Universitat Politècnica de Catalunya
Facultat d'Informàtica de Barcelona
Enginyeria Informàtica

Proyecto Final de Carrera

Diseño e implementación de algoritmos de unificación y matching para términos comprimidos

Lander Ramos Garrido
23 de junio de 2011

Director: Guillem Godoy Balil

Llenguatges i Sistemes Informàtics

Resumen

Las gramáticas incontextuales se pueden utilizar como herramienta para comprimir palabras o términos. Estas gramáticas pueden permitir en casos extremos comprimir hasta un factor exponencial. Sobre este tipo de compresiones existen diferentes algoritmos que se pueden implementar sin necesidad de descomprimir la palabra o término dados.

En este PFC diseñamos nuevos algoritmos probabilísticos que se puedan aplicar a este tipo de compresiones. Estos algoritmos mejoran el coste de los algoritmos deterministas ya existentes, y su eficacia con respecto a los algoritmos equivalentes sobre palabras y términos descomprimidos dependen sobre todo del factor de compresión.

Además de diseñar dichos algoritmos, también incluimos una implementación de todos ellos, que nos ha permitido comprobar que efectivamente mejoran el tiempo de ejecución de los algoritmos ya existentes

Índice general

Capítulo 1
Introducción

Los árboles son una estructura de datos muy habitual en las ciencias de la computación. A pesar de ser una herramienta muy básica, tiene numerosas aplicaciones, entre las que se incluyen las estructuras de datos como árboles de búsqueda, que permiten acceder a los datos de forma mucho más eficiente.

Algunas aplicaciones recientes necesitan trabajar con árboles muy grandes. Por ejemplo, uno de los sistemas más habituales de intercambio de datos es el formato XML, que guarda la información en forma de árbol sin rango; típicamente, cada conjunto de datos viene acompañado de varios árboles XML que describen su estructura. Esto da lugar a árboles realmente grandes, que pueden llegar a tener varios millones de nodos.

En ocasionas dichos árboles ni siquiera pueden caber enteros en memoria, lo que dificulta aún más un acceso eficiente. Por este motivo se han desarrollado diversas representaciones de árboles comprimidos en memoria. Como ejemplos encontramos los *succint trees* [**SN10**], o los árboles comprimidos con gramáticas [**BLM08, MS10**]

En este PFC trabajaremos con árboles comprimidos con gramáticas. La compresión mediante gramáticas es una variante de la compresión para palabras creada en los años noventa [**Ryt04**]. La idea básica es encontrar una gramática más pequeña que genere únicamente la palabra dada. Es una forma de compresión por diccionario donde los símbolos no terminales de la gramática representan subpalabras repetidas. En particular, una *gramática incontextual* mínima que genere una palabra dada puede ser (como mucho) exponencialmente más pequeña que dicha palabra.

Sabemos que encontrar la gramática mínima que comprime una palabra es NP-completo, pero se han encontrado algunos algoritmos que obtienen aproximaciones bastante razonables [**CLL$^+$05**]. En general, los algoritmos ejecutados sobre representaciones comprimidas son más lentos que ejecutados sobre las palabras originales, ya que normalmente necesitan descomprimir antes. Sin embargo, existen ciertos algoritmos que se pueden ejecutar sobre la gramática sin necesidad de descomprimirla. Algunos de ellos además tienen un tiempo de ejecución proporcional a la compresión, lo que eventualmente podría traducirse a una mejora

exponencial con respecto a los algoritmos que se ejecutan sobre términos descomprimidos. Por ejemplo, comprobar si dos gramáticas incontextuales generan la misma palabra se puede hacer en tiempo cúbico con respecto al tamaño de las gramáticas [**Lif07**].

La idea de compresión por gramáticas se generalizó de palabras a árboles en [**BLM08**], donde presentaron un algoritmo aproximativo que encuentra una gramática incontextual de árboles de tamaño reducido. Llamaremos *Singleton Tree Grammar* (STG) a una gramática incontextual de árboles que genera un solo árbol. Nótese que la idea clásica de representar un árbol por su Grafo Acíclico Dirigido (DAG) mínimo es un caso concreto de compresión de gramáticas: dicho DAG es equivalente la mínima gramática regular que representa ese árbol.

Para documentos XML típicos, los DAG permiten reducir su estructura de árbol hasta un 12 % del número original de nodos [**BGK03**]. El algoritmo de [**BLM08**] encuentra STG que contienen sólo un 4 % del número original de nodos. El nuevo compresor de gramáticas "TreeRePair" [**LMM10**] es capaz de conseguir una compresión aún mayor ($< 3\,\%$) y se ejecuta casi tan rápido como el algoritmo que encuentra el mínimo DAG.

Otros ejemplos de algoritmos que se ejecutan eficientemente (sin necesidad de descomprimir) sobre STG son evaluación de *tree automata* [**LM06**], procesado de consultas en XPath [**MS10**] y tests de equivalencia [**BLM08, SSS10**]. Las STG también se han usado para análisis de complejidad o algoritmos de unificación [**LSSV08**]. Recientemente se ha demostrado que la unificación de primer orden es un problema que se puede resolver en tiempo polinómico sobre STG [**GGSS08, GGSS09**].

Uno de los objetivos de este PFC es mostrar una implementación de los algoritmos de unificación y matching presentados en [**GGSS10**]. Dichos algoritmos ejecutan una variante del algoritmo estándar de unificación de Robinson [**Rob65**] sobre dos STG dadas; en primer lugar construye la gramática de palabras para el recorrido en preorden de las STG, y después hace un test de equivalencia para gramáticas de palabras. Para el test de equivalencia implementamos dos algoritmos diferentes, aunque uno de ellos con una variante. Dichos algoritmos son:

- El algoritmo exacto de Lifshits [**Lif07**].
- Un algoritmo probabilístico reciente de Schmidt-Schauß y Schnitger [**SSS10**]. Dentro de ese algoritmo usamos dos variantes:
  - Una variante que utiliza números primos como módulos a la hora de realizar los cálculos.
  - Otra variante que utiliza todo tipo de números como módulos.

Nuestra herramienta se integra con el compresor TreeRePair: recibe como entrada dos términos representados en XML y ejecuta TreeRePair para construir las STG correspondientes. A continuación ejecuta el algoritmo de unificación sobre STG.

Finalmente hemos realizado diversos experimentos que nos permiten evaluar la eficacia de los algoritmos realizados sobre STG con respecto a los algoritmos realizados sobre árboles. El resultado de estos experimentos se puede ver en el Capítulo 8. Básicamente se deduce que la unificación sobre STG es más eficiente si los términos que queremos unificar son altamente compresibles y suficientemente grandes.

Todo el código fuente de nuestro trabajo se puede encontrar en la página web `www.lsi.upc.edu/~agascon/unif-stg`, donde además hay un interfaz que permite ejecutar los algoritmos directamente en la web dados dos términos descomprimidos en el formato del compresor (XML).

Este PFC se basa en un trabajo anterior hecho por Adrià Gascón, que había implementado una primera versión de la unificación sobre términos comprimidos, usando un algoritmo determinista. Mi aportación a este proyecto queda reflejada en el Capítulo 7, pero se puede resumir en lo siguiente:

- Deducción de una variante del algoritmo explicado en [**SSS09**] para poder usarla como parte de la unificación.
- Estudio y comparación de ventajas e inconvenientes respecto al uso de números primos.
- Implementación de las nuevas funcionalidades del código:
  - Implementación de la clase term, que permite trabajar con términos no comprimidos y generar casos de prueba arbitrariamente grandes.
  - Implementación de la variante de los dos algoritmos probabilísticos que utilizamos en este PFC.
  - Implementación de parte de las modificaciones necesarias en el código para poder trabajar con el compresor.
  - Implementación del interfaz que permite realizar las pruebas y comparaciones oportunas sobre los diferentes algoritmos.
- Depuración de diversos bugs que aparecían ejecutando el código ya existente en casos límite.
- Diseño de varios casos de prueba que permiten probar la correctitud y la eficiencia del nuevo código.

Finalmente, añadir que a partir de este proyecto hemos enviado un artículo al congreso *Rewriting Techniques and Applications* (RTA) 2011, conjuntamente con Adrià Gascón y Sebastian Maneth [**GMR11**].

Capítulo 2
Preliminares

2.1. Términos

Un *alfabeto con rango* es un conjunto finito $\mathcal{F}$, junto con una función $\mathtt{ar} : \mathcal{F} \to \mathbb{N}$. Los miembros de $\mathcal{F}$ se llaman símbolos de función, y llamaremos a $\mathtt{ar}(f)$ la *aridad* del símbolo de función f. Llamaremos constantes a los símbolos de función de aridad 0. Sea $\mathcal{X}$ un conjunto disjunto con $\mathcal{F}$, cuyos elementos tienen aridad 0. Llamaremos variables de primer orden a los elementos de $\mathcal{X}$. El conjunto $\mathcal{T}(\mathcal{F} \cup \mathcal{X})$ de términos sobre $\mathcal{F}$ y $\mathcal{X}$, que escribiremos también $\mathcal{T}(\mathcal{F}, \mathcal{X})$, se define como el menor conjunto con la propiedad de que si $f \in (\mathcal{F} \cup \mathcal{X})$ es un símbolo de aridad m, y $t_1, \ldots, t_m \in \mathcal{T}(\mathcal{F} \cup \mathcal{X})$, entonces $f(t_1, \ldots, t_m) \in \mathcal{T}(\mathcal{F} \cup \mathcal{X})$. En particular todas las constantes y todas las variables pertenecen a $\mathcal{T}(\mathcal{F} \cup \mathcal{X})$, al ser símbolos de aridad 0. Un término *ground* es aquel que no tiene variables.

El *tamaño* $|t|$ de un término t es el número de ocurrencias de símbolos de variables y símbolos de función en t. Más formalmente se define como $|f(t_1, \ldots, t_m)| = 1 + |t_1| + \ldots + |t_m|$. La *altura* de un término t, que escribiremos $\mathrm{height}(t)$, se define como 0 si t es constante o variable, y $1 + \max\{\mathrm{height}(t_1), \ldots, \mathrm{height}(t_m)\}$ si $t = \alpha(t_1, \ldots, t_m)$.

El conjunto de *posiciones* de un término t, $\mathrm{Pos}(t)$, se define como $\mathrm{Pos}(t) = \{\lambda\}$ si t consiste en un solo símbolo, y $\mathrm{Pos}(t) = \{\lambda\} \cup \{1.p \mid p \in \mathrm{Pos}(t_1)\} \cup \ldots \cup \{m.p \mid p \in \mathrm{Pos}(t_m)\}$ si $t = f(t_1, \ldots, t_m)$. Aquí λ representa la secuencia vacía, y $p.q$ representa la concatenación de las secuencias p y q. Eventualmente describiremos la concatenación de p y q simplemente como pq, para abreviar.

Sea t un término y p una posición de $\mathrm{Pos}(t)$. Denotaremos como $t|_p$ el subtérmino de t en posición p. Más formalmente, $t|_\lambda = t$, y $f(t_1, \ldots, t_m)|_{i.p} = t_i|_p$.

Denotaremos por $\mathrm{Pre}(t)$ el recorrido en preorden de un término. Este recorrido será la palabra definida como $\mathrm{Pre}(t) = f.\mathrm{Pre}(t_1).\ldots.\mathrm{Pre}(t_m)$, donde $t = f(t_1, \ldots, t_m)$.

Sea k una posición de $\mathrm{Pre}(t)$. Definiremos $\mathrm{iPos}(t, k)$ como la posición de $\mathrm{Pos}(t)$ correspondiente a k en el recorrido el preorden, es decir, si $t = f(t_1, \ldots, t_n)$

entonces, si $k = 1$, $\text{iPos}(t, k) = \lambda$, y si por el contrario $k = 1 + m + l$, donde $m = \sum_{i=1}^{j} |t_i|$ y $l \leq |t_{j+1}|$, entonces $\text{iPos}(t, k) = \text{iPos}(t_{j+1}, l)$.

Sea t el término $f(t_1, \ldots, t_m)$. El símbolo f será la *raíz* de t, $\text{root}(t)$. Un término también lo podemos ver como una aplicación parcial que hace corresponder a cada posición un símbolo. Así podemos definir el símbolo en posición p de t como $t(p) = \text{root}(t|_p)$.

Ejemplo 2.1. *Salvo que se indique lo contrario, los ejemplos utilizarán el alfabeto* $\mathcal{F} = \{f : 2, g : 1, a : 0, b : 0\}$ *y* $\mathcal{X} = \{x, y\}$.

En este caso ejemplos de términos en $\mathcal{T}(\mathcal{F}, \mathcal{X})$ *son:* a, b, $f(a, b)$, $g(a)$, $f(f(a, b), b)$, x, $f(x, y)$, $g(x)$, $f(a, g(y)) \ldots$

Pero no serían términos de $\mathcal{T}(\mathcal{F}, \mathcal{X})$ *árboles como:* $a(b)$, $x(y)$, f, $g(x, a)$, $f(a, b, x)$ *por no respetar la aridad, o* c, $h(a, b)$, $g(z)$, *por incluir símbolos que no son de los alfabetos.*

Intuitivamente, un término se puede ver como un árbol general, donde el número de hijos de cada nodo es igual a la aridad del símbolo de función correspondiente.

2.2. Sustituciones

Una sustitución es una aplicación $\sigma : X \to \mathcal{T}(\mathcal{F}, \mathcal{X})$, es decir, que hace corresponder a cada variable un término. También se puede aplicar recursivamente sobre un término de $\mathcal{T}(\mathcal{F}, \mathcal{X})$ de la siguiente forma: $\sigma(a) = a$ si $a \in \Sigma$ es una constante, y $\sigma(f(t_1, \ldots, t_n)) = f(\sigma(t_1), \ldots, \sigma(t_n))$.

Si la sustitución hace corresponder cada variable en un término ground, entonces diremos que la sustitución es ground. Fijémonos en que una sustitución ground aplicada a un término siempre da como resultado un término ground.

Sean σ y τ dos sustituciones. Definimos la concatenación $\sigma \circ \tau$ como la sustitución tal que si $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, entonces $\sigma \circ \tau(t) = \sigma(\tau(t))$.

En algunos casos nos puede ser útil definir una sustitución como un conjunto de reglas de la forma $x \mapsto t$, con $x \in \mathcal{X}$ y $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, donde no puede haber dos reglas diferentes que den valor a la misma variable. Esta notación nos puede servir para definir parcialmente una sustitución que vamos a ir construyendo (por ejemplo, durante un algoritmo). En este caso, si aplicamos una sustitución σ a una variable x, pero no hay ninguna regla de la forma $x \mapsto t$ en σ, podemos suponer que no modificamos x, es decir, que $\sigma(x) = x$.

Ejemplo 2.2. *Usaremos el mismo alfabeto que en el Ejemplo 2.1. Sea* $t_1 = f(x, f(x, y))$, *y* $t_2 = g(y)$. *Sea* $\sigma = \{x \to g(a), y \to f(a, g(b))\}$. *Entonces tenemos que* $\sigma(t_1) = f(g(a), f(g(a), f(a, g(b))))$ *y* $\sigma(t_2) = g(f(a, g(b))$.

Nótese que una sustitución puede hacer corresponder una variable a un término en el que aparece otra variable. En el nuevo subtérmino no debemos aplicar recursivamente la sustitución, como se aprecia en el Ejemplo 2.3.

