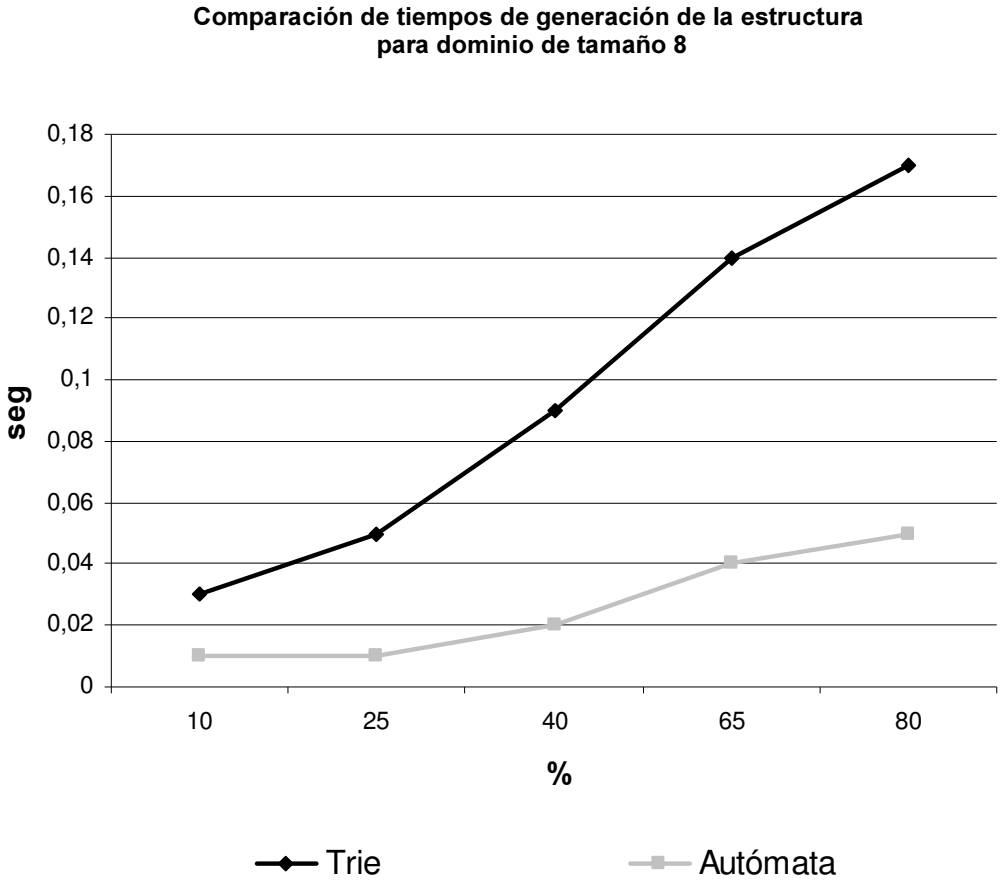
	Tiempo (seg)		Memoria (Mb)	
	Trie	Autómata	Trie	Autómata
10%	9.67	3.79	489	213
25%	10.96	4.59	1020	357
40%	13.03	5.95	1400	570
65%	trash	8.22	trash	923
80%	trash	9.65	trash	1100

Tabla 8.23: Dominio de tamaño 10

3. Comparaciones finales

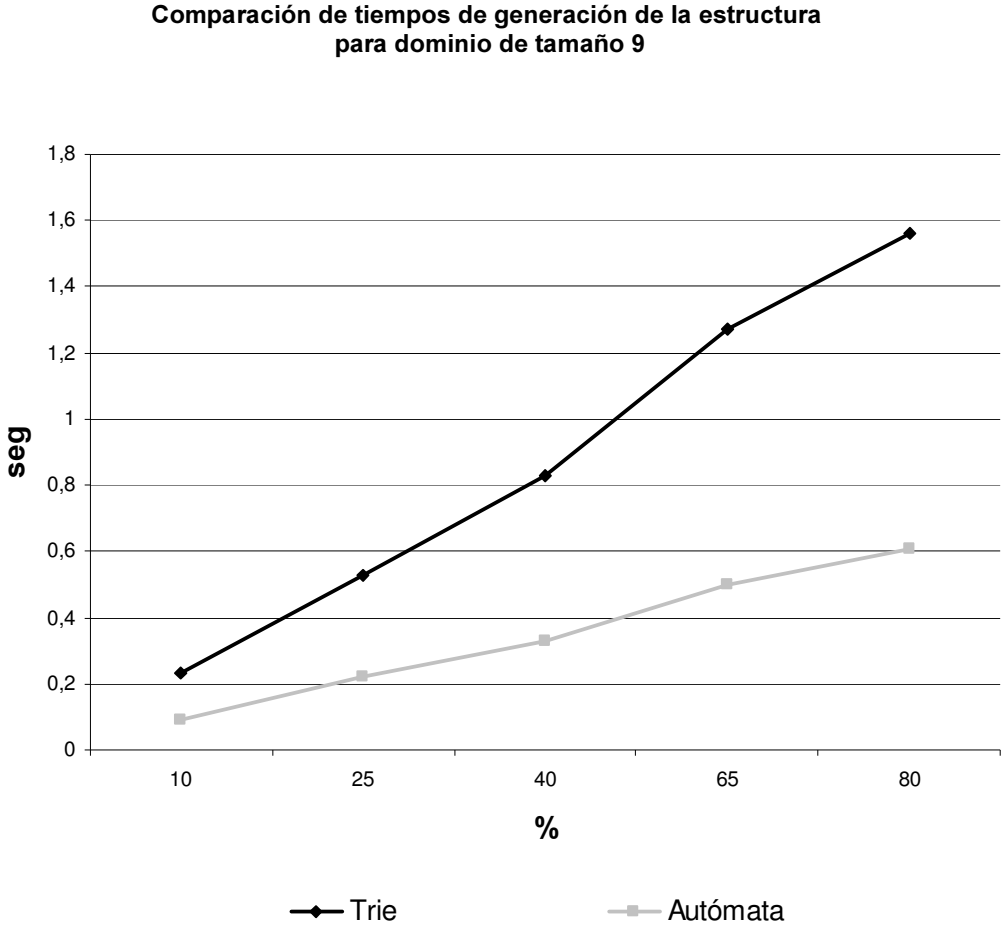
Tiempos de Generación de la estructura

Se compararon los tiempos que tardan la estrategia que utiliza tries y la estrategia descripta en la Sección 7.4 en generar la estructura necesaria para almacenar las permutaciones a evitar. En la primera estrategia se contempla el tiempo necesario para insertar en el trie todas las permutaciones del caso de test correspondiente, mientras que en la segunda, se tomó el tiempo que se tarda en crear el autómata mediante el último algoritmo mencionado (creación, totalización y complemento).



