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

ÁRBOLES DE SUFIJOS COMPRIMIDOS PARA TEXTOS ALTAMENTE REPETITIVOS

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

ANDRÉS JONATHAN ABELIUK KIMELMAN

PROFESOR GUÍA:
GONZALO NAVARRO BADINO

MIEMBROS DE LA COMISÍON:
CLAUDIO GUTIÉRREZ GALLARDO
RODRIGO PAREDES MORALEDA

Este trabajo ha sido parcialmente financiado por el Instituto Milenio
de Dinámica Celular y Biotecnología (ICDB) y el proyecto Fondecyt 1-080019.

SANTIAGO DE CHILE
ENERO 2012

Resumen

El árbol de sufijos es una de las estructuras más importantes que se han creado para el manejo de cadenas de caracteres. Esta estructura permite encontrar eficientemente las ocurrencias de un patrón, en tiempo proporcional al largo del patrón. Adicionalmente soporta operaciones para resolver problemas complejos sobre una secuencia. Esta estructura tiene muchas aplicaciones en variadas áreas de la investigación, destacándose en la bioinformática, donde los recientes avances tecnológicos han permitido recolectar grandes colecciones de secuencias de ADN.

La implementación clásica se vuelve impracticable para grandes volúmenes de información dado que ocupan demasiado espacio, que siempre muchas veces mayor que el texto mismo. Luego, no pueden ser almacenados en memoria principal, lo que en la práctica significa un aumento importante del tiempo de respuesta. Este problema es la principal motivación por la cual se buscan nuevas representaciones comprimidas de esta estructura, dando lugar a los árboles de sufijos comprimidos. Estos contienen la misma información que los árboles de sufijos pero ocupan un espacio considerablemente menor.

Existen variadas propuestas teóricas para representar un árbol de sufijos comprimido, que ofrecen espacios y tiempos diferentes. En la práctica, dos estructuras destacan por sobre las demás. La primera fue propuesta por Sadakane e implementada por Välimäki et al. Esta estructura soporta la mayoría de las operaciones de navegación en tiempo constante, pero en la práctica requiere entre 25 y 35 bits por símbolo. La segunda fue propuesta por Fischer et al. e implementada por Cánovas, incorporando variantes y nuevas ideas para todas las estructuras que componen el árbol de sufijos comprimido propuesto por ellos. Una de estas variantes resulta ser superior a la implementación de Sadakane tanto en espacio como en tiempo, utilizando alrededor de 8 a 12 bits por símbolo.

Dado que secuencias de ADN relacionadas son altamente similares, por ejemplo dos genomas humanos son muy parecidos, las colecciones pueden ser tratadas como un gran texto que contiene cadenas altamente similares.

En este trabajo se propone e implementa una nueva variante del árbol de sufijos comprimido de Fischer et al, optimizada para textos altamente repetitivos. Se reemplazan y/o modifican cada una de las estructuras que componen el árbol por nuevas que presentan mayor compresión en textos repetitivos. El resultado más importante consiste en crear una nueva estructura inspirada en una técnica de compresión basada en gramáticas, aplicable al árbol de sufijos comprimido, que con poco espacio extra acelera considerablemente las operaciones sobre el árbol. Finalmente, la variante se compara experimentalmente sobre textos altamente repetitivos y resulta ser superior a la implementación de Cánovas, tanto en tiempo como en espacio, ocupando entre 3 a 6 bits por símbolo.

Agradecimientos

Quiero agradecerle a mi familia por el apoyo incondicional en todas las deciciones que he tomado y por estar siempre presentes a lo largo de mi vida, siempre dandome su cariño.

Agradezco los grandes amigos que fui creando a lo largo de la carrera, sin los cuales no hubiese podido perdurar los 6 años que dura la carrera. Los mejores recuerdos de la universidad vienen de los momentos pasados con mis amigos, los cuales espero me acompañen el resto de la vida.

Finalmente quiero agradecer al profesor Gonzalo Navarro por guiarme con sus comentarios y consejos perspicaces, sin los cuales no hubiese podido realizar la memoria.

Índice

Lista de Figuras

Lista de Tablas

Capítulo 1

Introducción

Los recientes avances tecnológicos en diversas áreas han permitido recolectar grandes colecciones de datos, que luego tienen que ser procesados. Una de estas áreas es la bioinformática, donde se analizan cada vez más y mayores colecciones de secuencias de ADN. Dado que secuencias de ADN relacionadas son altamente similares, por ejemplo dos genomas humanos son muy parecidos, las colecciones pueden ser tratadas como un gran texto que contiene cadenas altamente similares. Estas colecciones serán una realidad en un futuro no muy lejano, y se necesitarán métodos para almacenar y analizar, por ejemplo, miles de genomas humanos.

Un auto-índice es una estructura que almacena un texto en forma comprimida y permite realizar búsquedas eficientemente. Adicionalmente, los auto-índices permiten extraer cualquier porción de la colección. Uno de los objetivos de estos índices es que puedan ser almacenados en memoria principal. Esta característica es sumamente importante ya que el disco puede llegar a ser un millón de veces más lento que la memoria principal. Existe una gran diversidad de propuestas para implementar auto-índices, dentro de los cuales algunos ofrecen una funcionalidad de árbol de sufijos. El árbol de sufijos es una de las estructuras más importantes que se han creado para el manejo de cadenas de caracteres. Esta estructura tiene muchas aplicaciones en variadas áreas de la investigación, destacándose en la bioinformática.

El árbol de sufijos se puede describir como una estructura de datos que representa todos los sufijos de un texto, de tal manera que permite resolver eficientemente muchos problemas de strings, en particular la búsqueda de un patrón dentro del texto en tiempo proporcional al largo del texto. El gran problema de los árboles de sufijos es que ocupan demasiado espacio, siempre muchas veces (10 a 20) mayor que el texto mismo. Por ejemplo, para una aplicación bioinformática, donde se trabaja con grandes volúmenes de información (mayores a 1 Gigabyte), se hace poco eficiente e impracticable el uso de árboles de sufijos, ya que no caben en memoria primaria.

Una forma de reducir el espacio que ocupa el árbol de sufijos es utilizar otro tipo de estructura de datos, como un arreglo de sufijos. El problema es que esta estructura tiene menos información que el árbol, por lo que no sirve para resolver problemas complejos de strings que el árbol sí puede. Luego, una forma de recuperar esa información es extender el arreglo de sufijos con otras estructuras de datos, de manera que con el conjunto de estructuras se tenga la misma información que el árbol original, dando lugar a una representación comprimida del árbol de sufijos.

La forma de representar los árboles de sufijos comprimidos es mediante un arreglo de sufijos comprimido junto con información de los prefijos comunes más largos (LCP) del texto, además de información de la topología del árbol, para poder simular que se están recorriendo los nodos.

En particular existen varias implementaciones para cada una de las estructuras mencionadas anteriormente. Cada implementación tiene un distinto compromiso entre espacio y tiempo de respuesta, pero siempre es dependiente del texto. Es decir, dependiendo de la estructura y distribución del texto es que una u otra implementación se comportará de mejor manera, siendo la mejor opción para cierto tipo de texto. Luego, es importante analizar para qué clases de texto se comporta mejor cada tipo de árbol de sufijos propuesto. La clase de textos a estudiar en este trabajo son aquellos altamente repetitivos.

Dos de las estructuras mencionadas anteriormente, el arreglo de sufijos y el LCP, ya han sido optimizadas para textos repetitivos.Ambas estructuras permiten que el espacio que usan disminuya a medida que el texto sea más repetitivo, obteniendo buenos resultados. Existe una tercera estructura jerárquica NPR que

se construye sobre el arreglo *LCP* para responder eficientemente consultas para navegar el árbol, entre otras operaciones. Lo que no existe, son propuestas que disminuyan el espacio para representar esta estructura en secuencias repetitivas.

La propuesta para esta memoria consiste en crear una estructura *NPR* que explote las repeticiones del texto, esperando que se generen suficientes repeticiones en el LCP diferencial, de manera que permita comprimirse con un método para secuencias repetitivas y sea competitiva tanto en espacio como en tiempo. La estructura esta inspirada en usar una compresión basada en gramáticas [LM00] sobre el *LCP* y además que soporte eficientemente las consultas para navegar el aŕbol con una técnica similar a la empleada en el range min-max tree.

Construiremos la nueva estructura sobre la reciente implementación de Cánovas de árboles de sufijos comprimidos que experimentalmente muestra una gran robustez y competitividad práctica sobre otras representaciones [CN10]. Todas las variantes y estructuras que se proponen es esta tesis se comparan experimentalmente con el árbol ya mencionado.

1.1. Descripción del Contenido y Contribuciones de la Memoria

Capítulo 2: Se describen todos los conceptos básicos relevantes para el estudio de los árboles de sufijos comprimidos presentados en esta memoria.

Capítulo 3: Se presentan las alternativas prácticas existentes para representar árboles de sufijos comprimidos.

Capítulo 4: Se presenta un nuevo y práctico árbol de sufijos comprimido para secuencias altamente repetitivas. La solución está inspirada en el árbol de Fischer et al. [FMN09], en la estructura *NPR-CN* de Cánovas y Navarro [CN10] y en la técnica de compresión *Re-Pair* [LM00]. Se compara experimentalmente en términos de espacio y tiempo con la estructura implementada por Cánovas[CN10], presentada en el capítulo 3.

Capítulo 5: Se presentan las conclusiones y futuras lineas de investigación basadas en los resultados de este trabajo.

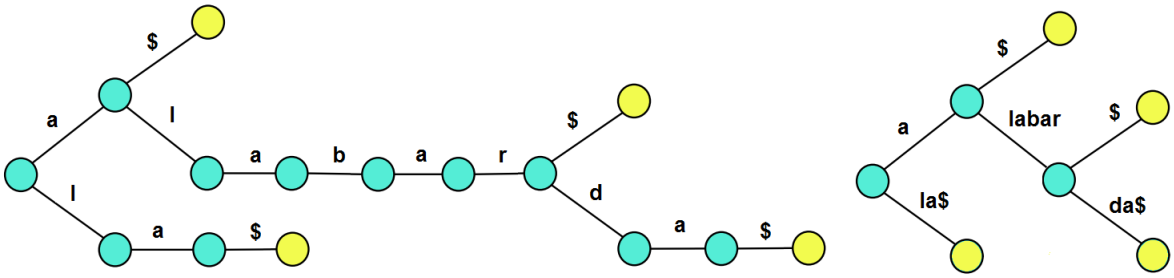

Fig. 2.1: A la izquierda, ejemplo de un trie para el conjunto S ={“alabar”, “a”, “la”, “alabarda”}. A la derecha ejemplo de árbol patricia de S. [Can10]

Capítulo 2

Conceptos Básicos

En esta sección se presentan conceptos básicos que luego son usados a lo largo del trabajo y son necesarios para la mejor comprensión de esta memoria. Para simplificar notación los logaritmos son todos en base a 2 ($\log = \log_2$) a menos que se diga lo contrario.

2.2. Strings

Un string $S = S_{1,n} = S[1,n] = s_1, s_2, \ldots, s$ es una secuencia de símbolos, donde cada símbolo pertenece al alfabeto Σ de tamaño σ. Un substring de S se escribe $S_{i,j} = S[i,j] = s_i s_{i+1} ... s_j$ Un prefijo de S es un substring de la forma $S_{1,j}$ y un sufijo es un substring de la forma $S_{i,n}$. Si $i > j$ entonces $S_{i,j} = \varepsilon$, el string vacío, de largo $|\varepsilon| = 0$. Un texto $T = T_{1,n}$ es un string terminado con un símbolo especial $t_n = \$ \notin \Sigma$, lexicográficamente menor que cualquier otro símbolo en Σ. El orden lexicográfico ($<$) entre strings se define como $aX < bY$ si $a < b \vee (a = b \wedge X < Y)$, donde a, b son símbolos y X, Y son strings sobre Σ.

2.3. Árboles trie

Un árbol *trie* es una estructura de datos que almacena un conjunto de strings. Permite encontrar un string dentro del conjunto en tiempo proporcional al largo del string.

Definición 2.1. Un *trie* sobre un conjunto S de strings distintos es un árbol etiquetado donde cada nodo representa un prefijo del conjunto y el nodo raíz representa el prefijo ε. Un nodo v que representa el prefijo Y es hijo de un nodo u que representa el prefijo X sii $Y = Xc$ para algún símbolo c que etiqueta la rama entre u y v. No pueden haber dos nodos representando un mismo prefijo.

Se asume que todos los strings terminan con el símbolo especial \$, no perteneciente al alfabeto. Esto asegura que ningún string es prefijo de otro string en el conjunto, lo que garantiza que el árbol tiene exactamente $|S|$ hojas. La figura 2.1 muestra un ejemplo de un trie.

Definición 2.2. Un *trie* de sufijos de un texto $T[1,n]$, es un *trie* compuesto por todos los sufijos $T_{i,n}$ del texto, para $i = 1 \ldots n$.

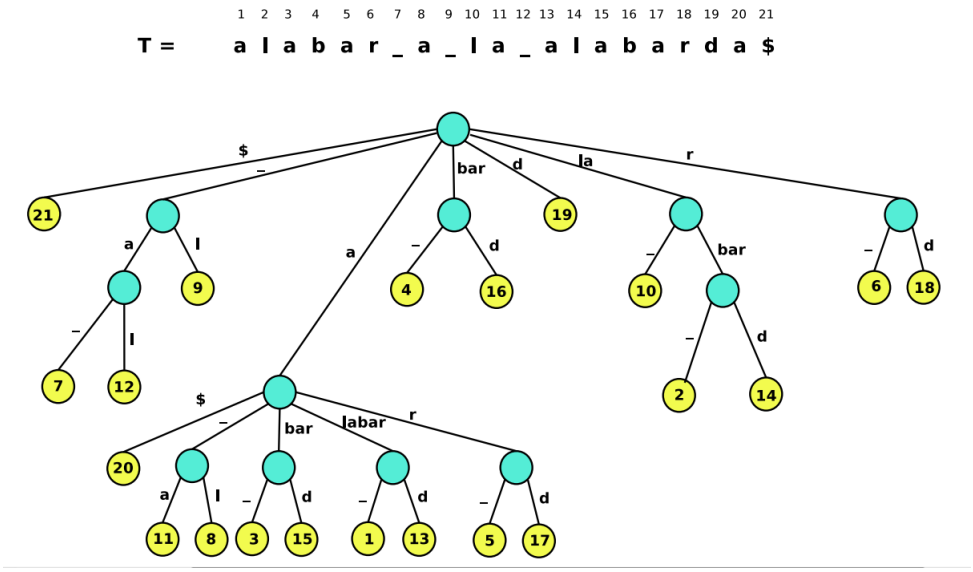

Fig. 2.2: Árbol de sufijos para el texto “alabar_a_la_alabarda$”. [CN10]

Un *trie* para el conjunto $S = \{S_1, \ldots, S_n\}$ se puede construir en tiempo $O(|S_1| + \cdots + |S_n|)$ mediante inserciones sucesivas de cada string en el árbol. Para buscar un patrón P, se parte de la raíz y se van siguiendo las ramas etiquetadas con los caracteres de P, lo que toma tiempo $O(|P|)$.

Definición 2.3. Un árbol *patricia* es una representación compacta de un *trie* que permite reducir el espacio de almacenamiento. Corresponde a que todo nodo que es un hijo único, es unido con su padre, renombrando la etiqueta como la concatenación de las etiquetas borradas. La figura 2.1 muestra un ejemplo de un árbol *patricia*.

2.4. Árbol de sufijos

Definición 2.4. Un árbol de sufijos de un texto $T[1, n]$ es un árbol *patricia* construido sobre todos los sufijos $T_{i,n}$ del texto, para $i = 1 \ldots n$.

Cada rama del árbol de sufijos representa un substring del texto y cada sufijo de T corresponde a exactamente un camino desde la raíz hasta una hoja, la cual indica la posición en el texto donde comienza el sufijo. La figura 3.10 muestra un ejemplo de un árbol de sufijos.

El árbol de sufijos permite buscar todas las ocurrencias del patrón $P_{1,m}$ en el texto, entrando en el árbol y siguiendo el camino dado por P hasta un nodo, lo que toma tiempo $O(m)$. Todas las hojas descendientes del nodo son sufijos que comienzan con una ocurrencia del patrón.

La construcción del árbol de sufijos toma tiempo lineal en el largo del texto. El árbol de sufijos tiene $O(n)$ nodos. Luego, dado que los substrings de cada rama pueden ser guardados como la posición y el largo de un substring de T, un árbol de sufijos de n nodos ocupa $O(n \log n)$ bits. En la práctica esto es un problema ya que significa un tamaño de almacenamiento de varias veces el texto original, entre 10 a 20 veces. El arreglo de sufijos reduce este factor pero a cambio de perder algunas funciones, lo que se traduce en un mayor tiempo para ciertas operaciones. Las operaciones más comunes sobre árboles de sufijos están listadas en la tabla 2.1.

Operación	Descripción
Root()	Entrega la raíz del árbol.
isleaf(v)	Verdad si v es una hoja.
Locate(v)	Entrega la posición i, si v es una hoja del sufijo $T_{i,n}$.
Ancestor(v, w)	Verdad si v es un ancestro de w.
Count(v)	Entrega el número de hojas que tiene el subárbol de raíz v.
Parent(v)	Entrega el nodo padre de v.
FChild(v)	Entrega el lexicográficamente primer hijo de v.
NSibling(v)	Entrega el nodo hermano que lexicográficamente sigue a v.
Letter(v, i)	Entrega la i-ésima letra del camino desde v hasta la raíz.
SLink(v)	*Suffix-link* de v, sea $\alpha\beta$, $\alpha \in \Sigma$ el camino etiquetado de v, entrega el nodo cuyo camino es β.
LCA(v, w)	Entrega el nodo que corresponde al ancestro común más bajo de v, w.
Child(v, a)	Entrega el nodo w tal que la primera letra de la arista (v, w) es a.

Tab. 2.1: Operaciones sobre los nodos de un árbol de sufijos.

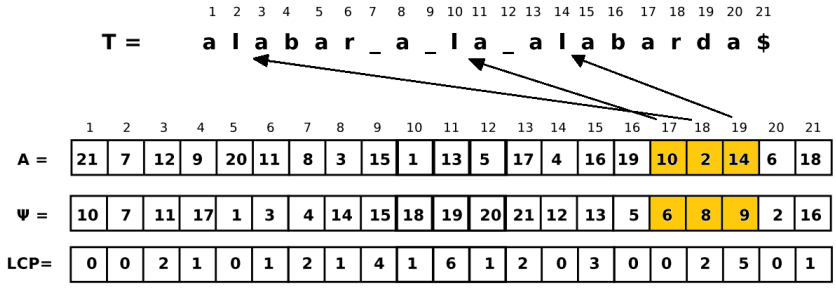

Fig. 2.3: Arreglo de sufijos, arreglo Ψ y arreglo LCP del texto "alabar_a_la_alabarda". El intervalo oscurecido tanto en el arreglo de sufijos como en Ψ corresponde a los sufijos que comienzan con el símbolo "l". [Can10]

2.5. Arreglo de sufijos

Definición 2.5. Un arreglo de sufijos $A[1, n]$ sobre un texto T de largo n, es una permutación tal que $T_{A[i],n} < T_{A[i+1],n}$, donde $<$ es la relación de orden lexicográfico. El arreglo de sufijos representa el conjunto de los n sufijos del texto ordenados lexicográficamente, guardando en cada posición de A el índice del sufijo.

La figura 2.3 muestra un ejemplo de un arreglo de sufijos. Notar que si están las ramas ordenadas por orden lexicográfico, al recorrer las hojas del árbol de sufijos (figura 3.10) de izquierda a derecha se obtiene el arreglo se sufijos.

Encontrar el patrón $P_{1,m}$ en el texto consiste en buscar el intervalo $A[sp, ep]$ que contiene todos los sufijos que comienzan con el patrón. Esto se logra mediante una búsqueda binaria sobre A, buscando lexicográficamente el menor sufijo que parte con P, y otra búsqueda binaria buscando el mayor sufijo que parte con P. En cada comparación de la búsqueda binaria se compara un sufijo con el patrón, lo que toma tiempo $O(m)$ en el peor caso, luego la búsqueda completa toma $O(m \log n)$. Junto con el arreglo de sufijos es necesario guardar el texto, y ambos juntos ocupan $O(n \log n)$ bits.

2.6. Entropía de un texto

El objetivo de compresión de datos es almacenar la información usando menos espacio que su representación plana (descomprimida). Existen, sin embargo, límites a la compresión que pueda lograrse. Una medida

común de la compresibilidad de un texto es su entropía empírica.

Definición 2.6. Para un texto T de largo n cuyos símbolos son parte del alfabeto Σ de tamaño σ, se define la entropía empírica de orden cero como:

$$H_0(T) = \sum_{c \in \Sigma} \frac{n_c}{n} \log \frac{n}{n_c},$$

donde n_c es el número de ocurrencias del símbolo c en T.

$H_0(T)$ representa el número promedio de bits necesarios para representar cada símbolo de T. Luego $nH_0(T)$ es un límite inferior de la compresión que se puede alcanzar sin considerar el contexto del texto, es decir, codificando cada símbolo independientemente del texto que le rodea. Si se considera el contexto de cada símbolo del texto, es posible lograr una mejor compresión.