Ejemplo 2.3. *De nuevo usaremos el alfabeto del Ejemplo 2.1. Definiremos la siguiente sustitución* σ*:* $\sigma = \{x \mapsto f(a,a), y \mapsto f(x,x)\}$*. Esta sustitución no equivale a* $\bar{\sigma} = \{x \mapsto f(a,a), y \mapsto f(f(a,a),f(a,a))\}$*, ya que la aplicación de reglas no es recursiva. De hecho* $\bar{\sigma} = \sigma \circ \sigma$*. Así, si aplicamos dichas sustituciones al término* $t = f(x,y)$*, obtenemos que* $\sigma(t) = f(f(a,a),f(x,x))$*, mientras que* $\bar{\sigma}(t) = f(f(a,a),f(f(a,a),f(a,a)))$*.*

Capítulo 3
Problema de unificación

En este proyecto se trabajará a fondo con el problema de unificación de términos. El problema de unificación es un problema clásico de la lógica y las ciencias de la computación, y consiste en decidir si dos términos sobre una misma pareja de alfabetos con rango, $\mathcal{F}$ y $\mathcal{X}$, unifican, es decir, si existe una sustitución que los hace iguales. En algunos casos nos puede ser también útil saber cual es esa sustitución.

3.1. Definición

En este proyecto trabajaremos con unificación de primer orden, que pasamos a definir.

Dados dos términos, t_1, t_2, pertenecientes a $\mathcal{T}(\mathcal{F}, \mathcal{X})$, el problema de unificación $t_1 \doteq t_2$, consiste en decidir si existe una sustitución σ tal que $\sigma(t_1) = \sigma(t_2)$. Dicha sustitución, si existe, se denomina unificador, y al resultado de aplicar el unificador a cada término lo llamaremos término unificado.

Ejemplo 3.1. *Sea $t_1 = f(x, g(a))$, y $t_2 = f(g(a), x)$. Entonces el problema $t_1 \doteq t_2$ tiene solución, ya que la sustitución $\sigma = \{x \to g(a)\}$ unifica ambos términos, pues $\sigma(t_1) = f(g(a), g(a)) = \sigma(t_2)$.*

Sin embargo, si consideramos $t_1 = f(x, x)$, y $t_2 = f(a, b)$, entonces el problema $t_1 \doteq t_2$ no unifica, ya que no podemos asignar a x los términos a y b simultáneamente.

3.2. Resultados

Se sabe que si dos términos tienen algún unificador, entonces existe un único unificador más general, esto es, un unificador σ tal que para cualquier otro unificador τ existe un $\bar{\tau}$ tal que $\tau = \bar{\tau} \circ \sigma$.

El problema de encontrar el unificador más general se sabe que está en la clase de complejidad P. Hay diversos algoritmos que resuelven el problema. El más simple

```
Unifica(s : término, t : término): booleano
  σ: sustitución
  σ := ∅
  Mientras (σ(s) ≠ σ(t)):
    Sea k la primera posición tal que Pre(σ(s))[k] ≠ Pre(σ(t))[k].
    Si tanto Pre(σ(s))[k] como Pre(σ(t))[k] son símbolos de función, Entonces
      Devuelve falso
    // Supondremos sin pérdida de generalidad que Pre(σ(s))[k] es un símbolo de variable x.
    Si x aparece en alguna posición de σ(t)|_p, donde p = iPos(σ(t), k), Entonces
      Devuelve falso
    σ := {x ↦ σ(t)|_p} ∘ σ
  Devuelve cierto
```

FIGURA 3.1. Esquema general del algoritmo de Robinson para resolver unificación de primer orden

de ellos es el algoritmo de Robinson, aunque existen casos artificialmente creados en los que se puede llegar a alcanzar tiempo exponencial. A continuación damos una visión general del algoritmo.

En líneas generales, lo que hace el algoritmo es buscar la primera posición (en recorrido en preorden), en la que ambos términos presenten una diferencia. Dado que ambos términos están construidos sobre los mismos alfabetos, la primera posición en la que difieren tiene existir en los dos términos, porque si en uno de ellos no existiera querría decir que algún antecesor tendría aridad diferente. Dicho antecesor iría antes en el recorrido en preorden y lo habríamos detectado antes.

Supongamos que en la primera posición en la que difieren ambos términos hay un símbolo de función (ya sea constante o no). En ese caso es evidente que no habrá unificación, ya que una sustitución sólo modifica variables, y por tanto ninguna sustitución hará que los dos términos sean iguales.

Si en cambio hay un símbolo de variable en ambos términos, digamos x e y, entonces es evidente que cualquier unificador válido sustituirá x e y por el mismo término. Así que podríamos simplemente sustituir todas las ocurrencias de y en el problema por x, y seguir resolviendo.

Por último, si hay un símbolo de variable en un término, y uno de función en el otro, se plantean dos opciones. Sea p la posición en la que difieren, y $t|_p$ el subtérmino en posición p del término donde hay un símbolo de función. Sea x la variable del otro término. Es evidente que si x tiene alguna aparición en $t|_p$ no podremos unificar. Si en cambio x no tiene ocurrencias en $t|_p$, será necesario asignar a la variable x el término $t|_p$.

En la Figura 3.1 se expone un esquema del algoritmo utilizado.

El coste del algoritmo dependerá de la implementación utilizada. La implementación más básica consiste en representar los términos como árboles. En un lenguaje como C++ un árbol se puede guardar en memoria como una estructura consistente en un valor (en este caso el símbolo raíz), y un vector de punteros a árboles,

de tamaño igual a la aridad de la raíz. Si cada vez que componemos una regla $x \mapsto t$ con la sustitución reemplazamos todas las ocurrencias de la variable x por una copia del término t, el tamaño final del árbol será el tamaño del término unificado, y el coste del algoritmo no podrá ser inferior a este.

Esto puede causar problemas, ya que hay situaciones en las que el tamaño del término unificado puede ser exponencialmente más grande que los árboles a unificar, como se aprecia en el siguiente ejemplo.

Ejemplo 3.2. *Consideraremos los alfabetos siguientes:* $\mathcal{F} = \{f : 2, a : 0\}$ *y* $\mathcal{X}_n = \{x_1, x_2, \ldots, x_n\}$. *Definimos la familia de términos* $t_0 = s_0 = f(a, a)$, *y si* n *es mayor que* 0, *entonces* $t_n = f(x_n, t_{n-1})$, $s_n = f(s_{n-1}, x_n)$. *Nótese que, fijado un* n, *tanto* t_n *como* s_n *son términos de* $\mathcal{F}, \mathcal{X}_n$, *es decir,* $t_n, s_n \in \mathcal{T}(\mathcal{F}, \mathcal{X}_n)$. *Es fácil de comprobar que, dado un* $n \geq 0$, *el tamaño de* s_n *y* t_n *es lineal con respecto a* n. *Más concretamente* $|s_n| = |f_n| = 2n + 3$.

Fijado un $n \geq 0$, *consideremos el problema de unificación* $s_n \doteq t_n$ *Si calculamos el unificador más general con el algoritmo de Robinson obtendremos:* $\sigma_n = \{x_1 \mapsto f(a, a)\} \circ \{x_2 \mapsto f(x_1, x_1)\} \circ \ldots \circ \{x_n \mapsto f(x_{n-1}, x_{n-1})\}$. *En efecto,* $\sigma_n(t_n) = \sigma_n(s_n)$, *pero* $\sigma_n(t_n)$ *se corresponde con el árbol completo de altura* $n + 1$, *que se sabe que tiene tamaño* 2^{n+1}, *lo cual es exponencial con respecto al tamaño de la entrada.*

Existen implementaciones alternativas que evitan este problema. la idea principal consiste en utilizar una estructura de datos que permita implementar un DAG (grafo acíclico dirigido). Así, en el momento de realizar una sustitución de una variable x por un término t, en lugar de copiar todo el término t en cada posición donde aparezca la variable x, lo que haremos será situar en esa posición un puntero al término t.

Si además hacemos una implementación cuidadosa de la búsqueda de la primera posición en la que los dos términos difieren, intentando aprovechar el hecho de que muchos subtérminos serán iguales (porque originalmente se correspondían con la misma variable), podemos conseguir que el coste total del algoritmo sea polinómico. Hay varios algoritmos que consiguen este objetivo. Se puede obtener una lista de ellos en [**HV09**]. Además, se comprueba que el coste del Robinson modificado con una implementación cuidadosa es de $\mathcal{O}(n^3)$. Existen algoritmos con una complejidad inferior, pero que en caso medio se comportan peor que este.

3.3. Matching

Una variante del problema de unificación es el problema de matching. Este problema consiste en unificar dos términos, t_1, t_2, pero con la condición de que uno de los dos, por ejemplo t_2, sea un término ground, es decir, que no contiene variables.

El problema de matching de primer orden es entonces un caso particular del problema de unificación de primer orden, y por tanto se puede resolver también con los mismos algoritmos, como por ejemplo el de Robinson. Sin embargo, se

```
Matching(s : término, t : término): booleano
// Precondición: t no tiene símbolos de variable
    σ: sustitución
    σ := ∅
    Para cada p ∈ Pos(s) en preorden:
        Si s(p) es una símbolo de función y t(p) es un símbolo de función diferente, Entonces
            Devuelve falso
        Si s(p) es una variable que no está definida en σ, Entonces
            σ := {s(p) ↦ t(p)} ∘ σ
    FinMientras
    Si σ(s) = t, Entonces
        Devuelve cierto
    Si no
        Devuelve falso
```

FIGURA 3.2. Esquema general del algoritmo para resolver matching de primer orden

puede aprovechar el hecho de que t_2 no contiene variables para implementar un algoritmo mucho más eficiente, como el que se puede ver en la Figura 3.2.

Se puede ver que en este caso, en vez de ir haciendo sucesivas sustituciones cada vez que veamos una diferencia, podemos simplemente guardar todas las sustituciones que tenemos previsto hacer, y hacerlas todas al final, para a continuación comprobar si el término resultante es equivalente. Esto nos evita el caso del Ejemplo 3.2, en el que el tamaño crecía hasta llegar a exponencial.

Este algoritmo, con la implementación del esquema de la Figura 3.2 podría tener un caso peor en el que el coste fuera cuadrático, ya que al hacer la sustitución, es posible tener que sustituir un término de tamaño $|t|$ en un número de posiciones del orden de $|s|$. Sin embargo este problema no es difícil de solucionar, simplemente teniendo en cuenta que podemos calcular $|\sigma(s)|$ antes de realizar las sustituciones, y si $|\sigma(s)| \neq |t|$ entonces no es necesario que las realicemos, ya que necesariamente los términos serán diferentes.

Capítulo 4
Resultados sobre números primos

En la implementación de algunos de los algoritmos de este PFC se necesitaba generar aleatoriamente un número primo. Como esta tarea tiene suficiente dificultad de por sí, dedicaremos este capítulo a explicar los resultados necesarios utilizados para la implementación de dicho algoritmo.

4.1. El problema

Dado un número natural n, nos interesa generar aleatoriamente un número primo p que sea menor que n. Además queremos que la generación sea uniforme. Es decir, que todos los primos comprendidos en el rango $[2, \ldots, n]$ tengan las mismas probabilidades de ser elegidos. Nos interesa también que la generación sea rápida, ya que la obtención de un primo forma parte de un algoritmo probabilístico, como veremos en el Capítulo 5, y por tanto será un proceso al que llamaremos repetidamente, con el fin de reducir la probabilidad de error.

Parece claro que este problema se puede dividir en dos subproblemas. Por un lado necesitamos un test de primalidad, que nos permita saber si un cierto p es un número primo o es compuesto. Por otro lado, una vez que sepamos distinguir entre primos y compuestos, tenemos que encontrar una manera de generar aleatoriamente un primo de forma uniforme. Trataremos estos dos subproblemas de forma independiente.

4.2. Test de primalidad

Un test de primalidad es un algoritmo que decide si un número dado es primo o no. El algoritmo trivial (probar entre todos los números menores si existe algún divisor), y su mejora de probar sólo hasta la raíz cuadrada, son demasiado lentos para nuestro objetivo. En efecto, si el tamaño de un número k es $n := |k|$, para decidir la primalidad de k necesitaremos un coste de $\mathcal{O}(2^{\frac{n}{2}})$, que no nos podemos permitir.

```
EsPrimo( p : natural ): booleano
  Si 2^{p-1} ≢ 1 mód p, Entonces
    Devuelve falso.
  Si 3^{p-1} ≢ 1 mód p, Entonces
    Devuelve falso.
  Si 5^{p-1} ≢ 1 mód p, Entonces
    Devuelve falso.
  Si 7^{p-1} ≢ 1 mód p, Entonces
    Devuelve falso.
  Devuelve cierto
```

FIGURA 4.1. Test de Fermat para decidir la primalidad de un natural p

De hecho, el mejor algoritmo que se conoce hasta ahora tiene coste polinómico $\mathcal{O}(n^6)$ [**AKS02**]. Sin embargo ese algoritmo no se utiliza, por tener un coste muy elevado en la práctica, incluso para los rangos en los que nos moveremos. Así que utilizaremos tests probabilísticos que, a cambio de tener una pequeña probabilidad de error, nos permite costes mucho más reducidos.

En este PFC utilizaremos el test de primalidad de Fermat. Este test se basa en el Teorema Pequeño de Fermat, que afirma lo siguiente:

Teorema 4.1 (Pequeño de Fermat)**.** *Sea p un primo. Entonces para cualquier número natural a, se cumple que $a^p \equiv a \mod p$.*

Equivalentemente, si p es primo entonces para cualquier natural a se cumple que $a^{p-1} \equiv 1 \mod p$. El recíproco no es necesariamente cierto (es decir, que se cumpla la relación para algún valor a no implica que p sea primo). Sin embargo, se ha comprobado que el número de falsos positivos es muy reducido. Por tanto lo que haremos será seleccionar algunos valores de a y comprobar si se cumple la igualdad. En caso de que no se cumpla para alguna elección de a, p no podrá ser primo, puesto que contradeciría el Teorema 4.1. Si se cumple para todos supondremos que es primo, aunque puede ser que no lo sea. El algoritmo concreto utilizado es el de la Figura 4.1.

Dado un natural a, diremos que p es un pseudoprimo de Fermat, o simplemente pseudoprimo, de base a, si p no es primo pero pasa el Test de Fermat en base a, es decir $a^{p-1} \equiv 1 \mod p$. En el Ejemplo 4.2 se muestran algunos de estos pseudoprimos para algunas bases.

Ejemplo 4.2. *El menor pseudoprimo en base 2 es el 341. Eso quiere decir que si tomamos un número compuesto menor que 341, como por ejemplo 15, obtendremos que $2^{14} = 16384$, y $16384 \equiv 4 \mod 15$. Como $4 \neq 1$ deducimos inequívocamente que 15 es un número compuesto. En cambio, $2^{340} \equiv 1 \mod 341$, lo que llevaría a pensar que 341 es un número primo. Sin embargo $341 = 11 \cdot 31$, es decir, es un número compuesto.*

El menor pseudoprimo en base 3 es el 91. De nuevo, cualquier número compuesto menor que 91 pasará el test de primalidad. En particular $3^9 = 19683$, y $19683 \equiv$

$3 \text{ mód } 10$*, por lo que 10 es compuesto. Pero* $3^{90} \equiv 1 \text{ mód } 91$ *a pesar de que* $91 = 7 \cdot 13$*.*

Cabe destacar que cualquier primo pasará el test de primalidad para cualquier base. Por ejemplo, 11 es primo. Eso quiere decir que $\forall a \in \mathbb{N}$ *se cumple que* $a^{10} \equiv 1 \text{ mód } 11$*. En efecto,* $2^{10} = 1024 \equiv 1 \text{ mód } 11$*. También se cumple si a no es primo:* $6^{10} = 60466176 = 5496925 \cdot 11 + 1$*.*

Es evidente que el algoritmo de la Figura 4.1 funciona siempre si p es primo, pero fallará en los pseudoprimos de base 2, 3, 5 y 7 simultáneamente. Afortunadamente, el conjunto de pseudoprimos en todas esas base es muy reducido. En particular, para $p < 10^6$, sólo hay 19 pseudoprimos p de base 2, 3, 5 y 7. El menor de ellos es el 29341, que se puede descomponer como $13 \cdot 37 \cdot 61$, pero el algoritmo dirá que es primo.

