



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

1. Introducción	1
1.1. Objetivos	2
1.2. Metodología	3
2. Marco Teórico	4
2.1. Entropía	4
2.1.1. Entropía de orden cero	5
2.1.2. Entropía de orden n	5
2.2. Gramáticas	5
2.3. Memoización	6
2.4. Notación \mathcal{O} Grande	6
2.5. Búsqueda Lineal	6
2.6. Compresión Basada en Gramáticas	7
3. Estado del Arte	8
3.1. Representación de texto como gramática	8
3.1.1. Sequitur	8
3.1.2. Re-Pair	9
3.2. Compresión de gramática	10
3.2.1. Dos árboles <i>LOUDS</i>	10
3.2.2. Índice comprimido basado en gramática	11
3.2.3. Grilla con árboles <i>Wavelet</i>	11

3.3.	Implementaciones existentes	12
3.3.1.	SDSL - <i>Succinct Data Structure Library</i>	12
4.	Trabajo realizado	13
4.1.	Descripción General de la estructura	13
4.2.	Diseño de la implementación	14
4.3.	<i>Re-Pair</i>	14
4.3.1.	Generar reglas extras	15
4.4.	Normalizar secuencia	15
4.5.	Secuencia utilizando permutaciones	16
4.5.1.	Permutaciones	16
4.5.2.	Secuencia	16
4.6.	Reordenar secuencia	17
4.6.1.	Memoización	20
4.7.	Ejemplo práctico	21
4.8.	Grilla	23
4.8.1.	Matrices <i>Wavelet</i>	23
4.8.2.	Preparar puntos para grilla	23
4.9.	Calcular largo de las expansiones de las reglas	24
4.10.	Búsqueda de patrones	25
4.10.1.	Ocurrencias primarias	26
4.10.2.	Ocurrencias secundarias	29
5.	Evaluación	32
5.1.	<i>Unit Testing</i>	32
5.2.	Análisis empírico	33
5.2.1.	Espacio	33
5.2.2.	Tiempo	36

5.3. Análisis en textos altamente repetitivos	42
5.4. Análisis comparativo con la solución lineal de búsqueda sin compresión . . .	44
5.5. Análisis comparativo con índice comprimido basado en gramática	47
6. Conclusiones	48
6.1. Conclusiones generales	48
6.1.1. Objetivo general	48
6.2. Cumplimiento de objetivos específicos	49
6.3. Memoizar	49
6.4. Sobre la secuencia R	49
6.5. Otras mejoras	50
6.5.1. Potencial paralelismo	50
Bibliografía	50
Apéndice A. Anexo	51

Índice de Tablas

5.1.	Propiedades de cada colección	42
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 . . .	45
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.	46
5.4.	Tiempos de búsqueda en secuencia de ADN de largo 100.000, occ es la cantidad de ocurrencias del patrón	47
5.5.	Tiempos de búsqueda para patrones aleatorios de largo m , con occ ocurrencias	47

Índice de Ilustraciones

4.1.	Diagrama UML de la implementación	14
4.2.	Secuencia R y expansión de las reglas R_i , junto con secuencia l , el largo de cada expansión	21
4.3.	Árbol sintáctico para <i>abracadabrabra</i>	22
4.4.	DAG para <i>abracadabrabra</i> , las conexiones que corresponden a reglas que aparecen como lados izquierdos son grises, y las derechas son negras	22
4.5.	Grilla	24
4.6.	Grilla	26
4.7.	Búsqueda en el DAG para nodo R_0	29
4.8.	Búsqueda en el DAG para nodo R_2	30
5.1.	Tiempo de construcción de la estructura en función del número de reglas, comparado a tiempo teórico $\mathcal{O}(n + r \log r \log n)$	37
5.2.	Tiempo de búsqueda en función del largo del patrón para un mismo número de ocurrencias	38
5.3.	Tiempo de búsqueda en función del número de ocurrencias para un patrón de largo dos	39
5.4.	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 $\mathcal{O}((m + \log n)m \log r \log \log r + occ \log n \log r)$	40
5.5.	Tiempos de búsquedas en <i>ms</i> (milisegundos) en función del número de ocurrencias para un patrón de largo 2	41
5.6.	Tiempos de búsquedas en <i>ms</i> (milisegundos) en función del número de ocurrencias para un patrón de largo 2 (Continuación)	42

5.7.	Tiempos por ocurrencia en μs (microsegundos) de búsquedas en función del número de ocurrencias para distintas colecciones repetitivas. Ambos ejes están en escala logarítmica	43
5.8.	Tiempos por ocurrencia en μs (microsegundos) de búsquedas de patrones aleatorios de largo fijo 10 en función del número de ocurrencias en la colección <i>einstein.en</i> . El eje X está en escala logarítmica	44

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[?][?], 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 [?] 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* [?] 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[?]. Claude y Navarro en 2012 propusieron una estructura para la búsqueda de patrones en textos basados en gramática[?]. De esta última se desprende una versión simplificada descrita en *Compact Data Structures* [?, 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 empí-

ricamente 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[?], 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 [?, 1, Capítulo 6.1], y grillas representadas mediante Wavelet Trees [?, 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)[?, 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 $\{x_1, x_2, \dots, x_n\}$ 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 $P(X = 1) = \frac{m}{n}$. Entonces la entropía de orden cero es:

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

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:

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

Donde $P(x_1, \dots, x_n)$ es la probabilidad de ver la secuencia $x_1 \dots x_n$, y $P(x_{n+1}|x_1, \dots, x_n)$ es la probabilidad de ver el símbolo x_{n+1} 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, Σ, P, S) , donde:

- N : Es un conjunto de símbolos no terminales.
- Σ : 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:

$$A_i \rightarrow B_i C_i$$

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 $\mathcal{O}(n)$, 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 $\mathcal{O}(nm)$ 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[?], 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 pérdida 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 [?][?], y además esta gramática nunca es más pequeña que una codificación con LZ77[?] (con una ventana ilimitada) lo que motiva y justifica encontrar y utilizar heurísticas como *Re-Pair*[?] y *Sequitur*[?] 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*[?] 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 las 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	
$S \rightarrow a$	
$S \rightarrow ab$	
$S \rightarrow abr$	
$S \rightarrow abra$	
$S \rightarrow abrac$	
$S \rightarrow abraca$	
$S \rightarrow abracab$	Se repite el par ab !!
$S \rightarrow AracA$ $A \rightarrow ab$	
$S \rightarrow AracAr$ $A \rightarrow ab$	
$S \rightarrow AracAra$ $A \rightarrow ab$	Se repite el par ra !!
$S \rightarrow ABcAB$ $A \rightarrow ab$ $B \rightarrow ra$	Se repite el par AB !!
$S \rightarrow CcC$ $A \rightarrow ab$ $B \rightarrow ra$ $C \rightarrow AB$	
$S \rightarrow CcCc$ $A \rightarrow ab$ $B \rightarrow ra$ $C \rightarrow AB$	Se repite el par Cc !!
$S \rightarrow DD$ $A \rightarrow ab$ $B \rightarrow ra$ $C \rightarrow AB$ $D \rightarrow Cc$	
$S \rightarrow DDa$ $A \rightarrow ab$ $B \rightarrow ra$ $C \rightarrow AB$ $D \rightarrow Cc$	

Gramática	
$S \rightarrow DDab$ $A \rightarrow ab$ $B \rightarrow ra$ $C \rightarrow AB$ $D \rightarrow Cc$	Se repite el par ab !!
$S \rightarrow DDA$ $A \rightarrow ab$ $B \rightarrow ra$ $C \rightarrow AB$ $D \rightarrow Cc$	
$S \rightarrow DDAr$ $A \rightarrow ab$ $B \rightarrow ra$ $C \rightarrow AB$ $D \rightarrow Cc$	
$S \rightarrow DDArA$ $A \rightarrow ab$ $B \rightarrow ra$ $C \rightarrow AB$ $D \rightarrow Cc$	Se repite el par ra !!
$S \rightarrow DDAB$ $A \rightarrow ab$ $B \rightarrow ra$ $C \rightarrow AB$ $D \rightarrow Cc$	Se repite el par AB !!
$S \rightarrow DDC$ $A \rightarrow ab$ $B \rightarrow ra$ $C \rightarrow AB$ $D \rightarrow Cc$	A y B aparecen solo una vez a la derecha de las reglas, por lo tanto se eliminan
$S \rightarrow DDC$ $C \rightarrow abra$ $D \rightarrow Cc$	Gramática final

3.1.2. Re-Pair

El algoritmo *Re-Pair*[?] (*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, reemplazar 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 de 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 $S \rightarrow C$, 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[?]. La idea es representar el grafo dirigido acíclico generado por la gramática donde cada regla $A \rightarrow BC$ 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 $A \rightarrow B$ 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 $A \rightarrow BC$, 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 árboles *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[?] 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*[?, 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[?] 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 árboles *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[?] utiliza matrices *wavelet*, que debieron ser implementadas como parte del trabajo realizado.