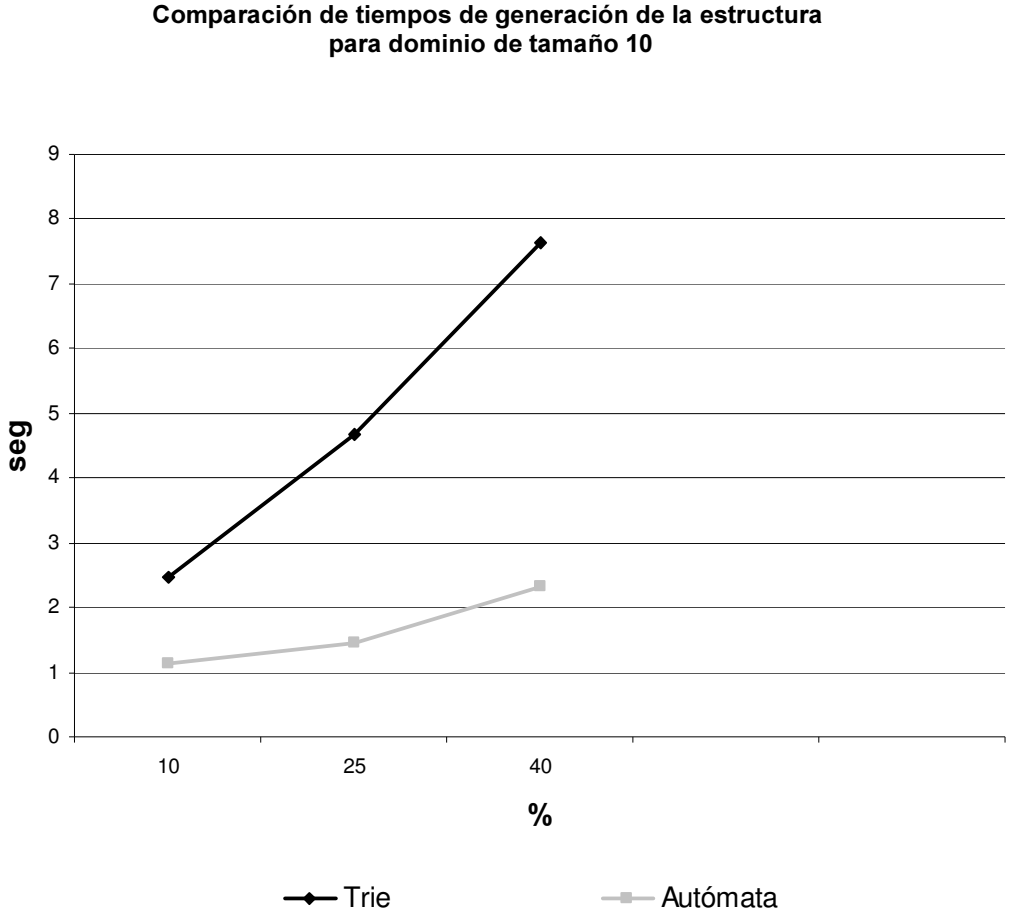
	Tiempo (seg)	
	Trie	*Autómata*
10%	0,03	0,01
25%	0,05	0,01
40%	0,09	0,02
65%	0,14	0,04
80%	0,17	0,05

Tabla 8.31: Dominio de tamaño 8



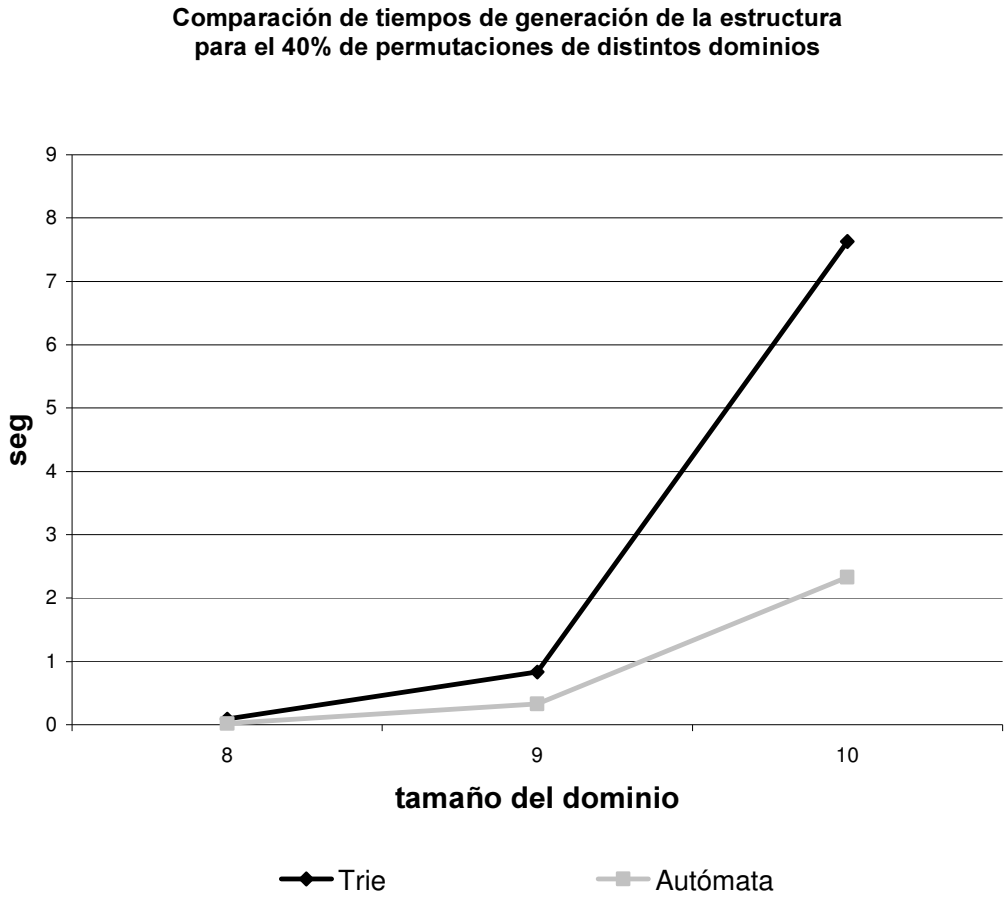
	Tiempo (seg)	
	Trie	*Autómata*
10%	0,23	0,09
25%	0,53	0,22
40%	0,83	0,33
65%	1,27	0,5
80%	1,56	0,61