Este test de primalidad lo utilizaremos como parte de un algoritmo probabilístico, de forma que este margen de error lo único que hará será aumentar la probabilidad de error, pero dada la baja densidad de pseudoprimos de Fermat en general, y en particular en los intervalos de tamaños en los que nos movemos, podemos afirmar que la probabilidad de error apenas varía por el test de primalidad utilizado.

En cuanto al coste de este algoritmo, dado que utilizaremos exponenciación rápida para calcular el valor de $a^{p-1} \text{ mód } p$, y que realizaremos un número constante de dichas exponenciaciones (cuatro), podemos deducir que nuestro algoritmo utilizará un tiempo del orden de $\mathcal{O}(\log p)$.

4.3. Generación aleatoria de primos

Una vez resuelto el problema de detectar la primalidad de un cierto p, falta resolver el de elegir un primo al azar. La solución trivial en este caso sería generar todos los primos del rango que buscamos, y elegir aleatoriamente uno de ellos, pero esta solución no nos vale dado que tendríamos que recorrer todos los valores de nuestro rango, y hay un número exponencial respecto al tamaño de la entrada. Otra opción es precalcular una única vez todos los primos del rango que necesitamos, guardarlos en un fichero, y acceder a él cada vez que queramos generar un primo, pero nuestro algoritmo podría necesitar primos hasta 10^9, y en ese rango hay unos 30 millones de primos, por lo que necesitaríamos varios centenares de megabytes para almacenarlos, y dado que el algoritmo se debe poder ejecutar desde un servidor no es una opción técnicamente viable.

Ante eso parece claro que tenemos que encontrar los primos cada vez. Como podemos generar cualquier número dentro de un rango se nos presentan dos algoritmos para generar un número primo: El de la Figura 4.2 y el de la Figura 4.3.

```
Generar primo( a : natural ): natural
    //Precondición: a ≥ 2
    //Devuelve un número primo aleatorio entre 2 y a
    p: natural
    p := aleatorio (1, a) //genera un número aleatorio entre 1 y a
    Mientras p no sea primo, Hacer
        p := p + 1
        Si p > a, Entonces
            p := 2
    Devuelve p
```

FIGURA 4.2. Una forma de generar un número primo aleatorio entre 2 y a

```
Generar primo( a : natural ): natural
    //Precondición: a ≥ 2
    //Devuelve un número primo aleatorio entre 2 y a
    p: natural
    p := aleatorio (1, a) //genera un número aleatorio entre 1 y a
    Mientras p no sea primo, Hacer
        p := aleatorio (1, a)
    Devuelve p
```

FIGURA 4.3. Otra forma de generar un número primo aleatorio entre 2 y a

La principal ventaja del algoritmo de la Figura 4.2 es que acaba siempre. En efecto, si ejecutamos el algoritmo de la Figura 4.3 podría suceder que nunca generáramos un número primo, aunque esto sucederá con probabilidad 0. El número medio de iteraciones, sin embargo, está acotado en ambos casos por $\mathcal{O}(\log a)$, como se aprecia en el Teorema 4.3.

Teorema 4.3 (de los números primos)**.** *Sea $\pi(x)$ el número de primos menores o iguales que x. Entonces:*

$$\lim_{x\to\infty} \frac{\pi(x)}{x/\ln(x)} = 1$$

Este hecho se peude expresar usando notación asintótica como:

$$\pi(x) \sim \frac{x}{\ln x}$$

La principal ventaja del algoritmo de la Figura 4.3 es que todos los números primos tienen la misma probabilidad de ser elegidos. En el caso del algoritmo de la Figura 4.2 un primo tendrá más probabilidades de ser escogido cuanto más lejos esté del primo anterior, como se puede ver en el Ejemplo 4.4

Ejemplo 4.4. *Usando el algoritmo de la Figura 4.2, si el primer valor de p que generamos aleatoriamente es* 6 *o* 7*, entonces el primo que devolverá será el* 7*. Sin embargo, si generamos inicialmente el* 8*,* 9*,* 10 *u* 11*, el primer primo que encontraremos será el* 11*. Es decir, suponiendo que todos los números de nuestro*

rango los generamos con la misma probabilidad, deducidmos que es el doble de probable que elijamos 11 *a que elijamos* 7.

Como el algoritmo que describiremos en la Sección 5 exige que los primos que seleccionemos sean equiprobables, en este proyecto hemos implementado el algoritmo de la Figura 4.3, es decir, generar números aleatorios hasta que obtengamos un primo.

Finalmente podemos deducir la siguiente Proposición, que utilizaremos más adelante.

Proposición 4.5. *Dado un natural* $a \geq 2$ *podemos generar aleatoriamente un número primo o pseudoprimo de Fermat entre* 2 *y* a *en tiempo medio* $\mathcal{O}(\log^2 a)$.

Demostración. El coste de generar un número aleatorio $p \in [1, \ldots, a]$ es de $\log a$, así como comprobar si p es primo o pseudoprimo de Fermat. El número medio de iteraciones de nuestro algoritmo es también de $\log a$, por lo que finalmente obtenemos el coste esperado □

Capítulo 5
Compresión de palabras

La parte más importante de este PFC consiste en aplicar los algoritmos de unificación y matching a términos comprimidos. En general, cualquier algoritmo se puede aplicar a un término comprimido, sea cual sea el tipo de compresión, siempre que lo descomprimamos previamente, claro está. La pregunta es entonces si existe algún tipo de compresión que nos permita aplicar ciertos algoritmos sin necesidad de descomprimir el término.

Las compresiones más comunes como .zip o .tar tienen un buen ratio de compresión, pero dado que actúan comprimiendo directamente bits es probable que el resultado de la compresión no guarde apenas relación con el archivo original, por lo que para la mayoría de algoritmos lo más eficiente es descomprimir el archivo y luego aplicarlos.

En nuestro PFC utilizaremos un tipo de compresión basado en gramáticas incontextuales. El principal problema de esta compresión es que el problema de, dado un término, obtener la mejor compresión posible mediante este sistema, se ha demostrado que es NP-completo, aunque se han encontrado buenas aproximaciones como la que aparece en [**Ryt03**], que permite una compresión de tamaño como mucho $\log n$ veces más que la mejor compresión, siendo n el tamaño original de la palabra o término.

La principal ventaja es que podemos realizar operaciones sobre palabras o términos sin necesidad de descomprimirlos. Más aún: para ciertos algoritmos, el coste de aplicarlos sobre una palabra o término comprimido es un coste relativo al tamaño de la palabra o término comprimido, y no al tamaño de la palabra o término original.

En [**Lif06**] se explican algunos de estos algoritmos para el caso de palabras. Por ejemplo, en tiempo $\mathcal{O}(n^4)$ podemos decidir si dos compresiones representan la misma palabra, comprobar si una aparece dentro de la otra, o incluso contar el número de ocurrencias. Estas comprobaciones se pueden realizar en tiempo lineal en caso de términos descomprimidos, por lo que necesitaríamos que la palabra comprimida tuviera tamaño inferior a la raíz cuarta del tamaño de la palabra original para que salga rentable aplicar los algoritmos sin descomprimir las palabras.

A continuación pasamos a definir la implementación de la compresión de palabras que utilizaremos.

5.1. Singleton Context Free Grammars

La estructura que utilizaremos para comprimir palabras será la Singleton Context Free Grammar (SCFG). Una SCFG es una gramática incontextual que produce una única palabra.

Más formalmente, una gramática incontextual es una tupla $\langle \mathcal{N}, \Sigma, R \rangle$, donde $\mathcal{N}$ es un conjunto finito de símbolos no terminales, Σ es un conjunto finito de símbolos (un alfabeto), y R es un conjunto finito de reglas de la forma $N \to \alpha$, donde $N \in \mathcal{N}$ y $\alpha \in (\mathcal{N} \cup \Sigma)^*$.

Una SCFG es una gramática incontextual en la que cada variable aparece en una única parte izquierda de regla, y que además está en forma normal de Chomsky y es no recursiva, es decir, que existe una numeración de los símbolos no terminales, $\{N_1, \ldots, N_n\}$, tales que toda regla $N_i \to \alpha$ es tal que $\alpha = a$, con $a \in \Sigma$, o bien $\alpha = N_j N_k$, donde además $i > j, k$.

Se puede comprobar fácilmente que un símbolo no terminal de una SCFG genera una única palabra. Denotaremos la palabra generada por el no términal N de la SCFG G como $w_{G,N}$, o w_N cuando G quede claro por el contexto. Definimos entonces $w_{G,N}$ dependiendo de la regla de G que tenga como parte izquierda N, es decir, de la regla $(N \to \alpha) \in R$. Si $\alpha = a$, siendo a un símbolo terminal, entonces $w_{G,N} = a$. Si en cambio $\alpha = N_i N_j$ entonces la palabra generada por N se define recursivamente como $w_{G,N} = w_{G,N_i}.w_{G,N_j}$.

Con este sistema de compresión se puede conseguir que la palabra generada por un símbolo no terminal tenga un tamaño exponencial con respecto al número de reglas de la SCFG. Esto se puede apreciar mejor en el siguiente ejemplo:

Ejemplo 5.1. *Fijado un natural n, Consideramos la SCFG $G = \langle \mathcal{N}, \Sigma, R \rangle$, donde $\mathcal{N} = \{N_i \mid 0 \leq i \leq n\}$, $\Sigma = \{a\}$ y $R = \{N_0 \to a\} \cup \{N_i \to N_{i-1}N_{i-1} \mid 1 \leq i \leq n\}$. La SCFG tiene $n + 1$ reglas, y podemos comprobar fácilmente que el no terminal N_n genera la palabra a^{2^n}, que tiene por tanto tamaño exponencial respecto al número de reglas.*

Sin embargo la palabra abbaa no se puede expresar con una SCFG que tenga menos de 6 reglas. Un ejemplo sería la SCFG $G = \langle \mathcal{N}, \Sigma, R \rangle$, con $\mathcal{N} = \{N_i \mid 0 \leq i \leq 5\}$, $\Sigma = \{a, b\}$ y R el conjunto de reglas definido a continuación:

- $N_5 \to N_4 N_0$
- $N_4 \to N_3 N_0$
- $N_3 \to N_2 N_1$
- $N_2 \to N_0 N_1$
- $N_1 \to b$

- $N_0 \to a$

5.2. Algoritmos deterministas

Utilizando esta estructura para comprimir una palabra se pueden aplicar ciertos algoritmos sin necesidad de descomprimir la palabra. Si, por ejemplo, tenemos una SCFG G con dos no terminales, N_v y N_w que representan las palabras v y w respectivamente, podemos calcular el número de ocurrencias de v como subpalabra de w, y por tanto podemos decidir si v es subpalabra de w (si el número de ocurrencias es al menos 1), y también si las dos palabras representadas son iguales (si w también ocurre en v). Además, si nos dan un valor k, podemos decidir si v es subpalabra de w entre las posiciones k y $k + |v| - 1$.

Estos algoritmos se fundamentan en una estructura llamada AP-table. Esta estructura será una matriz cuadrada con tantas filas y columnas como no terminales tenga la gramática G. Si AP es la AP-table de la gramática G, entonces en la posición $AP[i, j]$ se guardará la información sobre las ocurrencias de N_i en N_j. Los dos principales resultados sobre la AP-table son:

- Si G es una SCFG con n no-terminales, entonces se puede calcular la AP-table de G en tiempo $\mathcal{O}(n^4)$.
- Si tenemos la AP-table calculada, entonces podemos contar el número de ocurrencias de la palabra generada por cualquier no terminal la palabra generada por en cualquier otro en tiempo $\mathcal{O}(n)$.

Para ver la demostración y detalles de implementación se puede consultar la referencia [**Lif06**]

El problema de decidir la primera posición en la que una palabra aparece en otra también se puede resolver trivialmente con la AP-table, y es parte del algoritmo que resuelve el problema de unificación con términos comprimidos, que es el tema principal del PFC. Sin embargo el coste $\mathcal{O}(n^4)$ en el cómputo de la AP-table es claramente el cuello de botella de dicho algoritmo, como se puede ver en [**GGSS09**], por lo que buena parte del PFC consistía en buscar una alternativa menos costosa, que es la que se explica en el siguiente apartado.

5.3. Igualdad probabilística de palabras

Hemos visto antes que existen algoritmos que se pueden aplicar a palabras comprimidas mediante SCFG, consiguiendo un coste que está en función del tamaño de la gramática, y no del tamaño del término comprimido. El principal problema es que el coste puede ser considerablemente mayor que el coste de los algoritmos equivalentes aplicados a palabras comprimidas, por lo que sólo saldría rentable aplicarlos si la compresión es muy buena, lo que sólo se consigue en algunos casos límite, como el del Ejemplo 5.1.

Por ejemplo, saber si dos palabras son iguales se puede calcular fácilmente en tiempo lineal. Pero contar el número de ocurrencias de una palabra en otra también se puede calcular en tiempo lineal, utilizando el algoritmo de Knuth-Morris-Pratt [**KJP77**]. El algoritmo que utiliza la AP-table necesita tiempo $\mathcal{O}(n^4)$, por lo que haria falta que el tamaño de la gramática comprimida fuera del orden de $o(n^{\frac{1}{4}})$ con respecto al tamaño de la palabra original para mejorar el coste.

Sin embargo, utilizando algoritmos alternativos con una pequeña probabilidad de error el coste se mejora drásticamente. En este caso utilizaremos la implementación de [**SSS09**], que pasamos a explicar.

5.3.1. Valor de una palabra. Sea G una gramática, y sea Σ el conjunto de símbolos terminales de G. Llamaremos b al tamaño del alfabeto, es decir, $b := |\Sigma|$. Asignaremos a cada símbolo terminal un número natural diferente entre 0 y $b-1$, y denotaremos por i_a el valor del símbolo $a \in \Sigma$. Definimos el valor de una palabra w, que denotaremos i_w, como el siguiente número natural:

$$i_w = \begin{cases} 0 & \text{si} \quad w = \lambda \\ b \cdot i_{w'} + i_a & \text{si} \quad w = w'.a \text{ y } a \in \Sigma \end{cases}$$

Es decir, consideramos cada palabra como la representación de un número natural en base b, asignando a cada símbolo un posible valor entre 0 y $b-1$, como se expresa en el siguiente ejemplo.