Definición 2.7. La definición de entropía empírica puede ser extendida para considerar el contexto, dando lugar a la entropía empírica de orden k:

$$H_k(T) = \sum_{s \in \Sigma^k} \frac{|T^s|}{n} H_0(T^s),$$

donde T^s es la subcadena formada por todos los símbolos que ocurren seguidos por el contexto s en T.

Es posible demostrar que $0 \leq H_k(T) \leq H_{k-1}(T) \leq ... \leq H_0(T) \leq \log \sigma$. La entropía empírica de orden k representa una cota inferior del número de bits necesarios para comprimir T considerando un contexto de k símbolos.

2.7. Codificaciones

Usualmente para representar números se usa una cantidad fija de *bits*, independientemente de su valor, por ejemplo 4 *bytes* para representar los enteros. En muchas aplicaciones la mayor parte de los números son pequeños, luego para optimizar el espacio se usan representaciones de largo variable que dependen del número a codificar. Las codificaciones más conocidas son:

Unaria: Representa un positivo n como $1^{n-1}0$, usando exactamente n *bits*.

Gamma(γ): Sea un positivo n, $b(n)$ la representación binaria de n, luego la codificación gamma de n es la concatenación en unario de $|b(n)|$, seguido por $b(n)$, pero sin el bit más significativo. Esto usa $2\lfloor \log n \rfloor + 1$ bits: $\lfloor \log n \rfloor + 1$ para el largo y $\lfloor \log n \rfloor$ para la representación binaria.

Delta(δ): Es una extensión de los códigos gamma que funciona mejor en números mayores. Representa el positivo n como la concatenación de $|b(n)|$ representado con codificación γ, seguido por $b(n)$ pero sin el bit más significativo. Esto usa $\lfloor \log n \rfloor$ (número) $+ 2\log(\log n) + 1$ *bits*.

Vbyte: [WZ99] Particiona los $\lfloor \log(n+1) \rfloor$ *bits* necesarios para representar n en bloques de b *bits* y guarda cada bloque en trozos de $b+1$ bits. El bit más grande es 0 en el trozo que contiene los *bits* más significativos de n y 1 en el resto de los trozos. Por ejemplo, si $n = 25 = 110001$ y $b = 3$, se necesitan dos trozos de 4 *bits,* siendo la representación $0011 \cdot 1001$. Comparado con la codificación óptima de $\lfloor \log(n+1) \rfloor$ *bits*, este código pierde un *bit* por cada b *bits* de n, además de un posible trozo final casi vacío. Comparado con los códigos δ, incluso para la mejor selección de b el espacio usado es mayor, pero a cambio los códigos *Vbyte* son muy rápidos de decodificar.

n	Binario ($b = 4$)	Unario	Código γ	Código δ	Vbyte ($b = 2$)
1	0001	0	0	0	001
2	0010	10	100	1000	010
3	0011	110	101	1001	011
4	0100	1110	11000	10100	001100
5	0101	11110	11001	10101	001101
6	0110	111110	11010	10110	001110
7	0111	1111110	11011	10111	001111

Tab. 2.2: Ejemplo de las distintas codificaciones de enteros.

Estructura	Espacio	Rank	Select
ggmn [GGMN05]	$n + o(n)$	$O(1)$	$O(\log n)$
rrr [CN08]	$nH_0(B) + o(n)$	$O(1)$	$O(\log n)$
darray [OS07]	$n + o(n)$	$O(1)$	$O(\frac{\log^4 m}{\log n})$
sarray [OS07]	$m \log \frac{n}{m} + 2m + o(n)$	$O(\log \frac{n}{m} + \frac{\log^4 m}{\log n})$	$O(\frac{\log^4 m}{\log n})$

Tab. 2.3: Espacio en bits ocupado y tiempo que toma rank y select para distintas estructuras de datos.

En la tabla 2.2 se dan ejemplos de las codificaciones para los números enteros del 1 al 7.

2.8. Bitmaps

Muchas estructuras de datos sucintas se implementan en base a dos funciones básicas, *rank y select*. Dada una secuencia S de símbolos,

- $rank_a(S, i)$ cuenta el número de ocurrencias de a hasta la posición i en S y
- $select_a(S, i)$ encuentra la posición de la i-ésima ocurrencia de a en S.

Las estructuras que soportan estas dos operaciones también implementan la operación $access(S, i)$ que retorna el símbolo en la posición i de la secuencia S. El caso más básico es cuando la secuencia es binaria. Se encuentra un ejemplo de las operaciones en la figura 2.4. Sea $B[1, n]$ una secuencia binaria (*bitmap*) de largo n que contiene m unos. La tabla 2.3 muestra las complejidades de espacio y tiempo de distintas soluciones prácticas.

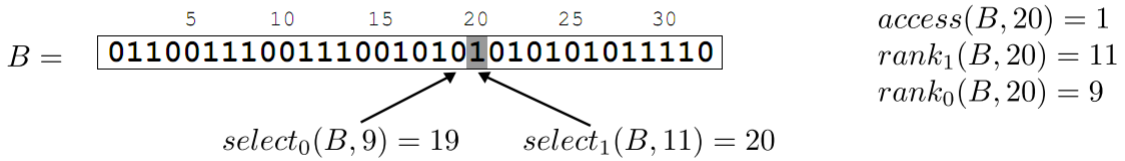

Fig. 2.4: Ejemplo de rank y select. Notar que los valores de ambos $rank_1(B, 20) = 11$ y $rank_0(B, 20) = 9$ suman 20, dado que es la cantidad de unos y ceros.

2.8.1. Munro y Clark

Las primeras estructuras desarrolladas capaces de calcular *rank* y *select* en tiempo constante [Cla98, Mun96] utilizan $n + o(n)$ bits: n bits del arreglo B y $o(n)$ bits adicionales para responder las consultas de *rank* y *select*. Esta solución fue implementada en la práctica por González et al. [GGMN05] soportando *rank* en tiempo constante y *select* en $O(\log n)$, y la llamaremos *ggmn*.

- Para responder *rank* la idea general es dividir el arreglo en bloques de largo fijo y guardar el *rank* relativo al bloque, luego agrupar los bloques en super-bloques, nuevamente guardando el *rank* relativo a éstos, y por último tener pre-computado un muestreo con valores absolutos de *rank*. Esto permite las consultas en tiempo constante.
- Para responder $select(S, i)$, se hace búsqueda binaria en S para encontrar la posición i tal que $rank(S, i) = j$ y $rank(S, i-1) = j-1$, lo que hace uso de la estructura creada para responder *rank* en tiempo constante.

La estructura *ggmn* es fácil de implementar y de rápido acceso en la práctica, pero el espacio extra es aproximadamente $37,5\,\%$ de n [GGMN05], lo que se hace no despreciable. Pero también implementaron la variación de un sólo nivel, ocupando mucho menos espacio.

2.8.2. Raman, Raman y Rao

Raman, Raman y Rao [RRR02] proponen una estructura para *rank* y *select (rrr)* que permite comprimir S de modo que ocupe $nH_0(S)+o(n)$ bits, manteniendo los mismos tiempos de consultas que la estructura anterior. Implementada por Claude y Navarro [CN08], consiste en dividir la secuencia en bloques de largo $u = \frac{\log n}{2}$, donde cada bloque es una tupla (c_i, o_i): c_i representa la clase del bloque, que corresponde al número de $1\acute{}$s que contiene, y o_i es el *offset* del bloque, que corresponde al identificador del bloque dentro de los de su clase c_i. Se definen tres tablas: E, R y O. La tabla E guarda todas las posibles soluciones de *rank* para todas las posibles combinaciones dentro de cada clase y *offset*. La tabla R guarda la concatenación de todos los $c_i\acute{}s$ y O la concatenación de todos los $o_i\acute{}s$.

- Para responder *rank* de i, se calcula $sum(R, \lfloor i/u \rfloor) = \sum_{j=0}^{\lfloor i/u \rfloor} R_j$, que es el número de $1\acute{}$s hasta el bloque que contiene a i, luego se calcula el *rank* hasta la posición i dentro del bloque usando la tabla E e identificando el bloque usando la tabla O.
- Para responder *select* se procede igual que con la estructura anterior, usando búsqueda binaria sobre la secuencia.

2.8.3. Okanohara y Sadakane

Okanohara y Sadakane [OS07] propusieron alternativas basadas en la cantidad m de unos que contiene la secuencia. Proponen dos estructuras: *sarray (sparse-array)* para conjuntos poco densos, óptima para cuando m es chico, y *darray (dense-array)* para conjuntos densos.

Bitmaps densos (*darray*)

Para conjuntos densos con $m \simeq n/2$ unos, dada la secuencia $S[1, n]$ se particiona en bloques de tal manera que cada bloque contenga L unos. Sea el arreglo $P_l[1, n/L]$ tal que $P_l[i] = select(S, iL)$. Se definen los parámetros L_2, L_3 tal que los bloques se pueden clasificar en dos grupos: si el largo del bloque $(P_l[i] - P_l[i-1])$ es mayor

a L_2, se guardan todas las posiciones de los unos del bloque explícitamente en S_l. Si el largo del bloque es menor a L_2, se guarda la posición de los $i \cdot L_3{}'s$ unos en S_s $(i = 0, ..., L_2/L_3)$.

- Para responder $select(S, i)$ se busca $P_l[\lceil i/L \rceil]$, si el bloque es mayor a L_2, el valor que se busca está explícitamente en S_l, de lo contrario se hace una búsqueda secuencial en el bloque a partir de la posición correspondiente que nos entrega S_s.
- Para responder *rank* se ocupa la misma estructura de *ggmn* sobre los bloques guardando el *rank* de cada bloque, permitiendo así responder en tiempo constante.

Bitmaps poco densos (*sarray*)

Dada la secuencia $S[1, n]$ con m 1's ($m \ll n$), se define $x[1, m]$ tal que $x[i] = select(S, i+1)$. Cada x se divide en dos, en los $z = \lceil \log m \rceil$ *bits* superiores y en $w = \lfloor \log \frac{n}{m} \rfloor$ *bits* inferiores. Los *bits* inferiores se guardan explícitamente en $L[1, m]$ usando $m \cdot w$ *bits,* y los superiores se representan con un *bit array* $H[1, 2m]$ tal que $H[x_i/2^w + i] = 1$ y los demás valores 0. H contiene m unos y $2^z = m$ ceros y puede ser visto como la codificación unaria de las distancias entre los 1's de los bits superiores.

- Para responder *select* usando H y L se calcula $select(S, i) = (select(H, i) - i) \cdot 2^w + L[i]$.
- Para responder $rank(S, i)$, primero se calcula $y = select_0(H, i/2^w)+1$ para encontrar el menor elemento que sea mayor a $\lceil i/2^w \rceil \cdot 2^w$. Luego, entre y y el uno que sigue, se cuenta el número de elementos que son iguales o menores a i mediante una búsqueda secuencial en H y L en tiempo $O(\log \frac{n}{m})$.

En esta memoria se usa la implementación de Rodrigo González, disponible en `http://libcds.recoded.cl`

2.8.4. Codificación delta para bitmaps poco densos

Cuando el número m de unos del *bitmap* es bajo, una solución práctica es codificar las distancias entre los unos consecutivos con códigos delta (sección 2.7). Adicionalmente se guarda un muestreo de valores absolutos $select(S, i)$ cada s posiciones y los punteros a la respectivas posiciones de la secuencia δ-codificada. El espacio necesario para la estructura es $W + n/s(\lfloor \log m \rfloor + 1 + \lfloor \log W \rfloor + 1)$, donde W es el número de bits necesarios para representar los códigos δ, que en el peor caso es $W = 2m \left\lfloor \log(\left\lfloor \log \frac{n}{m} \right\rfloor + 1) \right\rfloor + m \left\lfloor \log \frac{n}{m} \right\rfloor + m = m \log \frac{n}{m} + O(m \log \log \frac{n}{m})$.

- Para responder $select(S, i)$ se busca el primer valor muestreado antes que i y luego se decodifica secuencialmente la secuencia δ-codificada hasta i, tomando tiempo $O(s)$.
- Para responder $rank(S, i)$, mediante búsqueda binaria sobre los valores muestreados, se encuentra la última posición l tal que $select(S, l \cdot s) \leq i$. Desde l se decodifica secuencialmente la secuencia δ-codificada hasta p tal que $select(S, p) \geq i$. Luego la consulta toma $O(s + \log \frac{m}{s})$.

La implementación tiene la ventaja que algunas operaciones cuestan $O(1)$ después de haber resuelto otras, por ejemplo $select(S, p)$ y $select(S, p + 1)$ toman $O(1)$ luego de haber resuelto $p \leftarrow rank(S, i)$.

En esta memoria se usara la implementación de Sebastián Kreft [KN10], disponible en `http://pizzachili.dcc.uchile.cl/indexes/LZ77-index`

$C =$ | $C_{1,1}C_{1,2}$ | $C_{2,1}$ | $C_{3,1}C_{3,2}C_{3,3}$ | $C_{4,1}C_{4,2}$ | $C_{5,1}$ | ... |

$A_1 =$ | $C_{1,1}$ | $C_{2,1}$ | $C_{3,1}$ | $C_{4,1}$ | $C_{5,1}$ | ... |
$B_1 =$ | 1 | 0 | 1 | 1 | 0 | ... |

$A_2 =$ | $C_{1,2}$ | $C_{3,2}$ | $C_{4,2}$ | ... |
$B_2 =$ | 0 | 1 | 0 | ... |

$A_3 =$ | $C_{3,3}$ | ... |
$B_3 =$ | 0 | ... |

Fig. 2.5: Ejemplo de la estructura DAC.

2.9. *Directly addressable codes*

Con el objetivo de guardar un arreglo de números enteros ocupando el menor espacio posible, pero proporcionando un rápido acceso aleatorio es que Brisaboa el al. [BLN09] introducen una técnica de reordenamiento de símbolos llamada *directly addressable variable-length codes* (DAC). Basada en la codificación *Vbyte* (sección 2.7), la técnica DAC permite acceder directamente a la i-ésima posición sin necesidad de un método de sampleo necesario para las codificaciones de largo variable.

Sea la secuencia a codificar $C = C_1, ...C_n$ de n enteros, se calcula la codificación *Vbyte* de cada número. Los bloques menos significativos (es decir, los más a la derecha) se guardan en un arreglo A_1 y los *bits* más grandes de los trozos menos significativos son guardados en un *bitmap* B_1. Los trozos restantes son organizados de la misma forma en arreglos A_i y *bitmaps* B_i. Por lo tanto los arreglos A_i guardan los *bits* $(i-1)\cdot b+1, ...i\cdot b$ y los *bitmaps* B_i guardan si es que un número tiene o no más trozos. En la figura 2.5 se puede ver un ejemplo donde se comprime la secuencia $C = C_{1,1}, C_{1,2}\cdot C_{2,1}, C_{3,1}\cdot C_{3,2}\cdot C_{3,3}, C_{4,1}\cdot C_{4,2}, C_{5,1}, ...$, donde cada $C_{i,j}$ es un trozo obtenido al codificar con *Vbyte* el número C_i.

Para obtener el número en la posición i_1,C_{i_1}, se procede de la siguiente forma: Si $B_1[i_1] = 0$, éste es el último trozo, por lo que $C_i = A_1[i]$, de lo contrario hay que buscar los trozos restantes. Se calcula la posición del siguiente trozo, lo cual se sabe contando los números antes que i_1 que tienen más trozos restantes, de tal manera que $i_2 = rank_1(B_1, i_1)$, donde i_2 es el número de unos hasta la posición i_1 en el arreglo B_1 (ver también sección 2.8). Si $B_2[i_2] = 0$, es el último trozo, luego el número es $C_{i_1} = A_1[i_1] + A_2[i_2]\cdot 2^b$, de lo contrario se sigue en los siguientes niveles de igual forma hasta que no hayan más trozos. El peor caso en el que se acceden todos los niveles toma tiempo $O(\log(\text{máx}\, C_i)/b)$.

En esta memoria se usa la implementación de Susana Ladra [BLN09] con la variante que usa distintos valores de b para cada nivel, obteniendo un resultado óptimo en cuanto a espacio usado.

2.10. Estructura de datos sucintas y autoíndices

Una estructura de datos sucinta es una representación comprimida de una estructura, que usa una cantidad de espacio cercana al límite inferior teórico. Una clase importante de estas estructuras sucintas son los índices comprimidos que operan en textos usando un espacio proporcional al del texto comprimido. Las estructuras de índices comprimidos pre-procesan un texto T tal que cualquier patrón de búsqueda P sobre T pueda ser encontrado rápidamente sin la necesidad de un escaneo completo del texto mismo. Si estas estructuras, además de responder consultas, son capaces de reproducir cualquier substring del texto, entonces pueden reemplazar el texto mismo. Los índices con esta propiedad se conocen como autoíndices. Es importante mencionar que las estructuras estudiadas y a proponer en esta memoria son autoíndices.

Definición 2.8. Un autoíndice sobre un texto T ocupa un espacio cercano a su entropía y soporta al menos

las siguientes operaciones básicas eficientemente:

- *count*(P): Cuenta las veces que aparece P como un substring del texto en T.
- *locate(P):* Lista todas las posiciones donde aparece P en T.
- *extract*(i, j): Retorna $T_{i,j}$.

2.11. Arreglo de sufijos comprimido

Un arreglo de sufijos comprimido propuesto por Sadakane [Sad03] representa el arreglo de sufijos $A[1, n]$ y el texto usando la función Ψ. $\Psi(i)$ se define como la posición en el arreglo de sufijos del sufijo que sigue al sufijo $A[i]$ en el texto. Es decir, $\Psi(i)$ dice donde en A esta el valor $A[i] + 1$.

Definición 2.9. $\Psi(i)$ es la posición i' en el arreglo de sufijos donde donde $A[i'] = (A[i] \bmod n) + 1$, que es equivalente a $A[i'] = A[i] + 1$ salvo en el caso $A[i] = n$, donde se cumple $A[i'] = 1$. Es decir,

$$\Psi(i) = A^{-1}[(A[i] \bmod n) + 1].$$

Teorema 2.10. *[GV05]* Ψ *es monótonamente creciente en las áreas que comienzan con el mismo símbolo, es decir* $\Psi(i) < \Psi(i + 1)$ *si se cumple* $T[A[i]] = T[A[i + 1]]$.

La aplicación consecutiva de Ψ permite recorrer el texto secuencialmente de izquierda a derecha. Además, dado que Ψ es monótonamente creciente en las áreas que comienzan con el mismo símbolo (ver figura 2.3), se puede representar guardando las diferencias codificadas $\Psi(i) - \Psi(i - 1)$.

Esta estructura es muy rápida para la consulta *count,* sin embargo una de sus desventajas es que para la consulta *locate* necesita un muestreo del arreglo de sufijos, que necesita $(n \log n)/s$ *bits* extra para lograr un tiempo de $O(s)$, donde s es el paso de muestreo.

La estructura requiere de $nH_0(T) + o(n \log \sigma)$ *bits* de espacio en la práctica. El tiempo para contar el número de ocurrencias de un patrón P es $O(|P| \log n)$, el tiempo para encontrar una ocurrencia de P es $O(\log^{1+\epsilon} n)$, y el tiempo de extraer una porción del texto de largo l es $O(l + \log^{1+\epsilon} n)$, donde $0 < \epsilon \leq 1$ es una constante arbitraria que depende del paso de muestreo, $s = \log^{\epsilon} n$.

2.12. La Transformada de Burrows-Wheeler (BWT) y el FM-index

Backward search es un método alternativo para encontrar el intervalo $[sp, ep]$ en el arreglo de sufijos que contiene los sufijos que parten con el patrón P. Busca el intervalo considerando P de derecha a izquierda, a diferencia del método clásico, y está basado en la transformada de *Burrows-Wheeler (BWT) [BW94]. La BWT* de un texto T es una permutación del texto que suele ser más compresible que T.

Definición 2.11. Dado un texto T, y su arreglo de sufijos $A[1, n]$, la transformada de *Burrows-Wheeler* de T es definida como $T^{bwt}[i] = T[A[i] - 1]$, excepto cuando $A[i] = 1$, donde $T^{bwt}[i] = T[n]$.

Es decir, T^{bwt} se forma recorriendo secuencialmente el arreglo de sufijos A, y concatenando el carácter que precede a cada sufijo.

Otra forma de ver la transformada de Burrows-Wheeler es tomando todas las secuencias de la forma $T_{i,n}T_{1,i-1}$ ordenadas lexicográficamente. Si se considera una matriz M con todas estas secuencias, la última columna corresponde a T^{bwt}. La figura 2.6 muestra un ejemplo de cómo M es creado para el texto "alabar_a_la_alabarda$".

alabar_a_la_alabarda$		**$**alabar_a_la_alabarda
labar_a_la_alabarda$a		**_a_la_alabarda$**alabar
abar_a_la_alabarda$al		**_alabarda$**alabar_a_la
bar_a_la_alabarda$ala		**_la_alabarda$**alabar_a
ar_a_la_alabarda$alab		**a$**alabar_a_la_alabard
r_a_la_alabarda$alaba		**a_alabarda$**alabar_a_l
_a_la_alabarda$alabar		**a_la_alabarda$**alabar_
a_la_alabarda$alabar_		**abar_a_la_alabarda$**al
_la_alabarda$alabar_a		**abarda$**alabar_a_la_al
la_alabarda$alabar_a_		**alabar_a_la_alabarda$**
a_alabarda$alabar_a_l	$\rightarrow$	**alabarda$**alabar_a_la_
_alabarda$alabar_a_la		**ar_a_la_alabarda$**alab
alabarda$alabar_a_la_		**arda$**alabar_a_la_alab
labarda$alabar_a_la_a		**bar_a_la_alabarda$**ala
abarda$alabar_a_la_al		**barda$**alabar_a_la_ala
barda$alabar_a_la_ala		**da$**alabar_a_la_alabar
arda$alabar_a_la_alab		**la_alabarda$**alabar_a_
rda$alabar_a_la_alaba		**labar_a_la_alabarda$**a
da$alabar_a_la_alabar		**labarda$**alabar_a_la_a
a$alabar_a_la_alabard		**r_a_la_alabarda$**alaba
$alabar_a_la_alabarda		**rda$**alabar_a_la_alaba

Fig. 2.6: Matriz M para la transformada de Burrows Wheeler del texto "alabar_a_la_alabarda$". La última columna corresponde a la transformada, $L =$ "araadl_ll$_bbaar_aaaa". Notar que la primera columna de la matriz ordenada, F, corresponde al arreglo de sufijos.

