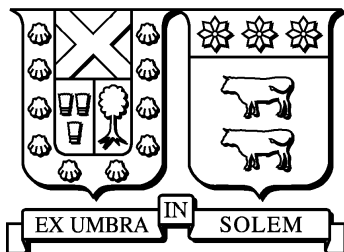


UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA

DEPARTAMENTO DE INFORMÁTICA

SANTIAGO – CHILE



“REDISEÑO DE AUTOÍNDICES COMPRIMIDOS
PARA TEXTOS BASADOS EN LEMPEL ZIV”

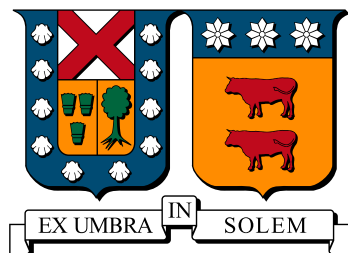
MATÍAS ARAYA COHEN

MEMORIA DE TITULACIÓN PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL INFORMÁTICO

PROFESOR GUÍA: DIEGO ARROYUELO BILLIARDI

AGOSTO 2018

UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE INFORMÁTICA
SANTIAGO – CHILE



**“REDISEÑO DE AUTOÍNDICES
COMPRIMIDOS PARA TEXTOS BASADOS EN
LEMPER ZIV”**

MATÍAS ARAYA COHEN

**MEMORIA DE TITULACIÓN PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL INFORMÁTICO**

PROFESOR GUÍA: DIEGO ARROYUELO BILLIARDI
PROFESOR CORREFERENTE: LIUBOV DOMBROVSKAIA

AGOSTO 2018

MATERIAL DE REFERENCIA, SU USO NO INVOLUCRA RESPONSABILIDAD DEL AUTOR O DE LA INSTITUCIÓN

Agradecimientos

El primer agradecimiento es para mi familia en Viña, quienes me ayudaron a comenzar este camino en la sede de Valparaíso, por lo que sin ellos esto no hubiera sido posible. Agradecimiento también a mis padres que me enseñaron que con esfuerzo y perseverancia todo es posible. A mis hermanas que han sido un modelo a seguir. A Nicole Aída quien me entrega un apoyo incondicional. Al profesor Diego Arroyuelo quien me guió durante todo el camino de este trabajo. Todos ellos fueron piezas fundamentales de este primer logro que estoy alcanzando en mi vida profesional.

Resumen

El problema de búsqueda en textos es cada vez más utilizado dada la cantidad de literatura e investigación que se realizan en formato digital. Para resolver este problema se encuentran varios métodos, pero el mayor problema que tienen es la cantidad de espacio que utilizan, ya que deben almacenar el texto y las estructuras de datos necesarias para la búsqueda, quedando muchas veces tamaños inmanejables en memoria principal. Las soluciones más eficientes consisten en crear índices del texto, que a su vez se comprimen, lo que facilita la búsqueda y ocupan el menor espacio posible. En este trabajo se crea un rediseño de un autoíndice comprimido basado en el algoritmo de compresión Lempel-Ziv (LZ-index abreviado). Se dice que es un rediseño ya que toma la solución creada por Navarro, la que se actualiza a tecnologías y métodos de desarrollo modernos.

Abstract

The problem of searching in texts is increasingly used given the amount of literature and research that is done in digital format. To solve this problem are several methods, but the biggest problem they have is the amount of space they use, since they must store the text and data structures necessary for the search, often leaving unmanageable sizes in main memory. The most efficient solutions consist of creating indexes of the text that are compressed, which facilitates the search and use the least possible space. In this work, a redesign of a compressed self-index based on the Lempel-Ziv compression algorithm (abbreviated LZ-index) is created. It is said to be a redesign as it takes the solution created by Navarro, which is updated to modern technologies and development methods.

Índice de Contenidos

Agradecimientos	III
Resumen	IV
Abstract	V
Índice de Contenidos	VI
Lista de Tablas	IX
Lista de Figuras	XII
Glosario	XV
Introducción	1
1. Definición del Problema	3
1.1. Objetivos	3
1.2. Formas de Solucionar el Problema	4
1.3. Búsquedas de Texto Indexada Clásicas	5
1.3.1. Suffix Trees	6
1.3.2. Suffix Arrays	7

1.4.	Estructuras de Datos	10
1.4.1.	Representación de Strings	10
1.4.2.	Representando un Conjunto de Strings con Tries	11
1.4.3.	Propiedades de los Stings y Tries	13
1.5.	Compresión de Texto	14
1.5.1.	Compresión Lempel-Ziv (LZ78)	16
1.6.	Análisis Inicial de la Solución LZ-index	21
2.	Estado del Arte	24
2.1.	LZ-índice de Kärkkäinen y Ukkonen (KU-LZI)	25
2.1.1.	Solución de KU-LZI	26
2.2.	LZ-índice de Ferragina y Manzini (FM-LZI)	27
2.2.1.	Solución de FM-LZI	28
2.3.	LZ-índice de Navarro (NAV-LZI)	29
2.3.1.	Componentes de NAV-LZI	29
2.3.2.	Espacio Experimental del Indice	32
2.3.3.	El Algoritmo de Búsqueda NAV-LZI	33
2.3.4.	Solución de NAV-LZI	35
2.4.	LZ-índice de Russo y Oliveira (ILZI)	36
2.4.1.	Solución de ILZI	37
2.5.	LZ-índice de Arroyuelo, Navarro y Sadakane (ANS-LZI)	37
2.5.1.	Solución de ANS-LZI	38
2.6.	Resumen de los LZ-indices	39
3.	Nueva Implementación de LZ-index	41
3.1.	Estructuras de Datos Sucintas	42

3.1.1.	Paréntesis Balanceados	42
3.1.2.	Representación Sucinta de Árboles	44
3.2.	Implementación del Nuevo LZ-index	45
3.2.1.	Definición de la Clase	46
3.2.2.	Principales Métodos	47
3.3.	Implementación del LZTrie	48
3.3.1.	Definición de la Clase	49
3.3.2.	Principales Métodos	50
3.4.	Implementación del Revtrie	52
3.4.1.	Definición de la Clase	53
3.4.2.	Principales Métodos	53
4.	Experimentos	56
4.1.	Configuración Experimental Inicial	56
4.2.	English Text	58
4.3.	Secuencia ADN	60
4.4.	Secuencia MIDI	63
4.5.	Archivo XML	66
4.6.	Proteínas	68
4.7.	Sources	71
	Conclusiones	74
5.	Anexos	77
	Bibliografía	79

Índice de cuadros

1.1. Comparación de distintos métodos de indexación.	23
2.1. Comparación de uso de espacio en bits que requiere cada índice.	39
2.2. Comparación entre los distintos índices para las operaciones básicas sobre un texto T , dado un patrón de búsqueda $P[1..m]$	40
3.1. Ejemplo de paréntesis balanceado con su respectiva representación de bitvector	43
3.2. Tipos de alfabeto con los que se puede parametrizar el LZ-index.	47
3.3. Tipos de paréntesis balanceados, conocidos en la librería SDSL como BPS (balanced parentheses support), con los que se puede parametrizar el LZ-index	47
4.1. Cantidad de nodos en LZTrie y cantidad de nodos en RevTrie para el texto English.	59
4.2. Cantidad de ocurrencias encontradas para los 1000 patrones de cada largo, para el texto English.	59
4.3. Porcentaje de tipo de ocurrencias para patrones de distinto largo para el texto English.	59
4.4. Cantidad de nodos en LZTrie y cantidad de nodos en RevTrie para el texto secuencia ADN.	61

4.5. Cantidad de ocurrencias encontradas para los 1000 patrones de cada largo, para el texto Secuencia ADN.	62
4.6. Porcentaje de tipo de ocurrencias para patrones de distinto largo, para el texto Secuencia ADN.	62
4.7. Cantidad de nodos en LZTrie y cantidad de nodos en RevTrie para el texto Secuencia MIDI.	64
4.8. Cantidad de ocurrencias encontradas para los 1000 patrones de cada largo, para el texto Secuencia MIDI.	65
4.9. Porcentaje de tipo de ocurrencias para patrones de distinto largo, para el texto Secuencia MIDI.	65
4.10. Cantidad de nodos en LZTrie y cantidad de nodos en RevTrie para el Archivo XML.	67
4.11. Cantidad de ocurrencias encontradas para los 1000 patrones de cada largo, para el Archivo XML.	67
4.12. Porcentaje de tipo de ocurrencias para patrones de distinto largo, para el Archivo XML.	67
4.13. Cantidad de nodos en LZTrie y cantidad de nodos en RevTrie para el texto Proteínas.	69
4.14. Cantidad de ocurrencias encontradas para los 1000 patrones de cada largo, para el texto Proteínas.	70
4.15. Porcentaje de tipo de ocurrencias para patrones de distinto largo, para el texto Proteínas.	70
4.16. Cantidad de nodos en LZTrie y cantidad de nodos en RevTrie para el texto Sources.	71

4.17. Cantidad de ocurrencias encontradas para los 1000 patrones de cada largo, para el texto Sources.	72
4.18. Porcentaje de tipo de ocurrencias para patrones de distinto largo, para el texto Sources.	72
5.1. Ocurrencias encontradas para el texto English	77
5.2. Ocurrencias encontradas para el texto Secuencia de ADN	77
5.3. Ocurrencias encontradas para el texto Secuencia MIDI	78
5.4. Ocurrencias encontradas para Archivo XML	78
5.5. Ocurrencias encontradas para el texto Proteinas	78
5.6. Ocurrencias encontradas para el texto Sources	78