Ejemplo 5.2. *Consideramos un alfabeto con tres símbolos,* $\Sigma = \{a, b, c\}$*. Daremos valores* 0*,* 1 *y* 2 *respectivamente, es decir,* $i_a = 0$*,* $i_b = 1$*,* $i_c = 2$*. Como en este caso el valor de* $b = |\Sigma|$ *es* 3*, el valor de la palabra* $w = abcabc$ *será:*

$$i_w = i_a \cdot b^5 + i_b \cdot b^4 + i_c \cdot b^3 + i_a \cdot b^2 + i_b \cdot b + i_c =$$

$$= 0 \cdot 243 + 1 \cdot 81 + 2 \cdot 27 + 0 \cdot 9 + 1 \cdot 3 + 2 = 140$$

Nótese que es posible que dos palabras tengan el mismo valor. Por ejemplo la palabra $\bar{w} = bcabc$ *también tiene valor* $i_{\bar{w}} = 140$*.*

De esta definición podemos deducir trivialmente que si dos palabras son iguales tendrán el mismo valor, pero no podemos afirmar el recíproco, por lo que acabamos de ver. Ante esto tenemos dos soluciones: o bien asignar valores entre 1 y b, en lugar de entre 0 y $b-1$, o bien comprobar también el tamaño de las palabras. Optaremos por esta segunda opción, utilizando el hecho reflejado en la siguiente proposición:

Proposición 5.3. *Sean* v *y* w *palabras sobre un alfabeto* Σ*, al que hemos asignado un valor* i *a cada uno de sus símbolos. Entonces* $v = w \Leftrightarrow i_v = i_w \wedge |v| = |w|$*.*

Este hecho, de fácil demostración, se basa en que, dado que las palabras son representaciones de números en base $b-$arias, la única forma de conseguir dos

palabras con el mismo valor pero diferentes es añadiendo ceros por la izquierda, lo que podemos comprobar a la hora de calcular el tamaño de ambas palabras.

A partir de aquí obtenemos un algoritmo trivial que nos permite ver si dos SCFG representan la misma palabra: calculamos el valor de la palabra generada por cada una de ellas, y comprobaremos si es el mismo. El problema es que el valor de una palabra tiene un tamaño del orden del tamaño de la palabra descomprimida. De hecho este algoritmo es equivalente al de descomprimir ambas palabras y comprobar si son iguales.

Pero la idea se puede reaprovechar. En lugar de calcular el valor de una palabra, seleccionamos un número suficientemente grande, p, preferiblemente primo, y calculamos su valor módulo p. Es evidente que si dos palabras tienen diferente valor módulo p, entonces también tienen diferente valor, y por tanto son palabras diferentes. Sin embargo nuestro algoritmo podría dar falsos positivos, es decir, podría decir que dos palabras son iguales cuando en realidad son diferentes, como se muestra en el siguiente ejemplo:

Ejemplo 5.4. *Con el alfabeto $\Sigma = \{a, b, c\}$, y los valores $i_a = 0$, $i_b = 1$, $i_c = 2$, si tomamos un valor p para aplicar nuestro algoritmo, siempre podremos encontrar dos palabras con el mismo tamaño y con el mismo valor módulo p, que sin embargo son palabras diferentes. La forma más sencilla de obtener esas palabras es la representación en base $|\Sigma|$ de p, a la que podemos llamar w_p, y la representación de 0, que son tantos símbolos a como el tamaño de w_p.*

Por ejemplo, si p es 107, en base 3 lo podemos representar como 10211_3. Es decir, $w_p = bacbb$. Si ahora consideramos la palabra con cinco símbolos a, $w_0 = aaaaa$, tenemos que $|w_p| = 5 = |w_0|$, y además $i_{w_p} \equiv i_{w_0} \bmod p$, por lo que nuestro algoritmo concluirá que w_p y w_0 son iguales, cuando realmente no lo son.

El valor de p más pequeño que tendríamos que tomar para distinguir siempre dos palabras de tamaño n sobre un alfabeto con b símbolos es el menor natural que no se puede representar con n símbolos en base b, es decir, b^n. Por lo que de nuevo estaríamos realizando un proceso equivalente a descomprimir las palabras y compararlas. Lo que haremos será seleccionar un valor de p más pequeño, que pueda dar falsos positivos, pero con una probabilidad controlada.

En general, si se toma un valor de p suficientemente grande y primo, por ejemplo, 1.000.000.007, que es uno de los mayores primos que cabe en un entero de 32 bits, el algoritmo funcionará bien prácticamente siempre. Si utilizamos palabras aleatorias sólo una pareja de cada mil millones dará un falso positivo, por lo que utilizar un p fijo en la práctica es un buen método para implementar este algoritmo. Sin embargo en la teoría no se puede considerar un algoritmo probabilístico, ya que hay entradas con las que fallaría siempre.

En general, un algoritmo probabilístico exige que para cualquier pareja de palabras la probabilidad de error sea menor que un cierto valor $P < 1$. Lo que

podemos hacer entonces es seleccionar un primo aleatoriamente y aplicar el algoritmo. Si nos dice que los valores de ambas palabras son diferentes acabamos, pues hemos visto antes que en este caso las dos palabras son necesariamente diferentes. Sin en cambio son iguales podemos volver a aplicar el algoritmo con otro primo aleatorio. Este proceso lo podemos repetir suficientes veces hasta que la probabilidad de error sea tan baja como queramos.

El motivo de seleccionar primos y no números cualquiera es porque si dos palabras tienen el mismo valor módulo p y q, entonces también tendrán el mismo valor módulo el mínimo común múltiplo de p y q. Si p y q son primos diferentes entonces tendrán el mismo valor módulo $p \cdot q$. Así conseguiremos que la probabilidad de error baje.

Por tanto, si nos dan dos palabras v, w, y hemos calculado que $|v| = |w| = m$, tenemos que seleccionar un conjunto de primos $\mathcal{P} = \{p_1, \ldots, p_n\}$ tal que si $v = w$ entonces $i_v \equiv i_w$ mód p_i para todo $p_i \in \mathcal{P}$, pero si $v \neq w$ entonces $i_v \not\equiv i_w$ mód p_i para al menos la mitad de primos p_i de $\mathcal{P}$. Nos interesa además que el conjunto no dependa de las palabras en concreto, sino de su longitud m.

Lo que haremos será, dada la longitud m, calcular una cota $k(m)$, y seleccionar un primo al azar entre 2 y $k(m)$ utilizando el algoritmo de la Figura 4.3. En [**SSS09**], Fact 3.1, se da una cota de $k = 2c \cdot \ln a$, siendo c proporcional al logaritmo de la probabilidad deseada, y a el máximo valor representable para palabras de tamaño m. El valor de c es constante, ya que la probabilidad que deseamos es constante. El valor de a se puede acotar, ya que con palabras de tamaño m, con un alfabeto Σ de tamaño $|\Sigma| = b$, el maximo valor alcanzable es el de $b^m - 1$, por lo que $\ln a < \ln b \cdot m$. Así $k(m) = 2c \cdot \ln b \cdot m$. En este PFC hemos dado un valor de $k(m) = 35m$, de lo que se deduce que $c \cdot \ln b = 17{,}5$. Dado que el tamaño del alfabeto está acotado por 26 por motivos técnicos (hemos utilizado caracteres entre la a y la z), podemos concluir que $c \approx 5{,}37$, lo que nos lleva a una probabilidad de error P algo menor a 0,5.

Un esquema del algoritmo utilizado es el de la Figura 5.1.

5.3.2. Cálculo de los valores. Ya sólo falta explicar cómo se calculan los valores que hemos usado en el algoritmo de la Figura 5.1, es decir, $|w_N|$, $|\Sigma|^{|w_N|}$ mód p y i_{w_N} mód p. En [**SSS09**] se explica la manera de calcular dichos valores, que pasamos a resumir.

- Para calcular $|w_N|$, si existe $a \in \Sigma$ tal que $(N \to a) \in R$ entonces $|w_N| = 1$. Si, por el contrario, $(N \to ML) \in R$, y suponiendo que hemos calculado el tamaño de w_M y w_L, entonces $|w_N| = |w_M| + |w_L|$.
- Para calcular $|\Sigma|^{|w_N|}$ mód p, si existe $a \in \Sigma$ tal que $(N \to a) \in R$ entonces $|w_N| = b$. Si, por el contrario, $(N \to ML) \in R$, y suponiendo que hemos calculado ya $|\Sigma|^{|w_M|}$ mód p y $|\Sigma|^{|w_L|}$ mód p, entonces $|\Sigma|^{|w_N|} \equiv |\Sigma|^{|w_M|} \cdot |\Sigma|^{|w_L|}$ mód p. Alternativamente podríamos calcular directamente

```
Igualdad( G : Gramática, X : no terminal, Y : no terminal ): booleano
// Precondición: G = ⟨N, Σ, R⟩ es una SCFG y X, Y ∈ N
    Para cada N ∈ N:
        Calcular el tamaño de w_N
    Si |w_X| ≠ |w_Y|, Entonces
        Devuelve falso
    Generar un primo p aleatorio entre 2 y 35 · |w_X|
    Para cada N ∈ N:
        Calcular |Σ|^|w_N| mód p
    Para cada N ∈ N:
        Calcular i_{w_N} mód p
    Si i_{w_X} ≡ i_{w_Y} mód p, Entonces
        Devuelve cierto
    Si no
        Devuelve falso
```

FIGURA 5.1. Esquema general del algoritmo probabilístico para decidir la igualdad de dos palabras comprimidas mediante una SCFG

$|\Sigma|^{|w_N|}$ mód p a partir de $|w_N|$ mediante el algoritmo de exponenciación rápida, pero añadiría un factor logarítmico en el coste que podemos evitar haciendo un único producto.

- Para calcular i_w mód p, si existe $a \in \Sigma$ tal que $(N \to a) \in R$ entonces $i_N \equiv i_a$ mód p. Si, por el contrario, $(N \to ML) \in R$, y suponiendo que hemos calculado i_M mód p e i_L mód p, entonces $i_N \equiv i_M \cdot |\Sigma|^{|w_L|} + i_L$ mód p.

Nótese que, dado que las definiciones de arriba son recursivas, tenemos que tener cuidado de haber calculado previamente los valores que necesitamos. Para eso tenemos dos opciones. O bien ordenamos los no terminales, $\{N_1, \ldots, N_k\}$, de forma que si $N_i \to N_j N_k$ entonces $i < j, k$, y de este modo calcular dichos valores por orden decreciente, o bien calculamos recursivamente los valores que necesitamos a cada momento, teniendo en cuenta que no calcularemos dos veces un valor para un mismo no terminal. En este PFC hemos optado por la primera opción, ya que la construcción de la estructura del SCFG nos permite mantener un orden que cumple la propiedad deseada.

Finalmente calcularemos el coste del algoritmo. Si la gramática G tiene n símbolos no terminales, quiere decir que la palabra generada tendrá tamaño de hasta 2^n. Como nuestro primer paso es generar un primo hasta $35 \cdot 2^n$, usaremos la Proposición 4.5 para acotar el coste de esto como $\mathcal{O}(n^2)$. El siguiente paso es calcular todos los valores. Tenemos que calcular $3 \cdot n$ valores, y cada uno de ellos lo calculamos en tiempo constante utilizando los valores ya calculados, lo que nos lleva a un coste aparente de $\mathcal{O}(n)$. Sin embargo no estamos teniendo en cuenta el coste de las operaciones (suma y producto), que normalmente se consideran constantes, pero en este caso el tamaño de los números con los que operaremos es mayor cuanto mayor es la entrada. En particular trabajaremos con números hasta $35 \cdot 2^n$, y el coste de sumar o multiplicar dos de ellos es prácticamente $\mathcal{O}(n)$.

En cualquier caso esto no varía el coste final de nuestro algoritmo, $\mathcal{O}(n^2)$, como afirma la siguiente proposición:

Proposición 5.5. *Dada una SCFG $G = \langle \mathcal{N}, \Sigma, R \rangle$, y dos no terminales $X, Y \in \mathcal{N}$, podemos decidir en tiempo $\mathcal{O}(|\mathcal{N}|^2)$ si $w_X = w_Y$, con una probabilidad de error menor de 0.5 si $w_X \neq w_Y$ y de 0 si $w_X = w_Y$.*

5.3.3. Alternativa sin utilizar números primos. En nuestro algoritmo para decidir la igualdad de dos términos comprimidos mediante SCFG hemos utilizado siempre un primo p que es el módulo de nuestras operaciones. Sin embargo existe una alternativa que no necesita que el módulo sea primo, como se explica en [**SSS10**], que es una segunda versión del artículo de [**SSS09**].

El algoritmo es exactamente el mismo que el que utiliza números primos. La única diferencia es que el módulo con el que haremos los cálculos no tiene por qué ser primo. A cambio, para mantener la probabilidad de error, necesitaremos que el rango sobre el cual se genere el número aleatorio sea más grande. En concreto será del orden del cuadrado del tamaño de la palabra, en lugar del tamaño de la palabra. Concretamente en nuestro PFC lo cuantificamos como $35 \cdot |w_n|^2$.

La principal ventaja de este algoritmo es que no tenemos que comprobar que el número que generamos sea primo. Esto nos elimina una fuente de error (el hecho de que pudiéramos generar pseudoprimos), pero además evita el caso peor de coste infinito que teníamos con nuestra generación de primos. Por lo demás el coste se mantiene a $\mathcal{O}(n^2)$.

Como inconveniente, tenemos que trabajar con números potencialmente más largos. Esto no se traduce en un aumento del coste asintótico, ya que si multiplicar dos números del orden de 2^n tiene coste amortizado $\mathcal{O}(n)$, multiplicar dos números del orden de 2^{2n} tiene coste amortizado $\mathcal{O}(2n) = \mathcal{O}(n)$. Sin embargo en las posteriores secciones veremos que trataremos de evitar el uso de enteros de tamaño arbitrario, ya que el coste constante que añaden a efectos prácticos es mucho mayor que el coste del algoritmo. Por tanto el hecho de necesitar trabajar con números más grandes limita el tamaño máximo de las palabras con las que podemos trabajar.

En este PFC están implementadas las dos versiones: usando primos y sin usar primos. Como veremos más adelante el tiempo real que tardan es muy parecido, aunque no llega a ser igual, como sí que es su coste asintótico.

5.4. Subcadena probabilística

Otro algoritmo que hemos implementado, muy relacionado con el de la Sección 5.3, es el de la subcadena probabilística. En este caso, además de la SCFG, G, y de dos no terminales, N y M, nos dan también un cierto entero k, y tenemos que decidir si la palabra generada por el no terminal N, w_N, está contenida en w_M entre las posiciones k y $k + |w_N| - 1$.

```
Valor( G : Gramática, X : no terminal, k : natural, l natural): natural
// Precondición: G = ⟨N, Σ, R⟩ es una SCFG y X, Y ∈ N, 1 ≤ k ≤ l ≤ |w_X|
// Postcondición: Devuelve el valor de w_X entre las posiciones k y l
    Si (X → a ∈ Σ) ∈ R, Entonces
        Devuelve i_a
    Si (X → YZ) ∈ R, Entonces
        Si l ≤ |w_Y|, Entonces
            Devuelve Valor (G, Y, k, l)
        Si k > |w_Y|, Entonces
            Devuelve Valor (G, Z, k − |w_Y|, l − |w_Y|)
        Devuelve Valor (G, Y, k, |w_Y|)·|Σ|^{l−|w_Y|} + Valor(G, Z, 1, l − |w_Y|)
```

FIGURA 5.2. Esquema general del algoritmo que calcula el valor de cierta subpalabra comprimida mediante una SCFG

De nuevo utilizaremos el concepto de valor de una palabra. En este caso no tenemos que comprobar que los tamaños de las palabras a comparar coincidan, ya que por definición serán iguales. Lo que sí tendremos que comprobar es que exista la subcadena. Es decir, que $k + |w_N| - 1 \leq |w_M|$. En caso de que esto no sea cierto no tenemos que hacer más cálculos, y podemos afirmar directamente que la palabra w_N no está contenida en la palabra w_M a partir de la posición k.

