UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

(POSGRADO EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN)

DISTRIBUCIÓN DEL ACELERAMIENTO NO TRIVIAL EN LA DEMOSTRACIÓN DE SISTEMAS AXIOMÁTICOS ALEATORIOS EN EL CÁLCULO DE PROPOSICIONES

T E S I S
QUE PARA OPTAR POR EL GRADO DE:
MAESTRO EN CIENCIAS (COMPUTACIÓN)

PRESENTA:
SANTIAGO HERNÁNDEZ OROZCO

TUTORES PRINCIPALES:

DR. FRANCISCO HERNÁNDEZ QUIROZ
FACULTAD DE CIENCIAS – UNAM

DR. HÉCTOR ZENIL CHÁVEZ
POSGRADO EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN - UNAM

MÉXICO, D. F. MARZO 2014

Índice general

Resumen

En este trabajo se busca expandir el conocimiento sobre la relación entre las medidas de complejidad de un objeto. En particular se explora de manera empírica el espacio del cálculo de proposiciones. El comportamiento de un sistema de demostración para el cálculo de proposiciones es estudiado al variar el número de premisas de un sistema axiomático, registrando la varianza de las longitudes de demostración en una estructura matricial llamada matriz de aceleramiento.

Debido a la imposibilidad de realizar una búsqueda exhaustiva, se propuso una exploración estadística sobre una sección del espacio usando dos demostradores automatizados: APROS y PROVER9. Así mismo, se definió de manera formal una condición de *normalidad* para discernir si los resultados obtenidos por medio de los demostradores automatizados representan una aproximación adecuada al comportamiento estudiado.

Al aplicar esta condición a los resultados obtenidos se concluyó que los demostradores automatizados usados no presentan un comportamiento *normal* con respecto a las longitudes de demostración en función del número de axiomas. Finalmente, se formularon dos conjeturas que tienen el potencial de explicar la conducta encontrada.

Parte de esta investigación la realicé durante una estancia en el Centro de Medicina Molecular del Instituto Karolinska en Estocolmo, Suecia, bajo la dirección del Dr. Héctor Zenil Chávez. Durante esta estancia empecé a estudiar el concepto de distancia de compresión normalizada. Presente varios resultados preliminares durante el congreso "Computabilidad en Europa 2013"(CiE 2013), el día 2 de julio en la Universidad de Milán-

Biccoca.

Entre las *contribuciones originales* presentadas en este trabajo se encuentran: la definición formal de la delta de aceleramiento para espacios de cálculo lógico; un algoritmo para enumerar y generar de manera pseudoaleatoria conjuntos de argumentos válidos en el cálculo de proposiciones de acuerdo con una medida de complejidad sintáctica; una forma de visualizar los resultados obtenidos por medio de matrices; una condición necesaria de normalidad para demostradores automatizados en función de la delta ya mencionada y dos conjeturas con respecto a esta condición.

Capítulo 1

Introducción.

1.1. El fenómeno del aceleramiento

Entre las medidas existentes que buscan definir y cuantificar la complejidad de un objeto se encuentran aquellas que se enfocan en la cantidad de recursos computacionales requeridos para su descripción. El fenómeno del *aceleramiento* (speed-up) y del *alentamiento* (slow-down) se refieren al compromiso, la relación existente, entre dos medidas computacionales de complejidad ([1]):

- La complejidad de ejecución, conocida como complejidad computacional y
- la complejidad de contenido de información algorítmica o de tamaño de programa [1].

Para poder dar una descripción formal al fenómeno del aceleramiento es preciso primero definir en términos no ambiguos las dos medidas de complejidad mencionadas.

1.1.1. La complejidad descriptiva.

Podemos pensar en la complejidad de contenido de información algorítmica de un objeto como el mínimo número de caracteres requeridos para describir el sistema en un lenguaje dado. Para evitar ambigüedades es necesario dar la formulación de un lenguaje en términos matemáticos precisos.

Definición 1.1. Definimos un *alfabeto* como un conjunto finito, cuyos elementos son llamados *símbolos* o *caracteres*. Ahora, sea Σ un alfabeto, llamaremos a las cadenas finitas ordenadas w de caracteres en Σ *palabras* y al conjunto Σ^* el conjunto de todas las palabras posibles con caracteres en Σ. Denotaremos por ϵ a la palabra vacía.

Dado un alfabeto Σ, definimos a un *lenguaje* L con caracteres en Σ como un subconjunto de Σ^*.

En la literatura existen varios modelos equivalentes de computabilidad. Si seguimos la Tesis de Church-Turing [4], estos modelos definen a la clase de funciones *efectivamente calculables*. En esta sección usaré el modelo propuesto por Alan Turing [5], llamado máquinas de Turing.

Definición 1.2. Una *máquina de Turing*, denotada por *TM*, es una n-ada $M = (Q, \Sigma, \Gamma, \vdash, \sqcup, \sigma, s, t, r)$, donde

- Q es un conjunto finito, llamado *conjunto de estados*;
- Σ es un alfabeto, llamado *alfabeto de entrada*;
- Γ es un alfabeto tal que $\Sigma \subset \Gamma$, conocido como el *alfabeto de cinta*;
- $\sqcup$ es un carácter tal que $\sqcup \in \Gamma - \Sigma$, llamado el *símbolo blanco*;

- $\vdash \in \Gamma - \Sigma$, llamado símbolo de inicio de cinta;
- σ es una función total $\sigma : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$, llamada función de transición;
- $s \in Q$ es el estado inicial;
- $t \in Q$ es el estado de aceptación; y
- $r \in Q$ es el estado de rechazo.

Definición 1.3. Diremos que una función parcial $f : \Sigma^* \to \Sigma^*$ es *computable* si se puede definir por medio de una máquina de Turing, la cual usamos para procesar el argumento de entrada para f usando el siguiente procedimiento:

Para computar $f(c)$, donde la palabra c esta compuesta por la cadena de caracteres $c_1, c_2, ..., c_n$, primero consideremos a la cadena infinita de caracteres $\vdash, c_1, c_2, ..., c_n, \sqcup, \sqcup, ...$; esta cadena es llamada la *cinta o memoria de la máquina*. Empezamos por evaluar $\sigma(s, c_1)$. Sea (p, c, d) el resultado de la evaluación, entonces reemplazamos en la cinta el carácter c_1 por c y, si $d = R$ evaluamos $\sigma(p, c_2)$ (el carácter a la derecha de nuestra posición actual); de lo contrario evaluamos $\sigma(p, \vdash)$ (el carácter a la izquierda). Repetimos el proceso para la salida de cada evaluación hasta obtener una tripleta con $p = t$ o $p = r$.

Si $p = t$ entonces el estado de cinta, menos los caracteres que no pertenecen a Σ, nos define la *salida* de la función para c. De lo contrario decimos que f no esta definida para c.

Llamaremos el *lenguaje aceptado por* f al conjunto rango de f, denotado por $L(f)$, donde el rango de f está dado por el

conjunto

$$\{c \in \Sigma^* | f(c) \text{ esta definido}\}.$$

Si la máquina de Turing M define la función computable f entonces usaremos indistintamente M o f para denotar a la función.

Definición 1.4. Un *conjunto libre de prefijos* es un conjunto de palabras S con la propiedad de que ninguna palabra en S es prefijo de una palabra distinta en S. Es decir, si $\alpha, \beta \in S$ y $\alpha \neq \beta$, entonces no existe cadena θ tal que $\alpha = \beta\theta$ o viceversa.

Definición 1.5. Dado un alfabeto Σ, definimos a una *computadora* C como una función parcial computable de $\Sigma^* \times \Sigma^*$ en Σ^* que mapea una palabra p, llamada cadena de programa, y una palabra q, llamada cadena de datos o de entrada, en una palabra $C(p, q)$, llamada cadena de salida. Además, esta cadena de salida debe tener la propiedad de que para cada p el conjunto

$$\{C(p, q) | p \in \Sigma^* \text{ y } C(p, q) \text{ está definido}\}$$

es libre de prefijos. Conoceremos a esta última propiedad como *autodelimitación* de un programa.

En el contexto de máquinas de Turing, definimos a C como el procedimiento descrito en 1.3 y a los programas como las máquinas de Turing que definen a cada función computable. Es decir, si f esta definido por T, entonces $f(x) = C(T, x) = U(p', x)$, donde la computadora C es el procedimiento descrito para la interpretación de máquinas de Turing, U es una computadora y p' es la codificación de T para U. En particular U puede ser la máquina de Turing expuesta en [5].

Definición 1.6. Una computadora U es una *computadora universal* si y sólo si, para cada computadora C, existe una constante $sim(C) \in \mathbb{N}$ con la siguiente propiedad: si $C(p, q)$ está definido,

entonces existe $p' \in \Sigma^*$ tal que $U(p', q) = C(p, q)$ y $|p'| \leq |p| + sim(C)$, donde $|p|$ es la *longitud* de la cadena p, es decir el número de caracteres que la componen.

Definimos a p' como una *codificación* de $f(x) = C(p, x)$, o simplemente p, para U.

Teorema 1.7. *Existe una computadora universal.*

Demostración. La demostración para máquinas de Turing fue dada por Alan Turing en su articulo de 1937 [5].

□

Ahora tenemos todos los elementos para definir de manera formal la complejidad de contenido de información algorítmica, concepto que definiremos en términos de una computadora universal.

Definición 1.8. Dada una palabra s, definimos a s^* como la cadena p tal que $|p| = min\{|r| : U(r, \epsilon) = s\}$ y $U(p, \epsilon) = r$. La palabra s^* es llamada el *programa canónico* de s con respecto a U.

Para una computadora C, podemos pensar en la palabra p como un programa para dicha computadora, y en q como una entrada para este programa. Ahora, en la definición anterior, notemos que s^* es la cadena más corta que es un programa para U y que,dada la entrada vacía, produce a p como salida. Si hay varias cadenas con está propiedad simplemente tomamos la más pequeña, usando el orden lexicográfico inducido por un orden total en Σ, el cual existe ya que es un conjunto finito.

Definición 1.9. Dada una computadora C y una cadena s, definimos a la *complejidad del tamaño del programa de s con respecto a C*, también conocida como el *contenido de información algorítmica de s con respecto a C*, denotado por $H_C(s)$ como:

$$H_C(s) := min\{|r| : C(r, \epsilon) = s\} = |s^*|; \tag{1.10}$$

de no existir r, entonces $H_C(s)$ se define como ∞.

Es importante notar que la definición anterior se puede usar en diferentes modelos de computación. Por otra parte, el valor de la complejidad depende de dicho modelo; este valor incluso depende de la máquina universal que se esté usando (no establecí unicidad de la misma). Sin embargo, dados dos modelos distintos de computabilidad, los valores de complejidad de Kolmogorov-Chaitin difieren únicamente en una constante:

Teorema de la invariancia **(1.11).** Sean U y U' dos computadoras universales. Para toda cadena s tenemos que existe una constante $C(U, U')$ tal que

$$|H_U(s) - H_{U'}| \leq C(U, U'). \tag{1.12}$$

Demostración. La demostración fue dada por Solomonoff [11], Kolmogorov [10] y Chaitin [7]. □

Ahora, si U es una máquina universal denotamos el contenido de información algorítmica de s con respecto a U como simplemente $H(s)$; y denotamos por $H(s, t)$ a la complejidad de la concatenación de s y de t. Finalmente, si retiramos la condición de *autodelimitación* de los programas denotaremos por $K(s)$ a la función resultante.

Concluyo esta sección con un conjunto de definiciones complementarias.

Definición. 1.13. Dada las palabras s y t, la computadora C y la computadora universal U, definimos:

- Las complejidades

$H_C(s/t) := min\{|r||C(r,t^*) = s\}$, llamada *complejidad de Kolmogorov–Chaitin, complejidad del tamaño del programa de s relativa a la entrada t con respecto a C* o el *contenido de información algorítmica de s relativo a la entrada t con respecto a C*;

$H(s/t) := H_U(s/t)$, llamada *complejidad condicional de Kolmogorov*, es la medida de la *complejidad del tamaño del programa de s relativa a la entrada t* o el *contenido de información algorítmica de s relativo a la entrada t*;

$H_C(s:t) := H_C(t) - H_C(t/s)$, y

$H(s:t) := H_U(s:t)$.

- Las probabilidades

  $P_c(s) := \sum_{C(p,\epsilon)=s} 2^{-|p|}$,

  $P(s) := P_U(s)$, llamada la *probabilidad algorítmica* de s;

  $P_C(s/t) := \sum_{C(p,t^*)} = s^{2^{-|p|}}$,

  $P(s/t) := P_U(s/t)$, llamada la *probabilidad algorítmica condicional de s y t*; y

  $\Omega := \sum_{U(p,\Lambda) \text{ está definido}} 2^{-|p|}$, llamada la constante de Chaitin [6].

1.1.2. La complejidad de ejecución.

La siguiente medida de complejidad a introducir es la complejidad computacional, o complejidad de ejecución, de un sistema. Esta medida analiza la complejidad de un sistema en términos de los recursos utilizados. En general, durante la ejecución de un programa, consumimos dos recursos computacionales: el tiempo y la cantidad de memoria utilizados; por lo cual definiremos

dos medidas de complejidad de ejecución distintas, pero relacionadas.

Definición 1.14. Sean C una computadora, p un programa y q una cadena de entrada para el programa p. Recordando la definición de computabilidad (1.3), durante la ejecución de un programa primero evaluamos $\sigma(s, c_1) = (p_1, c_{i(1)}, d_1)$, seguido por la evaluación de $\sigma(p_1, q_1)$ y proseguimos hasta terminar la ejecución del programa al encontrar una evaluación de la forma $\sigma(p_k, q_k) = (p_k, c_{i(k)}, d)$, donde $p_k = t$ o $p_k = r$. Llamaré a la sucesión $\langle (p_1, c_{i(1)}, d_1), (p_2, c_{i(2)}, d_2), ..., (p_k, c_{i(k)}, d) \rangle$ el *historial de ejecución* de $C(p, q)$.

Notemos que el historial de ejecución puede ser una sucesión infinita.

Definición 1.15. Definimos el *tiempo de ejecución* de $C(p, q)$, denotado por $t(C(p, q))$, como el número de entradas o elementos en su historial de ejecución. Si el historial de ejecución es una sucesión infinita entonces diremos que el tiempo de ejecución es infinito.

Ahora, dado el alfabeto Σ y un número natural n, consideremos al conjunto $\Sigma[n] = \{s | s \in \Sigma^* \text{ y } |s| = n\}$. Notemos que, ya que Σ es un conjunto finito, $\Sigma[n]$ también es un conjunto finito.

Definición 1.16. Dada una computadora C con alfabeto Σ, definimos a la *complejidad computacional* de un programa p como la función $t_p(n) : \mathbb{N} \to \mathbb{N}$, donde

$$t_p(n) := max\{t(C(p, q)) | q \in \Sigma[n]\}.$$

Notemos que si la función $f(x) = C(p, x)$ es total entonces la complejidad computacional esta definida para todas sus entradas.