Definición 2.12. La función $LF(i)$ mapea una posición i de la última columna de M, $L = T^{bwt}$ a su aparición en la primera columna de M, F.

Lema 2.13.

$$LF(i) = C[c] + rank_c(T^{bwt}, i)$$

donde $c = T^{bwt}[i]$ *y* $C[c]$ *es el número de símbolos lexicográficamente menores que* c *en* T*.*

Dada la directa relación entre BWT y el arreglo de sufijos, ya que M corresponde al arreglo de sufijos (mirando la primera letra de cada sufijo), se construye un autoíndice llamado FM-index [BW94] que utiliza *backward search* sobre T^{bwt} para buscar P en $O(|P|)$ operaciones de *rank*. Lo que hace es, en cada paso, re-calcular el intervalo$[sp, ep]$ del arreglo de sufijos en que los sufijos parten con $P[i, m]$. En algoritmo 1 se detalla *backward search*.

Por ejemplo, el FM-*index* de de Ferragina et al. [FMMN07] ocupa$nH_k(T) + o(n \log \sigma)$ bits para $k \leq \alpha \log_\sigma n$ y $0 < \alpha < 1$ constante. El tiempo para contar el numero de ocurrencias de un patrón P es $O(|P| \log \sigma)$, el tiempo para encontrar una ocurrencia de P es $O(\log^{1+\epsilon} n)$, y el tiempo de extraer una porción del texto de largo l es $O(l \log \sigma + \log^{1+\epsilon} n)$, donde $0 < \epsilon$ es una constante arbitraria que depende del paso se muestro $s = \frac{\log^{1+\epsilon} n}{\log \sigma}$.

Algoritmo 1 Algoritmo Backward search (BWS)

BWS(P)
1. $i \leftarrow len(P)$
2. $sp \leftarrow 1$
3. $ep \leftarrow n$
4. **while** $sp \leq ep$ **and** $i \geq 1$ **do**
5. $\quad c \leftarrow P[i]$
6. $\quad sp \leftarrow C[c] + rank_c(T^{bwt}, sp-1) + 1$
7. $\quad ep \leftarrow C[c] + rank_c(T^{bwt}, ep)$
8. $\quad i \leftarrow i - 1$
9. $\quad$ **if** $sp > ep$ **then**
10. $\quad\quad$ **return Ø**
11. **return** (sp, ep)

2.13. Longest common prefix (LCP)

Sea $lcp(x, y)$ la longitud del prefijo común más largo entre dos strings x e y. Los lcps entre sufijos que están adyacentes en el arreglo de sufijos $A[1, n]$ definen el arreglo $LCP[1, n]$.

Definición 2.14. $LCP[1] = 0$ y $LCP[i] = lcp(T_{A[i-1],n}, T_{A[i],n})$ para $i > 1$.

Kasai et al. [KLAAP01] mostraron que se puede simular el recorrido del árbol de sufijos, usando el arreglo de sufijos A y el arreglo LCP. Más aún, Sadakane [Sad07] mostró que se pueden simular todas las operaciones del árbol de sufijos usando los arreglos A y LCP, sin necesidad de la topología del árbol. $LCP[i]$ corresponde a la longitud del camino desde la raíz hasta el nodo LCA de las hojas i y $i-1$ (ver tabla 2.1). Por ejemplo en la figura 2.4 $LCP[18] = 2$ dado que $lcp(T_{A[17],n}, T_{A[18],n}) = lcp(\text{“la\_alabarda\$”,“labar\_a\_la\_alabarda\$”}) = 2$. Por lo tanto en el árbol de la figura 2.2, el LCA de las hojas 2 y 10 tiene una profundidad de 2, correspondiente a la etiqueta “la”.

2.14. Colecciones repetitivas

Una colección repetitiva de secuencias C es un conjunto de r secuencias que contienen pequeñas variaciones unas con otras. Un ejemplo práctico son secuencias de genoma para distintos individuos de la misma especie.

Para modelar una colección repetitiva se define una mutación como el evento de que un símbolo se cambie por uno distinto dentro de un string, se elimine, o un nuevo símbolo se inserte.

Estas colecciones pueden ser indexadas usando los métodos usuales de autoíndices para la concatenación $T = T^1\#T^2\#...T^r\$$, donde $\# \neq \$$ es un símbolo especial que no pertenece a Σ. Sin embargo, incluso con compresores de alta entropía los resultados no son atractivos para estas colecciones. Esto ocurre porque la compresión de orden k que los autoíndices logran, está definida por las frecuencias de los símbolos en un contexto local de largo k (k pequeño en comparación con el largo de una secuencia), contextos que no cambian al agregar secuencias idénticas a la colección. Por ende estos autoíndices no explotan el hecho que las secuencias de la colección son altamente similares.

2.15. Re-Pair: Compresión basada en gramáticas

Re-pair [LM00] es un método de compresión basado en gramáticas que factoriza repeticiones en una secuencia y las guarda en un diccionario, permitiendo descompresión rápida y local. Consiste en reemplazar

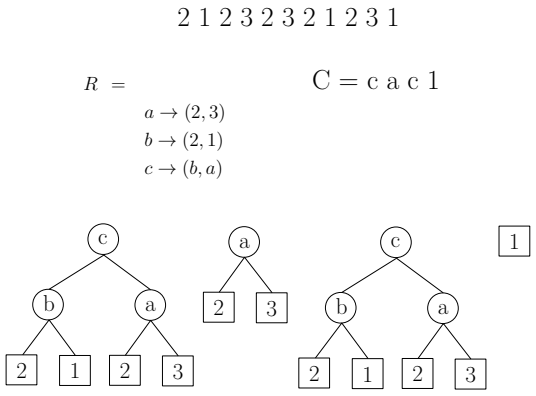

Fig. 2.7: Ejemplo de compresión usando Re-pair de una secuencia de números, representado como una secuencia de árboles.

las parejas de símbolos ab que se repiten más frecuentemente por un nuevo símbolo s. Esto se repite hasta que ya no se repitan parejas de símbolos. Dado un texto T sobre el alfabeto Σ de tamaño σ, es algoritmo es el siguiente:

1. Identificar la pareja de símbolos ab más frecuente en T.
2. Crear un nuevo símbolo s, lexicográficamente mayor a todos los existentes en T, y añadir la regla $s \to ab$ al diccionario R.
3. Reemplazar toda ocurrencia de ab en T por s.
4. Iterar hasta que cada pareja tenga frecuencia 1.

El resultado de la compresión es el diccionario R y la nueva secuencia C con los símbolos nuevos y originales. Cualquier porción de C puede ser descomprimida fácilmente: primero se verifica si $C[i] < \sigma$. De ser así, el símbolo es el original y se emite, en caso contrario se busca la regla $C[i] \to ab$ en R, y se expande recursivamente siguiendo los mismos pasos.

El diccionario R corresponde a una gramática libre de contexto y la nueva secuencia comprimida C corresponde a los símbolos iniciales de los cuales se deriva la secuencia original. Asimismo una derivación puede ser representada mediante un árbol sintáctico, donde las hojas representan los símbolos terminales, y cada nodo del árbol binario corresponde a una regla de R. Por último, la estructura completa puede ser vista como una secuencia de árboles donde cada símbolo en C es la raíz de un árbol que representa un intervalo de la secuencia original.

En la figura 2.7 se muestra un ejemplo de esta representación para una secuencia de números. Notar que si uno recorre de izquierda a derecha las hojas (señaladas con un cuadrado) de cada árbol en $C = 1\,c\,b\,c$ se obtiene la secuencia original. Además se muestra el diccionario de reglas que se crea según el algoritmo visto, que es la representación de la reglas.

El problema de construir la gramática óptima es *NP-completo,* luego *Re-Pair* es una heurística de tiempo lineal. Recorre el texto secuencialmente guardando todos los pares de símbolos y sus frecuencias en un *heap*. Luego extrae el par con mayor frecuencia del *heap* y lo reemplaza en el texto. Al mismo tiempo extrae las nuevas reglas que surgen al reemplazar por el nuevo símbolo que identifica a la regla. Notar que para obtener los nuevos pares y saber si cambió la frecuencia de los antiguos, sólo basta mirar localmente donde se reemplazó el par por la regla obtenida. Se procede así hasta que no queden pares con frecuencia

mayor a 1. En vez de recorrer secuencialmente el texto es posible acelerar el proceso marcando con punteros dónde están los pares extraídos para reemplazarlos en tiempo constante.

Análisis del espacio

A pesar del buen desempeño que *Re-Pair* ha demostrado en la práctica mediante resultados experimentales, el método ha resistido mayores intentos de análisis formales. Sin embargo, Navarro y Russo [NR08] logran obtener una cota superior importante. Usan el siguiente teorema demostrado por Kosaraju y Manzini.

Teorema 2.15. *[KM99] Sea $y_1, ..., y_t$ una partición del texto $T[1, n]$ sobre el alfabeto σ, tal que cada frase y_i aparece a lo más b veces en T. Para cualquier $k > 0$ se tiene*

$$t \log t \leq nH_k(T) + t \log(n/t) + t \log b + \Theta(t(1 + k \log \sigma))$$

Luego notando que, después de aplicar *Re-Pair,* cada par de símbolos consecutivos es único en el texto comprimido, y que al reemplazar todos los pares que se repiten más de b veces, cada par de símbolos consecutivos aparece a lo más b veces, se obtiene el siguiente teorema

Teorema 2.16. *[NR08] Sea $T[1, n]$ un texto sobre un alfabeto de tamaño σ con una entropía empírica de orden k ,$H_k(T)$. Luego, el algoritmo de compresión Re-Pair logra una representación usando a lo más $2nH_k(T) + o(n \log \sigma)$ bits para cualquier $k = o(\log_\sigma n)$ (implica que $\log \sigma = o(\log n)$, a menos que $k = 0$).*

En secuencias altamente repetitivas, *repair* comprime a espacios menores que la entropía empírica, por lo que es mejor tomar $|R|$ y $|C|$ como medida de compresibilidad.

Capítulo 3

Representaciones Sucintas de Árboles de Sufijos

3.16. Árboles sucintos

Una representación clásica de un árbol con n nodos utiliza $O(n)$ punteros, y como cada puntero debe distinguir todos los nodos, requiere $\log n$ bits. Por lo tanto el árbol ocupa $\Theta(n \log n)$ *bits*, lo que trae problemas para manipular grandes árboles. Afortunadamente la cota inferior teórica de información para guardar un árbol cuyos nodos tienen un orden es $2n - \Theta(\log n)$ dado que existen $\binom{2n-1}{n-1}/(2n-1) = 2^{2n}/\Theta(n^{3/2})$ distintos árboles. Esto ha dado lugar a muchas representaciones sucintas que logran $2n + o(n)$ bits de espacio, las cuales difieren en su funcionalidad. Mientras que algunas representaciones sólo soportan operaciones básicas de navegación como *child* y *parent*, otras permiten toda una gama de operaciones más avanzadas (ver tabla 2.1 con algunas de las operaciones generales sobre árboles).

Las tres clasificaciones más usadas de tipos de representaciones sucintas son: *balanced parentheses* (BP), *level-order unary degree sequence* (LOUDS), y *depth-first unary degree sequence* (DFUDS), todas ocupan $2n + o(n)$ bits de espacio y respondiendo la mayoría de las consultas en tiempo constante. La figura 3.8 muestra un ejemplo de estas representaciones sucintas para un árbol.

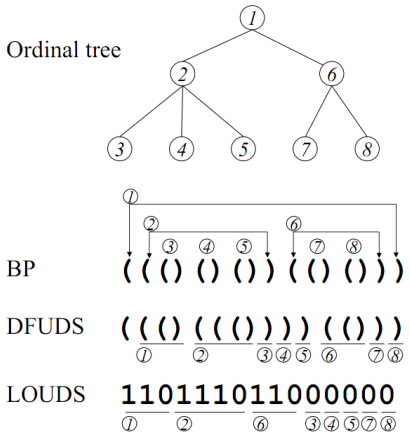

Fig. 3.8: Ejemplo de representaciones sucintas basadas en paréntesis balanceados de un árbol.[SN10]

BP [Jac89]: Se obtiene haciendo un recorrido en profundidad en preorden del árbol y poniendo un paréntesis abierto por cada nodo visitado y un paréntesis cerrado cuando se han visitado todos los descendientes del nodo. De esa manera se obtiene una secuencia de $2n$ paréntesis balanceados, donde cada nodo x queda representado por un par de paréntesis balanceados y el subárbol de x queda representado por la secuencia de paréntesis balanceados dentro de los que representan a x.

DFUDS [BDM05]: Representa un nodo de grado (número de hijos) d con d paréntesis abiertos seguido de un paréntesis que cierra. Los nodos se listan haciendo un recorrido en profundidad en preorden del árbol, resultando en una secuencia balanceada.

LOUDS [JAC89, BDM05]: Los nodos son procesados y representados al igual que con *DFUDS*, salvo que los nodos se listan haciendo un recorrido por niveles en vez de profundidad. Este consiste en listar los

Operación	BP	Descripción
isleaf(x)	$P[x+1] = ')'$	*Si el nodo x es una hoja.*
depth(x)	$excess(x)$	*Profundidad del nodo x.*
ancestor(x, y)	$x \leq y \leq findclose(x)$	*Si x es ancestro de y.*
subtree_size(x)	$(findclose(x) - x + 1)/2$	*Número de nodos en el sub-árbol de x.*
parent(x)	$enclose(x)$	*Nodo padre de x.*
FChild(x)	$x + 1$	*Primer hijo (más a la izquierda) de nodo x.*
NSibling(x)	$findclose(x) + 1$	*Siguiente nodo hermano de x.*
child(x, i)		*i-esimo hijo de nodo x.*
degree(x)		*Número de hijos que tiene nodo x.*

Tab. 3.4: Operaciones sobre árboles, algunas reducidas en tiempo constante a las operaciones básicas de una secuencia balanceada de paréntesis *BP*. Algunas operaciones no soportan reducciones en tiempo constante, pero es fácil ver que pueden ser respondidas usando otras operaciones sobre árboles, por ejemplo, $child(x, i)$ se resuelve en tiempo $O(i)$ aplicando $Fchild(x)$ y luego $i - 1$ veces $Nsibling(y)$.

nodos poniendo la raíz primero, luego todos sus hijos de izquierda a derecha, luego todos los nietos y así sucesivamente. El nodo se representa por su grado codificado en unario como en *DFUDS*. Notar que el árbol es unívocamente determinado por su secuencia de grados.

Todos estas representaciones ocupan básicamente el mínimo teórico de información, soportando varias operaciones de navegación en tiempo constante, pero distintas representaciones soportan diferentes operaciones. *LOUDS* carece de operaciones básicas como saber el tamaño del subárbol de un nodo. BP soporta muchas otras operaciones [Jac89]. *DFUDS* es similar a *BP*, y se creó para permitir operaciones adicionales como obtener el i-ésimo hijo en tiempo constante [BDM05], pero actualmente *BP* soporta las mismas operaciones [SN10]. Otro tipo de representación de árboles sucintos se basa en tree covering [HMR07], que también soporta operaciones en tiempo constante, sin embargo ocupa un espacio mayor que las anteriores.

La mayoría de las operaciones sobre árboles se basan en las siguientes operaciones básicas definidas sobre una secuencia de paréntesis S:

- *findclose*(S, i): Posición del *')'* que cierra el paréntesis en la posición i en la secuencia S.
- *findopen*(S, i): Posición del *'('* que abre el paréntesis en la posición i en la secuencia S.
- *enclose*(S, i): Posición del paréntesis más cercano que encierra al nodo i.
- excess(S, i): Cantidad de paréntesis '(' que no están cerrados hasta la posición i en la secuencia S.

En la tabla 3.4 se pueden ver algunas reducciones a tiempo constante de las operaciones sobre árboles ocupando la operaciones básicas previamente definidas para representaciones basadas en *BP*. Algunas operaciones no soportan reducciones en tiempo constante, sino que requieren sus propios algoritmos. En *DFUDS*, en cambio, todas las operaciones de la tabla 3.4 se pueden reducir a las operaciones de los paréntesis.

3.16.1. Range min-max tree

Recientemente Sadakane y Navarro [SN10] propusieron una ingeniosa representación basada en *BP* y en una nueva estructura, *range min-max tree,* que permite resolver en tiempo constante muchas operaciones que no son admitidas en las otras representaciones y sin agregarle demasiada información a la estructura, ocupando en total $2n + O(n/\text{polylog}(n))$ bits de espacio.

El *range min-max tree* de una secuencia de paréntesis balanceados $P[1,n]$ particiona P en bloques de tamaño b, y para cada bloque guarda el máximo y mínimo exceso de paréntesis abiertos en el bloque, donde los excesos se calculan como $excess(i) = rank_{(}(P,i) - rank_{)}(P,i)$. Luego se agrupan k bloques y se calcula el mínimo y máximo exceso para cada agrupación, siendo ;este el segundo nivel del árbol. Se agrupan de nuevo estos bloques y así sucesivamente hasta formar el árbol completo, ocupando $O(n\log(b)/b) = o(n)$ *bits*, si $b = \omega(1)$.

Para soportar las operaciones básicas es necesario definir las siguientes funciones:

Definición 3.17. Dada una secuencia P de paréntesis, se define

- *excess(i)* $= rank_{(}(P,i) - rank_{)}(P,i)$
- *excess(i,j)* $= excess(j) - excess(i-1)$
- *fwd_search*(i,x)= mínimo $j > i$ tal que $excess(i,j) = x$
- *bwd_search*(i,x)= máximo $j < i$ tal que $excess(j,i) = x$

Para resolver *fwd_search*(i,x) se escanea el bloque i en busca de la posición j más a la izquierda tal que $excess(i,j) = x$. De no encontrarse, se sube un nivel en el árbol y dado que este nodo contiene el rango *min/max* de excesos de los nodos hijos, se busca el primer nodo cuyo rango contenga x. Una vez encontrado se baja a sus bloques para encontrar el j exacto. Si no se encuentra un rango apropiado en el nodo, se recorre su padre, y así recursivamente. De forma análoga se resuelve *bwd_search*(i,x).

Con estas funciones se pueden implementar las operaciones básicas, así como otras más sofisticadas, sobre la secuencia de paréntesis que permiten simular el árbol, en particular:

$$\begin{aligned} findclose(x) &\equiv fwd\_search(x,0) \\ findopen(x) &\equiv bwd\_search(x,0) \\ enclose(x) &\equiv bwd\_search(x,2) \end{aligned}$$

En la figura 3.9 hay un ejemplo de un *range min-max tree*. Para encontrar el paréntesis que cierra al tercer paréntesis (partiendo de 0), se busca el primer $i > 3$ tal que $E[i] = E[3-1] + 0 = 1$, donde E son los bloques. Se comienza en el nodo $[3/b] = 1$ que corresponde al nodo d, bloque que no contiene 1 por lo que se avanza al nodo e, el cual tampoco puede contener a 1. Luego se examina el nodo f , y dado el rango, este sí contiene a 1, por lo tanto se escanean los hijos de f en busca del primer 1, que corresponde al nodo h y al paréntesis en la posición 12.

Arroyuelo et al. [ACNS10] compararon experimentalmente todas las representaciones anteriores de árboles sucintos, siendo la de *range min-max tree* la que en promedio ofrece una excelente combinación de uso de espacio, rendimiento en tiempo y funcionalidad. Implementa un amplio conjunto de operaciones sencillas y sofisticadas, y en la mayoría de los casos supera a todas las otras estructuras en tiempo y espacio.

3.17. Árboles de sufijos comprimidos

Los árboles de sufijos comprimidos son representados con un arreglo de sufijos comprimido, junto con información de los prefijos comunes más largos (LCP) del texto, además de información de la topología del árbol, para poder simular los recorridos por los nodos.

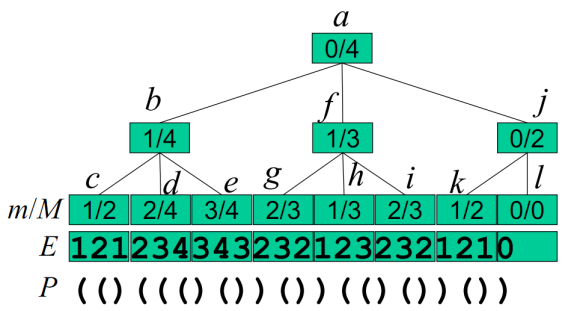

Fig. 3.9: Ejemplo de un range min-max tree.[SN10]

3.17.1. Árbol de sufijos de Sadakane

El árbol de sufijos de Sadakane [Sad07] usa $O(n)$ *bits* sobre el arreglo de sufijos comprimido, y soporta todas las operaciones de navegación sobre el árbol de forma eficiente, lo que es una gran mejora comparándolo con las estructuras vistas anteriormente, que ocupan $O(n \log n)$ *bits*. La figura 3.10 muestra un ejemplo con los componentes básicos de un árbol de sufijos de Sadakane para el texto "alabar_a_la_alabarda$" mostrando el arreglo LCP y A sin comprimir.