Índice de figuras

1.1. Suffix Tree con texto $T = \text{“mississippi”}$	7
1.2. Suffix Array para el Ejemplo 1.3.1, el sufijo es el texto sombreado en cada línea. La columna $SA_T[i]$ guarda la posición del texto inicial para el i -ésimo sufijo en el orden lexicográfico	9
1.3. Representación Trie del conjunto de strings de S_1	12
1.4. Representación Trie del conjunto de strings de S_2	12
1.5. Descomposición por frases y con su respectivo <i>id</i> del LZ78 para el ejemplo visto en [Bil09]. $T = \text{alabar_a_la_alabarda_para_apalabrarla}$	18
1.6. Representación en formato de arreglo para la descomposición del LZ78 del ejemplo de 1.5.	19
1.7. Representación <i>LZTrie</i> para la descomposición del LZ78 del ejemplo en Figura 1.5, los números de afuera de los nodos representan el recorrido en preorden y los números de los nodos representan el <i>id</i> de la frase.	19
1.8. Arreglo <i>Node</i> para el ejemplo de Figura 1.5, donde la notación $\textcircled{k}$ representa un puntero al nodo del <i>LZtrie</i> (Figura 1.7) con preorden k	19
2.1. Descomposición LZ78 y sus correspondientes <i>phrase id</i> para un ejemplo compuesto de $T = \text{“alabar_a_la_alabarda_para_apalabrarla”}$	25
2.2. Descomposición del LZ78, con el respectivo <i>LZTrie</i> , <i>RevTrie</i>	30

2.3.	Componentes restantes del LZ-índice de Navarro. <i>Range</i> y <i>Nodo</i> del ejemplo previamente visto. Por simplicidad en la práctica se utiliza la Estructura de Datos <i>RNode</i> donde se obtiene la misma información que en <i>Range</i>	31
2.4.	Esquema original de NAV-LZI.	38
2.5.	Esquema de ANS-LZI que logra una mejora en la utilización de espacio. . .	38
3.1.	Paréntesis balanceado del árbol mostrado en 3.2	44
3.2.	Árbol correspondiente a paréntesis balanceados de la Figura 3.1	44
4.1.	Tamaño en MB del texto English, LZ-index original y el nuevo LZ-index desarrollado en este trabajo.	58
4.2.	Tiempo requerido en contar 1000 patrones de largo 5, 8, 12 y 20 en texto English.200MB con el LZ-index original y versión nueva.	60
4.3.	Tamaño en MB que utiliza el texto Secuencia de ADN, LZ-index original y el nuevo LZ-index desarrollado en este trabajo.	61
4.4.	Tiempo requerido en contar 1000 patrones de largo 5, 8, 12 y 20 en texto Secuencia ADN con el <i>LZ-index</i> original y versión nueva.	63
4.5.	Tamaño en MB que utiliza el texto Secuencia MIDI, LZ-index original y el nuevo LZ-index desarrollado en este trabajo.	64
4.6.	Tiempo requerido en Contar 1000 patrones de largo 5, 8, 12 y 20 en Texto Secuencia MIDI Pitches con el <i>LZ-index</i> original y versión nueva.	65
4.7.	Tamaño en MB que utiliza el texto Archivo XML, LZ-index original y el nuevo LZ-index desarrollado en este trabajo.	66
4.8.	Tiempo requerido en contar 1000 patrones de largo 5, 8, 12 y 20 del Archivo XML con el <i>LZ-index</i> original y versión nueva.	68

4.9. Tamaño en MB que utiliza el texto Proteínas, LZ-index original y el nuevo LZ-index desarrollado en este trabajo.	69
4.10. Tiempo requerido en contar 1000 patrones de largo 5, 8, 12 y 20 en texto Proteínas con el <i>LZ-index</i> original y versión nueva.	70
4.11. Tamaño en MB que utiliza el texto Sources, LZ-index original y el nuevo LZ-index desarrollado en este trabajo.	71
4.12. Tiempo requerido en contar 1000 patrones de largo 5, 8, 12 y 20 en texto Sources con el <i>LZ-index</i> original y versión nueva.	73

Glosario

1. **LZ-index**: índice comprimido basado el algoritmo Lempel Ziv.
2. **LZTrie**: Árbol con formato de estructura de datos Trie.
3. **RevTrie**: Árbol con formato de estructura de datos Trie inverso al LZTrie.
4. **SDSL**: Succint Data Structure Library
5. **KU-LZI**: LZ-índice de Kärkkäinen y Ukkonen.
6. **FM-LZI**: LZ-índice de Ferragina y Manzini.
7. **NAV-LZI**: LZ-índice Navarro.
8. **ILZI**: LZ-índice Russo y Oliveira.
9. **ANS-LZI**: LZ-índice Arroyuelo, Navarro y Sadakane.
10. **LZ78**: Algoritmo de Compresión Lempel Ziv realizado en 1978.

Introducción

La *Búsqueda en Texto* es un problema clásico en las Ciencias de la Computación, este problema principalmente trata que dado un texto, que tiene una cantidad de símbolos (letras, números, signos de puntuación, etc.), se desea buscar un patrón que puede ser una palabra, subpalabra o frase. Este patrón se busca a lo largo de todo el texto y el algoritmo retorna todas las ocurrencias y lugar específico donde se encuentra. El objetivo de esta memoria es resolver y analizar la solución a través del índice comprimido con Lempel Ziv, el cual es comúnmente llamado en la literatura como **LZ-index**.

Dada la conectividad global que existe gracias a Internet, se pueden encontrar grandes colecciones de textos digitales disponibles, en los cuales usualmente se debe buscar información específica a lo largo de todo el texto. Cuando este es de un tamaño considerable este problema se vuelve interesante, ya que ejecutar un algoritmo complejo que tenga estructuras de datos asociadas (índices) solamente en memoria principal se vuelve un desafío.

Seguramente todas las personas que suelen utilizar literatura digital, han utilizado la herramienta de buscar un patrón en todo el texto, sin embargo existen otras áreas en las que la búsqueda de texto cumple un rol fundamental en sus funciones:

1. Bioinformática: Desde la llegada de las tecnologías de ADN secuenciales, estas se han relacionado enormemente con el procesamiento de texto. Esto se debe a que este tipo de secuencias y otras dentro de esta ciencia se puede representar de igual manera que un texto.
2. Procesamiento de Música: Musical Instrument Digital Interface (MIDI) es un formato de un protocolo que permite la comunicación entre un computador y un instrumento

musical. Un archivo MIDI consiste en una secuencia de eventos, usualmente textos, que permite reconstruir el audio original.

3. Librerías Digitales: Hoy en día se suele convertir los documentos a su formato digital y estos son guardados en repositorios. Para poder encontrar y usar la información de este repositorio (colecciones de textos) es necesario proveer buenas capacidades de búsquedas.
4. Documentos XML: El bien conocido *eXtensible Markup Language* (XML) se ha convertido en uno de los estándares para compartir información entre aplicaciones en la Web, entre otros usos. Como un documento XML puede contener texto, es natural que se tenga la capacidad de una búsqueda completa.
5. Repositorios de Código: Los repositorios de códigos son cada vez más utilizados y corresponden a una herramienta indispensable en el Desarrollo de Software. Ya sea en Bitbucket, GitHub u otro, permiten al usuario ver las distintas versiones del software que contiene usualmente una gran cantidad de texto, con lo que una búsqueda rápida sobre este se vuelve fundamental.

El gran desafío de la búsqueda completa de texto es que sea lo más rápido posible y que a su vez ocupe la menor cantidad de espacio para que así sea factible poder trabajar en memoria principal. En este trabajo, para resolver este problema, se implementa el LZ-index. Se verá su comportamiento y eficiencia, la que se comparará con los otros índices más comunes existentes en la literatura.

Capítulo 1

Definición del Problema

El problema de *Búsqueda Completa en Texto* tiene como entrada una secuencia de símbolos $T[1..u]$, el cual es el *texto*, sobre un alfabeto $\Sigma = \{1, \dots, \sigma\}$ de tamaño σ . Dada una secuencia $P[1..m]$, llamada *patrón de búsqueda*, se debe encontrar las *occ* ocurrencias de P en T .

El desafío principal es que, dado un texto T , se pueda acceder a las ocurrencias del patrón lo más rápido posible. Este problema se vuelve interesante cuando el tamaño de T es lo suficientemente grande, ya que una búsqueda secuencial por todo el texto deja de ser una alternativa viable, y se debe implementar un algoritmo como el que se verá en esta memoria, el cual se basa en una estructura de datos construida sobre el texto.

1.1. Objetivos

El objetivo principal de esta memoria consiste en rediseñar una estructura de datos para la búsqueda en texto tal que utilice el menor espacio posible y sea posible buscar patrones de manera rápida. Los objetivos específicos que se buscan en este trabajo son:

1. Rediseñar la estructura de datos LZ-index, de manera tal que pueda agregarse a librerías modernas como SDSL.

2. Agregar flexibilidad al índice, tal que permita ser parametrizado en aspectos importantes como tipo de alfabeto, y estructuras base que lo soportan.
3. Análisis comparativo entre espacio utilizado de nuevo índice y LZ-index original de Navarro [Nav08].
4. Análisis comparativo de tiempo de respuesta de nuevo índice y LZ-index original de Navarro [Nav08].

1.2. Formas de Solucionar el Problema

Como se menciona anteriormente, este problema se puede solucionar con distintos algoritmos y cada uno tiene sus ventajas y desventajas. Las diferencias principales radican en: dificultad de implementación, tiempo de respuesta y espacio en memoria utilizado.

En esta sección se verán dos algoritmos principales los cuales son: *Búsqueda de Texto Secuencial* y *Búsqueda de Texto Indexado*. El algoritmo implementado en esta memoria se encuentra dentro del segundo tipo. La solución de indexación implementada en este trabajo, no es la única que existe en la literatura, por lo que también se describirá los competidores más usados.

Búsqueda de Texto Secuencial: Para este tipo de solución se asume que no se realiza procesamiento previo al texto, por lo que dado un patrón P , se realiza una búsqueda secuencial por todos los símbolos del texto T , encontrando todas las ocurrencias. Si se considera que el largo del patrón de búsqueda es m y el texto es de largo n , la posición inicial de P se podría encontrar en cualquier posición desde $1, \dots, n - m + 1$. La complejidad en un peor caso es de $O(nm)$, considerando el algoritmo como fuerza bruta.