Otras librerías

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

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)* [?, Capítulo 10.5.6]. La implementación se encuentra disponible en el repositorio *SimpleTextIndexingBasedOnGrammar* [?].

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.

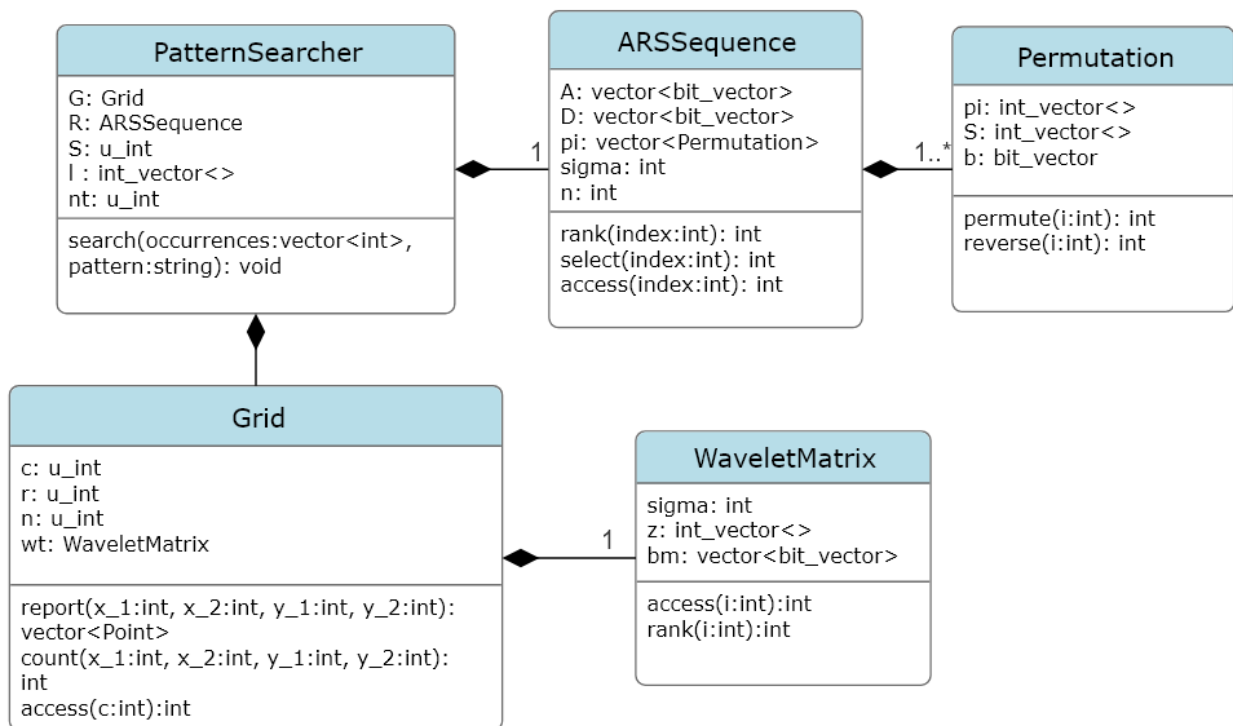
Las partes particulares y el detalle del proceso de búsqueda se explican en el presente texto, en las siguientes secciones.

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[?] que utiliza las estructuras propuestas en la publicación original de Larsson & Moffat[?]. 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 : $N_1N_2N_3\dots N_{\lceil |C|/2 \rceil}$. 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 $R = B_0C_0B_1C_1\dots B_{r-1}C_{r-1}$. 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*[]) 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 soporte 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 $i + \text{número de terminales}$. 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[?] 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 [?, 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 σ 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 π de $[1, n]$ es un reordenamiento de valores entre 1 y n . Descrita en el capítulo 5.1 [?], 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[?] (Véase encabezado A.4).

4.5.2. Secuencia

Dada la secuencia S de tamaño n sobre un alfabeto Σ , 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 σ vectores de bit A_c , con $c \in \Sigma$, donde $A_c = 1^{\text{rank}_c(S_0, \sigma)} 0 1^{\text{rank}_c(S_1, \sigma)} \dots 0 1^{\text{rank}_c(S_{\lceil n/\sigma \rceil - 1}, \sigma)}$. En esencia, A_c

indica de forma unaria las ocurrencias del símbolo c en cada 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.

$$select_c(S, j) = (s - j + 1) \cdot \sigma + select_c(S_{s-j+1}, s - pred_0(A_c, s)) \text{ donde } s = select_1(A_c, j)$$

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 $\pi = L_0 L_1 L_2 L_3 \dots L_{\sigma-1}$ y la lista D que marca las posiciones donde empieza cada lista en π , $D = 0^{|L_0|} 10^{|L_1|} \dots 0^{|L_{\sigma-1}|}$

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 $R_i \rightarrow B_i C_i$ 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

```

1 int_vector reverseIndexMap(n_non_terminals);
2 for (int i = 0; i < n_non_terminals; i++) {
3     reverseIndexMap[i] = i;
4 }

```

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*

```

1 sort(
2     reverseIndexMap.begin(),
3     reverseIndexMap.end(),
4     [&](int a, int b) {
5         return compareRulesLazy(arsSequence, a, b, n_terminals,
6                                 select, true);
7     });

```

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

```

1 Generator<char> expandRuleSideLazy(
2     ARSSequence& arrs, int i, int nt,
3     std::vector<char>& sl, bool left = false)
4 {
5     int lr_i = left? i: i+1;
6     if (arrs[lr_i] < nt) {
7         co_yield sl[arrs[lr_i] + 1];
8     } else {
9         auto gen = expandRuleLazy(arrs, 2*(arrs[lr_i]-nt), nt, sl,
10                                left);
11         for (char c : gen) {
12             co_yield c;
13         }
14     }
15 bool compareRulesLazy(ARSSequence& arrs, int i, int j, int nt,
16                       std::vector<char>& sl, bool rev = false)
17 {
18     auto gen_i = expandRuleSideLazy(arrs, 2 * i, nt, sl, rev);
19     auto gen_j = expandRuleSideLazy(arrs, 2 * j, nt, sl, rev);
20     auto it_i = gen_i.begin();
21     auto it_j = gen_j.begin();
22     while (it_i != gen_i.end() && it_j != gen_j.end()) {
23         char char_i = *it_i;

```

```

23     char char_j = *it_j;
24     if (char_i != char_j) {
25         return char_i < char_j;
26     }
27     ++it_i;
28     ++it_j;
29 }
30 // If one sequence is shorter, the shorter one is considered "less"
31 return (it_i == gen_i.end()) && (it_j != gen_j.end());
32 }

```

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

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

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*

```

1 vector<int> distance_of_find(reverseIndexMap.size(), 0);
2 for (int i = 0; i < reverseIndexMap.size(); i++) {
3     distance_of_find[reverseIndexMap[i]] = i;
4 }
5 int_vector<> sortedSequenceR = int_vector(n_non_terminals * 2 + 1, 0);
6 for (u_int i = 0; i < reverseIndexMap.size(); i++) {
7     int a_i = reverseIndexMap[i];
8     int b_i = normalized_sequenceR[a_i*2];
9     int c_i = normalized_sequenceR[a_i*2+1];
10    int n_b_i, n_c_i;
11    if (b_i < n_terminals) {
12        n_b_i = b_i;
13    } else {
14        n_b_i = distance_of_find[b_i - n_terminals] + n_terminals;
15    }
16    if (c_i < n_terminals) {
17        n_c_i = c_i;
18    } else {
19        n_c_i = distance_of_find[c_i - n_terminals] + n_terminals;
20    }
21    sortedSequenceR[i*2] = n_b_i;
22    sortedSequenceR[i*2+1] = n_c_i;
23 }
24 int S_i = distance(reverseIndexMap.begin(),
25     find(reverseIndexMap.begin(), reverseIndexMap.end(),
26         n_non_terminals-1));
27 sortedSequenceR[n_non_terminals*2] = S_i;
28 R = ARSSequence(sortedSequenceR, max_normalized + 1 + 1);