Por lo demás, el algoritmo parece sencillo. Simplemente calculamos el valor de la palabra w_N, y el valor de la palabra w_N entre las posiciones k y $k+|w_N|-1$. Si son iguales, devolvemos cierto, y si son diferentes, devolvemos falsos. La probabilidad de error será de nuevo menor a 0,5.

Lo único que falta por calcular es el valor de una palabra entre dos posiciones, k y l. La idea natural es tratar de calcular este valor recursivamente, y una posible implementación es la que aparece en la Figura 5.2. Previamente se han calculado todos los valores de la Sección 5.3, entre ellos el tamaño de las palabras o sus valores.

El problema de esta implementación es que puede llegar a tener coste exponencial. En efecto, si calculamos el valor de una subpalabra de w_X, donde X es un no terminal de $\mathcal{N}$, y hay una regla de la forma $X \to YZ$, entonces en caso peor haremos dos llamadas recursivas para calcular el valor de X: una que calcule el valor de una subpalabra de Y, y otra que calcule el valor de una subpalabra de Z. Si para calcular dichos valores hacemos dos llamadas recursivas, y así sucesivamente, podemos llegar a necesitar un número exponencial de llamadas.

Para evitar eso haremos una mejora muy simple: si $k = 1$ y $l = |w_X|$ entonces lo que estamos calculando es el valor de w_X, y como ya lo hemos calculado lo devolvemos directamente sin hacer ninguna llamada recursiva. Aunque parece que esta mejora sólo afecta a casos muy puntuales en realidad es muy sustancial, y evita el coste exponencial. Para ver esto necesitamos dos lemas.

Lema 5.6. *Dada una SCFG $G = \langle \mathcal{N}, \Sigma, R \rangle$, un no terminal $x \in \mathcal{N}$, y dos enteros $k \leq l$ tales que $k = 1$ o $l = |w_X|$. Entonces podemos calcular el valor de w_X entre las posiciones k y l con un número de llamadas recursivas del orden de $\mathcal{O}(|\mathcal{N}|)$.*

Demostración. Supondremos que $k = 1$, ya que si $l = |w_X|$ la prueba es análoga. Supondremos también que existe una regla de la forma $(X \to YZ)$ en R, ya que si existe una regla de la forma $(X \to a)$ el resultado es trivial. Utilizando el algoritmo de la Figura 5.2 con la mejora descrita anteriormente, tenemos dos opciones. Si $l \leq |w_Y|$ entonces haremos una única llamada para calcular el valor de una subpalabra de w_Y, y por hipótesis de inducción esto tomará tiempo lineal. Si por el contrario $l > |w_Y|$ haremos dos llamadas recursivas, pero una de ellas servirá para calcular el valor de w_Y entre las posiciones 1 y $|w_Y|$, es decir, el valor de w_Y, el cual ya hemos calculado y por tanto no realizaría más llamadas. La otra llamada recursiva es para calcular el valor de w_Z entre las posiciones 1 y $l - |w_Y|$, que por inducción se puede calcular usando un número lineal de llamadas. □

Por tanto con esta mejora no solo calculamos de forma rápida el valor de la subpalabra total, sino también el valor de todos los prefijos y sufijos de una palabra. Con este resultado ya podríamos calcular el valor de cualquier subpalabra, ya que el valor de w_X entre las posiciones k y l equivale al valor de w_X entre 1 y l, dividido entre el valor de w_X entre 1 y $k - 1$. Sin embargo no calcularemos el valor entero, ya que, como hemos visto anteriormente, esto equivale a descomprimir la palabra, sino que lo calcularemos módulo un cierto valor p. Si p es primo no hay problema, ya que $\mathbb{Z}_p$ es un cuerpo, y por tanto todo elemento tiene inversa. Pero como también trabajaremos con módulos no primos, como se explica en la Sección 5.3.3, no podremos usar esta opción. En realidad veremos que no es necesario, como se explica en el siguiente Lema:

Lema 5.7. *Dada una SCFG $G = \langle \mathcal{N}, \Sigma, R \rangle$, un no terminal $x \in \mathcal{N}$, y dos enteros $k \leq l$. Entonces podemos calcular el valor de w_X entre las posiciones k y l en con un número de llamadas recursivas del orden de $\mathcal{O}(|\mathcal{N}|)$.*

Es decir, no es necesario que la subpalabra sea un prefijo o un sufijo. Sin embargo usaremos el Lema 5.6 para demostrar este resultado.

Demostración. De nuevo supondremos que existe una regla de la forma $(X \to YZ)$ en R, ya que si no el resultado es trivial. Si $l \leq |w_Y|$ o $k > |w_Y|$ sólo haremos una llamada recursiva que calcule una subpalabra de Y o Z, respectivamente, y por tanto el resultado será cierto por inducción. Si, en cambio, $k \leq |w_Y| < l$ haremos dos llamadas recursivas, pero una será para calcular el valor de un sufijo de w_Y, y la otra para calcular el valor de un prefijo de w_Z. Como el Lema 5.6 afirma que un prefijo o sufijo lo podemos calcular con un número lineal de llamadas recursivas obtenemos el resultado que queríamos demostrar. □

Una vez que sabemos que un número lineal de llamadas recursivas es suficiente tenemos que preguntarnos cual es el coste de cada llamada recursiva. Como hemos

visto antes, no calcularemos los valores, sino los valores módulo un cierto p. Parece claro que el cuello de botella está en el cálculo de $|\Sigma|^{l-|w_Y|}$. Utilizaremos una exponenciación rápida, lo que en caso peor requerirá un número de operaciones del orden de $\log|w_X|$, y como las operaciones las realizamos módulo un valor p que puede tener tamaño del orden de $|w_X|$, necesitaremos un tiempo total del orden de $\mathcal{O}(\log^2|w_X|)$ para cada operación, lo que se traduce a un tiempo del orden de $\mathcal{O}(\log^2|w_X|)$ en el total del algoritmo.

Sin embargo hemos introducido otra mejora que permite reducir el coste del cálculo de $|\Sigma|^{l-|w_Y|}$. Haremos una inmersión de parámetro, de forma que además de devolver el valor de la subpalabra que buscamos, devolveremos también el valor de $|\Sigma|^{l-k+1}$. Es decir, de $|\Sigma|$ elevado al tamaño de la subpalabra. Como $l-|w_Y|$ es el tamaño de la subpalabra de la segunda llamada recursiva siempre tendremos ese valor disponible, y no tendremos que usar la exponenciación rápida para calcularlo. Además, podemos calcular $|\Sigma|^{l-k+1}$ a partir de las llamadas recursivas, ya que $|\Sigma|^{l-k+1} = |\Sigma|^{l-|w_Y|+|w_Y|-k+1} = |\Sigma|^{l-|w_Y|} \cdot |\Sigma|^{|w_Y|-k+1}$. Es decir, devolveremos el producto del valor de retorno de las llamadas recursivas.

Como caso base, si $k = 1$ y $l = |w_X|$ lo que tendremos que devolver será $|\Sigma|^{|w_X|}$, pero eso ya lo hemos calculado previamente, como se explica en la Sección 5.3, por lo que el coste será constante. Por tanto esta mejora nos permite calcular el valor de una subpalabra módulo un cierto valor p en tiempo $\mathcal{O}(|\mathcal{N}|^2)$, y consecuentemente podemos decidir probabilísticamente si una palabra es subpalabra de otra en tiempo también $\mathcal{O}(|\mathcal{N}|^2)$. Esto se refleja en la siguiente Proposición:

Proposición 5.8. *Dada una SCFG $G = \langle \mathcal{N}, \Sigma, R\rangle$, dos no terminales $X, Y \in \mathcal{N}$, y un cierto valor k, podemos decidir en tiempo $\mathcal{O}(|\mathcal{N}|^2)$ si w_X es la subpalabra de w_Y entre las posiciones k y $k+|w_X|-1$, con una probabilidad de error menor de 0.5 si no es subpalabra en esa posición y de 0 si lo es.*

Nótese que usando el mismo algoritmo de cálculo de valores de subpalabras podemos también decidir si una palabra entre posiciones l_1 y k_1 coincide con otra palabra entre las posiciones l_2 y k_2, simplemente calculando los valores correspondientes y comparándolos. Este principio será el que utilicemos en el algoritmo presentado en la siguiente sección.

5.5. Primera diferencia

Otro algoritmo que hemos implementado sobre SCFG, también relacionado con los anteriores, es aquel que permite calcular la primera posición en la que dos palabras generadas mediante SCFG difieren. Este algoritmo permitiría en particular decidir si dos palabras son iguales, que sería el caso en el que no difieren en ninguna posición.

Dicho algoritmo se basa principalmente en el algoritmo de subcadena, que decide si una palabra está contenida en otra en una determinada posición. De nuevo

PrimeraDiferencia(G : Gramática, X, Y : no terminal, l_1, k_1, l_2, k_2 : natural) : natural
// *Precondición:* $G = \langle \mathcal{N}, \Sigma, R \rangle$ *es una SCFG,* $X, Y \in \mathcal{N}$, $w_X \neq w_Y$, *y*
// $1 \leq l_1 \leq k_1 \leq |w_X|$, $1 \leq l_2 \leq k_2 \leq |w_Y|$
// *Postcondición: Devuelve la primera posición en la que* w_X
// *entre las posiciones* l_1 *y* k_1 *diifiere de* w_Y *entre las posiciones* l_2 *y* k_2
Si $l_1 = k_1$ o $l_2 = k_2$, Entonces
Calcular trivialmente
// *Supondremos que* $(X \to X_1 X_2) \in R$ *y* $(Y \to Y_1 Y_2) \in R$
Si $k_1 \leq |w_{X_1}|$, Entonces
Devuelve PrimeraDiferencia (G, X_1, Y, l_1, k_1, l_2, k_2)
Si $l_1 > |w_{X_1}|$, Entonces
Devuelve PrimeraDiferencia (G, X_2, Y, $l_1 - |w_{X_1}|$, $k_1 - |w_{X_1}|$, l_2, k_2)
// *Procedemos análogamente si* $k_2 \leq |w_{Y_1}|$ *o* $l_2 > |w_{Y_1}|$
// *Supondremos que* $|w_{X_1}| - l_1 \leq |w_{Y_1}| - l_2$, *ya que si no procederemos análogamente*
Si w_{X_1} entre las posiciones l_1 y $|w_{X_1}|$ coincide con w_{Y_1} entre las posiciones l_2 y $l_2 + |w_{X_1}| - l_2$, Entonces
Devuelve PrimeraDiferencia (G, X_2, Y, 1, $k_1 - |w_{X_1}|$, $|w_{X_1}|$, k_2) $+|w_{X_1}|$
Si no
Devuelve PrimeraDiferencia (G, X_1, Y_1, 1, $|w_{X_1}|$, 1, $|w_{X_1}|$)

FIGURA 5.3. Esquema general del algoritmo que calcula la primera posición en la que difieren dos subpalabras de dos palabras generadas mediante SCFG

tenemos la versión determinista y la versión probabilística, según qué algoritmo utilicemos para determinar si una palabra está o no contenida en otra. Si utilizamos el algoritmo de la Sección 5.2, que es determinista, nuestro algoritmo será también determinista, y si en cambio usamos el algoritmo de la Sección 5.4 nuestro algoritmo será probabilístico. La principal diferencia entre los dos es el coste, ya que hemos visto que el algoritmo probabilístico para decidir subcadena es más eficiente que el determinista, lo cual se ve reflejado en el coste final del algoritmo de primera diferencia que veremos a continación.

A la hora de implementarlo usaremos un algoritmo más general, que dados dos símbolos no terminales X e Y, y cuatro posiciones $1 \leq l_1 \leq k_1 \leq |w_X|$, $1 \leq l_2 \leq k_2 \leq |w_Y|$, calcula cual es la primera diferencia entre w_X entre las posiciones l_1 y k_1, y w_Y entre las posiciones l_2 y k_2. Para decidir entonces la primera diferencia entre las palabras generadas por no terminales X e Y bastará con llamar al algoritmo con parámetros X, Y, 1, $|w_X|$, 1, $|w_Y|$. El motivo de usar este algoritmo más general es que necesitamos la inmersión de parámetros para poder hacerlo recursivo. En la Figura 5.3 se explica un esquema de este algoritmo.

Se puede encontrar una justificación de la correctitud del algoritmo en [**GGSS09**]. Básicamente se basa en utilizar el algoritmo que nos permite calcular si una subpalabra es igual a otra subpalabra. Lo aplicaremos a la primera parte de cada palabra, y en caso de que coincidan buscaremos las diferencias directamente en la segunda parte.

En cuanto al coste del algoritmo dependerá de si utilizamos la versión determinista o probabilística. El número de llamadas recursivas que hagamos dependerá de la profundidad de la gramática, que en caso peor será $|\mathcal{N}|$. Además, en cada

llamada calculamos el valor de dos subcadenas, cosa que podemos hacer en tiempo cuadrático usando la APTable de la Sección 5.2, pero una vez que hayamos calculado la APTable, que necesita tiempo cúbico por el número de cifras que tratamos, es decir, $\mathcal{O}(|\mathcal{N}|^4)$. También lo podemos hacer en tiempo cuadrático con un precálculo cuadrático usando el algoritmo de la Sección 5.4, lo que nos da un coste global cúbico. Esto se refleja en la siguiente proposición:

Proposición 5.9. *Sea $G = \langle \mathcal{N}, \Sigma, R \rangle$ una SCFG, y sean X e Y no terminales de $\mathcal{N}$ que generan palabras diferentes. Entonces podemos calcular la primera diferencia entre w_X y w_Y en tiempo $\mathcal{O}(|\mathcal{N}|^4)$ usando un algoritmo determinista, o bien en tiempo $\mathcal{O}(|\mathcal{N}|^3)$ usando un algoritmo probabilístico que falla menos de la mitad de las veces.*

Capítulo 6
Compresión de términos

En el Capítulo 5 hemos explicado una manera de comprimir palabras utilizando un cierto tipo de gramáticas incontextuales llamado SCFG. En esta sección explicaremos el mismo principio pero aplicado a términos. De la misma manera aplicaremos los algoritmos del Capítulo 3 a los términos comprimidos de esta forma, mediante una variante que nos garantizará que el coste total será relativo al tamaño del término comprimido, y no al tamaño del término antes de descomprimir.

6.1. Singleton Tree Grammar

Las Singleton Tree Grammar (STG) utilizan el mismo principio de aprovechar las repeticiones que las SCFG, pero en este caso para comprimir un término. La principal diferencia con respecto a las SCFG, además de que generarán términos en lugar de palabras, es que no son gramáticas incontextuales. Es decir, el lenguaje que genera un símbolo no terminal puede depender del contexto en el que están. El motivo de usar este tipo de gramática es porque con gramáticas incontextuales no se podrían aprovechar las repeticiones de ciertos términos muy altos pero estrechos.

Más formalmente, una Tree Grammar (TG) es una tupla $\langle \mathcal{N}, \mathcal{F}, R \rangle$, donde $\mathcal{N}$ es un alfabeto finito con rango de símbolos no terminales, $\mathcal{F}$ es un alfabeto con rango, y R es un conjunto finito de reglas de la forma $N(y_1 \dots y_n) \to \alpha$, donde $N \in \mathcal{N}$, N tiene rango n, y α es de la forma, o bien $f(y_1, \dots, y_n)$, donde $f \in \mathcal{F}$ tiene aridad n, o bien $M(y_1, \dots, y_i, K(y_{i+1}, \dots, y_j), y_{j+1}, \dots y_n)$, donde $M, K \in \mathcal{N}$, $\texttt{rank}(K) = j - i$ y $\texttt{rank}(K) + \texttt{rank}(M) = n + 1$.