Tabla 8.32: Dominio de tamaño 9



	Tiempo (seg)	
	Trie	*Autómata*
10%	2,47	1,14
25%	4,66	1,45
40%	7,63	2,33

Tabla 8.33: Dominio de tamaño 10

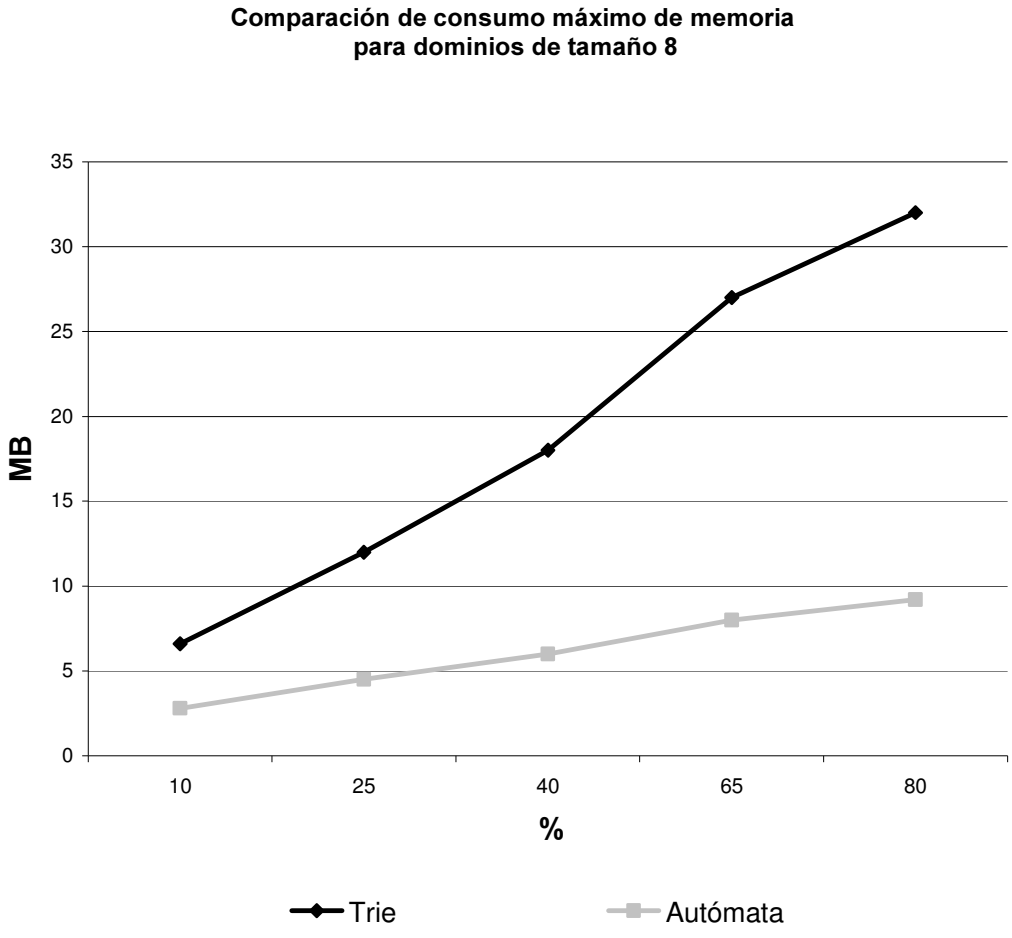


	Tiempo (seg)	
	Trie	*Autómata*
Dom 8	0,09	0,02
Dom 9	0,83	0,33
Dom 10	7,63	2,33

Tabla 8.34: 40 % de permutaciones para distintos dominios

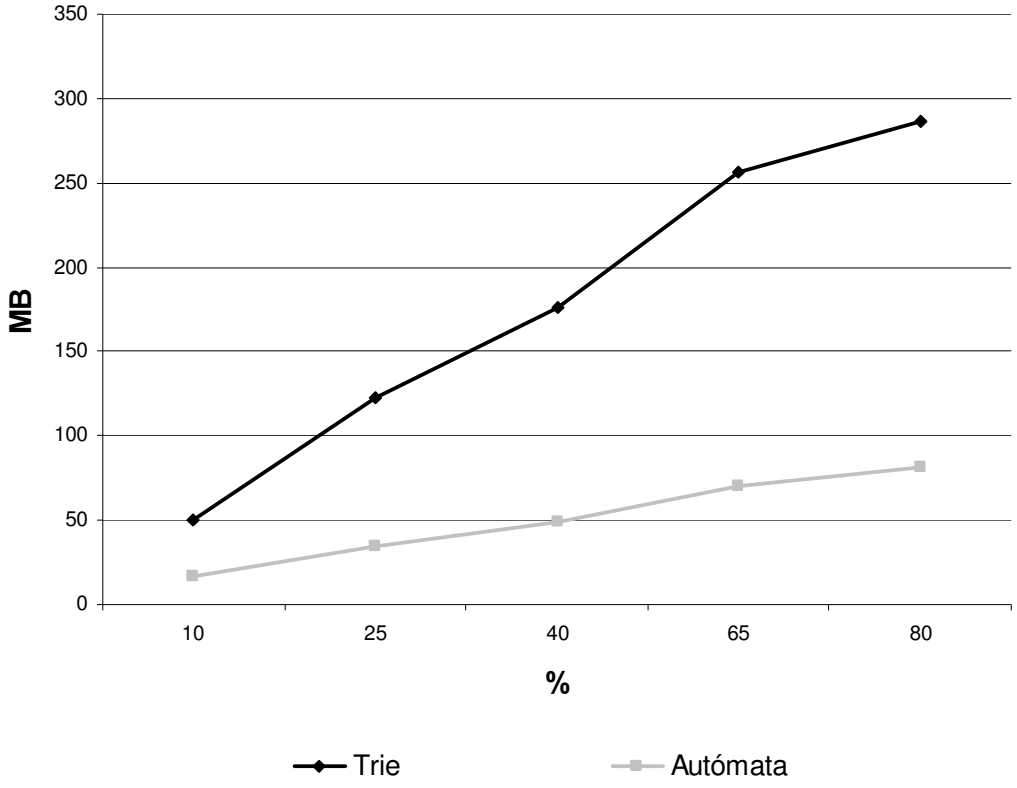
Como podemos ver en los datos que tenemos para los distintos dominios, la creación del autómata es como mínimo dos veces más rápida en todos los casos y parece ser esperable una diferencia aún mayor en dominios más grandes.

Consumo máximo de memoria



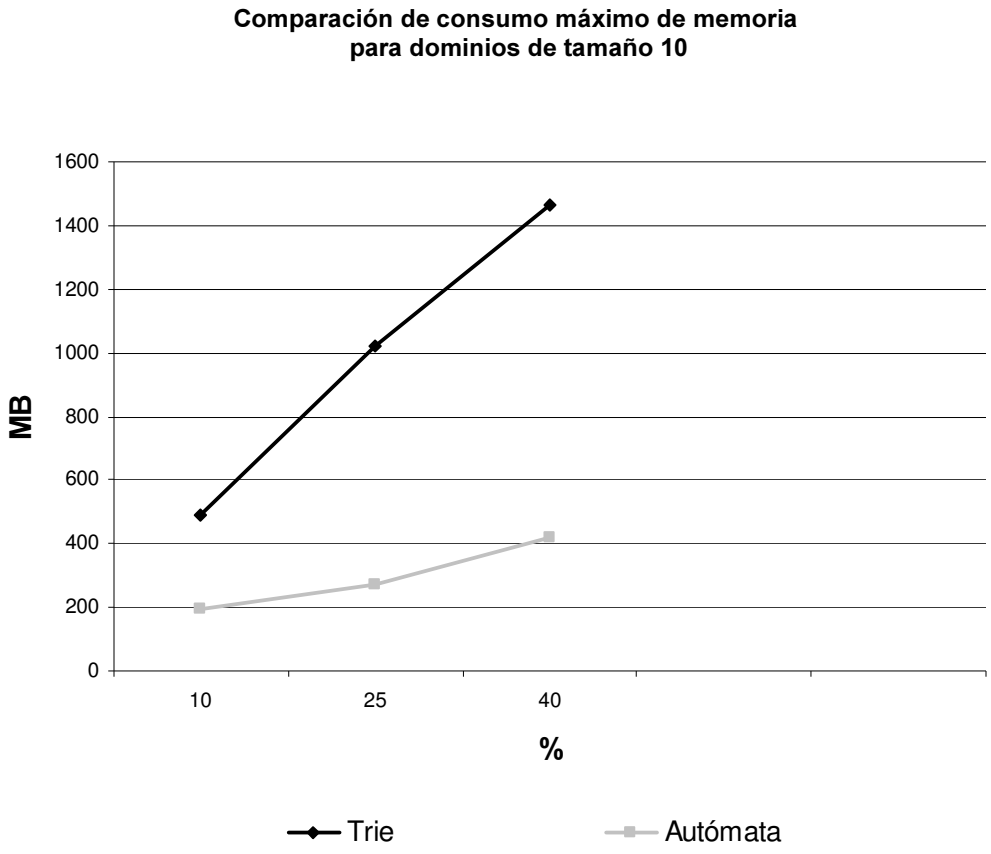
	Memoria (Mb)	
	Trie	*Autómata*
10%	6,6	2,8
25%	12	4,5
40%	18	6
65%	27	8
80%	32	9,2