```

La última línea 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

```

1 vector<int> distance_of_find(reverseIndexMap.size(), 0);
2 for (int i = 0; i < reverseIndexMap.size(); i++) {
3     distance_of_find[reverseIndexMap[i]] = i;
4 }
5 int_vector<> sortedSequenceR = int_vector(n_non_terminals * 2 + 1, 0);
6 for (u_int i = 0; i < reverseIndexMap.size(); i++) {
7     int a_i = reverseIndexMap[i];
8     int b_i = normalized_sequenceR[a_i*2];
9     int c_i = normalized_sequenceR[a_i*2+1];
10    int n_b_i, n_c_i;
11    if (b_i < n_terminals) {
12        n_b_i = b_i;
13    } else {
14        n_b_i = distance_of_find[b_i - n_terminals] + n_terminals;
15    }
16    if (c_i < n_terminals) {
17        n_c_i = c_i;
18    } else {
19        n_c_i = distance_of_find[c_i - n_terminals] + n_terminals;
20    }
21    sortedSequenceR[i*2] = n_b_i;
22    sortedSequenceR[i*2+1] = n_c_i;
23 }
24 int S_i = distance(reverseIndexMap.begin(),
25                    find(reverseIndexMap.begin(), reverseIndexMap.end(),
26                        n_non_terminals-1));
27 sortedSequenceR[n_non_terminals*2] = S_i;
28 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 $T = \text{abracadabrabra}$ 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 $s(i) = \text{select}_1(b, i + 1)$ (b es el vector de bits obtenido durante la normalización de la secuencia) y σ 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.

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

$$(R = aR_4\ R_3R_6\ R_0R_4\ R_2R_5\ bR_7\ ca\ dR_2\ ra)$$

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

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

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*) (Figura 4.4).

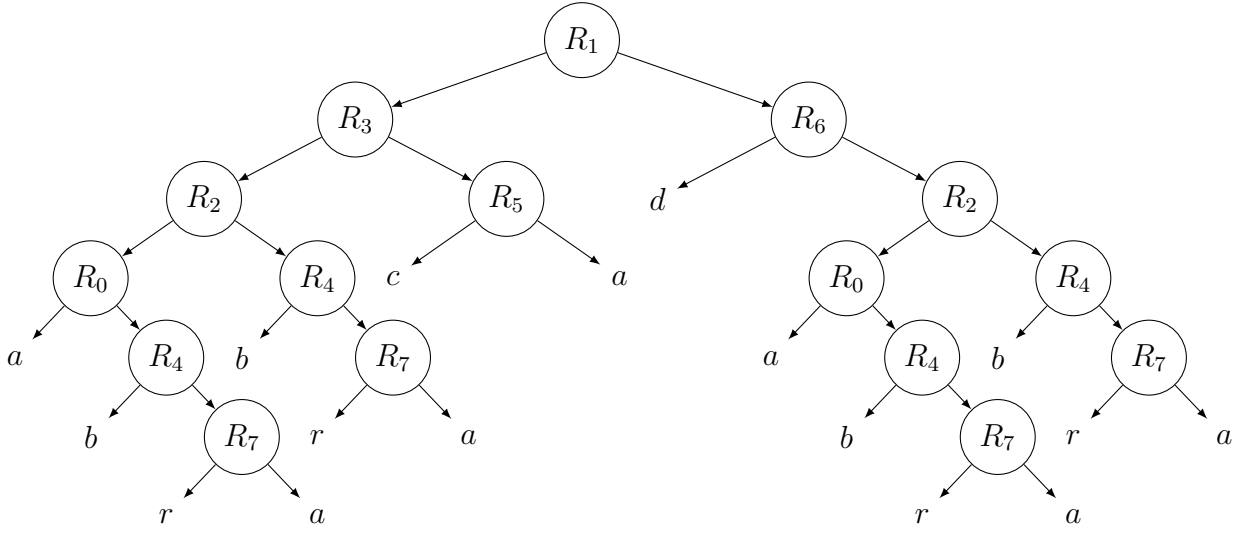


Figura 4.3: Árbol sintáctico para *abracadabra*

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 .

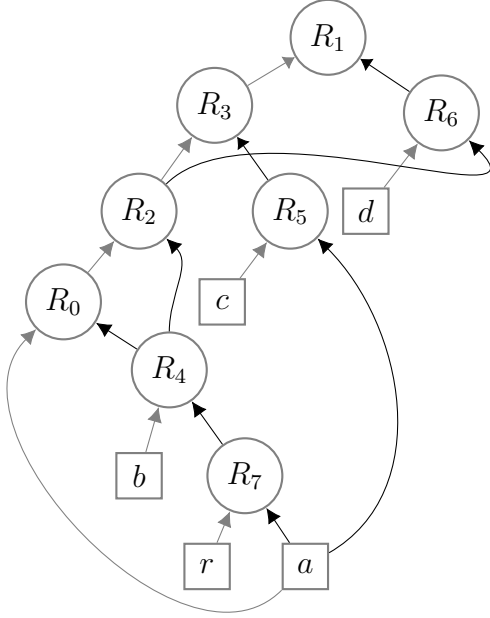


Figura 4.4: DAG para *abracadabra*, las conexiones que corresponden a reglas que aparecen como lados izquierdos son grises, y las derechas son negras

4.8. Grilla

Compact Data Structures[?] 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*[?, 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":

$$\begin{array}{llll}
 S_1 & : & \text{tobeornottobethatisthequestion} & \\
 B_1 & : & 110011011110010010110011011010 & z_1 = 13 \\
 \\
 S_2 & : & \text{benbehaiheein toorottottstqusto} & \\
 B_2 & : & 0010010110011 10000110111101110 & z_2 = 14 \\
 \\
 S_3 & : & \text{bebeaee oorooqo nhihin tttttstust} & \\
 B_3 & : & 0101011 0010000 100001 0000000100 & z_3 = 22 \\
 \\
 S_4 & : & \text{bba ooooqo hihi tttttstst eeee r nn u} & \\
 B_4 & : & 110 000010 0101 111110101 & z_4 = 10 \\
 \\
 S_5 & : & \text{a ooooo hh ss bb q ii ttttttt} &
 \end{array}$$

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*[?]. Con esto, se implementó la estructura de grilla usando matrices *wavelet*[?, Capítulo 10.1] (Véase A.7).

4.8.2. Preparar puntos para grilla

Cada regla $R_i \rightarrow B_i C_i$ 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:

```

1 std::vector<Point> points(n_non_terminals);
2 u_int j, k;
3 for (u_int i = 0; i < indexMap.size(); i++) {
4     k = std::distance(indexMap.begin(), std::find(indexMap.begin(),
5         indexMap.end(), i));
6     points[i] = Point(k, i);
7 }

```

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 "abracadabrabra" (Véase la figura 4.2), conforman la siguiente grilla:

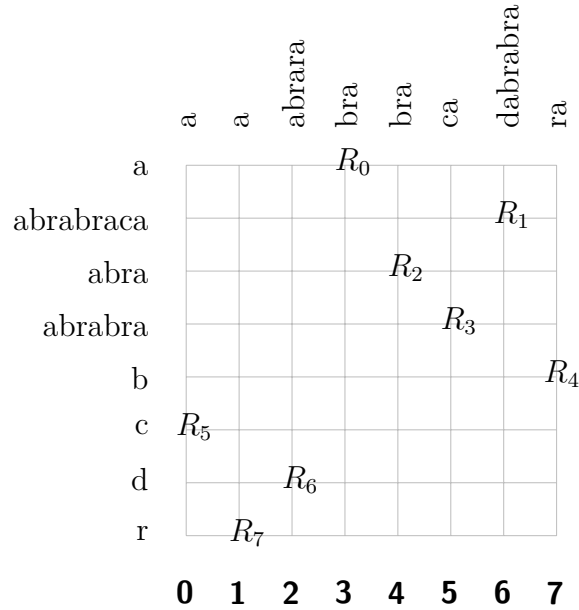


Figura 4.5: Grilla

Si se lee la grilla fila por fila aparecen las reglas en este orden: $R_0, R_1, R_2, \dots, R_7$, 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 .

```

1 l = int_vector(n_non_terminals, 0); // largos de cada regla
2 for (int i = 0; i < n_non_terminals; i++) {
3     l[i] = ruleLength(i);
4 }

```

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).

```

1 int PatternSearcher::ruleLength(int_vector<> *l, int i) {
2     if (l[i] != 0) { //memoization
3         return l[i];
4     }
5     int left, right;
6     if (R[i*2] < nt) {
7         left = 1;
8     } else {
9         left = ruleLength(R[i*2] - nt);
10    }
11    if (R[i*2+1] < nt) {
12        right = 1;
13    } else {
14        right = ruleLength(R[i*2+1] - nt);
15    }
16    l[i] = left + right;
17    return l[i];
18 }

```

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 azul 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).

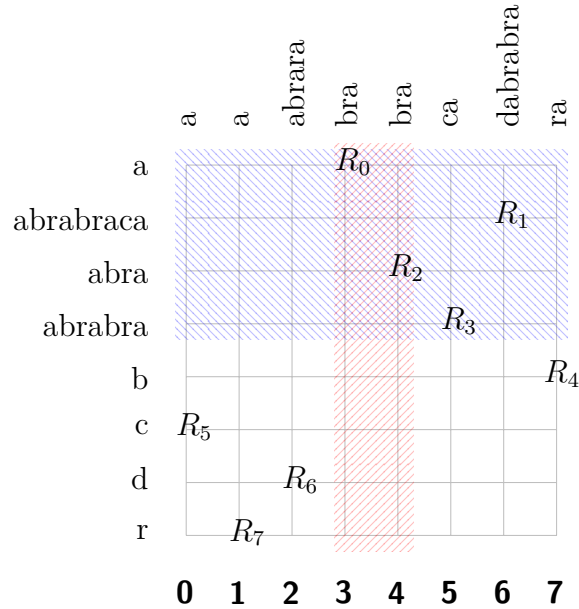


Figura 4.6: Grilla

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).