Éste tipo de algoritmo para la mayoría de los casos es insatisfactorio, debido a que si se tiene un T lo suficientemente largo, el algoritmo podría tomar un tiempo considerable en obtener todas las ocurrencias. Éste tipo de solución se utiliza mayormente para cuando el largo del texto es pequeño, y tanto el texto como el patrón de búsqueda son conocidos al mismo tiempo. Por ejemplo, el comando de Unix llamado **grep** utiliza este tipo de algoritmo.

La Búsqueda Secuencial dada su complejidad $O(mn)$ y la forma de solucionarse está dentro de la categoría de algoritmos de *Fuerza Bruta*. El algoritmo de **Boyer-Moore** [MS77] aplica una mejora, el cual busca las ocurrencias de P en T realizando comparaciones explícitas de los caracteres en distintos índices del texto. Esto permite al algoritmo usar la información preprocesada de P para saltar la mayor cantidad de símbolos posibles, esto dado que identifica previamente que en cierto lugar del texto no está el patrón buscado. La complejidad en tiempo de éste algoritmo es $O(n + m)$ en el peor caso.

Búsqueda de Texto Indexada: A diferencia de la forma anterior, en este tipo de búsqueda se construye una estructura de datos o *índice* en el texto para restringir la búsqueda a una porción más pequeña del texto. Esto se realiza para evitar la búsqueda secuencial y así obtener una menor complejidad de tiempo. Como se puede suponer, el tiempo para resolver el problema es menor, sin embargo, se utiliza una mayor cantidad de espacio ya que se debe construir una estructura de datos para almacenar el texto de manera que se pueda recorrer para encontrar las palabras y patrones. Este tipo de búsqueda se utiliza principalmente cuando el largo del texto es lo suficientemente grande para que una búsqueda secuencial sea muy costosa. También se utiliza cuando el texto no cambia frecuentemente (si lo hiciera sería costoso volver a construir nuevamente la estructura de datos). Esta búsqueda supone que se cuenta con el espacio suficiente para indexar el texto y también para realizar las búsquedas a lo largo de este índice.

Esta memoria se concentra en la Búsqueda de Texto Indexada, ya que se busca solucionar el problema en textos de tamaño grandes. Es por esto que en la próxima sección se detallará las distintas maneras de realizar búsqueda de texto indexada y las diferencias con la utilizada en esta memoria **LZ-index**.

1.3. Búsquedas de Texto Indexada Clásicas

En esta sección se revisan en más detalle las principales tendencias de la Búsqueda de Texto Indexada distintas a la desarrollada en esta memoria. Existen principalmente dos, las cuales son *suffix trees* y *suffix arrays*. Para estas se considera que existe un texto T sobre un alfabeto

Σ , y se añade un símbolo extra que marca el fin del texto: $\$ \notin \Sigma$.

1.3.1. Suffix Trees

Los *suffix trees* [McC76] son de los más usados en la literatura, los que permiten buscar un patrón P dentro de un texto T . Estos fueron introducidos en [Wei73].

La forma de operar de este algoritmo es sencillo, donde se crea un árbol con todos los sufijos posibles del texto. En la Figura 1.1 se puede ver el ejemplo de crear un *suffix tree* con el texto “mississippi”. Para buscar las ocurrencias de algún patrón P en este árbol se debe descender por el árbol por los símbolos de P hasta que:

1. Se encuentra un nodo v el cual represente el patrón P .
2. Se llega a una hoja sin encontrar P .

Cuando se encuentra un nodo hoja (último nodo por el cual no se puede seguir descendiendo), se sabe que P no ha sido encontrado en T . Para el primer caso, donde si se encuentra P , toma un tiempo de $\Theta(m)$, ya que debe buscar exactamente los m símbolos de P .

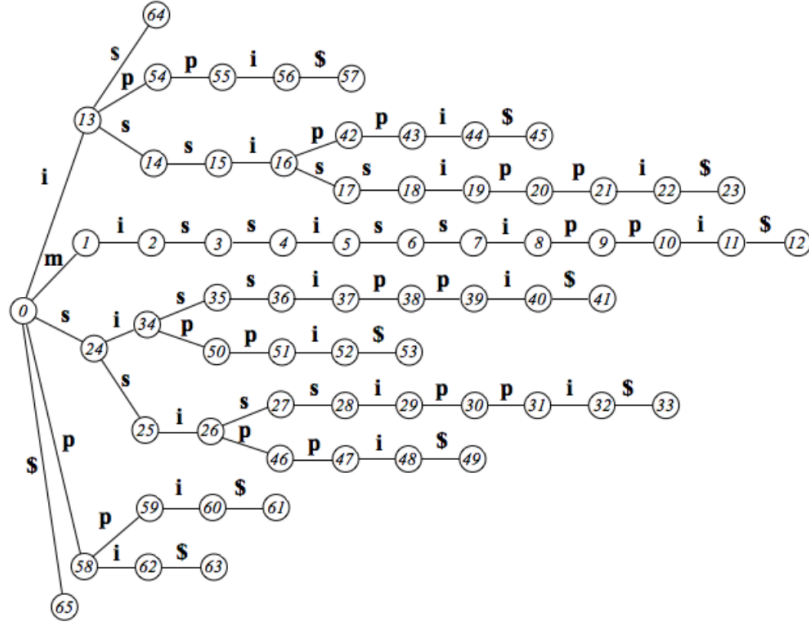


Figura 1.1: Suffix Tree con texto $T = \text{"mississippi"}$

Para contar la cantidad de ocurrencias de P , se deben contar la cantidad de hojas que contenga el subárbol de nodo v . Esto se puede realizar en tiempo constante si se guarda el tamaño del subárbol de cada nodo. Dado que el *suffix tree* tiene occ hojas, el subárbol tiene $O(occ)$ nodos, por lo que el tiempo de búsqueda es óptimo $O(m + occ)$.

1.3.2. Suffix Arrays

Los *suffix arrays* [MM93] son una alternativa de los *suffix trees*, donde se diferencian a que en la práctica utilizan desde tres a cinco veces menos espacio. Su manera de funcionar no es compleja y lo que hace en términos generales es listar ordenadamente todos los sufijos de un texto $T[1..u]$ en una lista. Por lo que para buscar un patrón $P[1..m]$ solo se debe buscar dentro de esta lista. Para no tener que hacer una búsqueda secuencial en esta lista se aplica el algoritmo "*longest common prefixes (lcp)*" el cual compara entre elementos adyacentes de la lista y retorna el prefijo más largo común entre estos, lo que resulta en que las búsquedas en esta lista se pueden realizar en tiempo $O(m \log u)$ realizando búsqueda binaria.

Ejemplo 1.3.1 Este ejemplo es rescatado desde [Bil09], donde el texto de entrada es:

T = alabar_a_la_alabarda_para_apalabrarla

En el texto por simplicidad se ha remplazado los espacios en blanco por “_”. Este texto genera el arreglo que se puede ver en la Figura 1.2. Si se busca el patrón ala, este se obtiene en los índices de intervalo [6.,8]. Estos están marcados en la Figura 1.2.

i	$SA_T[i]$	suffix (cyclically shifted)
1	38	\$alabar a la alabarda para apalabrarla
2	37	a\$alabar a la alabarda para apalabrarla
3	15	abarda para apalabrarla\$alabar a la al
4	3	abar a la alabarda para apalabrarla\$al
5	31	abrarla\$alabar a la alabarda para apal
6	13	alabarda para apalabrarla\$alabar a la
7	1	alabar a la alabarda para apalabrarla\$
8	29	alabrarla\$alabar a la alabarda para ap
9	27	apalabrarla\$alabar a la alabarda para
10	23	ara apalabrarla\$alabar a la alabarda p
11	17	arda para apalabrarla\$alabar a la alab
12	34	arla\$alabar a la alabarda para apalabr
13	5	ar a la alabarda para apalabrarla\$alab
14	11	a alabarda para apalabrarla\$alabar a l
15	25	a apalabrarla\$alabar a la alabarda par
16	8	a la alabarda para apalabrarla\$alabar
17	20	a para apalabrarla\$alabar a la alabard
18	16	barda para apalabrarla\$alabar a la ala
19	4	bar a la alabarda para apalabrarla\$ala
20	32	brarla\$alabar a la alabarda para apala
21	19	da para apalabrarla\$alabar a la alabar
22	36	la\$alabar a la alabarda para apalabrar
23	14	labarda para apalabrarla\$alabar a la a
24	2	labar a la alabarda para apalabrarla\$a
25	30	labrarla\$alabar a la alabarda para apa
26	10	la alabarda para apalabrarla\$alabar a
27	28	palabrarla\$alabar a la alabarda para a
28	22	para apalabrarla\$alabar a la alabarda
29	33	rarla\$alabar a la alabarda para apalab
30	24	ra apalabrarla\$alabar a la alabarda pa
31	18	rda para apalabrarla\$alabar a la alaba
32	35	rla\$alabar a la alabarda para apalabra
33	6	r a la alabarda para apalabrarla\$alaba
34	12	alabarda para apalabrarla\$alabar a la
35	26	apalabrarla\$alabar a la alabarda para
36	7	a la alabarda para apalabrarla\$alabar
37	9	la alabarda para apalabrarla\$alabar a
38	21	para apalabrarla\$alabar a la alabarda

Figura 1.2: Suffix Array para el Ejemplo 1.3.1, el sufijo es el texto sombreado en cada línea. La columna $SA_T[i]$ guarda la posición del texto inicial para el i -ésimo sufijo en el orden lexicográfico

1.4. Estructuras de Datos

Antes de comenzar explicando la solución particular propuesta para este problema, es necesario realizar un recorrido sobre las estructuras de datos utilizadas en esta memoria como también sus propiedades.

Se comienza explicando algunas propiedades de los Strings que son gran base de esta memoria al trabajarse con textos.

1.4.1. Representación de Strings

Se define un string como un arreglo $S[1..\ell]$ sobre un alfabeto $\Sigma = \{1, \dots, \sigma\}$ el cual es una secuencia de símbolos. Se tiene que $|S| = \ell$ es el largo del string S , y s_i corresponde al i -ésimo símbolo de S .