Para la topología utiliza secuencias de paréntesis balanceados (BP) y se usa el arreglo LCP comprimido en un *bitmap* H. El arreglo LCP codificado ocupa a lo más $2n$ bits, basado en el hecho que al ordenarse según las posiciones del texto, contrario a ordenar por los sufijos, los números consecutivos del LCP decrecen en a lo más 1. Por lo tanto se cumple lo siguiente:

Lema 3.18. *[Sad07] Los valores $i + LCP[i]$ son no decrecientes al ordenarse según las posiciones del texto, luego la secuencia $S = s_1, ..., s_{n-1}$, donde $s_j = j + LCP[A^{-1}[j]]$, es no decreciente.*

Para representar la secuencia S, dado que es no decreciente, se codifica cada diferencia $diff(j) = s_j - s_{j-1}$ en unario como $0^{diff(j)}1$ y se asume que $s_0 = 0$. De esa manera que ocurren runs de 1's y 0's que pueden ser codificadas en un *bitmap* H que representa el arreglo LCP. Se tiene que $diff(j) = select_1(H, j) - select_1(H, j-1) - 1$, por lo tanto $LCP[i] = select_1(H, A[i] + 1) - 2A[i] - 1$. En la figura 3.10 hay un ejemplo de H construido para el texto "alabar_a_la_alabarda$", donde LCP' es el LCP ordenado según el texto, es decir $LCP'[i] = LCP[SA[i]]$, y además muestra un ejemplo de cómo computar $LCP[i]$ a partir de H.

Por último, existe un tercer arreglo $P'[i] = rank_(P, i) - rank_)(P, i)$ sin el cual no es posible implementar algunas operaciones sobre el árbol. Pero no es necesario guardarlo en memoria, ya que es computable en tiempo constante usando P. Sobre P' se aplica la siguiente función:

Definición 3.19. Para indices l y r entre 1 y n, para un arreglo S, la función *range minimun query* $RMQ_S(l, r)$ retorna el índice del menor elemento en el subarreglo $S[l, r]$.

A continuación se explica cómo están implementadas las operaciones descritas en la tabla 2.1 para el árbol de Sadakane:

Root: La raíz corresponde al primer paréntesis abierto de P, por lo que siempre retorna la posición 1.

isleaf(v): Verdad si $P(v + 1) =')'$, falso de lo contrario.

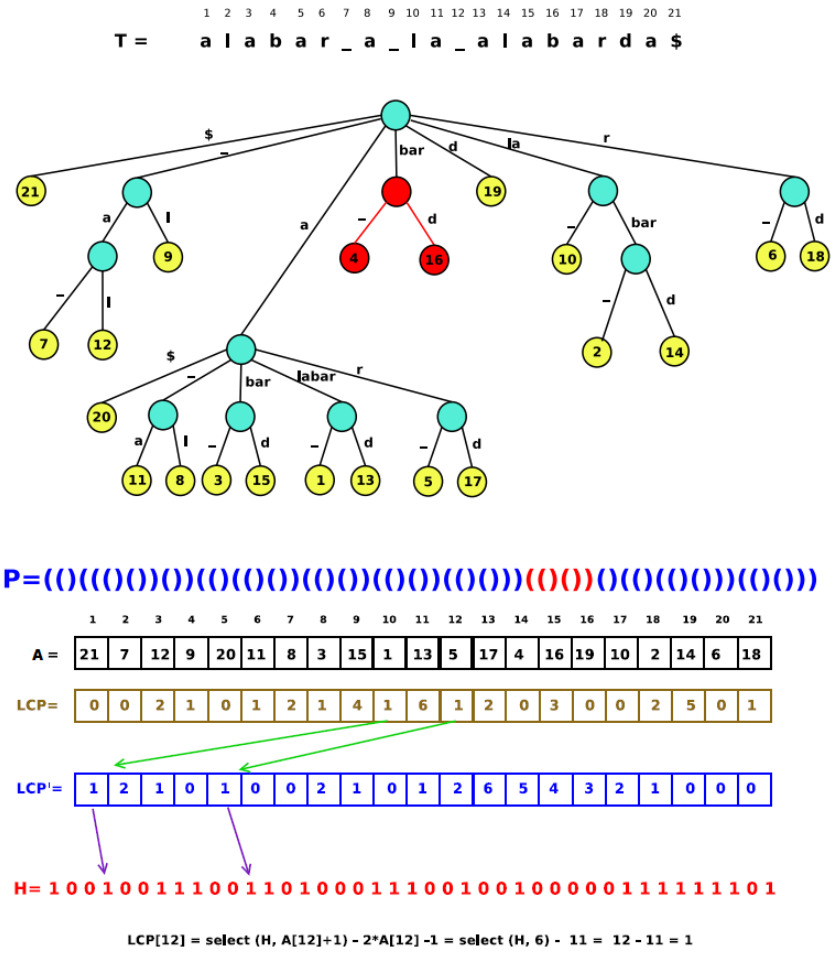

Fig. 3.10: Árbol de sufijos de Sadakane para el texto “alabar_a_la_alabarda$”. [CN10]

Locate(v): Si $P(v+1) = ')'$ quiere decir que v es una hoja, por lo que se retorna $w = rank_{()}(P, v)$. En caso contrario se retorna -1. Para computar $rank_{()}$, Välimäki et al. [VGDM07] agregaron un bitmap $brLeaf$ donde los '()' de P están marcados, luego $rank_{()}(P, v) = rank_1(brLeaf, v)$.

Ancestor(v, w): Si $v \leq w \leq findclose(P, v)$ retorna *true*, de lo contrario retorna *false*.

SDepth(v): Si $v = 1$ retorna 0, si v es una hoja retorna $n - Locate(v)$. De lo contrario responder $LCP[i+1]$, donde $i = rank_{()}(P, findclose(P, v+1))$.

TDepth(v): El número de paréntesis abiertos menos los cerrados hasta la posición v, retorna $rank_{(}(P, v) - rank_{)}(P, v) = 2rank_{(}(P, v) - v$.

Count(v): Retorna $rank_{()}(P, findclose(P, v)) - rank_{()}(P, v)$.

Parent(v): Retorna $w = enclose(v)$, al menos que v sea la raíz, en cuyo caso retornar -1.

FChild(v): Retorna $v + 1$, al menos que v sea hoja, en cuyo caso retornar -1.

NSibling(v): Sea $w = findclose(P, v) + 1$. Si $P(w) = ')'$ entonces el nodo v no tiene siguiente hermano y se retorna -1, de lo contrario retorna w.

SLink(v): Primero se encuentran las hojas de los extremos del subárbol de raíz v, $x = rank_{()}(P, v-1)+1$ e $y = rank_{()}(P, findclose(P, v))$. Se retorna $LCA(select_{()}(P, \Psi(x)), select_{()}(P, \Psi(y)))$.

SLink^i(v): Igual que el método anterior pero en vez de calcular $\Psi(z)$ se usa $\Psi^i(z) = A^{-1}[((A[z]+1) mod n)+1]$.

LCA(v, w): Si uno de los nodos es ancestro del otro entonces se responde con su ancestro. De lo contrario retornar $Parent(RMQ_{P'}(v, w)+1)$. $RMQ_{P'}$ entrega el índice m del menor elemento en $P'[v, w]$, luego se cumple $P[m] =')'$ y $P[m+1] =' ('$, y $P[m+1]$ corresponde a un hijo de $LCA(v, w)$.

Child(v, a): De forma iterativa se consiguen todos los hijos de v, en cada paso se compara el primer símbolo de la arista entre v y el hijo w, comparándolo con a.

3.17.2. Árbol de sufijos de Russo et al.

Russo et al. [RNO11] proponen una nueva representación del árbol de sufijos llamada *FCST* (Fully-Compressed Suffix Tree). Esta representación reemplaza la estructura de paréntesis balanceados que describen la topología, identificando nodos del árbol de sufijos v por el correspondiente intervalo en el arreglo de sufijos $v = [v_l, v_r]$. La idea principal, es tener un muestreo de nodos del árbol y mediante el arreglo de sufijos obtener los nodos cercanos a los muestreados. Por lo tanto, la representación de árbol se construye solamente sobre el arreglo de sufijos comprimido. Esto permite que el espacio que ocupa la representación sea muy bajo, siendo el primero en romper la barrera de los $\Theta(n)$ bits. Sin embargo, la estructura es lenta, en general por un factor de $O(\log n (\log \sigma + \log \log n))$ sobre el árbol de Sadakane.

Definición 3.20. Un δ-muestreo S de un árbol de sufijos τ con t nodos se forma escogiendo $s = O(t/\delta)$ nodos de τ, tal que para cada nodo v de τ, hay un entero $i < \delta$ tal que el nodo $SLink^i(v)$ es parte del muestreo.

Como se menciono el árbol de sufijos comprimido de Russo et al. consiste en el arreglo de sufijos comprimido y en el δ-muestreo del árbol de sufijos. Notar que partiendo de un nodo v y aplicando recursivamente *SLink,* se obtiene la muestra en a lo mas δ pasos.

Para hacer uso del δ-muestreo, es necesario mapear cualquier nodo v del árbol de sufijos, a su *lowest sampled ancestor* $LSA(v) \in S$. Otra operación importante es *lowest common sampled ancestor* $LCSA(u, w) = LSA(LCA(v, w))$. Ambas soportadas en tiempo constante, usando $O((n/\delta) \log \delta)$ bits extra de espacio. Además, sobre todos los nodos muestreados v se guardan los valores $SDepth(v)$ y $TDepth(v)$.

Las operaciones de navegación se basan en una extensión de la función LF definida en la sección **2.12.**

Definición 3.21. $LF(X, v)$, donde X es un string y v una posición del arreglo de sufijos, entrega el ranking lexicográfico del sufijo $X.T_{A[v],n}$ dentro de todos los sufijos. La definición se puede extender a un nodo v de árbol de sufijos como: $LF(X, v) = LF(X, [v_l, v_r]) = [LF(X, v_l), LF(X, v_r)]$.

A continuación se explica cómo están implementadas las operaciones descritas en la tabla 2.1 para el árbol de Russo et al.

Root/Count/Ancestor: La raíz es representada por el intervalo $[1, n]$, $Count(v)$ es $v_r - v_l + 1$ y $Ancestor(v, w)$ es verdad sii $w_l \leq v_l \leq v_r \leq w_r$.

Locate(v): Si v es una hoja ($v_l = v_r$) retorna $A[v_l]$, de lo contrario -1.

SDepth(v): Si v es un nodo de la muestra, la solución ya esta guardada. De lo contrario se retorna $max_{0 \leq i \leq \delta}\{i + SDepth(LCSA(\Psi^i(v_l), \Psi^i(v_r)))\}$. Ψ se puede computar como la inversa de LF.

LCA(v, w): Si alguno v o w es ancestro del otro, retornar el ancestro. De lo contrario, dado que $LCA(v, w)$ es equivalente a $LCA(min\{v_l, w_l\}, max\{v_r, w_r\})$. Se retorna $LF(v[0, i-1], LCSA(\Psi^i(min\{v_l, w_l\}), \Psi^i(max\{v_r, w_r\})))$, donde i es el valor que maximiza $SDepth(v)$.

Parent(v): Si v es la raíz, retorna -1. En caso contrario, retorna $LCA(v_l - 1, v_l)$ o $LCA(v_r, v_r + 1)$, el que sea menor.

SLink(v): Retorna $LCA(\Psi(v_l), \Psi(v_r))$.

SLink$^i(v)$: Retorna $LCA(\Psi^i(v_l), \Psi^i(v_r))$.

TDepth(v): Es necesario añadir más nodos al muestreo, para garantizar que, para cualquier nodo v, $Parent^j(v)$ está en la muestra para algún $0 \leq j < \delta$. Luego $TDepth(v)$ retorna $TDepth(LSA(v)) + j$, donde $TDepth(LSA(v))$ está almacenado y $j < \delta$ se obtiene iterando la operación *Parent* desde v hasta llegar a $LSA(v)$.

LAQ$_S(v, d)$: Mediante búsqueda binaria se encuentra el valor i tal que $v' = Parent^i_S(SLink^{\delta-1}(v))$ y $SDepth(v') \geq d-(\delta-1) > SDepth(Parent_S(v')$. Luego se encuentra el nodo $v_{i,j} = Parent^j_S(LSA(LF(v[i..\delta,1], v')))$, tal que $v_{i,j}$ minimice $SDepth(v_{i,j}) - (d - i) \geq 0$, y se le aplica la función LF.

FChild(v): Si v es una hoja, retorna -1. En caso contrario retorna $LAQ_S(v_l, SDepth(v) + 1)$.

NSibling(v): Se v es la raíz, retorna -1. En caso contrario retorna $LAQ_S(v_r + 1, SDepth(Parent(v)) + 1)$.

Child(v, a): Sea $v_1 = v$ y v_2 el subintervalo del arreglo de sufijos donde los sufijos comienzan con a, dado por $LF(a, Root())$. Luego se busca en el intervalo v_1, el subintervalo w, donde $\Psi^m(w) \in v_2$, $m = SDepth(v_1)$.

Letter(v, i): Si $i = 1$ se retorna $charT[rank_1(D, v_l)]$, donde D es un bitmap que marca en el arreglo de sufijos el primer sufijo que comienza con cada letra distinta, $charT$ es un arreglo de tamaño σ, donde las letras que aparecen en $T_{1,n}$ están listadas en orden alfabético. Si $i > 1$ se retorna $Letter(\Psi^{i-1}(v_l), 1)$

3.17.3. Árbol de sufijos de Fischer et al.

Fischer et al. [FMN09] mostraron que la información de la topología no es necesaria, ya que se pueden simular los nodos como intervalos del arreglo de sufijos, y se navega el árbol usando tres operaciones que se ejecutan sobre el arreglo LCP: *next smaller value (NSV), previous smaller value (PSV) y range minimum query (RMQ)*. Estas operaciones son resueltas ingeniosamente en tiempos sub-logaritmicos usando sólo $o(n)$ extra *bits* sobre la representación del *LCP. A*demás probaron que el arreglo H de Sadakane es compresible. Finalmente, el árbol comprimido ocupa $nH_k \left(2\log(\frac{1}{H_k}) + \frac{1}{\epsilon} + O(1)\right) + o(n)$ bits para cualquier $k \leq \alpha \log_\sigma n$, con $0 < \alpha < 1$ constante.

Definición 3.22. [FMN09] Sea $S[1, n]$ una secuencia de elementos extraídos de un conjunto con un orden total. Se definen las siguientes tres operaciones (**NPR**):

$$\begin{aligned} NSV_S(i) &= \text{mín}\{j, (i < j \leq n \ \wedge \ S[j] \prec S[i]) \ \vee \ j = n+1 \\ PSV_S(i) &= \text{máx}\{j, (1 \leq j \leq i \ \wedge \ S[j] \prec S[i]) \ \vee \ j = 0 \\ RMQ_s(i, j) &= \text{argmin}_{i \leq l \leq j} S[l] \end{aligned}$$

donde *argmin* se refiere al orden $\prec$.

Es decir $NSV_{LCP}(i)$ encuentra la primera posición $x > i$ (más a la izquierda) del arreglo LCP tal que $LCP[x] < LCP[i]$. $PSV_{LCP}(i)$ encuentra la última posición $x < i$ del arreglo tal que $LCP[x] < LCP[i]$. $RMQ_s(i, j)$ encuentra la primera posición con el mínimo valor del LCP en el rango $[i, j]$.

Al igual que el árbol de Russo, los nodos de esta representación están basados en los intervalos del arreglo de sufijos, por lo que un nodo v esta asociado al intervalo $[v_l, v_r]$ del arreglo. Por lo tanto algunas operaciones están implementadas de la misma forma que en el árbol de Russo. A continuación se explica cómo están implementadas las operaciones descritas en la tabla 2.1 en base *a RMQ*, *NSV* y *PSV*:

Root/Count/Ancestor: La raíz es representada por el intervalo $[1, n]$, $Count(v)$ es $v_r - v_l + 1$ y $Ancestor(v, w)$ es verdad sii $w_l \leq v_l \leq v_r \leq w_r$.

isleaf(v): Verdad si $v_l = v_r$, falso de lo contrario.

Locate(v): Si v es una hoja ($v_l = v_r$) retorna $A[v_l]$, de lo contrario -1.

SDepth(v): Retorna $LCP[k]$, donde $k = RMQ(v_l + 1, v_r)$.

Parent(v): Si v es la raíz, retorna -1. En caso contrario $k \leftarrow v_l$ si $LCP[v_l] > LCP[v_r + 1]$, si no $k \leftarrow v_r + 1$. Luego el intervalo que representa al padre de v es $[PSV(k), NSV(k) - 1]$.

FChild(v): Si v es una hoja, retorna -1. En caso contrario el primer hijo de v es el intervalo $[v_l, RMQ(v_l + 1, v_r) - 1]$.

NSibling(v): Primero se va al nodo padre $w = Parent(v)$. Si $v_r = w_r$ retornar -1 dado que v no tiene siguientes hermanos. Si $v_r + 1 = w_r$ el nodo hermano es una hoja, por lo que se retorna $[w_r, w_r]$. En caso contrario retornar $[v_r + 1, RMQ(v_r + 2, w_r) - 1]$.

SLink(v): Si v es la raíz retorna -1. De lo contrario se encuentran los *suffix links* de las hojas v_l y v_r, $x = \Psi(v_l)$ e $y = \Psi(v_r)$. Luego se calcula $k = RMQ(x + 1, y)$ para retornar $[PSV(k), NSV(k) - 1]$.

Slink^i(v): Igual que el método anterior salvo que $x = \Psi^i(v_l)$ e $y = \Psi^i(v_r)$.

LCA(v, w): Si alguno de los dos nodos es ancestro del otro responder ese ancestro. De lo contrario se calcula $k = RMQ(v_r + 1, w_l)$ para responder $[PSV(k), NSV(k) - 1]$.

Child(v, a): Si v es hoja retorna -1. De lo contrario con búsqueda binaria sobre el intervalo $p \in [v_l+1, v_r]$ usando la función RMQ se encuentra la posición donde $LCP[p] = \min_{i \in [v_l+1, v_r]} LCP[i]$, y $T[A[p] + LCP[p]] = Letter([p, p], LCP[p] + 1) = a$. Se retorna $[p, RMQ(p + 1, v_r) - 1]$ o -1 si no se encontró p.

Cánovas y Navarro [CN10] comparan distintas implementaciones de arreglos LCP, concluyendo que las estructuras basadas en el bitmap H son las que mejor presentan un mejor espacio. A pesar de no ser las que presentan el mejor tiempo de rendimiento, este es razonable.

3.18. NPR: next smaller value, previous smaller value y range minimum query

3.18.1. NPR de Fischer et al. (NPR-FMN)

Dado una secuencia de números de largo n, todas estas operaciones *NSV, PSV* y *RMQ* (definidas en 3.22) se pueden resolver fácilmente en tiempo constante usando $O(n)$ bits extra de espacio sobre el arreglo

LCP, pero Fischer et al. [FMN09] proponen un algoritmo de tiempo sub-logarítmico con sólo $o(n)$ bits extra.

Para responder NSV y de manera simétrica PSV sobre un arreglo de enteros $S[1,n]$, se divide S en bloques consecutivos de b valores cada uno. Una posición i es llamada *near* si está en el mismo bloque que $NSV(i)$, luego el primer paso para resolver NSV es escanear el bloque que contiene a i, escaneando los valores $S[i+1, b \cdot \lceil i/b \rceil]$ en busca del primer valor $S[j] < S[i]$. Por lo tanto para posiciones *near* la consulta toma tiempo $O(b)$.

En caso que estén en distintos bloque se le llama *far.* Una posición *far* i sera *pioneer* si $NSV(i)$ no está en el mismo bloque que $NSV(j)$, siendo j la posición *far* más a la derecha anterior a i. Sigue que si i no es un *pioneer* y j es el último *pioneer* anterior a i, luego $NSV(i)$ está en el mismo bloque que $NSV(j) \geq NSV(i)$. Por lo tanto para resolver $NSV(i)$, se encuentra j y luego se escanea el bloque $S[\lceil SNV(j)/b \rceil - b+1, NSV(j)]$ para encontrar el primer valor tal que $S[j'] < S[i]$ en tiempo $O(b)$.

Por lo tanto el problema se reduce a encontrar eficientemente los *pioneer* que preceden a cada posición i. Para eso se marcó en un *bitmap* $P[1,n]$ las posiciones de los *pioneers* usando $O(\frac{n \log b}{b}) + O(\frac{n \log \log n}{\log n})$ *bits* de espacio, donde $j = select(P, rank(P, i))$ es el *pioneer* que precede a la posición i.

