UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

# IMPLEMENTACIÓN DE ESTRUCTURA COMPRIMIDA SIMPLIFICADA PARA INDEXAR TEXTO BASADA EN GRAMÁTICAS

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN COMPUTACIÓN

CRISTÓBAL ENRIQUE FUENTES ALVARADO

PROFESOR GUÍA:
GONZALO NAVARRO

MIEMBROS DE LA COMISIÓN:
JUAN MANUEL BARRIOS
CLAUDIO GUTIÉRREZ
GONZALO NAVARRO

SANTIAGO DE CHILE
2025

# Resumen

El presente trabajo documenta la implementación de una estructura propuesta en el libro *Compact Data Structures, A Practical Approach* para la búsqueda de los índices de las ocurrencias de patrones en un texto.

La estructura es una representación comprimida del texto orientada a textos repetitivos, que representan el texto usando una gramática libre del contexto y permiten la búsqueda de patrones en tiempo sublineal.

El trabajo comprende también las mediciones de la implementación en términos de robustez y consistencia con la propuesta y sus predicciones teóricas del comportamiento de la estructura y su función de búsqueda de patrones en términos tanto temporales como espaciales.

La estructura implementada fue analizada en los aspectos relevantes de robustez y consistencia con el análisis teórico y según esto se tomaron conclusiones respecto a los logros del trabajo realizado, la utilidad de la estructura en un contexto real y las posibles mejoras a los tiempos de búsqueda en ámbitos de eficiencia y compresión efectiva en textos reales repetitivos.

*A Gloria.*

# Tabla de Contenido

# Índice de Tablas

# Índice de Ilustraciones

# Capítulo 1

# Introducción

El estudio de las estructuras de datos compactas es crucial en la actualidad, dado que la cantidad de información generada crece a un ritmo exponencial[22][21], superando ampliamente la capacidad de almacenamiento y procesamiento de los sistemas computacionales modernos. Este desequilibrio subraya la necesidad de técnicas eficientes que permitan manejar grandes volúmenes de datos utilizando menos espacio, sin comprometer significativamente tiempos de acceso y procesamiento. Desde los campos de *Big Data* y *Business Analytics* hasta las áreas de aprendizaje de máquinas es relevante la capacidad de procesar cantidades gigantescas de información de forma eficiente y rápida en un entorno arquitectónico que limita el espacio de memoria que a estas se les tiene permitido.

Desde los años 50, dentro del estudio de la teoría de la información y de la mano de Claude Shannon, se han desarrollado algoritmos de compresión de datos que permiten reducir el espacio de almacenamiento o el tiempo de transmisión, sin pérdida de la información contenida en los datos. Posteriormente, surgieron las estructuras de datos compactas, que permiten acceder a los datos comprimidos directamente, sin necesidad de descomprimirlos previamente. En cuanto a lo que compete el presente trabajo es menester mirar a un tiempo más cercano al presente: hitos importantes como el trabajo de Cook, Rosenfeld y Aronson [5] en 1976 sentaron las bases para que Kieffer y Yang publicaran en el 2000 *Grammar-Based Codes: A New Class of Universal Lossless Source Codes* [11] donde la compresión de texto en base a reglas de gramática simple se acerca a la entropía estadística de la fuente. Tabei, Takabatake y Sakamoto en 2013 utilizaron árboles para representar la gramática compacta[23]. Claude y Navarro en 2012 propusieron una estructura para la búsqueda de patrones en textos basados en gramática[3]. De esta última se desprende una versión simplificada descrita en *Compact Data Structures* [15, Capítulo 10.5.6] que concierne al trabajo a realizar en esta memoria.

La elección de cuál algoritmo y/o estructura utilizar depende primariamente de lo qué se desee hacer con el texto a comprimir. Si consideramos la búsqueda de patrones sobre textos de un largo cualquiera como la operación deseada entonces pasa a tomar más relevancia en la decisión de la elección el desempeño de los algoritmos y estructuras según los parámetros de los patrones y los textos de búsqueda. En muchos casos, distintas estructuras presenta desempeños similares en el análisis teórico, sin embargo, implementaciones muestran empíricamente que algunas se comportan mejor en función de ciertas características los datos.

Por esto, es necesario aseverar según las características de los datos que se desea procesar qué estructuras son mejores para cada una de las operaciones que se requieran, y para eso es esencial desarrollar implementaciones para las estructuras hasta ahora solo teorizadas.

La estructura comprimida simplificada para indexar texto basada en gramáticas ofrece una solución al problema de identificar todas las ocurrencias de un patrón de texto en un texto dado. Aunque no es la única estructura diseñada para abordar este desafío[4], presenta ventajas y desventajas que dependen de las características específicas del texto y del patrón de búsqueda. Su principal atractivo radica en la simplicidad de sus componentes (secuencias comprimibles, secuencias con permutaciones [15, 1, Capítulo 6.1], y grillas representadas mediante Wavelet Trees [15, 1, Capítulo 10.1]), lo que sugiere un posible buen desempeño. Sin embargo, el análisis teórico de su eficiencia en términos de tiempo y espacio no es suficiente para determinar su viabilidad práctica. Es necesario implementar la estructura y realizar evaluaciones empíricas comparativas que permitan determinar cuantitativamente si resulta más adecuada que otras soluciones de complejidad similar.

Del análisis de resultados fue posible concluir el correcto funcionamiento de la solución, su congruencia con la predicción teórica de su comportamiento, su utilidad con respecto a una solución estándar de búsqueda y posibles mejoras a la implementación.

## 1.1. Objetivos

### Objetivo General

El objetivo del trabajo presente consistió en programar una buena, esto es, optimizada y congruente al espacio y tiempo teórico de la estructura, implementación de lo descrito en el libro Compact Data Structures (Indexed Searching in Grammar-Compressed Text)[15, 1, Capítulo 10.5.6]. Utilizando pruebas de robustez y tiempo, fue posible un análisis empírico en función de los parámetros de entrada, obteniéndose conclusiones sobre el desempeño de la estructura. Fue posible comparar su desempeño con los algoritmos y estructuras actuales (y sus implementaciones) para la búsqueda de patrones en texto.

### Objetivos Específicos

1. Implementación la estructura de forma correcta. Esto incluye la implementación de cada una de las estructuras que componen la solución propuesta.
2. Implementación de pruebas de robustez y consistencia de la estructura.
3. Implementación de pruebas de desempeño espacial y temporal de la implementación.
4. Análisis de los resultados de las pruebas para obtener conclusiones respecto al desempeño empírico de la estructura.

## 1.2. Metodología

Para llevar a cabo este trabajo de investigación y cumplir con los objetivos planteados, se siguieron los pasos descritos a continuación:

1. Revisión bibliográfica y conceptualización de la solución: Se realizó un análisis detallado de la estructura comprimida basada en gramáticas descrita en el libro *Compact Data Structures*, específicamente el capítulo sobre *Indexed Searching in Grammar-Compressed Text*. Esta revisión incluyó la comprensión de las técnicas utilizadas, los algoritmos propuestos y sus posibles aplicaciones. Además, se investigaron estructuras y algoritmos actuales para la búsqueda de patrones en texto como punto de comparación. Se estudió la bibliografía pertinente a los conceptos teóricos utilizados en el trabajo presente y

2. Diseño de la implementación: Se definió una arquitectura modular para la implementación de la estructura propuesta. Esto incluyó la elección de patrones de diseño adecuados, la división del trabajo en componentes individuales y los algoritmos necesarios para crear la instancia de la estructura y la búsqueda.

3. Implementación de la estructura propuesta: Cada componente identificado fue implementado de forma incremental, priorizando los componentes independientes, y luego aquellos dependientes de los primeros, escribiendo al mismo tiempo pruebas unitarias para cada una de estas estructuras con el fin de garantizar la corrección de las operaciones, garantizando que cada módulo fuera funcional antes de la integración de cada parte necesaria para el funcionamiento del buscador de patrones.

4. Diseño y ejecución de pruebas de validación: Se desarrollaron casos de prueba enfocados en evaluar la robustez y consistencia de la estructura. Estas pruebas incluyeron escenarios con datos sintéticos y reales para validar que los resultados de las operaciones fueran correctos y se comportaran según lo esperado.

5. Pruebas de desempeño: Para evaluar el desempeño espacial y temporal de la estructura, se realizaron pruebas con conjuntos de datos de diferentes tamaños y características. Estas pruebas incluyeron mediciones de tiempo de búsqueda de patrones por cantidad de ocurrencias y largo de patrones, además de mediciones del uso de memoria. Los resultados se compararon con implementaciones existentes de estructuras similares.

6. Análisis de resultados: Se analizaron los datos obtenidos de las pruebas de desempeño, comparando los resultados de la estructura propuesta con las alternativas existentes. Este análisis permitió identificar fortalezas, debilidades y posibles mejoras para la estructura implementada.

7. Documentación y presentación de resultados: Finalmente, los hallazgos fueron documentados de manera estructurada, destacando las conclusiones principales y proporcionando recomendaciones basadas en los resultados del análisis en el trabajo presente.

# Capítulo 2

# Marco Teórico

## 2.1. Entropía

En problemas de compresión, la entropía indica el límite teórico mínimo para codificar un mensaje sin perder información. Una noción básica de entropía es el mínimo número de bits requeridos por identificadores, llamados códigos, si se asigna un código único a cada elemento de un conjunto $\mathcal{U}$ y todos los códigos tienen el mismo largo de bits. Esto corresponde a la entropía del peor caso de $\mathcal{U}$ y se denota $\mathcal{H}(\mathcal{U})$ y es equivalente a:

$$\mathcal{H}(\mathcal{U}) = \log |\mathcal{U}|$$

Donde log es el logaritmo en base 2.

La entropía es una medida de incertidumbre o desorden en un sistema. En el contexto de la teoría de la información, se utiliza para cuantificar la cantidad promedio de información que se obtiene al observar un evento aleatorio. Formalmente, la entropía $\mathcal{H}(X)$ de una variable aleatoria $X$ con un conjunto de posibles valores $\{x_1, x_2, \ldots, x_n\}$ y probabilidades asociadas $P(X = x_i)$, se define como:

$$\mathcal{H}(X) = -\sum_{i=1}^{n} P(X = x_i) \log(P(X = x_i)).$$

Equivalente a:

$$\mathcal{H}(X) = \sum_{i=1}^{n} P(X = x_i) \frac{1}{\log(P(X = x_i))}$$

La fórmula muestra que mientras más predecible es una secuencia de elementos, menos bits son necesarios para codificarla.

### 2.1.1. Entropía de orden cero

Si una secuencia $B$ de largo $n$ contiene $m$ 1s, (asumiendo que hay más 1s que 0s) se puede asumir que $P(X = 1) = \frac{m}{n}$. Entones la entropía de orden cero es:

$$\mathcal{H}(B) = \mathcal{H}_0(\frac{m}{n}) = \frac{m}{n} \log \frac{n}{m} + \frac{n-m}{n} \log \frac{n}{n-m}$$

En términos prácticos, la entropía de orden cero tiene el siguiente significado: si se intenta comprimir la secuencia $B$ usando códigos fijos $C_1$ para los 1s y $C_0$ para los 0s, entonces el tamaño total no puede ser menos que $n\mathcal{H}_0$ bits.

### 2.1.2. Entropía de orden $n$

La entropía de orden $n$, $\mathcal{H}_n$, considera las dependencias entre los símbolos de una secuencia, hasta el orden n. Mide la incertidumbre promedio de un símbolo si se conocen los $n$ símbolos anteriores:

$$\mathcal{H}_n = - \sum_{x_1,\ldots x_{n+1}} P(x_1, \ldots x_n) \log(P(x_n|x_1, \ldots x_n))$$

Donde $P(x_1, \ldots x_n)$ es la probabilidad de ver la secuencia $x_1 \ldots x_n$, y $P(x_n|x_1, \ldots x_n)$ es la probabilidad de ver el símbolo $x_n$ si se acabad de ver la secuencia mencionada.

En general, la entropía de mayor orden es menor o igual a la de menor orden, ya que se tienen en cuenta las dependencias que reducen la incertidumbre de la secuencia. Por ejemplo, en el lenguaje español, si se tiene la secuencia *ció* es muy probable que la siguiente letra es *n*. En aplicaciones de compresión de datos, esto implica que se puede obtener una mayor compresión en lenguajes donde hay secuencias muy repetitivas (como lo son textos reales).

## 2.2. Gramáticas

En el contexto de la computación, una gramática es un conjunto de reglas que describen la estructura de un lenguaje. Una gramática formal $G$ se define como un cuádruplo $(N, \Sigma, P, S)$, donde:

- $N$: Es un conjunto de símbolos no terminales.
- $\Sigma$: Es un conjunto de símbolos terminales.
- $P$: Es un conjunto de producciones o reglas de reescritura.
- $S$: Es el símbolo inicial.

En el trabajo presente, se trabajó con gramáticas "binarias", esto es, gramáticas donde las reglas de $P$ son de la forma:

$$A_i \to B_i C_i$$

Donde $A_i$ es un símbolo no terminal y $B_i$ y $C_i$ pueden ser terminales o no terminales. $B_i$ es referido como la expansión izquierda de $A_i$ y $C_i$ la expansión derecha.

## 2.3. Memoización

La memoización es una técnica de optimización utilizada para acelerar algoritmos mediante el almacenamiento de los resultados de cálculos costosos y su reutilización cuando sea necesario. Se emplea frecuentemente en problemas de programación dinámica, donde los subproblemas se resuelven de manera repetitiva. Al reducir el número de recomputaciones, la memoización mejora significativamente la eficiencia temporal, a cambio de utilizar espacio extra.

## 2.4. Notación $\mathcal{O}$ Grande

La notación $\mathcal{O}$ grande es una herramienta utilizada para describir la complejidad asintótica de algoritmos. Representa el peor caso del tiempo de ejecución o el uso de recursos como una función del tamaño de entrada $n$. Formalmente, un algoritmo tiene complejidad $\mathcal{O}(f(n))$ si existen constantes positivas $c$ y $n_0$ tales que:

$$T(n) \leq c \cdot f(n), \quad \forall n \geq n_0.$$

Esto permite comparar el comportamiento relativo de diferentes algoritmos independientemente de los detalles específicos de implementación o las constantes multiplicativas.

## 2.5. Búsqueda Lineal

La búsqueda lineal es un algoritmo simple para localizar un elemento en una lista. Consiste en recorrer secuencialmente la lista desde el principio hasta el final, comparando cada elemento con el valor buscado. Si el elemento se encuentra, el algoritmo retorna su posición; en caso contrario, indica que no está presente. La complejidad temporal de este método cuando se busca un único elemento en un conjunto de elementos es $O(n)$, donde $n$ es el número de elementos en la lista. Para el caso de búsqueda de un patrón de largo $m$ en una secuencia de largo $n$, la complejidad es $O(nm)$ en el peor caso.

## 2.6. Compresión Basada en Gramáticas

La compresión basada en gramáticas es una técnica para reducir el tamaño de datos generando una representación compacta en forma de gramática. En lugar de almacenar explícitamente los datos, se almacena un conjunto de reglas que permiten reconstruirlos. Esto es particularmente útil para datos con patrones repetitivos, ya que la gramática compacta captura dichas repeticiones de manera eficiente.

# Capítulo 3

# Estado del Arte

## 3.1. Representación de texto como gramática

En su artículo *Grammar-Based Codes: A New Class of Universal Lossless Source Codes* John C. Kieffer y En-hui Yang estudiaron el código basado en gramática[11], un tipo de codificación sin pérdida de información, el cual, en respuesta a cualquier cadena de datos de entrada $x$ sobre un alfabeto finito fijo, selecciona una gramática libre de contexto $G_x$ que representa a $x$ en el sentido de que $x$ es la única cadena o *string* generada por $G_x$. La compresión sin perdida de $x$ corresponde, indirectamente, a la compresión de estas reglas de gramática. Demostraron que, bajo ciertas restricciones, un código basado en gramática es un código universal, esto es, logra comprimir independiente de la fuente finita de generación de información a algo cercano a la compresión óptima, sobre un alfabeto finito.

Encontrar la gramática más pequeña que representa a un texto cualquiera $x$ es un problema NP-completo [2][20], y además esta gramática nunca es más pequeña que una codificación con LZ77[24] (con una ventana ilimitada) lo que motiva y justifica encontrar y utilizar heurísticas como *Re-Pair*[12] y Sequitur[16] que en la práctica compriman el texto a una cantidad de reglas cercanas al óptimo de forma rápida. A pesar de ser estrictamente inferior a LZ77, una de estas heurísticas, *Re-Pair*, se comporta bien en la práctica, tanto en textos clásicos como repetitivos.

### 3.1.1. Sequitur

El algoritmo *Sequitur*[16] funciona escaneando la secuencia de símbolos, agregando cada nuevo símbolo a una regla gramatical S y generando una lista con todos los pares que ha leído. Cuando un par es leído por segunda vez, se genera un símbolo no terminal, esto es, una regla que genera el par en la gramática, para reemplazar ambas ocurrencias en regla S y en todas la reglas donde aparezca. En otras palabras, se debe cumplir que cada par aparece solo una vez en S. El proceso se repite hasta que no hayan más pares repetidos. Si al finalizar el proceso, existen símbolos no terminales que sólo aparecen una vez a la derecha de la gramática, entonces deben ser reemplazados por los símbolos que generan. Esto ayuda

a reducir la cantidad de reglas.

Por ejemplo, sea la secuencia *abracabracabra*. Se avanza linealmente por esta secuencia agregando cada símbolo a la regla generadora S:

| Gramática | |
|---|---|
| S → a | |
| S → ab | |
| S → abr | |
| S → abra | |
| S → abrac | |
| S → abraca | |
| S → abracab | Se repite el par ab !! |
| S → AracA<br>A → ab | |
| S → AracAr<br>A → ab | |
| S → AracAra<br>A → ab | Se repite el par ra !! |
| S → ABcAB<br>A → ab<br>B → ra | Se repite el par AB !! |
| S → CcC<br>A → ab<br>B → ra<br>C → AB | |
| S → CcCc<br>A → ab<br>B → ra<br>C → AB | Se repite el par Cc !! |
| S → DD<br>A → ab<br>B → ra<br>C → AB<br>D → Cc | |
| S → DDa<br>A → ab<br>B → ra<br>C → AB<br>D → Cc | |

| Gramática | |
|---|---|
| S → DDab<br>A → ab<br>B → ra<br>C → AB<br>D → Cc | Se repite el par ab !! |
| S → DDA<br>A → ab<br>B → ra<br>C → AB<br>D → Cc | |
| S → DDAr<br>A → ab<br>B → ra<br>C → AB<br>D → Cc | |
| S → DDAra<br>A → ab<br>B → ra<br>C → AB<br>D → Cc | Se repite el par ra !! |
| S → DDAB<br>A → ab<br>B → ra<br>C → AB<br>D → Cc | Se repite el par AB !! |
| S → DDC<br>A → ab<br>B → ra<br>C → AB<br>D → Cc | A y B aparecen solo una vez a la derecha de las reglas, por lo tanto se eliminan |
| S → DDC<br>C → abra<br>D → Cc | Gramática final |

### 3.1.2. Re-Pair

El algoritmo *Re-Pair*[12] (*Recursive Pairing*) es una heurística de construcción de gramáticas a partir de un texto. Es utilizado en el trabajo presente para comprimir la secuencia de entrada de caracteres basado en los patrones repetitivos que aparecen en esta. La idea

básica detrás del algoritmo *Re-Pair* es encontrar pares de *substrings* repetidos en el texto y reemplazarlas con símbolos no terminales. Al aplicar este proceso de manera iterativa, se genera una representación gramatical comprimida que se puede utilizar para reconstruir el texto original. El método para comprimir consiste en recorrer el texto reemplazando los dos caracteres más comunes por un símbolo no terminal, generando una regla de gramática, remplazar los caracteres por el nuevo símbolo en la secuencia y repetir este proceso, hasta obtener un texto comprimido y una serie de reglas.