```
1 u_int m = P.size();
2 u_int t;
3 for (t = 1; t < m; t++) {
4     string P_left = P.substr(0, t); // P_<
5     string P_right = P.substr(t, m-t); // P_>
6     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

```
1 int left = 0, right = G.getRows() - 1;
2 int result = -1;
3 while (left <= right) {
4     int mid = left + (right - left) / 2;
5     int r_i = mid;
6     int compare = compareRuleWithPatternLazy(R, r_i, nt, sl, P_left,
7         true);
8     if (compare >= 0) {
9         if (compare == 0)
10             result = mid;
11         right = mid - 1;
12     } else {
13         left = mid + 1;
14     }
15 }
16 if (result == -1) continue;
17 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:

```
1 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*):

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

```

3         result = mid;
4     }
5     left = mid + 1; // instead of mid - 1
6 } else {
7     right = mid - 1; // instead of mid + 1
8 }

```

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

```

1 template <typename Iterator>
2 int compareRuleWithPatternLazyImpl(
3     ARSSequence& arrs, int i, int nt, std::vector<char>& sl, Iterator
4     pattern_begin, Iterator pattern_end,
5     bool rev = false)
6 {
7     auto gen = expandRuleSideLazy(arrs, 2*i, nt, sl, rev);
8     auto it = gen.begin();
9     while (it != gen.end() && pattern_begin != pattern_end) {
10         char c = *it;
11         char p = *pattern_begin;
12         if (c < p) return -1;
13         if (c > p) return 1;
14         ++it;
15         ++pattern_begin;
16     }
17     if (it == gen.end() && pattern_begin != pattern_end) return -1;
18     return 0;
19 }
20 int compareRuleWithPatternLazy(ARSSequence& arrs, int i, int nt,
21     std::vector<char>& sl, std::string pattern, bool rev = false)
22 {
23     if (rev) {
24         return compareRuleWithPatternLazyImpl(arrs, i, nt, sl,
25             pattern.rbegin(), pattern.rend(), rev);
26     } else {
27         return compareRuleWithPatternLazyImpl(arrs, i, nt, sl,
28             pattern.begin(), pattern.end(), rev);
29     }
30 }

```

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:

```

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

```

```

3   int r_i = p.second-1; // rule index
4   if ((u_int)R[r_i*2] < nt) {
5       secondaries(occurrences, R, S, r_i, nt, 1, 0);
6   } else {
7       secondaries(occurrences, R, S, r_i, nt, 1, 1[R[r_i*2] - nt]-t);
8   }
9 }

```

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.

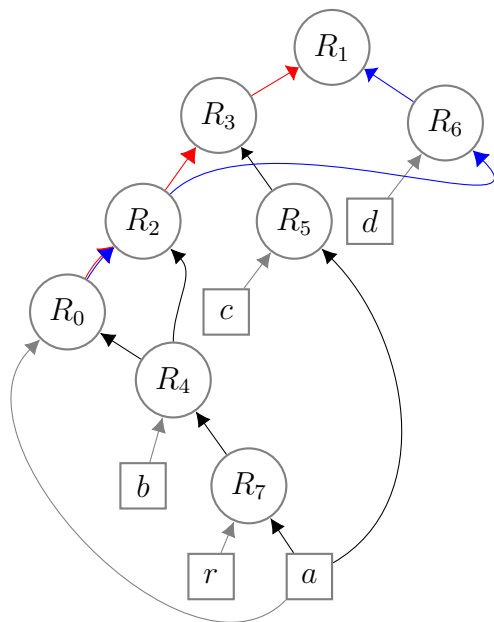


Figura 4.7: Búsqueda en el DAG para nodo R_0

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

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 $R_0 = \mathbf{a\ bra}$. Los dos caminos posibles (en rojo y azul) 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.

The diagram shows a directed graph with eight states labeled R_0 through R_7 . The transitions are as follows:

- $R_7 \xrightarrow{r} R_4$
- $R_7 \xrightarrow{a} R_0$
- $R_4 \xrightarrow{b} R_4$
- $R_4 \xrightarrow{c} R_5$
- $R_4 \xrightarrow{} R_2$
- $R_2 \xrightarrow{} R_0$
- $R_2 \xrightarrow{\text{red}} R_3$
- $R_3 \xrightarrow{\text{red}} R_1$
- $R_3 \xrightarrow{\text{blue}} R_5$
- $R_5 \xrightarrow{d} R_6$
- $R_6 \xrightarrow{\text{blue}} R_7$

El camino rojo llega con un desfase acumulado de 3 (el desfase inicial del patrón $P = \mathbf{ab}$ respecto a la regla $R_2 = abra\ \mathbf{bra}$), 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.

30

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 $r = \text{rank}(R, R_k)$), y cada ocurrencia $j \leq r$ está en $i = \text{select}(R, R_k, j)$, 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

```
1 void secondaries(vector<int> *occs, ARSSequence R, u_int S,  
2   u_int A_i, u_int nt, int_vector<> l, u_int offset=0,  
3   bool terminal = false) {  
4   if (!terminal && A_i == S) {  
5       occs->push_back(offset); return;  
6   }  
7   int c = terminal? A_i: A_i + nt; // nt = number of terminals  
8   for (int j=1; j <= R.rank(c, R.size()); j++) {  
9       int k = R.select(c, j);  
10      int D_i = k / 2;  
11      int offset_prime = offset;  
12      if (k % 2 == 1) { // if A_i is right side  
13          if (R[k-1] < nt) offset_prime++;  
14          else offset_prime += l[R[k-1] - nt];  
15      }  
16      secondaries(occs, R, S, D_i, nt, l, offset_prime, false);  
17  }  
18 };
```

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*?, 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*

```
1 TEST_CASE("PatternSearcher", "[pattern]") {
2     REQUIRE_FALSE(g_fileName.empty());
3     string input_filename = g_fileName;
4     FILE *input = fopen(input_filename.c_str(), "rb");
5     string filecontent = "";
6     char c;
7     while (fread(&c, 1, 1, input) == 1) {
8         filecontent += c;
9     }
10    fclose(input);
11    PatternSearcher PS(input_filename);
12    for (int i = 0; i < 50; i++) {
```

```

13     string pattern = filecontent.substr(rand() %
14         (filecontent.size() - 10), rand() % 10 + 1);
15     cout << i << ": Searching for pattern: \"" << pattern << "\""
16         << endl;
17     vector<int> occurrences;
18     PS.search(&occurrences, pattern);
19     sort(occurrences.begin(), occurrences.end());
20     vector<int> expected_occurrences = findOccurrences(filecontent,
21         pattern);
22     REQUIRE(occurrences == expected_occurrences);
23 }
24 }
25 vector<int> findOccurrences(const string& filecontent,
26     const string& pattern) {
27     vector<int> occurrences;
28     for (size_t i = 0; i < filecontent.size(); i++) {
29         if (filecontent.substr(i, pattern.length()) == pattern) {
30             occurrences.push_back(i);
31         }
32     }
33     return occurrences;
34 }

```

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 contenido (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 σ , 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, σ 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 σ 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 σ 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:

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

Construcción

El proceso de construcción de la estructura, según lo 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: Construcción de matriz

```
1 void WaveletMatrix::build(vector<u32>& S, u32 n, u32 sigma) {
2     vector<u32> S_hat(n);
3     bit_vector M(n, 0);
4     bit_vector M_hat(n, 0);
5     u32 m = sigma;
6     for (u32 l = 0; l <= ceil(log2(sigma))-1; l++) {
7         u32 z_l = 0;
8         bit_vector B_l(n, 0);
9         for (u32 i = 0; i < n; i++) {
10             if (S[i] <= (m - M[i] + 1) / 2) {
11                 B_l[i] = 0;
12                 z_l++;
13             } else {
14                 B_l[i] = 1;
15                 S[i] = S[i] - (m - M[i] + 1) / 2;
16             }
17         }
18         bm.push_back(ppbv(B_l));
19         z.push_back(z_l);
20         if (l < ceil(log2(sigma)) - 1) {
21             u32 p_l = -1; // max value + 1 = 0
22             u32 p_r = z[l] - 1;
23             u32 p;
24             int n_ = n;
25             for (u32 i = 0; i < n_; i++) {
26                 u32 b = bm[l][i];
27                 if (b == 0) {
28                     p_l++;
29                     p = p_l;
30                 } else {
31                     p_r++;
32                     p = p_r;
33                 }
34                 S_hat[p] = S[i];
35                 if (m % 2 == b) {
36                     M_hat[p] = b;
37                 } else {
38                     M_hat[p] = M[i];
39                 }
40                 if ((m+1)/2==2 && M_hat[p] == 1) {
41                     n = n-1;;
42                 }
43             }
44         }
45     }
46 }
```

```

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 $\mathcal{O}(c + n \log n)$ [?, 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:

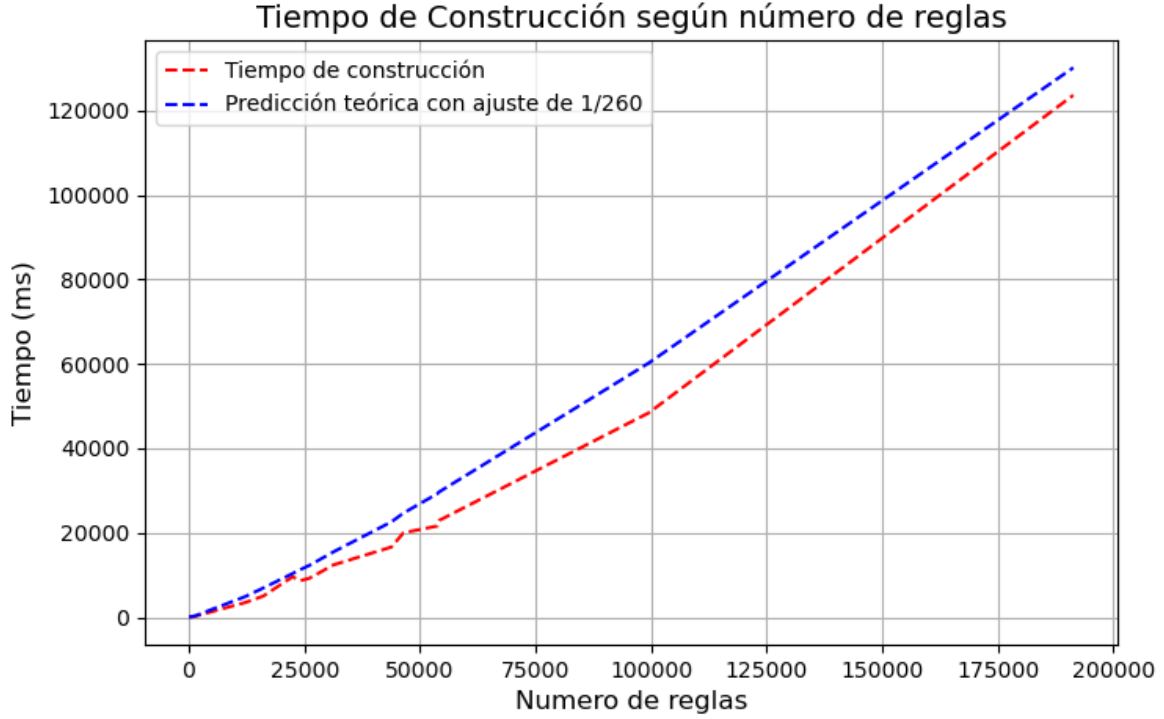
$$\mathcal{O}(n + r \log r \log n) \tag{5.2}$$

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*[?]. 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 memoria 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:

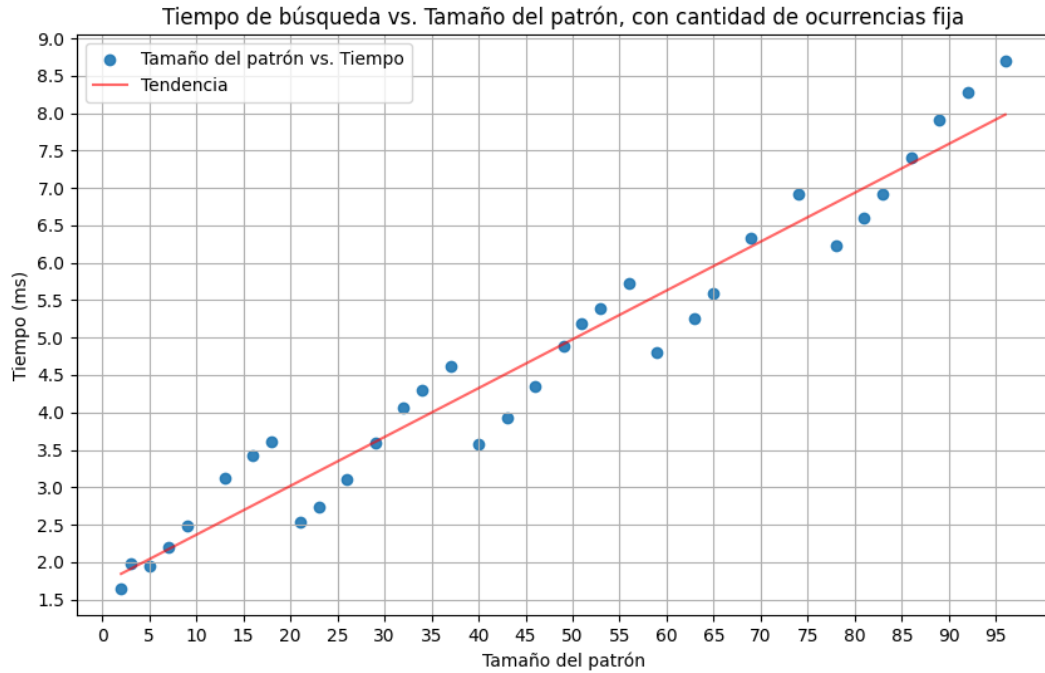
$$\mathcal{O}((m + \log n)m \log r \log \log r + occ \log n \log \log r)$$

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. Por ejemplo, dado un texto $T = (a^i b)^j$ con $i \geq 0$ y $j > 0$, las búsquedas para cualquier patrón $P = a^k b$ con $0 \leq k \leq i$ reportan siempre j ocurrencias. Con esto se puede medir como varía el tiempo en función del largo de P . La figura 5.2 muestra las mediciones de tiempo para la búsqueda de un patrón $a^k b \backslash n$ en el texto donde ese patrón entrega la misma cantidad de ocurrencias, independiente del valor de k .