La proposición teórica de Fischer et al.[FMN09] para responder las consultas RMQ en tiempo sub-lineal asume que los accesos a S son en tiempo constante, cuando en la práctica si $S = LCP$ representado con H los accesos son muy costosos por lo que se necesita un método que use la mínima cantidad de accesos al arreglo LCP.

3.18.2. NPR de Cánovas y Navarro (NPR-CN)

La novedosa estructura *NPR-CN,* propuesta por Cánovas y Navarro [CN10], permite responder las operaciones NPR de manera eficiente, basada en una técnica similar al *range min-max tree* ocupado sobre la estructura BP [SN10]. Esta logra una mejora tanto en espacio y tiempo sobre otras representaciones de NPR. Además la propuesta se comparó con el árbol de sufijos que guarda la topología explícitamente, obteniéndose que el primero logra un menor espacio, mientras que el segundo tiene un mejor tiempo de rendimiento, sin embargo para varias operaciones, el basado en NPR responde mejor que el otro.

La estructura que llamaremos *NPR-CN,* consiste en dividir el arreglo LCP en bloques de largo fijo L, y guardar el mínimo valor de cada bloque. De la misma forma se construye un nuevo arreglo de bloques sobre lo anterior guardando el mínimo de los L bloques inferiores, y así sucesivamente, formando un árbol L-ario perfectamente balanceado salvo por los bordes. Dada la construcción, el espacio que ocupa solo depende del parámetro L y no del texto, ocupando $\frac{n}{L} \log n(1 + O(1/L)$ bits, luego si se elige $L = \omega(\log n)$, ω constante, el espacio usado es $o(n)$ bits.

Para responder $NSV(i)$, se busca el primer $j > i$ tal que $LCP[j] < p = LCP[i]$ que usando el árbol toma $O(L \log(n/L))$. Primero se busca secuencialmente en el bloque que contiene a i, si no esta ahí se sube un nivel y se verifica tanto el padre como el nodo a la derecha del padre. Si el mínimo de alguno de estos nodos es menor a p, se baja al bloque correspondiente para buscar secuencialmente el primer valor menor a p. Si los nodos visitados no contienen un valor menor a p se sube un nivel y se visita tanto el padre como el hermano de la derecha, así hasta que en algún nodo se encontrara un mínimo menor a p, luego se comienza a recorrer el árbol hacia abajo encontrando en cada nivel el primer hijo que contiene un nodo con un mínimo menor a p hasta llegar al bloque correspondiente para escanear el arreglo LCP. Notar que se necesitan $O(L)$ accesos al LCP que en la práctica es la parte más pesada del proceso dado que los mínimos del árbol están guardados explícitamente. Para responder las consultas PSV, se procede de manera simétrica a NSV pero

recorriendo de derecha a izquierda.

Para calcular $RMQ(x, y)$ se calcula secuencialmente sobre el LCP el mínimo en el intervalo $\left[x,\ L\left\lceil\frac{x}{L}\right\rceil - 1\right]$ y en el intervalo $\left[L\left\lfloor\frac{y}{L}\right\rfloor,\ y\right]$, que corresponde a los bloques que contienen x e y respectivamente. Luego se calcula $RMQ(L\left\lceil\frac{x}{L}\right\rceil, L\left\lfloor\frac{y}{L}\right\rfloor - 1)$ usando la estructura visitando el nodo que representa ese rango, que además contiene la posición en el arreglo LCP donde esta el mínimo ocupando un espacio extra de sólo $\frac{n}{L}\log L(1 + O(1/L))$ bits. Finalmente se comparan los tres resultados y se responde con el menor de todos, en caso de empate el de más a la izquierda. Por lo tanto el costo de la consulta viene mayoritariamente del los $O(L)$ accesos al LCP ya solo requiere un acceso a la estructura *NPR-CN*.

Capítulo 4

Nuestro Árbol de Sufijos Comprimido para Secuencias Repetitivas

Lo que se busca es encontrar un árbol de sufijos comprimido que explote las repeticiones existentes. Este se basa en el árbol de Fischer et al. [FMN09] e implementado por Cánovas [CN10]. El árbol está representado por tres estructuras comprimidas: el arreglo de sufijos comprimido, el arreglo comprimido *Longest Common Prefix y* la estructura *NPR*. Lo que se busca es adecuar las estructuras para colecciones repetitivas con el fin de obtener una mejor compresión y tiempos de respuesta.

4.19. Arreglo de sufijos comprimido con run-length (RLCSA)

Mäkinen y Navarro [MN05] mostraron que Ψ se descompone en a lo sumo $R < nH_k + \sigma^k$ *runs,* para todo k, donde un *run* es una secuencia de números iguales. Sirén et al. [SVMN08] mostraron que un texto repetitivo genera *runs* largos, en particular mostraron que la variante del arreglo de sufijos comprimido basado en codificaciones de *run-length* (RLCSA [Sir09, SVMN08]) aplicado a textos repetitivos muestra un gran rendimiento tanto en espacio como en tiempo comparado con otros variantes de arreglos de sufijos comprimidos, así como también tiempos razonables.

Es importante resaltar que **en textos altamente repetitivos la cantidad de *runs* R es mucho menor que** n, por lo que el espacio ocupado decrece para estos textos.

Más precisamente el RLCSA comprime Ψ (sección 2.11) codificando los *runs* que se generan por las diferencias $\Psi(i)-\Psi(i-1)$. La codificación transforma un *run* $\Psi(i)\Psi(i+1)...\Psi(i+l)$ en $\Psi(i+1)-\Psi(i)$ seguido por el largo $l+1$ del *run*. Además es necesario un muestreo de los valores absolutos $\Psi(i)$ para soportar un acceso rápido al arreglo. Sea N el tamaño del texto, σ el tamaño del alfabeto y R el número de *runs.* La suma de todas las diferencias $\Psi(i) - \Psi(i-1)$ es a lo más σN y el largo total de los *runs* de 1's es $N - R$. Por lo tanto usando códigos δ (sección 2.7) para codificar los números, el tamaño del RLCSA en *bits* es [SVMN08]:

$$R\left(\log\frac{\sigma N}{R} + \log\frac{N}{R} + O\left(\log\log\frac{\sigma N}{R}\right)\right)\left(1 + \frac{O(\log N)}{B}\right) + O(\sigma \log N),$$

donde B es el paso de muestreo en bits.

Para obtener $\Psi(i)$, se hace búsqueda binaria en el muestreo y luego se suman las diferencias hasta llegar a la posición i, tomando un tiempo de $O(\log\frac{|\Psi|}{B} + B)$. Luego por ejemplo para encontrar el número de ocurrencias de un patrón P en el texto ($count(P)$), usando el algoritmo de *backward search* (sección 2.12) que es proporcional a $|P|$, tomaría tiempo $O\left(|P|\left(\log\frac{|\Psi|}{B} + B\right)\right)$.

En esta memoria se usara la implementación de Jouni Sirén optimizada para colecciones de textos altamente repetitivos, disponible en `http://www.cs.helsinki.fi/group/suds/rlcsa/`. Esta implementación muestra una gran eficiencia de espacio [SVMN08] en comparación con alternativas que logran una compresión de orden superior relativa a Ψ, pero pagando por una pérdida de eficiencia en el tiempo de acceso.

4.19.1. Resultados Experimentales

Todos los resultados experimentales que se muestran en esta sección y en los siguientes capítulos fueron realizados en un computador de 2 procesadores Intel Core2 Duo, cada uno de 3 GHz, 6 MB cache y 8 GB RAM. Los textos a indexar son obtenidos del corpus de textos repetitivos del sitio Pizza&Chili (`http://pizzachili.dcc.uchile.cl/repcorpus.html`), dentro de los cuales, los experimentos se

Nombre	Tamaño	Re-Pair	Descripción
Para	410 MB	2,80 %	ADN de *Saccharomyces Paradoxus* obtenidas del *Saccharomyces Genome Resequencing Project*.
Influenza	148 MB	3,31 %	Secuencias de Haemophilus Influenzae obtenidas del *National Center for Biotechnology Information* (NCBI).
Escherichia Coli	108 MB	9,63 %	ADN de una bacteria de la especie *Escherichia Coli* obtenidos del NCBI.
Einstein (de)	89 MB	0,16 %	Todas las versiones de los artículos sobre *Albert Einstein* en aleman obtenidos de *Wikipedia*.
DNA	50 MB		*Secuencias de ADN obtenidas del Gutenberg Proyect.*

Tab. 4.5: Textos usados para realizar todos los experimentos.

Nombre	Tamaño	Bits/c
PARA	45 MB	0,84
Influenza	19 MB	0,96
Escherichia Coli	35 MB	2,46
Einstein (de)	2 MB	0,17
DNA	39 MB	5,91

Tab. 4.6: Arreglo de Sufijos RLCSA.

realizaron sobre cinco textos. Además se incluyo una colección de ADN normal, no repetitivo para saber el comportamiento límite de nuestro método. La descripción de los textos se muestra en la tabla 4.5 y los tamaños y compresión de los arreglos de sufijos obtenidos con *RLCSA* sobre los textos en la tabla 4.6.

4.20. Representación del Arreglo *Longest Common Prefix* (LCP)

Fischer et al. [FMN09] describen una representación del *bitmap* H de Sadakane (ver sección 3.17.1), de una forma más eficiente en espacio. Esta se basa en la observación de que el número de *runs* de 1-bit en H está acotado por los R *runs* en Ψ, y por ende los *runs* de 0's en H también están acotados por R. De tal manera, demostraron que H se puede comprimir en $O(R\log(n/R))$ bits manteniendo el acceso a tiempo constante.

Análogamente al RLCSA, Fischer et al. [FMN09] representan H basado en codificaciones de *run-length*, y codificando los *runs* maximales tanto de 0's como de 1's en H, codificando separadamente los largos de los *runs* de 1's, $O = o_1, o_2, \ldots$ y los largos de los *runs* de 0's $Z = z_1, z_2, \ldots$. A partir de esa codificación se puede calcular $select_1(H, j)$ (operación necesaria para computar LCP, sección 3.17.1), encontrando el mayor r tal que $\sum_{i=1}^{r} o_i < j$, dado que $select_1(H, j) = j + \sum_{i=1}^{r} z_i$.

A continuación se formaliza la relación de los *runs* en H y Ψ, para concluir con el espacio necesario para la representación de H.

Definición 4.23. Un *run* en Ψ dado el texto T, se dice que es una secuencia maximal si es una secuencia consecutiva de valores i, donde $\Psi(i) - \Psi(i-1) = 1$ y $T_{SA[i]} = T_{SA[i-1]}$, salvo el primer i, donde lo anterior no se cumple.

Llamemos a la posición i un *stopper* si $i = 1$ o $\Psi(i) - \Psi(i-1) \neq 1$ o $T[SA[i-1]] \neq T[SA[i]]$. Luego Ψ tiene exactamente R *stoppers* por la definición de *runs en* Ψ. Definimos una *cadena* en Ψ como una secuencia maximal $i, \Psi(i), \Psi(\Psi(i)), \ldots$ tal que cada $\Psi^j(i)$ no sea un *stopper* excepto el último. Como Ψ es una permutación con un solo ciclo, se sigue que en el camino de $\Psi^j(SA^{-1}[1]), 0 \leq j \leq n$, (correspondiente a recorrer las posiciones del SA iniciando desde $SA^{-1}[1]$) se encuentran los R *stoppers* y por lo tanto se tienen también R *cadenas* en Ψ.

Lema 4.24. *[FMN09]* H tiene a lo más R *runs* de 1's (donde incluso un solo 1 cuenta como un *run*).

Ahora mostraremos que cada *cadena* en Ψ induce un *run* de 1's del mismo largo que la *cadena* en H, así quedando demostrado el lema. Sea $i, \Psi(i), ..., \Psi^l(i)$ una *cadena*, luego $\Psi^j(i) - \Psi^j(i-1) = 1$ para $0 \leq j < l$. Sea $x = SA[i-1]$ e $y = SA[i]$, luego $SA[\Psi^j(i-1)] = x + j$ y $SA[\Psi^j(i)] = y + j$. Además por definición $LCP[i] = |lcp(T_{SA[i-1],n}, T_{SA[i],n})| = |lcp(T_{x,n}, T_{y,n})|$. Notar que $T[x+LCP[i]] \neq T[y+LCP[i]]$, por lo tanto $SA^{-1}[y+LCP[i]] = \Psi^{LCP[i]}(i)$ es un *stopper*, luego $l \leq LCP[i]$. Incluso, $LCP[\Psi^j(i)] = |lcp(T_{x+j,n}, T_{y+j,n})| = LCP[i]-j \geq 0$ para $0 \leq j < l$. Consideremos ahora, $s_{y+j} = y+j = LCP[SA^{-1}[x+j]] = y+j+LCP[\Psi^j(i)] = y + j + LCP[i] - j = y + LCP[i]$, para $0 \leq j < l$. Esto produce $l-1$ diferencias $s_{y+j} - s_{y+j-1} = 0$ para $0 < j < l$, que es equivalente a un *run* de l 1-bits en H. Al recorrer todas las *cadenas* del ciclo de Ψ se recorre la secuencia completa a comprimir, de izquierda a derecha, produciendo a lo más R *runs* de 1's.

Dado que ambos O y Z tienen a lo más R 1's, pueden ser representados usando $2R \log \frac{n}{R} + O\left(R + \frac{n \log\log n}{\log n}\right)$ *bits* [RRR02].

Teorema 4.25. *[FMN09] El arreglo LCP de un texto de largo n cuya función Ψ tiene R runs, puede ser representado usando $2R \log \frac{n}{R} + O\left(R + \frac{n \log\log n}{\log n}\right) = nH_k\left(2 \log \frac{1}{H_k} + O(1))\right) + O\left(\frac{n \log\log n}{\log n}\right)$ bits, para cualquier $k \leq \alpha \log_\alpha n$ y cualquier constante $0 < \alpha < 1$. El Acceso al $LCP[i]$ toma tiempo constante dado el valor de $SA[i]$.*

En esta memoria se usa la implementación de Cánovas, que experimentalmente muestra la mayor robustez al representar H [CN10], cuya representación tiene dos alternativas de codificar los *bitmaps* O y Z (FMN-RRR y FMN-OS). El trabajo propone una nueva alternativa, codificando O y Z usando códigos delta (FMN-δ), lo cual en textos repetitivos muestra una mayor compresión.

FMN-RRR: Propuesto por Fischer et al. [FMN09], representa H mediante las codificaciones de los *bitmaps* O y Z. Se usa la codificación de Raman, Raman y Rao [RRR02] (recordar sección 2.8.2) implementada por Claude [CN08]. Ocupa $0{,}54n$ *bits* sobre la entropía de ambos *bitmaps,* $2R \log \frac{n}{R}$.

FMN-OS: Como el anterior, pero para codificar los *bitmaps* O y Z se ocupa la técnica de Okanohara y Sadakane [OS07] para arreglos poco densos (recordar sección 2.8.3). Esto requiere $2R \log \frac{n}{R} + O(R)$ *bits.*

FMN-DELTA: Igual que los anteriores, pero para codificar los *bitmaps* O y Z se ocupa la codificación delta (sección 2.8.4), técnica práctica para arreglos poco densos implementada por Kreft [KN10]. Requiere en el peor caso $2R \log \frac{n}{R} + O(R \log\log \frac{n}{R}) + o(\frac{n}{s})$, donde s es el parámetro de muestreo.

4.20.1. Resultados Experimentales

Para comparar las estructuras que representan el LCP, se midió el tiempo promedio de acceder a una posición al azar del arreglo LCP y su desviación estándar. Para eso se accedieron 10.000.000 de posiciones al azar. La figura 4.11 muestra los resultados de tiempo/espacio obtenidos.

Como se puede apreciar en los resultados, la representación *FMN-DELTA* es la que ocupa un menor espacio seguida por *FMN-OS.* Esto se debe a que en textos altamente repetitivos hay menos *runs* y más largos ($R \ll n$) y la cantidad de unos en los arreglos O y Z está acotada por R. Por lo tanto, dado que ambas estructuras ocupan técnicas de codificación para arreglos poco densos, es que ocupan un menor espacio a medida que la cantidad m de unos es menor (recordar sección 2.8.3). Mientras que *FMN-RRR* ocupa $R \log \frac{n}{R} + O(R) + o(n)$ *bits,* es el último termino el que aporta demasiado espacio haciéndolo impracticable para secuencias repetitivas. Notar que en el texto *DNA* siendo el menos repetitivo *FMN-RRR* presenta la mejor compresión.

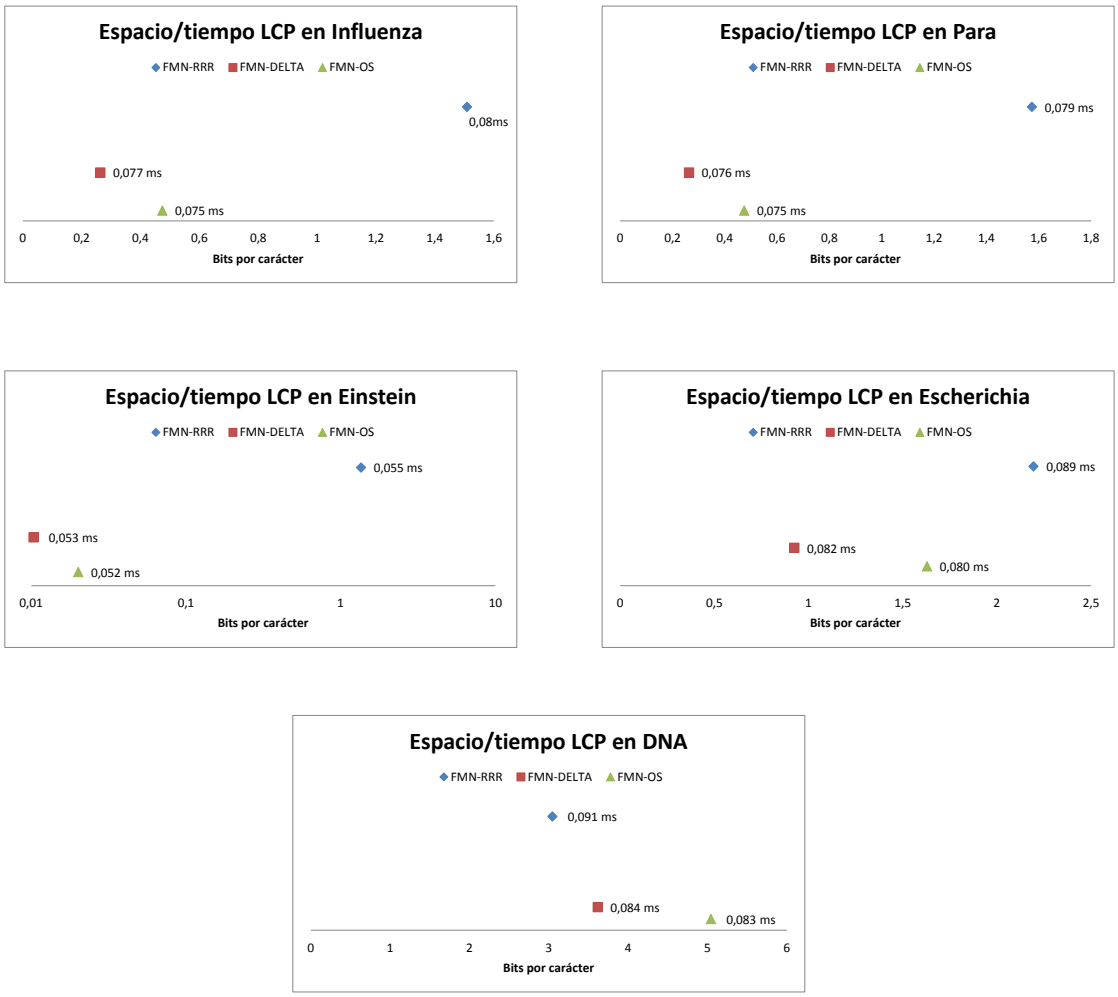

Fig. 4.11: Resultados de tiempo/espacio al acceder a posiciones al azar del arreglo LCP. Notar el eje x logarítmico en Einstein.

Notar que las estructuras sólo difieren entre sí, por la forma en como se codifican O y Z, ya que la descompresión de H se mantiene invariante para las tres estructuras una vez decodificados los valores correspondientes en O y Z. Luego las diferencias de los tiempos de acceso al LCP sólo dependen de la decodificación de los arreglos O y Z. Análogamente al espacio, las codificaciones usadas en las estructuras *FMN-OS* y *FMN-DELTA* se comportan mejor para arreglos poco densos, tomando un tiempo $O(\log \frac{n}{m} + \frac{\log^4 m}{\log n})$ y $O(s + \frac{\log m}{s})$ respectivamente, donde s es el salto de posiciones para el muestreo ($s = 32$ en el experimento). Mientras, la codificación usada en *FMN-RRR* toma un tiempo constante, que en los resultados muestran que en la práctica resulta ser mayor que las otras alternativas sobre textos repetitivos.

FMN-DELTA presenta la mejor compresión, usando en promedio menos a 0,25 bits por cada elemento del arreglo LCP ocupando 20 % a 45 % menos espacio que la estructura *FMN-OS.* Si bien los resultados muestran que *FMN-OS* tiene un tiempo de acceso menor, en la práctica esta diferencia es insignificante y no difiere en más de un 5 % por sobre *FMN-DELTA.*

Por lo tanto, para el resto del trabajo se ocupa la estructura *FMN-DELTA* como representación del LCP, que en la práctica muestra el mejor rendimiento, requiriendo el menor espacio y la mejor relación

espacio/tiempo.