Tabla 8.41: Dominio de tamaño 8



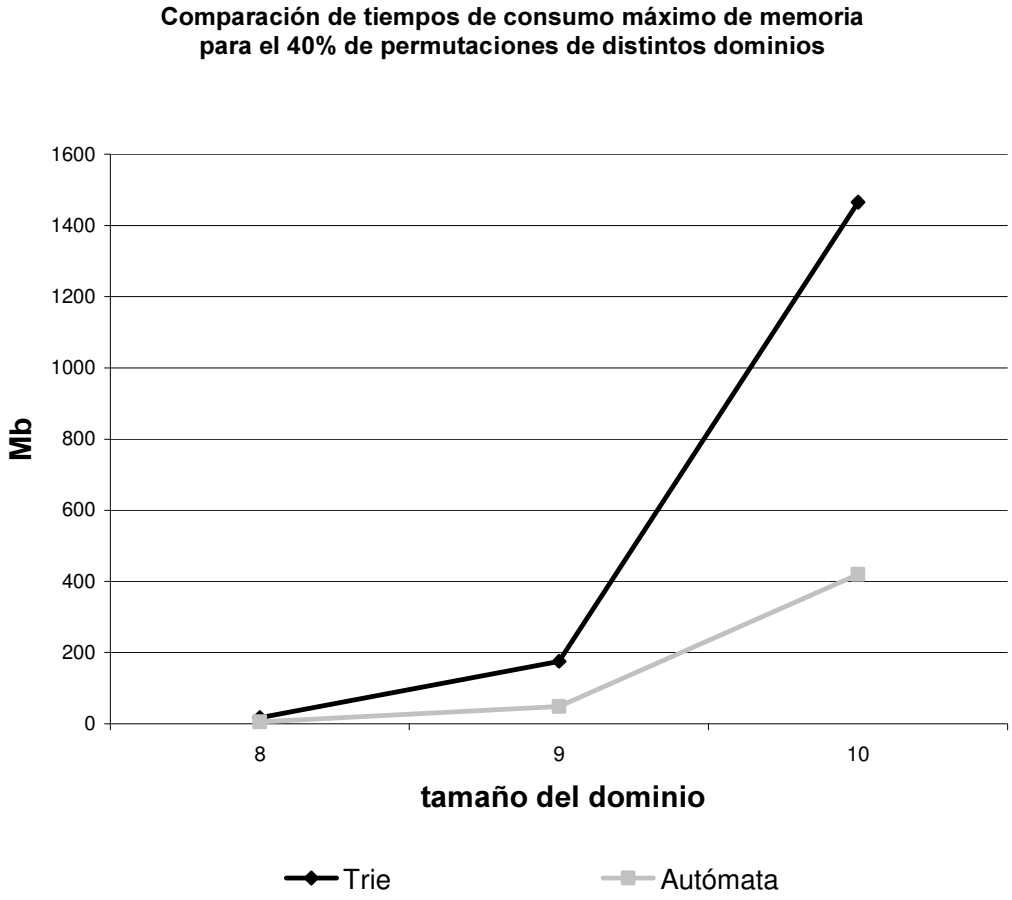
	Memoria (Mb)	
	Trie	*Autómata*
10%	50	17
25%	123	35
40%	176	49
65%	256	70
80%	287	81

Tabla 8.42: Dominio de tamaño 9



	Memoria (Mb)	
	Trie	*Autómata*
10%	489	195
25%	1020	272
40%	1465	420

Tabla 8.43: Dominio de tamaño 10



	Tiempo (seg)	
	Trie	*Autómata*
Dom 8	18	6
Dom 9	176	49
Dom 10	1465	420

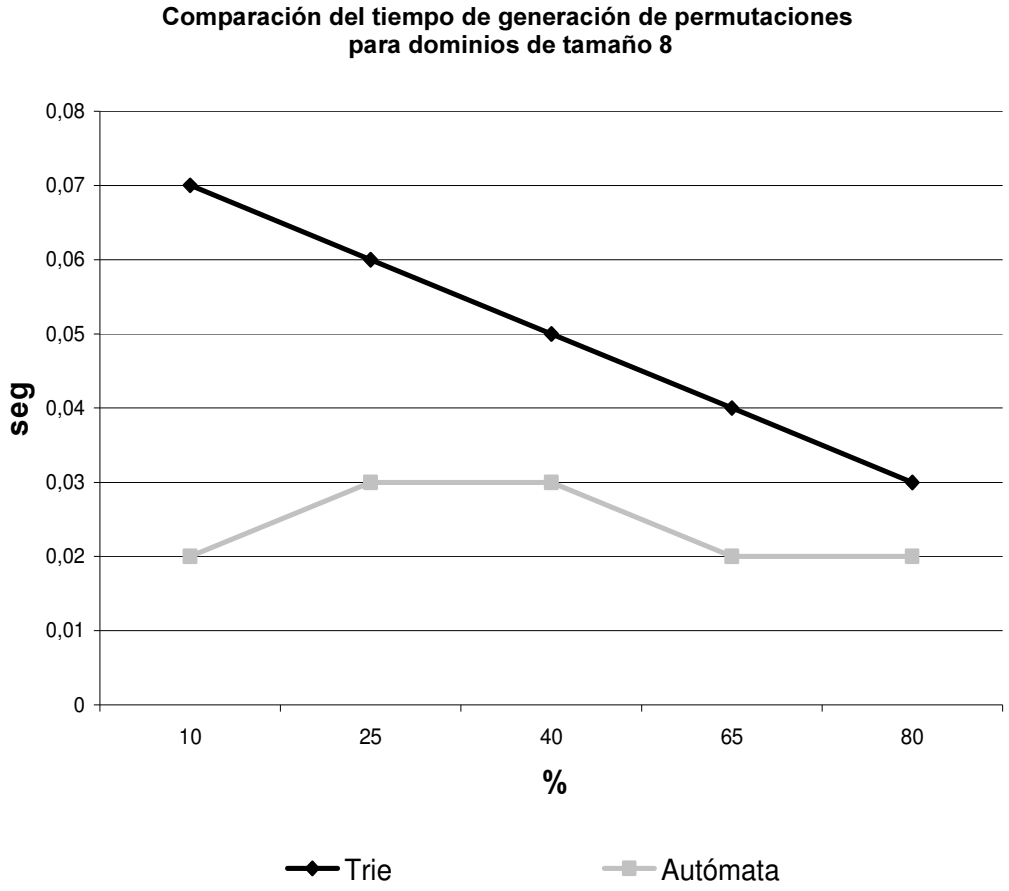
Tabla 8.44: 40 % de permutaciones para distintos dominios

Según los datos que se observan, el consumo máximo de memoria de la estrategia que usa autómatas como estructura es como mínimo tres veces menor a la que usa el trie, con una diferencia un poco mayor en los casos en los cuales, para un dominio dado, es necesario evitar una mayor cantidad permutaciones. Esto permite suponer que la eficiencia del autómata como estructura de almacenamiento es mayor a medida que se aumenta la cantidad de permutaciones a almacenar.