Además, dado un string $S = s_1, \dots, s_\ell$, se tiene que:

1. $S[i..j]$, para $1 \leq i \leq j \leq \ell$, es cualquier substring de S .
2. $S[1..i]$, para $1 \leq i \leq \ell$, es un *prefijo* de S . Si $1 \leq i < \ell$, entonces $S[1..i]$ es un *prefijo propio* de S .
3. $S[j..\ell]$, para $1 \leq j \leq \ell$, es un *sufijo* de S . Si $1 \leq j < \ell$, entonces $S[j..\ell]$ es un *sufijo propio* de S .
4. $S^r = s_\ell, \dots, s_1$ es un string inverso. Por lo tanto, $S^r[i..j]$ es lo mismo que decir $(S[i..j])^r = s_j, \dots, s_i$

Dado que cada símbolo s_i es representado utilizando $\log \sigma$ bits, un string S requiere $\ell \log \sigma$ bits para ser representado. Se utiliza el signo ϵ para representar el *string vacío*, el cual es un string que no contiene ningún símbolo.

1.4.2. Representando un Conjunto de Strings con Tries

Dado un conjunto de strings $\mathcal{S}$, un *digital tree*, *cardinal tree* o simplemente un *trie* [Knu73] de $\mathcal{S}$ es simplemente un árbol donde cada string de $\mathcal{S}$ es representado como un nodo en el árbol. Sea v el nodo que representa el string $S_i \in \mathcal{S}$. Esto significa que los nodos desde la raíz hasta v corresponde al string S_i . Notar que todos los nodos que descienden desde v en el trie representan strings que tiene a S_i como prefijo. La raíz representa el string vacío ϵ

Ejemplo 1.4.1 Consideremos nuevamente el ejemplo de [Bil09]. Donde se tiene un conjunto de strings $\mathcal{S}_1 = \{a, l, ab, ar, -, a_, la, _a, lab, ard, a_p, ara, _ap, al, abr, arl, a\$ \}$. El trie generado por esta representación es el mostrado en la Figura 1.3. Otro ejemplo es el que se muestra en la Figura 2.3b, donde su conjunto de strings es:

$\mathcal{S}_2 = \{a, l, ba, ra, -, _a, al, a_, bal, dra, p_a, ara, pa_, la, rba, lra, \$a \}$. En ambos casos los nodos mostrados en celeste corresponde a la conformación de un string, mientras que los nodos negros no representan un string como es en el caso de la Figura 2.3b, estos son llamados *nodos vacíos*.

Una representación trie de $\mathcal{S}$ es un *prefijo cerrado* de $\mathcal{S}$ si cualquier prefijo del conjunto de strings de $\mathcal{S}$ se puede encontrar en algún nodo del árbol. Más formalmente, se dice que un conjunto de $\mathcal{S}$ es *prefijo cerrado* cuando para todo string $S_i \in \mathcal{S}$, y cualquier prefijo de S_i también está contenido en $\mathcal{S}$. De la misma manera, se dice que un conjunto de strings de $\mathcal{S}$ es *sufijo cerrado* cuando para todo string de $S_i \in \mathcal{S}$, y cualquier sufijo de S_i también está contenido en $\mathcal{S}$.

Ejemplo 1.4.2 El conjunto $\mathcal{S}_1$ definido el Ejemplo 1.4.1 en *prefijo cerrado*, mientras que $\mathcal{S}_2$ no lo es. Se puede ver la diferencia en los tries generados, donde $\mathcal{S}_2$ tiene nodos vacíos, mientras que $\mathcal{S}_1$ no los tiene.

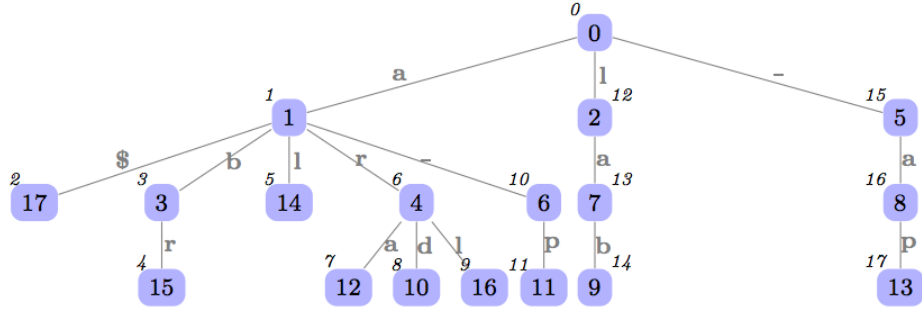


Figura 1.3: Representación Trie del conjunto de strings de S_1

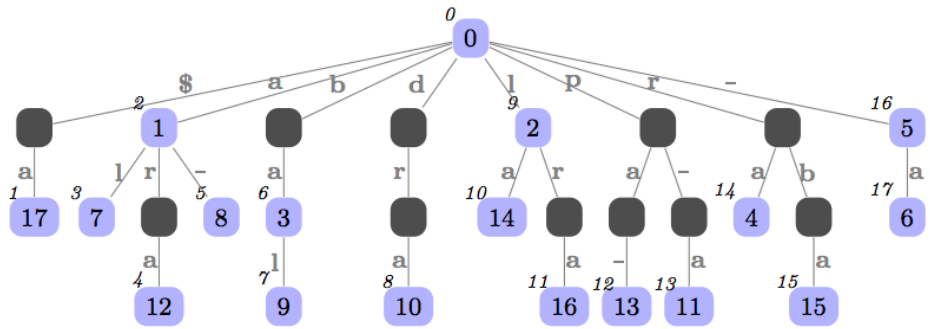


Figura 1.4: Representación Trie del conjunto de strings de S_2

Si se desea buscar cierto patrón P , se debe buscar los símbolos de P e ir bajando desde la raíz hasta el nodo donde se hayan encontrado todos los símbolos de P . El procedimiento mencionado se refiere a búsqueda de prefijo, que es el método que se utiliza en esta memoria para resolver el problema de búsqueda de texto. A continuación se da una definición más formal sobre el problema.

Definición 1.4.1 Dado un conjunto strings $\mathcal{S} = \{S_1, \dots, S_N\}$, el problema de búsqueda de prefijo consiste en encontrar todos los strings $S_i \in \mathcal{S}$, tal que S_i tenga un string $P[1..m]$ como prefijo.

A continuación se detallan formatos tradicionales para representar los tries, asumiendo que la cantidad de nodos es n :

1. La primera consiste en guardar, para cada nodo, un arreglo de tamaño σ con punteros a los hijos (el i -ésimo puntero apunta al hijo correspondiente a la i -ésimo símbolo del alfabeto o a *nulo* si es hijo no existe). Esto permite bajar por cualquier hijo en tiempo constante. Sin embargo, el espacio requerido para esta representación es $O(\sigma n \log n)$ bits. Esto puede ser excesivo en los casos donde el número promedio de hijos de un nodo es mucho menor a σ , lo cual es usual en la práctica.
2. Otra alternativa es usar una *linked list* en cada nodo, guardando los punteros a los hijos (con el correspondiente símbolo por el cual cada nodo desciende). Esto toma un espacio de $O(n \log n)$ bits y encontrar un hijo en el peor caso toma un tiempo de $O(\sigma)$. En vez de esto, también se puede almacenar los punteros en un arreglo, requiriendo asintóticamente el mismo espacio, pero encontrando a un hijo en un tiempo mejor en el peor caso de $O(\log \sigma)$ (búsqueda binaria entre los símbolos). Sin embargo, en este escenario, agregar un nuevo hijo sería costoso. Finalmente, se puede usar un árbol balanceado de búsqueda (ordenado por los símbolos), donde la búsqueda de cualquier hijo se podría realizar en un tiempo de $O(\log \sigma)$, soportando al mismo tiempo eficientes actualizaciones en el trie.
3. La última opción presentada consiste en utilizar hashing para almacenar el hijo de un nodo, obteniendo un tiempo en el peor caso de $O(1)$ para encontrar un hijo, y requiriendo un espacio de $O(n \log n)$ bits [Ram96]. La construcción toma un tiempo de $O(n)$ si se utiliza la forma clásica de Fredman [MFS84, TCS01]. En el peor caso, se tiene que n vértices del trie distribuidos por $\Theta(n/\sigma)$ nodos, por lo que estos nodos tienen $\Theta(\sigma)$ hijos. Así el costo es $O(\sigma^2)$ por nodo en el peor caso, y en total queda un costo de $O(\sigma n)$ cuando se agregan esos $\Theta(n/\sigma)$ nodos.

1.4.3. Propiedades de los Stings y Tries

Dado un alfabeto ordenado Σ , el orden lexicográfico de Σ^* es definido por $x \leq y$ si y solo si x es un prefijo del string y o $x = uav$ y $y = ubw$, con $a, b \in \Sigma, a < b$ y $u, v, w \in \Sigma^*$. De esta manera, un conjunto de strings $\mathcal{S}$, puede estar lexicográficamente ordenado. Para cualquier string $x, y \in \Sigma^*$ y símbolos $a, b \in \Sigma$ se cumplen las siguientes propiedades:

1. $\epsilon \leq x$
2. $ax \leq by$ si $a < b$
3. $ax \leq ay$ si $x \leq y$

Si un conjunto de strings es almacenado en un trie, entonces todos los strings representados en un subárbol forman un intervalo lexicográfico del universo. Si se numeran los nodos no vacíos del trie en preorden, entonces el subárbol de un nodo forma un intervalo en preorden.

Ejemplo 1.4.3 En el trie de la Figura 1.3 se muestran los números en preorden (rank lexicográfico de los strings correspondientes) afuera de cada nodo (solo se cuentan los nodos no vacíos). Por ejemplo, el intervalo en preorden $[1.,11]$ corresponde a todos los strings del conjunto de $\mathcal{S}_1$ que comienzan con la letra ‘a’, y el intervalo en preorden de $[6.,9]$ corresponde a todos los strings de $\mathcal{S}_1$ que comienzan con ‘ar’.