4.21. Una solución novedosa para NPR basada en Re-Pair

La solución que se propone en esta memoria está inspirada en la representación de la estructura *NPR-CN* (sección 3.18.2) propuesta por Cánovas y Navarro [CN10] y en la técnica de compresión *Re-Pair* [LM00] (sección 2.15). Recordar que la estructura *NPR-CN* no depende de la distribución del arreglo sino que del parámetro L elegido, construyendo un árbol balanceado sobre el LCP y ocupando $\frac{n}{L}\log n(1+O(1/L))$ *bits*. En cambio el árbol generado al comprimir por *Re-Pair* sí aprovecha las repeticiones que contiene el arreglo. Se propone construir un árbol sobre el LCP basado en el de la estructura *NPR-CN,* pero cuya forma esté dada por el método *Re-Pair,* así aprovechándose de las repeticiones y logrando un menor espacio y tiempo de respuesta. A esta estructura la llamaremos *NPR-RP.*

González y Navarro [GN07] usan *Re-Pair* para comprimir el arreglo de sufijos diferencial $A'[i] = A[i] - A[i-1]$ ya que los *runs* en Ψ se convierten en repeticiones en A'. Si tomamos un *run* de Ψ de largo l, $\Psi(i+s) = j+s$ para $0 \leq s \leq l$, este induce una seudo-copia en A de largo l de la forma $A[j+s] = A[i+s]+1$ para $0 \leq s \leq l$, y luego en el arreglo diferencial se cumple que $A'[j+s] = A'[i+s]$ para $1 \leq s \leq l$. Lo que muestra que en A' hay a lo más $2R$ pares distintos y por ende el par más frecuente aparece al menos $\frac{n}{2R}$. Luego sigue que debido a superposiciones una primera iteración reemplaza al menos βn pares con $\beta = \frac{1}{4N}$, quedando A' con a lo más $n - \beta n$ símbolos. Notar que la cantidad de *runs* no disminuye ya que un reemplazo no puede separar un *run,* por lo tanto en la segunda iteración, usando el mismo argumento se remueven al menos $\beta(n - \beta n) = \beta n(1-\beta)$ símbolos y así sucesivamente que por inducción fácilmente da que en la iteración i se remueven $\beta n(1-\beta)^{i-1}$ símbolos. Por último, después de M iteraciones se han removido $\sum_{i=1}^{M} \beta n(1-\beta)^{i-1} = n - n(1-\beta)^M$ y por ende el largo de C es $n(1-\beta)^M$ y un diccionario de $2M$ reglas. Optimizando M se obtiene que *Re-Pair* logra una compresión de $O(R\log\frac{n}{R}\log n)$ *bits* [GN07, NM07].

Lema 4.26. *[FMN09] Definimos similarmente el arreglo* LCP *diferencial como* $LCP'[1] = LCP[1]$ *y* $LCP'[i] = LCP[i] - LCP[i-1]$ *si* $i > 1$. LCP' *contiene las mismas repeticiones que el arreglo de sufijos diferencial.*

Dado un *run* en Ψ, $\Psi(i+s) = j+s$ para $0 \leq s \leq l$, se tiene que $LCP[\Psi(j+s)] = LCP[j+s] - 1$, luego la codificación diferencial LCP' satisface $LCP'[i+s] = LCP'[j+s]$ para $0 < s \leq l$.

Luego *Re-Pair* debiese comportarse de la misma forma que al comprimir el arreglo A', por ende ocupando en el peor caso $O\left(R\log\frac{n}{R}\log n\right)$ bits.

Aprovechando las grandes repeticiones que presenta el LCP diferencial, dado que se está trabajando sobre textos altamente repetitivos, la estructura del árbol que se crea sobre el LCP se basa en la representación comprimida del arreglo *LCP'* usando la técnica *Re-Pair*. El arreglo comprimido por *Re-Pair* se representa como una secuencia de reglas y símbolos $C[1,m]$ y varias reglas, donde cada regla es representada por un árbol binario cuyas hojas son una subsecuencia contigua de *LCP'*. De esa manera el arreglo original *LCP'* queda representado por la concatenación de varios árboles binarios, donde cada sub-árbol cubre una subsecuencia de *LCP'*. Notar que los árboles no necesariamente son balanceados, ya que dependen de la distribución de las repeticiones, y que no son disjuntos (ver figura 2.7).

Para soportar eficientemente las consultas NPR, en cada nodo i del árbol se guarda el valor mínimo de la secuencia generada por el sub-árbol (min_i), la suma de la secuencia (sum_i) y la cantidad de elementos en la secuencia ($cover_i$). Además, con fin de acelerar las consultas RMQ, que son las más costosas, se agrega

en cada nodo la posición ($left\_min\_pos_i$) del valor min; en caso de empates la posición de más a la izquierda y simétricamente $right\_min\_pos_i$ la posición del mínimo más a la derecha.

El mínimo de cada nodo min es sobre la suma parcial de la secuencia representada de LCP' . Para las operaciones sera necesario conocer el valor absoluto del LCP correspondiente a la primera posición (más a la izquierda) que representa la regla.

Proposición 4.27. *Dada la regla i que cubre el intervalo $LCP'[x, x + cover_i]$ y siendo $lcp = LCP[x]$, se cumple que:*

1. $lcp + min_i$ es el valor mínimo de la secuencia $LCP[x, x + cover_i]$,
2. $x + lcp\_min\_pos$ es la primera posición de la secuencia $LCP[x, x + cover_i]$ que contiene el mínimo min_i,
3. $LCP[cover_i + 1] = lcp + sum_i$,
4. La regla i', nodo hermano a la derecha de i, cubre la secuencia $LCP[cover_i + 1, cover_i + cover_{i'} + 1]$, y análogamente para la izquierda.

Notar que al recorrer C, si $i = C[j]$, $LCP[cover_{C[j]} + 1]$ corresponde al valor del LCP correspondiente a la primera posición de la regla $C[j + 1]$. De esa manera es posible iterar sobre el arreglo C buscando el mínimo por bloques heterogéneos de largos $cover_{C[j]}$ con un solo acceso al arreglo LCP.

Hasta ahora, cada regla contiene los valores del LCP diferencial guardadas en sus hojas. Esto ocupa demasiado espacio, además de ser información replicada ya que tenemos el arreglo LCP comprimido. Para ahorrar espacio, sobre la estructura *NPR-RP* se fija un parámetro T para podar todos los subárboles que no representan una secuencia de largo al menos T, obteniendo así una mayor compresión a medida que aumenta T, a cambio de tener que acceder a $O(T)$ elementos del LCP para resolver consultas. Notar que para cualquier $T > 0$ se podan al menos todas las hojas de las reglas, y para $T = 0$ se tiene el arreglo LCP' completo. En la figura 4.12 se muestra un ejemplo simplificado de la estructura a la cual se le podan las hojas.

Dado que las reglas representan secuencias de largo variable, se guarda un muestreo de C, guardando cuál es la posición x del arreglo LCP que representa el primer valor de la secuencia generada por la regla $C[j]$, es decir, se guarda la posición x tal que $C[j]$ cubre la secuencia $LCP[x, x + cover_{C[j]}]$. La construcción del muestreo toma tiempo $O(|C|)$, recorriendo secuencialmente C para calcular $\sum_{i=1}^{j} cover_{C[i]}$, $j = r, 2r, \ldots, \lfloor \frac{n}{r} \rfloor r$ e ir guardando las sumas parciales cada r pasos. El muestreo ocupa $\frac{|C| \log n}{r}$ *bits* ya que se guarda explícitamente en un arreglo de largo $\frac{|C|}{r}$. Esto permite, mediante búsqueda binaria, encontrar qué regla $C[j']$ contiene la posición i' del LCP en tiempo $O(\ln n/r + r)$.

4.21.1. Consultas *NSV, PSV y RMQ*

Para responder $NSV(i)$, se busca la regla $C[j]$ que contiene a i mediante búsqueda binaria sobre C. $C[j]$ puede no estar al principio de un muestreo, de ser así, se recorre desde el último hasta $C[j]$ sumando sum_i. Como $C[j]$ es la raíz de un árbol, tiene almacenado el mínimo de éste. Se busca secuencialmente la regla $C[k]$ $k \geqslant j$, que contenga un valor menor a $LCP[i]$, es decir que se cumpla: $v + \sum_{u=j}^{k-1} sum_u + min_k < LCP[i]$, dónde v es el valor LCP correspondiente a la primera posición que representa la regla $C[j]$, tal que el mínimo de la regla sea menor que $LCP[i]$. Encontrada la regla, hay que recorrer el árbol recursivamente de la siguiente manera para encontrar el mínimo (notar que necesariamente el mínimo se encuentra en ese árbol salvo que esa regla contenga la posición i):

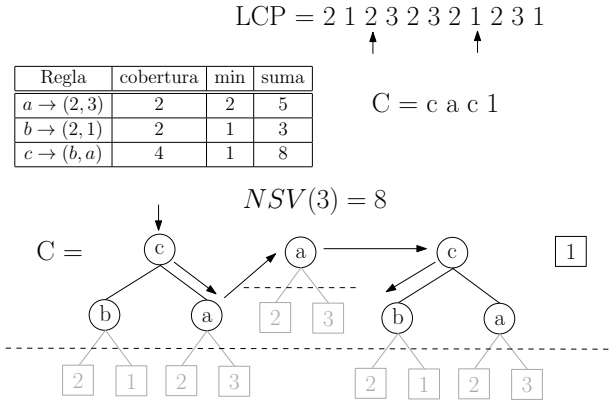

Fig. 4.12: Ejemplo simplificado de estructura NPR-RP. Puede verse como un árbol podado de un arreglo de enteros, las reglas con cobertura menor o igual a 2 fueron recortadas. Dentro de cada recuadro el mínimo de la regla. Las flechas muestran la secuencia de reglas accesadas para resolver $NSV(4)$.

1. Se visita el hijo izquierdo comparando el mínimo *min_val* (se calcula como en la proposición 4.27, punto 1) de este nodo con el valor $LCP[i]$.

2. Si $min\_val < LCP[i]$ quiere decir que ese sub-árbol tiene la solución por lo tanto prosigue en el paso 1.

3. Si $min\_val \geq LCP[i]$, ese sub-árbol no contiene la solución, por lo que se visita el nodo hermano, y se prosigue como en el paso 1, pero sumando *sum* de hijo izquierdo.

4. Si se llega a una hoja, quiere decir que el mínimo está en el intervalo $LCP[x, y]$ que por construcción es de largo menor a T. Dado que se puede calcular con *min_pos* la posición dentro del intervalo donde está el mínimo, si $LCP[i] = LCP[x + min\_pos] + 1$ entonces $NSV(i) = x + left\_min\_pos$ (notar que no hubo necesidad de acceder al LCP). En caso contrario, dado que a la izquierda de la posición del mínimo puede haber un valor mayor a éste, pero menor a $LCP[i]$, hay que buscar secuencialmente en el intervalo (de izquierda a derecha) por el primer valor $LCP[j] < LCP[i]$. Esto requiere acceder al arreglo LCP en a lo más $left\_min\_ pos < T$ posiciones, dado que $x < j < x + left\_min\_pos$.

Hay que tener especial cuidado si se entra en la regla que contiene a i, ya que puede que el mínimo de la regla esté a la izquierda de i, caso en que no es solución válida. En tal caso se ocupa el mismo algoritmo, pero descartando visitar los nodo izquierdos cuyo intervalo esté a la izquierda de i. Notar que la parte más pesada en la práctica es el acceso al arreglo LCP del paso 4, que cuesta tiempo $O(T)$.

En la figura 4.12 se puede ver la estructura de un árbol podado de un arreglo de enteros donde se recortaron todas las reglas con una cobertura menor o igual a 2 ($T = 2$). Para simplificar la vista en el ejemplo, el arreglo representa los valores absolutos del LCP y no el diferencial (razón por la cual no es necesario ocupar la suma de las reglas), pero la idea del algoritmo se mantiene. Las flechas muestran las reglas que se leen para calcular $NSV(4)$, partiendo por la regla que contiene a $lcp[4] = 2$, cuyo mínimo es 2 por lo que no es necesario seguir buscando en profundidad esa regla. Se procede recorriendo C en busca de una regla cuyo mínimo sea menor que 2. Una vez encontrada, se busca en profundidad hasta llegar a una hoja. Por último se busca secuencialmente en el arreglo LCP el primer valor menor a 2. Notar que por construcción nunca se tienen que obtener más que $O(T)$ valores del arreglo LCP.

$PSV(i)$ es simétrico a $NSV(i)$, salvo que recorriendo el árbol de derecha a izquierda, de igual manera que con las búsquedas secuenciales. Notar que saber la posición del mínimo no contribuye en nada debido a que en el mismo intervalo puede repetirse incluso varias veces el valor del mínimo mientras que *min_pos* guarda el de más a la izquierda. Por eso se decidió agregar de manera análoga a *min_pos*, la posición relativo a la regla del mínimo más a la derecha que llamaremos *right_min_pos*. Esta información agrega un espacio extra marginal ya que puede guardarse como la diferencia $right\_min\_pos - left\_min\_pos$ que es una secuencia más compresible.

Para calcular $RMQ(x, y)$ con $x < y$, se encuentra la regla $C[i]$ que contiene a x y la $C[j]$ que contiene a y con búsquedas binarias sobre C, y se separa la búsqueda en tres partes:

1. Si $i < j$ se recorren secuencialmente todas las reglas entre $C[i + 1]$ y $C[j - 1]$, llevando la suma $lcp(k) = \sum_{u=i+1}^{k} sum_{C[u]}$, $k < j$ y en cada regla comparando el mínimo $lcp(k) + min_{C[k+1]}$, tomando tiempo $O(j - i)$, recordando la que tiene el valor mínimo que llamaremos m_m, y en caso de empates eligiendo la de más a la izquierda. La posición de m_m en el LCP se obtiene en tiempo constante ya que se obtiene con $left\_min\_pos$ como se mencionó en la consulta NSV sin tener que acceder al LCP.

2. Hay que tener especial cuidado en los casos bordes que corresponden a las reglas $C[i]$ y $C[j]$, ya que el mínimo que la regla representa puede estar a la izquierda o derecha respectivamente de las posiciones x e y, por lo que no serían resultado válidos. En el caso de $C[i]$ y $C[j]$, hay que entrar a recorrer esas reglas sólo si $min_{C[i]} \leq min$ ($min_{C[j]} < min$), en tal caso se busca el mínimo cuya posición sea mayor o igual a x (menor a y) y que esté contenida en el intervalo que representa $C[i]$ de la siguiente forma:

   a) Sea *min_val* el mínimo valor encontrado hasta el momento para la regla k, si $min_k \geq min\_val$, fin.

   b) Si es una hoja, recorrerla secuencialmente en busca del mínimo, fin.

   c) Si el hijo derecho está totalmente contenido en el rango ($[x, x + cover_k)$, actualizar *min_val* con el mínimo de la hoja derecha min_{k_r} y entrar recursivamente al hijo izquierdo.

   d) Si no, entrar recursivamente al hijo derecho.

   Para el caso de la regla $C[j]$ donde se busca la posición menor a y se hace lo mismo intercambiando derecho e izquierdo en el algoritmo. Llamaremos m_l al mínimo encontrado en $C[i]$ y m_r el mínimo en $C[j]$ dado x e y respectivamente.

3. Luego se retorna la posición del mínimo entre los valores m_l, m_m, m_r, y en caso de empate el de más a la izquierda.

Notar que sólo se accede al arreglo LCP al buscar el mínimo en los extremos.

4.21.2. Construcción del NPR

La implementación de la estructura se construyó sobre un compresor *Re-Pair* a tiempo lineal (sección 2.15), implementado por Gonzalo Navarro y disponible en `http://www.dcc.uchile.cl/gnavarro/software`. El compresor presenta dos alternativas, basadas en una pequeña variante en la implementación del heap, que se usa para almacenar las frecuencias de los pares de símbolos. Los pares de símbolos (reglas) con la misma frecuencia son almacenados en arreglos, la variante consiste en la opción de apilar o encolar

Nombre	$\|LCP\|$	$\|C\|$ ($\|C\|/\|LCP\|$)	Reglas R ($\|R\|/\|LCP\|$)	Max Cover ($max/\|LCP\|$)
PARA	429265759	24910167 (5,8 %)	11239424 (2,6 %)	11096 (0,0026 %)
Influenza	154808556	2196278 (1,4 %)	3916617 (2,5 %)	11584 (0,0074 %)
Escherichia Coli	112689516	18663082 (16,6 %)	5940074 (5,3 %)	2048 (0,0018 %)
Einstein (de)	92758442	113538 (0,12 %)	132733 (0,14 %)	384356 (0,41 %)
DNA	52428801	8976100 (17,1 %)	1095019 (2,1 %)	400 (0.00076 %)

Tab. 4.7: Compresión con técnica *Re-Pair* apilando.

Nombre	$\|LCP\|$	$\|C\|$ ($\|C\|/\|LCP\|$)	Reglas R ($\|R\|/\|LCP\|$)	Max Cover ($max/\|LCP\|$)
PARA	429265759	24937174 (5,8 %)	11227913 (2,6 %)	11096 (0,0026 %)
Influenza	154808556	2201424 (1,4 %)	3916540 (2,5 %)	11585 (0,0074 %)
Escherichia Coli	112689516	18672689 (16,6 %)	5935713 (5,3 %)	2048 (0,0018 %)
Einstein (de)	92758442	113750(0,12 %)	132672 (0,14 %)	201166 (0,21 %)
DNA	52428801	8977461 (17,1 %)	1095019 (2,1 %)	407 (0.00076 %)

Tab. 4.8: Compresión con técnica *Re-Pair* encolando.

las reglas en estos arreglos. Experimentalmente la segunda opción hace que sea muy poco probable que se generen reglas muy profundas, generandose reglas más balanceadas. Analizaremos ambas alternativas, *NPR-RP* es la estructura generada por la implementación de *repair* que apila, y llamaremos *NPR-RPBal* a la que encola. En las tablas 4.7 y 4.8 se ven los resultados de comprimir el LCP diferencial de los textos con ambas alternativas de *Re-Pair* , a pesar de ser parecidos, se mantiene que la primera alternativa genera secuencias C más cortas, pero tienen mayor cantidad de reglas. Esto nos da una idea de cuáles son los textos más repetitivos, destacando *Einstein* con una gran compresión de C y pocas reglas, mientras que el texto *DNA* es el menos comprimible. Notar que a pesar de que los resultados de *DNA* y *Escherichia* son parecidos, en *DNA* la máxima cobertura de una regla es muy baja lo cual es una mala propiedad para la propuesta ya que significa que cada regla en C representa pocos elementos del LCP' y además deja poco espacio para podar el árbol como se menciona más adelante.

A este algoritmo de compresión se le agregó, en tiempo de construcción y sin aumentar los recorridos por el texto, la información extra por nodo (min_i, $cover_i$, sum_i,min_pos_i). El árbol binario y en particular las reglas se construyen de abajo hacia arriba, es decir de las hojas a la raíz, dado que cada sub-árbol es una regla, y se parte por los símbolos originales del arreglo. Luego cada nueva regla que se crea tiene como hijo izquierdo una regla existente o un valor del arreglo LCP', y lo mismo para el hijo derecho. Esto permite que la obtención de min_i, $cover_i$, sum_i y min_pos_i para una regla i sólo dependa de los valores de sus hijos (i_l, i_r), tomando tiempo $O(1)$ en la construcción del árbol Re-Pair por cada regla nueva que se crea. Para obtener los valores, recordando que se está trabajando sobre el LCP diferencial, se hacen los siguientes cálculos:

$$\begin{aligned}
min_i &= \text{mín}\,(min_{i_l}, sum_{i_l} + min_{i_r}) \\
cover_i &= cover_{i_l} + cover_{i_r} \\
sum_i &= sum_{i_l} + sum_{i_r} \\
min_pos_i &= min_pos_{i_l} \text{ si } min_i = min_{i_l} \\
min_pos_i &= sum_{i_l} + min_pos_{i_r} \text{ si } min_i = sum_{i_l} + min_{i_r}
\end{aligned}$$

Finalmente, las reglas son representadas como una matriz de 6 filas, para guardar los valores min_i, $cover_i$, sum_i, min_pos_i, $left_i$ y $right_i$; donde $left_i$ y $right_i$ son las posiciones de la matriz de los hijos

izquierdo y derecho del nodo i. Un séptimo arreglo guarda la secuencia comprimida C.

Una vez construido el árbol completo, se podan todas las reglas que representan menos que T elementos del *LCP*. Esto toma tiempo lineal sobre el número de reglas y el texto comprimido tomando tiempo $O(|R| + |C|)$. Se recorre secuencialmente la matriz que representa las reglas y cuando $cover_i < T$ se guarda la posición i en un tabla de *hash* con las reglas a borrar. Notar que la regla a borrar puede aparecer en la secuencia de reglas C, en cuyo caso no debe borrarse, razón por la cual se recorre secuencialmente C sacando de la tabla de *hash* toda regla $C[i]$ que pertenezca. Por último se crea la nueva matriz que representa el árbol podado, recorriendo la matriz original copiando los valores y saltándose las reglas podadas que están en la tabla de *hash.* Esto provoca un re-ordenamiento de las reglas, por lo que hay que renombrar de manera correcta los punteros a los hijos, que están representados por los valores $left_i$ y $right_i$. Por ejemplo, si $k = left_i$, quiere decir que el hijo izquierdo del nodo en la posición i de la matriz, es aquél que está en la posición k de la misma, pero luego del podado es posible que la regla k, si es que no fue podada, está en otra posición $k' < k$. Para ello se crea un arreglo temporal A que guarda todas los cambios de posiciones a medida que se recorre la matriz podándola. En el caso anterior, se guardaría el valor $A[k] = k'$, lo que permitiría en una pasada renombrar los hijos cuyo valor es k con la nueva posición correcta k'.