Tiempo de generación de permutaciones

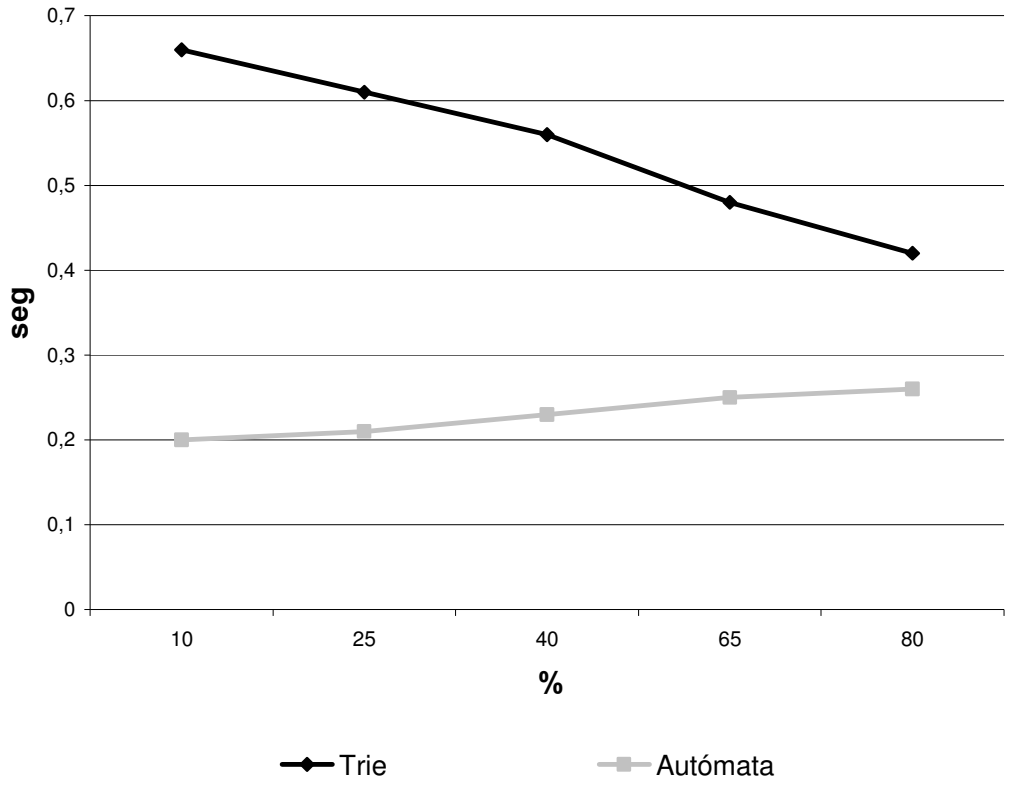
Una vez creadas las estructuras, se compararon los tiempos de generación de las permutaciones en cada una de las estrategias. En la primera, este tiempo contempla la generación de todas las permutaciones de un dominio, la determinación para cada una de ellas, de su pertenencia al trie y en caso de que no pertenezcan, el tiempo que se consume en el pasaje del formato cíclico al formato cartesiano.

En el caso de la estrategia que usa al autómata, este tiempo contempla la ejecución del algoritmo que lo recorre generando las cadenas que pertenecen al lenguaje que es aceptado por el autómata y que además son permutaciones válidas para el dominio especificado.



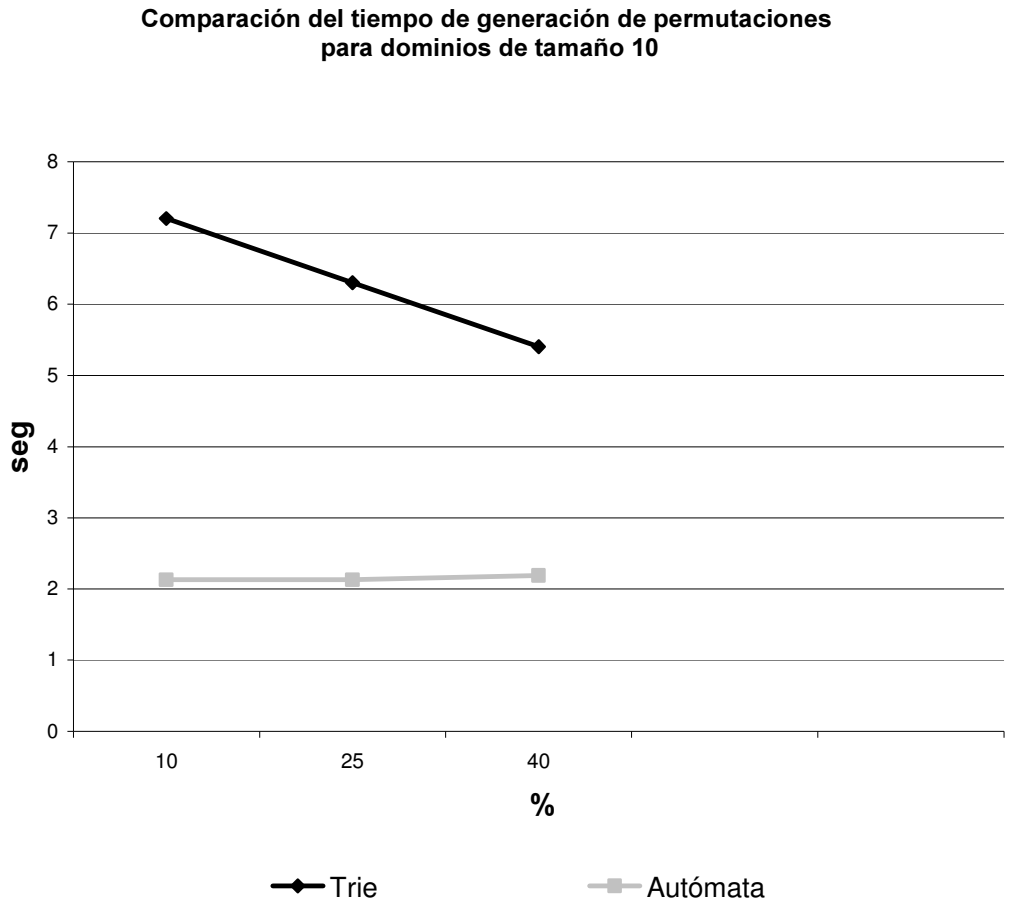
	Tiempo (seg)	
	Trie	*Autómata*
10%	0,07	0,02
25%	0,06	0,03
40%	0,05	0,03
65%	0,04	0,02
80%	0,03	0,02

Tabla 8.51: Dominio de tamaño 8



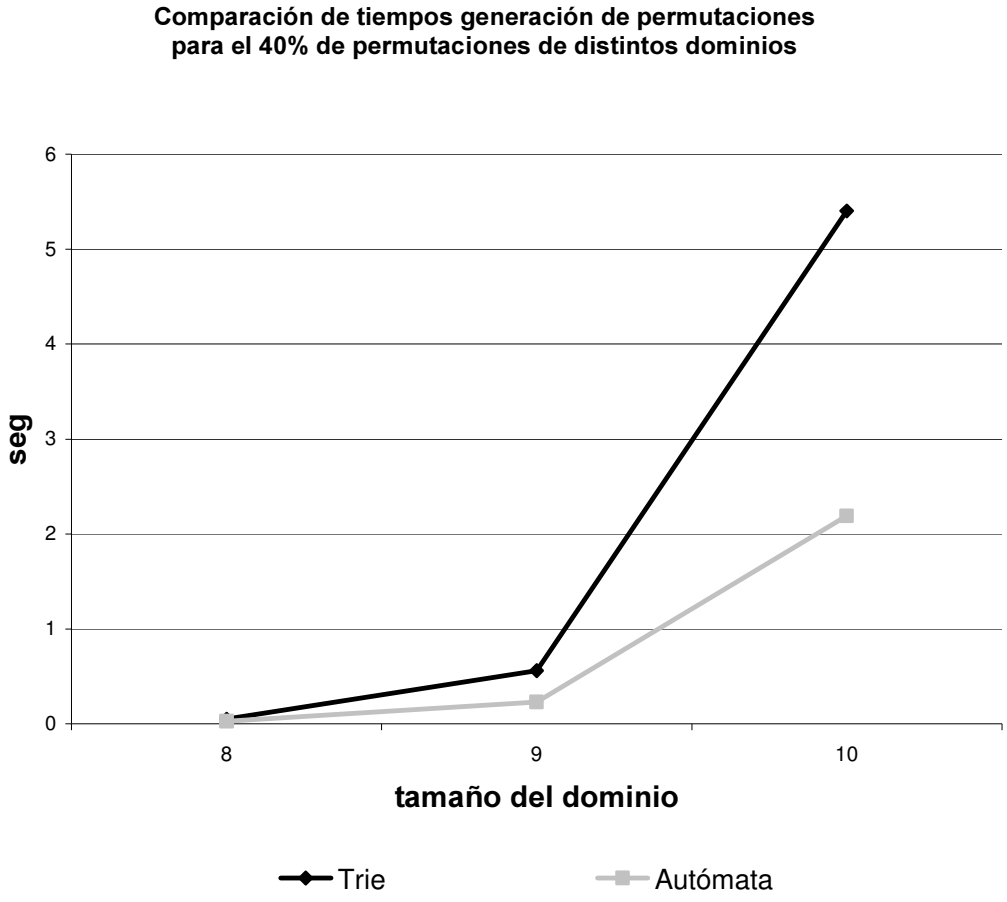
	Tiempo (seg)	
	Trie	*Autómata*
10%	0,66	0,2
25%	0,61	0,21
40%	0,56	0,23
65%	0,48	0,25
80%	0,42	0,26