1.5. Compresión de Texto

Como ya se ha visto, uno de los principales desafíos es el espacio que utilizan las técnicas de Búsqueda de Texto Indexada, esto ya que debe lograr crear un índice que probablemente ocupe mayor espacio que un texto que ya es suficientemente grande. Es por esto que la *Compresión de Texto* es una técnica útil para representar un texto usando menos espacio. En [TBW90], se señalan dos principales ventajas:

1. Reduce el espacio requerido para almacenar el texto.
2. Incrementa la velocidad de transmisión en el texto. Ya sea desde textos que se encuentran en la red o memoria secundaria a memoria principal. Está comprobado que un texto comprimido es leído/transferido más rápido que uno no comprimido.

La desventaja principal para comprimir un texto es que el tiempo de procesamiento se ve incrementado, dado que se debe descomprimir (una parte de) el texto para poder procesarlo.

Sin embargo, la cantidad de veces que se accede a memoria secundaria son menores, por lo que existe una compensación entre utilizar una mayor cantidad de CPU por menos I/O, lo cual es ventajoso.

La definición formal de la Compresión de Texto, es que el k -ésimo orden de la entropía empírica de la secuencia de un texto T sobre un alfabeto Σ , se denota como: $H_k(T)$ [Man01]. Este valor $uH_k(T)$ es el límite inferior de cantidad de bits necesarios para comprimir un texto T usando cualquier compresor que puede codificar cada símbolo considerando solamente el contexto de los k símbolos que vienen de T .

Definición 1.5.1 Dado un texto $T[1..u]$ sobre un alfabeto Σ , el *orden-cero entropía empírica* de T se define como:

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

Donde un n_c es la cantidad de ocurrencias del símbolo c en T . La sumatoria incluye solo aquellos símbolos c que ocurren en T , por lo que $n_c > 0$.

Definición 1.5.2 Dado un texto $T[1..u]$ sobre un alfabeto Σ , el k -ésimo *orden de la entropía empírica* de T se define como:

$$H_k(T) = \sum_{S \in \Sigma^k} \frac{|T^S|}{u} H_0(T^S)$$

Donde T^S es la subsecuencia de T formado por todos los símbolos que vienen de S .

Propiedad 1.5.1 Dado un texto T sobre un alfabeto de tamaño σ , se tiene que:

$$0 \leq H_k(T) \leq H_{k-1}(T) \leq \dots \leq H_0(T) \leq \log \sigma$$

Para cualquier $k \geq 0$. Por lo tanto esto significa que entre mayor sea el contexto, mayor es la compresión.

Es necesario también recordar la propiedad acerca de los límites en la entropía de orden cero de un string binario.

Propiedad 1.5.2 Sea $\mathcal{B}[1..u]$ un string binario de largo u , con $n \leq u$ bits. Entonces:

1. $\log \binom{u}{n} \leq uH_0(\mathcal{B}) \leq \binom{u}{n} + O(\log n)$
2. $n \log \frac{u}{n} \leq uH_0(\mathcal{B}) \leq n \log \frac{u}{n} + n \log e$

En la literatura existen dos grandes tipos de compresión de texto que permiten trabajar con el texto indexado: *Burrows-Wheeler Transform* [BW94] y *Lempel-Ziv* [ZL78], el cual es el que se enfoca esta memoria.

1.5.1. Compresión Lempel-Ziv (LZ78)

La *Compresión Lempel-Ziv* es un algoritmo para comprimir que no genera pérdidas en la data. No es simplemente un algoritmo, sino corresponde a toda una familia de estos. Los algoritmos de Lempel Ziv fueron propuestos desde 1976 por Abraham Lempel y Jacob Ziv.

Este algoritmo aparece en una primera instancia en [LZ76], donde la idea principal es reemplazar substrings en un texto por un puntero a la ocurrencia previa de este. Se obtiene compresión cuando el puntero requiere menos espacio que el string que está reemplazando. Existen dos principales variantes que surgieron después, las cuales están detallados en [ZL77, ZL78]. Este trabajo utiliza el LZ78 dado que sus cualidades son de utilidad para luego construir el LZ-index y poder realizar la Búsqueda de Texto.

La compresión LZ78 esta basada en diccionario de frases, en donde se van agregando las *frases* a medida que se van leyendo en el texto. En un comienzo la compresión solo contiene una frase llamada b_0 de largo 0 (string vacío). El siguiente paso de la compresión es: Se asume que un prefijo $T[1..j]$ de T ha sido comprimido en una secuencia de *frases* $Z = b_1 \dots b_r$, todas estas se encuentran en el diccionario, luego se busca por el prefijo más largo del resto del texto $T[j + 1..u]$ el cual es un *frase* del diccionario. Una vez que esta *frase* ha sido encontrada, b_s de largo ℓ_s , se construye una nueva frase $b_{r+1} = (s, T[j + \ell_s + 1])$, se escribe este par al final del archivo comprimido Z , con lo que queda, $Z = b_1 \dots b_r b_{r+1}$ y se agrega la frase al diccionario.

Se denota B_i al string representado por la *frase* b_i , de este modo $B_{r+1} = B_r T[j + \ell + 1]$. Así el algoritmo LZ78 comprime el texto T en $n + 1$ *frases* $B_0, \dots, B_n$. Por lo tanto, $T = B_0, \dots, B_n$, y $B_0 = \epsilon$ que representa el string vacío. Se dice que i es el *identificador* de la *frase* correspondiente a B_i , para todo $0 \leq i \leq n$.

Propiedad 1.5.3 Para todo $1 \leq t \leq n$, existe $\ell < t$ y $c \in \Sigma$, tal que $B_t = B_\ell \cdot c$.

Esto quiere decir que cualquier frase B_t (excepto B_0) es formada por una previa frase B_ℓ mas un símbolo c al final. Esto implica que un conjunto de frases es *prefijo cerrado*, esto significa que cualquier prefijo de una frase de B_t es también una frase en el diccionario. Dado esto, una forma de representar este conjunto de frases es un *LZtrie* que es en la que se basa la implementación propuesta en esta memoria.

Propiedad 1.5.4 Toda frase B_i , $0 \leq i \leq n$, representa un substring diferente.

La única excepción a esta regla es en la última frase B_n . Es por esto que se añade al final del texto T un símbolo especial '\$' $\in \Sigma$, asumiendo que este es un menor que cualquier otro en el alfabeto. La última frase contiene este símbolo por lo que también es único.

Definición 1.5.3 Sea $b_r = (r_1, c_1)$, $b_{r_1} = (r_2, c_2)$, $b_{r_2} = (r_3, c_3)$, hasta $r_k = 0$ frases del LZ78 parseadas desde T . La secuencia de identificadores de frases $r_1, r_2, \dots$ es llamada la *cadena de referencia* comenzando desde la frase r .

Alternativas para Codificar la Descomposición del LZ78. Existen dos alternativas para codificar un texto T a través de la compresión de LZ78.

Variante 1. Las frases del LZ78 están representadas por dos arreglos, $S[1..n]$ de $n \log \sigma$ bits y $A[1..n]$ de $n \log n$ bits. Si se tiene que $B_i = B_j \cdot c$ es una frase de la descomposición del LZ78 de T , se representa la i -ésima frase guardando la referencia $A[i] \leftarrow j$ y el símbolo $S[i] \leftarrow c$. Si la frase está compuesta por solo un símbolo se guarda un 0 en la posición correspondiente de A . Por lo tanto, esta representación necesita de un espacio de $n(\log n + \log \sigma)$ bits. Se puede ver un ejemplo de esta representación en la Figura 1.6.

Para poder descomprimir un string B_i correspondiente al i -ésima frase del LZ78, se debe seguir las referencias en la cadena obteniendo los símbolos correspondientes hasta llegar al valor 0 del arreglo A . Este procedimiento toma tiempo $\Theta(|B_i|)$.

Se puede descomprimir el texto completo, recuperando $B_1, \dots, B_n$ en tiempo $O(u)$. Sin embargo, esta representación no permite una rápida búsqueda por el prefijo más largo de una frase que ya esté en el diccionario, por lo que no es eficiente para la construcción del LZ78.

Variante 2. Se representan el conjunto de frases del LZ78 con el *LZTrie*. Se añade además un arreglo $Node[1..n]$, el cual es un mapeo tal que $Node[i]$ guarda el nodo del *LZTrie* correspondiente a la frase B_i . El *LZTrie* requiere de un espacio de $n \log \sigma$ bits, para representar los símbolos del *Trie* (estructura de datos que se verá en detalle en la próxima Sección 1.4) más el espacio necesitado para la topología del trie lo que es $2n + o(n)$ si se considera una codificación sucinta como la ocupada en esta memoria (detalle en Sección 3.1). El espacio utilizado por $Node$ es $n \log n$ bits si se guardan los nodos en preorden. Por lo tanto la representación total de esta variante es: $n(\log n + \log \sigma + 2)$. Continuando con el ejemplo mostrado en [Bil09], en las Figuras 1.7 y 1.8 se puede ver el *LZTrie* y $Node$ respectivamente.

Para obtener cierta frase B_i , primero se debe acceder en la posición $Node[i]$ en el *LZTrie*, para luego extraer todos los símbolos en el camino hasta la raíz, siguiendo todos los nodos padres. Por ejemplo, para extraer la frase *abr* que tiene *id* 15, se debe buscar primero su preorden del *LZTrie* en $Node[15]$, con lo que se obtiene 4. Este puntero nos indica que debemos posicionarnos en el cuarto nodo de en preorden del *LZTrie* y seguir los nodos padres hasta la raíz, con lo que se obtiene los símbolos *r*, *b*, *a*, lo que era la frase que se buscaba *abr*. Como se verá en la Sección 3.1, este procedimiento se puede realizar en tiempo constante con un árbol succinto y utilizando poco espacio, por lo que este método toma un tiempo de $O(|B_i|)$.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
a	l	ab	ar	_	a_	la	_a	lab	ard	a_p	ara	_ap	al	abr	arl	a\$

Figura 1.5: Descomposición por frases y con su respectivo *id* del LZ78 para el ejemplo visto en [Bil09]. $T = \text{alabar_a_la_alabarda_para_apalabrarla}$

phrase id.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
A	0	0	1	1	0	1	2	5	7	4	4	4	8	1	3	4	1
S	a	l	b	r	-	-	a	a	b	d	l	a	p	l	r	l	\$

Figura 1.6: Representación en formato de arreglo para la descomposición del LZ78 del ejemplo de 1.5.