Además del tiempo de ejecución, la complejidad computacional de un sistema también puede definirse en términos del espacio utilizado. En una máquina de Turing el espacio utilizado esta dado por el número de *celdas* (caracteres) de la cinta de máquina usadas durante la ejecución. Formalmente, podemos definir la complejidad de espacio de la siguiente forma:

Definición 1.17. Consideremos al programa p y la entrada q. Denotemos el historial de ejecución de $C(p,q)$ por $S = \langle s_i \rangle$. Definimos el *espacio de ejecución* de $C(p,q)$, denotado por $t'(C(p,q))$, como

$$t'(C(p,q)) = max\{h(\varsigma)|\varsigma \in PS\},$$

donde PS es el conjunto de todos los prefijos de S y h es la siguiente función:

$$h(\varsigma) = \sum_{s_i \in \varsigma} g(s_i),$$

$$g(s_i) = \begin{cases} 1 & \text{si } d_i = R \\ -1 & \text{si } d_i = L. \end{cases}$$

Por ejemplo, consideremos a la computadora $C(p,q)$ que, dada cualquier entrada q, escribe un 1 en la cinta, lo cambia por un 0 y termina la ejecución. Es decir, computa la función constante $f(x) = 0$ realizando las operaciones descritas. Esperamos que esta sencilla computadora use únicamente una celda. Consideremos el siguiente historial de ejecución:

$$\langle (p_1, \text{`1'}, R), (p_2, \text{` } \sqcup \text{ '}, L), (t, \text{`0'}, R) \rangle$$

La ejecución de la máquina descrita por este historial empieza escribiendo '1' en la primer celda, después se mueve a la derecha

en donde encuentra el símbolo blanco, regresa a la celda anterior y sobrescribe un '0', terminando su ejecución. El tiempo de ejecución de esta máquina es 3 (la longitud del historial), y su espacio de ejecución es:

$$\begin{aligned} t'(C(p,q)) &= max\{0, 1, 1-1, 1-1+1\} \\ &= max\{0, 1, 0, 1\} \\ &= 1, \end{aligned}$$

el cual es el valor esperado.

Ahora, de manera análoga a la definición 1.16, definimos la *complejidad computacional espacial* como la función $t'_p : \mathbb{N} \to \mathbb{N}$, donde

$$t'_p(n) := max\{t'(C(p,q)) | q \in \Sigma[n]\}.$$

Entre las relaciones existentes entre las medidas de complejidad espacial expuestas, una de las más inmediatas es la siguiente: está claro que no podemos usar más celdas que el número de elementos que componen un historial de ejecución, por lo cual

$$t'_p(n) \leq t_p(n) \text{ para todo } n \in \mathbb{N}. \tag{1.18}$$

Es por esto que podemos decir que la complejidad computacional domina a la espacial. En este texto nos enfocaremos en la complejidad computacional definida en términos del tiempo de ejecución al medir la complejidad de ejecución de un sistema.

Finalmente definiré la noción de órdenes de complejidad, la cual nos permite clasificar conjuntos de sistemas (como pueden ser las maquinas de Turing) de acuerdo con su complejidad de ejecución.

Definición 1.19. Sean $f : \mathbb{N} \to \mathbb{N}$ y $g : \mathbb{N} \to \mathbb{R}$ dos funciones no necesariamente de complejidad. Decimos que f es de *orden* g, expresado por la relación $f \in O(g)$, si f está acotada asintóticamente por g, es decir:

$$f \in O(g) \iff \exists c, N \in \mathbb{R}^+ . n > N \implies f(n) \leq c \cdot g(n).$$

En cambio, si g está acotada asintóticamente por f, decimos que $f \in \Omega(g)$. Si $f \in O(g)$ y $f \in \Omega(g)$ entonces decimos que $f \in \Theta(g)$.

De manera intuitiva, $f \in O(g)$ significa que la función f tiene una complejidad menor o igual a g, o que (el sistema que describe) pertenece a la clase de sistemas que son a lo más igual de complejos que g, salvo constantes. También podemos pensar que $f \in \Omega(g)$ nos dice que f es al menos igual de complejo que g y que $f \in \Theta(g)$ significa que f es igual de complejo que g, salvo constantes en ambos casos.

La definición de ordenes de complejidad se pueden aplicar a las tres medidas de complejidad expuestas en este texto.

1.1.3. El fenómeno del aceleramiento en TM.

Antes de poder proseguir con la definición formal del fenómeno del aceleramiento es necesario primero aclarar a qué nos referimos con la complejidad de un sistema.

Pensemos en lo que significa la complejidad de una función computable f, podemos pensar en ésta de acuerdo a su complejidad descriptiva, es decir el programa de complejidad descriptiva mínima que la codifica; esta es la complejidad definida en 1.13. También podemos pensar en la complejidad de ejecución de este programa, es decir el tiempo y el espacio necesario requeridos

por este programa. No hay razones para pensar que el programa de complejidad descriptiva mínima también es el programa de complejidad de ejecución mínima, o viceversa. Por lo anterior no se considera posible el asignarle un valor único de complejidad a f.

Definición 1.20. Sean f una función computable, C una computadora y p un programa tal que $f(x) = C(p, x)$. Decimos que p es una *descripción* de F para C.

Definición 1.21. Sean f una función computable, p y q dos descripciones de f tales que $H(p) < H(q)$ y n un número natural. Llamamos *delta de aceleramiento* de f entre p y q para n, o simplemente *aceleramiento*, a la función

$$\delta(p, q) = 1 - \frac{t_q(n)}{t_p(n)}.$$

donde t es la función definida en 1.15 aplicada a las descripciones p y q respectivamente.

Notemos que la definición dada de aceleramiento puede extenderse a otras medidas de complejidad.

En un trabajo anterior, Joosten, Soler-Toscano y Zenil ([1]) realizaron una exploración experimental exhaustiva del espacio de las máquinas de Turing pequeñas, definidas como aquellas máquinas que tienen alfabeto de entrada compuesto por dos símbolos y conjuntos de estados conformados por dos o tres estados.

Definiendo la complejidad descriptiva de una MT pequeña por su número de estados, encontraron que

> "El tiempo de ejecución promedio de las máquinas de Turing que computan una función aumenta -con una probabilidad cercana a uno- con el número de estados."

Es decir, encontraron que $\delta < 0$ en casi todos los casos [1].

1.2. El aceleramiento en P

1.2.1. Sistemas de demostración para P.

El objetivo de este trabajo es explorar el fenómeno del aceleramiento en una clase distinta de sistemas: los sistemas de cálculo lógico o de demostración, los cuales son herramientas para verificar la validez de enunciados lógicos de un sistema formal por medios estrictamente sintácticos. Para poder llevar a cabo este objetivo es necesario definir de manera formal la noción de aceleramiento para este tipo de espacios.

Dedico esta sección a dar una definición formal de la clase de sistemas de demostración al cual me voy a enfocar:

Definición 1.22. Un sistema de demostración para el cálculo de proposiciones es la tupla

$$L = (Var, Op, P, S, s, e)$$

donde:

- Var es un conjunto infinito numerable, donde sus elementos son llamados *variables*;
- Op es un conjunto finito de funciones $f_k^n(p_1, ..., p_n)$ cuyos elementos son llamados operadores lógicos;

- P es un conjunto, llamado conjunto de *proposiciones* o fórmulas, definidos por la gramática:

$$\alpha := X|C|f_k^n(\alpha_1, ..., \alpha_n),$$

donde $X \in Var$, $C \in \mathbb{B}$, $\alpha_1, ..., \alpha_n$ son proposiciones, f_k^n es un operador lógico y $\mathbb{B} = \{V, F\}$ es el dominio booleano;

- S, llamado conjunto de *reglas de inferencia o derivación*, es un conjunto que consta de reglas de la forma

$$\frac{\eta_1,...,\eta_k}{\eta}, \quad \frac{D(\eta_1,...,\eta_k)}{\eta}, \quad \frac{\eta_1, D(\eta_2,...,\eta_k)}{\eta}$$

donde $\eta, \eta_1, ..., \eta_k$ y α son fórmulas y $D(\eta_1, ..., \eta_k)$ es una derivación con $\eta_1, ..., \eta_k$ como premisas; este último par de términos se definen posteriormente. Denotaré a cada regla de inferencia por $r(\eta_1, ..., \eta_k, \eta)$.

- s es una función $s : P \times Var \times P \to P$, llamada *substitución*;

- e, llamada función de evaluación, es una función de la forma $e : P \times \mathbb{B}^* \to \mathbb{B} \cup \{\perp\}$.

Ahora, sean α y β dos proposiciones, decimos que existe una *sustitución* de α a β si existen $x, y \in Var$ tales que $s(\alpha, x, y) = \beta$. Si α es una fórmula, llamamos a la cadena $\overline{b} = b_1, ..., b_n \in \mathbb{B}^*$ una *evaluación* para α. Decimos que la fórmula α es *verdad* para la evaluación $\overline{b}$ si $e(\alpha, \overline{b}) = V$; denotaremos a $e(\alpha, \overline{b})$ por $\alpha(\overline{b})$. Si $e(\alpha, \overline{b}) = \perp$ decimos que el valor de verdad para la expresión no esta definido.

Definición 1.23. Un sistema de demostración es *decidible* si la función de evaluación es total y computable.

Teorema 1.24. *El cálculo de proposiciones es decidible.*

Demostración. Podemos definir a la función e de la siguiente forma:

$$e(V, \overline{b}) = V$$
$$e(F, \overline{b}) = F$$

Si x_i es la i-esíma variable en Var,

$$e(x_i, \overline{b}) = b_i$$

donde b_i es el i-esímo elemento de $\overline{b}$. Si $i > |\overline{b}|$ entonces le función regresa $\perp$.

Finalmente, para cada $f_k^n \in Op$ definimos una función de la forma $tf_k^n : \mathbb{B}^n \to \mathbb{B}$ donde:

$$e(f_k^n(\alpha_1, ..., \alpha_n), \overline{b}) = tf_k^n(e(\alpha_1, \overline{b}), ..., e(\alpha_{n_i}, \overline{b})).$$

Notemos que tenemos 2^{n_i} cadenas $\overline{e} \in \mathbb{B}^{n_i}$ distintas, por lo cual tenemos un número finito de casos de la forma $tf_k^n(e_1, ..., e_{n_i})$.

Luego, ya que el conjunto Op es finito, la definición de e es finita, recursiva primitiva y por lo tanto computable [12]. □

Definición 1.25. Llamamos *tautología* a una proposición α, denotada por $\models \alpha$, si tenemos que $\alpha(\overline{b}) = V$ para toda evaluación $\overline{b}$ con $|\overline{b}|$ mayor o igual al índice máximo de las variables en α. Decimos que α es satisfactible si existe una evaluación $\overline{b}$ tal que $\alpha(\overline{b}) = V$.

Definición 1.26. Sean $\alpha_1, ..., \alpha_t$ y β fórmulas. Decimos que el *argumento* denotado por $\alpha_1, ..., \alpha_t \models \beta$ es correcto o *válido* si, para toda evaluación $\overline{b}$ tal que

$$\alpha_1(\overline{b}) = V, ..., \alpha_t(\overline{b}) = V,$$

tenemos que $\beta(\bar{b}) = V$. Llamamos a las fórmulas $\alpha_1, ..., \alpha_t$ *premisas o axiomas*, y a su conjunto una *teoría* en L; llamamos a la fórmula β una conclusión. Decimos que β es un teorema para la teoría definida por $\alpha_1, ..., \alpha_t$ si $\alpha_1, ..., \alpha_t \models \beta$ es válido. Sea $t = \{\alpha_1, ..., \alpha_t\}$ una teoría. Si existe una evaluación $\bar{b}$ tal que $\alpha_1(\bar{b}) = V, ..., \alpha_t(\bar{b}) = V$ entonces decimos que t es *satisfactible* y lo denotamos por $e(t) = V$. Si una teoría es satisfactible entonces decimos que es *consistente*.

Definición 1.27. Sean t y t' dos teorías en L. Decimos que t y t' son *equivalentes*, denotado por $t \simeq t'$, si todo teorema para t también es teorema para t' y viceversa. Si t y t' son teorías no equivalentes, y todo teorema para t también es un teorema para t', decimos que t es una teoría *más fuerte* que t'.

Lema 1.28. *Si β es un teorema para t, entonces $t \simeq t \cup \{\beta\}$.*

Demostración. Sea γ un teorema para t. Denotemos por $t(\bar{b}) = V$ cuando, para toda $\alpha_i \in t$, tenemos que $\alpha(\bar{b}) = V$. Sea $\bar{b}$ una evaluación tal que $t \cup \{\beta\}(\bar{b}) = V$, lo que implica que $t(\bar{b}) = V$ y $\gamma(\bar{b}) = V$; por lo tanto γ es un teorema para $t \cup \{\beta\}$ y todo teorema para t es un teorema para $t \cup \{\beta\}$.

Ahora, sea γ un teorema para $t \cup \{\beta\}$. Sea $\bar{b}$ una evaluación tal que $t(\bar{b}) = V$. Luego, ya que β y γ son teoremas para t, tenemos que $\beta(\bar{b}) = V$, $t \cup \{\beta\}(\bar{b}) = V$ y $\gamma(\bar{b}) = V$; por lo tanto γ es un teorema para t y todo teorema para $t \cup \{\beta\}$ es un teorema para t.

Juntando los dos párrafos anteriores obtenemos la equivalencia. □

Definición 1.29. Una *derivación* en una teoría t es una sucesión finita $\langle \gamma_1, ..., \gamma_l \rangle$ de fórmulas; donde, para toda $0 \leq i \leq l$, tenemos que $\gamma_i \in t$, o bien existe una regla en $r(\eta_1, ..., \eta_k, \eta) \in S$ y un conjunto $\gamma_{i_1}, ..., \gamma_{i_k}$ con $i_j < i$ tales que, para toda γ_{i_j}, existe

una substitución a η_j y existe una substitución de γ_i a η.

Definición 1.30. Sean β una fórmula y t una teoría, decimos que β es demostrable en t, denotado por $t \vdash \beta$, si existe una derivación en t donde β es el último elemento de esta cadena. Decimos que un argumento $t \models \beta$ es demostrable si $t \vdash \beta$. Llamamos a dicha derivación una *demostración* para β en t o una demostración para el argumento $t \models \beta$. Decimos que un sistema de demostración L es *correcto* si todo argumento demostrable es válido. Un sistema de demostración L bien definido es *(fuertemente) completo* si, para toda fórmula β, tenemos que $t \vdash \beta$ si $t \models \beta$.

Finalmente, denotaré a un argumento demostrable por $t \vdash \beta$.