*Re-Pair* logra construir una gramática razonablemente óptima en tiempo $\mathcal{O}$(n), siendo $n$ el largo se la secuencia.

## 3.2. Compresión de gramática

El trabajo presentado consiste en la implementación una estructura basada en una representación sucinta de grilla para comprimir la gramática que genera el texto de entrada. En la siguiente sección se profundiza la representación de la gramática utilizando estructuras de árboles.

### 3.2.1. Dos árboles *LOUDS*

Dada una gramática R (obtenida, en este ejemplo, con *Re-Pair*) que genera un texto T, se tiene la regla S → C, donde C es el texto T luego de haberse hecho los remplazos por símbolos por *Re-Pair*. Para saber exactamente qué porción de C se debe expandir para obtener un T [i .. j] es útil guardar un vector de bits disperso que indica en qué posición de T aparece cada símbolo de C. Este vector solo necesita soportar la operación Rank en tiempo constante. Ya sabidos qué símbolos se deben expandir, lo único que se necesita es saber a qué expande cada símbolo no terminal (los símbolos terminales aparecen en C).

Tabei, Takabatake y Sakamoto introdujeron compresión de una gramática utilizando estructuras de árboles[23]. La idea es representar el grafo dirigido acíclico generado por la gramática donde cada regla A → BC induce una arista "izquierda" desde A a B y otra "derecha" de A a C. Tomando solo las aristas izquierdas, se puede interpretar una arista A → B como si B fuese el padre de A, obteniendo así un conjunto de árboles, ya que cada nodo puede tener a lo más un padre (símbolos terminales no tienen reglas y cada no terminal A tiene exactamente una regla con un término izquierdo B). Se añade una raíz como padre de todos los nodos sin padres, y se llama al árbol resultante $T_L$. Similarmente, se forma un árbol $T_R$ con las aristas derechas. Así, dada una no terminal A → BC, B es el padre de A en $T_L$ y C es el padre de A en $T_R$.

Como son necesarias solo las operaciones de arboles *parent, root, childrank, nodemap*, y *nodeselect*, una estructura de árbol LOUDS es ideal.

El árbol *Level-Order Unary Degree Sequence* (LOUDS) es una estructura que codifica los nodos del árbol en orden nivel, es decir, se recorren los nodos que están a la misma profundidad primero de izquierda a derecha antes de seguir al siguiente nivel. Cada nodo

se describe en una secuencia de bits con un código unario $1^c0$ donde $c$ es la cantidad de hijos. Las distintas operaciones requeridas son combinaciones de operaciones *Rank, Select* y *Predecessor Zero* sobre la secuencia de bits.

### 3.2.2. Índice comprimido basado en gramática

Claude, Navarro y Pacheco[4] implementaron una estructura que permite almacenar y consultar texto de manera eficiente, especialmente en colecciones de texto altamente repetitivas. Esta estructura permite tanto la extracción de subcadenas como la búsqueda de patrones directamente sobre una representación comprimida del texto. El texto es representado como la gramática libre de contexto que genera al texto, y esta gramática es a su vez representada como un árbol.

Las búsquedas de patrones de texto corresponde a ocurrencias primarias en el árbol (vistas como múltiples nodos en el árbol gramatical) y ocurrencias secundarias en las hojas.

La estructura utiliza $G \log n + o(G \log G)$ bits de espacio y la búsqueda de patrones toma tiempo $\mathcal{O}((m^2 + occ) \log G)$, donde $G$ es el tamaño de la gramática definido como la suma de las longitudes del lado derecho de las reglas.

### 3.2.3. Grilla con árboles *Wavelet*

En el trabajo presente, las $r$ reglas de la gramática son representadas en una grilla de $r \times r$, de forma que cada regla A $\rightarrow$ BC corresponde a un punto en la grilla en la posición C,B (columna correspondiente a C, fila correspondiente a B). La idea es que las columnas de la grilla corresponden a la parte C de cada regla, ordenadas según el valor lexicográfico del *string* al que se expande C, mientras que las filas corresponden a B, ordenadas por el valor lexicográfico del *string* invertido al que expande B. La grilla se representa utilizando árboles *Wavelet*[15, Capítulo 10.1].

La idea es que todas las operaciones que se necesitan sobre esta grilla para la gramática son equivalentes a operaciones sobre otra grilla donde por cada punto de $X_i, Y_i$ de la primera, hay un punto $i, Y_i$ en la segunda. Esto cerciora que solo haya un punto por columna, con lo cual la grilla se puede representar con una secuencia de los $Y_i$s. Esta secuencia es a su vez representada usando un árbol *Wavelet*.

La representación con árbol *Wavelet* consiste en lo siguiente: dado una secuencia $S_{[1,\sigma]}$ de símbolos sobre el alfabeto $\Sigma = [1, \sigma]$, se crea un nodo que corresponde a una secuencia de bits $B_{[1,\sigma]}$ de largo igual a la secuencia $S_{[1,\sigma]}$ donde por cada carácter de la secuencia original se coloca un 0 en la secuencia de bits si el carácter corresponde a un símbolo en $[1, \lceil \sigma/2 \rceil]$ o 1 si pertenece a $[\lceil \sigma/2 \rceil + 1, \sigma]$.

El nodo de $S_{[1,\sigma]}$, esto es, la secuencia de bits correspondiente a $S_{[1,\sigma]}$ obtenida del paso anterior, indica en qué mitad del alfabeto de la secuencia $S_{[1,\sigma]}$ se encuentra cada carácter de esta. Esto particiona virtualmente la secuencia original en dos partes: la secuencia $S_{[1,\lceil \sigma/2 \rceil]}$

de los caracteres de $S_{[1,\sigma]}$ que pertenecen a la primera mitad del alfabeto, y la secuencia $S_{[\lceil\sigma/2\rceil+1,\sigma]}$ de los caracteres que pertenecen a la segunda mitad. Para estas dos secuencias, se crean nodos de la misma forma en que se hizo para la secuencia original. El nodo resultante correspondiente a la secuencia $S_{[1,\lceil\sigma/2\rceil]}$ se agrega como hijo izquierdo del nodo de $S_{[1,\sigma]}$, y el nodo correspondiente a $S_{[\lceil\sigma/2\rceil+1,\sigma]}$ como hijo derecho.

El proceso de seguir dividiendo el alfabeto en dos se repite hasta llegar a secuencias mono-simbólicas. Las operaciones de *Rank* y *Select* sobre la secuencia original corresponden a recorrer el árbol haciendo operaciones *Rank* y *Select* sobre las secuencias de bits.

## 3.3. Implementaciones existentes

### 3.3.1. SDSL - *Succinct Data Structure Library*

La librería SDSL para C++ escrita por Simon Gog[10] es la más completa y profesional de las librerías dedicadas a estructuras de datos sucintas. La librería implementa estructuras sucintas relevantes para el trabajo realizado como lo son vectores de enteros, vectores de bits y soporte para operaciones *Access*, *Rank* y *Select* sobre ellos.

Implementaciones de arboles *wavelet* de distintas formas (balanceados, formas de Huffman, etc.) están presentes en la librería, pero la estructura en particular usada en propuesta del capítulo 10[15] utiliza matrices *wavelet*, que debieron ser implementadas como parte del trabajo realizado.

#### Otras librerías

La implementación original de Re-Pair en C[13] por R. Wan en C está basada en la propuesta del artículo original[12] y es la implementación usada en el trabajo presente. Otras implementaciones existen, incluyendo una hecha por G. Navarro[14].

# Capítulo 4

# Trabajo realizado

El trabajo realizado consistió en implementar y evaluar de forma empírica la estructura comprimida simplificada para indexar texto basada en gramáticas simple propuesta en el libro *Compact Data Structures,(Indexed Searching in Grammar-Compressed Text)* [15, Capítulo 10.5.6]. La implementación se encuentra disponible en el repositorio *SimpleTextIndexingBasedOnGrammar* [8].

## 4.1. Descripción General de la estructura

La idea principal de la estructura es representar un diccionario $\mathcal{R}$ de $r$ reglas $A \rightarrow BC$ correspondiente a la gramática generada sobre un texto $T$ usando el algoritmo *Re-Pair* como una grilla $G$ y una secuencia $R$ de símbolos que permite encontrar ocurrencias de patrones de texto en el texto original.

La secuencia $R$ corresponde a la sucesión de reglas generadas por *Re-Pair* expresadas como sus lados derechos, además de las reglas añadidas por la estructura con el fin de eliminar la secuencia $C$ generada por *Re-Pair*.

La grilla en tanto corresponde a una grilla de $r \times r$ dimensiones con $r$ puntos que corresponden a las $r$ reglas predispuestos en la grilla de uan forma particular (explicada en las siguientes secciones) que permite obtener rangos de reglas que contienen ciertos patrones.

La búsqueda de patrones corresponde a primero encontrar en la grilla el área de esta que contiene los puntos correspondientes a reglas que expresan al patrón, que corresponde a la búsqueda primaria, para entonces obtener las posiciones de las ocurrencias como los desfases de cada uno de estos puntos con respecto respecto al símbolo inicial de la gramática extendida, llamada búsqueda secundaria de ocurrencias.

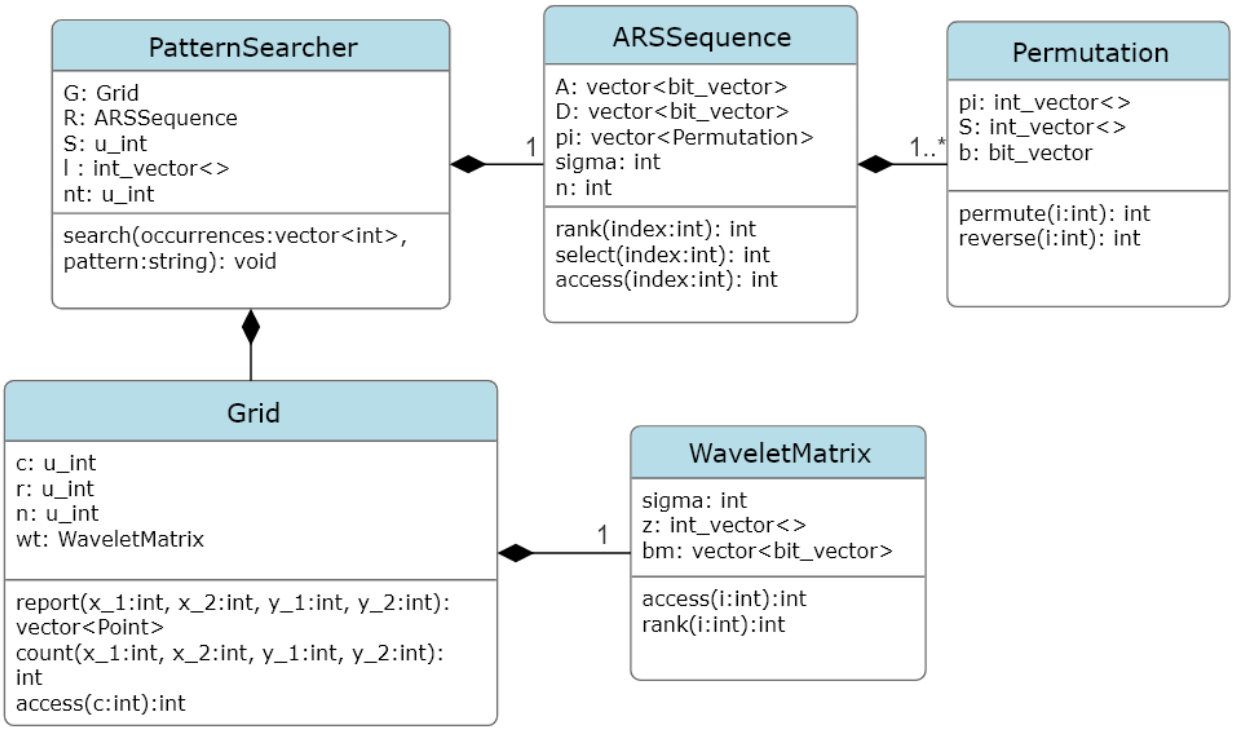
Las partes particulares y el detalle del proceso de búsqueda se explican en el presente texto, en las siguientes secciones.

## 4.2. Diseño de la implementación

Se implementó una clase *facade* llamada ***PatternSearcher*** que procesa la entrada e instancia las clases ***ARSSequence*** (secuencia representada por permutaciones) y ***Grid*** (grilla representada por matrices *wavelet*), además de los otros miembros necesarios (como primitivas, vectores de bit, *etc.*). *PatternSearcher* expone, además de su constructor, un único método ***search*** que corresponde a la búsqueda de patrones.

El diagrama UML 4.1 muestra el diseño general de la implementación.

Figura 4.1: Diagrama UML de la implementación




## 4.3. *Re-Pair*

La implementación de *Re-Pair* de Shirou Maruyama[13] que utiliza las estrcuturas propuestas en la publicación original de Larsson & Moffat[12]. Esta versión retorna una estructura que contiene un arreglo de reglas, además de la secuencia $C$ que corresponde a aquella que se obtiene una vez se aplican todas los reemplazos de las reglas en el texto original $T$, y otros valores como la cantidad de reglas y el largo del texto (Véase A.1). Las reglas corresponden a pares de enteros sin signo, donde los valores son menores a 256 si corresponden a una terminal o mayores si corresponden a una no terminal. Acceder a la posición $256 + i$ del arreglo entrega la regla $i$.

### 4.3.1. Generar reglas extras

Con objetivo de eliminar la secuencia $C$ para derivar el texto $T$ exclusivamente a partir de las reglas se crearon nuevas reglas $N_1 \rightarrow C[1]C[2]$, $N_2 \rightarrow C[3]C[4]$, $N_3 \rightarrow C[5]C[6]$, *etc*, reemplazándolas en $C$: $N_1N_2N_3...N_{\lceil |C|/2 \rceil}$. Luego se hizo lo mismo con este nuevo $C$, creando nuevas reglas $N_1^{'} \rightarrow N_1N_2$, $N_2^{'} \rightarrow N_3N_4$ y así sucesiva y recursivamente hasta obtener una única no terminal $S$ de la cual se puede derivar el texto original (Véase A.2).

Sea $r = |\mathcal{R}|$ el número de reglas, y sea $A_i \rightarrow B_iC_i \ \forall \ 0 \leq i < r$, el conjunto $\mathcal{R}$ es representado por la secuencia de enteros $R = B_0C_0B_1C_1...B_{r-1}C_{r-1}$. Nótese que la secuencia es auto-referencial: sea $R_i \rightarrow B_iC_i$, esta regla aparece en la secuencia en las posiciones $2i$ y $2i + 1$, y reglas que deriven en $R_i$ tendrán uno de sus dos símbolos $B$ o $C$ con un valor 256 $+ i$.

Tómese en cuenta que en el trabajo presente cuando se habla de los lados izquierdo y derecho de una regla $R_i$, estos se refieren, respectivamente, a $B_i$ y $C_i$. También, cuando se hable de la compresión del texto a gramática, se hace referencia a la combinación de los procesos de comprimir por *Re-Pair* seguido de la expansión de reglas con el fin de eliminar $C$.

La representación de la secuencia $R$, según las instrucciones del libro, utiliza permutaciones, y el detalle se explica más adelante, pero para hacer esta representación es necesario primero normalizar la secuencia de forma que los elementos en esta partan de 0 y sean continuos, es decir, el alfabeto de la secuencia no tiene saltos, y el mayor elemento es igual a la suma de las cantidades de terminales y no terminales menos 1.

## 4.4. Normalizar secuencia

Se creó un vector de bits $b$ (utilizando la librería *SDSL*[9]) de tamaño 256. La idea fue marcar con 1 las posiciones correspondientes a los símbolos terminales que aparecen en el texto $T$, que son los símbolos terminales que aparecen en $R$. Esto conllevó a la restricción de que el texto debe tener formato donde cada carácter utiliza solo 1 byte (por ejemplo, UTF-8). Añadiendo suporte para $Select_1(i)$ (reportar la posición del $i$-ésimo uno en el vector) y $Rank_1(i)$ (reportar la cantidad de unos hasta la posición $i$) sobre el vector se puede obtener el símbolo original de la secuencia normalizada. Se guardan entonces los resultados de *select* y *rank* sobre el vector de bits, y estos dos vectores de largo 256 con elementos de tamaño 8 bits son los que se usarán en la estructura.

La secuencia normalizada ahora tiene símbolos entre 0 y la suma de las cantidades de terminales y no terminales menos 1. Elementos en la secuencia menores a la cantidad de terminales corresponden a símbolos terminales, mientras los demás corresponden a no terminales. La regla $R_i$ aparece en las posiciones $i$ y $i+1$ correspondiente a $B_i$ y $C_i$ respectivamente. Reglas que expanden a $R_i$ son las reglas $R_j$ donde alguno de sus $B_j$ o $C_j$ tienen como valor $i+$ número de terminales. El número de terminales es equivalente a $rank_1(b, |R|)$. (Véase A.3)

Nótese que normalizar la secuencia es necesario solo para la implementación específica de

*Re-Pair* utilizada. En contraste, la versión de *Re-Pair* de Navarro[14] normaliza automáticamente las reglas, entregando la secuencia $C$, la secuencia $R$ de reglas, un valor numérico que indica la cantidad de símbolos terminales en el alfabeto usado en el texto y una secuencia numérica para obtener el símbolo original en el texto a partir del símbolo en la secuencia normalizada, exactamente como la implementación del trabajo presente.

## 4.5. Secuencia utilizando permutaciones

Se implementó la estructura descrita en el libro [15, Capítulo 6.1] para representar secuencias de números utilizando permutaciones. Esta estructura permite las operaciones *Access* y *Rank* en tiempo $\mathcal{O}(\log \log \sigma)$, donde $\sigma$ es el tamaño del alfabeto que compone la secuencia, y la operación *Select* en tiempo $\mathcal{O}(1)$. Esta última es importante pues es utilizada de forma frecuente en la búsqueda de ocurrencias secundarias en las reglas, lo cual será explicado más adelante.