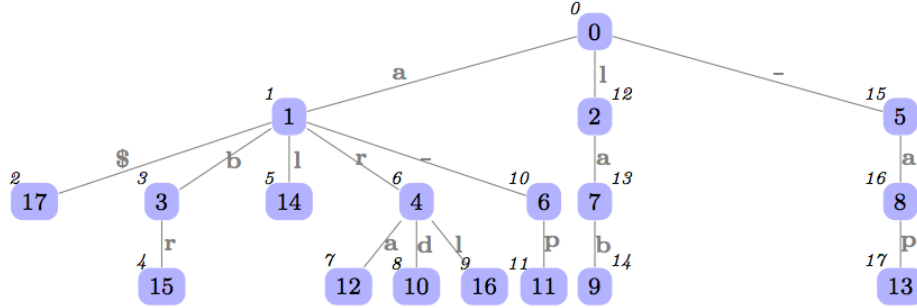


Figura 1.7: Representación *LZTrie* para la descomposición del LZ78 del ejemplo en Figura 1.5, los números de afuera de los nodos representan el recorrido en preorden y los números de los nodos representan el *id* de la frase.

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
$Node[i]$	①	⑫	③	⑥	⑮	⑩	⑬	⑮	⑭	⑧	⑪	⑦	⑰	⑤	④	⑨	②

Figura 1.8: Arreglo *Node* para el ejemplo de Figura 1.5, donde la notación $\textcircled{k}$ representa un puntero al nodo del *LZtrie* (Figura 1.7) con preorden k

Resumiendo, estas dos últimas variantes requieren un espacio de $n(\log n + \log \sigma)$ bits para representar la descomposición del algoritmo del LZ78. En esta memoria se utiliza la *Variante 2*, dado que el *LZTrie* permite buscar rápidamente en el texto comprimido.

Propiedades de la descomposición LZ78.

Propiedad 1.5.5 [ZL78] Se cumple que $\sqrt{u} \leq n \leq \frac{u}{\log_{\sigma} u}$. Por lo tanto, $\log n = \Theta(\log u)$ y siempre se cumple que $n \log u \leq u \log \sigma$.

Para los strings de $T[1..u]$ generado desde una *fente ergódica estacionaria* (modelo donde se representa un string indefinidamente largo que representa todo el texto [IWB99]), se tiene que el ratio conseguido por la descomposición del LZ78: $\frac{n \log n}{u}$ tiende a H , la cual es la entropía mostrada anteriormente en la Sección 1.5. Esto se obtiene gracias a que la representación del LZ78 usa un espacio cercano a $n(\log n + \log \sigma)$ bits, el ratio de compresión del LZ78 es asintóticamente óptima, esto dado a que el término $n \log n$ domina (es mayor).

De este modo, el LZ78 es capaz de comprimir un string indefinidamente largo con un espacio óptimo, de acuerdo al modelo de entropía. Esto diferencia al LZ78 de otros muchas técnicas de compresión. Sin embargo, esto se cumple cuando $u \rightarrow \infty$. Además el LZ78 converge al óptimo lentamente a medida que el largo del texto aumenta. Aún así, el rendimiento del LZ78 es muy bueno en la práctica.

En esta memoria se utiliza el resultado descubierto por Kosarju y Manzini [KM99] para limitar el resultado de la descomposición del texto del LZ78 en término del k -ésimo orden empírico de T . El espacio generado de la ‘o grande’ extra refleja los textos finitos, lo cual genera la convergencia asintótica lenta.

Lema 1.5.1 [KM99]. *Se cumple que $n \log n = uH_k(T) + O(u \frac{1+k \log \sigma}{\log_\sigma u})$*

En este trabajo se asume que $k = o(\log_\sigma u)$ (por lo tanto $\log \sigma = \sigma(\log u)$ para permitir $k > 0$). Así, $n \log n = uH_k(T) + o(u \log \sigma)$

Usando la Propiedad 1.5.4, se puede probar lo siguiente:

Lema 1.5.2 *Sea $T[1..u]$ el texto sobre un alfabeto de tamaño σ , y sea n la cantidad de frases en la descomposición del LZ78 de T . Sea $\mathcal{B}[1..u]$ un string binario de largo u y n bits. Entonces, se tiene que $nH_0(\mathcal{B}) = o(u \log \sigma)$.*

Demostración. Utilizando la Propiedad 1.5.4, donde se tiene que $uH_o(\mathcal{B}) \leq n \log \frac{u}{n} + n \log e$. Dado que $\log \frac{u}{n}$ crece con n y dado que $n \leq \frac{u}{\log_\sigma u}$ (Propiedad 1.5.5), remplazando n por $\frac{u}{\log_\sigma u}$, queda que: $n \log \frac{u}{n} + n \log e \leq \frac{u}{\log_\sigma u} (\log \log_\sigma u + \log e)$, lo cual es $o(u \log \sigma)$.

Lema 1.5.3 *Se cumple que $n \log u = uH_k(T) + o(u \log \sigma)$ para cualquier $k = o(\log_\sigma u)$.*

Demostración. Hasta ahora se tiene que $n \log u = n \log n + n \log \frac{u}{n}$. Esto dado a lo visto en la demostración del Lema 1.5.2 que se tiene: $n \log \frac{u}{n} = o(u \log \sigma)$. Por otro lado, gracias al Lema 1.5.1 se tiene que $n \log n = uH_k(T) + o(u \log \sigma)$ para cualquier $k = o(\log_\sigma u)$. Por lo tanto, se tiene que $n \log u = uH_k(T) + o(u \log \sigma)$ para cualquier $k = o(\log_\sigma u)$.

1.6. Análisis Inicial de la Solución LZ-index

El problema principal de los autoíndices comprimidos es representar un texto y buscar o contar ocurrencias de patrones dentro de este texto utilizando poco espacio. Sin embargo, existen varias otras funcionalidades que un autoíndice comprimido debe cumplir para que sea realmente útil. Muchas de estas han sido obtenidas desde distintos índices de búsqueda, por ejemplo existen:

- Los índices como los suffix tree, permiten buscar un patrón en tiempo óptimo $O(m + occ)$, requiriendo un espacio de $O(u \log u)$ bits. Al igual que el LZ-index de Ferragina y Manzini [FM05] logrando el mismo tiempo óptimo de búsqueda $O(m + occ)$ y requiriendo un espacio de $o(u \log u)$ bits cuando σ es pequeño.
- Los suffix arrays, los cuales son los más usados en la práctica, pueden ser construidos en un tiempo de $O(u)$ [MM93], y requieren un espacio de $\Theta(u \log u)$ bits, lo cual puede ser excesivo para textos suficientemente largos.
- índices comprimidos que usan poco espacio proporcional a la entropía del texto (a veces incluido el texto) [NM07].
- índices que requieren poco espacio para ser construidos [KSY07, WHS03, NP07, GN08]: Los índices comprimidos son usualmente derivados de un formato clásico. Usualmente es simple construir un índice clásico y luego realizar su versión comprimida, pero puede pasar que no exista suficiente espacio para construirlo, si es así, se construye en memoria secundaria. Sin embargo, muchos de los índices clásicos son

costosos para construir de esta manera. Es por esto, que un gran desafío es construir el índice comprimido sin construir la versión clásica antes.

- Algunos índices que responden de buena manera al ser contruidos en memoria secundaria: [FG99, CM99] A pesar de que no necesiten de mucho espacio, hay veces que es necesario utilizar memoria secundaria.

Algunos de los índices cumplen con varios de los puntos mencionados como [GN08]. Sin embargo, ninguna estructura de datos existente cumple con todos los requisitos anteriores.

Esta memoria está centrada en la solución basada en el índice de Lempel Ziv (LZ-index), esto dado a que en la práctica y teóricamente se ha comprobado que son efectivos en varios aspectos: son rápidos encontrando occurencias de los patrones, mostrando el contexto donde están estas ocurrencias, como también extrayendo el texto buscado [Nav04, Nav08]. Sin embargo, este índice aún puede obtener mejoras las cuales se experimentarán y explicarán en esta memoria la cual continua el trabajo realizado por [Bil09] la que está basada en el algoritmo de Lempel Ziv de 1978 [ZL78].

Como ya se ha visto en la literatura mencionada, el LZ-index de Navarro es de los últimos vistos en el estado del arte, el que fue continuado por Arroyuelo en [Bil09] que son los principales trabajos en los que se basa esta memoria. Es por esto, que a continuación se detallan las características principales del LZ-index de Navarro:

- Una búsqueda completa del texto rápida.
- Recuperación rápida del texto
- Utiliza poco espacio (esto comparado con otros LZ-index anteriores, dado que comparado con índices de otros tipos seguía utilizando bastante espacio).
- No permite inserciones o eliminaciones (no es dinámico).
- Solo trabaja en memoria principal.
- Necesita de mucho espacio para ser construido.

El objetivo de esta memoria es, a partir de la solución propuesta por Navarro en [Nav08], agregar nuevas características al LZ-index con la cual se pueda mejorar las características anteriormente señaladas. Esto desde un punto de vista práctico como también teórico. La finalidad es llegar a ser competitivo con otros índices comprimidos que son más utilizados hoy en día como los suffix arrays.

A continuación un resumen comparativo de los distintos índices:

Cuadro 1.1: Comparación de distintos métodos de indexación.

Indice	espacio indexación (bits)	tiempo indexación
Suffix Arrays (in-place suffix sorting) [FM07]	$u \log u$	$O(u \log u)$
Suffix Arrays (optimal space) [WHS03]	$O(u \log \sigma)$ (*)	$O(u \log \log \sigma)$
Compressed Suffix Arrays [KSY07]	$u(H_0(T) + 2 + \epsilon) + o(u \log \sigma)$ (§)	$O(u \log u)$
Compressed Suffix Arrays [NP07]	$O(u \log \sigma \log_{\sigma}^{\log_3 2} u)$ (*)	$O(u)$
Alphabet Friendly FM-index [GN08]	$uH_k(T) + o(u \log \sigma)$ (◇)	$O(u \log u(1 + \frac{\log \sigma}{\log \log u}))$
LZ-index Original (esta memoria)	$4uH_k(T) + o(u \log \sigma)$	$O(u(\log \sigma + \log \log u))$

(*) Esto es $o(u \log u)$ bits, si $\log \sigma = o(\log u)$.