1.2.2. Aceleramiento en P.

Consideremos al conjunto de todas las teorías en P, denotado por $\mathbb{T}$. Notemos que la definición 1.27 nos induce una relación de equivalencia en P. Llamemos *clase de teorías equivalentes* a los elementos de $\mathbb{T}/\simeq$, la partición inducida por la relación de equivalencia. Además, decimos que dos argumentos $t \vdash \beta$ y $t' \vdash \beta'$ son equivalentes si $t \simeq t'$ y $\beta = \beta'$. Notemos que esta relación también es una relación de equivalencia para el espacio de todos los argumentos en P.

Definición 1.31. Sea $[t] \in P/\simeq$ una teoría. Decimos que $t \in [t]$ es una *representación* de $[t]$.

Notemos que, por el lema 1.28, existen distintas descripciones de una teoría con distinto número de axiomas, por lo cual podemos dar la siguiente definición:

Definición 1.32. Definimos la *complejidad descriptiva* de una representación t de una teoría como el número de axiomas que la componen, denotado por $|t|$. Análogamente, definimos la complejidad descriptiva de un argumento $t \vdash \beta$ como $|t|$, denotado por $|t \vdash \beta|$.

Además, definimos la complejidad descriptiva de una teoría $[t]$ como

$$|[t]| = min\{|r| : r \in [t]\}.$$

Sí $|t| = |[t]|$ decimos que t es una *representación mínima*. Análogamente, decimos que un argumento $t \vdash \beta$ es mínimo si t es una representación mínima.

Definición 1.33. Sea d una demostración para el argumento $t \vdash \beta$. Definimos a la *complejidad de la demostración* d como el número de elementos que componen a la sucesión $d = \langle \gamma_1, ..., \gamma_l \rangle$, denotado por $|d|$. Definimos a la *complejidad de demostración* del argumento, denotada por $D(t \vdash \beta)$, como

$$D(t \vdash \beta) = min\{|d||d \text{ es un demostración para } t \vdash \beta\}.$$

Si $|d| = D(t \vdash \beta)$ entonces decimos que d es una *demostración mínima*.

Notemos que si $t \vdash \beta$ es demostrable entonces $D(t \vdash \beta)$ existe y, al ser Op y S finitos, es computable. Sin embargo, el siguiente resultado nos sugiere que calcular dicho valor es *difícil* en un sentido computacional.

Teorema 1.34. *Calcular $D(t \vdash \beta)$ es un problema al menos* NP-difícil.

Demostración. Alekhnovich, Buss, Moran y Pitassi demostraron que el aproximar la mínima longitud de demostración de un proposición en un factor lineal es un problema *NP-difícil* [13].

Sigue que encontrar $D(t \vdash \beta)$ es un problema al menos *NP-difícil*.

□

El describir de manera detallada la clase de complejidad conocida por *NP-difícil* esta fuera de los alcances de este trabajo, así que me limitaré a decir que el pertenecer a esta clase de complejidad computacional significa estar dentro de los problemas decidibles más difíciles de resolver en un tiempo computacional razonable [28].

Definición 1.35. Sean $t \vdash \beta$ una argumento demostrable, t y t' dos representaciones de $[t]$ tales que $|t| < |t'|$. Llamaremos *delta de aceleramiento* de $t \vdash \beta$ entre t y t', o simplemente *aceleramiento*, a la función:

$$\delta(t, t') = 1 - \frac{D(t' \vdash \beta)}{D(t \vdash \beta)}.$$

Lema 1.36. *Sean $t \vdash \beta$ un argumento y t y t' dos descripciones de la teoría $[t]$. Si $t \subset t'$ entonces $D(t' \vdash \beta) \leq D(t \vdash \beta)$; en otras palabras $\delta(t, t') \geq 0$.*

Demostración. El resultado es directo. Por definición, toda derivación en t también es una derivación en t', por lo que si d es una demostración mínima para $t' \vdash \beta$, d es una demostración para $t' \vdash \beta$; por lo cual $D(t \vdash \beta)$ es una cota superior de $D(t' \vdash \beta)$.

Ahora, si $\beta \in t'$ y $\beta \notin t$ obtenemos la desigualdad, ya que la sucesión $\langle \beta \rangle$ es una demostración para $t' \vdash \beta$. □

Definición 1.37. Llamaremos *aceleramiento trivial* al caso descrito en el segundo párrafo de la demostración anterior. Es decir, $\delta(t, t') > 0$ es un aceleramiento trivial para el argumento $t \vdash \beta$, $t \subset t'$, si $\beta \in t'$ y $\beta \notin t$.

La definición provista de aceleramiento para sistemas de cálculo lógico para el cálculo de proposiciones puede adaptarse fácilmente para otro tipo de lógicas. Para este trabajo hemos decidido concentrarnos en el cálculo de proposiciones, ya que este espacio tiene la propiedad de ser decidible, simplificando de manera significativa la exploración del espacio.

Capítulo 2

El problema

El objetivo principal de este trabajo es el explorar la respuesta a la siguiente pregunta: *en el cálculo de proposiciones, si hacemos un sistema axiomático más fuerte al agregar teoremas como axiomas, que tan frecuentemente podemos encontrar casos en los cuales algún conjunto de teoremas decidibles para el sistema axiomático más débil se puede demostrar de forma significativamente más rápida en el sistema axiomático más fuerte.*

Formalmente, el objetivo principal de este trabajo es encontrar la distribución de los casos de aceleramiento positivo ($\delta > 0$) dentro del espacio de argumentos.

Denotemos por $\mathbb{A}$ al espacio de argumentos en el cálculo de proposiciones y por $\mathbb{S}$ al conjunto de todos los argumentos $t_i \vdash \beta_i$ para los cuales el conjunto $A_j \subset P$, $A_j \cap t_i = \emptyset$, tiene la siguiente propiedad: $D(t_i \cup A_j) < D(t_i)$, lo que es equivalente a $\delta(t_i, t_i \cup A_j) > 0$ para β_i. Nos preguntamos por la relación y la distribución entre el orden del conjunto $\mathbb{S}$ con respecto al orden del conjunto $\mathbb{A}$ en función del conjunto A_j.

2.1. Argumentos para la existencia de aceleramiento no trivial

La demostración del lema 1.36 nos asegura la existencia de al menos un tipo de aceleramiento. Sin embargo, el tipo de aceleramiento cuya existencia fue establecida es llamado trivial ya que no es un caso considerado interesante. Durante el planteamiento y el desarrollo de este trabajo se manejaron varias razones para la existencia de *aceleramiento no trivial* en el espacio propuesto.

A continuación presento tres de las observaciones principales, no limitadas al cálculo de proposiciones, que nos llevaron a pensar en la probable existencia de aceleramiento no trivial en una cantidad significativa de casos.

Debido a que el estudio explícito de varios de los conceptos que introduciré en las siguiente secciones se encuentra fuera del alcance de este trabajo, me limitaré a ofrecer una descripción concisa de los mismos. Mas información sobre los dos primeros temas descritos se puede encontrar en los libros "Set Theory: An Introduction to Indepence Proofs" [14] y "Lectures on the Curry-Howard Isomorphism" [15] respectivamente.

2.1.1. El principio de reflexión.

Sea $\Phi = \{\phi_i | i \in \mathbb{N}\}$ el conjunto de todas las fórmulas en el cálculo de predicados con fórmulas atómicas $x = y$ y $x \in y$.

Definición 2.1. Decimos que una fórmula ϕ con a lo más n variables x_i libres es *absoluta* para un conjunto A si y sólo si

$$\forall x_1, ..., x_n \in A.(\phi^A(x_1, ..., x_n) \leftrightarrow \phi(x_1, ..., x_n)),$$

donde $\phi^A(x_1, ..., x_n)$ es la *relativización* de ϕ en A, la cual se obtiene al reemplazar el cuantificador existencial $\exists x$ por $\exists x \in A$,

el cuantificador universal $\forall x$ por $\forall x \in A$, $\phi = \gamma$ por $\phi^A = \gamma^A$, $(\phi \wedge \gamma)^A$ por $\phi^A \wedge \gamma^A$ y de manera similar con los demás conectivos lógicos. Si ϕ^A entonces decimos que ϕ es verdadero en A.

Lema 2.2. *Consideremos una fórmula de la forma*

$$\exists a.\phi_k(x_1, ..., x_m, a).$$

En ZF, para cada conjunto V_α existe un conjunto V_β tal que $V_\alpha \subset V_\beta$ y

$$\exists a.\phi_k(x_1, ..., x_m, a) \rightarrow \exists a \in V_\beta.\phi_k(x_1, ..., x_m, a)$$

donde ZF es conjunto de axiomas conocido como la teoría de conjuntos de Zermelo-Fraenkel y V_α esta definido de forma recursiva de la siguiente forma:

$$V_0 = \emptyset$$
$$V_{\alpha+1} = \wp(V_\alpha),$$

donde $\wp(A)$ es el conjunto potencia de A, y si α es un ordinal límite entonces

$$V_\alpha = \bigcup_{\gamma<\alpha} V_\gamma.$$

Esta clase de conjuntos bien fundados es llamada Universo de Von Neumann.

Demostración. La demostración de este lema y del siguiente teorema se encuentran en el libro de Kunen [14]. □

El lema anterior se usa en la demostración del siguiente teorema:

El teorema de la reflexión 2.3. *Dada cualquier cadena de fórmulas $\phi_1, ..., \phi_n$, entonces*

$$ZF \vdash \forall\alpha\exists\beta.((\beta > \alpha) \wedge (\phi_1, ..., \phi_n \text{ son absolutas para } V_\beta)).$$

Ahora consideremos una fórmula en Φ de la forma $\forall a.\phi_k(x_1, ..., x_m, a)$. Podemos encontrar un conjunto V_α tal que $\forall a \in V_\alpha.\phi_k(x_1, ..., x_m, a)$ es verdadero, el cual es equivalente al enunciado $\phi_k(x_1, ..., x_m, a \in V_\alpha)$ en un proceso similar a la skolemización. Por inducción podemos extender el razonamiento anterior a un número finito de fórmulas y cuantificadores, por lo que podemos concluir que, dado cualquier ordinal α y un natural n, existe un ordinal $\beta > \alpha$ tal que para cada enunciado verdadero en V_α existe una fórmula con menos de n cuantificadores equivalente en V_β.

Por otro lado, una forma de clasificar la complejidad de una fórmula en el cálculo de predicados es por el número de cuantificadores presentes: Notemos que el teorema 1.24 nos dice que un sistema de cálculo lógico sin cuantificadores es decidible, mientras que Turing demostró que la lógica de primer orden es indecidible [5]. Además, la estructura de los grados de Turing nos establece la existencia de una estrecha relación entre los cuantificadores de una fórmula y la dificultad de decidir su veracidad [17].

El razonamiento anterior, aunado al principio de reflexión, nos sugiere que al reforzar un sistema axiomático agregando nuevos axiomas, y consiguientemente movernos dentro del universo de von Neumann, es razonable esperar obtener casos de aceleramiento.

2.1.2. La relación entre el cálculo computacional y el cálculo lógico.

En la sección 1.1.3 se estableció la existencia de aceleramiento *no trivial* para el espacio de las máquinas de Turing: podemos

considerar que el caso análogo para el aceleramiento trivial para este espacio es, dada una función computable f, el incluir en la descripción la codificación de una tabla de evaluación para f. Esta tabla se puede consultar en tiempo lineal. Es claro que, en la mayoría de los casos, esperamos que la existencia de esta tabla de evaluación aumente de manera significativa la complejidad de Kolmogorov de la descripción de la función y que requiera una codificación específica, misma que es muy poco probable que se dé en la mayoría de los miembros del espacio, por lo cual considero que no tiene sentido esperar que el aceleramiento trivial se dé de manera significativa en el espacio de las máquinas de Turing y, en particular, en el espacio de las máquinas de Turing pequeñas.

Ahora, en la literatura se han establecido profundas relaciones entre el cálculo computacional y las demostraciones matemáticas. Un ejemplo notable de estas relaciones es el isomorfismo de Curry-Howard:

> "El isomorfismo de Curry-Howard establece una asombrosa correspondencia entre los sistemas de la lógica formal como se encuentran en la teoría de las demostraciones y el cálculo computacional tal como aparece en la teoría de tipos. Por ejemplo, la lógica proposicional mínima corresponde al cálculo lambda simplemente tipificado, la lógica de primer orden corresponde a los tipos dependientes, etcétera."[15]

Dentro de este contexto, Zenil investigó de manera experimental la relación entre el tiempo de ejecución de las máquinas de Turing de un tamaño en específico (en función de su número de estados) y la longitud de las demostraciones de teoremas en la lógica ecuacional de primer orden. Los resultados encontrados

sugieren que los demostradores automáticos de teoremas son sujetos a *la misma relación no lineal entre la longitud y el tiempo encontrados en las máquinas de Turing* [2].

Dada esta relación, un problema interesante que surge bajo este argumento es el siguiente: mientras que los resultados citados sugieren que las máquinas de Turing tienden a una delta de aceleramiento negativo en una mayoría de los casos, por el lema 1.36 esperamos que todo caso de aceleramiento sea positivo para los sistemas de cálculo lógico.

2.1.3. La naturaleza del cálculo lógico

Podemos pensar en una demostración como una lista consistente de fórmulas lógicas inferidas mediante un conjunto finito de reglas. Ahora, si agregamos una de estas fórmulas a la lista de premisas podemos esperar acortar la longitud de la demostración al eliminar varios de los pasos que fueron requeridos para poder incluir la fórmula en la demostración.

Para definir en términos formales el concepto de "pasos requeridos" es necesario introducir una nueva definición. De acuerdo con la definición (1.29) una demostración de un argumento $t \vdash \gamma_l$ es una sucesión $d = \langle \gamma_1, ..., \gamma_i, ..., \gamma_l \rangle$ de fórmulas. Luego:

Definición 2.4. Consideremos una demostración d y dos fórmulas γ_r, γ_s en d. Decimos que γ_s es *directamente dependiente* de γ_r en d si γ_s es el resultado de aplicar una regla de inferencia $r \in S$ usando γ_r. Denotemos a esta relación por $\gamma_r \prec_d \gamma_s$. Consideremos a la cerradura transitiva de esta relación; diremos que γ_s es *dependiente* de γ_r en d, denotado por $\gamma_r \prec \gamma_s$, si se encuentran relacionados en la cerradura transitiva.

Intuitivamente, podemos decir que una fórmula γ_s *requiere*, para ser incluida en la demostración, de γ_r si $\gamma_r \prec \gamma_s$. Lo que nos lleva al siguiente lema:

Lema 2.5. *Si $d = \langle \gamma_1, ..., \gamma_i, ..., \gamma_l \rangle$ es una demostración mínima del argumento $t \vdash \gamma_l$ entonces, para toda $i < l$, tenemos que $\gamma_i \prec \gamma_l$.*

Demostración. La veracidad de este lema es evidente. Si existiera γ_i tal que no cumple la propiedad descrita, entonces $d - \langle \gamma_i \rangle$ sería una demostración de longitud menor, lo cual es una contradicción. □

Definición 2.6. Consideremos nuevamente a la demostración d. Llamamos a una subsucesión $d_{\gamma_i} = \langle \gamma_{i_1}, ..., \gamma_{i_r} = \gamma_i \rangle \subset d$ una *sucesión de dependencia* en d si, para toda $s < r$, tenemos que $\gamma_{i_s} \prec \gamma_i$. Una sucesión s es *trivial* si todos sus elementos se encuentran en t. De ahora en adelante consideraremos únicamente sucesiones no triviales. Decimos que d_{γ_i} es *completa* si es una demostración para $t \vdash \gamma_i$. Un sucesión completa d_{γ_i} es *única* si no existe otra subsucesión completa que termine en γ_i.