### 4.5.1. Permutaciones

Una permutación $\pi$ de [1,n] es un reordenamiento de valores entre 1 y n. Descrita en el capítulo 5.1 [15], la estructura que compete al trabajo realizado permite la operación $\pi^{-1}(i)$, esto es, la permutación inversa de $i$: encontrar un $j$ tal que $\pi(j) = i$ en tiempo $\mathcal{O}(t)$, donde $t$ es un parámetro de la estructura.

La idea de la estructura es aprovechar el concepto de descomposición en ciclos de la permutación. Si se aplica una permutación sobre un valor inicial se obtiene un segundo valor, y luego se aplica sobre este valor la permutación, y así sucesivamente, se terminará llegando al valor inicial. Este recorrido de valores se llama ciclo, y una permutación puede tener uno o más ciclos.

Para calcular la permutación inversa de $i$ se aplica la permutación recursivamente hasta tener un $j$ cuya permutación es $i$. Esto requiere recorrer todo el ciclo que contiene a $i$. Sin embargo, si se guardan atajos de tamaño $t$, con la idea de que si el elemento sobre el cual se está aplicando la permutación durante el recorrido del ciclo tiene un atajo, se toma ese atajo, saltándose una gran parte de los pasos recursivos, asegurando encontrar el inverso en no más de $t$ pasos.

Esta estructura se implemento satisfactoriamente utilizando los vectores de bit de la librería SDSL[9] (Véase encabezado A.4).

### 4.5.2. Secuencia

Dada la secuencia $S$ de tamaño $n$ sobre un alfabeto $\Sigma$, se divide esta, conceptualmente, en $\lceil n/\sigma \rceil$ pedazos $S_i = S[i \dots i+\sigma)$. Para resolver *access*, *rank* y *select* se utilizan $\sigma$ vectores de bit $A_c$, con $c \in \Sigma$, donde $A_c = 1^{rank_c(S_0,\sigma)}01^{rank_c(S_1,\sigma)} \dots 01^{rank_c(S_{\lceil n/\sigma \rceil - 1},\sigma)}$. En esencia, $A_c$

indica de forma unaria las ocurrencias del símbolo $c$ en casa pedazo de $S$. Con esto, las operaciones a nivel de los pedazos son:

Para todas $k = \lfloor i/\sigma \rfloor$.

$$access(S, i) = access(S_k, i \bmod \sigma)$$

Para *rank*, se debe calcular la cantidad de unos que aparecen en los pedazos anteriores al que corresponde a $i$:

$$rank_c(S, i) = \begin{cases} rank_c(S_k, i \bmod \sigma) & \text{si } k = 0, \\ rank_c(S_k, i \bmod \sigma) + select_0(A_c, k) - k & \text{si } k > 0 \end{cases}$$

Para *select* se debe encontrar el pedazo al que pertenece el $i$ buscado, luego la respuesta es la suma de la posición donde parte este pedazo y *select* sobre el pedazo, menos la posición del último cero antes del pedazo.

$$select_c(S, j) = (s - j + 1) \cdot \sigma + select_c(S_{s-j+1}, s - \text{pred}_0(A_c, s)) \text{ donde } s = select_1(A_c, j)$$

Las operaciones dentro de cada pedazo $C$ requieren representar estos como la permutación inducida por su índice invertido. Sea $L_c$ la secuencia de las posiciones de los símbolos $c$ en el pedazo $C$. Considérese la permutación $\pi = L_0 L_1 L_2 L_3 ... L_{\sigma-1}$ y las lista $D$ que marca las posiciones donde empieza cada lista en $\pi$, $D = 0^{|L_0|}10^{|L_1|} \dots 0^{|L_{\sigma-1}|}$

Utilizando la estructura anteriormente implementada se pueden resolver las operaciones dentro de los pedazos. Por ejemplo:

$$access(C, i) = select_0(D, j) - j, \text{ donde } j = \pi^{-1}(i)$$

Esta estructura se implementó correctamente (Véase el encabezado A.5)

## 4.6. Reordenar secuencia

La secuencia $R$ obtenida a partir de las reglas una vez completado el proceso de normalización es tal que estas reglas aparecen en el orden en que fueron creadas por el algoritmo *Re-Pair* y extendidas con el fin de eliminar la secuencia $C$. Lo que se quiere es que las reglas $R_i \to B_i C_i$ aparezcan ordenadas de forma creciente según el valor lexicográfico de la expansión inversa del lado izquierdo ($B_i$).

La función de la librería estándar de C++ *sort* puede ordenar la secuencia mientras se le otorgue una forma de expandir las reglas, pero esto no es suficiente pues se necesita que la secuencia de reglas mantenga la propiedad de auto-referencia, es decir, que cada regla $R_i$ aparezca en las posiciones $2i$ y $2i + 1$ y que referencias a esta regla tenga el valor $i + \sigma$. Para lograr esto, se creó un vector de enteros que guarda los índices de cada regla.

Listing 4.1: Vector de índices

```
int_vector reverseIndexMap(n_non_terminals);
for (int i = 0; i < n_non_terminals; i++) {
   reverseIndexMap[i] = i;
}
```

Luego se ordenó utilizando una función que compara las expansiones de los lados izquierdo de la regla apuntada por el índice, de forma inversa.

Listing 4.2: *sort*

```
sort(
    reverseIndexMap.begin(),
    reverseIndexMap.end(),
    [&](int a, int b) {
        return compareRulesLazy(arsSequence, a, b, n_terminals,
           select, true);
    }
);
```

La función de comparación es una función perezosa que entrega el siguiente símbolo de la expansión pedida a demanda, esto evita tener que expandir el lado requerido por completo, lo cual, en un texto largo con miles de reglas, puede llevar demasiado tiempo. Para esto se utilizaron generadores:

Listing 4.3: Comparación perezosa

```
Generator<char> expandRuleSideLazy(
    ARSSequence& arrs, int i, int nt,
    std::vector<char>& sl, bool left = false)
{
    int lr_i = left? i: i+1;
    if (arrs[lr_i] < nt) {
        co_yield sl[arrs[lr_i] + 1];
    } else {
        auto gen = expandRuleLazy(arrs, 2*(arrs[lr_i]-nt), nt, sl,
           left);
        for (char c : gen) {
            co_yield c;
        }
    }
}
bool compareRulesLazy(ARSSequence& arrs, int i, int j, int nt,
   std::vector<char>& sl, bool rev = false)
{
    auto gen_i = expandRuleSideLazy(arrs, 2 * i, nt, sl, rev);
    auto gen_j = expandRuleSideLazy(arrs, 2 * j, nt, sl, rev);
    auto it_i = gen_i.begin();
    auto it_j = gen_j.begin();
    while (it_i != gen_i.end() && it_j != gen_j.end()) {
        char char_i = *it_i;
```

```
        char char_j = *it_j;
        if (char_i != char_j) {
            return char_i < char_j;
        }
        ++it_i;
        ++it_j;
    }
    // If one sequence is shorter, the shorter one is considered "less"
    return (it_i == gen_i.end()) && (it_j != gen_j.end());
}
```

Con esto, el vector *reverseIndexMap* (rim) ahora contiene los índices de las reglas de forma tal que:

$$\forall i, j \; expansion\text{-}reversa(B_{rim[i]}) < expansion\text{-}reversa(B_{rim[i]}) \longleftrightarrow i < j$$

Se creó un vector del mismo tamaño que $R$ y se colocaron en este las reglas en el orden que aparecen en *reverseIndexMap*, pero actualizando los valores $B$ y $C$:

Listing 4.4: *Nueva secuencia R*

```
vector<int> distance_of_find(reverseIndexMap.size(), 0);
for (int i = 0; i < reverseIndexMap.size(); i++) {
    distance_of_find[reverseIndexMap[i]] = i;
}
int_vector<> sortedSequenceR = int_vector(n_non_terminals * 2 + 1, 0);
for (u_int i = 0; i < reverseIndexMap.size(); i++) {
    int a_i = reverseIndexMap[i];
    int b_i = normalized_sequenceR[a_i*2];
    int c_i = normalized_sequenceR[a_i*2+1];
    int n_b_i, n_c_i;
    if (b_i < n_terminals) {
        n_b_i = b_i;
    } else {
        n_b_i = distance_of_find[b_i - n_terminals]  + n_terminals;
    }
    if (c_i < n_terminals) {
        n_c_i = c_i;
    } else {
        n_c_i = distance_of_find[c_i - n_terminals] + n_terminals;
    }
    sortedSequenceR[i*2] = n_b_i;
    sortedSequenceR[i*2+1] = n_c_i;
}
int S_i = distance(reverseIndexMap.begin(),
   find(reverseIndexMap.begin(), reverseIndexMap.end(),
   n_non_terminals-1));
sortedSequenceR[n_non_terminals*2] = S_i;
R = ARSSequence(sortedSequenceR, max_normalized + 1 + 1);
```

La última linea guarda el índice de la regla inicial (anteriormente, la regla inicial era aquella expresada por los dos últimos valores en $R$, ahora debe guardarse su posición).

Se creó también un vector similar a *reverseIndexMap*, llamado *indexMap* que guarda los índices de las reglas en el arreglo anteriormente ordenado, ordenadas por el valor lexicográfico de la expansión (no inversa) del lado derecho de cada regla. Este vector se utilizará para crear la grilla.

### 4.6.1. Memoización

Es posible aplicar la técnica de memoización para reducir el tiempo de la función de comparación, al guardar los valores de las expansiones de las reglas:

Listing 4.5: *Nueva secuencia R*

```
vector<int> distance_of_find(reverseIndexMap.size(), 0);
for (int i = 0; i < reverseIndexMap.size(); i++) {
    distance_of_find[reverseIndexMap[i]] = i;
}
int_vector<> sortedSequenceR = int_vector(n_non_terminals * 2 + 1, 0);
for (u_int i = 0; i < reverseIndexMap.size(); i++) {
    int a_i = reverseIndexMap[i];
    int b_i = normalized_sequenceR[a_i*2];
    int c_i = normalized_sequenceR[a_i*2+1];
    int n_b_i, n_c_i;
    if (b_i < n_terminals) {
        n_b_i = b_i;
    } else {
        n_b_i = distance_of_find[b_i - n_terminals]  + n_terminals;
    }
    if (c_i < n_terminals) {
        n_c_i = c_i;
    } else {
        n_c_i = distance_of_find[c_i - n_terminals] + n_terminals;
    }
    sortedSequenceR[i*2] = n_b_i;
    sortedSequenceR[i*2+1] = n_c_i;
}
int S_i = distance(reverseIndexMap.begin(),
   find(reverseIndexMap.begin(), reverseIndexMap.end(),
   n_non_terminals-1));
sortedSequenceR[n_non_terminals*2] = S_i;
R = ARSSequence(sortedSequenceR, max_normalized + 1 + 1);
```

Esto sin embargo requiere mucho espacio extra y no comprime el texto, por lo que no es parte de la estructura, sin embargo posibles casos de utilidad son discutidos al final del trabajo.

## 4.7. Ejemplo práctico

Considérese el texto $T =$ *abrabracadabrabra* y su versión normalizada:

$$T = 0\ 1\ 4\ 0\ 1\ 4\ 0\ 2\ 0\ 3\ 0\ 1\ 4\ 0\ 1\ 4\ 0$$

Considérese también la gramática representada por la secuencia $R$, normalizada y reordenada:

$$R = 0\ 9\ 8\ 11\ 5\ 9\ 7\ 10\ 1\ 12\ 2\ 0\ 3\ 7\ 4\ 0$$

Donde la regla inicial es $R_1 = 8\ 11$.

Sea $s(i) = select_1(b, i+1)$ ($b$ es el vector de bits obtenido durante la normalización de la secuencia) y $\sigma$ el tamaño del alfabeto de terminales (en este caso, con $\Sigma = [0, 1, 2, 3, 4]$ se tiene $\sigma = 5$), la figura 4.2 ilustra la expansión de las reglas, con $R_1$ la regla inicial que expande al texto original.

$$R = 0\ 9\ 8\ 11\ 5\ 9\ 7\ 10\ 1\ 12\ 2\ 0\ 3\ 7\ 4\ 0$$
$$(R = aR_4\ R_3R_6\ R_0R_4\ R_2R_5\ bR_7\ ca\ dR_2\ ra)$$

$$\begin{aligned}
R_0 &\to 0\ 9 \iff s(0)\ R_{9-\sigma} \iff \mathbf{a}\ R_4 \iff \mathbf{a}\ \text{bra} \\
R_1 &\to 8\ 11 \iff R_{8-\sigma}\ R_{11-\sigma} \iff R_3\ R_6 \iff \mathbf{abrabraca}\ \text{dabrabra} \\
R_2 &\to 5\ 9 \iff R_{5-\sigma}\ R_{9-\sigma} \iff R_0\ R_4 \iff \mathbf{abra}\ \text{bra} \\
R_3 &\to 7\ 10 \iff R_{7-\sigma}\ R_{10-\sigma} \iff R_2\ R_5 \iff \mathbf{abrabra}\ \text{ca} \\
R_4 &\to 1\ 12 \iff s(1)\ R_{12-\sigma} \iff \mathbf{b}\ R_7 \iff \mathbf{b}\ \text{ra} \\
R_5 &\to 2\ 0 \iff s(2)\ s(0) \iff \mathbf{c}\ \text{a} \\
R_6 &\to 3\ 7 \iff s(3)\ R_{7-\sigma} \iff \mathbf{d}\ R_2 \iff \mathbf{d}\ \text{abrabra} \\
R_7 &\to 4\ 0 \iff s(4)\ s(0) \iff \mathbf{r}\ \text{a}
\end{aligned}$$

$$l = 4\ 17\ 7\ 9\ 3\ 2\ 8\ 2$$

Figura 4.2: Secuencia $R$ y expansión de las reglas $R_i$, junto con secuencia $l$, el largo de cada expansión

Como se aprecia al expandir cada regla, estas están ordenadas de forma ascendente por el valor lexicográfico de la expansión invertida del lado izquierdo (en negrita).

Es posible visualizar esta gramática como un **árbol sintáctico** o *parsing tree* (Véase figura 4.3) que se obtiene de recorrer $R$ desde la regla inicial $R_1$. Esta figura permite visualizar la idea de **Gramática Balanceada**: en una gramática balanceada, la altura del árbol sintáctico es del orden $\mathcal{O}(\log n)$, con $n$ es el largo del texto original, es decir, existe una constante $c$ tal que la altura es $\leq c \log n$ para todos los textos de largo $n$.

Para que la gramática representada por $R$ sea balanceada es menester que la implementación de *Re-Pair* genere una gramática balanceada, luego la expansión de las reglas es balanceada naturalmente.

Otra visualización de la gramática que será de utilidad para visualizar la búsqueda de patrones es la de un grafo acíclico dirigido o DAG (del inglés *Directed Acyclic Graph*) (Fígura 4.4).

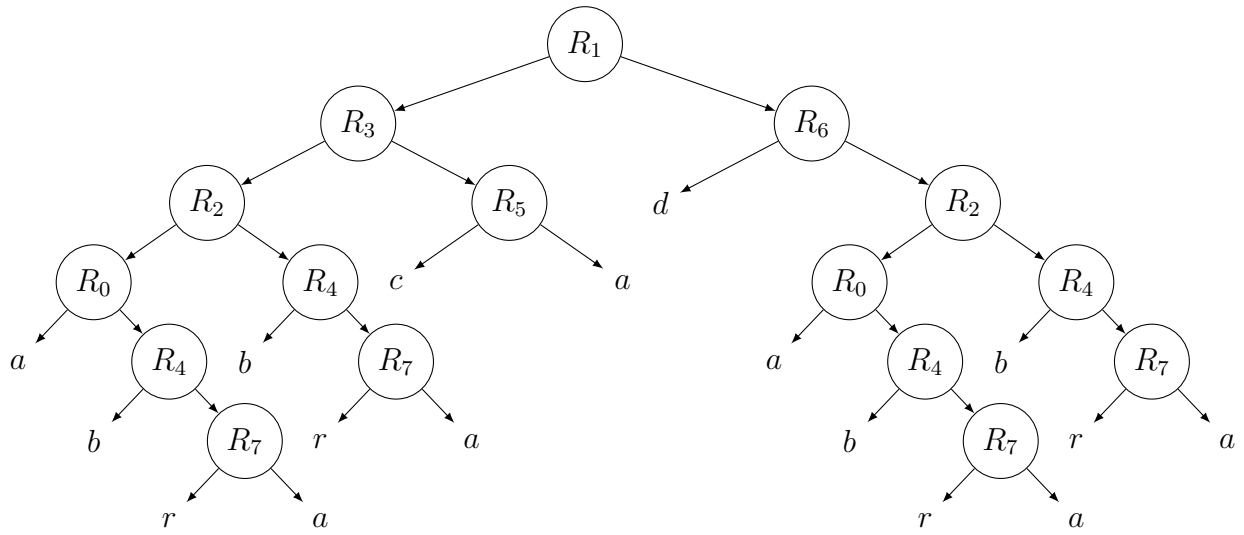



Figura 4.3: Árbol sintáctico para *abrabracadabrabra*

El DAG se forma de la siguiente forma. Cada regla tiene un único nodo correspondiente en el grafo. Cada vez que una regla $R_i$ aparece ya sea como lado izquierdo o derecho de otra regla $R_j$, esto induce una conexión desde el nodo $R_i$ al nodo $R_j$. El único nodo sin conexiones salientes corresponde a la regla inicial, en este caso, $R_1$.

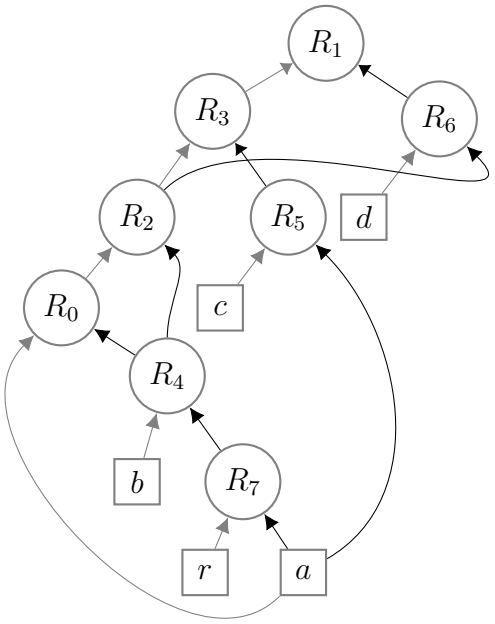



Figura 4.4: DAG para *abrabracadabrabra*, las conexiones que corresponden a reglas que aparecen como lados izquierdos son grises, y las derechas son negras

## 4.8. Grilla

*Compact Data Structures*[15] describe en su décimo capítulo la estructura de grilla en base a árboles *wavelet* (más precisamente, las estructuras utilizan matrices *wavelet*, pero los algoritmos descritos en el libro utilizan árboles). Para esto primero se ordenan los puntos de entrada por la coordenada $x$. Luego, cada punto $(x_i, y_i)$ es representado en la grilla por el punto $(i, y_i)$. El mapeo entre los puntos originales y los nuevos se guarda en un vector de bits, sin embargo, esto es innecesario para el caso presente debido a que los valores $x_i$ son únicos y continuos, con lo cual al ordenar los puntos por $x$, $(x_i, y_i) = (i, y_i)$. Una vez se tienen los puntos ordenados, si se consideran ahora solo los valores $y_i$ de cada punto, se tiene una secuencia $S$. Es a partir de esta secuencia que se crea la matriz *wavelet*[15, Capítulo 6.2.5].