(§) Esto es $O(u \log \sigma)$ bits, en el peor caso.

(◇) Esto se cumple, para cualquier $k \leq \alpha \log_{\sigma} u$ y cualquier constante $0 < \alpha < 1$.

Capítulo 2

Estado del Arte

Como se explica en la sección anterior, este estudio se centra en la compresión basada en los algoritmos Lempel-Ziv (LZ), por lo que este estado del arte, detalla los distintos avances que se han logrado con respecto a este tema en el pasar de la historia, sus principales diferencias y cuales son las tendencias actuales.

La idea general de los algoritmos de compresión de LZ, donde se pueden ver en más detalle en [LZ76, ZL77, ZL78], y para recordar de secciones anteriores, la idea general es lograr descomponer un texto $T[1..u]$ en un conjunto de n frases $T = B_1...B_n$. Con lo que cada frase B_i está conformada por un substring de T , el cual aparece en el texto si se realiza una búsqueda secuencial. La única excepción en este modelo es que se debe agregar un símbolo '\$', que se asume que es menor a cualquier otro símbolo en el alfabeto del texto. La última frase contendrá este símbolo, lo que ayuda para saber cuando se termina el texto.

Este trabajo se concentra en LZ78, por lo que volvemos al ejemplo previamente trabajado de [Bil09] donde el texto es:

$T = \text{"alabar a la alabarda para apalabrarla"}$ y este queda descompuesto por el LZ78 como se puede apreciar en la Figura 2.1.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
a	l	ab	ar	_	a_	la	_a	lab	ard	a_p	ara	_ap	al	abr	arl	a\$

Figura 2.1: Descomposición LZ78 y sus correspondientes *phrase id* para un ejemplo compuesto de $T = \text{“alabar a la alabarda para apalabrarla”}$.

Recordar también que lo que se quiere lograr es luego de tener esta descomposición comprimida del LZ, se requiere buscar un patrón $P[1..m]$. La gran dificultad para buscar este patrón radica en que lo buscado se puede encontrar en más de una frase consecutiva de la descomposición realizada por el LZ. Se llaman *ocurrencias de tipo 1*, cuando se encuentra el patrón en solo una frase, *ocurrencias de tipo 2*, cuando se encuentra el patrón en dos frases consecutivas y *ocurrencias de tipo 3*, cuando se encuentran en 3 o más frases consecutivas.

Ejemplo 2.0.1 Una ocurrencia de tipo 1, sería buscar el patrón $P = \text{‘ab’}$, ya que se encuentra dentro de solo una frase con id 3. Mientras que si lo que se desea buscar es el patrón $P = \text{‘abar’}$, este es de ocurrencia tipo 2, dado que se encuentra en dos frases consecutivas con id 4 y 5. Por último, un ejemplo de ocurrencia de tipo 3 sería el patrón $P = \text{‘abar_a’}$, esto porque se encuentra en cuatro frases consecutivas con id 3,4,5 y 6.

Lo recién detallado se ha trabajado de distintas maneras a lo largo del estado del arte y a continuación se detalla los principales trabajos previos a este trabajo y como se maneja la búsqueda de texto en una compresión LZ generando un índice.

2.1. LZ-índice de Kärkkäinen y Ukkonen (KU-LZI)

El primero se trata del implementado por Kärkkäinen y Ukkonen [KU96, Kä99] (KU-LZI), este índice esta basado en la compresión Lempel Ziv de 1976 LZ76 [LZ76].

En la compresión de LZ76, el texto $T[1..u]$ se representa con una secuencia de frases $T = B_1, \dots, B_n$ y esta se construye de la siguiente manera: Se asume que el prefijo $T[1..j]$ de T ha sido comprimido en las secuencias de frases $B_1, \dots, B_r$. Luego, se busca por el prefijo más

largo $T[j + 1..j']$ del resto del texto $T[j + 1..u]$ lo que es lo mismo a $T[s..s + j - j + 1]$ de $T[1..j]$. Si $j' > j$, entonces $B_{r+1} = T[j + 1..j']$. Se sigue de la misma manera hasta completar todo el texto $T[j + 1..u]$. El substring $T[s..s + j - j + 1]$ es el origen para crear la frase B_{r+1} . De no ser así significa que el símbolo $T[j + 1]$ no ha aparecido previamente, por lo que $B_{r+1} = T[j + 1]$, y se sigue el proceso con $T[j + 2..u]$. La ejecución termina una vez que se obtiene $B_n = “\$”$

Una propiedad importante de esta descomposición, a diferencia de la vista en este trabajo que es la LZ78, es que toda frase ha aparecido previamente, a menos que sea un nuevo símbolo de Σ . Esto produce que la primera ocurrencia del patrón $P[1..m]$ en T no puede estar contenida completamente en una sola frase, ya que o si no hubiera aparecido anteriormente en T . Por lo tanto, en esta descomposición, las ocurrencias de tipo 1 son repeticiones de las ocurrencias de tipo 1 o 2. Kärkkäinen y Ukkonen llaman *ocurrencias primarias* a las de tipo 2 y *ocurrencias secundarias* a las de tipo 1. Las ocurrencias de tipo 3 son tratadas de una forma particular como si fueran de tipo 2.

Para mas detalle acerca de la implementación de este índice visitar [KU96, Kä99].

2.1.1. Solución de KU-LZI

La solución que logra KU-LZI es: Dado un texto $T[1..u]$, sobre un alfabeto σ , con una entropía empírica de $H_k T$ requiere un espacio de $O(uH_k(T)) + u \log \sigma + o(u \log \sigma)$ bits. Y dado un patrón de búsqueda $P[1..m]$, el índice es capaz de:

1. Ubicar la ocurrencia del patrón P en el texto T en un tiempo de $O(m^2 + (m + occ) \log u)$.
2. Contar el número de ocurrencias del patrón en un tiempo $O(m^2 + m \log u + occ)$.
3. Determinar si el patrón P existe en T en un tiempo de $O(m^2 + m \log u)$.
4. Extraer cualquier substring de largo ℓ en un tiempo de $O(\ell / \log_\sigma u)$ (el substring tiene que existir).

2.2. LZ-índice de Ferragina y Manzini (FM-LZI)

El LZ-índice de Ferragina y Manzinini (FM-LZI) [FM05] es un autoíndice comprimido basado en la combinación la descomposición de un texto en frases del LZ78 y la transformación Burrows-Wheeler [BW94]. El resultado es un eficiente autoíndice comprimido.

Las siguiente estructuras de datos son las utilizadas en este índice:

1. *LZtrie*: El ya conocido trie formado por la descomposición del LZ78 el que a partir de un texto T genera las frases $B_0, \dots, B_n$. Todo nodo del trie guarda la posición inicial del correspondiente texto de la frase del LZ78. Dado que la cantidad de nodos generados por el LZtrie es n , y además se utiliza una representación con punteros, el espacio requerido para esta estructura de datos es: $O(n \log n) = O(uH_k(T)) + o(u \log \sigma)$.
2. *FMI(T)*: El índice FM (FM-index) de un texto T , el cual está basado en la transformación Burrows-Wheeler y este requiere un espacio de $O(uH_k(T)) + o(u \log \sigma)$ bits. Por si solo este índice es capaz de realizar búsquedas por ocurrencias pero es más lento que índices basados en LZ para encontrar ocurrencias.
3. *FMI(T_#^r)*: El FM-index de un texto T , donde $T_{\#} = B_1\#\dots\#B_n\#$ es el string formado por añadir un símbolo especial '#' $\notin \Sigma$ después de todas las frases formadas por el LZ78. Por lo que el tamaño de $T_{\#}$ es $n + u$ símbolos. Utilizando esta estructura de datos se puede encontrar todas las frases que terminen con cierto substring. Los autores de este índice Ferragina y Manzini en [FM05] probaron que para cualquier $k \geq 0$ se tiene que $(u + n)H_k(T_{\#}^r) \leq uH_k(T) + o(u \log \sigma)$. Por lo tanto, *FMI(T_#^r)* requiere de un espacio de $O(uH_k(T)) + o(u \log \sigma)$ bits.
4. *N[1..n]*: Se trata de un arreglo, el que para cada posición de *FMI(T_#^r)* que indexa un sufijo de $T_{\#}^r$ empezando con $\#B_i^r$, guarda el correspondiente nodo del *LZtrie* para la frase B_i . Este arreglo requiere de un espacio de $n \log n = uH_k(T) + o(u \log \sigma)$ bits.
5. *Range*: Estructura de Datos en dos dimensiones que permite buscar en una grilla de $[1..u] \times [1..u]$. Esta grilla permite realizar búsquedas por rango en un tiempo de $O(m +$

occ), y requiere de un espacio de $O(n \log^{1+\gamma} n)$ bits. Por lo tanto para cualquier $\gamma > 0$ esto es $O(uH_k(T) \log^\gamma u) + o(u \log \sigma \log^\gamma u)$ bits.

Para encontrar ocurrencias de tipo 1 usando estas estructuras de datos, se comienza asumiendo que la frase B_t contiene a P . Si B_t no termina con P y si $B_t = B_\ell \cdot c$, con $\ell < t$ y $c \in \Sigma$, entonces B_ℓ también contiene a P . De esta manera, se debe buscar frases de menor tamaño que contengan a P . Por propiedad de la compresión LZ78, todas esas frases terminan con P . El resto de las frases que contienen a P pueden ser buscadas usando el *LZtrie*.

Para ocurrencias de tipo 2 y 3, se procede de forma similar al índice anterior de KU-LZI. Para toda posible partición $P[1..i]$ y $P[i+1..m]$ de P , se buscan todas las frases que terminan con $P[1..i]$ y todas las frases que contengan una ocurrencia de $P[i+1..m]$. En este índice se tiene la ventaja que se cuenta con más estructuras de datos para resolver el problema.