Tabla 8.52: Dominio de tamaño 9



	Tiempo (seg)	
	Trie	*Autómata*
10%	7,2	2,13
25%	6,3	2,13
40%	5,4	2,19

Tabla 8.53: Dominio de tamaño 10



	Tiempo (seg)	
	Trie	*Autómata*
Dom 8	0,05	0,03
Dom 9	0,56	0,23
Dom 10	5,4	2,19

Tabla 8.54: 40 % de permutaciones para distintos dominios

Podemos ver que la estrategia que utiliza el autómata requiere una cantidad de tiempo prácticamente constante, una vez especificado el dominio, para la generación de las permutaciones. Esto se debe a que la parte de la estructura que debe recorrerse para determinar las permutaciones válidas del lenguaje que acepta el autómata es la misma independientemente de la cantidad de éstas que el mismo acepte.

Por otro lado, si bien sería lógico suponer que la estrategia que utiliza tries también debería mantener constante el tiempo de generación una vez fijado el dominio, dado que es necesario convertir cada permutación que se genera nuevamente al formato cartesiano, existe un costo adicional por cada permutación que efectivamente se devuelve, lo que explica la reducción del tiempo de generación a medida que la cantidad de permutaciones que efectivamente se devuelven disminuye. El hecho de que el trie contenga más permutaciones no redunda en un tiempo mayor para determinar la pertenencia al mismo ya que no

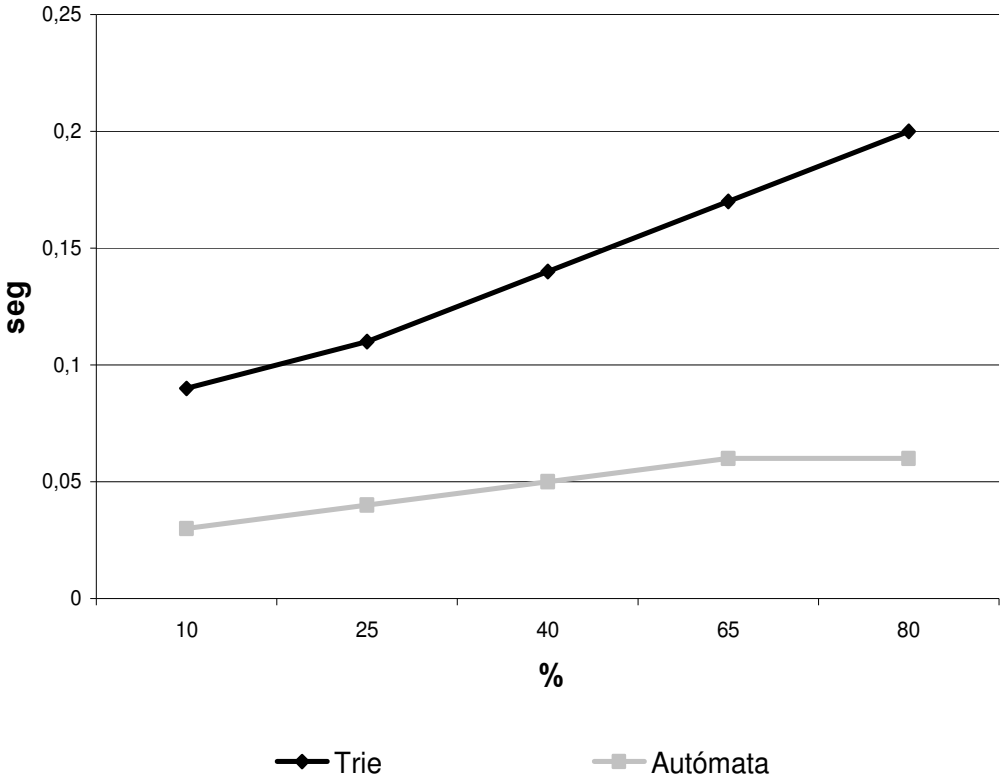
importa cuán grande la estructura sea, la pertenencia a la misma se determina en un tiempo que está en O(d), siendo d la cantidad de elementos del dominio.

Esto nos permite ver que la reducción del espacio utilizado que se obtiene por el uso de la notación de ciclos tiene su contrapartida en el costo temporal. Podría ser interesante analizar hasta qué punto conviene utilizar la notación de ciclos en lugar de la cartesiana en esta estrategia.

Tiempos totales

A continuación se muestran los tiempos correspondientes tanto a la construcción de la estructura como a la generación de permutaciones que deben evitarse:

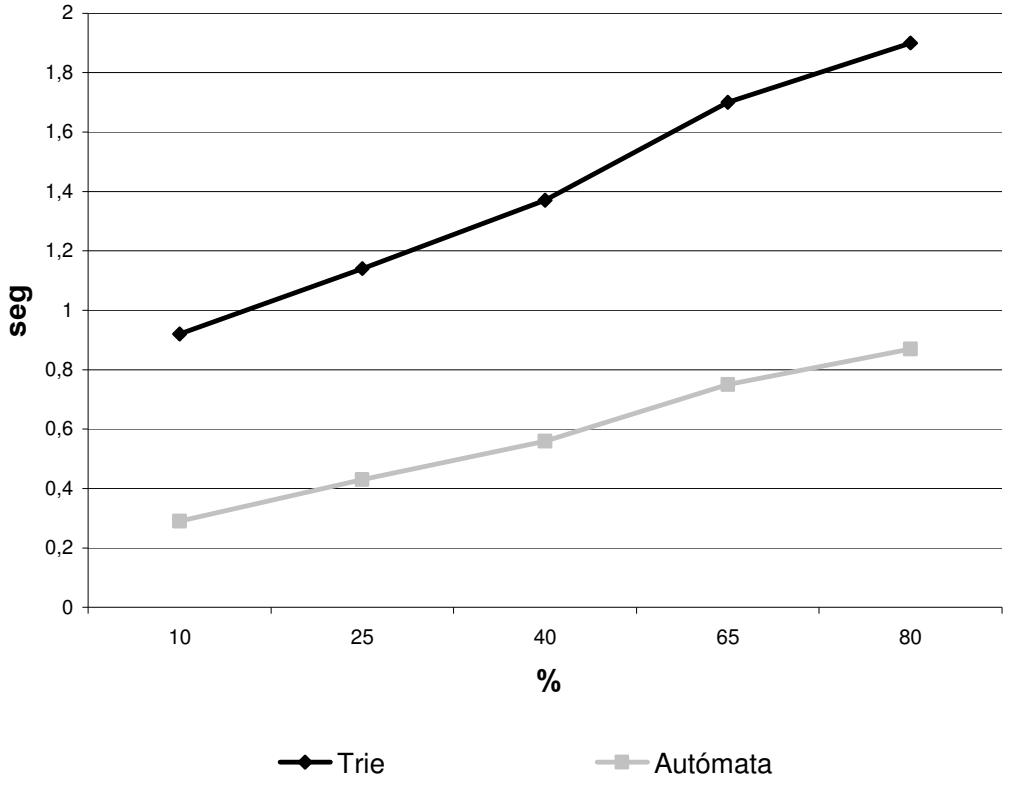
Comparación del tiempo total de generación para dominios de tamaño 8