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



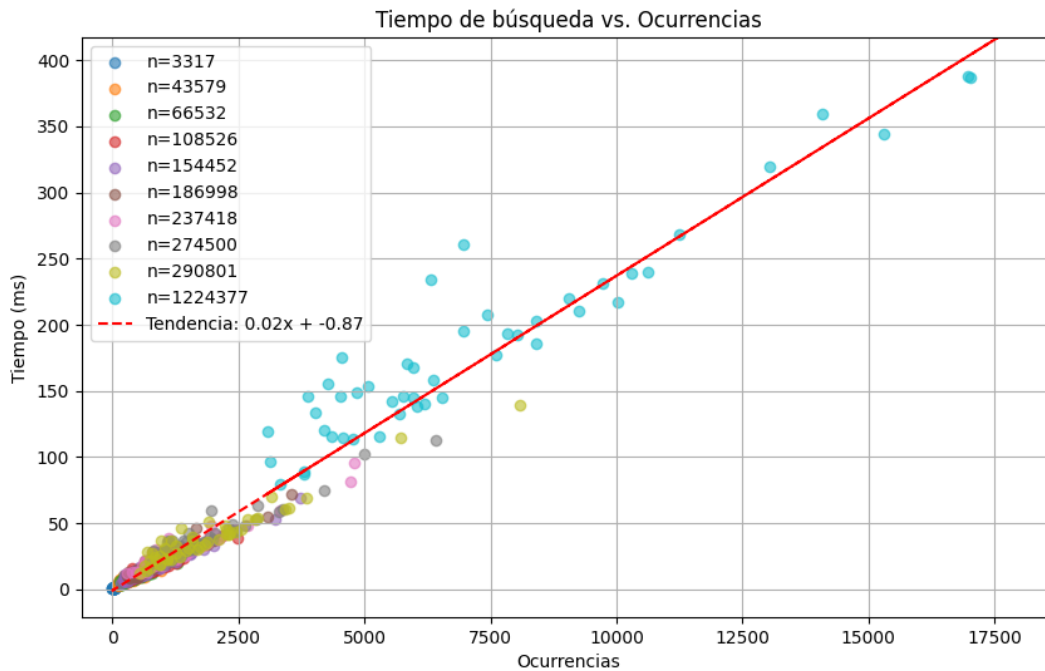
La implementación demuestra un rendimiento equivalente al análisis teórico de la función de búsqueda.

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[?] 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 5.5 y

5.6. El gráfico 5.3 muestra la combinación de todas las mediciones y la tendencia combinada polinomial de primer grado.

La figura 5.4 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 5.3: Tiempo de búsqueda en función del número de ocurrencias para un patrón de largo dos

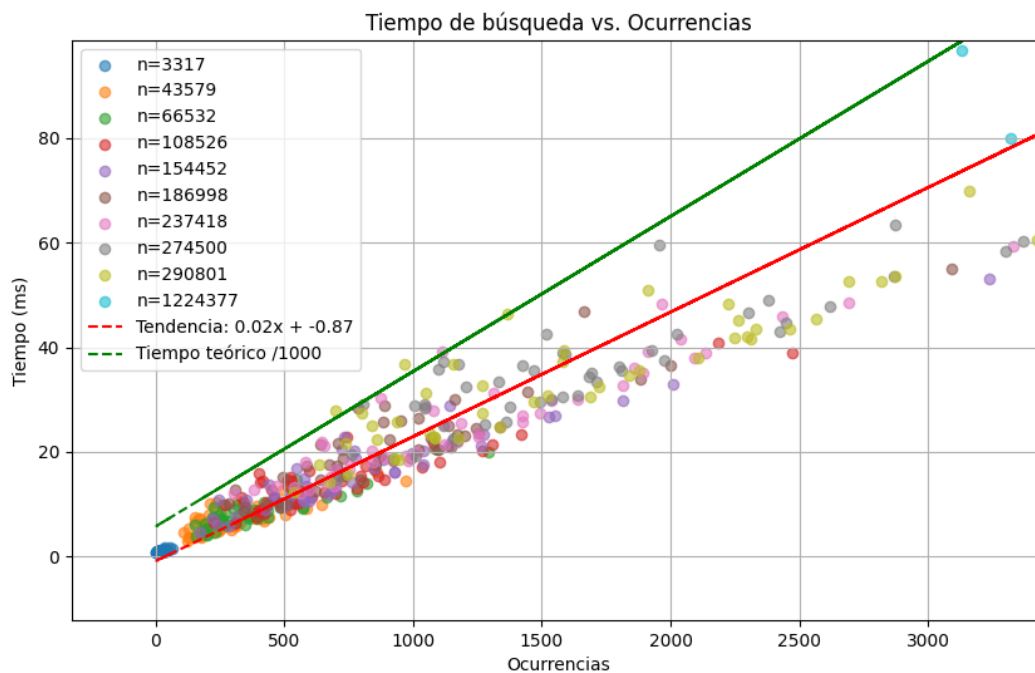


El tiempo de búsqueda corresponde a la suma de los tiempos de búsqueda binaria de rangos en la grilla, el costo temporal de reportar las reglas dentro de estos rangos y los tiempos de búsqueda de ocurrencias para cada regla reportada.

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

Figura 5.4: 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 $O((m + \log n)m \log r \log \log r + occ \log n \log r)$.



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

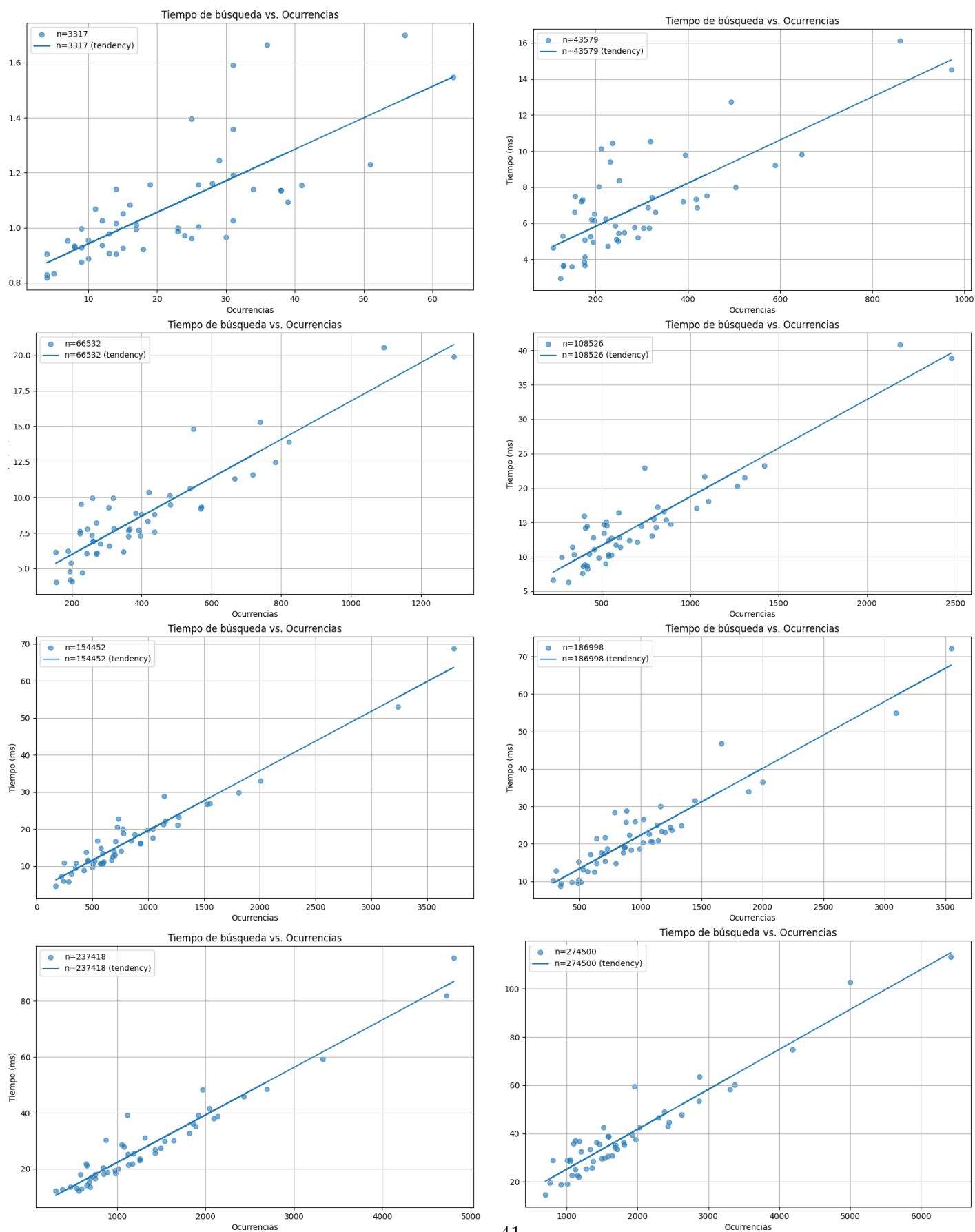
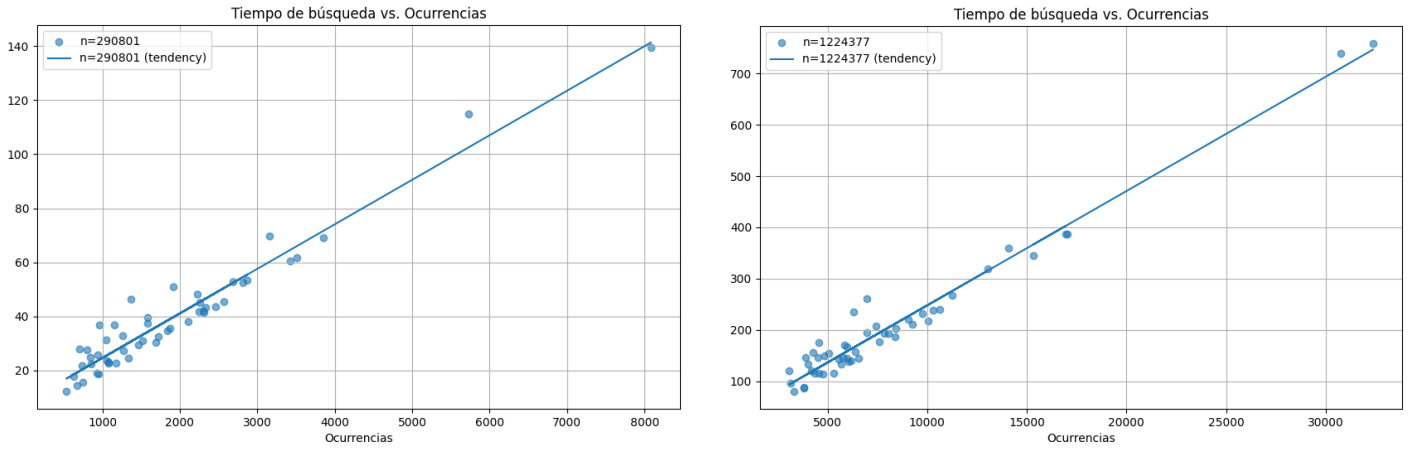


Figura 5.6: Tiempos de búsquedas en *ms* (milisegundos) en función del número de ocurrencias para un patrón de largo 2 (Continuación)



5.3. Análisis en textos altamente repetitivos

Las colecciones de textos reales altamente repetitivos de *Pizza & Chili Corpus*[?] sirven como entradas para pruebas del mismo tipo a las utilizadas por el índice comprimido basado en gramática[?] 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	<i>bps</i>
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 5.7 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 5.7: Tiempos por ocurrencia en μs (microsegundos) de búsquedas en función del número de ocurrencias para distintas colecciones repetitivas. Ambos ejes están es escala logarítmica

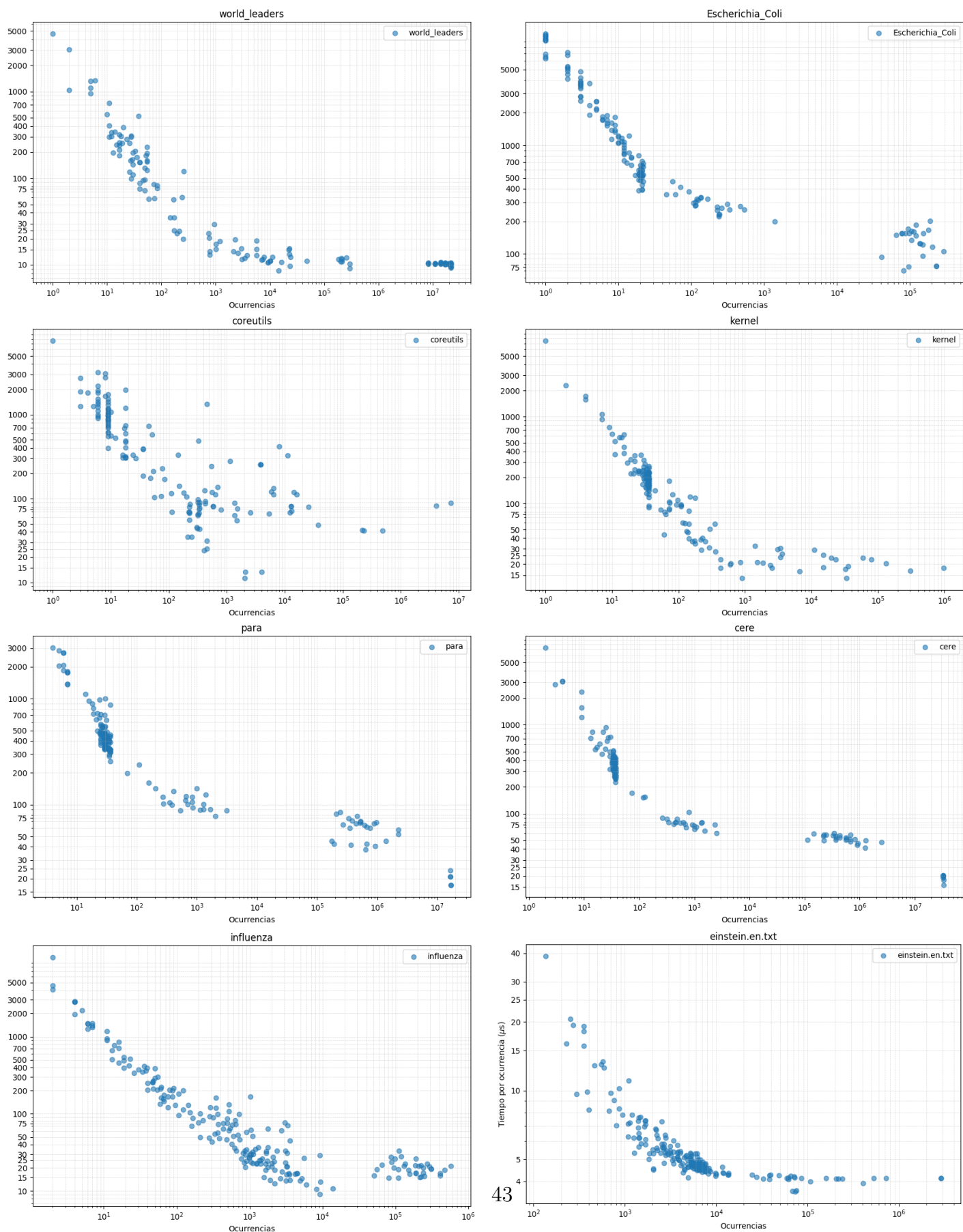
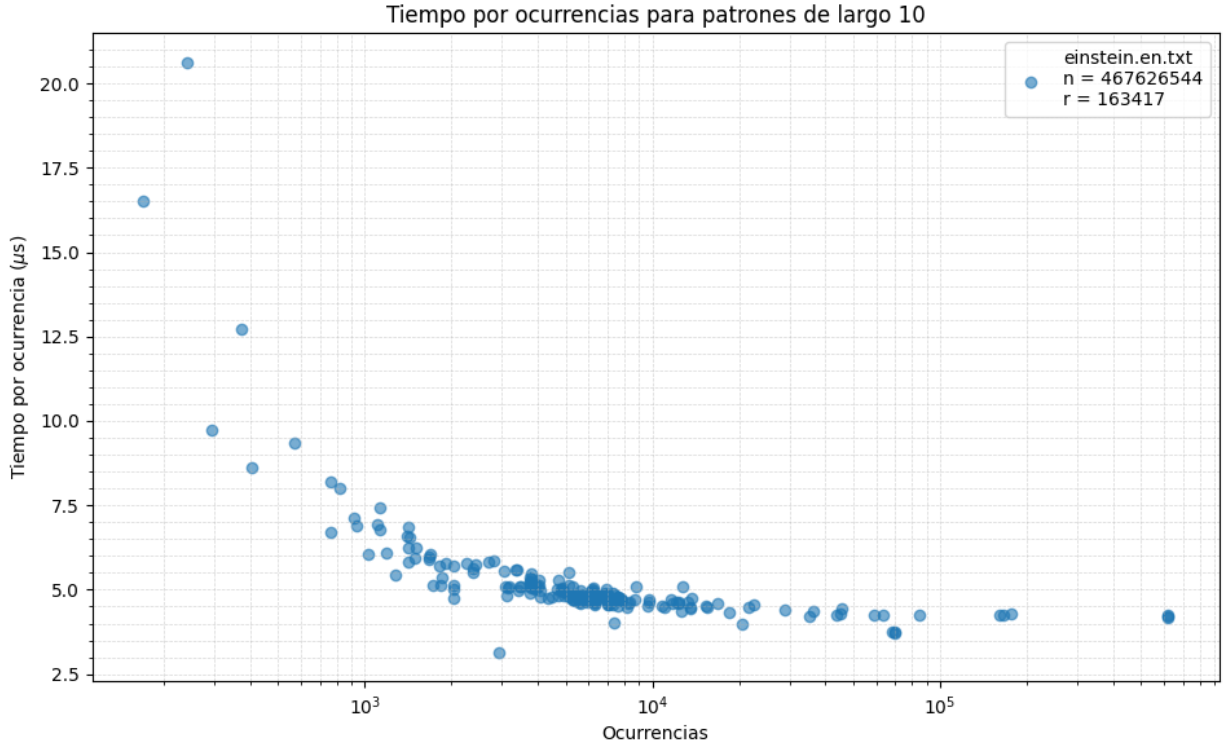


Figura 5.8: Tiempos por ocurrencia en μs (microsegundos) de búsquedas de patrones aleatorios de largo fijo 10 en función del número de ocurrencias en la colección *einstein.en*. El eje X está en escala logarítmica