Notemos que por el lema 2.5 tenemos que toda demostración mínima es una sucesión completa única. Además, es claro que dada $\gamma_i \in d$, existe una subsucesión de dependencia completa d_{γ_i}.

Definición 2.7. Decimos que γ_s *depende fuertemente* de γ_r, denotado por $\gamma_r < \gamma_s$, si toda sucesión completa que contiene a γ_s contiene a γ_r. Llamamos a una subsucesión $s = \gamma_{i_1}, ..., \gamma_{i_s}$ un *camino fuerte* si $\gamma_{i_1} < ... < \gamma_{i_s}$ y si, para todas $j, k < s$, no existe γ_m tal que $\gamma_{i_j} < \gamma_m < \gamma_{i_k}$. Si s cumple las mismas condiciones para la dependencia directa $\prec_d$ entonces decimos que s es un *camino débil* o simplemente *camino*. Denotemos por $c(\gamma_i, \gamma_j)$ al camino con extremos γ_i y γ_j. Finalmente, decimos que γ_s

está *acotada* por γ_r, o γ_r es una cota para γ_s, si todo camino $c(\gamma_s, \gamma_l)$ pasa por γ_r. Si la anterior se da para caminos fuertes entonces decimos que γ_r es una *cota débil*. Notemos que toda camino fuerte también es una camino débil y toda cota también es una cota débil.

Proposición 2.8. *Consideremos al argumento $t \vdash \beta$, a d una demostración mínima del argumento y u un conjunto de fórmulas. Si u contiene una cota para algún elemento en d, entonces u induce un aceleramiento positivo, es decir $\delta(t, t \cup u) > 0$.*

Demostración. Sea $\gamma_r \in u$ una cota para γ_s en d. Notemos que, ya que d es una sucesión finita, podemos suponer sin pérdida de generalidad que γ_s es el elemento de índice máximo que está acotado por γ_r.

Sea $d' = d - \{\gamma_s\}$. Demostraré que la sucesión d' es completa para la teoría $t \cup u$ mostrando que todo $\gamma_i \in d'$ cumple con las reglas de construcción dadas en 1.29. En otras palabras, demostraré que la cadena d' es una demostración valida, o *bien fundamentada*, de $t \cup u \vdash \beta$.

Notemos que, ya que γ_r pertenece al conjunto de premisas, la presencia de γ_r está bien fundamentada. Tenemos tres casos:

i) Si $i < s$, no hemos realizado ningún cambio a sus antecedentes, por lo cual $\langle \gamma_1, ..., \gamma_s \rangle$ es una derivación.

ii) Sea $s < i < r$. Si γ_i no cumple con las reglas de construcción entonces no es una derivación de ningún subconjunto de fórmulas con índice menor a i y, ya que γ_s fue el único elemento que quitamos, entonces $\gamma_s < \gamma_i$. Luego, ya que γ_s es el elemento de índice mayor que esta acotado por γ_r entonces existe un camino $c(\gamma_i, \gamma_l)$ que no pasa por γ_r, por

lo cual existe un camino $c(\gamma_s, \gamma_i) \cup c(\gamma_i, \gamma_l) = c(\gamma_s, \gamma_l)$ que no pasa por γ_r, lo que es una contradicción.

iii) Finalmente, sea $i > r$. Igual que el caso anterior, si γ_i no está bien fundamentada entonces $\gamma_s < \gamma_i$. Luego, por el caso *ii*, debe existir un índice mínimo i' mayor a r tal que tal $\gamma_s \prec_d \gamma'_i \leq \gamma_i$, lo que es una contradicción al existir un camino débil entre γ_s y γ'_i que no pasa por γ_r.

Por *i*, *ii* y *iii* tenemos que d' es una demostración para β de longitud menor, y $\delta(t, t \cup u) > 0$. □

El caso anterior no es el único en el cual podemos encontrar aceleración. En particular, si u contiene una cota débil, también podemos esperar aceleramiento positivo en una cantidad importante de casos, o de forma menos directa, si con las nuevas premisas podemos encontrar una cota en un número menor de derivaciones a la cantidad de fórmulas acotadas, entre otras.

Debido a la dificultad del problema, ejemplificada por el resultado 1.34, hemos decidido embarcarnos en una exploración experimental del problema, dejando una exploración analítica de la estructura descrita para trabajos posteriores.

Empiezo por seccionar el espacio de acuerdo a su *complejidad sintáctica* en la siguiente sección.

2.2. Partición del espacio

Para generar el conjunto de todas las fórmulas válidas en el cálculo de proposiciones sin constantes, donde el conjunto de operaciones Op consta de las cuatro conectivas lógicas de deducción natural [16], vamos a tomar un enfoque recursivo al generar subconjuntos de P usando la función recursiva $en : \mathbb{N} \times \mathbb{N} \to \mathbb{P}(P)$

dada por:

$$en(n,m) = \{f(a,b) | f \in \{ \iff , \implies , \wedge, \vee\} \text{ y } a,b \in S\} \ \cup \ S, \tag{2.9}$$

donde

$$S = en(n-1,m) \cup Neg(en(n-1,m)),$$

$$Neg(X) = \{\sim a | a \in X\}.$$

y $en(0,m)$ es el conjunto que contiene a las m primeras variables. Las cuatro conectivas lógicas usadas son conjunción ($\wedge$), disyunción ($\vee$), implicación material ($\implies$) y doble implicación ($\iff$).

En cada instancia, la función en genera un subconjunto de P, donde la variable n representa la profundidad máxima de composición de operadores lógicos binarios y m indica el número máximo de variables distintas que pueden estar presentes en una proposición. Por definición, es claro que $en(n',m') \subseteq en(n,m)$ si y sólo si $n' \leq n$ y $m' \leq m$.

Definición 2.10. Si $\alpha \in en(n,m)$ y $\alpha \notin en(n-1,m)$ definimos a n como la *complejidad sintáctica* de α.

Notemos que la función en nos secciona el espacio P de acuerdo con la complejidad sintáctica. Además, en la definición anterior no he incluido la variable m, la cual podríamos pensar debe de tener un impacto importante en una *medida de complejidad sintáctica*. Sin embargo, la motivación por la cual he decidido incluir otra medida de complejidad en este trabajo es la siguiente:

Consideremos un argumento demostrable de la forma $a_1 : \alpha \vdash \alpha$. Es fácil ver que la teoría $t = \{\alpha\}$ es equivalente a $t' = \{\alpha \wedge ... \wedge \alpha\}$, ambas fórmulas son equivalentes bajo los axiomas del álgebra booleana, y además tenemos que $|t'| = |t|$.

Sin embargo, $|t \vdash \alpha| < |t' \vdash \alpha|$, lo cual puede sugerir una debilidad de la definición dada de complejidad descriptiva de una teoría (1.32).

Una solución a este problema radica en una definición más fuerte de complejidad, en la cual, para calcular la complejidad de cada teoría, es preciso primero transformar cada axioma a su *representación mínima*, la cual se puede definir como la fórmula equivalente de complejidad sintáctica mínima. Una opción distinta radica en la longitud de la expresión. Sin embargo, esta definición presenta nuevos problemas empezando por que el encontrar la representación mínima no es un problema sencillo. Segundo, los axioma de t es son fórmulas válidas en el cálculo de proposiciones y hasta el momento no tenemos razones para pensar que teorías importantes estén libres de cualquier tipo de *redundancia*, término que aún no está bien definido y cuya presencia puede ser más sutil que en el ejemplo dado. Finalmente, tampoco tenemos los elementos para predecir el comportamiento que puede tenerse al sustituir una fórmula por su representación mínima: al igual que establecimos la existencia de un aceleramiento positivo, es fácil encontrar casos de aceleramiento negativo al usar la representación mínima (ej: $\{\alpha \wedge ... \wedge \alpha\} \vdash \alpha \wedge ... \wedge \alpha$).

Sin embargo, la definición 2.10 nos ofrece una manera de acotar la complejidad sintáctica de una teoría:

Definición 2.11. Definimos la *complejidad sintáctica de una teoría* t como

$$max\{s(\alpha) : \alpha \in t\},$$

donde $s(\alpha)$ es la complejidad sintáctica de α.

Notemos que t es de complejidad sintáctica n sólo si $t \subset en(n, m)$ para alguna m. En otras palabras, la función $en(n, m)$ también nos induce una forma de construir y seccionar el espacio

de acuerdo con la complejidad sintáctica definida. Denotemos a cada una de estas secciones por $\mathbb{T}(n, m)$.

Por último, notemos que también podemos definir la complejidad sintáctica de una teoría en términos del axioma de complejidad sintáctica mínima:

Definición 2.12. Definimos la *complejidad sintáctica mínima de una teoría* t como

$$min\{s(\alpha)|\alpha \in t\},$$

donde $s(\alpha)$ es la complejidad sintáctica de α.

2.3. Cómo computar $\mathbb{T}$

Ya que la función *en* es una función recursiva primitiva sobre conjuntos finitos, tenemos que *en* es una función computable [12].

El lenguaje de programación funcional Haskell nos ofrece una forma natural de definir la función *en*:

```
data Expresion = Lit Int | And Expresion Expresion |
                 Or Expresion Expresion |
                 Impl Expresion Expresion |
                 DImpl Expresion Expresion |
                 Not Expresion

en 0 n = [(Lit 1)..(Lit n)]

en n m = [c (a, b) |c <- conj, a <- s, b <- s ] ++ s
         where s = ant ++ (map fNeg ant);
               ant = en ( n - 1) m;
               fNeg x = Not x
```

```
conj = [\(x,y) -> And x y, \(x,y) -> Or x y,
        \(x,y) -> Impl x y, \(x,y) -> DImpl x y ]
```

La estructura de datos producida por la definición de la función *en* es una lista simplemente ligada, estructura que nos induce un orden total en $en(n, m)$ y por ende una enumeración para P.

Definición 2.13. Dados n y m la función $ax(n, m, k)$ nos da la proposición que ocupa la k-ésima posición en la lista $en(n, m)$. En Haskell esto es:

```
ax n m k = en n m !! ( k - 1 )
```

Ahora, consideremos a la función $f : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$, la cual esta definida como

$$f(n, m) = |en(n, m)|, \tag{2.14}$$

donde $|X|$ es la cardinalidad del conjunto X.

Recordemos que una teoría se define como un conjunto de fórmulas, a cada una de las cuales llamamos axiomas. Para *enumerar las teorías* aprovecho el orden de las fórmulas establecido con anterioridad: consideremos una cadena o palabra de $f(n, m)$ bits, donde el x-ésimo bit es uno si y sólo si la x-ésima fórmula forma parte de los axiomas de la teoría. Estas cadenas corresponden a la representación binaria de un número natural k en notación *little-endian*[1], número que representa la k-esíma teoría, para un total de $2^{f(n,m)}$ teorías posibles. Denotemos a la k-ésima teoría que cumple con n y m por $T(n, m, k)$.

[1]La notación little-endian corresponde al inverso de la notación *natural*. Es decir, los dígitos más significativos se encuentran a la derecha.

Definición 2.15. Dados n y m definimos a la teoría k, denotada por la función $T(n, m, k)$, por la teoría que ocupa la k-ésima posición en la lista $\mathbb{T}(n, m)$. En Haskell esto es:

```
T k n x = T' (decToBin' k) (en n x)

T' (x:xs) (y:ys) = if (x /= 0)
                        then y:(T' xs ys)
                        else (T' xs ys)

T' x [] = []
T' [] y = []
```

donde $decToBin'\ k$ es una lista con la representación binaria de k en notación *little-endian*.

Con el desarrollo expuesto en este capítulo tenemos una forma de construir, seccionar y enumerar a los espacios P y $\mathbb{T}$. Así como una forma de construir y seccionar a $\mathbb{S}$, el espacio de todos los argumentos: consideremos a la función $arg : \mathbb{N}^4 \to \mathbb{S}$ definida como:

$$arg(n, m, k, x) = T(n, m, k) \vdash ax(n, m, x),$$

notemos que podemos generar el espacio de todos los argumentos en $en(n, m)$ variando los parámetros k y x desde 0 hasta los límites $f(n, m)$ y $2^{f(n,m)}$ respectivamente. Denotemos por $\mathbb{S}(n, m)$ a este espacio.

No daré la función de enumeración para $\mathbb{S}$ de manera explícita por razones que explicaré en secciones subsecuentes.

2.3.1. La intratabilidad de $\mathbb{S}$

Decimos que un algoritmo es *computacionalmente intratable* si es computable, pero en la práctica no existen recursos computacionales (tiempo y memoria) suficientes para su ejecución

[18].

Dados sus requerimientos es fácil ver que la función f, la cual nos dice el número de elementos en cada instancia $en(n, m)$, está dada por la siguiente definición:

$$\begin{aligned} f(0, m) &= m \\ f(n, m) &= 4|S|^2 + |S| \\ &= 4(2f(n-1, m))^2 + 2f(n-1, m) \\ &= 16f(n-1, m)^2 + 2f(n-1, m). \end{aligned}$$

Esta función tiene un orden de crecimiento

$$\sim \frac{(16m)^{2^n}}{16},$$

por lo que el espacio $\mathbb{T}(n, m)$, el cual esta compuesto por todos los subconjuntos de $en(n, m)$, tiene un orden de crecimiento

$$\sim 2^{\frac{(16m)^{2^n}}{16}}.$$

Por último, el espacio $\mathbb{S}(n, m)$, tiene un orden de crecimiento de

$$\sim \frac{(16m)^{2^n}(2^{\frac{(16m)^{2^n}}{16}})}{256}.$$

el cual es un crecimiento de orden triple exponencial.

Ahora, para $n, m > 1$ obtuvimos que el número de elementos en $en(n, m)$ es mayor a $74,000$, en $\mathbb{T}(n, m)$ es superior a $2^{74,000}$ y la cardinalidad de $\mathbb{S}$ esta por encima de $74,000 \times 2^{74,000}$. Si tomamos 0̈1 segundos en generar y encontrar una demostración para cada argumento, entonces tardaremos mas de

$$\begin{aligned} 0\ddot{1} \times 74000 \times 2^{74000} \text{ segundos} &\approx 3\ddot{8}9138 \times 10^{22272} \text{ años} \\ &\approx 2\ddot{8} \times 10^{22262} \times U, \end{aligned}$$

donde U es la edad del universo conocido, en explorar todo el espacio [2]. Por lo anterior podemos concluir que, aún para valores pequeños de n y m, el espacio $\mathbb{S}(n, m)$ es claramente computacionalmente intratable para una búsqueda exhaustiva, razón por la cual se propuso realizar un muestreo estadístico sobre todo el espectro de las variables k y x (con sus límites correspondientes) para un conjunto de argumentos de $en(n, m)$, usando un demostrador automatizado para encontrar cada demostración.