Una Singleton Tree Gramar (STG) es una TG en la que cada no terminal produce un único término. En particular, si $N \in \mathcal{N}$ y N tiene rango n, entonces el término generado por N, que denotaremos t_N, será un término sobre $\mathcal{F} \cup \{y_1, \dots, y_n\}$, donde cada y_i con $1 \leq i \leq n$ tiene aridad 0 y aparecerá exactamente una vez.

Como en el caso de SCFG, para garantizar que cada no terminal de un STG genere exactamente un término impondremos la condición de no recursividad. Eso quiere

decir que existe una numeración de símbolos terminales, $\{N_1, \ldots, N_k\}$ tal que si $(N_p(y_1, \ldots, y_n) \to N_q(y_1, \ldots, y_i, N_r(y_{i+1}, \ldots, y_j), y_{j+1}, \ldots y_n)) \in R$ entonces $p > q, r$.

Definimos el término generado por un no terminal de forma recursiva usando sustituciones. Si $(N \to f(y_1, \ldots, y_n)) \in R$, y además se cumple que $f \in \mathcal{F}$, entonces definimos $t_N = f(y_1, \ldots, y_n)$. Si en cambio $N(y_1, \ldots, y_n) \to M(y_1, \ldots, y_i, K(y_{i+1}, \ldots, y_j), y_{j+1}, \ldots y_n) \in R$ entonces $t_N = \sigma(t_M)$, donde $\sigma = \{y_{i+1} \to t_K\}$, con la adecuada reordenación de los nombres de y_i, de forma que si $i < j$ entonces y_i aparece antes de y_j en el recorrido en preorden de t_N.

El siguiente ejemplo ilustra cómo generan términos los no terminales de una STG.

Ejemplo 6.1. *Consideramos la siguiente STG* $G = \langle \mathcal{N}, \mathcal{F}, R \rangle$*, donde* $\mathcal{N} = \{N_0 : 0, N_1 : 2, N_2 : 1, N_3 : 0, N_4 : 1, N_5 : 0\}$*,* $\mathcal{F} = \{f : 2, a : 0\}$*, y el conjunto de reglas es el siguiente:*

- $N_5() \to N_4(X_3())$
- $N_4(y_1) \to N_1(X_3(), y_1)$
- $N_3() \to N_2(X_0())$
- $N_2(y_1) \to N_1(X_0(), y_1)$
- $N_1(y_1, y_2) \to f(y_1, y_2)$
- $N_0() \to a$

Cada N_i *genera un término sobre el alfabeto* $\mathcal{F} \cup \{y_1, \ldots, y_{n_i}\}$*, donde* n_i *es el rango de* N_i*. En este ejemplo los términos generados por los diferentes no terminales son:*

- $t_{N_5} = f(f(a, a), f(a, a))$
- $t_{N_4} = f(f(a, a), y_1)$
- $t_{N_3} = f(a, a)$
- $t_{N_2} = f(a, y_1)$
- $t_{N_1} = f(y_1, y_2)$.
- $t_{N_0} = a$.

Como en el caso de SCFG, una STG puede generar un término de tamaño exponencial respecto al número de no terminales, como se puede apreciar en el ejemplo siguiente:

Ejemplo 6.2. *Definimos la familia de STG* $\{G_n\}_n$ *de la siguiente forma:* $G_n = \langle \mathcal{N}_n, \mathcal{F}, R_n \rangle$*, donde* $\mathcal{F} = \{f : 1, a : 0\}$*,* $N_n = \{N_a, N_0, N_1, \ldots, N_n, N_f\}$*, y* R_n *es el siguiente conjunto de reglas:*

- $N_f() \to N_n(N_a())$
- $N_n(y_1) \to N_{n-1}(N_{n-1}(y_1))$
- $N_{n-1}(y_1) \to N_{n-2}(N_{n-2}(y_1))$
- $\ldots$
- $N_1(y_1) \to N_0(N_0(y_1))$

```
Unificación( G : TG, X : no terminal, Y: no terminal): booleano
// Precondición: G = ⟨N, F, R⟩ es una STG y X, Y ∈ N
    σ : sustitución
    p : posición
    σ := ∅
    Mientras σ(t_X) ≠ σ(t_Y), Hacer
        p := primera posición en preorden en la que σ(t_X) y σ(t_Y) difieren
        Si σ(t_X)(p) es un símbolo de función, y σ(t_Y)(p) es un símbolo de función diferente, Entonces
            Devuelve falso
        // Supondremos sin pérdida de generalidad que σ(t_X)(p) es un símbolo de variable x
        Si x aparece en alguna posición de σ(t_Y)|_p, Entonces
            Devuelve falso
        Z es un nuevo no terminal de N y t_Z = σ(t_Y)|_p
        σ := {x ↦ t_Z} ∘ σ
    Devuelve cierto
```

FIGURA 6.1. Esquema general del algoritmo que decide si dos no terminales de una STG unifican

- $N_0(y_1) \to f(y_1)$
- $N_a() \to a()$

Podemos comprobar que G_n tiene $n+3$ reglas, y sin embargo el término t_{N_f} tiene $2^n + 1$ símbolos.

6.2. Unificación sobre STG

En el Capítulo 3 hemos explicado los algoritmos más conocidos para resolver el problema de unificación sobre términos, pero nos interesa resolver este problema en STG. Este tema se ha tratado en artículos anteriores. En particular en [**GGSS09**] se explica una forma de implementar la unificación de términos generados por dos no terminales de una STG. Un resumen esquemático del algoritmo es el de la Figura 6.1.

A continuación hacemos un breve resumen de los cálculos previos que tenemos que realizar para poder ejecutar todo el algoritmo adecuadamente.

6.2.1. Decisión de igualdad. Podemos ver que nuestro bucle acabará cuando decidamos que dos términos son iguales. Sin embargo, para decidir la igualdad de los términos generados por dos no terminales X e Y no podemos usar el algoritmo trivial de descomprimir los términos y comprobar si son iguales, ya que como hemos visto los términos pueden tener un tamaño exponencial con respecto al número de no terminales de la gramática.

En su lugar utilizaremos el Lema 6.3, de demostración trivial, que reducirá el problema sobre términos a un problema sobre palabras.

Lema 6.3. *Sean s y t términos sobre un mismo alfabeto con rango $\mathcal{F}$. Entonces* $s = t \Leftrightarrow Pre(s) = Pre(t)$

Nótese que este lema no es cierto si el alfabeto con rango permite varios rangos para un mismo símbolo, como se muestra en el siguiente ejemplo.

Ejemplo 6.4. *Supongamos que un alfabeto $\mathcal{F}$ permite que los símbolos de función f y g tengan aridad 1 o 2, y que el símbolo a sea una constante. Entonces, si consideramos los términos sobre $\mathcal{F}$, $t = f(g(a), a)$ y $s = f(g(a, a))$ tenemos que $t \neq s$, pero $Pre(t) = fgaa = Pre(s)$.*

En este PFC consideramos siempre alfabetos con rango fijo, por lo que no tendremos este problema. Es decir, hemos reducido el problema de decidir si dos términos son iguales al problema de decidir si dos palabras (los recorridos en preorden) son iguales.

Sin embargo esto no soluciona el principal problema que tenemos, ya que igualmente tenemos que descomprimir ambos términos para decidir si su recorrido en preorden es el mismo o no. Para solucionar esto utilizaremos la siguiente proposición.

Proposición 6.5. *Sea $G = \langle \mathcal{N}, \mathcal{F}, R \rangle$ una STG. Entonces se puede computar en tiempo $\mathcal{O}(\mathcal{N})$ una SCFG $H = \langle \mathcal{N}', \Sigma, R' \rangle$ tal que $\mathcal{N}'$ contiene todos los no terminales de $\mathcal{N}$ de rango 0, $|\mathcal{N}'| \in \mathcal{O}(|\mathcal{N}|)$, Σ es el conjunto de símbolos de $\mathcal{F}$, y para todo $N \in \mathcal{N}$, $Pre(t_N)$ en G se corresponde con w_N en H.*

Esta proposición básicamente afirma que dada una STG G podemos calcular una SCFG que represente los recorridos en preorden de los términos generados por los no terminales de G.

Se puede encontrar una demostración constructiva de este hecho en [**BLM08**]. Básicamente la SCFG resultante se obtiene creando un no terminal para el recorrido en preorden de cada término entre dos parámetros. Si un término no tiene parámetros entonces habrá un no terminal de la SCFG que genere su recorrido en preorden.

Una vez hayamos generado la SCFG resultante podemos aplicar los algoritmos presentados en la Sección 5.2 y en la Sección 5.3 para decidir si dos términos son iguales o no de forma determinista o probabilista, según el coste que deseemos obtener.

6.2.2. Primera diferencia. Lo siguiente que tenemos que calcular es la primera posición del recorrido en preorden en la que difieren. El principal motivo por el que calculamos la posición como índice del recorrido en preorden y no como posición absoluta es porque una posición puede tener un tamaño del orden de la altura del término descomprimido, y hemos visto que este tamaño puede llegar a tener altura exponencial con respecto al tamaño de la gramática.

Para calcular la primera diferencia el procedimiento que seguiremos es simple. En primer lugar calculamos la SCFG que genera los recorridos en preorden de los no terminales de G con rango 0, utilizando la Proposición 6.5. Y en segundo lugar

utilizaremos los algoritmos de la Sección 5.2 y la Sección 5.3 para determinar la primera posición en la que los recorridos en preorden difieren. De nuevo se pueden ver más detalles en [**GGSS09**].

6.2.3. Existencia de símbolos en términos. Otra cosa que necesitamos saber es, dada una STG $G = \langle \mathcal{N}, \mathcal{F}, R \rangle$, un símbolo $x \in \mathcal{F}$, y un no terminal $N \in \mathcal{N}$, decidir si x aparece en alguna posición de t_N. El algoritmo trivial consistiría en descomprimir t_N y comprobar si x aparece, pero esto podría tener coste exponencial y no nos interesa.

Para solucionarlo utilizaremos programación dinámica. Para eso calcularemos de forma recursiva los no terminales en los que x aparece en alguna posición del término que generan. Esto se puede hacer con un solo recorrido en tiempo lineal.

6.2.4. Generación de nuevo no terminal. El último paso que falta por realizar es, dada una posición p del recorrido en preorden, añadir un no terminal a la gramática que genere el subtérmino en la posición p del término generado por otro no terminal. El objetivo de este cálculo es poder realizar fácilmente la asignación de una varaible a un subtérmino, ya que podría darse el caso de que ningún no terminal generara el subtérmino de un cierto término en una cierta posición, como se puede ver en el siguiente ejemplo:

Ejemplo 6.6. *Consideramos la siguiente STG* $G = \langle \mathcal{N}, \mathcal{F}, R \rangle$*, donde* $\mathcal{N} = \{N_a : 0, N_f : 1, N_1 : 1, N_2 : 1, N_3 : 0\}$*,* $\mathcal{F} = \{f : 1, a : 0\}$*, y el conjunto de reglas es el siguiente:*

- $N_3() \to N_2(N_a())$
- $N_2(y_1) \to N_1(N_1(y_1))$
- $N_1(y_1) \to N_f(N_f(y_1))$
- $N_f(y_1) \to f(y_1)$
- $N_a() \to a()$

Podemos ver que los términos generados por cada no terminal son los siguiente:

- $t_{N_3} = f(f(f(f(a))))$
- $t_{N_2} = f(f(f(f(y_1))))$
- $t_{N_1} = f(f(y_1))$
- $t_{N_f} = f(y_1)$
- $t_{N_a} = a$

En particular, el subtérmino de t_{N_3} *en la posición en preorden* 2 *es* $t_{N_3}|_1 = f(f(f(a)))$*, y se puede ver que ningún no terminal genera dicho término.*

La generación de este no terminal se hace recursivamente, y se puede encontrar una explicación detallada de cómo se hace exactamente en [**GGSS09**].

El coste global del algoritmo dependerá de cómo busquemos la primera diferencia entre los recorridos en preorden, como se explica en la Sección 6.2.2. Puede variar

entre coste $\mathcal{O}(|V||G|^4)$ usando un algoritmo determinista y $\mathcal{O}(|V||G|^3)$ usando un algoritmo no determinista.

Capítulo 7
Implementación

En este capítulo explicaremos los detalles de la implementación de los algoritmos anteriormente descritos. Además diferenciaremos entre aquellas partes que ya se encontraban implementadas, aquellas que ha habido que modificar o rehacer, y aquellas que he implementado de cero.

El código completo se puede encontrar en la página web http://racso.lsi.upc.edu/unif/unif.php, donde además se puede ejecutar y comprobar que el resultado es correcto.

La implementación está realizada en C++, que es un lenguaje orientado a objetos, pero a la vez eficiente al ser de un nivel relativamente bajo. Aprovechando la orientabilidad a objetos hemos dividido el código en diferentes clases para mayor modularidad. Dichas clases se detallan a continuación:

7.1. SCFG

La clase SCFG nos permite trabajar con Singleton Context Free Grammars, como las que se explican en la Sección 5.1. Un elemento de la clase SCFG es una Gramática Incontextual, con una información asociada: qué algoritmo debe utilizar para decidir igualdad entre palabras, ya sea el algoritmo determinista de la Sección 5.2, o los algoritmos probabilistas de la Sección 5.3. Además, en este ultimo caso, se guarda información sobre la cota de la probabilidad de error deseada, que se puede asignar dinámicamente en tiempo de ejecución.

Las reglas están implementadas mediante un vector del estándar de C++. Se pueden añadir reglas mediante la operación `addrule`, que realiza una operación de `push_back`, que en un vector puede tener coste lineal. El motivo de usar vector y no otras estructuras de datos que evitan el coste lineal de la inserción es porque tiene un mejor coste en las operaciones más comunes que sus principales alternativas:

- Una lista de C++ permitiría la inserción en tiempo constante, lo que mejoraría sustancialmente el proceso de creación de las reglas de una SCFG. Sin embargo las listas en general no permiten acceso aleatorio a sus posiciones,

lo que aumentaría el coste de generar una palabra, o en general de realizar cualquier algoritmo, por lo que esta opción no es viable.
- Una estructura en forma de árbol balanceado, como los sets y maps de C++, tiene coste logarítmico en todas sus operaciones. Esto quiere decir que nos permitiría crear una SCFG en tiempo $\mathcal{O}(n \log n)$, siendo n el número de reglas, y además puede realizar todas las operaciones de un vector incrementando el tiempo en un factor logarítmico. Sin embargo, como la creación de una SCFG sólo se realiza una vez hemos preferido pagar este coste extra en la creación a cambio de poder realizar los algoritmos sin incrementar su coste.

En general una SCFG requiere realizar ciertos precálculos para realizar cualquier algoritmo, ya sea determinista (que requiere calcular la APTable, como se aprecia en la Sección 5.2), o probabilista (que requiere calcular el valor de cada palabra, como puede ver en la Sección 5.3). En el interfaz anterior había que hacer sendas llamadas a las funciones `do_precomputations` y `substring` para poder calcular si una palabra era substring de otra. En la nueva implementación sólo es necesario llamar a las operaciones que se quieran realizar, que comprobarán si los precálculos necesarios se han hecho ya, y de no ser así los harán. De esta forma se simplifica el interfaz sin comprometer la eficiencia.