### 4.8.1. Matrices *Wavelet*

La idea de la matriz *wavelet* es concatenar todos los vectores de bits en un mismo nivel para deshacerse de la topología de árbol. La forma particular en que son concatenados los vectores busca evitar espacios vacíos que aparecen en el árbol (pues no todos los caminos raíz-hoja tienen el mismo largo) es la siguiente: se colocan primero los vectores de bits correspondientes a hijos izquierdos del nivel anterior y luego los hijos derechos. Por ejemplo, para la secuencia "*tobeornottobethatisthequestion*":

$S_1$ : `tobeornottobethatisthequestion`
$B_1$ : `11001101111001001011001101 1010` $z_1 = 13$

$S_2$ : `benbehaiheein toorottottstqusto`
$B_2$ : `0010010110011 10000110111101110` $z_2 = 14$

$S_3$ : `bebeaee oorooqo nhihin ttttttstust`
$B_3$ : `0101011 0010000 100001 0000000100` $z_3 = 22$

$S_4$ : `bba ooooqo hihi ttttststst eeee r nn u`
$B_4$ : `110 000010 0101 111110101` $z_4 = 10$

$S_5$ : `a ooooo hh ss bb q ii ttttttt`

Donde $z_l$ es un valor pre-calculado equivalente a $Rank_0(B_l, n)$.

La estructura y sus operaciones se implementaron correctamente (Véase A.6) siguiendo las instrucciones del capítulo 6.2.5 de *Compact Data Structures*[15]. Con esto, se implementó la estructura de grilla usando matrices *wavelet*[15, Capítulo 10.1] (Véase A.7).

### 4.8.2. Preparar puntos para grilla

Cada regla $R_i \to B_i C_i$ se guarda en la grilla en un punto con coordenadas $(B_i, C_i)$. Las filas de la grilla están ordenadas por orden lexicográfico del reverso de la expansión de $B_i$ (esto

ya se hizo). Las columnas en tanto están ordenadas por orden lexicográfico de la expansión de $C_i$. Para lograr esto se usó el vector *indexMap* descrito previamente:

```
std::vector<Point> points(n_non_terminals);
u_int j, k;
for (u_int i = 0; i < indexMap.size(); i++) {
    k = std::distance(indexMap.begin(), std::find(indexMap.begin(),
       indexMap.end(), i));
    points[i] = Point(k, i);
}
```

Estos puntos se usaron para inicializar la grilla (La implementación de la grilla usa valores indexados desde 1, por lo que hay que sumar 1 a los valores de los puntos antes de usarlos).

Por ejemplo, las reglas generadas por "*abrabracadabrabra*"(Véase la fígura 4.2), conforman la siguiente grilla:

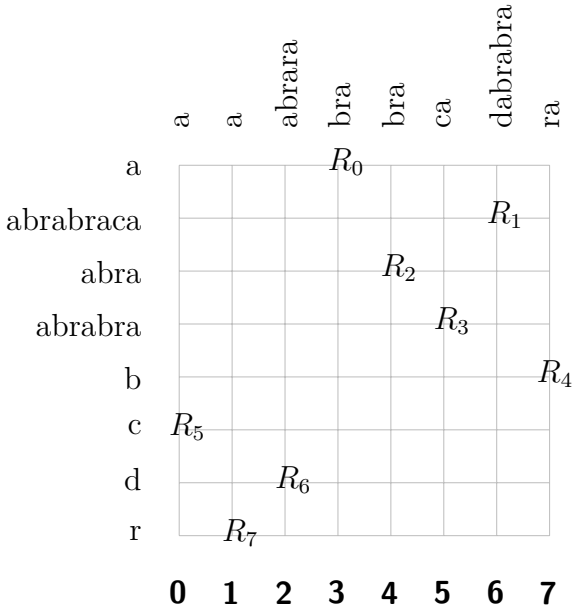



Figura 4.5: Grilla

Si se lee la grilla fila por fila aparecen las reglas en este orden: $R_0, R_1, R_2, ...R_7$, es decir, en el orden preexistente, pues ya fueron ordenadas por orden lexicográfico de la expansión reversa del lado izquierdo (mostrado en la figura a la izquierda de cada fila). Si se leen las reglas columna a columna, el orden es $R_5, R_7, R_6, R_0, R_2, R_3, R_1, R_4$, pues las columnas están ordenadas por orden lexicográfico de la expansión del lado derecho (mostrado arriba de la grilla sobre cada columna correspondiente).

## 4.9. Calcular largo de las expansiones de las reglas

Es menester, para poder responder consultas de búsqueda de patrones, pre-calcular los valores de los largos de las expansiones de cada regla. El detalle se ve más adelante, pero

en resumen, si una ocurrencia de un patrón sucede en una regla que aparece como el lado derecho $C_i$ de otra regla, el índice del patrón estará desfasado $l_{B_i}$ con respecto al índice de la regla padre, donde $l_{B_i}$ es el largo de la expansión de la regla $B_i$ que es el lado izquierdo de la regla $R_i$.

```
l = int_vector(n_non_terminals, 0); // largos de cada regla
for (int i = 0; i < n_non_terminals; i++) {
    l[i] = ruleLength(i);
}
```

Como las reglas son referenciadas por otras reglas (y dependiendo de lo repetitivo del texto, son referenciadas más de una vez), con el fin de evitar calcular el largo para una misma regla cada vez que esta es parte de la expansión de otra, se usó *memoización* (en este caso, la misma lista de largos funciona como la memoria).

```
int PatternSearcher::ruleLength(int_vector<> *l, int i) {
    if (l[i] != 0) { //memoization
        return l[i];
    }
    int left, right;
    if (R[i*2] < nt) {
        left = 1;
    } else {
        left = ruleLength(R[i*2] - nt);
    }
    if (R[i*2+1] < nt) {
        right = 1;
    } else {
        right = ruleLength(R[i*2+1] - nt);
    }
    l[i] = left + right;
    return l[i];
}
```

## 4.10. Búsqueda de patrones

La búsqueda de patrones aprovecha la grilla para encontrar las reglas en las que aparece el patrón de texto buscado. La idea es la siguiente: si el patrón $P$ a buscar aparece en el texto, entonces existe al menos una división del patrón $P$ en dos *strings* $P_<$ y $P_>$ que son prefijo y sufijo del patrón respectivamente y que concatenados forman el patrón $P$, tales que $P_<$ es sufijo de la expansión izquierda de una regla $R_i$ y $P_>$ es prefijo de la expansión derecha de la misma regla. Si se tienen todas las reglas que cumplen esta condición, basta con recorrer virtualmente el árbol sintáctico o *parsing tree* hasta el símbolo inicial, y entregar la posición donde parte $P_<$ tomando en cuenta los desfases con respecto al nodo padre.

La idea entonces es, primero, y por cada división $P_<$ y $P_>$ del patrón, encontrar todas la reglas que expresan el patrón de la forma descrita (ocurrencias primarias), y luego, por cada

una de estas reglas encontradas, hacer accesos en $R$ hasta encontrar todas las posiciones de esta regla en el símbolo inicial (ocurrencias secundarias). El detalle se ve en las siguientes secciones.

### 4.10.1. Ocurrencias primarias

Para cada posible división del patrón $P$ en dos *strings*, uno prefijo y otro sufijo $P_<$ y $P_>$, se buscan las reglas cuya expansión izquierda es $P_<$ y derecha $P_>$.

Como las filas están ordenadas por orden lexicográfico de la expansión reversa del lado izquierdo de estas, las reglas que cuyo lado izquierdo terminan en $P_<$ forman un rango de filas en la grilla. De forma análoga, las columnas están ordenadas de forma lexicográfica por la expansión del lado derecho, por lo que las reglas con lado derecho que empieza con $P_>$ forman un rango de columnas. Esto significa que se puede encontrar el rango de filas y columnas (y por lo tanto, el cuadrante donde se encuentran las reglas que cumplen con la condición buscada) usando búsqueda binaria.

Por ejemplo, sea el patrón de búsqueda $P = \mathbf{ab}$ sobre el texto *abrabracadabrabra*, se tienen los posibles (y en este caso únicos) $P_< = \mathbf{a}$ y $P_> = \mathbf{b}$. Las reglas que tienen como sufijo en su extensión izquierda a $P_< = \mathbf{a}$ están en el rango de filas $[0, 1, 2, 3]$ (en ázul en la figura 4.6), mientras que las reglas que tienen como prefijo en el lado derecho a $P_> = \mathbf{b}$ están en el rango de columnas $[3, 4]$ (en rojo en la figura 4.6).

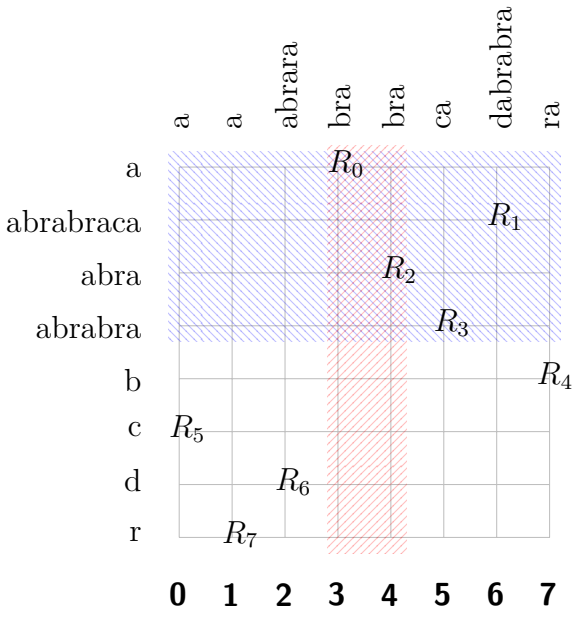



Figura 4.6: Grilla

Las reglas $R_0$ y $R_2$ se encuentran en el cuadrante definido por los dos rangos encontrados. Estas reglas pueden obtenerse mediante la operación *report* de la grilla. El siguiente paso es determinar los índices de las reglas en el texto original (Véase sección 4.10.2).

### Implementación

La búsqueda de las reglas que contienen el patrón consiste en primero dividir este en dos *sub-strings* ($P_<, P_>$), según una variable $t$ (el largo de $P_<$). Por cada $t$ entre 1 y el largo del patrón menos uno, se deben encontrar $s_y$ (primera fila del rango de filas), $e_y$ (última fila del rango), $s_x$ (primera columna del rango de columnas) y $e_x$ (última columna del rango).

```
u_int m = P.size();
u_int t;
for (t = 1; t < m; t++) {
    string P_left = P.substr(0, t); // P_<
    string P_right = P.substr(t, m-t); // P_>
    uint s_x, e_x, s_y, e_y;
```

Para buscar los rangos $s_x, e_x, s_y, e_y$ , se utilizó búsqueda binaria, como se ve en 4.6, donde se muestra la búsqueda binaria para $s_y$. En este caso, la fila tiene el mismo identificador que la regla (línea 5) gracias a la disposición de los puntos usados en la grilla.

Listing 4.6: Búsqueda binaria para $s_y$

```
int left = 0, right = G.getRows() - 1;
int result = -1;
while (left <= right) {
    int mid = left + (right - left) / 2;
    int r_i = mid;
    int compare = compareRuleWithPatternLazy(R, r_i, nt, sl, P_left,
       true);
    if (compare >= 0) {
        if (compare == 0)
            result = mid;
        right = mid - 1;
    } else {
        left = mid + 1;
    }
}
if (result == -1) continue;
s_y = result + 1;
```

En el caso de las columnas, la línea 5 de 4.6 debe cambiar, el índice de la regla corresponde al valor del punto en la columna:

```
int r_i = G.access(mid+1)-1; // rule index
```

Donde *G.access* entrega el valor del único punto en la columna de entrada.

Para encontrar el final de cada rango, lo único que cambia en la búsqueda binaria es como se mueven los límites de la búsqueda (*left* y *right*):

```
    if (compare <= 0) {
        if (compare == 0) {
```

```
            result = mid;
        }
        left = mid + 1; // instead of mid - 1
    } else {
        right = mid - 1; // instead of mid + 1
    }
```

La función *compareRuleWithPatternLazy* compara el patrón con la expansión ya sea izquierda o derecha de una regla, como se ve en 4.7.

Listing 4.7: Ocurrencias

```
template <typename Iterator>
int compareRuleWithPatternLazyImpl(
    ARSSequence& arrs, int i, int nt, std::vector<char>& sl, Iterator
       pattern_begin, Iterator pattern_end,
    bool rev = false)
{
    auto gen = expandRuleSideLazy(arrs, 2*i, nt, sl, rev);
    auto it = gen.begin();
    while (it != gen.end() && pattern_begin != pattern_end) {
        char c = *it;
        char p = *pattern_begin;
        if (c < p) return -1;
        if (c > p) return 1;
        ++it;
        ++pattern_begin;
    }
    if (it == gen.end() && pattern_begin != pattern_end) return -1;
    return 0;
}
int compareRuleWithPatternLazy(ARSSequence& arrs, int i, int nt,
   std::vector<char>& sl, std::string pattern, bool rev = false)
    {
    if (rev) {
        return compareRuleWithPatternLazyImpl(arrs, i, nt, sl,
           pattern.rbegin(), pattern.rend(), rev);
    } else {
        return compareRuleWithPatternLazyImpl(arrs, i, nt, sl,
           pattern.begin(), pattern.end(), rev);
    }
}
```

Esta operación utiliza las funciones de expansión perezosa descritas en la sección 4.6, en el fragmento 4.3.

Una vez encontrados los rangos, se deben encontrar las ocurrencias de las reglas que se encuentran en este:

```
vector<Point> points = G.report(s_x, e_x, s_y, e_y);
for (Point p: points) {
```

```
    int r_i = p.second-1; // rule index
    if ((u_int)R[r_i*2] < nt) {
        secondaries(occurences, R, S, r_i, nt, l, 0);
    } else {
        secondaries(occurences, R, S, r_i, nt, l, l[R[r_i*2] - nt]-t);
    }
}
```

El desfase inicial es cero si el lado izquierdo es una terminal, en el caso contrario el desfase es la diferencia entre el largo de la expansión del lado izquierdo y el largo de $P_<$.

### 4.10.2. Ocurrencias secundarias

Determinar las posiciones de las reglas encontradas en el símbolo inicial corresponde a recorrer virtualmente el DAG desde los nodos correspondientes a cada regla encontrada en la búsqueda de ocurrencias primarias hasta el nodo correspondiente al símbolo inicial, acumulando el desfase de cada nodo en el camino: si la regla es el hijo derecho del nodo destino, su desfase respecto a este es igual al largo de la expansión de la correspondiente regla izquierda.

La idea es recorrer todos los caminos posibles hasta el nodo inicial, y entregar los desfases para cada recorrido.

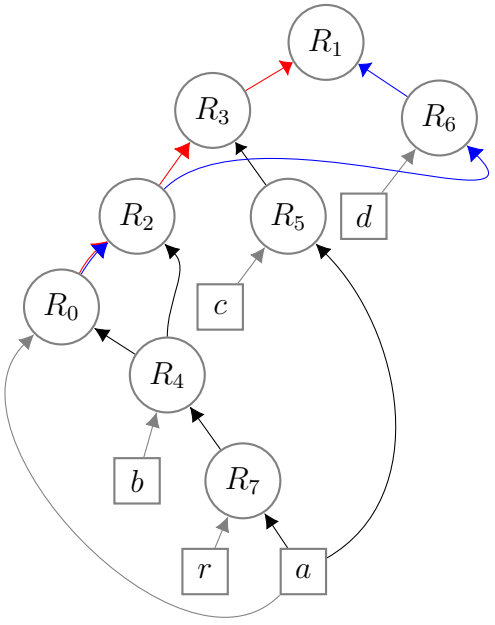



Figura 4.7: Búsqueda en el DAG para nodo $R_0$

Siguiendo el ejemplo de la sección anterior, se necesita ahora encontrar las ocurrencias de

las reglas $R_0$ y $R_2$ en el texto, considerando el desfase inicial del patrón $P = \mathbf{ab}$ con respecto a cada regla.

La figura 4.7 muestra el recorrido por el DAG que corresponde a la búsqueda de ocurrencias secundarias para el patrón $P = \mathbf{ab}$ expresado en $R_0 = \mathbf{a}\ \mathbf{b}ra$. Los dos caminos posibles (en rojo y ázul) llegan cada uno a $R_1$ con distintos desfases. El camino rojo llega con un desfase acumulado de 0 (el desfase inicial es 0 pues el patrón coincide con el inicio de la regla), lo que indica que el patrón (expresado por la regla $R_0$) aparece en la posición 0 del texto (indexado desde cero). El camino azul acumula un desfase igual a $l[d] + l[R_3] = 1 + 9 = 10$ (Véase figura 4.2 para los valores de $l$), con lo que el patrón (en la regla $R_0$) aparece también en la posición 10 del texto.

Considere ahora la búsqueda para la regla $R_2$. La fígura 4.8 muestra los recorridos realizados por la búsqueda.

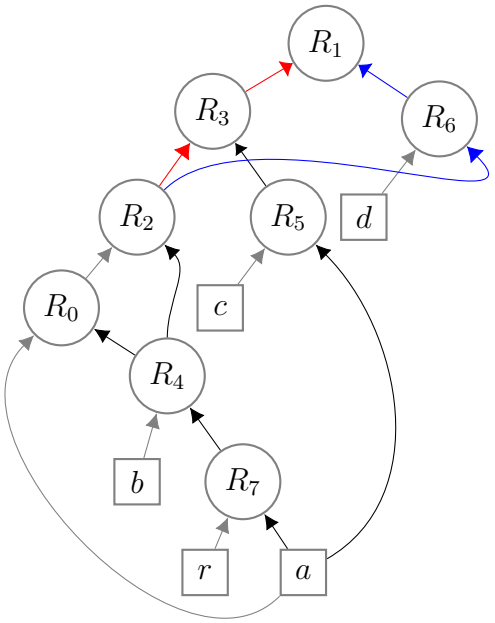



Figura 4.8: Búsqueda en el DAG para nodo $R_2$

El camino rojo llega con un desfase acumulado de 3 (el desfase inicial del patrón $P = \mathbf{ab}$ respecto a la regla $R_2 = abr\mathbf{a}\ \mathbf{b}ra$), mientras que el camino azul llega con un desfase $3 + l[d] + l[R_3] = 3 + 1 + 9 = 13$. Con esto se concluye que el patrón aparece (expresado en la regla $R_2$) en las posiciones 3 y 14. En total, sumando a las ocurrencias encontradas para $R_0$, el patrón aparece en las posiciones 0, 3, 10 y 14 del texto.