2.3.2. La sección de $\mathbb{T}(n, m)$.

Otro desafío de computabilidad es presentado por la definición de una teoría (1.26): dentro del espacio $\mathbb{T}(n, m)$ tenemos teorías como $T(2, 2, 2^{50000} - 1)$, la cual cuenta con 50,000 axiomas, un número demasiado grande para ser tratado por un demostrador automatizado, por lo que necesitamos una forma de limitar el número de axiomas en una teoría.

Para solucionar este problema agregué el parámetro j en la enumeración de $\mathbb{T}$, el cual indica el número de axiomas de los cuales consta cada teoría, es decir su complejidad descriptiva. Bajo esta esquema, únicamente consideraremos cadenas binarias para k cuyo peso de Hamming[3], denotado por $h(k)$, es j. Esto es:

Definición 2.16. Dados n, m y j definimos a la teoría k, denotada por la función $T(n, m, j, k)$, por la teoría que ocupa la k-ésima posición en la lista $\mathbb{T}(n, m, j)$, donde

$$\mathbb{T}(n, m, j) = \{t | t \in \mathbb{T}(n, m) \text{ y } |t| = j\}.$$

[2] Cálculos realizados por http://www.wolframalpha.com

[3] El peso de Hamming se define como el número de caracteres no cero en una cadena.

Una forma eficiente de representar estas teorías es por medio de la n-ada $T(n, m, k, j) = (n, m, r_1, ..., r_j)$, donde r_i corresponde al número de ceros antes del i-ésimo 1. Por ejemplo, la tupla $(n, m, 0, 1, 2)$ corresponde a la cadena binaria $1r_3 1r_2 1r_1 = 100101$. De esta forma podemos cubrir todas las teorías cuya representación binaria tiene peso de Hamming j ocupando un cantidad de memoria relativamente pequeña.

En Haskell es:

```
data Teoria = Palabra [Int] Int Int |
              TEntero Integer Int Int

palATeo x = T k n m
            where TEntero k n m = palAtEnt x
```

donde *palAtEnt* x implementa la función inversa a la descrita en el párrafo anterior. *palabra* es la estructura de datos cuyas instancias son las teorías en la representación descrita y $TEntero$ es la representación descrita en 2.15.

Notemos que la función $\mathbb{T}(n, m, j)$ nos induce una partición (de conjuntos ajenos) en $\mathbb{T}(n, m)$, y que al visitar todas las tuplas de la forma $(n, m, r_1, ..., r_j)$, dentro de sus límites correspondientes, obtenemos de nuevo todo el espacio, donde el límite para las entradas de una tupla $(r_1, ..., r_j)$ está dado por

$$\sum_{i=0}^{j} r_j \leq f(n, m) - j.$$

La enumeración en $\mathbb{T}(n, m, j)$ es la heredada por el orden en $\mathbb{T}(n, m)$. Es decir, para $t, t' \in \mathbb{T}(n, m, j)$, decimos que $t < t'$ si y

sólo si $t < t'$ en $\mathbb{T}(n, m)$.

De ahora en adelante consideraremos que, en toda teoría denotada por $T(n, m, k)$, k es una lista de enteros. También usaremos la notación $T(k)$ al referirnos a una teoría $T(n, m, k)$ dentro del contexto de valores establecidos para n y m.

Capítulo 3

El Experimento

La técnica estadística experimental que se va ha emplear es la técnica de muestreo. Esta técnica es empleada cuando el análisis completo de fenómenos, representados como conjuntos discretos, resulta demasiado complejo o incluso imposible, como es en nuestro caso. Esta técnica consiste en representar la serie completa de valores parciales por medio de una parte del conjunto original. Este subconjunto, llamado muestra, debe ser de un tamaño manejable pero lo suficientemente grande para poder obtener la información requerida con una precisión adecuada [19].

En particular, usaremos las técnicas de muestreo llamadas muestreo aleatorio simple y muestreo por conglomerados.

Definición 3.1. Llamaremos a un conjunto ordenado $\langle a_i \rangle$ una *población* y a un subconjunto ordenado $\{a_{j_1}, ..., a_{j_r}\}$ una *muestra* extraída de la población.

Richard Levin [20] define la técnica de muestreo aleatorio simple como aquella que consiste en seleccionar los elementos del conjunto muestra por "*métodos que le permiten a cada muestra posible tener igual probabilidad de ser tomada y a cada elemento*

de toda la población tener una misma probabilidad de ser incluido en la muestra"; y el muestreo por conglomerados como el "*dividir a la población en subconjuntos o conglomerados, entre los cuales se selecciona una muestra aleatoria de estos*". En donde para cada conglomerado realizamos un muestreo aleatorio simple.

Finalmente, una muestra se considera *representativa* si las características de la muestra por estudiar coinciden con las características de la población con un *margen de error aceptable*.

3.1. Diseño del experimento

El experimento que propongo consiste en, dados n y m, generar de manera aleatoria dos conjuntos:

- Un conjunto muestra O de $en(n, m)$, generado por un muestreo aleatorio simple, y
- un conjunto muestra T de $\mathbb{T}(n, m)$, seleccionado usando la combinación del muestreo simple y por conglomerados;

donde, para todo $t \in T$ y $\alpha \in O$, tenemos que el argumento $t \vdash \alpha$ es demostrable.

Una vez generados los conjuntos procedemos a registrar la longitud de las demostraciones de los teoremas. Posteriormente agregamos, usando algún orden, nuevos axiomas y registramos nuevamente los resultados, obteniendo en cada caso un valor de aceleramiento.

En las siguientes secciones se explicará de manera detallada el algoritmo de selección de los conjuntos muestra.

3.1.1. La selección en $en(n, m)$

Para generar los conjuntos muestra para cada subespacio $en(n, m)$ usamos una selección aleatoria simple. Dados los números naturales n y m recordemos que, para cada instancia de la función en, tenemos definida una enumeración 2.13, la cual le asigna a cada elemento de $en(n, m)$ un número natural k, el cual varía desde el número 1 hasta el límite $f(n, m)$.

Ahora, si o es el número de elementos que requerimos para la muestra, usando el generador de números aleatorios de Haskell podemos general una lista pseudoaleatoria de números enteros compuestos de valores elegidos dentro de los límites 1 y $f(n, m)$. El código usado es el siguiente:

```
eleccion = take o (randomRs (1 :: Int,
                                  limite :: Int) g)
```

donde el valor de la variable *limite* es $f(n, m)$ y la función *randomRs* genera una lista infinita de números aleatorios dentro de los límites definidos por la pareja ordenada usando la semilla g, lista de la cual tomamos un número o de elementos. De acuerdo con la especificación de Haskell, cada uno de los valores devueltos por *randomRs* se encuentra "*uniformemente distribuido*"dentro de los límites dados, es decir que todos los valores dentro de los límites tienen la misma probabilidad de ser elegidos para la muestra, por lo cual es una muestra aleatoria válida.

Finalmente, reemplazando cada entero por su fórmula correspondiente obtenemos una muestra aleatoria de $en(n, m)$ con o elementos.

Procedemos a evaluar cada fórmula, removiendo tautologías y fórmulas no satisfactibles usando las siguientes funciones definidas en Haskell:

```
-- Decide si una fórmula es satisfactible.

sat :: Expresion -> Int -> Bool
sat e m = sat' e (Mismo $ ceros m )

sat' e (Mismo l) | ev = True
                 | otherwise = sat' e ( sucB l)
                 where ev = eval e $ binToBool l;
sat' e (Cambio l ) = eval e $ binToBool l

-- Decide si una fórmula es tautologia.

isTaut :: Expresion ->Int-> Bool isTaut

e m = isTaut' e (Mismo $ ceros m )

isTaut' e (Mismo l) | ev = isTaut' e ( sucB l)
                    | otherwise = False
                    where ev = eval e $ binToBool l;

isTaut' e (Cambio l ) = True
```

donde *Mismo* y *Cambio* son funciones constructoras para una instancia del tipo de dato *Orden a*, el cual fue definido para especificar los casos en los cuáles se ha producido un acarreo en una suma de listas binarias; la función *ceros* genera una lista de constantes 0 de longitud m; la función *binToBool* transforma una lista binaria a una lista de valores booleanos realizando la correspondencia natural[1]; *sucB* encuentra el sucesor de una lista binaria, regresando el resultado en términos del tipo de dato *Orden*; y la función *eval* regresa el resultado de la evaluación de una fórmula (definida por el tipo de dato *Expresion*) y una lista

[1]La correspondencia natural es $1 \mapsto True$ y $0 \mapsto False$

binaria. La función evaluación usada fue definida formalmente en la demostración 1.24, y su implementación en Haskell es simple.

3.1.2. La selección en $\mathbb{T}(n, m)$

Como hemos visto en la sección 2.3.2, el limitar el espacio de teorías $\mathbb{T}$ con j nos permite codificar el índice k para cada teoría $T(n, m, k)$ por una lista de j enteros. En esta lista, cada elemento representa el número de ceros entre los dígitos de valor 1 en la representación binaria de k (en notación little-endian).

La definición de esta estructura tiene dos objetivos:

1. Nos permite evitar el tener que generar, recorrer y guardar en memoria listas de longitud del orden de $O(f(n, m))$. En su lugar generamos listas de longitud fija j, donde cada elemento ocupa 32-bits en memoria.

2. Segundo, nos provee de una forma natural de generar teorías pseudoaleatorias al asignar valores pseudoaleatorios a cada valor de j, y por consiguiente, argumentos pseudoaleatorios.

Es la segunda razón por la cual evité definir la función de enumeración de manera explícita para $\mathbb{S}$ y $\mathbb{T}(n, m, j)$.

Dada una teoría $t = T(n, m, j, k)$ consideremos la lista de enteros positivos $k = [k_1, ..., k_j]$ y la suma de sus valores

$$gs(k) = \sum_{i=1}^{j} k_i.$$

El valor de $g(k)$ nos indica el orden del dígito más significativo en la representación binaria de k. En otras palabras, $g(k)$ es el

grado de separación entre la primera teoría en la enumeración de $\mathbb{T}(n, m, j)$, codificada por la lista $[0, 0, ..., 0]$ con j elementos, y la teoría representada por k, definida por la diferencia entre el orden de los bits más significativos de la representación binaria de la enumeración de la teoría t.

Definición 3.2. Dado $\mathbb{T}(n, m, j)$, definimos una *clase de separación* $[g]$ en el espacio como el conjunto:

$$[g] = \{t(k)|gs(k) = g\}.$$

Finalmente, si $t = T(k)$, denotaremos a $g(k)$ por $g(t)$.

Recordemos que, por la construcción del espacio (2.9), la posición asignada por la enumeración es dependiente de la complejidad sintáctica, las variables y las conectivas lógicas presentes, por lo cual el grado de separación nos mide qué *tan distintas* son dos teorías en términos del axioma $ax(gs(k))$. En otros términos, ya que en la representación binaria de una teoría el dígito más significativo corresponde al axioma de índice mayor, podemos decir que si t y t' pertenecen a la clase $[g]$ entonces ambas teorías tienen la misma complejidad sintáctica mínima. Es decir, los elementos de $[g]$ están acotados por la complejidad de $ax([g])$.

El desarrollo presentado en el párrafo anterior, aunado al tamaño doble exponencial de los subespacios $\mathbb{T}(n, m)$, es la razón por la cual decidí usar un muestreo por conglomerados, definiendo a cada clase $[g]$ como un conglomerado.

Notemos que el número total de clases distintas en $\mathbb{T}(n, m, j)$ es $f(n, m) - j$, el cual es un número de magnitud significativa para $n, m > 1$, por lo que decidí implementar primero la opción de realizar un muestreo simple sobre el conjunto ordenado de clases usando el mismo método que se usó para obtener la muestra en $en(n, m)$. En Haskell esto es:

```
eleccionG = take x (randomRs (1 :: Int,
                                limiteC :: Int) g)
```

donde x es el número requerido para los elementos de la muestra de clases y $limiteC$ es el número de $f(n,m) - j$ clases distintas.

Ahora, sea G la muestra de clases de separación a usar. Para cada $g \in G$ necesitamos una muestra de teorías elegidas aleatoriamente. Recordemos que, dada la definición de una teoría por medio de listas de enteros, podemos generar teorías aleatorias al asignar valores aleatorios a cada entrada de una lista de j elementos, pero ahora tenemos la condición de pertenencia a $[g]$, la cual implica que la suma de todos los elementos de la lista sea g, razón por la cual empiezo por general una lista de r números que cumplan con la condición usando la siguiente función en Haskell:

```
listaRand j rP s  = trandLoop j rP s' 0
                    where
                        g = mkStdGen s;
                        s' = fst (next g)

trandLoop j acc s c =
    x : (trandLoop (j - 1) (acc - x) s' (c + 1) )
       where
         g = mkStdGen s;
         gen = randomR (fromIntegral 0,
                  fromIntegral acc) g;
         x = fst (gen);
         s' = fst (next g)

trandLoop 1 acc s c = [acc]
```

En este código la función *listaRand* recibe los parámetros:

- j, el cual sabemos que parametriza el número de elementos de la lista a generar;

- rP, valor que hemos denotado por g hasta el momento, el cual define la clase de separación; y por último

- s, una semilla para el generador de valores pseudoaleatorios $randomR$, función que nos regresa un entero pseudoaleatorio dentro de los limites establecidos por la pareja ordenada que recibe.

La función recursiva *trandLoop* genera y guarda en una lista un valor pseudoaleatorio dentro de los límites 0 y *acc*, donde la variable *acc* es el valor pseudoaleatorio generado en la iteración anterior. Esto es, empezando por el valor rP para *acc*, generamos primero un valor aleatorio x entre 0 y *acc*; guardamos x en una lista y procedemos a llamar a la misma función, ahora con $acc - x$ como límite. Seguimos iterando por un número j de veces obteniendo una lista de j valores aleatorios cuya suma es rP. Con respecto a las demás funciones presentes en el código, notemos que generamos nuevas semillas de manera aleatoria tras cada iteración usando las funciones $mkStdGen\ s$ y $fst\ (next\ g)$.

Después, para reducir la posibilidad de un sesgo sistemático inducido por la dependencia de valores en la definición de la función anterior, usamos la siguiente función:

```
tuplaRand j g s = tuplaRand' xs ys
    where
        xs = listaRand j g s
        ys = fst ( shuffle [0..(j - 1)] s' )
        s' = fst (next (mkStdGen s) )


tuplaRand' _ [] = []
```

```
tuplaRand' (xs) (i:is) =
    (xs !! i):tuplaRand' (xs) (is)
```

la cual usa la función *shuffle* para aleatorizar el orden de la lista generada por $listaRand$.