Las principales funciones de la clase SCFG son las que permiten leer y escribir una SCFG desde entrada y salida estándar, las consultoras que permiten ver cosas como el símbolo de una palabra en una posición concreta, las modificadoras que permiten elegir el tipo de algoritmo utilizado (determinista, probabilístico con primo y probabilístico sin primo) y las funciones que ejecutan los algoritmos de subcadena y primera diferencia.

En cuanto a mis aportaciones a esta clase, he implementado completamente los algoritmos probabilísticos, además de modificar la implementación interna y la interfaz de la clase para aumentar la eficiencia y simplicidad. También he implementado las nuevas operaciones para calcular el recorrido en preorden de un término, para adaptarlas a la nueva herramienta de compresión usada.

7.2. STG

La clase STG nos permite trabajar con Singleton Tree Grammars, como las que se explican en la Sección 6.1. Un elemento de la clase STG es una Tree Grammar, representada como un conjunto de reglas. En este caso también utilizaremos vectores de C++ para guardar las reglas, por los mismos motivos que se explican en la Sección 7.1.

En general, la clase STG ha habido que rehacerla casi por completo, para poder adaptarla al nuevo compresor, que usa reglas en Forma Normal de Chomsky. Se ha modificado así la implementación (que usa 3 tipos de regla, en lugar de las 7 que había antes), y todos los algoritmos que se utilizaban, como el de unificación o el de generar preorden.

Mi trabajo ha consistido sobre todo en modificar todas las operaciones para adaptarlas al nuevo formato. Algunas como la de unificación, que incluye operaciones como generar recorrido en preorden, ha habido que rehacerlas casi por completo, al no poder aprovecharse nada de la implementación usada para el formato anterior. La operación de matching era una novedad de este PFC y por tanto la he implementado de cero.

7.3. ARPR

La clase ARPR permite trabajar con progresiones aritméticas. Esta clase es la base de la APTable descrita en la Sección 5.2. Sin embargo, la clase ya estaba implementada y funcionaba correctamente, así que mi trabajo únicamente ha consistido en arreglar algunos errores de inicialización de variables que no habían dado problemas en los test utilizados hasta el momento.

7.4. BigInt

La clase BigInt permite trabajar con enteros de precisión arbitraria. En estos momentos esta clase está desactivada, y sólo se pueden usar enteros de hasta 64 bits (clase `long long` de C++). Esta es la principal limitación de la implementación actual, ya que sólo permite trabajar con palabras de tamaño hasta $2^{31} - 1$, pero además, en el caso de los algoritmos probabilísticos, sólo se puede garantizar la cota máxima del error con palabras de tamaño hasta 61 millones en el caso del algoritmo que usa primos, y de 7000 en el caso del algoritmo que no usa primo.

El principal motivo de la desactivación de la clase BigInt es que introduce un factor constante considerable a la hora de realizar cualquier operación. En los experimentos pudimos comprobar que el tiempo de ejecución usando palabras de tamaño menor que $2^{31} - 1$ se incrementaba en hasta 20 veces más si se utilizaba la clase BigInt. Por otro lado, mientras que teóricamente el uso de enteros de 64 bits podría hacer que el error fuera mayor del 0.5 exigido, en la práctica no hemos encontrado un solo caso que falle.

En cualquier caso la implementación con BigInts está realizada, y sólo requiere cambiar una definición (dos líneas de un archivo) para poder usarse si se desea.

7.5. Term

La clase term permite trabajar con términos sin comprimir. El principal motivo de hacer una clase así es poder comparar la eficiencia de los algoritmos aplicados a términos comprimidos, respecto a los mismos algoritmos aplicados a términos sin comprimir.

La clase permite trabajar con términos de manera muy intuitiva. En particular puede crear términos recursivamente a partir de un símbolo de raíz y n términos. También puede crear términos aleatorios a partir de un tamaño y un alfabeto con rango. Todo esto nos permite crear todo tipo de términos, como términos mixtos en los que una parte está generada aleatoriamente, y la otra tiene forma de árbol completo, por ejemplo. De esta forma podemos generar todos los problemas de unificación que se nos ocurra y comprobar así el tiempo.

También se han implementado dos formas de leer y escribir términos:

- La notación clásica $f(g(a), a)$, que da una forma intuitiva de ver un término y que permite introducir por la entrada estándar términos pequeños de forma rápida y sencilla
- La notación XML, $< f >< g >< a/ >< /g >< a/ >< /f >$, que aunque es más dificil de leer y escribir, es más estándar. Además es la codificación que usa el nuevo compresor utilizado, de forma que es necesaria su utilización para poder realizar las comparaciones entre términos comprimidos y descomprimidos.

Además se han implementado los algoritmos de unificación y matching explicados en el Capítulo 3, para términos no comprimidos. Para este fin se ha utilizado una implementación del algoritmo de Robinson usando DAGs (Grafos Acíclicos Dirigidos), que evitan el coste exponencial que puede darse en algunos casos.

Esta clase también es una novedad del PFC, y por tanto también la he implementado en su totalidad, así como diversos problemas de unificación generados con esta herramienta para comprobar el tiempo de ejecución.

Capítulo 8
Experimentos

Una vez implementados ambos algoritmos, el último paso es ejecutarlos y comprobar en qué casos es mejor aplicar uno, y en qué casos es mejor aplicar el otro. Para eso hemos generado una batería de casos de prueba, algunos más compresibles, y otros menos, para poder comprobar la diferencia.

8.1. Datos del experimento

Todos los tests se ejecutan en un ordenador con Intel Xeon Core 2 Duo, procesador de 3 Ghz, con 4GM de RAM. Usamos la distribución 9.10 de Ubuntu (Linux), con kernel 2.6.32 y de 64 bits. Nuestra implementación se compiló con g++ 4.4.1.

El compresor utilizado fue TreeRePair, con fecha 19 de enero de 2011. El código está disponible en `http://code.google.com/p/treerepair`.

8.1.1. Protocolo. Ejecutamos cada test tres veces, y seleccionamos el tiempo más rápido entre ellos. Sólo tenemos en cuenta el tiempo que tarda el algoritmo de unificación en sí, ignorando cosas como la lectura de términos o la inicialización.

8.1.2. Diseño de los experimentos. Los tests de problemas de unificación que hemos usado no son problemas reales que puedan aparecer en la práctica, sino ejemplos artificiales que puedan probar los casos límite de los dos tipos de algoritmos. Los principales aspectos que tuvimos en cuenta a la hora de hacer el diseño fueron:

- Términos muy o muy poco compresibles por TreeRePair
- Términos que unifican o que no unifican
- Términos con muchas o con pocas variables

Teniendo en cuenta estos criterios hemos construido una familia de instancias que nos permiten evaluar el comportamiento de los algoritmos en varios de estos aspectos.

8.2. Resultados

A continuación mostraremos tablas comparativas de los tests que hemos ejecutado en los algoritmos explicados en las secciones anteriores.

8.2.1. Bind y Mon. En primer lugar mostraremos un par de ejemplos sencillos que tienen una muy buena compresión.

Sea n un natural. Definiremos mediante $f^n(a,b)$ el árbol binario completo sobre el alfabeto $\mathcal{F} = \{f : 2, a : 0, b : 0\}$. Es decir, $f^1(a,b) = f(a,b)$, y $f^n(a,b) = f(f^{n-1}(a,b), f^{n-1}(a,b))$ si $n \geq 2$. A partir de ahí definimos Bin(n) como el siguiente problema de unificación sobre los alfabetos $\mathcal{F} = \{f : 2, g : 1, a : 0, b : 0\}$ y $\mathcal{X} = \{x, y\}$:

$$\text{Bin}(n) = g(g(f^n(a,b), f^n(a,b)), g(f^n(a,b), f^n(a,b)) \doteq g(g(x,x), g(x, f^n(a,y)))).$$

Análogamente, definimos $f^n(a)$ como el árbol monádico de altura n con nodos internos etiquetados con f, y a como única hoja. Es decir, $f^0(a) = a$ y $f^n(a) = f(f^{n-1}(a))$. A partir de ahí definimos Mon(n) como el siguiente problema de unificación sobre los alfabetos $\mathcal{F} = \{f : 1, h : 3\}$ y $\mathcal{X} = \{x, y, z, t\}$:

$$\text{Mon}(n) = (h(f^n(x), f^n(y), y) \doteq h(t, z, t))$$

Es fácil de ver que tanto Bin como Mon son problemas unificables para cualquier valor de n. Además, el ratio de compresión con TreeRePair es muy bueno, ya que como se puede ver en el Ejemplo 6.2, el tamaño de una STG que genere $f^n(a)$ es logarítmico respecto de n.

En la Tabla 8.1 y la Tabla 8.2 se incluyen los tiempos de ejecución de cada uno de los 4 algoritmos (sin comprimir, comprimido determinista, comprimido probabilístico con primo, y comprimido probabilístico sin primo), para cada uno de los dos problemas, Mon y Bin, con diferentes valores de n.

	Tiemo de ejecución (en ms)				Input			
n/1000	tUnif	STG-randp	STG-rand	STG-exacto	nodos	tiempo de compresión	nodos de la STG	archivo en CNF
5	**2**	8	7	24	10008 (69K)	55ms	38 (388B)	1K
10	**5**	10	8	28	20008 (137K)	62ms	40 (398B)	1.1K
20	11	11	**9**	30	40008 (157K)	140ms	42 (420B)	1.1K
50	44	11	**9**	30	100k (684K)	341ms	45 (434B)	1.2K
100	107	12	**10**	31	200k (1.4M)	681ms	47 (457B)	1.3K
200	232	13	**10**	32	400k (2.7M)	1387ms	49 (467B)	1.3K

TABLA 8.1. El ejemplo Mon(n)

Las otras columnas de la tabla representan los siguientes datos sobre la compresión:

	Tiemo de ejecución (en ms)				Input		
n	tUnif	STG-rp	STG-r	STG-e	nodos	nodos de la STG	archivo en CNF
10	**3**	18	18	78	20484 (111K)	62 (578B)	1.9K
11	**7**	20	20	96	40964 (221K)	66 (596B)	2.1K
12	**16**	22	21	108	81924 (441K)	70 (638B)	2.2K
13	35	23	**22**	131	163844 (881K)	74 (656B)	2.3K
14	72	26	**25**	146	327684 (1.8M)	78 (698B)	2.4K
16	290	30	**28**	(*)	1310724 (6.9M)	86 (758B)	2.7K

(*) STG-exacto excedió los límites de la precisión de un `int`

TABLA 8.2. El ejemplo Bin(n)

- nodos: El número de nodos del término original sobre el que realizamos el problema de unificación.
- tiempo de compresión: El tiempo que tarda TreeRePair en comprimir el término.
- nodos de la STG: el número de reglas que tiene la STG que representa el término comprimido.
- archivo en CNF: tamaño del archivo que contiene la STG en forma CNF, que es el tipo especial de STG sobre el que realizamos nuestros algoritmos.

En ambos casos se puede observar el mismo patrón. Para valores pequeños de n, la compresión no es lo suficientemente buena como para compensar el hecho de que el algoritmo sea más lento. Sin embargo, según va avanzando la n, el ratio de compresión es cada vez mayor (recordemos que es exponencial respecto a n), y por tanto los algoritmos sobre términos comprimidos acaban siendo mucho más rápidos.

Además se puede observar que los algoritmos probabilísticos siempre son más rápidos que los deterministas. En efecto, en los capítulos anteriores vimos que el coste del algoritmo determinista es $\mathcal{O}(n^4)$ en caso peor, frente al $\mathcal{O}(n^3)$ de los algoritmos probabilísticos. Hay también una pequeña diferencia entre el algoritmo que usa primos y el que no, que consiste básicamente en el tiempo de generación del número primo.

Cabe ver también que el tiempo de compresión es en cualquier caso considerablemente mayor que cualquier tipo de unificación, lo que lleva a pensar que sólo es aconsejable realizar la unificación a términos comprimidos cuando estos ya están directamente guardados como STG, y que no es aconsejable comprimir términos únicamente para realizar la unificación.

8.2.2. Peor caso para las STG. En la sección anterior hemos visto casos en los que la unificación se comportaba mejor con términos comprimidos que sin comprimir. Veremos el caso opuesto, en el que la unificación se comporta peor mediante STG que directamente sobre términos sin comprimir.

Es evidente que si los términos no son compresibles la unificación funcionará peor, como veremos en la siguiente sección. Pero hay una razón mucho más simple para que esto ocurra. Consideremos el problema de unificación $f(t) \doteq g(t')$, para términos cualquiequiera t y t', suficientemente grandes. El tiempo de ejecución del

algoritmo de unificación sobre términos, es de tan solo 0,005ms para este ejemplo. En cambio, incluso para términos muy compresibles $t = t'$, la unificación basada en STG no baja de los 15ms.

El motivo de esta diferencia es que la unificación basada en STG siempre necesitará, como mínimo, encontrar el símbolo raíz, y para ello puede necesitar recorrer toda la gramática, mientras que la unificación sobre términos detectará instantáneamente que la raíz no coincide, y que por tanto el problema no unifica.

8.2.3. Meta. Vamos a definir un ejemplo en el que haya una parte más fácilmente compresible, y otra más difícilmente compresible, pero que además nos permita configurar la proporción exacta de cada parte, para poder comprobar el resultado en diversos casos.

Consideraremos el problema de unificación $t_1 \doteq t_2$, donde t_1 y t_2 son árboles binarios completos de altura n. En las hojas de t_1 y t_2 incluiremos términos monádicos de alturas arbitrarias h comprendidas entre dos valores $minH \leq h \leq maxH$. Estos términos monádicos son idénticos en t_1 y t_2. A continuación, en las hojas de dichos términos t_1 contendrá variables, aleatoriamente escogidas de un cierto conjunto $\mathcal{X}$, mientras que t_2 contendrá términos aleatorios de tamaño hasta $randT$ elegidos de entre todos los términos de $\mathcal{T}(\mathcal{F})$, para un cierto valor de $\mathcal{F}$.

Además, podemos forzar que los árboles de t_2 correspondientes a las mismas variables en t_1 sean iguales (y consecuentemente el problema unificará), o bien que no lo sean (y el problema no unificará). Esto lo decidiremos con un cierto valor booleano U, de forma que quede definido un problema de unificación Meta como: Meta$(n, minH, maxH, \mathcal{X}, randT, \mathcal{F}, U)$. A partir de aquí experimentamos variando los diferentes parámetros.

8.2.3.1. Número de variables. En primer lugar experimentamos con el número de variables diferentes. Los resultados no mostraron ningún tipo de correlación entre el número de variables y la diferencia entre tiempos de ejecución; a ambos algoritmos les afectaba igual. Por ejemplo, para $n = 4$, $maxH = minH = 1000$, $randT = 1$ y $|\mathcal{F}| = 3$ obtenemos, para 3 variables: 3ms/18ms (tUnif/STG-rand), para 5 variables: 7ms/32ms, y para 10 variables: 10ms/44ms.

8.2.3.2. Términos incompresibles. Un caso interesante a tener en cuenta es aquel en el que los términos son incompresibles. Para obtener uno de esos términos a partir de Meta basta con dar un valor mucho más grande al parámetro $randT$ que al resto de parámetros. Por ejemplo, consideramos $n = 1$, $minH = maxH = 0$, $\mathcal{X} = \{x, y\}$, $\mathcal{F} = \{g : 2, f : 1, a : 0\}$, y $U =$ cierto, para forzar la unifiación.