En este ejemplo, solo se necesitó encontrar un cuadrante, pues solo existía un par ($P_<$, $P_>$) para dividir **ab**, pero un patrón más largo tiene múltiples divisiones, por lo que para cada par ($P_<$, $P_>$) que tengan un cuadrante en la grilla válido se debe hacer el proceso de encontrar las ocurrencias secundarias.

### Implementación

La búsqueda de ocurrencias secundarias en la implementación consiste en acceder $R$ de forma recursiva. La idea es la siguiente, en cada recursión, para una regla $R_k$ y un desfase acumulado se buscan todas las ocurrencias de la regla en $R$ (la cantidad de ocurrencias es dada por $r = rank(R, R_k)$, y cada ocurrencia $j \leq r$ está en $i = select(R, R_k, j)$), y por cada una de estas, si la ocurrencia corresponde a un hijo derecho (es decir, si la posición $i$ de la $j$-ésima $R_k$ en $R$ es impar) entonces se agrega al desfase acumulado el largo de la regla en la posición $i - 1$. Seguido de esto se llama la búsqueda de ocurrencias secundarias para la regla $R_{i/2}$ que es la que contiene esta ocurrencia específica $j$ de $R_k$ en $R$, es decir, es el nodo padre en el árbol sintáctico, con el desfase acumulado. La implementación de esto se ve en el fragmento 4.8. Cada vez que se llega al símbolo inicial $S$ el desfase será distinto, correspondiente a cada índice del patrón buscado en el texto.

Listing 4.8: Ocurrencias

```
void secondaries(vector<int> *occs, ARSSequence R, u_int S,
    u_int A_i, u_int nt, int_vector<> l, u_int offset=0,
    bool terminal = false) {
    if (!terminal && A_i == S) {
        occs->push_back(offset); return;
    }
    int c = terminal? A_i: A_i + nt; // nt = number of terminals
    for (int j=1; j <= R.rank(c, R.size()); j++) {
        int k = R.select(c, j);
        int D_i = k / 2;
        int offset_prime = offset;
        if (k % 2 == 1) { // if A_i is right side
            if (R[k-1] < nt) offset_prime++;
            else offset_prime += l[R[k-1] - nt];
        }
        secondaries(occs, R, S, D_i, nt, l, offset_prime, false);
    }
};
```

En el fragmento de código 4.8 se observa cómo se recorre la secuencia $R$. La operación *rank* devuelve la cantidad de ocurrencias de la regla indicada por el parámetro $A_i$. que puede corresponder al índice de una regla o a un símbolo terminal, dependiendo del valor del *flag terminal*. Por cada ocurrencia, se utiliza *select* para obtener la posición correspondiente.

Si el texto original es repetitivo, pueden existir muchas ocurrencias de una misma regla, lo que hace que *select* sea la operación más utilizada. Es por esta razón que la secuencia se representa mediante permutaciones, lo que permite realizar *select* en tiempo $\mathcal{O}(1)$. Durante el proceso recursivo, el desfase se acumula en la variable *offset*. La recursión termina cuando $A_i$ es el símbolo inicial $S$, en cuyo caso el desfase acumulado hasta entonces se añade a las ocurrencias.

# Capítulo 5

# Evaluación

Para evaluar la solución se deben considerar dos aspectos de la implementación: el funcionamiento correcto del código en términos de entrada y salidas de datos en cada una de sus partes (unidades) y la consistencia del programa con las proyecciones de tiempo y espacio teóricos.

## 5.1. *Unit Testing*

Para probar el funcionamiento del programa se utilizó la librería *Catch2*[18], que permite fácilmente crear pruebas unitarias (*Unit Testing* en Inglés). Las unidades en este caso son las distintas clases creadas para representar las estructuras necesarias para el programa: ***Permutation*** (véase A.4: permutación con vectores de bits y atajos), ***ARSSequence*** (véase A.5: Secuencias utilizando permutaciones), ***WaveletMatrix*** (véase A.6: secuencia representada como matriz *wavelet* utilizando vectores de bit), ***Grid*** (véase A.7: grilla utilizando matriz *wavelet* y ***PatternSearcher*** (véase A.8: buscador de patrones utilizando grilla y secuencia).

Como ejemplo, considérese la clase buscadora de patrones, se puede hacer pruebas que corroboren los resultados de la búsqueda:

Listing 5.1: *Test de búsqueda*

```
TEST_CASE("PatternSearcher","[pattern]") {
    REQUIRE_FALSE(g_fileName.empty());
    string input_filename = g_fileName;
    FILE *input  = fopen(input_filename.c_str(), "rb");
    string filecontent = "";
    char c;
    while (fread(&c, 1, 1, input) == 1) {
        filecontent += c;
    }
    fclose(input);
    PatternSearcher PS(input_filename);
    for (int i = 0; i < 50; i++) {
```

```
13          string pattern = filecontent.substr(rand() %
               (filecontent.size() - 10), rand() % 10 + 1);
14          cout << i << ": Searching for pattern: \"" << pattern << "\""
               << endl;
15          vector<int> occurences;
16          PS.search(&occurences, pattern);
17          sort(occurences.begin(), occurences.end());
18          vector<int> expected_occurences = findOccurrences(filecontent,
               pattern);
19          REQUIRE(occurences == expected_occurences);
20      }
21  }
22  vector<int> findOccurrences(const string& filecontent,
23      const string& pattern) {
24      vector<int> occurrences;
25      for (size_t i = 0; i < filecontent.size(); i++) {
26          if (filecontent.substr(i, pattern.length()) == pattern) {
27              occurrences.push_back(i);
28          }
29      }
30      return occurrences;
31  }
```

La prueba mostrada cerciora que el método utilizado *search* encuentre los índices de las ocurrencias del patrón generado aleatoriamente a partir del contenido del texto de entrada, comparándolos con los resultados arrojados por una función de búsqueda sobre el contendido (visto como un *string*) que utiliza funciones estándar en C++ para encontrar, en tiempo $\mathcal{O}(n)$, las ocurrencias.

## 5.2. Análisis empírico

### 5.2.1. Espacio

El espacio total de la estructura corresponde a la suma de los valores del espacio de la grilla $G$, el espacio de la secuencia $R$, el espacio de la secuencia $l$, los vectores que mapean los símbolos normalizados a los originales y el símbolo inicial:

$$SPACE(PS) = 32 + r \log n + 2 \times 8 \times 256 + SPACE(G) + SPACE(R)$$

El espacio de la grilla $G$ es igual al espacio de la matriz *wavelet WM* y los valores para guardar la cantidad de columnas, filas y puntos:

$$SPACE(G) = 3 \times 32 + SPACE(WM)$$

El espacio de la matriz *wavelet WM* corresponde al valor de $\sigma$, el vector de $z_l$ de largo $\log \sigma$ y el vector de largo $\log \sigma$ de vectores de bits de largo $n$. En este caso, como la matriz

se construye sobre la secuencia formada por los índices de las reglas, $\sigma$ y $n$ son ambos la cantidad de reglas:

$$SPACE(WM) = 32 + 32\log r + SIZE(BV)\log r$$

El vector de bits tiene, en el peor caso, un tamaño de $1{,}5n$, con lo que el tamaño total de la matriz queda:

$$SPACE(WM) = 32 + 32\log r + 1{,}5r\log r$$

Con esto, el espacio de la grilla G queda:

$$SPACE(G) = 3 \times 32 + 32 + 32\log r + 1{,}5r\log r$$

$$SPACE(G) = 96 + 32\log r + 1{,}5r\log r$$

El espacio usado por una secuencia $R$ de largo $n$ sobre un alfabeto $\sigma$ representada por permutaciones es igual a la suma de los $A_i$ y $D_i$ que hacen un total de $4n + o(n)$ más las permutaciones que usan un espacio total de $n\log\sigma + no(\log\sigma)$. Como $R$ se construye sobre la secuencia de largo $2r$ y el alfabeto corresponde a la suma del alfabeto del texto $\sigma$ y la cantidad de reglas $r$, sobrestimando los ordenes $o$ queda el espacio como:

$$SPACE(R) = 10r + 2r\log(r + \sigma)$$

El espacio total en bits de la estructura es entonces:

$$SPACE(PS) = 4224 + r\log n + +32\log r + 1{,}5r\log r + 10r + 2r\log(r + \sigma) \qquad (5.1)$$

### Construcción

El proceso de construcción de la estructura, según los propuesto en el libro, utiliza extras $\mathcal{O}(c+n)$ bits. En la implementación, el tamaño extra usado aparece en el proceso de construcción de la matriz:

Listing 5.2: Construccion de matriz

```
void WaveletMatrix::build(vector<u32>& S, u32 n, u32 sigma) {
    vector<u32> S_hat(n);
    bit_vector M(n, 0);
    bit_vector M_hat(n, 0);
    u32 m = sigma;
    for (u32 l = 0; l <= ceil(log2(sigma))-1; l++) {
        u32 z_l = 0;
        bit_vector B_l(n, 0);
        for (u32 i = 0; i < n; i++) {
            if (S[i] <= (m - M[i] + 1) / 2) {
                B_l[i] = 0;
                z_l++;
            } else {
                B_l[i] = 1;
                S[i] = S[i] - (m - M[i] + 1) / 2;
            }
        }
        bm.push_back(ppbv(B_l));
        z.push_back(z_l);
        if (l < ceil(log2(sigma)) - 1) {
            u32 p_l = -1; // max value + 1 = 0
            u32 p_r = z[l] - 1;
            u32 p;
            int n_ = n;
            for (u32 i = 0; i < n_; i++) {
                u32 b = bm[l][i];
                if (b == 0) {
                    p_l ++;
                    p = p_l;
                } else {
                    p_r ++;
                    p = p_r;
                }
                S_hat[p] = S[i];
                if (m % 2 == b) {
                    M_hat[p] = b;
                } else {
                    M_hat[p] = M[i];
                }
                if ((m+1)/2==2 && M_hat[p] == 1) {
                    n = n-1;;
                }
```

```
43              }
44              swap(S, S_hat);
45              swap(M, M_hat);
46              m = (m+1)/2;
47          }
48      }
49      S.clear();
50      S.shrink_to_fit();
51  }
```

Esto ocupa, efectivamente, $2n$ bits para los vectores de bit y $\mathcal{O}(n)$ bits para la secuencia auxiliar, lo que indica que la implementación es consistente con lo esperado según el análisis.

Si se utiliza memoización para expandir las reglas al momento de comparar y ordenar la secuencia, se requiere, en el peor caso, $\mathcal{O}(2n)$ **bytes** (¡no bits!) de memoria extra, considerando un árbol sintáctico binario balanceado de $n$ nodos. Incluso en el mejor caso, la memoria de la memoización requiere al menos $n$ bytes. Es posible mejorar esto guardando las expansiones de las reglas que más se repiten en el árbol (por ejemplo, las que otrora apareciesen en la secuencia $C$ generada por *Re-Pair*), y memoizar sólo prefijos de las reglas de cierto tamaño. En este caso es quizás posible utilizar otras estructuras como un *trie* o un *suffix-tree* en vez del mapa *int* a *string* utilizado como memoria.

### 5.2.2. Tiempo

#### Construcción

En teoría, la construcción de una estructura grilla usando matrices *wavelet* con $n$ puntos y $c$ columnas demora tiempo $\mathcal{O}(c + n \log n)$[15, Capítulo 10.6]. Esto se debe a que primero se deben ordenar los puntos por la coordenada $x$, y luego se recorren estos para inicializar los vectores de bits.

En el caso particular de la búsqueda de patrones, no es necesario ordenar los puntos. En efecto, la clase buscadora primero ejecuta *Re-Pair* sobre el texto de largo $n$ (tiempo $\mathcal{O}(n)$), después la normalización (tiempo $\mathcal{O}(n)$), para luego ordenar la secuencia $R$ de reglas por orden lexicográfico de la expansión reversa del lado izquierdo ($\mathcal{O}(r \log r)$ comparaciones donde expandir toma, en promedio, $\mathcal{O}(\log n)$). Los puntos son entonces creados de forma que las columnas estén ordenadas por el valor lexicográfico del lado derecho (tiempo $\mathcal{O}(r \log r \log n)$).

Todo lo anterior significa que la grilla entonces toma tiempo $\mathcal{O}(r)$, que sumado al tiempo necesario para ordenar la secuencia $R$, conlleva a un tiempo total de:

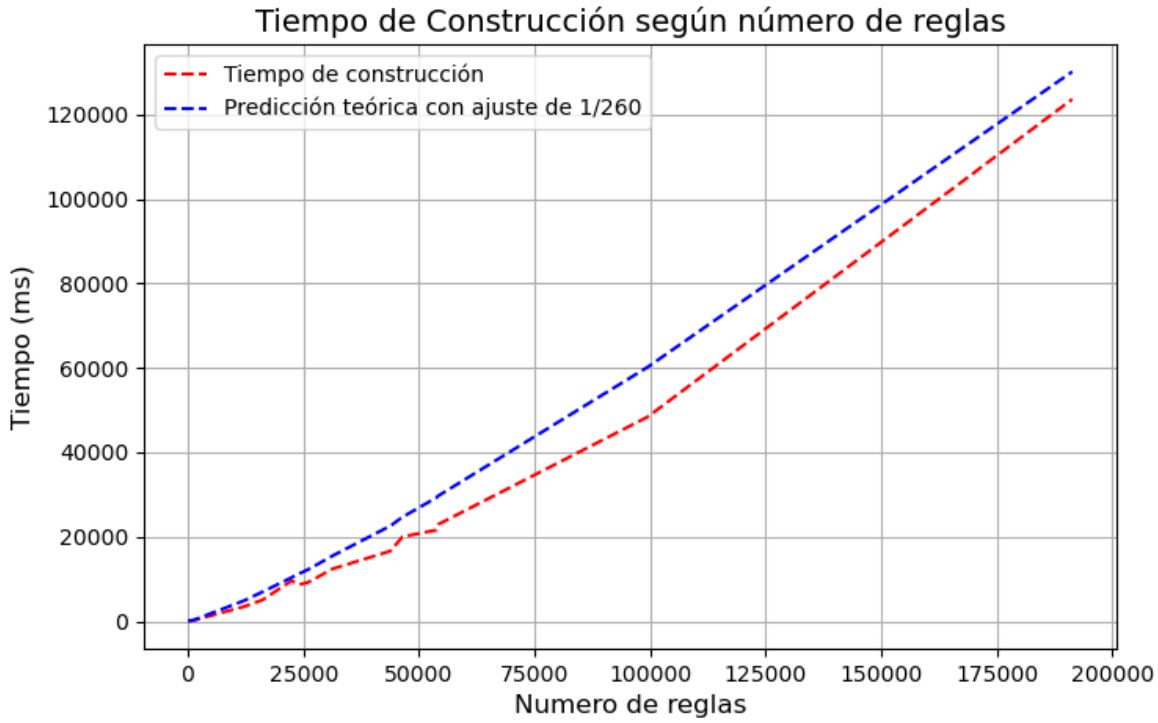
$$\mathcal{O}(n + r \log r \log n) \tag{5.2}$$

Con $n$ el largo del texto y $r$ la cantidad de reglas.

Se pudo medir el tiempo de construcción de la implementación utilizando textos reales,

obtenidos del sitio *Project Gutenberg*[7]. Para cada texto, se ejecutó la construcción de la clase buscadora de patrones varias veces, y se obtuvo el promedio de todas estas medidas. Los resultados se pueden ver en la figura 5.1, donde se comparan con la predicción teórica calculada 5.2.

Figura 5.1: Tiempo de construcción de la estructura en función del número de reglas, comparado a tiempo teórico $\mathcal{O}(n + r \log r \log n)$




La implementación demuestra comportarse exactamente como lo predicho por la fórmula obtenida del análisis teórico.

ES posible, con memoización, reducir el tiempo de construcción significativamente. Como las primeras reglas corresponden a las reglas creadas por *Re-Pair*, estas se expanden primero y se guardan. Luego, el resto de las reglas corresponden a las reglas extras creadas para reducir la secuencia $C$, y por lo tanto utilizan todas la memoización de forma consecutiva. Sin embargo, esta técnica utiliza significativa memora extra, y no logra comprimir el texto.

### Búsqueda

El costo de tiempo teórico para reportar $occ$ ocurrencias de un patrón $P$ de largo $m$ en el texto $T$ de largo $n$ es:

$$\mathcal{O}((m + \log n)m \log r \log \log r + occ \log n \log \log r)$$

Esto se debe a que en una gramática balanceada, el árbol sintáctico tiene altura $\log n$, y

la operación de acceso en la secuencia representada por permutaciones ($R$) tiene un tiempo $\mathcal{O}(\log \log r)$. De aquí que el primer sumando en la expresión teórica corresponde a expandir $m$ símbolos de una regla ($(m + \log n) \log \log r$), por cada comparación en la búsqueda binaria ($\log r$), por cada división de sufijo y prefijo del patrón ($m$). El segundo sumando en tanto corresponde a recorrer virtualmente hasta la raíz el árbol sintáctico ($\log n$), por cada ocurrencia encontrada ($occ$), haciendo accesos en $R$ ($\log \log r$).

Para medir el tiempo de búsqueda de la implementación en función de los parámetros, se crearon textos que permitiesen mantener fijos algunos de estos y variar el parámetro relevante.

El tiempo en función de la cantidad de ocurrencias requiere mantener fijo el largo de los patrones de búsqueda (además de los otros parámetros). Una solución simple fue usar los bigramas (secuencias de largo 2) más comunes en Inglés[17] como los patrones a buscar. Luego se buscan las ocurrencias de estos patrones en varios textos en inglés. Los gráfico de estas medición para cada texto real utilizado correspondes a la figuras de la tablas 5.4 y 5.5. El gráfico 5.2 muestra la combinación de todas las mediciones y la tendencia combinada polinomial de primer grado.

La figura 5.3 ilustra el tiempo teórico para fines de cerciorar el mismo crecimiento, utilizando valores promedios de $n$ y $r$. La justificación para esto es que, aunque el tiempo de búsqueda es función de $n$ y $r$, además de $occ$, es posible un análisis más simple considerando $r$ y $n$ como funciones lineales de $occ$ en textos reales, donde independiente del largo el texto no se vuelve menos o más predictivo, y la "densidad"de ocurrencias se mantiene igual.