3.1.3. Evitar sistemas inconsistentes

Existe un caso específico que consideramos importante evitar: el caso de teorías inconsistentes.

Recordando la definición 1.26, decimos que a una teoría $t = \{\alpha_i, ..., \alpha_j\}$ es inconsistente si no es satisfactible. Ya que en el espacio generado por $en(n, m)$ contamos con la conjunción lógica '$\wedge$', podemos decir que t es inconsistente si, para toda evaluación $\overline{b}$, tenemos que $\alpha_i(\overline{b}) \wedge , ..., \wedge\alpha_j(\overline{b}) = F$. En particular, la constante F es un teorema para t.

Ahora, en deducción natural y otros sistemas de cálculo lógico tenemos la siguiente regla de inferencia:

$$\text{Fe} \quad \frac{F}{\alpha}$$

La cual nos dice que podemos concluir cualquier fórmula a partir de la constante F [2].

Esta regla, conocida en lógica clásica como el *principio de explosión*, implica que si t es una teoría inconsistente entonces, para toda β, la demostración del argumento $t \vdash \beta$ se reduce a una aplicación de Fe, lo que consideramos un caso indeseable

[2]Para el cálculo de proposiciones sin constantes podemos usar la regla equivalente $\frac{\alpha \wedge \sim\alpha}{\beta}$

para su estudio. Es por eso que, una vez obtenido el conjunto muestra T, recorremos toda la lista de teorías en busca de satisfactibilidad, removiendo de la lista las teorías inconsistentes.

Debido a todas los operaciones requeridas para la generación (y posterior demostración) de los conjuntos muestra, es evidente que el tamaño de la muestra está limitado por los recursos computacionales disponibles para la realización de este experimento. Es por esto que el proceso de filtrado de teorías inconsistentes presenta nuevos problemas, ya que esperamos que un número importante (mayoritario inclusive) de teorías sean inconsistentes, dependiendo de las tres variables m, n y j (la complejidad sintáctica, el número de variables y el número de axiomas de cada teoría respectivamente). Razón por la cual me dispuse a buscar un *rendimiento* en función del número de teorías generadas y aquellas consideradas utilizables.

Con este objetivo se realizaron 32 experimentos con muestras de 1000 elementos cada uno, todos con complejidad sintáctica máxima de 2. En particular, encontramos que la distribución de teorías consistentes observa el siguiente par de comportamientos:

- Un crecimiento lineal con respecto al número máximo de variables presentes en cada uno de los axiomas y
- un decrecimiento hiperbólico con respecto al número de axiomas o complejidad descriptiva de las teorías estudiadas.

Es decir, como es de esperarse, el número de teorías consistentes en el espacio $\mathbb{T}(n, m, j)$ aumenta de manera aparentemente lineal con respecto a m y decrece de forma acotada asintóticamente con respecto a j.

Para el espacio $\mathbb{T}(2, 4, 4)$ obtenemos un rendimiento cercano al 38 %, el cual consideré adecuado para una primer exploración

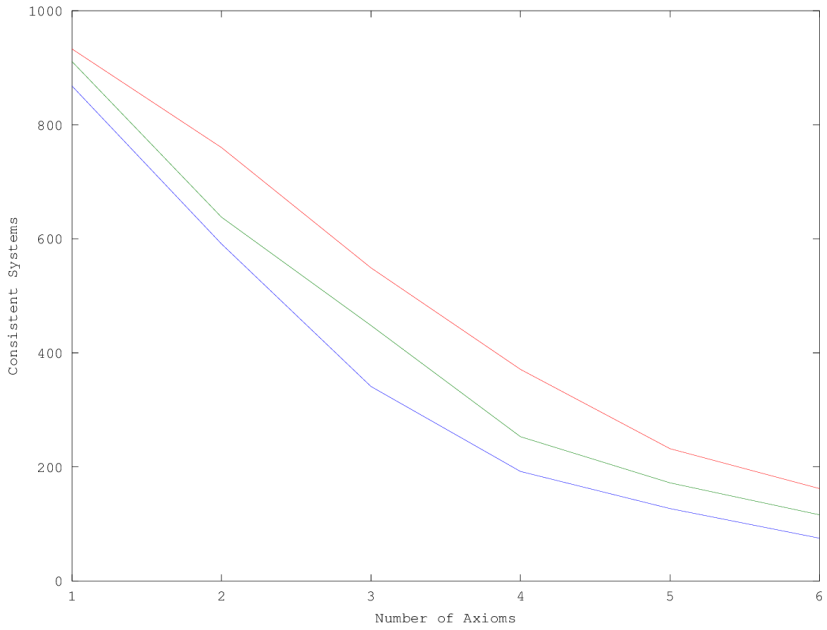

Figura 3.1: Número de teorías consistentes con respecto a la complejidad descriptiva de las teorías. Notamos un comportamiento hiperbólico.

del espacio. La complejidad sintáctica de las proposiciones se fijó a un máximo de dos ya que este último valor es el que considero que presenta una complejidad sintáctica interesante y un orden de población manejable en cuanto a la generación de muestras.

Los resultados de 18 de los experimentos se presentan en las siguientes páginas en forma de una tabla y dos gráficas.

n	m	j	T. Consistentes	Porcentaje
2	2	1	868	86.8 %
2	2	2	591	59.1 %
2	2	3	341	34.1 %
2	2	4	192	19.2 %
2	2	5	127	12.7 %
2	2	6	75	7.5 %
2	3	1	911	91.1 %
2	3	2	638	63.8 %
2	3	3	448	44.8 %
2	3	4	253	25.3 %
2	3	5	172	17.2 %
2	3	6	116	11.6 %
2	4	1	933	93.3 %
2	4	2	760	76 %
2	4	3	549	54.9 %
2	4	4	371	37.1 %
2	4	5	232	23.2 %
2	4	6	162	16.2 %

Cuadro 3.1: Tabla de teorías consistentes: nos dice el número de teorías consistentes encontradas y el porcentaje de rendimiento de un total de 1000 para cada caso, donde n es la complejidad sintáctica, m es el número de variables y j es el la complejidad descriptiva (número de axiomas).

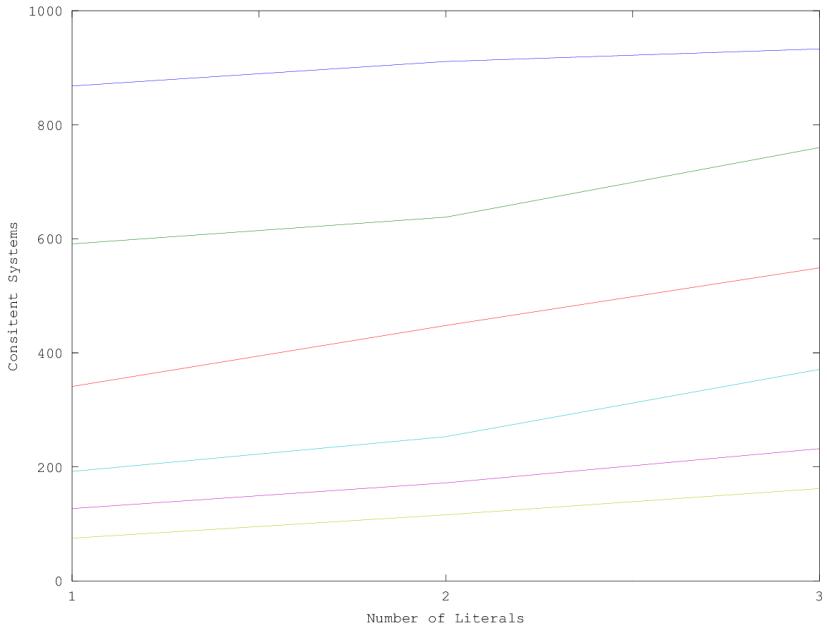

Figura 3.2: Número de teorías consistentes con respecto al número máximo de variables. Notamos un crecimiento lineal, el cual es esperado ya que al aumentar el número de variables disminuimos la probabilidad de *colisiones* (negaciones) entre estas.

3.1.4. La selección en $\mathbb{S}(n, m)$

Una vez obtenidos los conjuntos muestra O y T procedemos a construir una lista llamada *Experimento* donde cada uno de sus elementos es un argumento. Los argumentos están definidos en Haskell por la siguiente estructura:

```
data Experimento = Casos [Expresion] Expresion
```

Aunque formalmente, dada la definición anterior, cada *Casos* genera una instancia de *Experimento*, me tomé la libertad de referirme a una lista del tipo de dato *Experimento* como simplemente experimento, así que consideraremos un experimento como una lista de casos.

El proceso de construcción de un experimento a partir de las listas O y T es el siguiente:

Sean $T = [t_1, ..., t_x]$ y $O = [O_1, ..., O_o]$ dos listas y t_b un elemento de T. Consideremos a la lista de todos los prefijos de la lista O

$$S = \{S_i | S_i \text{ es un prefijo de O}\}. \tag{3.3}$$

Procedemos a concatenar cada uno de los prefijos con la lista t_b, generando una nueva lista de teorias T'_b de longitud o. Finalmente, emparejamos cada elemento de la lista T'_b con cada elemento de la lista o, generando un argumento en cada caso.

En otras palabras, para cada $t_b \in T$ la lista de argumentos generada es la siguiente:

$$\{t_b \vdash O_1, ..., t_b \vdash O_o,$$
$$t_b +\!\!+ [O_1] \vdash O_1, ..., t_b +\!\!+ [O_1] \vdash o_o,$$
$$...$$
$$t_b +\!\!+ [O_1, ..., O_o] \vdash, O_1, ..., t_b +\!\!+ [O_1, ..., O_\alpha] \vdash O_o\}.$$

donde '++' es la concatenación de listas, para un total de $x(o+1)$ argumentos generados. En Haskell esto es:

```
casosIt ts os = [Casos ( t ++ y ) o |
                 t <- ts, y <- (subLC os), o<-os]
```

donde la función *subLC* regresa una lista que consta de todos los prefijos de la lista *os*. Llamamos a la teoría t_b la *teoría base*, a las teorías de la forma $t_b : [O_1, ..., O_k]$ las *teorías generadas* a partir de t_b y a la lista $[O_1, ..., O_k]$ la *lista añadida* de la teoría generada correspondiente. Conoceremos a la lista O como la *lista de objetivos* y llamaremos *objetivo* a cada uno de sus elementos.

Finalmente, limpiamos la lista quitándole los casos no válidos, usando la siguiente función:

```
-- Decide si un caso es válido.

isProv (Casos [] o) m = isTaut o m

isProv (Casos ts o) m = isTaut (Impl e o) m
                        where e = ft ts


-- Dada una lista de casos, regresa una lista
-- de casos válidos.
```

```
isProvL (c:cs) m | ev = c : isProvL cs m
                 | otherwise = isProvL cs m
                 where ev = isProv c m

isProvL [] _ = []
```

Llamamos a la lista completa de argumentos resultante un *experimento*. Notemos que un experimento es una muestra para $\mathbb{S}(n, m, j)$.

Es importante notar que un experimento contiene un número importante de *argumentos triviales*, donde decimos que un argumento $t \vdash O$ es trivial si $O \in t$. Es fácil ver que estos argumentos triviales inducen un caso de aceleramiento trivial. Esta inclusión es por diseño.

3.2. Demostradores automatizados

En general, el algoritmo desarrollado genera muestras aleatorias para un subespacio de $\mathbb{S}$ a partir de seis números naturales: los primeros tres valores, denotados por n, m, y j, determinan el subespacio $\mathbb{S}(n, m, j)$, del cual se van ha obtener las muestras. Las variables x y o determinan el número máximo de elementos en la muestra, el cual es $x(o^2 + o)$. Finalmente, la variable s sirve para inicializar el generador de números pseudoaleatorios.

Una vez obtenido un *experimento*, el paso siguiente es determinar la longitud de la demostración (mínima) de cada argumento. El tamaño de las muestras está determinado por los recursos de cómputo disponibles y en general estamos hablando de varios miles de argumentos demostrables, por lo cual es necesario encontrar algún método para automatizar este proceso. El

método propuesto es el uso de demostradores automatizados de teoremas (o ATP por sus siglas en inglés).

Los sistema ATP son programas de computadora diseñados para encontrar demostraciones de argumentos en uno o varios sistemas de cálculo lógico.

Existen varios sistemas ATP disponibles, entre los que se encuentran APROS, OTTER, E, SPASS, VAMPIRE y WALDMEISTER, los cuales trabajan con lógicas de primer orden.

El experimento fue diseñado desde un principio para el sistema APROS (Automated Proof Search), diseñado bajo la dirección de Wilfried Sieg del Departamento de Filosofía de la Universidad de Carnegie Mellon. APROS usa el método conocido como *intercalación* [26] para generar demostraciones con deducción natural para la cálculo de proposiciones y la lógica de predicados. Sin embargo, debido a una series de dificultades técnicas con APROS descritas en este capítulo, se decidió también usar un segundo sistema ATP.

El segundo sistema elegido es PROVER9. Este último es un sistema ATP desarrollado por William McCune, de la Universidad de Nuevo México, el cual usa la regla de inferencia conocida como resolución binaria (y formas especializadas de esta llamadas hiperresolución y UR-resolución) [3] y paramodulación[4] para encontrar demostraciones en lógica ecuacional y de primer orden. Este sistema de inferencia es correcto para lógica ecua-

[3]Los ATP usan la regla resolución para obtener demostraciones por contradicción

[4]Paramodulación es una regla para sistemas de cálculo con igualdad, la cual infiere una versión equivalente de una fórmula con igualdad dada otra fórmula.

cional de primer orden y completo para el cálculo de predicados.

El programa desarrollado, dado un experimento E, procede a traducir cada instancia del tipo de dato *Experimento* al formato de entrada requerido por APROS y PROVER9:

- APROS lee archivos de texto de extensión '.aps' con un formato en XML. Estos archivos pueden contener, en principio, un número ilimitado de casos. Sin embargo, encontré que APROS se torna inestable si el archivo sobrepasa los 100MB de peso, por cual el programa desarrollado construye un archivo '.aps' por cada mil casos de un experimento.

  La forma de interactuar con APROS es por medio de una interfaz gráfica, por lo que la selección de cada archivo es manual.

  Como salida, para cada '.aps' APROS genera un archivo en formato '.csv' con información sobre el tiempo y la longitud de la demostración para cada caso. Este archivo puede ser leído por una aplicación de hoja de cálculo.

- PROVER9 recibe archivos de texto sin requerimientos de extensión, los cuales deben de estar en un formato específico para PROVER9, donde se especifica los argumentos a demostrar y varias opciones para el demostrador. Estos archivos únicamente pueden contener un conjunto de premisas, por lo cual el programa desarrollado genera un archivo de entrada para PROVER9 por cada caso. De acuerdo con los parámetros seleccionados, PROVER9 puede encontrar una o varias demostraciones.