corto.

En colecciones altamente repetitivas como *einstein.en* logra tiempos de búsqueda por ocurrencia bajo 5 μs 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 μs . En otras colecciones el tiempo por ocurrencia es menos estable: en *influenza* altas ocurrencias tienen tiempo por ocurrencia entre 5 a 30 μs .

Se analizó el caso para patrones aleatorios de largo fijo $m = 10$ sobre la colección *einstein.en*. Los resultados se muestran en la figura 5.8 donde la búsqueda alcanza tiempos estables de 5 μs 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:

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

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 :

"aaaaaaaaabaaaaaaaaab...aaaaaaaaab\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 $a^i b \backslash n$ 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[?] es una estructura de datos que permite buscar patrones en un texto comprimido por gramática. La tabla 5.5 muestra los tiempos de búsqueda de patrones aleatorios en textos reales con la estructura propuesta en este trabajo y con la estructura del índice comprimido basado en gramática.

set	tamaño	m	occ	t(ms) estructura	t(ms) índice
------------	---------------	----------	------------	-------------------------	---------------------

Tabla 5.5: Tiempos de búsqueda para patrones aleatorios de largo *m*, con *occ* ocurrencias

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[?]. 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, y se logró una implementación funcional de la estructura propuesta en el libro *Compact Data Structures*.
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.
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.
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.

6.3. 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 \rightarrow 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 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.

6.4. 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.

6.5. Otras mejoras

6.5.1. Potencial paralelismo

Es posible también utilizar múltiples *threads* o multihilos en ciertas partes del programa en donde el paralelismo podría mejorar considerablemente la búsqueda, 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.

Apéndice A

Anexo

Listing A.1: Llamando Re-Pair

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

Listing A.2: Añadir más reglas hasta eliminar C