	Tiempo (seg)	
	Trie	*Autómata*
10%	0,09	0,03
25%	0,11	0,04
40%	0,14	0,05
65%	0,17	0,06
80%	0,2	0,06

Tabla 8.61: Dominio de tamaño 8

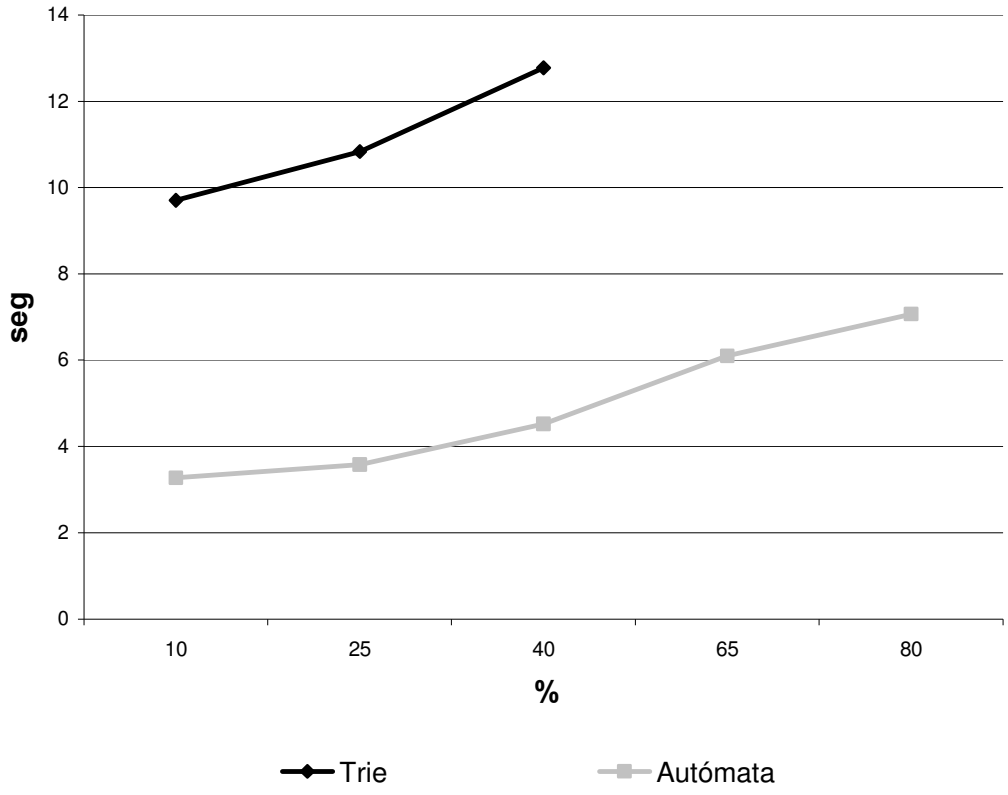
Comparación del tiempo total de generación para dominios de tamaño 9



	Tiempo (seg)	
	Trie	*Autómata*
10%	0,92	0,29
25%	1,14	0,43
40%	1,37	0,56
65%	1,7	0,75
80%	1,9	0,87

Tabla 8.62: Dominio de tamaño 9

Comparación del tiempo total de generación para dominios de tamaño 10



	Tiempo (seg)	
	Trie	*Autómata*
10%	9,71	3,27
25%	10,84	3,58
40%	12,78	4,52
65%	trash	6,1
80%	trash	7,07

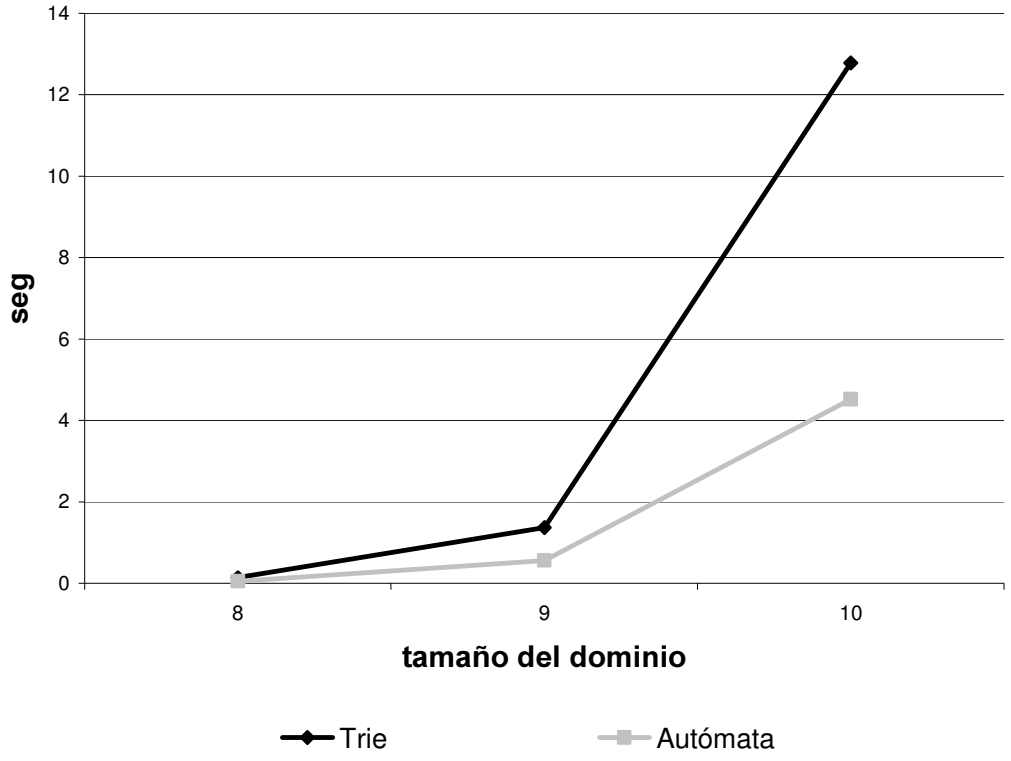
Tabla 8.63: Dominio de tamaño 10

Comparación del tiempo total de generación para dominios de tamaño 11

	Tiempo (seg)	
	Trie	*Autómata*
10%	Trash	29.98
25%	trash	trash

Tabla 8.64: Dominio de tamaño 11

Comparación del total de generación
para el 40% de permutaciones de distintos dominios



	Tiempo (seg)	
	Trie	*Autómata*
Dom 8	0,14	0,05
Dom 9	1,37	0,56
Dom 10	12,78	4,52

Tabla 8.65: 40 % de permutaciones para distintos dominios

Como era de esperar, parte de la mayor ventaja que la estrategia del autómata obtiene por sobre la que utiliza tries en el tiempo de construcción de la estructura en los casos en los cuales ésta debe albergar a una mayor cantidad de permutaciones se contrarresta con la menor ventaja que obtiene en estos casos durante la generación propiamente dicha, lo que termina causando que la ventaja total sea de 1 a 3 en los casos en los que hay que evitar muchas permutaciones, y 1 a 2,5 en los casos en los hay que evitar pocas, aproximadamente.