Sin embargo, al contrario de APROS, PROVER9 puede ser controlado desde la línea de comandos del sistema operativo empleado, por lo cual todo el proceso puede ser automatizado dado un manejo de memoria adecuado.

Los archivo de salida generados por PROVER9 para cada caso contiene la entrada completa, la demostraciones e información adicional para cada demostración, así como el tiempo total de ejecución, el número de cláusulas usadas y la longitud de la demostración.

Dado un experimento E, llamaremos al conjunto de archivos de salida generados por un demostrador los *archivos de datos* para el experimento E.

3.2.1. El aceleramiento relativo a un ATP

Otra diferencia importante entre APROS y PROVER9 es que, mientras APROS genera una sola demostración para cada entrada, los archivos de entrada de PROVER9 pueden contener la etiqueta

```
assign(max_proofs, N)
```

la cual especifica un número máximo de demostraciones a generar. El conjunto de demostraciones generadas a partir de un mismo argumento suele estar compuesto de demostraciones de longitudes distintas; situación que, aunada al teorema 1.34, nos lleva al siguiente punto: *No tenemos argumentos para pensar que los sistemas* ATP *usados nos den la demostración de longitud mínima.*

Sin embargo, podemos pensar en la complejidad de demostración *relativa* a un sistema ATP:

Definición 3.4. Sean A un sistema ATP y $t \vdash \beta$ un argumento en un sistema de demostración empleado por A. Definimos la *complejidad de demostración relativa* a A (del argumento), denotado por $D_A(t \vdash \beta)$, como la longitud mínima de las demostraciones encontradas por A para $t \vdash \beta$.

También podemos definir el concepto de aceleramiento relativo:

Definición 3.5. Sean A un sistema ATP, $t \vdash \beta$ un argumento demostrable en A, t y t' dos descripciones de $[t]$ tales que $|t| < |t'|$. Llamamos *delta de aceleramiento relativo a* A de $t \vdash \beta$ entre t y t' , o simplemente *aceleramiento relativo*, a la función:

$$\delta_A(t, t') = 1 - \frac{D_A(t' \vdash \beta)}{D_A(t \vdash \beta)}.$$

Ahora, notemos que encontrar un resultado análogo al teorema de la invariancia 1.11 para sistemas de cálculo lógico distintos es poco factible: la demostración del teorema de invariancia se basa en el hecho de que, dadas dos computadoras universales, es posible construir un *traductor universal* (compilador) de un lenguaje a otro; y el concepto de un compilador no tiene sentido dentro del cálculo lógico: no hay forma de procesar expresiones que no son fórmulas bien formadas dentro de un sistema de demostración. Sin embargo, podemos pensar en la existencia de una invariancia entre un sistema de cálculo lógico y un programa ATP que lo automatiza. Si es posible acotar esta invariancia de una forma *efectiva* entonces podemos decir que la aproximación del aceleramiento dada por el sistema ATP es adecuada.

En la literatura existe ya un ejemplo de aproximación computacional a una función no computable de complejidad: la *distancia de compresión normalizada* (NCD por sus siglas en inglés), la cual emplea sistemas de compresión sin pérdida para

aproximar la función llamada *distancia de información normalizada*, la cual no es computable [23]. Para esta distancia tenemos que un compresor es *normal* si cumple con una serie de axiomas que definen el *comportamiento correcto* de un compresor sin pérdida hasta un término logarítmico aditivo.

Para este trabajo usaré este antecedente como punto de referencia para definir formalmente el comportamiento que esperamos de una aproximación efectiva. En particular, emplearé el lema 1.36 como base para definir el concepto de un *comportamiento correcto* para el espacio de sistemas de demostración en función de la delta de aceleramiento.

Formalmente:

Definición 3.6. Sean L un sistema de cálculo lógico, A un sistema ATP para L, $t \vdash \beta$ un argumento demostrable en A y t' una descripción para $[t]$ tal que $t \subset t'$. Llamamos a una función $\epsilon_t : \mathbb{N} \to \mathbb{R}$ una cota para A con respecto a t si

$$\delta_A(t, t') \geq 0 + \epsilon_t(|t'|).$$

Ahora, el problema que tenemos es el de determinar un límite *adecuado* para la cota $\epsilon_t(|t'|)$. Para el caso de NCD, esta cota es una función de orden logarítmico [23]. Este límite se obtiene a partir de *regla de la cadena* para la complejidad de Kolmogorov, la cual nos dice que

$$K(s,t) = K(s) + K(t/s) + O(log(K(s,t)))$$

de donde se obtiene que la distancia de información

$$E(x,y) = \text{máx}(K(x/y), K(y/x))$$

cumple con las propiedades de una distancia hasta un término de orden logarítmico[5].

Luego, notemos que la desigualdad obtenida 1.36 es exacta, así que asignaré un valor exacto (constante) a la cota ϵ. También observemos que no contamos en este momento con un conjunto exhaustivo de propiedades esperadas para el aceleramiento en P, por lo cual me limitaré a dar una condición necesaria, más no suficiente, para la *normalidad* de un sistema ATP. Esto es:

Definición 3.7. Un sistema ATP es *posiblemente normal* para L si, para casi toda t, tenemos que $\epsilon_t(x) = 0$.

Lema 3.8. *Sea A un demostrador posiblemente normal y $t \vdash O$ un argumento no trivial. Entonces*

$$\delta_A(t, t \cup \{O\}) > 0. \tag{3.9}$$

Demostración. Ya que $t \vdash O$ es un argumento no trivial, tenemos que $D(t \vdash O) > 1$. Además, notemos que $\langle O \rangle$ es una demostración para $t \cup \{O\}$, por lo cual $D(\langle O \rangle) = 1$. El lema se sigue de la definición de la función δ_A (3.5). □

Finalmente, cabe mencionar que para cada argumento elegimos la longitud de demostración mínima encontrada por PROVER9. El parámetro max_proofs fue establecido en 20, pero en todas los casos analizados PROVER9 generó un número menor de demostraciones.

[5]Para H tenemos una resultado más fuerte al poder acotar hasta una constante.

3.3. La matriz de aceleramiento

Una vez obtenidos los archivos de datos, procedemos a analizarlos para obtener la estadística buscada. La forma elegida para representar los resultados es una matriz bidimensional, llamada la matriz de aceleramiento, similar a como se hizo en [2]:

Definición 3.10. Sea E un experimento generado a partir de la lista de teorías $T = [t_1, ..., t_{x'}]$ y la lista de objetivos $O = [O_1, ..., O_o]$. La *matriz de aceleramiento* de E, denotada por $M(E)$, es la matriz construida por

- $x' \times o$ columnas, cada una de ellas representa una teoría generada, y las columnas están ordenadas a partir del índice de la teoría base en T y la longitud de la lista añadida correspondiente;
- o número de filas, donde cada columna corresponde a un objetivo de la lista O en el orden dado; y
- el valor asignado a cada entrada es

$$\delta_{i,j} = \delta(t(i), t_i)$$

donde la función δ es la delta de aceleramiento de entre $t(i)$ y t_i para O_j y la teoría $t(i)$ es igual a

$$t(i) = \begin{cases} t_i \text{ si } t_i \text{ es una teoría base} \\ t_{i-1} \text{ si } t_i \text{ no es una teoría base} \end{cases} \tag{3.11}$$

La implementación en Haskell de la matriz descrita es la siguiente:

```
data Delta = Encontro Float | Invalido | Error
```

```
matrixDeltas :: Indice -> Indice
                -> [ResultadosDep] -> Delta
matrixDeltas ts io rs | not ev = Invalido
                      | b = Error
                      | otherwise = Encontro dlt
     where ev = maybeB $ matrix ts io rs;
           dlt = findD ts io rs;
           b = delt < 0

findD it io rs | acum == - 1 = 0
               | otherwise = delta acum actual
    where f (Just a) = a;
    acum = minTD it io rs;
    actual = f (matrix it io rs)
```

donde ts e io son los índices del teorema y el objetivo correspondiente, rs es una tabla con la información recopilada de los archivos de datos para E, la función $matrix$ regresa la longitud de la demostración correspondiente a ts y io, y la función $minTD$ regresa la longitud de la demostración mínima para el objetivo io y todas las demostraciones anteriores a ts que comparten la teoría base con ts.

Es importante notar dos detalles en la implementación. Primero, por el lema 1.36, el resultado de la función $minTD$ debe ser el mismo que la longitud de la demostración correspondiente a la teoría dada por la función 3.11. La segunda observación radica en la existencia del constructor $Error$ para el tipo de dato $Delta$. Este constructor es llamado cuando $\delta(t(i), t_j) < 0$, un caso que no debe suceder, por el lema 1.36.

3.3.1. Aceleramiento trivial en $M(E)$

Recordemos que, dada la construcción de un experimento (3.1.4), esperamos una cantidad importante de casos de aceleramiento trivial. La distribución esperada de estos casos es la siguiente:

Sea t_k una teoría base. Consideremos a la submatriz $M(t_k)$ de $M(E)$ que consta del mismo número de renglones y $O+1$ columnas, los cuales representan las teorías generadas a partir de t_k. Asignando a cada entrada de la matriz el argumento correspondiente obtenemos la siguiente matriz:

$$\begin{pmatrix} t_k \vdash O_1, & t_k +\!\!+ [O_1] \vdash O_1, & ... & t_k +\!\!+ [O_1, ..., O_o] \vdash O_1 \\ t_k \vdash O_2, & t_k +\!\!+ [O_1] \vdash O_2, & ... & t_k +\!\!+ [O_1, ..., O_o] \vdash O_2 \\ ... & ... & ... & ... \\ t_k \vdash O_o, & t_k +\!\!+ [O_1] \vdash O_2, & ... & t_k +\!\!+ [O_1, ..., O_o] \vdash O_o \end{pmatrix}$$

Es decir, a cada entrada $M(t_i)_{ij}$ le corresponde el argumento $t_k : [O_1, ..., O_{i-1}] \vdash [O_j]$. Ahora, si $i > j$ tenemos que $O_j \in [O_1, ..., O_{i-1}]$, lo que implica que el argumento correspondiente a la entrada i, j es trivial y, por lo tanto, tenemos un aceleramiento trivial en las entradas de la forma $\delta_{k+i,j}$ para toda k si $i = j + 1$. Es decir, podemos esperar obtener al menos $\frac{x' \times o}{2}$ entradas de aceleramiento positivo agrupadas en estructuras diagonales con un periodo de $o + 1$. También, notemos que en las entradas que se encuentran arriba de dichas diagonales no esperamos que se dé aceleramiento, ya que la longitud de la demostración del caso anterior es la longitud de la demostración trivial.

Por último, notemos que en un experimento la condición necesaria para que, en los casos descritos en el párrafo anterior, no obtengamos un aceleramiento positivo es $O_j \in t_k \cup [O_1, ..., O_{i-1}]$ con $i \leq j$. Es claro que la probabilidad que este

caso se es de

$$\frac{0.38 \times x}{f(n,m)},$$

probabilidad que podemos considerar despreciable (dados los parámetros definidos en la sección siguiente).

3.4. Los Parámetros

Como he mencionado en capítulos anteriores, el tamaño elegido para las muestras generadas está determinado por los recursos de cómputo disponibles: dados los parámetros x y o, el número de total de argumentos generados es aproximadamente $0.38 \times x(o^2 + o)$ de acuerdo con los resultados obtenidos en 3.1.3.

Para esta primer serie de experimentos, los parámetros elegidos son:

$$n = 2,\ m = 4,\ j = 4,\ x \leq 200 \text{ y } o \leq 10.$$

Con respecto a la variable n, la complejidad sintáctica fue fijada en 2, ya que empíricamente representó un buen compromiso entre la complejidad de las fórmulas generadas y el tamaño de los subespacios generados: recordemos que la lista *en* crece de manera exponencial con respecto a n y que la generación del conjunto muestra requiere un recorrido de la lista *en*. De manera análoga, el valor elegido para el parámetro m es 4, dado que este también mostró ser un buen compromiso durante la serie de experimentos iniciales. Dados estos parámetros tenemos que

$$f(2,4) = 1115664,$$

cantidad que representa la longitud de la lista *en*.

El valor para la variable j, la complejidad descriptiva de los teoremas generados, fue elegida en consideración a los resultados del experimento presentado en la sección 3.1.3: este valor se ubica entre 3 y 5, segmento donde se encuentran los *vértices* de la distribución hiperbólica; es decir, donde el decrecimiento del número de teorías consistentes disminuye. Ahora, dado j, notemos que el número de elementos del espacio $\mathbb{T}(n, m, j)$ es

$$\begin{aligned} |\mathbb{T}(n,m,j)| &= |\{(r_1,...,r_j)|\sum_{i=1}^{j} = f(n,m) - j\}| \\ &= \binom{f(n,m)-j}{j}, \end{aligned}$$

de donde $|\mathbb{T}(2,4,4)| \approx 1{,}2518\times10^{22}$. Esta cantidad de elementos, aunque es varios órdenes menor a la encontrada para $|\mathbb{T}(2,2)|$, sigue siendo computacionalmente intratable.

El valor de o es pequeño comparado con x (el número total de teorías generadas) dadas las siguientes consideraciones: Dado el diseño del experimento, los objetivos de la lista O deben de ser satisfactibles al ser emparejados mediante una conjunción[6]; además, al concatenar las listas añadidas con las teorías correspondientes debemos obtener sistemas consistentes, por lo que el generar un número grande de objetivos puede disminuir de manera importante el rendimiento en producción de argumentos demostrables (con respecto al número total de argumentos generados). Por último, el número de argumentos generados es proporcional en un orden lineal al número de teorías x, pero crece de manera cuadrática con respecto a o.

[6]De lo contrario, al menos uno de los argumentos no puede ser demostrable por la misma teoría.

Dados los parámetros definidos, estamos generando muestras con una longitud esperada de

$$0.38 \times (200) \times (10^2 + 10) = 8140 \text{ elementos}, \tag{3.12}$$

de las cuales alrededor de un 89 % son demostrables. Ahora, la longitud de las muestras tiende a ser menor, ya que la lista O es limpiada de elementos inconsistentes (con respecto al primer elemento). Aún así, una computadora personal con un procesador i5 a 2Ghz con 8GB de memoria RAM tardó más de un día en procesar cada experimento.

3.5. Matrices Encontradas

3.5.1. Prover9

En total se realizaron más de 15 experimentos para Prover9, todos ellos presentaron un mismo comportamiento, el cual considero antagónico para los objetivos planteados en este trabajo.

Entre los casos de comportamiento no deseado se encontró un número importante de casos de aceleramiento negativo. Esta conducta presenta una negativa a la condición expuesta en 3.7. Ahora, es posible que la condición dada sea *demasiada* estricta, es decir, que es posible que una condición más débil de *normalidad* represente un buen compromiso con respecto a la relación entre la delta de aceleramiento y la delta de aceleramiento relativo.