Figura 5.2: Tiempo de búsqueda en función del número de ocurrencias para un patrón de largo dos

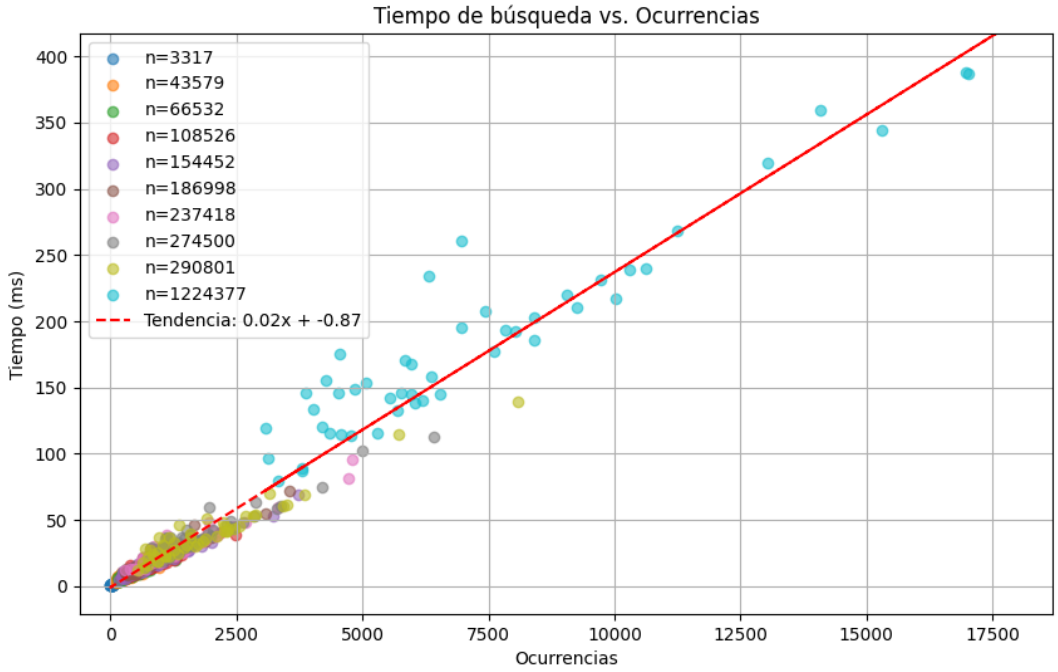
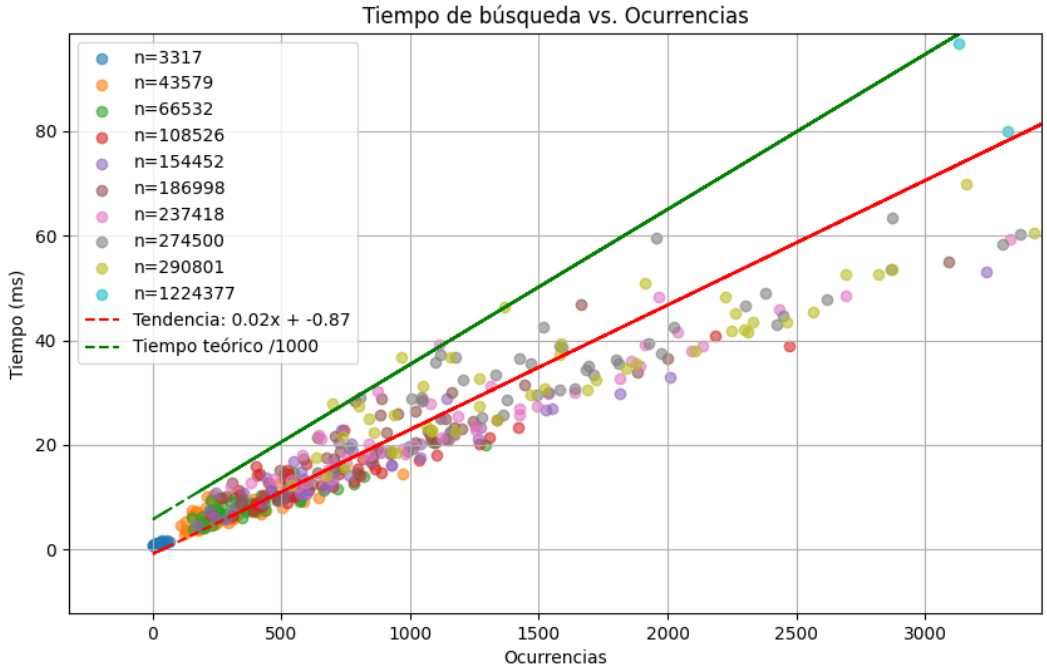


Figura 5.3: Tiempo de búsqueda en función del número de ocurrencias para un patrón de largo dos con predicción teórica usando promedio de $n$ y $r$, con tiempo teórico $O((m + \log n)m \log r \log \log r + occ \log n \log r)$.




El tiempo de búsqueda corresponde a la suma de los tiempos de búsqueda binaria de rangos en la grilla, el costo temporal de reportar las reglas dentro de estos rangos y los tiempos de búsqueda de ocurrencias para cada regla reportada.

La búsqueda binaria demora en el peor caso $m \log c$, pues la comparación puede requerir expandir la regla entera hasta el largo del patrón ($m$), y la regla puede corresponder a la regla inicial que expande al texto inicial de largo $n$. En la práctica, con un texto real, la gran mayoría de las comparaciones terminan en el primer símbolo de la expansión. Esto es fácil de ver: como la distribución de los primeros símbolos de las expansiones de las reglas es aproximadamente uniforme (en realidad, es la distribución según las frecuencias de los símbolos en el lenguaje específico) las comparaciones perezosas retornarán falso en el primer símbolo.

En un texto altamente repetitivo, las expansiones serán más largas, sin embargo, la cantidad de reglas es mucho menor que la de un texto real. Esto significa que no sólo la búsqueda binaria se hace sobre un espacio menor, el reporte de ocurrencias se hace sobre un árbol más corto (si se considera la búsqueda de ocurrencias como el recorrido del árbol sintáctico hasta la raíz).

Las pruebas de medición de tiempo muestran resultados consistentes a lo esperado en todos los casos, por lo cual se puede concluir que la implementación es correcta.

Figura 5.4: Tiempos de búsquedas en *ms* (milisegundos) en función del número de ocurrencias para un patrón de largo 2

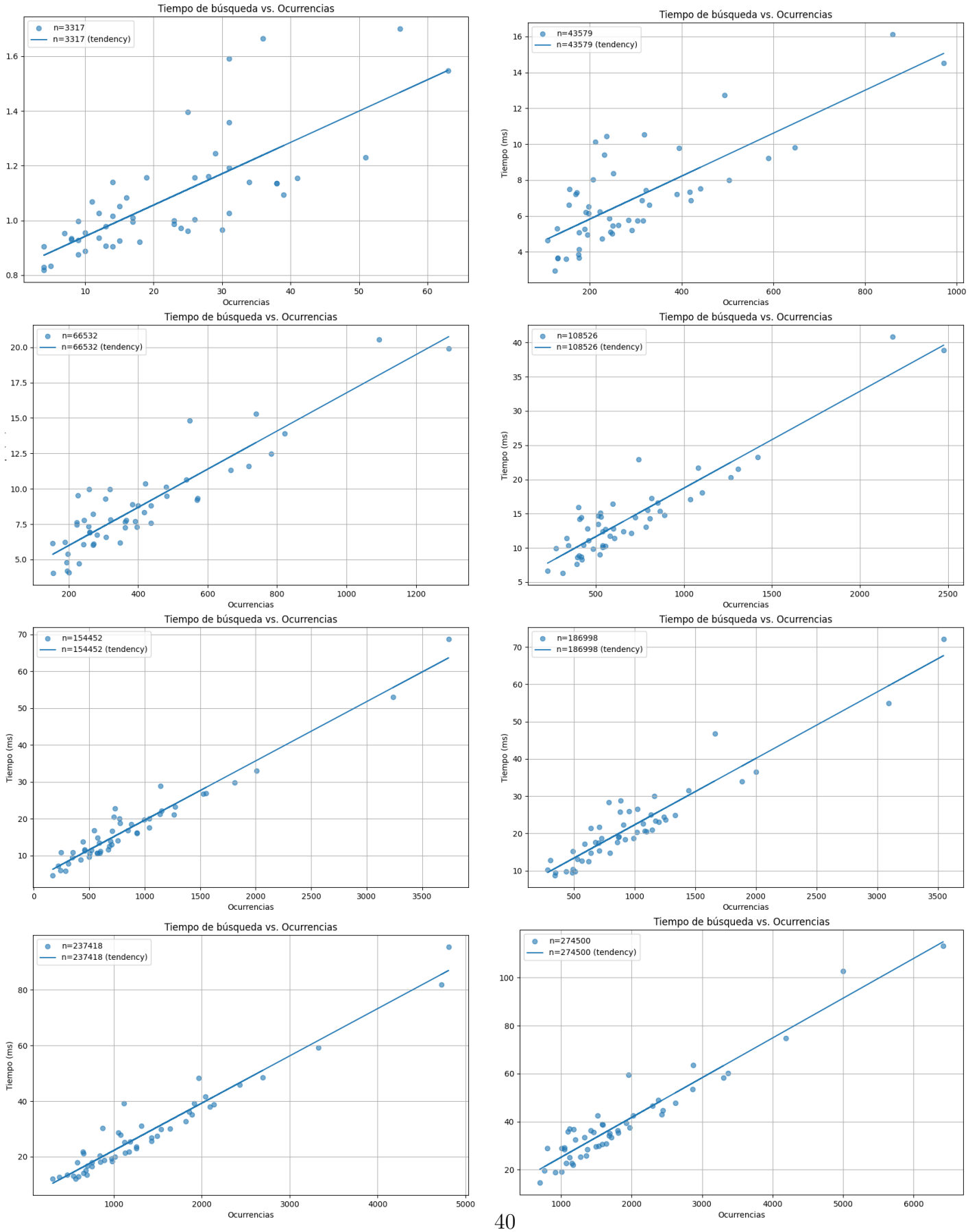
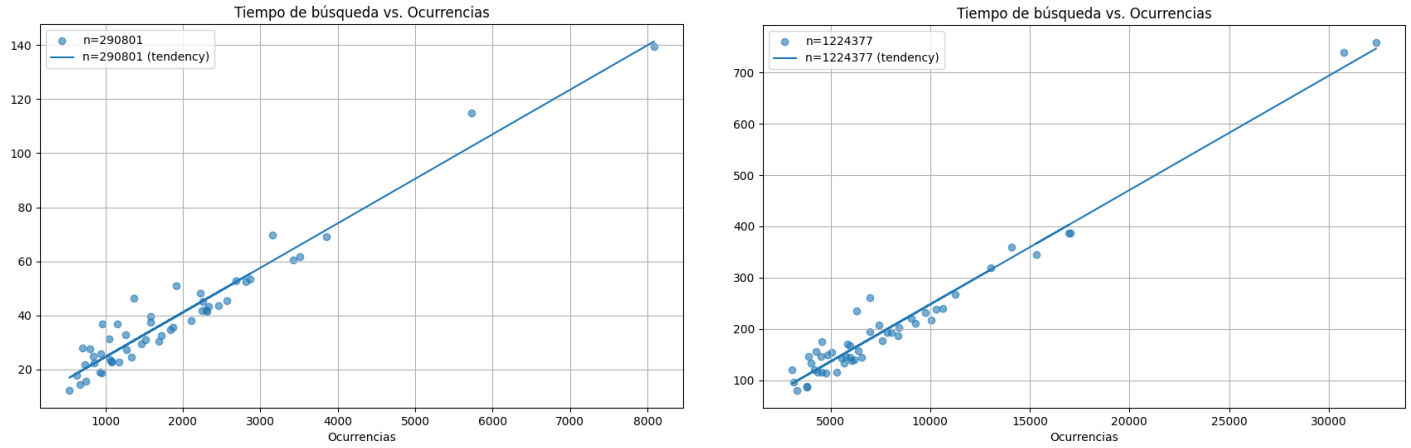


Figura 5.5: Tiempos de búsquedas en *ms* (milisegundos) en función del número de ocurrencias para un patrón de largo 2 (Continuación)




## 5.3. Análisis en textos altamente repetitivos

Las colecciones de textos reales altamente repetitivos de *Pizza & Chili Corpus*[1] sirven como entradas para pruebas del mismo tipo a las utilizadas por el índice comprimido basado en gramática[4] y son por lo tanto una buena forma de analizar la competitividad de la estructura utilizada. El tamaño de los textos varía desde 45 MiB (*world_leaders*) a 446 MiB (*einstein.en*), con variados grados de repetición, como se ve en la tabla 5.1. En la misma tabla aparece el espacio de la estructura para cada set de datos como ***bps*** (bits por símbolo).

| **Colección** | **Largo** | **Reglas** | *bps* |
|---|---|---|---|
| world_leaders | 46968181 | 307066 | 0.848511 |
| Escherichia_Coli | 112689515 | 3619577 | 4.55733 |
| influenza | 154808555 | 1557878 | 1.40496 |
| kernel | 257961616 | 1129349 | 0.619162 |
| coreutils | 205281778 | 1994376 | 1.36035 |
| para | 429265758 | 4222046 | 1.41742 |
| cere | 461286644 | 3212008 | 0.986939 |
| einstein.en | 467626544 | 163417 | 0.0475765 |

Tabla 5.1: Propiedades de cada colección

Para cada colección se hicieron búsquedas de patrones aleatorios de ciertos largos y se midieron los tiempos de búsqueda por ocurrencia. La figura 5.6 muestra los resultados obtenidos, ilustrados en un gráfico donde las escalas de ambos ejes son logarítmicas, y el eje $Y$ corresponde a los tiempos por ocurrencia (en teoría, $\mathcal{O}((m + \log n)m \log r \log\log r + occ \log n \log\log r)/occ)$, mientras que el eje $X$ corresponde a la cantidad $occ$ de ocurrencias detectadas.

La estructura toma tiempos menores por ocurrencia mientras más ocurrencias del patrón se encuentren en el texto. Patrones de menos largo aparecen con más incidencia en los textos y por lo tanto el proceso de dividir el patrón en todos sus posible sufijos y prefijos es más

Figura 5.6: Tiempos por ocurrencia en $\mu s$ (microsegundos) de búsquedas en función del número de ocurrencias para distintas colecciones repetitivas. Ambos ejes están es escala logarítmica

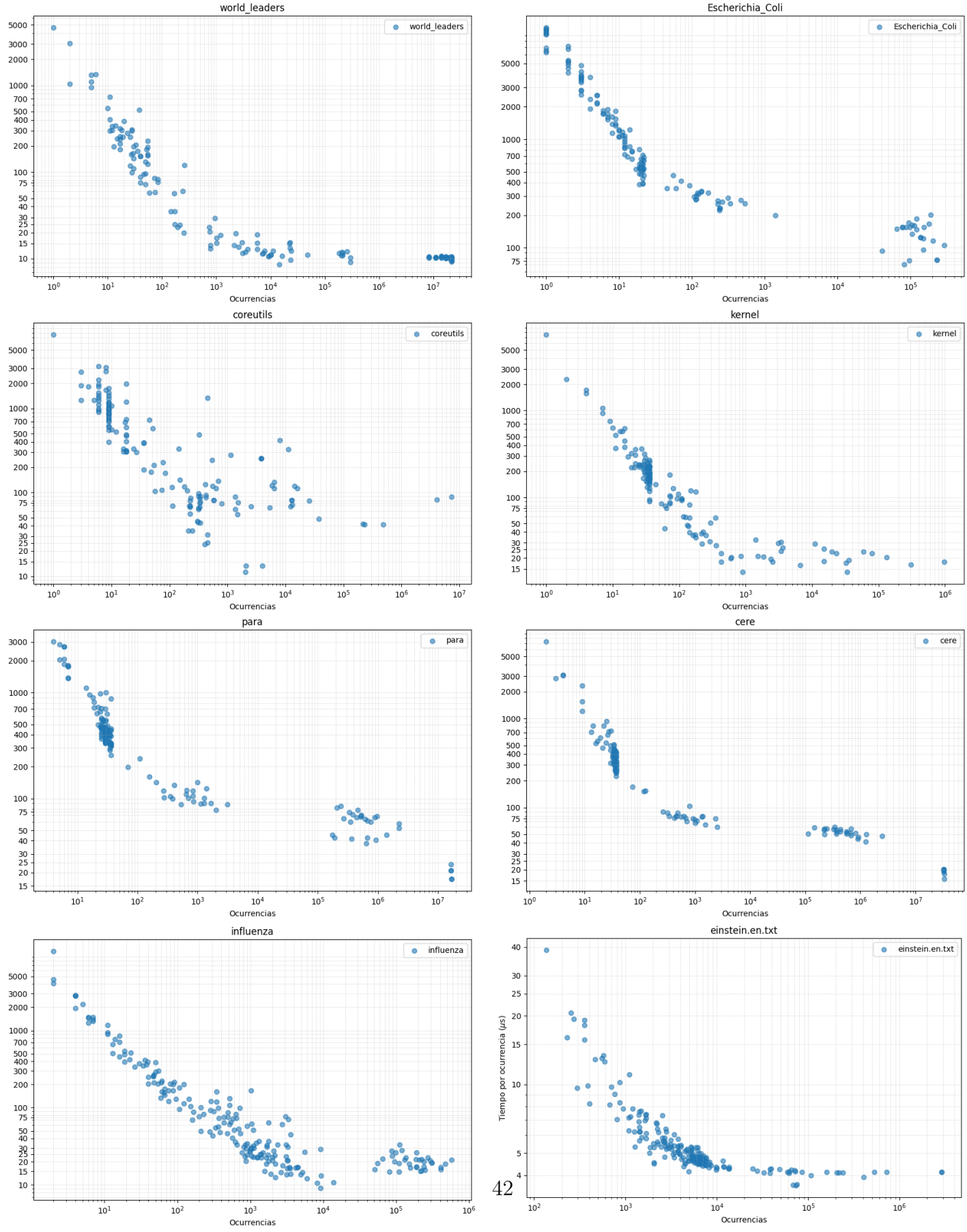
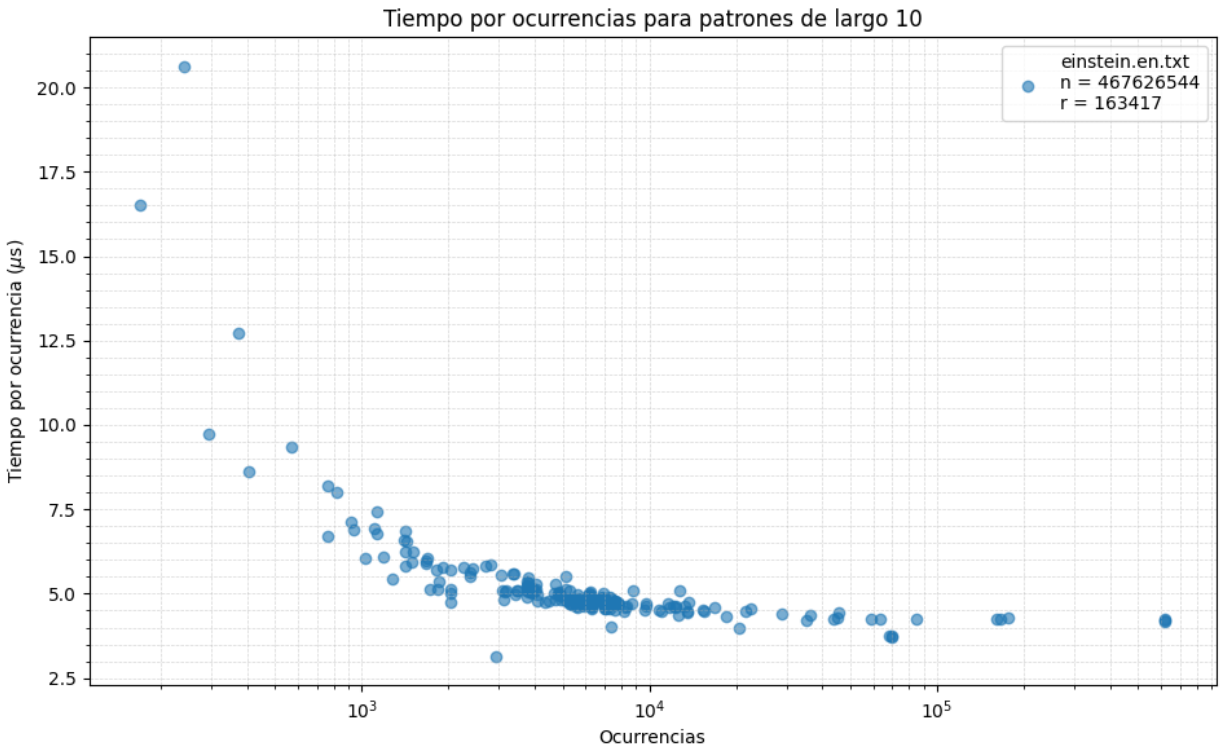


Figura 5.7: Tiempos por ocurrencia en $\mu s$ (microsegundos) de búsquedas de patrones aleatorios de largo fijo 10 en función del número de ocurrencias en la colección *einstein.en*. El eje X está en escala logarítmica




corto.