Para garantizar que cualquiera de las consultas que la estructura soporta no deba acceder más de T veces al arreglo *LCP* por regla que se decide recorrer en profundidad hasta llegar a una hoja, toda hoja cumple la siguiente propiedad:

Definición 4.28. Las hojas de los árboles representados por la estructura *NPR-RP* siempre van a tener un padre cuya cobertura de elementos es mayor a T, y la hoja va a tener una cobertura menor o igual a T.

Esto nos blinda del caso en que una regla i cubra más que T elementos, pero este compuesta de dos reglas que cubren menos que T elementos cada una, por lo que al podar el árbol i queda una hoja que representa una secuencia de largo mayor a T que habrá que recorrer secuencialmente. Dada la definición, no será necesario recorrer más de T elementos en búsqueda de la posición del mínimo que la hoja identifica. Incluso en casos de reglas cuya cobertura sea muy grande (mayor a T), pero todos los subárboles con una cobertura menor a T, se cumple que no se acceden más de T elementos del *LCP*. Esto tiene un costo de mantener en memoria información extra de algunas reglas cuya cobertura es menor a T, con el fin de garantizar un tiempo máximo por consulta. Para implementar esto se creó una nueva tabla de *hash* marcando todas las reglas que son hojas dada la definición anterior, de tal manera que no se eliminen.

La matriz de las reglas es guardada como seis arreglos que son comprimidos con el método *DAC [BLN09]* (sección 2.9), que brinda un acceso rápido y a tiempo constante. C es guardado explícitamente en un arreglo ocupando un largo fijo de *bits.*

Comprimiendo aún más la secuencia comprimida

Notar que en la secuencia C pueden haber reglas con cobertura menor T e incluso elementos de *LCP'* que no fueron incorporados en ninguna regla. Esto nos da espacio para comprimir la secuencia C sin perder la garantía de no tener que escanear más de T elementos de *LCP* por regla. De tal manera que se recorre secuencialmente y se agrupan dos o más reglas y/o símbolos cuya suma cubra menos que T elementos, reemplazándolas por una nueva regla artificial que se crea. Notar que se agrupan símbolos o reglas con cobertura menor T por lo tanto son hojas del árbol podado, luego estas nuevas reglas se guardan en la misma matriz de reglas y se tratan igual y con la misma información que una hoja del árbol. De tal manera que se obtiene una secuencia C más corta a costo de agregar nuevas reglas pero que en promedio hace que se reduzca el espacio

total, ya que C puede contener muchos símbolos terminales. Luego, en el mejor caso se reemplazan T símbolos por una regla, mientras que en el peor caso sólo dos símbolos son reemplazados, pero que probablemente son eliminadas de la matriz de reglas.

4.21.3. Resultados experimentales

Sobre los textos definidos en la tabla 4.5 y en la máquina presentada en sección 4.19.1 se comparan las estructuras que representan el RMQ. Para ello se eligieron hojas $((v_l, v_r), v_l = v_r)$ al azar y luego se fue subiendo por el árbol de sufijos hasta llegar a la raíz, dado que para saber el padre de un nodo se necesita resolver $NSV(v_r)$ y $PSV(v_l)$. Se midieron los tiempos promedios que toma cada consulta y el número de accesos promedios al arreglo *LCP*. Además por cada nodo se calculó su *string depth*, que corresponde a resolver $RMQ(v_l + 1, v_r)$, al cual también se midió el tiempo promedio y los accesos promedios al LCP. Se midieron 500.000 nodos por consulta sobre la estructura *NPR-RP* con los parámetros $T = \{64, 128, 256, 512\}$ y sobre *NPR-CN* con los parámetros $L = \{32, 64, 128, 254, 512\}$. Los resultados se pueden apreciar en las figuras 4.13 a 4.17.

Queda evidenciado que el mayor costo en la práctica al responder las consultas *NSV* y *PSV* consiste en los accesos al arreglo *LCP.* Los gráficos muestran para cada consulta los tiempos promedios (a la izquierda) y los accesos promedios al *LCP (*a la derecha), donde puede verse que ambas curvas tienen la misma forma y pendiente, evidencia de que el tiempo de consulta es directamente proporcional y en gran medida en base a los accesos al *LCP.* Hay excepciones como el caso del texto Para (Figura 4.15) donde en las consultas *NSV* y *PSV* a medida que los accesos al *LCP* disminuyen, la estructura ocupa más espacio pero el tiempo empieza a elevarse, no siguiendo la tendencia. Esto se debe a que para asegurar menos accesos al *LCP,* se podan menos reglas, cuya consecuencia es tener un árbol de mayor altura. Luego el tiempo de recorrer el árbol en profundidad comienza a ser considerable por sobre los accesos al *LCP* que son cada vez menores. En ambas estructuras, *NPR-RP* y *NPR-RPBal* se evidencia este fenómeno, que en *NPR-RPBal* es menos marcado debido a que las reglas son más balanceadas y menos profundas. Las dos alternativas, en ambas consultas *NSV* y *PSV* muestran en la mayoría resultados más eficientes en tiempo tanto como en espacio, especialmente cuando la estructura ocupa poco espacio, es decir para T mayores.

En *RMQ,* ambas estructuras basadas en *RP* presentan resultados variables, pero en muchos casos competitivos tanto en espacio como en tiempo, aunque se muestran muy dependiente de la compresión que logra *Re-Pair* sobre el arreglo *LCP.* Destacando la estructura *NPR-RPBal,* que es superior, en tiempo y espacio, tanto a *NPR-RP* como a *NPR-CN.* Si recordamos la tabla 4.7, ambos textos *DNA* y *Escherichia* tienen la secuencia C menos comprimida con respecto a las otras y por sobre todo sus coberturas máximas son muy bajas. Luego, dado que para responder *RMQ* es necesario recorrer todo el rango en busca del mínimo, que las coberturas sean bajas implica tener que recorrer más reglas en la secuencia C. Esto aumenta significativamente el tiempo en comparación con los textos más comprimibles por *Re-Pair,* donde el largo de C es mucho menor que el texto original. Notar que a pesar que los accesos al *LCP* sean bajos, el tiempo de recorrer C en esos casos es el que más contribuye al costo. Ejemplo notorio de eso ocurre en el texto *P*ara, en la estructura *NPR-RP*, mientras que su alternativa balanceada logra solucionar este efecto logrando excelente resultados. La compresión sobre C, agrupando las reglas menores que T, logra tanto una ganancia en espacio como en tiempo, ya que el rango que es necesario recorrer sobre C se acorta, lo que se puede ver en la tabla a medida que aumenta T (de derecha a izquierda).

Analizando el caso del texto *DNA,* que no es una colección repetitiva, es interesante nota que dado que la cobertura de las reglas es baja, al agrupar las reglas de C con un T grande (sobre el promedio), C

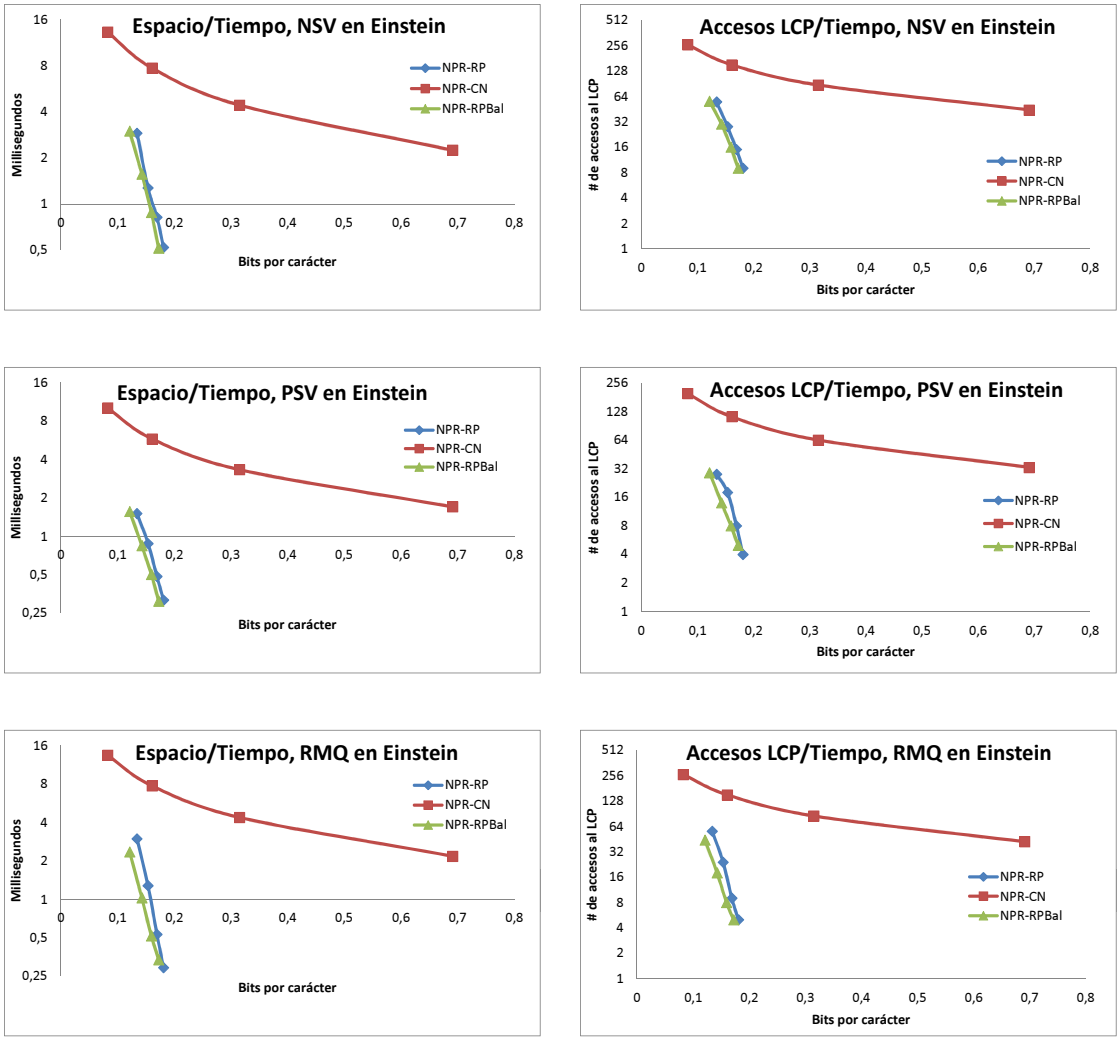

Fig. 4.13: Resultados de tiempos y accesos al LCP promedio para las consultas NSV, PSV y RMQ sobre el texto Einstein. Notar que en el eje vertical el tiempo está en escala logarítmica. Para la estructura *NPR-RP* y *NPR-RPBal* se ocuparon los parámetros $T = 64, 128, 256, 512$ y para estructura *NPR-CN* se ocuparon los parámetros $L = 64, 128, 256, 512$.

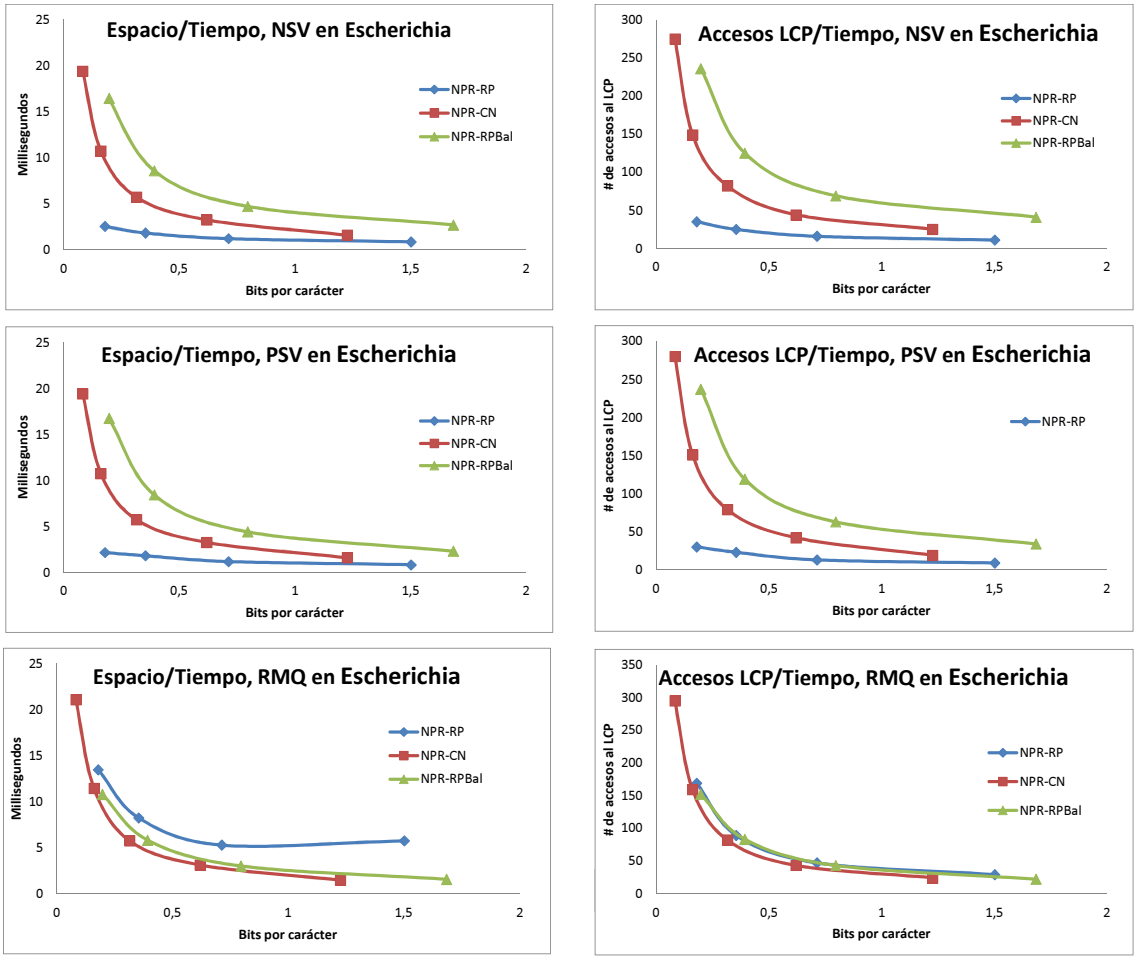

Fig. 4.14: Resultados de tiempos y accesos al LCP promedio para las consultas NSV, PSV y RMQ sobre el texto Escherichia Coli. Para la estructura *NPR-RP* y *NPR-RPBal* se ocuparon los parámetros $T = 64, 128, 256, 512$ y para estructura *NPR-CN* se ocuparon los parámetros $L = 32, 64, 128, 256, 512$.

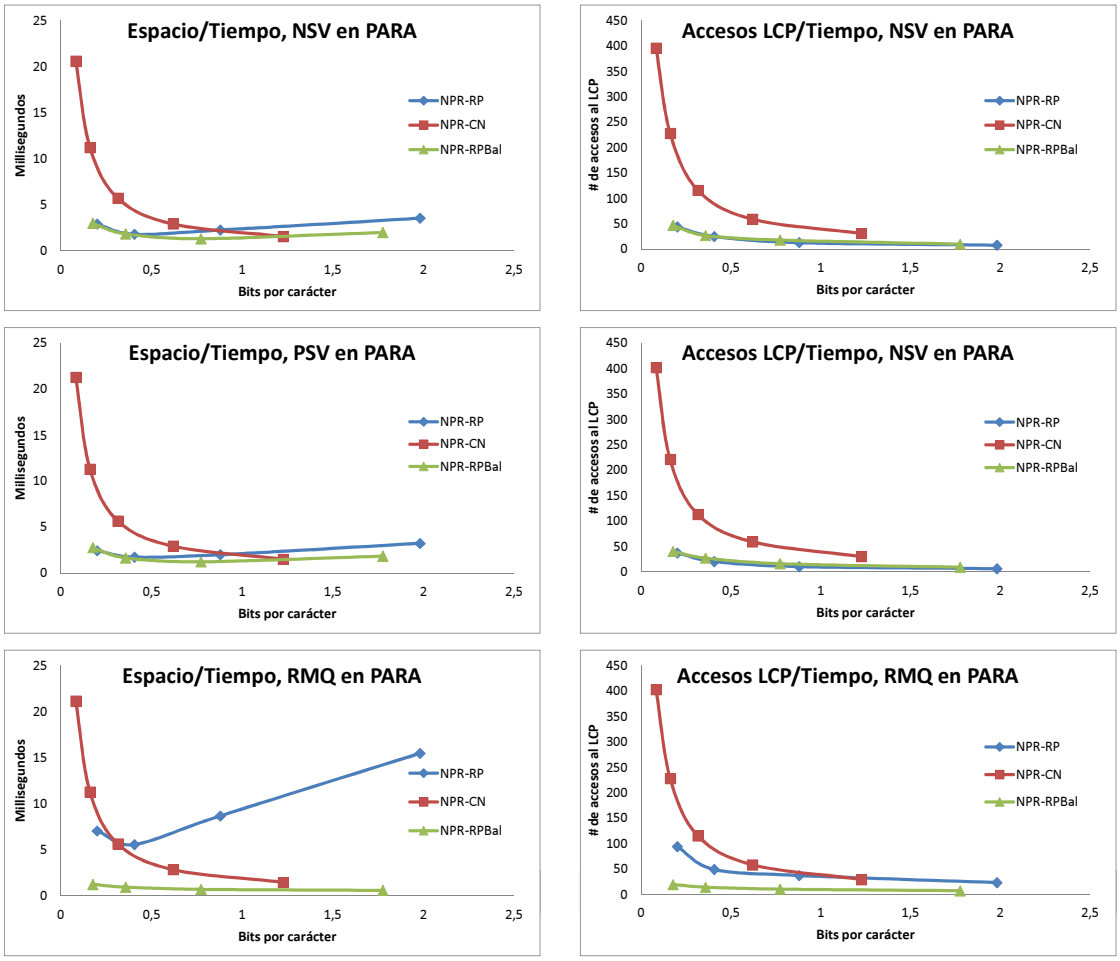

Fig. 4.15: Resultados de los tiempos y accesos al LCP promedio para las consultas NSV, PSV y RMQ sobre el texto PARA. Para la estructura *NPR-RP* y *NPR-RPBal* se ocuparon los parámetros $T = 64, 128, 256, 512$ y para estructura *NPR-CN* se ocuparon los parámetros $L = 32, 64, 128, 256, 512$.

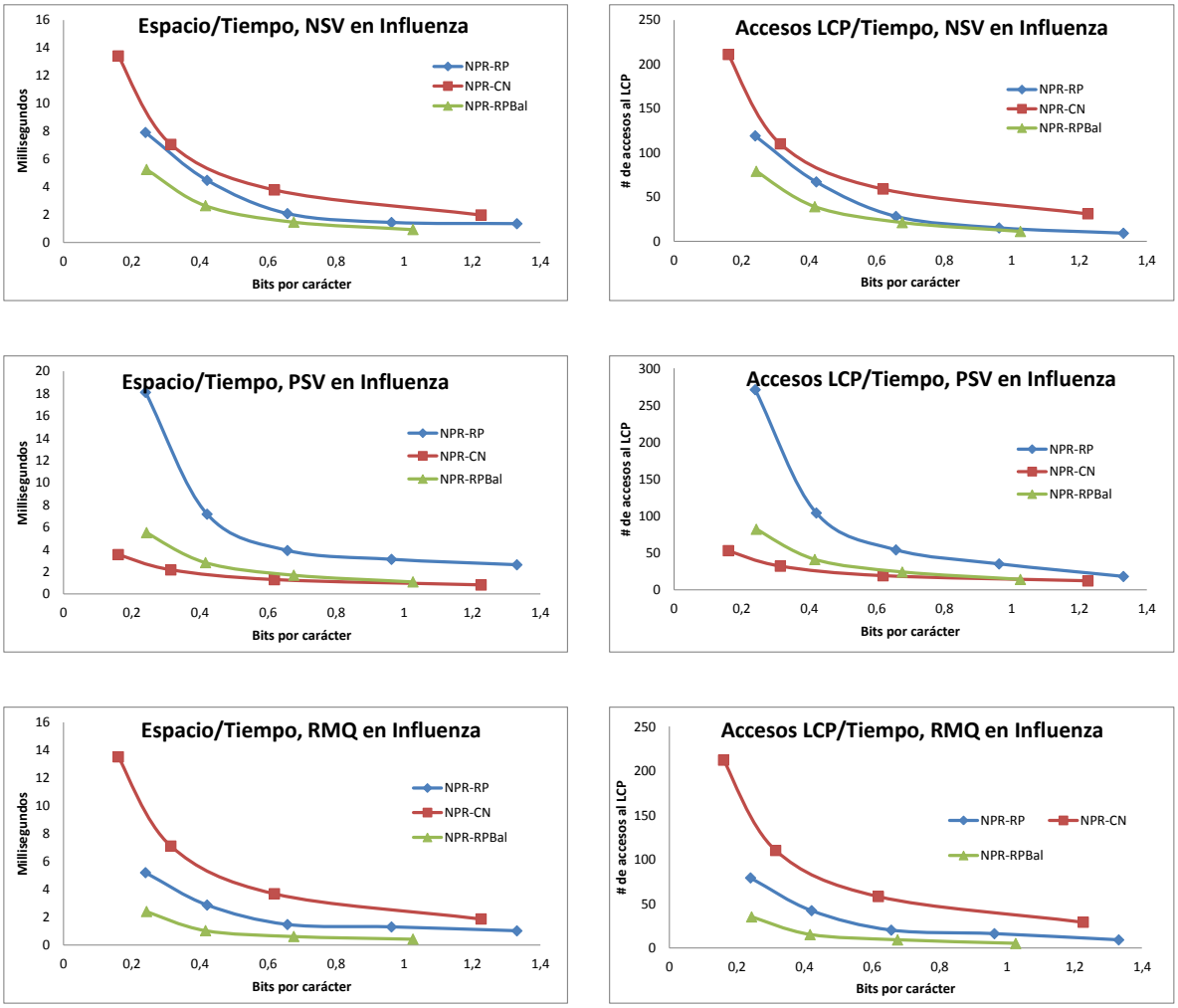

Fig. 4.16: Resultados de los tiempos y accesos al LCP promedio para las consultas NSV, PSV y RMQ sobre el texto Influenza. Para la estructura *NPR-RP* se ocuparon los parámetros $T = 64, 128, 256, 512, 1024$, para *NPR-RPBal* $T = 64, 128, 256, 512$ y para estructura *NPR-CN* se ocuparon los parámetros $L = 32, 64, 128, 256, 512$.