9. Conclusiones

En la presente tesis hemos provisto una estrategia alternativa para obtener el complemento de un conjunto de permutaciones a la que fue implementada en [M??].

Efectivamente la estrategia propuesta disminuye sustancialmente la cantidad de tiempo y espacio necesario para la generación de este conjunto, lo que permite esperar una mejora en el desempeño general de la herramienta ReMo, no sólo en los casos en los cuales se puede disponer de información de monotonía de las variables relacionales, sino también en los casos en los cuales esto no es posible. Considerando los resultados presentados en [FGSB05], en donde se comparan los tiempos entre una estrategia basada en el análisis de monotonía (ReMo) y el uso de SAT-Solvers (Alloy Analyzer) para verificar una especificación, se espera que con la integración de esta nueva estrategia de poda del espacio de búsqueda, ReMo pueda competir con el analizador Alloy incluso en los casos en los cuales no puede utilizarse la información de monotonía mencionada anteriormente.

Independientemente del contexto en el cual esta estrategia de generación se ha utilizado, la presente tesis pone de manifiesto que el uso de autómatas finitos puede ser una estrategia viable y eficiente para la implementación del complemento de conjuntos de elementos que puedan representarse como cadenas de caracteres. Si consideramos al alfabeto formado por los caracteres usados para esta representación, y el lenguaje formado por los elementos del conjunto, esta estrategia puede ser eficiente para generar un subconjunto del complemento de este lenguaje con respecto al mencionado alfabeto. Ese subconjunto puede obtenerse con un adecuado recorrido del autómata.

En los casos en los cuales se quiere obtener como complemento un conjunto infinito, esta estrategia permite también representarlo como una expresión regular, dado que el complemento de un autómata determinístico también lo es, y su lenguaje puede expresarse, como se mencionó anteriormente, como una expresión regular.

10. Trabajo futuro

Si bien la optimización realizada a la construcción de los autómatas permitió una reducción importante respecto del consumo de memoria en comparación con el uso de tries, los autómatas utilizados no son mínimos. En la presente tesis se mostraron dos enfoques que pueden permitir una mayor reducción de esta estructura. El análisis e implementación de las distintas variaciones que pueden tener estos enfoques puede ser parte de futuras optimizaciones.

Sería también deseable contemplar otro tipo de representación de las permutaciones de forma tal de albergar la misma información pero de una forma más compacta que permita la generación con un consumo de memoria más razonable (quizás alguna forma de codificación de secuencias).

El conjunto de automorfismos a evitar se obtiene a partir de la clausura de un conjunto de generadores. Esta operación es sumamente costosa e incide significativamente en los tiempos de generación de asignaciones a las variables relacionales. En [M??] esta operación no puede implementarse eficientemente debido a las estructuras utilizadas. Los autómatas finitos permiten obtener la composición de los automorfismos que albergan con un cambio poco significativo en el recorrido del lenguaje que aceptan.

Sería interesante contemplar la posibilidad de representar el conjunto de generadores como el lenguaje aceptado por un autómata finito e implementar la clausura del conjunto como una operación sobre el autómata a fin de optimizar la generación del conjunto de automorfismos a evitar y como una alternativa a las propuestas analizadas respecto de la construcción del autómata.

Por otro lado, en el marco de la implementación de la poda del espacio de búsqueda que supone la eliminación de isomorfismos, como se explicó en la sección 4.3 para obtener asignaciones no isomorfas de las variables relacionales es necesario calcular las permutaciones que sean automorfismos de todas las asignaciones que ya se realizaron. Si representamos los automorfismos de las asignaciones con autómatas finitos, una forma de obtener este conjunto es mediante la intersección de los autómatas. En la integración a ReMo será necesario, entonces, implementar un mecanismo eficiente de intersección de este tipo de autómatas que contemple la necesidad de realizar un backtracking en la asignación de las variables relacionales (ya que el conjunto de automorfismos a evitar depende de los automorfismos de las asignaciones ya realizadas y es necesario recalcularlo cuando se cambia la asignación de una variable relacional.)

11. Referencias

[E72] Herbert Enderton, *A Mathematical Introduction to Logic,* Academic Press Inc., 1972.

[FBH97] Marcelo Frías, Gabriel Baum y Armando Haeberer, *Fork algebras in algebra, logic and computer science.* Fundamenta Informaticae (32), 1-25. 1997.

[FGSB05] Marcelo Frías, Rodolfo Gamarra, Gabriela Steren, Lorena Bourg. *A strategy for efficient verification of relational specifications, based on monotonicity analysis*. Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering, páginas 305 – 308. Año 2005. ISBN:1-59593-993-4

[Fri02] Marcelo Frias, *Fork algebras in algebra, logic and computer science*. World Scientific Publishing Co., 2002.

[G73] David Gries, *Describing an Algorithm by Hopcroft*, Acta Informatica, vol. 2, pp. 97-109, 1973.

[G92] Gupta, A, *Formal Hardware Verification Methods: A Survey; Journal of Formal Methods in System Design*, No. 1, pp. 151-238, 1992.

(GJ79] M. Garey y D. Johnson. *Computers and intractability: a guide to the theory of NP-completeness*, W. H. Freeman 1979.

[GJM02] Carlo Ghezzi, Mehdi Jazayeri, Dino Mandrioli. *Fundamentals of software engineering*, Second Edition, 2002

[GS05] Rodolfo Gamarra, Gabriela Steren. *Implementación de una herramienta de model checking basada en álgebra relacional*, Tesis de Licenciatura, Departamento de Computación, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires, 2005.

[HU79] John E. Hopcroft, Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley Publishing Company, 1979.

[HV91] Armando Haeberer y Paulo Veloso, *Partial relations for program derivation: adequacy, inevitability and expressiveness.* Proceedings of the Working Conference on Constructing Programs from Specifications '91. Páginas 310-352, 1991.

[JJD98] Daniel Jackson, Somesh Jha y Craig A. Damon, *Isomorph-free Model Enumeration: A New Method for Checking Relational Specifications* ACM Trans. Programming Languages and Systems, Vol. 20, No. 2, March 1998, pp. 302-343.

[K55] Kurosh, A.G., *The Theory of groups*, Chelsea Publishing Company, New York, 1955.

[M??] Fernando Miranda, *Integrando la eliminación de isomorfismos y el análisis de monotonía a ReMo*, Tesis de Licenciatura en curso, Departamento de Computación, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires.

(McM94] Ken McMillan, *Symbolic Model Checking*. Kluwer Academia Publishers, 1994.

[MP92] Zohar Manna y Amir Pnueli, *The Temporal Logic of Reactive and Concurrent Systems:Specification.* Springer-Verlag, 1992.

[S97] Fred Schneider, *On Concurrent Programming*. Springer-Verlag, NY, 1997.

[T41] Alfred Tarski, *On the calculus of relations.* Journal of Symbolic Logic 6(3), 73-89, 1941.

[TG87] Alfred Tarski, Steve Givant. *A formalization of set theory without variables*, tomo 41 de “AMS colloquium publications”. AMS, 1987.