En colecciones altamente repetitivas como *einstein.en* logra tiempos de búsqueda por ocurrencia bajo 5 $\mu s$ cuando la cantidad de ocurrencias es del orden de $10^4$ o mayor. En *world_leaders* el tiempo por ocurrencia se estabiliza en 9 a 10 $\mu s$. En otras colecciones el tiempo por ocurrencia es menos estable: en *influenza* altas ocurrencias tienen tiempo por ocurrencia entre 5 a 30 $\mu s$.

Se analizó el caso para patrones aleatorios de largo fijo $m = 10$ sobre la colección *einstein.en*. Los resultados se muestran en la figura 5.7 donde la búsqueda alcanza tiempos estables de 5 $\mu s$ por ocurrencia para patrones con ocurrencias superiores a $10^3$.

## 5.4. Análisis comparativo con la solución lineal de búsqueda sin compresión

El reporte de ocurrencia de patrones utilizando un algoritmo lineal en un texto sin comprimir tiene un tiempo de búsqueda en el peor caso de $\mathcal{O}(nm)$, con $n$ el largo del texto y $m$ el largo del patrón. En un texto real sin embargo, el tiempo es más cercano a $\mathcal{O}(n)$, pues la gran mayoría de las comparaciones terminan en el primer símbolo, es decir, el tiempo es

relativamente independiente al largo del patrón.

| **n** | **r** | **m** | **occ** | **t(ms) estructura** | t(ms) lineal | t(ms) teórico |
|---|---|---|---|---|---|---|
| 111299 | 26149 | 5 | 14 | **4.34** | 10.85 | 10.08 |
| 111299 | 26149 | 10 | 1 | **4.36** | 10.76 | 10.17 |
| 111299 | 26149 | 20 | 1 | **5.41** | 15.76 | 10.49 |
| 111299 | 26149 | 30 | 1 | **6.14** | 15.74 | 10.98 |
| 111299 | 26149 | 40 | 1 | **7.11** | 15.77 | 11.61 |
| 111299 | 26149 | 50 | 1 | **7.85** | 15.88 | 12.41 |
| 111299 | 26149 | 70 | 1 | **9.81** | 16.16 | 14.46 |
| 111299 | 26149 | 100 | 1 | **11.72** | 15.72 | 18.72 |
| 290801 | 54170 | 5 | 93 | **7.88** | 28.77 | 10.17 |
| 290801 | 54170 | 10 | 3 | **6.37** | 28.68 | 10.20 |
| 290801 | 54170 | 20 | 1 | **7.18** | 42.79 | 10.56 |
| 290801 | 54170 | 30 | 1 | **8.15** | 41.55 | 11.10 |
| 290801 | 54170 | 40 | 1 | **9.24** | 42.41 | 11.81 |
| 290801 | 54170 | 50 | 1 | **10.25** | 42.17 | 12.70 |
| 290801 | 54170 | 70 | 1 | **12.41** | 41.76 | 14.98 |
| 290801 | 54170 | 100 | 1 | **15.33** | 41.75 | 19.71 |
| 704731 | 119734 | 5 | 640 | **21.11** | 69.06 | 10.79 |
| 704731 | 119734 | 10 | 216 | **12.15** | 68.66 | 10.46 |
| 704731 | 119734 | 20 | 75 | **10.71** | 99.65 | 10.72 |
| 704731 | 119734 | 30 | 1 | **9.41** | 99.67 | 11.24 |
| 704731 | 119734 | 40 | 1 | **10.68** | 99.75 | 12.04 |
| 704731 | 119734 | 50 | 1 | **11.78** | 99.69 | 13.02 |
| 704731 | 119734 | 70 | 1 | **13.63** | 99.64 | 15.56 |
| 704731 | 119734 | 100 | 1 | **16.94** | 99.26 | 20.80 |
| 1224377 | 191364 | 5 | 329 | **17.24** | 118.98 | 10.48 |
| 1224377 | 191364 | 10 | 10 | **9.06** | 118.91 | 10.25 |
| 1224377 | 191364 | 20 | 1 | **9.48** | 172.84 | 10.69 |
| 1224377 | 191364 | 30 | 1 | **10.55** | 172.58 | 11.33 |
| 1224377 | 191364 | 40 | 1 | **11.90** | 173.20 | 12.18 |
| 1224377 | 191364 | 50 | 1 | **13.23** | 173.08 | 13.22 |
| 1224377 | 191364 | 70 | 1 | **16.16** | 172.97 | 15.92 |
| 1224377 | 191364 | 100 | 1 | **19.24** | 172.31 | 21.47 |
| 2205984 | 333238 | 5 | 409 | **31.03** | 221.59 | 10.61 |
| 2205984 | 333238 | 10 | 11 | **10.39** | 222.08 | 10.28 |
| 2205984 | 333238 | 20 | 1 | **11.35** | 327.83 | 10.74 |
| 2205984 | 333238 | 30 | 1 | **11.98** | 320.78 | 11.43 |
| 2205984 | 333238 | 40 | 1 | **13.65** | 323.65 | 12.34 |
| 2205984 | 333238 | 50 | 1 | **14.92** | 323.42 | 13.46 |
| 2205984 | 333238 | 70 | 1 | **17.48** | 317.41 | 16.34 |
| 2205984 | 333238 | 100 | 1 | **22.68** | 326.57 | 22.27 |

Tabla 5.2: Tiempos de búsqueda en textos reales de largo $n$ de patrones aleatorios de largo $m$, con *occ* ocurrencias en promedio y con $r$ la cantidad de reglas

El costo temporal teórico de la estructura en reportar *occ* ocurrencias, considerando $r$ como el número de reglas al comprimir el texto por gramática es:

$$\mathcal{O}((m + \log n)m \log r \log\log r + occ \log n \log\log r)$$

Para que lo anterior sea menor al tiempo de búsqueda lineal, con un patrón de largo $m < \log n$ se debe cumplir que:

$$occ = o(\frac{n}{\log n \log\log r})$$

Los valores de *occ* y $r$ serán pequeños si el texto es repetitivo y las ocurrencias del patrón de búsqueda son pocas. Este es el caso para textos reales como se aprecia en la tabla 5.2. Con largos de patrones suficientemente largos, las ocurrencias en textos reales tienden a ser únicas. La naturaleza de los textos reales hace que la "densidad"de repetitividad sea relativamente constante, y por lo tanto, la cantidad de ocurrencias de patrones crece más lento que el largo del texto. Esto implica que el algoritmo termina venciendo con más holgura a la búsqueda lineal mientras más largo sea el texto, al menos en el contexto de textos reales como lo son las novelas, los ensayos, *etc.*

Con $r$ pequeño la grilla es pequeña y por lo tanto las búsquedas de rango son más cortas, además el árbol sintáctico es más pequeño y por lo tanto las búsquedas de ocurrencias hasta la raíz son más cortas. Con *occ* pequeño la cantidad de búsquedas de ocurrencias son menores. Las ocurrencias *occ* suelen ser proporcionales, tomando patrones aleatorios de un largo fijo, a qué tan repetitivo es el texto, es decir, en general, $occ \propto r^{-1}$.

Para ejemplificar lo anterior, considere el siguiente texto $T$:

`"aaaaaaaaaabaaaaaaaaaab...aaaaaaaaaab\n..."`

Esto es, un texto de muchas líneas donde cada línea corresponde a la repetición de un patrón de cierta cantidad de $a$ seguidas de una $b$. El texto es muy repetitivo, como se aprecia en la tabla 5.3 por la pequeña cantidad de reglas, y si se buscan patrones de la forma $a^ib$`\n` que son los más raros, el buscador de la estructura es más rápido que la búsqueda lineal:

| **n** | **r** | **m** | **occ** | **t(ms) estructura** | t(ms) lineal | t(ms) teórico |
|---|---|---|---|---|---|---|
| 69120 | 22 | 5 | 720 | 1.89 | 6.80 | 10.31 |
| 69120 | 22 | 10 | 720 | 2.59 | 6.85 | 10.32 |
| 69120 | 22 | 20 | 720 | 3.73 | 10.12 | 10.37 |
| 69120 | 22 | 30 | 720 | 3.65 | 9.97 | 10.44 |
| 69120 | 22 | 40 | 720 | 3.45 | 9.85 | 10.54 |
| 69120 | 22 | 50 | 720 | 4.90 | 9.83 | 10.65 |
| 69120 | 22 | 70 | 720 | 6.25 | 9.91 | 10.96 |

Tabla 5.3: Tiempos de búsqueda en texto repetitivo con $n$ el largo del texto original, $r$ la cantidad de reglas, $m$ el largo del patrón.

Considérese un ejemplo más realista. Se tiene una secuencia de ADN como un texto donde el alfabeto es $A, T, G, C$. Estas secuencias son sumamente largas, y buscar una cadena específica en esta usando una búsqueda lineal puede ser muy lento, en especial si se requiere

repetir el proceso varias veces con distintos patrones de pocas ocurrencias. En este caso, la estructura tiene un tiempo de búsqueda más corto que la búsqueda lineal, como se aprecia en la tabla 5.4.

| **n** | **r** | **m** | **occ** | **t(ms) estructura** | t(ms) lineal | t(ms) teórico |
|---|---|---|---|---|---|---|
| 1000000 | 182668 | 5 | 975 | 83.17 | 98.23 | 11.21 |
| 1000000 | 182668 | 10 | 2 | 10.49 | 98.78 | 10.24 |
| 1000000 | 182668 | 20 | 1 | 9.79 | 142.39 | 10.68 |
| 1000000 | 182668 | 30 | 1 | 11.17 | 142.58 | 11.32 |
| 1000000 | 182668 | 40 | 1 | 12.77 | 141.67 | 12.16 |
| 1000000 | 182668 | 50 | 1 | 14.05 | 141.11 | 13.19 |
| 1000000 | 182668 | 70 | 1 | 16.73 | 140.73 | 15.87 |
| 1000000 | 182668 | 100 | 1 | 20.58 | 140.75 | 21.39 |

Tabla 5.4: Tiempos de búsqueda en secuencia de ADN de largo 100.000, *occ* es la cantidad de ocurrencias del patrón

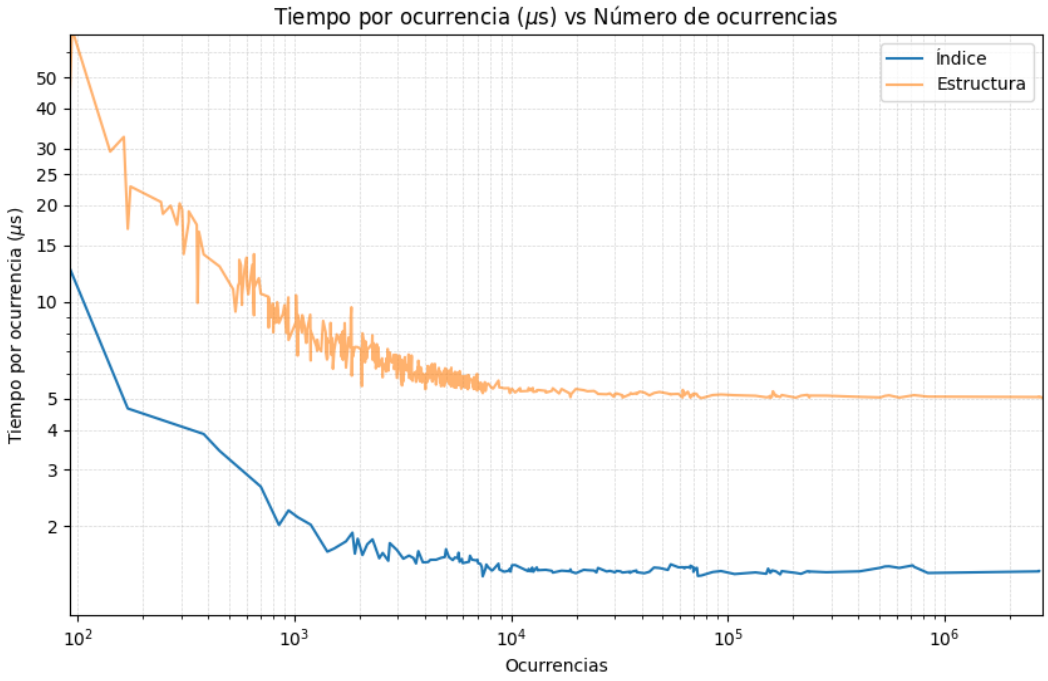
## 5.5. Análisis comparativo con índice comprimido basado en gramática

El índice comprimido basado en gramática[4] es una estructura de datos que permite buscar patrones en un texto comprimido por gramática libre de contexto generada por este, representada esta a su vez como un árbol. La figura 5.8 muestra los tiempos de búsqueda de patrones aleatorios en textos reales con la estructura propuesta en este trabajo y la estructura del índice comprimido basado en gramática. La tabla 5.5 en tanto muestra la cantidad de bits por símbolo de cada colección para ambas estructuras.

| **Colección** | *bps* **Estructura** | *bps* **Índice g-index/2** | *bps* **Índice g-ingex/32** |
|---|---|---|---|
| world_leaders | 0.848511 | 0.814413 | 0.642726 |
| Escherichia_Coli | 4.55733 | | |
| influenza | 1.40496 | 1.29044 | 1.02916 |
| kernel | 0.619162 | 0.562662 | 0.444946 |
| coreutils | 1.36035 | | |
| para | 1.41742 | | |
| cere | 0.986939 | 0.9352 | 0.748654 |
| einstein.en | 0.0475765 | 0.0456824 | 0.0355039 |

Tabla 5.5: Propiedades de cada colección

Figura 5.8: Tiempos de búsqueda en $\mu s$ (microsegundos) de búsquedas de patrones aleatorios en función del número de ocurrencias en la colección *einstein.en* para la estructura implementada y el índice comprimido con g-index/2.



# Capítulo 6

# Conclusiones

## 6.1. Conclusiones generales

### 6.1.1. Objetivo general

Finalizado este trabajo, se puede concluir con suficiente certeza que los objetivos propuestos, esto es, la correcta implementación de la estructura y el análisis empírico de su funcionamiento, fueron completados satisfactoriamente. La estructura se comporta en la práctica como lo teorizado.

En textos reales, la estructura comprimida simplificada para indexar texto basada en gramática logra el reporte de las posiciones de las ocurrencias de patrones de búsqueda en tiempos más cortos que la búsqueda lineal de patrones ($\mathcal{O}((m + \log n)m \log r \log\log r + occ \log n \log\log r)$ versus $\mathcal{O}(nm)$). El espacio usado es de orden similar al texto original, pues a pesar de que se comprime a una cantidad $r$ de reglas que es menor al largo $n$ del texto, estas reglas requieren más memoria para ser guardadas ($\log r$ bits por cada regla). Sin embargo, textos suficientemente repetitivo logra una compresión significativa, y se benefician de una velocidad de reporte de ocurrencias aún mayor.

En particular, si la cantidad de ocurrencias del patrón es muy pequeña en comparación al tamaño del texto, y el texto en sí es suficientemente repetitivo, la búsqueda es ordenes de magnitud más rápida que una búsqueda lineal. Textos altamente repetitivo son también comprimidos de forma significativa, por ejemplo, los textos correspondientes a las colecciones repetitivas analizados en 5.3

Los tiempos de búsqueda por patrón mejoran enormemente con la cantidad de ocurrencias de un patrón, y esto es consistente con lo esperado. Con respecto al estado del arte, en las colecciones repetitivas evaluadas, los tiempos de búsqueda por ocurrencia son mayores (alrededor de 4 microsegundos más) que los tiempos por ocurrencia de el índice comprimido basado en gramática[4]. Futuras optimizaciones en la implementación podrían mejorar este aspecto y equiparar los tiempos de búsqueda de la estructura.

## 6.2. Cumplimiento de objetivos específicos

1. Implementación la estructura de forma correcta: La implementación de la estructura fue realizada de forma correcta lo que permitió el correcto análisis del comportamiento de esta tanto en la construcción como en la búsqueda de patrones. Se implementaron cada una de las partes de la estructura de forma modular y se realizaron pruebas unitarias para garantizar la corrección de las operaciones.

2. Implementación de pruebas de robustez y consistencia de la estructura: Se implementaron pruebas de robustez y consistencia de la estructura, las cuales permitieron validar su correcto funcionamiento y su congruencia con el análisis teórico. Pruebas unitarias y de integración fueron realizadas para garantizar la corrección de las operaciones y la funcionalidad de la estructura.

3. Implementación de pruebas de desempeño espacial y temporal de la implementación: Se realizaron pruebas de desempeño espacial y temporal de la implementación, las cuales permitieron evaluar su eficiencia en términos de tiempo y espacio. A partir de estas pruebas se obtuvieron datos cuantitativos sobre el desempeño de la estructura y se pudo visualizar a través de gráficos y tablas la efectividad de la estructura en la búsqueda de patrones.

4. Análisis de los resultados de las pruebas para obtener conclusiones respecto al desempeño: Se analizaron los resultados de las pruebas para obtener conclusiones respecto al desempeño empírico de la estructura, identificando sus fortalezas y debilidades. Con esto se obtuvo una visión clara de la utilidad de la estructura en un contexto real y las posibles mejoras a los tiempos de búsqueda en ámbitos de eficiencia y compresión efectiva en textos reales y/o repetitivos.

## 6.3. Trabajo futuro

### 6.3.1. Memoizar

No obstante las virtudes de la estructura, esta requiere un tiempo de construcción no menospreciable. Si se utiliza extra memoria es posible aplicar memoización (regla $\longrightarrow$ expansión) para acelerar el proceso de ordenamiento de las reglas por sus expansiones y así disminuir el tiempo de comparación y por consiguiente construcción, pero esto requiera memoria extra durante el proceso equivalente al mismo texto, es decir, $\mathcal{O}(n \log \sigma)$ bits, con lo cual no hay compresión.