Como se puede ver en la Tabla 8.3, los algoritmos basados en STG son más de 100 veces más lentos que los algoritmos sobre términos. La diferencia de velocidades parece ser algo menor para inputs más grandes.

Si, en cambio, añadimos un árbol binario más grande en la cima de t_1 y t_2, es decir, aumentamos la n, entonces el término se puede comprimir mejor y por tanto la unificación basada en STG pasa a ser eficiente. Esto se puede ver en la Figura 8.1, donde seleccionamos un valor $randT = 20000$, pero utilizamos términos monádicos de tamaño entre 0 y 1000.

	Tiempo de ejecución (en ms)				Input		
randT	tUnif	STG-rp	STG-r	STG-e	nodos	nodos STG	CNF
1000	**0.1**	21	19	222	981 (5.9K)	214 (1.5K)	6.6K
5000	**0.6**	78	73	2430	4778 (29K)	774 (15K)	25K
20000	**2.4**	405	388	43632	26114 (157K)	3308 (21K)	107K
50000	**12**	1396	1143	(*)	94280 (564K)	9975 (63K)	327K
200000	**43**	5464	4980	(*)	334586 (2M)	30740 (196K)	1.1M

(*) STG-exacto excedió la memoria disponible

TABLA 8.3. Términos incompresibles en las posiciones de sustitución

Con respecto a la *unificabilidad*, si cambiamos unos pocos nodos para hacer el input no unificable, obtenemos que los algoritmos basados en STG tardan casi el doble del tiempo que aparece en la Tabla 8.3, mientras que tUnif va ligeramente más rápido.

8.2.3.3. 3-stack. También hay ejemplos en los que la solución consiste en árboles más profundos que los de la entrada. Esto también funciona mejor para términos comprimidos, ya que la solución también se da comprimida, con un factor de compresión del orden del tamaño de la entrada.

Consideremos $t_1 = h(x, y, z)$ y $t_2 = g(s_1, s_2, s_3)$, donde s_1 es el árbol binario completo $f^n(y, y)$, s_2 es el árbol binario completo $f^n(z, z)$, y s_3 el árbol binario completo $f^n(x, x)$.

El problema de unificación $t_1 \doteq t_2$ no tiene solución, ya que x se reemplazaría por un término que contiene y, y se reemplazaría por un término que contiene z, y z se reemplazaría por un término que contiene x, lo cual no es posible.

Hemos llamado a este ejemplo "3-Stack" y los tiempos de ejecución se muestran en la Figura 8.1.

n	tUnif	STG-rp	STG-r	STG-e
18	99	**7**	**7**	35
19	200	8	**7**	38
20	401	**8**	**8**	(*)

n	tUnif	STG-rp	STG-r	STG-e
7	**369**	1694	1437	(*)
8	**688**	1726	1512	(*)
9	**1730**	2391	2092	(*)

(*) STG-exacto excedió la memoria disponible

FIGURA 8.1. El ejemplo 3-Stack (left) y randT=20000 para la Tabla 8.3 (derecha)

Capítulo 9
Análisis económico

En esta sección haré un análisis del impacto económico que habría podido tener este proyecto si se hubiera realizado en una empresa.

9.1. Hardware

Toda mi parte del trabajo se ha realizado en un único ordenador, que tiene un coste de 600 euros. Cabe destacar que no ha sido el único propósito del ordenador, pero a efectos de cálculo lo consideraremos así.

9.2. Software

Todo el software que hemos utilizado es gratuito. En particular hemos necesitado:

- Ubuntu 10.10 como sistema operativo.
- El compilador g++.
- El editor de texto gedit.
- El editor de LaTeX Texmaker.
- La herramienta TreeRePair, disponible en Open Source.

Además hemos utilizado una página web para poder ejecutar el código, pero el dominio está alojado en los servidores de la UPC, por lo que el coste ha sido nulo.

9.3. Recursos humanos

Suponiendo que este proyecto se hubiera desarrollado en una empresa, habría que tener en cuenta el sueldo de los diferentes participantes de este proyecto. Hemos estimado que habría un jefe de proyecto, un diseñador, dos programadores y un supervisor. Desglosamos el coste de cada uno según horas de trabajo:

- El jefe de proyecto se encargaría de coordinar al grupo y comprobar que todo va según los plazos previstos. Hemos estimado unas 10 horas de este trabajo. Además se encargaría de realizar la memoria, para lo cual hemos estimado un

tiempo de 100 horas. En total 110 horas, que a un salario de unos 40 euros por hora darían un total de 4400 euros.

- El diseñador se encargaría de recopilar información sobre todos los algoritmos que hay que implementar, y a continuación diseñar la distribución de clases adecuada para que los algoritmos funcionen. Como este trabajo incluye entre otras cosas una cierta tarea de investigación sobre artículos ya existentes, hemos estimado que realizaría un total de 50 horas. A un salario de unos 30 euros por hora da un total de 1500 euros.
- Los programadores se encargarían de implementar los algoritmos que el diseñador ha creado. Hemos estimado que entre los dos necesitarían un total de 100 horas para su implementación, que a un salario de 20 euros por hora da un total de 2000 euros.
- El supervisor se encargaría de coordinar a los programadores comprobando que los dos realizan la parte que les toca, que puede hacer en unas 10 horas. Además le hemos asignado el diseño y ejecución de diversos casos de prueba, a lo que dedicaría un total de 50 horas, dando un total de 60. A un salario de 25 euros por hora daría un total de 1500 euros.

Es decir, el coste total del proyecto en recursos humanos sería de, aproximadamente, 9400 euros. Si sumamos los 600 euros de la parte de Hardware necesario obtenemos el coste real del proyecto, que sería de 10000 euros.

Capítulo 10
Conclusiones

En este capítulo explicaré los principales resultados obtenidos en este PFC, de la misma forma que indicaré qué cosas he aprendido haciéndolo, y qué conocimientos de la carrera he utilizado para su desarrollo. Por último expondré posibles ampliaciones futuras de este trabajo.

10.1. Resultados

En este PFC he implementado dos variantes de un algoritmo que aún no se había implementado, aunque se definía en [**SSS10**]. También he hecho una modificación que nos permite aplicarlo a nuestro problema de unificación. Además de eso hemos hecho diversas mejoras en un código ya existente que también resolvía el problema de unificación sobre términos comprimidos, pero utilizando un algoritmo determinista. También he implementado una clase que resolvía el problema de unificación sobre términos no comprimidos, aunque dicho problema ya había sido tratado anteriormente y probablemente ya se había implementado.

Por otro lado he diseñado diversos tests que permitían comprobar la eficacia de uno y otro algoritmo. Eso me ha permitido llegar a la conclusión de que los nuevos algoritmos probabilísticos mejoran en todos los casos la eficiencia de los algoritmos deterministas ya existentes, pero sólo mejoran la eficacia de los algoritmos sobre términos no comprimidos si el término es altamente compresible.

Finalmente, destacar que existe una página web habilitada donde se puede descargar todo el código utilizado, así como probarlo online y comprobar que el resultado es correcto. Dicha web se puede encontrar en http://racso.lsi.upc.edu/unif/unif.php

10.2. Aprendizaje

Durante el transcurso del PFC he tenido que leer diversos artículos anteriores y entender su contenido, como por ejemplo [**Lif06, SSS09**]. Aunque no es la primera vez que leo este tipo de artículos sí es la primera vez que aplico directamente un resultado leído en uno de ellos.

En general el PFC me ha ayudado a profundizar con las compresiones de palabras y términos, aunque he de decir que ya conocía de su existencia antes de empezar con el proyecto. De hecho, en el examen que hice de la asignatura de Teoria de la Computació, una de las preguntas estaba estrechamente relacionadas con este tipo de compresiones. Después de hacer este PFC he aprendido varios resultados sobre gramáticas de este tipo, muchos de ellos incluidos en esta memoria.

Por otro lado también he aprendido diversos resultados sobre el problema de unificación, que hasta ahora desconocía. De hecho es probable que siga trabajando este problema en otras versiones con Adrià Gascón.

A la hora de diseñar los algoritmos que utilizaban números primos he tenido que adentrarme en teoría básica de números, especialmente en su versión computacional, para descubrir los algoritmos generadores de primos que se explican en el Capítulo 4. Aunque eran resultados bastante conocidos, nunca me había visto en la necesidad de implementar ninguno de ellos.

Este PFC ha sido la base de un artículo enviado al RTA 2011. Así que también he tenido que escribir parte del artículo enviado a dicho congreso[**GMR11**]. Aunque no era el primer artículo que escribía, sí que era el primero en el que no era tutorizado por un profesor. Además esto me llevó a participar en mi primer congreso, lo que considero que también forma parte de mi aprendizaje.

10.3. Aplicaciones de la carrera

A continuación incluyo una lista algunos de los sujetos que he aprendido durante la carrera y que considero que me han sido útiles en la realización de este PFC.

- Conceptos básicos de árboles explicados en las diferentes asignaturas de programación, como PRAP o ADA.
- Iniciación a problemas de lógica en IL, que me ayudó a la hora de comprender el problema de unificación.
- Conocimientos básicos de algoritmia en la asignatura de ADA, que me ayudaron a seguir profundizando en esa rama, que me ha sido muy útil a la hora de diseñar e implementar un algoritmo alternativo al mencionado en [**SSS09**].
- Conocimientos de gramáticas incontextuales, que son la base del sistema de compresión utilizado en este PFC, y que se explican en la asignatura de TC.

- Diversos conocimientos matemáticos, especialmente en las asignaturas cursadas en la Licenciatura de Matemáticas convalidadas en la Ingeniería Informática, y que me ayudaron a la hora de comprender los algoritmos que trabajaban sobre números primos, además de diversas operaciones sobre módulos.
- Conceptos financieros sobre proyectos explicados en la asignatura de PESBD que me permitieron realizar el análisis económico del PFC.
- Conceptos básicos de Ingeniería del Software, explicados en la asignatura de ES2, que me permitieron mejorar la comunicación entre las diferentes clases existentes en el proyecto, mejorando la modularidad.
- Intuición sobre la estructura de computadores a bajo nivel, explicada en EC2 entre otras asignaturas, y que me ayudó a entender cosas como el motivo del aumento de tiempo provocado por el uso de la clase BigInt.
- Introducción a conceptos de Sistemas Operativos en la asignatura de SO que no conocía, y que me permitieron desenvolverme con mayor soltura en el entorno de programación para poder desarrollar el trabajo.

10.4. Trabajo futuro

Pese a que los algoritmos implementados en este PFC mejoran sustancialmente los existentes para el trabajo sobre STG, aún existe mucho margen de mejora. De hecho estos algoritmos aún no son competitivos si los términos dados no se pueden comprimir mucho. Por eso unos de los principales campos donde se podría seguir investigando es en la obtención de cada vez mejores mejores algoritmos, que se puedan equiparar en complejidad a los algoritmos tradicionales sobre palabras y términos no comprimidos. En [**Lif06**] se explican algunas de estas posibles mejoras.

También sería interesante investigar las cotadas dadas para los algoritmos probabilistas. Pese a que teóricamente las cotas actuales podrían tener una probabilidad de error de 0.5, aún no se ha encontrado ninguna instancia que se acerque remotamente a esa cota. De hecho en [**SSS09**] se especula con que dicha cota se pueda rebajar sin verse comprometida la definición de algoritmo probabilístico. Con algunas de estas mejoras se conseguirían algoritmos cada vez más parecidos a los equivalentes sobre palabras descomprimidas.

El código existente en la página web de momento no cumple ningún estándar, y puede ser difícil de seguir para alguien que no haya participado activamente en él. Es por eso que si queremos que el código sea realmente Open Source una de las principales mejoras que tendríamos que hacer es retocar el código para que realmente cualquiera lo pueda entender o modificar. Asimismo, se podrían añadir funcionalidades a la página web, como la posibilidad de unificar términos que ya están comprimidos mediante una compresión propia (que puede ser mejor que la que hace automáticamente la herramienta TreeRePair).

Por último, parecería interesante realizar las comparaciones hechas sobre instancias reales, en lugar de instancias artificiales como las del Capítulo 8, que tienen la intención de probar los casos límites de uno y otro algoritmo.

Bibliografía

[AKS02] M. Agrawal, N. Kayal, and N. Saxena, *Primes is in p*, Ann. of Math **2** (2002), 781–793.

[BGK03] P. Buneman, M. Grohe, and C. Koch, *Path queries on compressed XML*, VLDB, 2003, pp. 141–152.

[BLM08] G. Busatto, M. Lohrey, and S. Maneth, *Efficient memory representation of xml document trees.*, Inf. Syst. **33** (2008), no. 4-5, 456–474.

[CLL+05] M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat, *The smallest grammar problem.*, IEEE Transactions on Information Theory **51** (2005), no. 7, 2554–2576.

[GGSS08] A. Gascón, G. Godoy, and M. Schmidt-Schauß, *Context matching for compressed terms*, LICS, 2008, pp. 93–102.

[GGSS09] A. Gascón, G. Godoy, and M. Schmidt-Schauß, *Unification with singleton tree grammars*, 365–379.

[GGSS10] A. Gascón, G. Godoy, and M. Schmidt-Schauß, *Unification and matching on compressed terms*, CoRR **abs/1003.1632** (2010).

[GMR11] A. Gascón, S. Maneth, and L. Ramos, *First-order unification on compressed terms*, RTA, 2011, pp. 51–60.

[HV09] K. Hoder and A. Voronkov, *Comparing unification algorithms in first-order theorem proving*, 435–443.

[KJP77] D. Knuth, J. Morris Jr., and V. Pratt, *Fast pattern matching in strings*, SIAM J. Comput. **6** (1977), no. 2, 323–350.

[Lif06] Y. Lifshits, *Solving classical string problems on compressed texts*, no. 06201.

[Lif07] ———, *Processing compressed texts: A tractability border*, CPM, 2007, pp. 228–240.

[LM06] M. Lohrey and S. Maneth, *The complexity of tree automata and XPath on grammar-compressed trees*, Theor. Comput. Sci. **363** (2006), no. 2, 196–210.

[LMM10] M. Lohrey, S. Maneth, and R. Mennicke, *Tree structure compression with RePair*, CoRR **abs/1007.5406** (2010), Short version to appear as paper in *Proc. DCC'2011*.

[LSSV08] J. Levy, M. Schmidt-Schauß, and M. Villaret, *The complexity of monadic second-order unification*, SIAM J. Comput. **38** (2008), 1113–1140.

[MS10] S. Maneth and T. Sebastian, *Fast and tiny structural self-indexes for XML*, CoRR **abs/1012.5696** (2010).

[Rob65] J.A. Robinson, *A machine-oriented logic based on the resolution principle*, J. ACM **12** (1965), 23–41.

[Ryt03] W. Rytter, *Application of lempel–ziv factorization to the approximation of grammar-based compression*, Theor. Comput. Sci. **302** (2003), 211–222.

[Ryt04] W. Rytter, *Grammar compression, LZ-encodings, and string algorithms with implicit input*, Proc. ICALP, LNCS, vol. 3142, 2004, pp. 15–27.

[SN10] K. Sadakane and G. Navarro, *Fully-functional succinct trees*, SODA, 2010, pp. 134–149.

[SSS09] M. Schmidt-Schauß and G. Schnitger, *Fast equality test for straight-line compressed strings*, unpublished manuscript.

[SSS10] ———, *Fast equality test for straight-line compressed strings*, unpublished manuscript, December 2010.