```
1 while (dict->seq_len > 1) {
2     for (u_int i = 0; i < dict->seq_len; i = i+2) {
3         if (i == dict->seq_len - 1) { // odd case
4             comp_seq[i/2] = comp_seq[i];
5         }
6         else {
7             RULE new_rule;
8             new_rule.left = comp_seq[i];
9             new_rule.right = comp_seq[i+1];
10            rules[dict->num_rules] = new_rule; // append new rule
11            comp_seq[i/2] = dict->num_rules; // update sequence C
12            dict->num_rules++;
13        }
14    }
15    dict->seq_len = dict->seq_len % 2 == 0 ? dict->seq_len / 2 :
16        dict->seq_len / 2 + 1;
17 }
```

Listing A.3: Normalizar Secuencia

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

```

3 for (u_int i = 0; i < sequenceR.size(); i = i + 2) {
4     sequenceR[i] = rules[i/2 + 257].left;
5     sequenceR[i + 1] = rules[i/2 + 257].right;
6     if (sequenceR[i] <= 256) {
7         bbbb[sequenceR[i]] = 1;
8     }
9     if (sequenceR[i + 1] <= 256) {
10        bbbb[sequenceR[i + 1]] = 1;
11    }
12 }
13 rank_support_v<1> rank_bbbb(&bbbb);
14 select_support_mcl<1, 1> select_bbbb(&bbbb);
15 vector<char> rank(257, 0);
16 vector<char> select(257, 0);
17 for (int i = 0; i < 257; i++) {
18     rank[i] = rank_bbbb(i);
19     if (i==0) continue;
20     select[i] = select_bbbb(i);
21 }
22 u_int max_terminal = 0;
23 for (u_int i = 1; i <= rank_bbbb(257); i++) {
24     if (select_bbbb(i) > max_terminal) {
25         max_terminal = select_bbbb(i);
26     }
27 }
28 int_vector<> normalized_sequenceR(sequenceR.size(), 0, sizeof(CODE) *
    8);
29 int sz = sequenceR.size();
30 int r;
31 int max_normalized = 0; // maximum symbol in the normalized alphabet
32 for (int i = 0; i < sz; i++) {
33     if (sequenceR[i] < 256)
34         r = rank_bbbb(sequenceR[i] + 1) - 1;
35     else
36         r = sequenceR[i] - 257 + rank_bbbb(257);
37     normalized_sequenceR[i] = r;
38     if (r > max_normalized) {
39         max_normalized = r;
40     }
41 }

```

Listing A.4: Permutaciones utilizando vectores de bit

```

1 typedef struct abv {
2     bit_vector b;
3     rank_support_v<0> rank;
4     select_support_mcl<1, 1> sel_1;
5     select_support_mcl<0, 1> sel_0;
6 } abv; // rank, selects vector
7 typedef struct dbv {

```

```

8     bit_vector b;
9     select_support_mcl<1, 1> sel_1;
10    select_support_mcl<0, 1> sel_0;
11 } dbv; // selects vector
12
13 class Permutation {
14     friend class PowerPermutation;
15 protected:
16     int rank_b(int i);
17     int_vector<> pi; // permutation
18     int_vector<> S; // shortcuts
19     brv b; // bit vector to mark shortcuts
20 public:
21     Permutation();
22     int t; // parameter t
23
24     /// @brief Sole constructor, it does not check if pi is a
25     permutation
26     /// @param pi vector of integers (initially, 8 bit long integers)
27     /// @param t parameter t length of the shortcuts
28     Permutation(int_vector<> pi, int t);
29
30     /// @brief Return the position of the element i after applying the
31     permutation
32     int operator[](int i);
33     int permute(int i) { return this->operator[](i); };
34
35     /// @brief Return the inverse of the permutation,
36     /// that is, the position j such that permutation ( j ) = i
37     /// @param i
38     /// @return
39     int inverse(int i);
40 };

```

Listing A.5: Secuencia utilizando permutaciones

```

1 class ARSSequence {
2 private:
3     vector<abv> A;
4     vector<dbv> D;
5     vector<Permutation> pi;
6     int sigma; int n;
7     int select_1_D(int k, int i);
8     int select_0_D(int k, int i);
9     int select_0_A(int k, int i);
10    int select_1_A(int k, int i);
11    int rank_A(int c, int i);
12    int pred_0_A(int c, int s);
13 public:
14     /// @brief Builds structure to support rank, select and access

```

```

    queries
15  /// @param S integer vector representing the sequence
16  /// @param sigma size of alphabet [0 . . . sigma)
17  ARSSequence(int_vector<> S, int sigma);
18  /// @brief Access query
19  /// @param i position in the sequence
20  /// @return The symbol at position i
21  int access(int i);
22  int operator[](int i) { return access(i); };
23  /// @brief Rank query
24  /// @param c symbol in the alphabet
25  /// @param i position in the sequence
26  /// @return The number of occurrences of c in the sequence up to
    and including position i
27  int rank(int c, int i);
28  /// @brief Select query
29  /// @param c symbol in the alphabet
30  /// @param i the i-th occurrence of c in the sequence
31  /// @return The position of the i-th occurrence of c in the
    sequence
32  /// @note Position returned is 0-indexed, while parameter i is
    1-indexed as ordinal numbers are.
33  int select(int c, int j);
34  u_int size() { return n; }
35 };

```

Listing A.6: Matriz *Wavelet*

```

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

```

```

20  /// @param i the zero-indexed position
21  /// @param c the number
22  /// @return the number of occurrences until position i
23  u32 rank(u32 c, u32 i);
24  /// @brief Returns the 0-indexed position of the j-th occurrence of
    the number c
25  /// @param c a number
26  /// @param j a positive number
27  /// @return the position or the size of the sequence if not found,
    or -1 if c is not in the sequence
28  u32 select(u32 c, u32 j);
29  void printself();
30  ppbv operator[](u32 level);
31  u32 offset(u32 level);
32  u32 size() { return bm[0].size(); }
33 };

```

Listing A.7: Grilla

```

1  class Grid {
2  private:
3      u32 c; // number of columns
4      u32 r; // number of rows
5      u32 n; // nubmer of points
6      WaveletMatrix wt; // Wavelet tree
7      u32 count(u32 x_1, u32 x_2, u32 y_1, u32 y_2, u32 l, u32 a, u32 b);
8      vector<Point> report(u32 x_1, u32 x_2, u32 y_1, u32 y_2, u32 l,
    u32 a, u32 b);
9      u32 outputx(u32 level, u32 x);
10     u32 outputy(u32 level, u32 a, u32 b, u32 i);
11 public:
12     /// @brief Construct a grid from a binary file
13     /// @param fn Filename of the binary file
14     /// @note The binary file should contain the dimensions of the
    grid first
15     /// (columns, rows), followed by the points as pairs of integers.
    Every integer
16     /// in the file should be a 4-byte long unsigned integer
    (uint32_t).
17     /// The coordinates should be 0-indexed.
18     Grid(const string& fn);
19     Grid(std::vector<Point>& points, u32 columns, u32 rows);
20     /// @brief Count the number of points in the grid that are within
    the rectangle
21     /// @param x_1 1-indexed column range start
22     /// @param x_2 1-indexed column range end
23     /// @param y_1 1-indexed row range start
24     /// @param y_2 1-indexed row range end
25     /// @return The number of points in the grid that are
    within the rectangle as an integer
26

```



```

27     u32 count(u32 x_1, u32 x_2, u32 y_1, u32 y_2);
28     /// @brief Report the points in the grid that are within the
        rectangle
29     /// @param x_1 1-indexed column range start
30     /// @param x_2 1-indexed column range end
31     /// @param y_1 1-indexed row range start
32     /// @param y_2 1-indexed row range end
33     /// @return A vector of points that are within the rectangle
34     vector<Point> report(u32 x_1, u32 x_2, u32 y_1, u32 y_2);
35     void printself();
36     u32 getColumns() { return c; }
37     u32 getRows() { return r; }
38     WaveletMatrix getWaveletMatrix() { return wt; }
39     /// @brief Access the number in the i-th 1-indexed position
40     /// @param i
41     /// @return
42     u32 access(u32 i) {return wt.access(i-1);};
43     /// @brief Returns the 1-indexed position of the j-th occurrence
        of c
44     /// @param j
45     /// @param c
46     /// @return
47     u32 select(u32 j, u32 c) {return wt.select(j, c);};
48 };

```

Listing A.8: Buscador de patrones

```

1  class PatternSearcher {
2  private:
3      Grid G; // Grid
4      ARSSequence R; // ARS sequence
5      u_int S; // Initial symbol
6      int_vector<> l; // Lengths of the expansion of the rules
7      uint nt; // Number of terminals
8      vector<char> sl; // select vector for normalized alphabet
9      vector<char> rk; // rank vector for normalized alphabet
10     string expandRule( int i, unordered_map<int, string>& memo);
11     string expandRightSideRule(int i, unordered_map<int, string>
        &memo);
12     string expandLeftSideRule(int i, unordered_map<int, string>& memo);
13     int ruleLength(int i);
14     Generator<char> expandRuleLazy( int i, bool rev = false);
15     Generator<char> expandRuleSideLazy( int i, bool left = false);
16     bool compareRulesLazy(int i, int j, bool rev = false);
17     template <typename Iterator>
18     int compareRuleWithPatternLazyImpl ( int i, Iterator
        pattern_begin, Iterator pattern_end, bool rev = false);
19     int compareRuleWithPatternLazy(int i, string pattern, bool rev =
        false);
20     void secondaries(vector<int> *occurences, u_int A_i, u_int

```

```

    offset=0, bool terminal = false);
21 public:
22     PatternSearcher(){};
23     /// @brief Construct a pattern searcher from a text file
24     /// @param input_filename
25     PatternSearcher(string input_filename);
26     /// @brief Report all occurrences of a pattern in the text
27     /// @param occurrences Vector to store the occurrences
28     /// @param P Pattern to search
29     void search(vector<int> *occurrences, string P);
30     int numRules() { return R.size() / 2; }
31 };

```