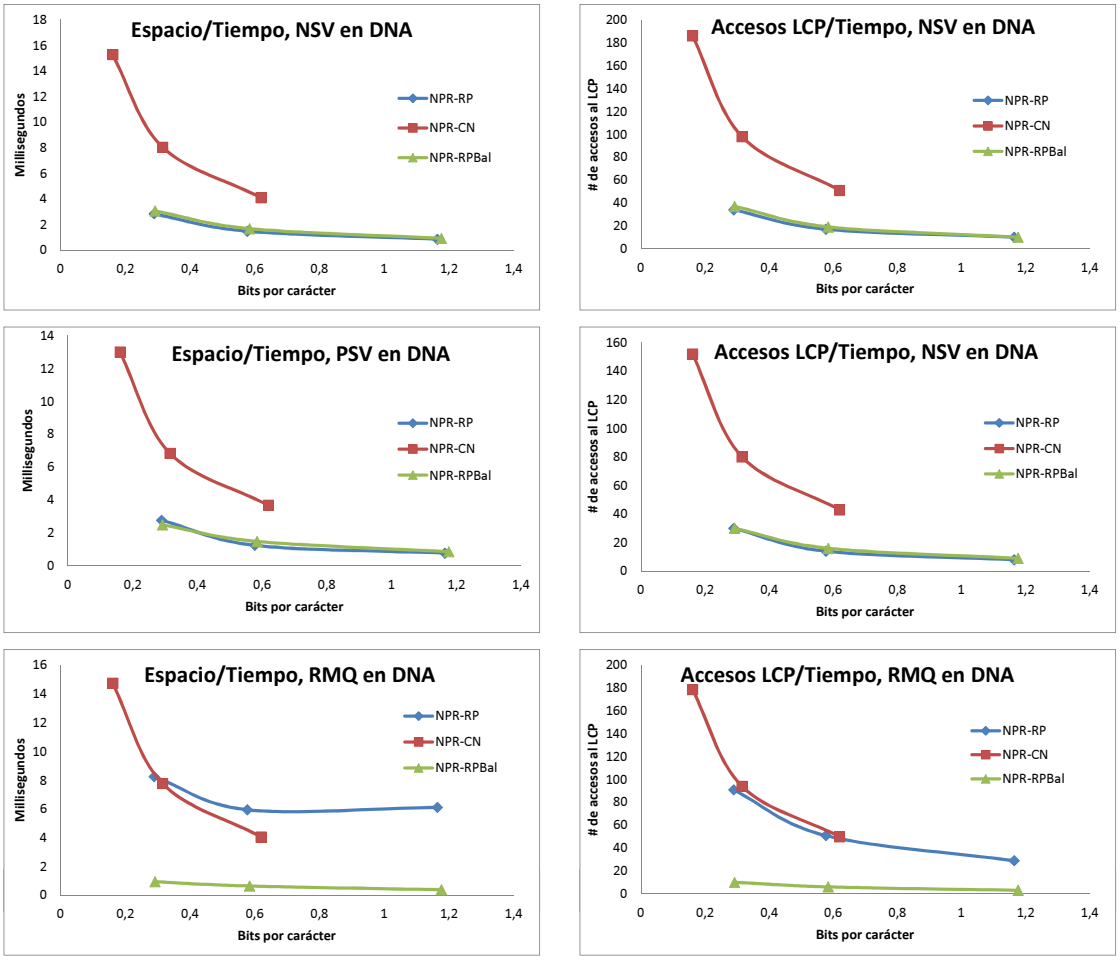

Fig. 4.17: Resultados de los tiempos y accesos al LCP promedio para las consultas NSV,PSV y RMQ sobre el texto DNA. Para la estructura *NPR-RP* y *NPR-RPBal* se ocuparon los parámetros $T = 64, 128, 256$ y para estructura *NPR-CN* se ocuparon los parámetros $L = 64, 128, 256$.

queda prácticamente particionado en bloques de largo homogéneo, muy parecido a la estructura *NPR-CN,* y aún así se obtienen resultados competitivos, en especial *NPR-RPBal* con tiempos casi constantes y muy pocos accesos al LCP en la consulta *RMQ*.

Mientras que en los textos menos repetitivos los resultados son competitivos a los de la estructura *NPR-CN,* en los textos más repetitivos la compresión y los bajos tiempos obtenidos en ambas alternativas basadas en *repair* superan a *NPR-CN*. Destacándose el caso de *Einstein,* cuya secuencia C es la más comprimida (0,12 %), permitiendo una búsqueda muy rápida donde incluso los mejores tiempos para *NPR-CN* son más lentos que los peores tiempos para *NPR-RP y NPR-RPBal.*

Cabe destacar los sorprendentes resultados obtenidos sobre el texto Einstein, texto que tiene la mayor compresión por *Re-Pair* (0,28 %), siendo altamente repetitiva. Mientras más repetitivo es el texto, *Re-Pair* genera menos reglas, lo que aumenta la compresión, razón por la cual al podar las reglas el tamaño no se ve tan afectado como en los otros textos ya que tienen más reglas y por ende de menor profundidad que podar. Además, los tiempos en ambas estructuras construidas superan a *NPR-CN* en todas las consultas y para todos los distintos parámetros T.

Sobre el espacio de nuestras estructuras, se tiene que el diccionario de reglas ocupa en promedio el 70 % del espacio ocupado por la estructura completa, lo restante es ocupado por la secuencia comprimida C y el arreglo de muestreos. Tanto las reglas como C se guardaron comprimidas con el método DAC (sección 2.9), mientras que los arreglos *min_pos* y *cover* presenta buena compresión dado que representan un conjunto de números acotado. Los otros arreglos donde los elementos son más variables tienen una pésima compresión, incluso ocupando en algunos casos más que $\log(\text{máx}_i(A[i]))$ *bits* en promedio por elemento, donde $\text{máx}_i(A[i])$ representa el máximo contenido en un arreglo A. Por lo tanto se decidió guardar los arreglos *min*, *sum* y C en un arreglo, ocupando $\log(\text{máx}_i(A[i]))$ *bits* constante por elemento, respectivamente para cada arreglo.

Aún para textos no repetitivos nuestras estructuras resultan competitivas a *NPR-CN.* En los casos que la compresión por *repair* no es buena, la agrupación de los símbolos en C logran que la estructura generada por *NPR-RP* sea parecida a la de *NPR-CN.* En estos casos, C es prácticamente equivalente al primer nivel de la jerarquía que tiene la estructura *NPR-CN.* Esto debido a que pocas reglas van a ser largas (mayores a T), por lo tanto la agrupación sobre C, va a crear en su mayoría bloques de largo constante T. Esto da lugar a dos propuestas: Primero dado los buenos resultados de *NPR-RP* es posible optimizar la estructura *NPR-CN* para obtener mejores tiempos basados en los algoritmos aplicados para las consultas en *NPR-RP,* en particular como fueron tratados los bordes para cada consulta, que de ser bien implementado puede bajar los accesos al *LCP.* Segundo se propone crear una jerarquía sobre C, análogo a lo que hace *NPR-CN* sobre el *LCP,* para reducir por sobre todo los tiempos a la consulta *RMQ,* la que incurre en un alto costo al tener que recorrer C.

Comparando nuestras estructuras, ambas se comportan de manera similar en la consultas *NSV* y *PSV.* En las consultas *RMQ, NPR-RPBal* muestra el mejor rendimiento, quedando en evidencia que las reglas son más balanceadas y las coberturas en C son más homogéneas, por ende la representación que experimentalmente muestra los mejores resultados y consistencia en los distintos textos es *NPR-RPBal.*

Capítulo 5

Conclusiones

Los índices comprimidos son una técnica vastamente usada, tanto por su rapidez para buscar patrones dentro de un texto, como por el pequeño espacio que ocupan, permitiendo incluso indexar secuencias de gran tamaño en un computador personal. Con el mismo afán, es que constantemente se están buscando mejoras y nuevas representaciones que permitan más eficiencia tanto en tiempo como en espacio. La mayoría de los métodos están pensados para textos en general, pero este trabajo se enfoca sobre una clase particular de textos, colecciones altamente repetitivas. Esta clase de textos son inherentemente muy compresibles dada sus repeticiones, propiedad que puede ser aprovechada para crear una estructura que responda rápidamente y ocupando el menor espacio posible. En particular se ha trabajado sobre los árboles de sufijos comprimidos, proponiendo e implementando una optimización para generar mejores resultados sobre la clase se textos repetitivos.

5.22. Contribuciones de esta memoria

5.22.1. Compresión del arreglo *LCP*

Se ha presentado una práctica y simple variante del LCP de Fischer representado por los arreglos O y Z. Basados en el hecho teórico que para textos altamente repetitivos el LCP diferencial tiene una cantidad de *runs* acotada y mucho menor que el largo del texto, se codificaron ambos arreglos O y Z usando un Bitmap basado en la codificación δ.

La variante presenta una considerable mejora (40 %) en la compresión del LCP comparado con las otras alternativas a cambio de un pequeño aumento (3 %) en los tiempos de accesos. En la práctica los tiempos de las tres alternativas son básicamente iguales, por lo tanto la variante propuesta proporciona el mejor rendimiento con un compromiso de espacio/tiempo superior a la alternativas.

5.22.2. Estructura que soporta la operaciones *NSV, PSV* y *RMQ*

Se ha presentado una novedosa estructura que soporta consultas *NSV, PSV* y *RMQ* sobre el arreglo *LCP,* esenciales para la mayoría de las operaciones sobre los árboles de sufijos. La estructura propuesta explota el hecho de que en los textos altamente repetitivos el arreglo LCP diferencial también muestra muchas repeticiones, logrando una mejor compresión del mismo y mejores tiempos a las consultas. La estructura básicamente corresponde al árbol obtenido comprimiendo el LCP diferencial *LCP'* con el método *Re-Pair* y agregando información extra en los nodos. Resulta que se obtienen muy buenos rendimientos tanto en espacio como en tiempo a medida que la secuencia *LCP'* sea más compresible por el método *Re-Pair,* hecho que ocurre a medida que el texto es más repetitivo.

A diferencia de la alternativa expuesta *NPR-CN,* que presentan un comportamiento constante dado que no dependen del texto, nuestra estructura por un lado muestra un comportamiento un tanto impredecible y variable, pero logra una ventaja comparativa tanto en espacio como en tiempo para la clase de textos repetitivos. Es más, la variante *NPR-RPBal* de nuestra estructura, logra hacer menos variable y más robusta la representación, dado que las reglas que son representadas por árboles binarios, tienden a ser más balanceadas y menos profundos, dando una mayor consistencia a los resultados. No obstante a pesar de la variabilidad, la estructura mediante el parámetro T que poda el árbol, garantiza un máximo de accesos al LCP, respondiendo

las consultas en $O(T)$. Luego es posible jugar con T para obtener el mejor compromiso de espacio/tiempo propio de cada texto.

5.23. Trabajo Futuro

Las estructuras *NRP-RP y NRP-RPBal,* presentan buenos resultados, sin embargo aún hay espacio para mejorarlos, en particular la estructura permite una mayor compresión del diccionario de reglas que representan aproximadamente el 70 % del espacio ocupado por la estructura completa.

González y Navarro [GN07] proponen una técnica para comprimir el diccionario de reglas que permite borrar toda definición de regla que es usada más de una vez por otras reglas, finalmente guardando el diccionario en un *bitmap* con resultados de hasta un 50 % de compresión del diccionario. Al podar el árbol uno esta quitando las reglas cortas y además estrechando el árbol, lo que intuitivamente da a pensar que esas reglas podadas son las más referenciadas por otras reglas, luego la compresión del diccionario no sería tan efectiva para nuestra estructura, a pesar de que sí debiese comprimir. Por esa razón, esta propuesta no se implementó dado que se dio más prioridad a obtener mejores tiempos de respuesta, mientras que esta compresión debiese subirlos prometiendo una baja compresión del espacio.

Por otro lado, las reglas fueron representadas como arreglos por lo que habría que analizar otras formas de comprimir aún más los arreglos permitiendo accesos en tiempo constante, en particular min, sum y C que están guardados explícitamente, pudiendo explotar alguna relación que exista entre los arreglos por ejemplo el hecho que $sum_i - min_i \geq 0$.

Dado la naturaleza impredecible de *Re-Pair,* al construir la estructura para cada texto se obtienen distintos resultados en términos de espacio/tiempo bajo el mismo parámetro T. Por eso el T debiese ser fijado para cada texto de acuerdo a lo que se busque optimizar, lo cual se puede haciendo un simple análisis de la distribución de las coberturas de todas las reglas antes de podar. Queda propuesto hacer un análisis del árbol generado por *Re-Pair* a fin de encontrar el T que mejor se ajuste según otro parámetro de espacio/tiempo.

Alentado por los buenos resultados de compresión, en especial el caso de los textos que tienen mayor compresión con el método *Re-Pair* (como el texto *Einstein) y* notando que si no se poda el árbol la estructura representa el *LCP* diferencial. completo. Surge la idea de reemplazar la representación del *LCP* y del *NPR* por el árbol completo de nuestra estructura, de tal manera que no sólo soporte las operaciones *NSV, PSV* y *RMQ,* sino que también la misma soporte accesar a cualquier posición del *LCP,* recordando que *Re-Pair* ya se ha implementado para comprimir el *LCP* [FMN09].

Bibliografía

Referencias

[ACNS10] D. Arroyuelo, R. Cánovas, G. Navarro, and K. Sadakane. Succinct trees in practice. *Proc. 11th Workshop on Algorithm Engineering and Experiments* (ALENEX), pages 84-97, 2010.

[AKO04] M. Abouelhoda, S. Kurtz, and E. Ohlebusch. Replacing Suffix Trees with Enhanced Suffix Arrays. *Journal of Discrete Algorithms*, pages 53-86, 2004.

[BDM05] D. Benoit, E. Demaine, I. Munro, R. Raman, V. Raman, and S. Rao. *Representing trees of higher degree*. Algorithmica, 43(4):275-292, 2005.

[BLN09] N. Brisaboa, S. Ladra, and G. Navarro. Directly addressable variable-length codes. In Proc. *16th String Processing and Information Retrieval Symposium (SPIRE)*, pages 122–130, 2009.

[BW94] M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. *Technical Report 124, Digital Equipment Corporation*, 1994.

[Can10] R. Cánovas, Estructuras Comprimidas para Árboles de Sufijos, Tesis para optar al grado de Magíster en Ciencias de la Computación, 2010. [`http://www.dcc.uchile.cl/~gnavarro/algoritmos/tesisCanovas.pdf`]

[Cla98] D. Clark. Compact Pat Trees. PhD thesis, University of Waterloo, Ontario, Canada, 1998.

[CN08] F. Claude and G. Navarro. Practical rank/select queries over arbitrary sequences. *In Proc. Symposium on 15th String Processing and Information Retrieval (SPIRE)*, LNCS 5280, pages 176–187, 2008.

[CN10] R. Cánovas and G. Navarro. Practical compressed suffix trees. *Proc. 9th International Symposium on Experimental Algorithms (SEA)*, pages 94-105, 2010.

[FGNV09] P. Ferragina, R. González, G. Navarro, and R. Venturini, Compressed text indexes: from theory to practice. *ACM Journal of Experimental Algorithmics (JEA)*, article 12, 2009.

[FMMN07] P. Ferragina, G. Manzini, V. Makinen, and G. Navarro. Compressed representations of sequences and full-text indexes. ACM Transactions on Algorithms, 3(2):article 20, 2007.

[FMN09] J. Fischer, V. Mäkinen, and G. Navarro. Faster Entropy-Bounded Compressed Suffix Trees. *Theoretical Computer Science,* pages 5354-5364, 2009.

[GGMN05] R. González, Sz. Grabowski, V. Makinen, and G. Navarro. Practical implementation of rank and select queries. *In Proc. 4th Workshop on Experimental Algorithms (WEA)*, pages 27-38, 2005.

[GN07] R. González and G. Navarro. Compressed text indexes with fast locate. In Proc. 18th a Annual Symposium on Combinatorial Pattern Matching (CPM), LNCS 4580, pages 216–227, 2007.

[GV05] R. Grossi and J. Vitter. *Compressed sux arrays and sux trees with applications to text indexing and string matching*. In SIAM Journal on Computing, pages 397-406, 2005.

[HMR07] M. He, J. Munro and S. Srinivasa Rao, Succinct ordinal trees based on tree covering, *Proc. of the 34th International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 509-520, 2007.

[Jac89] G. Jacobson. *Space-efficient static trees and graphs.* In Proc. 30th Symposium on Foundations of Computer Science (FOCS), pages 549-554, 1989.

[KLAAP01] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. *In Proc. 12th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 181-192, 2001.

[KM99] R. Kosaraju and G. Manzini. Compression of low entropy strings with Lempel- Ziv algorithms. *SIAM Journal on Computing*, pages 893–911, 1999.

[KN10] S. Kreft and G. Navarro. LZ77-like compression with fast random access. *In Proc. 20th Data Compression Conference (DCC)*, pages 239-248, 2010.

[LM00] J. Larsson and A. Moffat. Off-line dictionary-based compression. *Proc. of the IEEE*, pages 1722-1732, 2000.

[MN05] V. Mäkinen and G. Navarro. *Succinct Suffix Arrays based on Run-Length Encoding.* Nordic Journal of Computing (NJC) 12(1):40-66, 2005

[MNSV10] V. Mäkinen, G. Navarro, J. Sirén, and N. Välimäk, Storage and Retrieval of Highly Repetitive Sequence Collections. Journal of Computational Biology 17(3):281-308, 2010.

[Mun96] I. Munro. Tables. *In Proc. 16th Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, LNCS 1180, pages 37–42, 1996.

[NM07] G. Navarro and V. Makinen. Compressed full-text indexes. *ACM Computing Surveys, 39(1)*, article 2, 2007.

[NR08] G. Navarro and L. Russo. Re-Pair Achieves High-Order Entropy. *Proc. DCC'08*, page 537 (poster)

[OS07] D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. *In Proc. 9th Workshop on Algorithm Engineering and Experiments (ALENEX).* SIAM Press, 2007.

[RNO11] L. Russo, G. Navarro, and A. Oliveira. Fully-Compressed Suffix Trees. *ACM Transactions on Algorithms,* 7(4): article 53, 2011

[RRR02] R. Raman, V. Raman, and S. Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. *In Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 233-242. SIAM Press, 2002.

[Sad03] K. Sadakane. New text indexing functionalities of the compressed suffix arrays. J*ournal of Algorithms,* 48(2), pages 294-313, 2003.

[Sad07] K. Sadakane, Compressed suffix trees with full functionality, *Theory of Computing Systems*, 41(4), pages 589-607, 2007.

[SN10] K. Sadakane and G. Navarro. Fully-functional succinct trees. *In Proc. 21st Symposium on Discrete Algorithms (SODA)*, pages 134-149, 2010.

[Sir09] J. Sirén. Compressed suffix arrays for massive data. *16th String Processing and Information Retrieval Symposium (SPIRE), LNCS 5721*, pages 63-74, 2009.

[SVMN08] J. Sirén, N. Välimäki, V. Mäkinen, and G. Navarro. Run-Length compressed indexes are superior for highly repetitive sequence collections. *In Proc. 15th String Processing and Information Retrieval Symposium (SPIRE),* pages 199-208, 2008.

[VGDM07] N. Välimäki, W. Gerlach, K. Dixit, and V. Makinen. Engineering a compressed sufix tree implementation. *In Proc. 6th Workshop on Experimental Algorithms (WEA)*, pages 217-228, 2007.

[WZ99] H. Williams and J. Zobel. Compressing integers for fast file access. *The Computer Journal*, 42:193–201, 1999.