Sin embargo, considero que el siguiente resultado refuerza la necesidad de la condición dada: *las matrices de aceleramiento encontradas no muestran la regularidad esperada*. Recordemos que, por diseño (3.3.1), en cada matriz se espera un grado importante de regularidad en la forma de estructuras diagonales

de longitud y periodo constante. En su lugar, encontramos una distribución irregular de *líneas* verticales. Considero que, dadas su forma y distribución, las estructuras encontradas no pueden ser consideradas una aproximación adecuada a las estructuras esperadas y, por consiguiente, no nos encontramos en presencia de un sistema ATP que pueda ser considerado *normal*. Es decir, δ_{PROVER9} (3.5) no es una aproximación adecuada a δ (1.35).

El comportamiento tampoco es consistente al comparar las matrices correspondientes a cada experimento: el porcentaje del número de aceleramiento positivo encontrados en cada experimento difiere hasta en un 68 %; y la razón entre los dos tipos de aceleramiento difiere aún más (3.5.1).

Finalmente, analizando los archivos de datos generados en busca de una explicación para los resultados obtenidos, encontré casos de aceleramiento con comportamiento inverso al esperado:

El siguiente argumento genera una demostración de 16 pasos.

$$\{A \wedge C \iff -(-A \iff D)\} \vdash A \wedge C \iff -(-A \iff D).$$

Mientras que el siguiente toma únicamente 6:

$$\begin{aligned} \{A \wedge C \iff -(-A \iff D), \\ (-C \implies A) \wedge -(-D \implies -B), \\ D \vee -B \implies A, -A \vee B \implies -(D \implies -C), \\ (-C \implies B) \iff -A \vee -A, B \wedge -C \vee -(A \wedge -A), \\ (D \implies D) \vee (-A \implies -A)\} \vdash A \wedge C \iff -(-A \iff D). \end{aligned}$$

Figura 3.3: Una representación en escala de grises de la matriz de aceleramiento obtenida durante el experimento número 11 usando Prover9. La matriz se encuentra partida en cuatro secciones verticales. En esta imagen únicamente se encuentran resaltadas los casos de aceleramiento positivo, cuanto más oscura es una celda mayor es la delta de aceleramiento.

Exp. Núm.	Casos	$\delta > 0$	Porcentaje	$\delta < 0$	Razón
11	5400	606	11.2 %	94	6.44
10	6381	704	11.01 %	137	5.138
7	4848	389	8.02 %	231	1.683
5	5454	426	7.81 %	24	17.75
3	11297	856	7.57 %	70	12.228

Figura 3.4: Esta tabla muestra la irregularidad presente en los resultados al comparar varias matrices correspondientes a un número de experimentos seleccionados. El porcentaje del número de aceleramiento positivo encontrado en cada experimento difiere hasta en un 68 %.

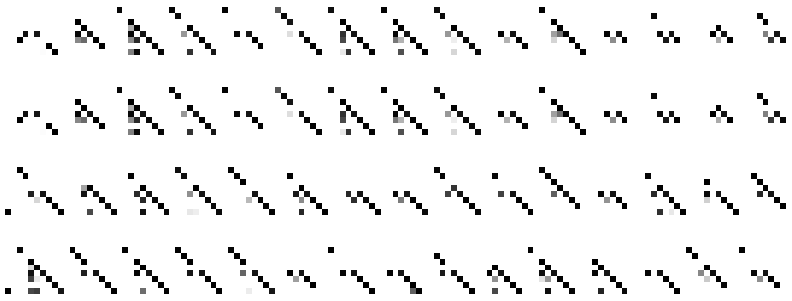

Figura 3.5: Una representación en escala de grises de la matriz de aceleramiento obtenida durante el experimento número 14 usando APROS. La matriz se encuentra partida en cuatro secciones verticales. En esta imagen únicamente se encuentran resaltadas los casos de aceleramiento positivo, cuanto más oscura es una celda mayor es la delta de aceleramiento. En esta imagen es clara la existencia de una distribución regular de las estructuras diagonales periódicas descritas en 3.3.1 y la escasez de aceleramiento no trivial positivo.

3.5.2. AProS

APROS presenta un comportamiento distinto a PROVER9, el cual también considero no normal. También es importante reportar que, debido a las limitadas opciones de interacción que ofrece APROS, se realizó un número limitado de experimentos usando este demostrador.

Al contrario de PROVER9, el sistema de inferencia usado por APROS es capaz de detectar argumentos triviales con regularidad: en la visualización de la matriz de aceleramiento para el experimento 13(3.5.1), es fácil constatar la existencia de una distribución regular de las estructuras diagonales y triangulares periódicas descritas en 3.3.1. Aún más, los pocos casos en los cuales encontramos alteraciones en el patrón esperado son causadas, en su mayoría, por argumentos no demostrables.

Esta última aserción es evidente al analizar una segunda matriz, llamada *matriz de casos*. Esta matriz de casos asigna uno de cuatro valores discretos a cada entrada de acuerdo los casos que podemos encontrar:

- La delta presenta un aceleramiento nulo,
- un aceleramiento positivo,
- un aceleramiento negativo o
- la entrada corresponde a un argumento no demostrable.

Sin embargo, esta última visualización pone en evidencia el comportamiento anormal principal encontrado en APROS: la presencia de aceleramiento negativo en una proporción similar a los casos de aceleramiento positivo: en el experimento 13 encontramos 555 casos de aceleramiento negativo por 463 de aceleramiento

Figura 3.6: Una representación en escala de grises de la matriz de casos obtenida durante el experimento número 14 usando APROS. El color blanco corresponde a los casos de aceleramiento nulo, el color negro a los casos no demostrables, el gris claro a los de aceleramiento positivo y el gris oscuro a los de aceleramiento negativo.

positivo. Esta distribución, además de ser varias veces mayor a la encontrada en los experimentos realizados con PROVER9, es en especial problemática ya que la mayoría de los casos de aceleramiento positivo son casos triviales y solamente un número menor a 100 son casos de aceleramiento positivo no trivial. Es decir, al incluir nuevos axiomas estamos alentando las demostraciones en un número significativo de instancias, con una esperanza menor de encontrar aceleramiento positivo.

Por lo anterior, considero que tampoco podemos considerar a APROS como un sistema APROS normal.

Capítulo 4

Conclusiones y trabajo a futuro

El objetivo de este trabajo fue explorar la distribución de casos de aceleramiento positivo en el cálculo de proposiciones. En particular, se exploraron de manera empírica dos sistemas de demostración distintos: deducción natural y resolución binaria, y se propuso aproximarlos mediante dos sistemas de demostración automatizados: AproS y Prover9. Además se definió una condición necesaria para considerar adecuadas estas aproximaciones.

Dados los resultados expuestos, es evidente que ninguno de los sistemas de demostración automatizada explorados cumplen con la condición de normalidad (3.7) para el subespacio definido: el número de casos *anormales* encontradas no puede ser considerado despreciable y se presentaron de manera sistemática durante la exploración del espacio.

Asimismo, considero que los resultados obtenidos refuerzan la necesidad de la condición de normalidad definida, la cual muestra ser significativamente más fuerte que la expuesta por el lema 3.8;

ya que mientras APROS detecta los casos triviales, el comportamiento observado tanto en PROVER9 como en APROS sugiere que esta propiedad de *normalidad* no se extiende más allá de un análisis sintáctico inicial. Es importante notar que, al desarrollar un sistema ATP, no se espera un número significativo de casos triviales, razón por la cuál los sistemas ATP no tienen por qué implementar una búsqueda del objetivo en la lista de premisas; en otras palabras, esperamos que el cumplimiento del lema 3.8 debe ser consecuencia del algoritmo de inferencia implementado.

Aun más, la propiedad descrita en el párrafo anterior sigue siendo un atributo más débil que el expuesto por la condición de normalidad definida, lo que me lleva a formular las dos conjeturas presentadas a continuación:

La primera conjetura presume que para APROS, PROVER9, y otros sistemas ATP no normales optimizados con respecto al tiempo de ejecución, tenemos que la cota ϵ_t es una función de orden exponencial.

Conjetura 4.1. *Para* APROS *y* PROVER9 *tenemos que la función ϵ_t, definida en 3.6, es de orden exponencial para casi toda t. Es decir $\epsilon_t \in O(c^n)$.*

La segunda conjetura es una presunción más fuerte: desarrollar un demostrador posiblemente normal para casi todas sus entradas es un problema NP-completo. En otras palabras:

Conjetura 4.2. *Existe un demostrador posiblemente normal que demuestra casi todos los argumentos en tiempo polinomial si y sólo si $P = NP$.*

Es importante notar que la segunda conjetura presupone un resultado más fuerte que el presentado por el teorema 1.34: esta

conjetura habla no sólo de la dificultad de aproximarse a la demostración mínima, sino también de la dificultad de mantener una regularidad entre las longitudes de las demostraciones encontradas, aún si éstas no son mínimas.

Dar una demostración a las proposiciones anteriores queda fuera de los alcances de este trabajo, por lo que quedan como preguntas abiertas para trabajos posteriores.

Ahora que establecí que el aceleramiento relativo definido por los sistemas de demostración automatizado usados no representó una buena aproximación para el aceleramiento en el cálculo de proposiciones, nos preguntamos sobre la distribución del aceleramiento relativo a cada demostrador: podemos pensar en cada sistema ATP como en un sistema de demostración en sí, donde las reglas de derivación están dadas por el algoritmo de demostración automatizada implementado.

Ahora, notemos que la tabla 3.5.1 nos muestra una falta de regularidad en la distribución de los casos de aceleramiento (positivo o negativo) encontrados en cada experimento. Puede que esta irregularidad en la distribución se deba a que los conjuntos muestra no son representativos del espacio. En particular, los conjuntos muestra son más sensibles a la lista de objetivos, la cual es de un orden menor en comparación a la lista de teoremas generados.

Sin embargo, en la mayoría de los experimentos realizados encontramos una presencia importante de aceleramiento negativo, por lo que podemos concluir que *el aceleramiento relativo negativo no es un fenómeno escaso y se da en una proporción mayor al aceleramiento relativo positivo.* Considero que esta última conducta es consecuencia directa del comportamiento no normal

encontrado en los sistemas explorados, con lo que podemos concluir que el estudio realizado refuerza la condición de normalidad definida.

Bibliografía

[1] J. Joosten, F. Soler-Toscano and H. Zenil, Program-size Versus Time Complexity, Speed-up and Slowdown Phenomena in Small Turing Machines, Int. Journ. of Unconventional Computing, special issue on Physics and Computation, vol. 7, no. 5, pp. 353-87, 2011.

[2] H. Zenil, From Computer Runtimes to the Length of Proofs: With an Algorithmic Probabilistic Application to Waiting Times in Automatic Theorem Proving. In M.J. Dinneen, B. Khousainov, and A. Nies (Eds.), Computation, Physics and Beyond, Theoretical Computer Science and Applications, WTCS 2012, LNCS 7160, pp. 223-240, Springer, 2012. arXiv:1201.0825 [cs.CC].

[3] J. Joosten, F. Soler-Toscano, H. Zenil, Speedup and Slowdown Phenomena in Turing Machines Wolfram Demonstrations Project Published: November 8, 2012

[4] A.M. Ben-Amram, The Church-Turing Thesis and its Look-Alikes. SIGACT News 36 (3): 113-116, 2005.

[5] Turing, A. M. On Computable Numbers, with an Application to the Entscheidungsproblem, Proceedings of the London Mathematical Society, 2 42: 230-65, 1937.

[6] G. J. Chaitin, Algorithmic Information Theory, Cambridge Tracts in Theoretical Computer Science Volume 1, Cambridge University Press, 1987.

[7] G. J. Chaitin, On the length of programs for computing finite binary sequences, Journal of the ACM, 13(4):547-569, 1966.

[8] M. Hutter, Algorithmic complexity. Scholarpedia, http://www.scholarpedia.org/article/Algorithmic_complexity, 3(1):2573, 2009

[9] K. Gödel, On formally undecidable propositions of Principia Mathematica and related systems I in Solomon Feferman, Collected works, Vol. I. Oxford University Press: 144-195, 1986.

[10] A. N. Kolmogorov, Three approaches to the quantitative definition of information. Problems of Information and Transmission, 1(1):17, 1965.

[11] R. J. Solomonoff, A formal theory of inductive inference: Parts 1 and 2. Information and Control, 7:1-22 and 224-254, 1964.

[12] S. Kleene, Introduction to Metamathematics, North-Holland Publishing Company, New York, 1952. In Chapter XI. General Recursive Functions

[13] M. Alekhnovich, S. Buss, S. Moran, y T. Pitassi, Minimum Propositional Proof Length is NPHard to Linearly Approximate, The Journal of Symbolic Logic, Volume 66: 171−191, 2001.

[14] K. Kunen, Set Theory: An Introduction to Independence Proofs, Elsevier: 133−138, 1980.

[15] M. Heine, S. and P. Urzyczyn, Lectures on the Curry-Howard Isomorphism, Studies in Logic and the Foundations of Mathematics 149, Elsevier Science, ISBN 978-0-444-52077-7, 1998.

[16] M. Huth, and M. Ryan, Logic in Computer Science: Modelling and reasoning about Systems, Cambridge University Press, UK, 2004.

[17] S. Kleene, E. L. Post, "The upper semi-lattice of degrees of recursive unsolvability", Annals of Mathematics. Second Series 59, (1954).

[18] J.E. Hopcroft, R. Motwani, and J.D. Ullman, Introduction to Automata Theory, Languages, and Computation, Addison Wesley, Boston/San Francisco/New York: 368, 2007.

[19] Encyclopedia Britannica. Hombre, ciencia y tecnologia, Volumen 6, Britannica, pagina: 2142, 1992.

[20] R. I. Levin, Estadística para administradores, Prentice Hall, 187-191, 1996.

[21] W. Feller, *Teoría de probabilidades y sus aplicaciones*, Limusa, 1985.

[22] B. Christoph, S. Geoff, Working with Automated Reasoning Tools, http://www.cs.miami.edu/∼geoff/Courses/TPTPSYS/, 2008.

[23] R. Cilibrasi, P.M.B. Vitanyi, Clustering by compression, IEEE Trans. Inform. Theory, 51:12(2005), 1523-1545.

[24] W. McCune, "Prover9 and Mace4", http://www.cs.unm.edu/∼mccune/Prover9, 2005-2010.

[25] W. Sieg, "AProS", http://www.phil.cmu.edu/projects/apros/ , 2006−2012.

[26] W. Sieg, The AProS Project: Strategic Thinking & Computational Logic, Logic Journal of the IGPL, 15(4): pp. 359-368, 2007.

[27] W. Sieg and J. Byrnes, Normal natural deduction proofs (in classical logic), Studia Logica 60, pp. 67-106, 1998.

[28] M. R. Garey and D. S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, W.H. Freeman, 1979.