Se puede limitar la memoización a sólo las reglas originalmente creadas por *Re-Pair* y aprovechar que la estructura del árbol gramatical está balanceada desde el nivel correspondiente a los sub-árboles que salen de tomar pares de símbolos de la secuencia $C$.

Es posible también aplicar memoización en la búsqueda de ocurrencias secundarias, lo cual reduciría enormemente el tiempo de búsqueda de patrones. Esto requeriría, en el peor caso, memoria de ejecución extra $\mathcal{O}(r \log r)$, pero evitaría re-calcular las ocurrencias de cada

regla en el símbolo inicial. Si se considera la búsqueda de ocurrencias como recorrer el árbol sintáctico desde cada nodo equivalente a la regla, este proceso de memoización permite evitar recorrer los mismos nodos más de una vez.

### 6.3.2. Sobre la secuencia $R$

La estructura expande la secuencia $R$ obtenida de *Re-Pair* con el fin de eliminar la secuencia $C$. Esto implica extender $R$ con extras $\mathcal{O}(|C|)$ reglas. Es posible hacer este proceso de extender $R$ de una forma puramente virtual, manteniendo $C$ y $R$ originales. En efecto, el proceso de extender $R$ es equivalente a construir un árbol binario con $C$ como las hojas del árbol. Con esto, si el programa requiere una regla específica de la secuencia virtual $R'$ como $R$ expandida, es fácil saber la posición de esta regla en este árbol virtual, y con eso, se puede saber con exactitud el rango en $C$ que corresponde a la regla (si la regla corresponde a las creadas en la expansión).

La expresión para obtener el rango de $C$ que le corresponde a una regla $R_i$ no es simple pero se puede obtener, pues la estructura del árbol virtual es conocida: Las reglas se crean a partir de $C$ tomando, en cada iteración, pares de símbolos de izquierda a derecha, reemplazándolos por un nuevo símbolo, dejando símbolos sin par para la siguiente iteración, y así hasta reducir $C$ a un solo símbolo. Así, en casos donde el largo de $C$ no es una potencia de 2 las posiciones de las reglas son aún calculables.

Cuando la estructura implementada requiera reordenar $R'$ por su expansión izquierda invertida por orden lexicográfico, basta con traducir este reordenamiento a la secuencia $R$ original y $C$.

Lo anterior permite expandir estas reglas extras en tiempo $\mathcal{O}(h_i)$, donde $h_i$ es la altura de la regla en el árbol, lo que implica que expandir todas estas reglas extras, utilizando la técnica descrita, tiene un costo total de tiempo $\mathcal{O}(n)$ y espacio equivalente a la expansión de las reglas originales $\mathcal{O}(n)$.

Si se añade memoización sobre las reglas originales, expandir cualquier regla extra toma tiempo constante por cada regla que pertenece al rango en $C$ correspondiente.

### 6.3.3. Potencial paralelismo

Es posible también utilizar múltiples *threads* o multihilos en ciertas partes del programa en donde el paralelismo podría mejorar considerablemente la búsqueda. La simplicidad de la estructura facilita el paralelismo en, por ejemplo, los dos ordenamientos de las reglas por sus expansiones, las múltiples divisiones del patrón en sus sufijos y prefijos, las cuatro búsquedas binarias para encontrar los rangos de estas divisiones, y las múltiples búsquedas para cada ocurrencia encontrada en la grilla. En teoría, y con suficientes hilos, se puede eliminar el largo del patrón y las ocurrencias. como factores en los tiempos de búsqueda. Utilizando solo 4 hilos, el tiempo de búsqueda puede ser reducido significativamente, logrando, en teoría, compararse al tiempo de búsqueda del índice comprimido basado en gramática[4].

# Bibliografía


[1] Manuel Baena, Travis Gagie, Gonzalo Navarro, et al. Pizza & chili highly repetitive corpus. `https://pizzachili.dcc.uchile.cl/repcorpus/real/`, 2009. A collection of highly repetitive text datasets for benchmarking pattern search algorithms and compressed data structures.

[2] Moses Charikar, Eric Lehman, Ding Liu, Rina Panigrahy, Manoj Prabhakaran, Amit Sahai, and Abhi Shelat. The smallest grammar problem. *IEEE Transactions on Information Theory*, 51(7):2554–2576, 2005.

[3] Francisco Claude and Gonzalo Navarro. Improved grammar-based compressed indexes. In *Proc. 19th International Symposium on String Processing and Information Retrieval (SPIRE)*, volume 7608 of *Lecture Notes in Computer Science*, pages 180–192, 2012.

[4] Francisco Claude, Gonzalo Navarro, and Alejandro Pacheco. Grammar-compressed indexes with logarithmic search time. `https://arxiv.org/abs/2004.01032`, 2020.

[5] Craig M. Cook, Azriel Rosenfeld, and Alan R. Aronson. Grammatical inference by hill climbing. *Information Sciences*, 10(2):59–80, 1976.

[6] Robert M. Corless, David J. Jeffrey, and Donald E. Knuth. A sequence of series for the lambert W function. In *International Symposium on Symbolic and Algebraic Computation, ISSAC '97, Maui, Hawaii, USA, July 21–23, 1997*, pages 197–204, 1997.

[7] Michael Hart (Founder). Project gutenberg, a library of over 70,000 free ebooks. `https://www.gutenberg.org/`. Último acceso: 2024, 5 de Diciembre.

[8] Cristóbal Fuentes. Simple text indexing based on grammar. `https://github.com/solzhen/SimpleTextIndexingBasedOnGrammar`, 2024. Accessed: 2024-12-15.

[9] Simon Gog. Succinct data structure library (sdsl-lite). `https://github.com/simongog/sdsl-lite`. Último acceso: 2024, 20 de Noviembre.

[10] Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practice: Plug and play with succinct data structures. In *13th International Symposium on Experimental Algorithms, (SEA 2014)*, pages 326–337, 2014.

[11] J.C. Kieffer and En-Hui Yang. Grammar-based codes: a new class of universal lossless source codes. *IEEE Transactions on Information Theory*, 46(3):737–754, 2000.

[12] Niklas Larsson and Alistair Moffat. Off-line dictionary-based compression. *IEEE Transactions on Computers*, 49(11):1196–1210, 2000.

[13] Shirou Maruyama. A grammar-based compressor by most-frequent-first substitution. `https://code.google.com/archive/p/re-pair/`. Último acceso: 2024, 1 de Agosto.

[14] Gonzalo Navarro. Re-pair compression and decompression (2010). `https://users.dcc.uchile.cl/~gnavarro/software/index.html`. Último acceso: 2024, 20 de Noviembre.

[15] Gonzalo Navarro. *Compact Data Structures - A Practical Approach*. Cambridge University Press, 2016.

[16] Craig G. Nevill-Manning and Ian H. Witten. Identifying hierarchical structure in sequences: A linear-time algorithm. *Journal of Artificial Intelligence Research*, 7:67–82, 1997.

[17] Peter Norvig. English letter frequency counts: Mayzner revisited or etaoin srhldcu. `http://norvig.com/mayzner.html`. Último acceso: 2024, 5 de Diciembre.

[18] Catch Org. Catch2. `https://github.com/catchorg/Catch2`. Último acceso: 2024, 20 de Noviembre.

[19] The Raven. *Edgar Allan Poe*. Harper & Brothers, 1884.

[20] Wojciech Rytter. Application of lempel-ziv factorization to the approximation of grammar-based compression. *Theoretical Computer Science*, 302(1-3):211–222, 2003.

[21] We Are Social. Informe digital global, abril 2024. `https://wearesocial.com/es/blog/2024/04/informe-digital-global-abril-2024/`, 2024. Accessed: December 17, 2024.

[22] Statista. Volume of data/information created, captured, copied, and consumed worldwide from 2010 to 2023, with forecasts from 2024 to 2028 (in zettabytes). `https://www.statista.com/statistics/871513/worldwide-data-created/`, 2024. Accessed: December 17, 2024.

[23] Yuto Tabei, Yasuo Takabatake, and Hideo Sakamoto. A succinct grammar compression. In *Proc. 24th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 7922 of *Lecture Notes in Computer Science*, pages 235–246, 2013.

[24] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

# Apéndice A

# Anexo

Listing A.1: Llamando Re-Pair

```
FILE *input;
DICT *dict;
input  = fopen(input_filename.c_str(), "rb");
dict = RunRepair(input);
fclose(input);
RULE *rules = dict->rule; // set or rules
CODE *comp_seq = dict->comp_seq; // sequence C
```

Listing A.2: Añadir más reglas hasta eliminar C

```
while (dict->seq_len > 1) {
    for (u_int i = 0; i < dict->seq_len; i = i+2) {
        if (i == dict->seq_len - 1) { // odd case
            comp_seq[i/2] = comp_seq[i];
        }
        else {
            RULE new_rule;
            new_rule.left = comp_seq[i];
            new_rule.right = comp_seq[i+1];
            rules[dict->num_rules] = new_rule; // append new rule
            comp_seq[i/2] = dict->num_rules; // update sequence C
            dict->num_rules++;
        }
    }
    dict->seq_len = dict->seq_len % 2 == 0 ? dict->seq_len / 2 :
       dict->seq_len / 2 + 1;
}
```

Listing A.3: Normalizar Secuencia

```
bit_vector bbbb(257, 0); // bit vector to mark which symbols are in
   the alphabet used by text
int_vector<> sequenceR((dict->num_rules - 257) * 2, 0, sizeof(CODE) *
   8);
```

```
for (u_int i = 0; i < sequenceR.size(); i = i + 2) {
    sequenceR[i] = rules[i/2 + 257].left;
    sequenceR[i + 1] = rules[i/2 + 257].right;
    if (sequenceR[i] <= 256) {
        bbbb[sequenceR[i]] = 1;
    }
    if (sequenceR[i + 1] <= 256) {
        bbbb[sequenceR[i + 1]] = 1;
    }
}
rank_support_v<1> rank_bbbb(&bbbb);
select_support_mcl<1, 1> select_bbbb(&bbbb);
vector<char> rank(257, 0);
vector<char> select(257, 0);
for (int i = 0; i < 257; i++) {
    rank[i] = rank_bbbb(i);
    if (i==0) continue;
    select[i] = select_bbbb(i);
}
u_int max_terminal = 0;
for (u_int i = 1; i <= rank_bbbb(257); i++) {
    if (select_bbbb(i) > max_terminal) {
        max_terminal = select_bbbb(i);
    }
}
int_vector<> normalized_sequenceR(sequenceR.size(), 0, sizeof(CODE) *
   8);
int sz = sequenceR.size();
int r;
int max_normalized = 0; // maximum symbol in the normalized alphabet
for (int i = 0; i < sz; i++) {
    if (sequenceR[i] < 256)
        r = rank_bbbb(sequenceR[i] + 1) - 1;
    else
        r = sequenceR[i] - 257 + rank_bbbb(257);
    normalized_sequenceR[i] = r;
    if (r > max_normalized) {
        max_normalized = r;
    }
}
```

Listing A.4: Permutaciones utilizando vectores de bit

```
typedef struct abv {
    bit_vector b;
    rank_support_v<0> rank;
    select_support_mcl<1, 1> sel_1;
    select_support_mcl<0, 1> sel_0;
} abv; // rank, selects vector
typedef struct dbv {
```

```
    bit_vector b;
    select_support_mcl<1, 1> sel_1;
    select_support_mcl<0, 1> sel_0;
} dbv; // selects vector

class Permutation {
    friend class PowerPermutation;
protected:
    int rank_b(int i);
    int_vector<> pi; // permutation
    int_vector<> S; // shortcuts
    brv b; // bit vector to mark shortcuts
public:
    Permutation();
    int t; // parameter t

    /// @brief Sole constructor, it does not check if pi is a
       permutation
    /// @param pi vector of integers (initially, 8 bit long integers)
    /// @param t parameter t length of the shortcuts
    Permutation(int_vector<> pi, int t);

    /// @brief Return the position of the element i after applying the
       permutation
    int operator[](int i);
    int permute(int i) { return this->operator[](i); };

    /// @brief Return the inverse of the permutation,
    /// that is, the position j such that permutation ( j ) = i
    /// @param i
    /// @return
    int inverse(int i);
};
```

Listing A.5: Secuencia utilizando permutaciones

```
class ARSSequence {
private:
    vector<abv> A;
    vector<dbv> D;
    vector<Permutation> pi;
    int sigma; int n;
    int select_1_D(int k, int i);
    int select_0_D(int k, int i);
    int select_0_A(int k, int i);
    int select_1_A(int k, int i);
    int rank_A(int c, int i);
    int pred_0_A(int c, int s);
public:
    /// @brief Builds structure to support rank, select and access
```

```
        queries
    /// @param S integer vector representing the sequence
    /// @param sigma size of alphabet [0 . . . sigma)
    ARSSequence(int_vector<> S, int sigma);
    /// @brief Access query
    /// @param i position in the sequence
    /// @return The symbol at position i
    int access(int i);
    int operator[](int i) { return access(i); };
    /// @brief Rank query
    /// @param c symbol in the alphabet
    /// @param i position in the sequence
    /// @return The number of occurrences of c in the sequence up to
        and including position i
    int rank(int c, int i);
    /// @brief Select query
    /// @param c symbol in the alphabet
    /// @param i the i-th occurrence of c in the sequence
    /// @return The position of the i-th occurrence of c in the
        sequence
    /// @note Position returned is 0-indexed, while parameter i is
        1-indexed as ordinal numbers are.
    int select(int c, int j);
    u_int size() { return n; }
};
```

Listing A.6: Matriz *Wavelet*

```
class WaveletMatrix {
private:
    u32 sigma; // highest symbol in the alphabet
    vector<u32> z; // right child pointer
    void build(vector<u32>& S, u32 n, u32 sigma);
    u32 select(u32 l, u32 p, u32 a, u32 b, u32 c, u32 j);
    vector<ppbv> bm; // bit matrix seen as vector of preprocessed bit
        vectors
public:
    /// @brief A wavelet matrix using bit_vectors over an alphabet [1,
        sigma]
    /// @param s 4-byte long unsigned integer vector
    /// @param sigma highest numerical symbol
    WaveletMatrix(vector<u32>& s, u32 sigma);
    WaveletMatrix();
    /// @brief Access the number in the i-th zero-indexed
    /// position of the original sequence.
    /// @param i positive 0-indexed position.
    /// @return The number or NULL if out of bounds
    u32 access(u32 i);
    /// @brief Counts the occurences of number c up until yet
        excluding the given zero-indexed position i
```

```
    /// @param i the zero-indexed position
    /// @param c the number
    /// @return the number of occurences until position i
    u32 rank(u32 c, u32 i);
    /// @brief Returns the 0-indexed position of the j-th occurence of
       the number c
    /// @param c a number
    /// @param j a positive number
    /// @return the position or the size of the sequence if not found,
       or -1 if c is not in the sequence
    u32 select(u32 c, u32 j);
    void printself();
    ppbv operator[](u32 level);
    u32 offset(u32 level);
    u32 size() { return bm[0].size(); }
};
```

Listing A.7: Grilla

```
class Grid {
private:
    u32 c; // number of columns
    u32 r; // number of rows
    u32 n; // nubmer of points
    WaveletMatrix wt; // Wavelet tree
    u32 count(u32 x_1, u32 x_2, u32 y_1, u32 y_2, u32 l, u32 a, u32 b);
    vector<Point> report(u32 x_1, u32 x_2, u32 y_1, u32 y_2, u32 l,
       u32 a, u32 b);
    u32 outputx(u32 level, u32 x);
    u32 outputy(u32 level, u32 a, u32 b, u32 i);
public:
    /// @brief Construct a grid from a binary file
    /// @param fn Filename of the binary file
    /// @note The binary file should contain the dimensions of the
       grid first
    /// (columns, rows), followed by the points as pairs of integers.
       Every integer
    /// in the file should be a 4-byte long unsigned integer
       (uint32_t).
    /// The coordinates should be 0-indexed.
    Grid(const string& fn);
    Grid(std::vector<Point>& points, u32 columns, u32 rows);
    /// @brief Count the number of points in the grid that are within
       the rectangle
    /// @param x_1 1-indexed column range start
    /// @param x_2 1-indexed column range end
    /// @param y_1 1-indexed row range start
    /// @param y_2 1-indexed row range end
    /// @return The number of points in the grid that are
    /// within the rectangle as an integer
```

```
    u32 count(u32 x_1, u32 x_2, u32 y_1, u32 y_2);
    /// @brief Report the points in the grid that are within the
       rectangle
    /// @param x_1 1-indexed column range start
    /// @param x_2 1-indexed column range end
    /// @param y_1 1-indexed row range start
    /// @param y_2 1-indexed row range end
    /// @return A vector of points that are within the rectangle
    vector<Point> report(u32 x_1, u32 x_2, u32 y_1, u32 y_2);
    void printself();
    u32 getColumns() { return c; }
    u32 getRows() { return r; }
    WaveletMatrix getWaveletMatrix() { return wt; }
    /// @brief Access the number in the i-th 1-indexed position
    /// @param i
    /// @return
    u32 access(u32 i) {return wt.access(i-1);};
    /// @brief Returns the 1-indexed position of the j-th occurrence
       of c
    /// @param j
    /// @param c
    /// @return
    u32 select(u32 j, u32 c) {return wt.select(j, c);};
};
```

Listing A.8: Buscador de patrones

```
class PatternSearcher {
private:
    Grid G; // Grid
    ARSSequence R; // ARS sequence
    u_int S; // Initial symbol
    int_vector<> l; // Lengths of the expansion of the rules
    uint nt; // Number of terminals
    vector<char> sl; // select vector for normalized alphabet
    vector<char> rk; // rank vector for normalized alphabet
    string expandRule( int i, unordered_map<int, string>& memo);
    string expandRightSideRule(int i, unordered_map<int, string>
       &memo);
    string expandLeftSideRule(int i, unordered_map<int, string>& memo);
    int ruleLength(int i);
    Generator<char> expandRuleLazy( int i, bool rev = false);
    Generator<char> expandRuleSideLazy( int i, bool left = false);
    bool compareRulesLazy(int i, int j, bool rev = false);
    template <typename Iterator>
    int compareRuleWithPatternLazyImpl ( int i, Iterator
       pattern_begin, Iterator pattern_end, bool rev = false);
    int compareRuleWithPatternLazy(int i, string pattern, bool rev =
       false);
    void secondaries(vector<int> *occurences, u_int A_i, u_int
```

```
        offset=0, bool terminal = false);
public:
    PatternSearcher(){};
    /// @brief Construct a pattern searcher from a text file
    /// @param input_filename
    PatternSearcher(string input_filename);
    /// @brief Report all occurences of a pattern in the text
    /// @param occurences Vector to store the occurences
    /// @param P Pattern to search
    void search(vector<int> *occurences, string P);
    int numRules() { return R.size() / 2; }
};
```