2.2.1. Solución de FM-LZI

La solución que logra *FM-LZI* es: Dado un texto $T[1..u]$, sobre un alfabeto σ con una entropía empírica de $H_k T$, este requiere un espacio de $O(uH_k(T) \log^\gamma u) + o(u \log \sigma \log^\gamma u)$ bits, para cualquier $\gamma > 0$ y cualquier $k = o(\log_\sigma u)$. Dado un patrón de búsqueda $P[1..m]$, este índice es capaz de:

1. Ubicar la occ ocurrencias del patrón P en el texto T en un tiempo de $O(m + occ)$.
2. Contar el número de ocurrencias del patrón en un tiempo $O(m)$.
3. Determinar si el patrón P existe en T en un tiempo de $O(m)$.
4. Extraer cualquier substring de largo ℓ en un tiempo de $O(\ell)$.

2.3. LZ-índice de Navarro (NAV-LZI)

El índice implementado por Navarro en [Nav04, Nav08] es la base que se toma para realizar este presente trabajo. Este usa un acercamiento diferente para definir el LZ-índice. Básicamente utiliza el *LZTrie* para las búsquedas y explota sus relaciones con las frases reversas del LZ78 e implementa el llamado *RevTrie*, el cuál es capaz de encontrar un patrón sin que sea el principio de una palabra. Dado que NAV-LZI ha demostrado ser efectivo en la práctica [Nav08], es el primer LZ-índice en ser implementado.

Como se puede notar el NAV-LZI no utiliza estructuras de datos como *suffix tree* o *arreglos*. Esto genera un problema el cual es que, como no existe ningún tipo de almacenamiento para los sufijos de las frases del LZ, no es posible encontrar ocurrencias de tipo 3, lo que si se podía realizar en KU-LZI y FM-LZI. Por lo tanto, en NAV-LZI solo funciona hasta ocurrencias de tipo 2, mientras que las ocurrencias de tipo 3 deben ser buscadas secuencialmente, por los $O(m^2)$ candidatos.

2.3.1. Componentes de NAV-LZI

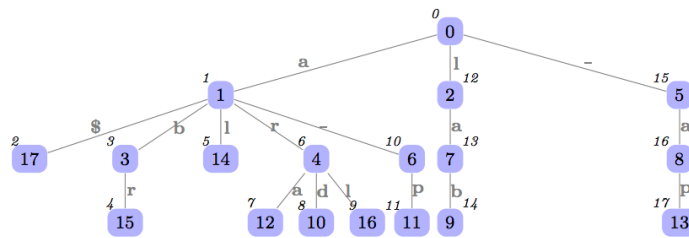
1. *LZTrie*: Árbol formado por las frases $B_0...B_n$. Dadas las propiedades del LZ78. Más información acerca de la construcción de un Trie a partir de las frases generadas por el LZ visitar Sección 1.4.2
2. *RevTrie*: Árbol formado por todas las frases en reverso $B_0^r...B_n^r$. En este árbol pueden haber nodos intermedios que no representan ni una frase. Estos son llamados nodos “vacíos”.
3. *Node*: Es un arreglo de n punteros a los nodos del *LZTrie*. Como veremos en el siguiente capítulo, el *LZTrie* esta implementado utilizando paréntesis balanceado, por lo que $Node[i]$ indica la posición de la secuencia para el abre paréntesis ‘(’ que representa el nodo de la frase i . Como existen $2n$ paréntesis, se necesita un espacio de $n \log 2n = n \log n + n$ bits para esta estructura.
4. *Range*: Es una estructura de datos en dos dimensiones que permite búsquedas por

rango en el espacio de $[0..n] \times [0..n]$. Se guardan los puntos:

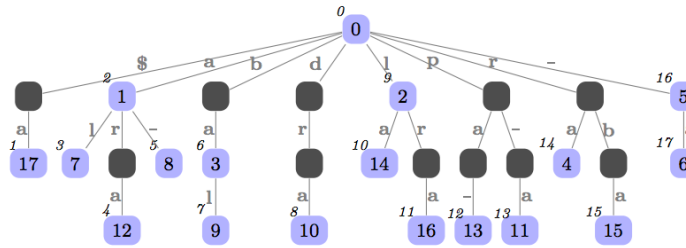
$$\{(preorder_r(t), preorder_{l_z}(t + 1)), t \in 0 \dots n - 1\}$$

Donde $preorder_r(t)$ es el *RevTrie* preorden del nodo de la frase t (se considera solo nodos no vacíos). Mientras que $preorder_{lz}(t+1)$ es el *LZTrie* preorden de los nodos de la frase $t+1$. Para cada uno de estos puntos, el valor correspondiente a t es guardado.

Continuando con la descomposición LZ vista en la Figura 2.1, ahora en la Figura 2.2 se puede apreciar como quedan compuestos los componentes *LZtrie*, *Revtrie* y *Range*.



(a) Lempel-Ziv Trie (*LZTrie*) para el ejemplo. Los números corresponden al recorrido en preorden.

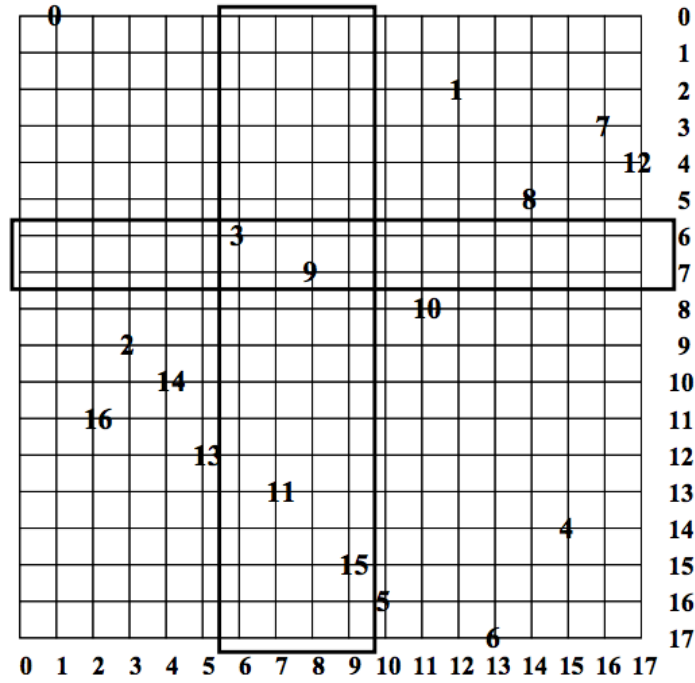


(b) Estructura de dato *RevTrie*. Los nodos vacíos corresponden a los marcados en grises.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39		
<i>par:</i>	(	(	)	(	(	)	(	)	(	)	)	(	(	)	)	(	(	)	(	)	(	)	)	(	(	)	(	)	)	(	(	)	(	)	)	(	(	)	(	)	)	)
<i>B:</i>	1	1		1	1		1	1		1	1		1	1		1	1		1	1		1	0	1		1		0	1		1		1	1		1	1		1	1		
<i>rltts:</i>	\$		a	l		r		-		b	l		d		l	a		r					p	a		-			r	a		b		-		a						
<i>rids:</i>	0	17		1	7		12		8			3	9			10		2	14		16				13		11			4		15				5	6					

(c) Representación de paréntesis balanceados del *RevTrie*, con los nodos vacíos comprimidos. El bitmap B marca con 0 los nodos vacíos no unarios. Arreglo *rlets* guarda el primer símbolo de cada vértice. Arreglo *rids* son los ids de esta estructura de datos.

Figura 2.2: Descomposición del LZ78, con el respectivo *LZTrie*, *RevTrie*.



(a) Estructura de Datos *Range*. Eje horizontal es la numeración preorden del *LZTrie*. Eje vertical es la numeración preorden del *RevTrie*.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
$Node[i]$	0	1	23	4	10	29	18	24	30	25	13	19	11	31	8	5	15	2

(b) Estructura de Dato *Node*. Se asume que la secuencia de paréntesis para el *LZtrie* comienza desde las posición cero. Las i corresponden a las frases de la descomposición LZ78.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
$RNode[i]$	0	3	17	11	30	35	36	4	8	12	15	26	6	24	18	32	20	1

(c) estructura de Dato *RNode*. Se asume que la secuencia de paréntesis para el *RevTrie* comienza desde las posición cero. Las i corresponden a las frases de la descomposición LZ78.

Figura 2.3: Componentes restantes del LZ-índice de Navarro. *Range* y *Nodo* del ejemplo previamente visto. Por simplicidad en la práctica se utiliza la Estructura de Datos *RNode* donde se obtiene la misma información que en *Range*.

En el siguiente ejemplo se puede ver como se utilizan las Estructuras de Datos detalladas recientemente y que componen al NAV-LZI, con lo que se puede entender de mejor manera.

Ejemplo 2.3.1 Para encontrar todas las frases que terminan con el substring ‘**ab**’, se debe recorrer el *Revtrie* y buscar la frase ‘**ba**’, y como se puede ver en la Figura 2.3b este se encuentra en el nodo con preorden [6, 7], por lo tanto las frases 3 y 9 son el resultado esperado de la búsqueda. Si se busca en *Range* en la Figura 2.3a se puede notar que en la vertical donde están los nodos en preorden del *Revtrie*, para el [6, 7] también se encuentran las frases 3 y 9 de la descomposición del LZ. Para encontrar ahora todas las frases que comienzan con ‘**ar**’, es coincidentemente también en el nodo 6 pero ahora del *LZtrie* de la Figura 2.2a, por lo que todas las frases compuestas por los nodos [6, 9] comienzan con el patrón buscado. Estas frases corresponden a 4, 12, 10 y 16 (orden leyendo el árbol en preorden). Entonces, en *Range* estas frases se encuentran buscando en la horizontal el conjunto desde el 6 al 9. Donde las frases encerradas son 3, 9, 11 y 15 realizando la conversión $i + 1$ de estas frases se llega a las buscadas. Finalmente en el rango de *Range* $[6, 9] \times [6, 7]$ marca todas las frases i que terminan con ‘**ab**’ y todas las $i + 1$ que comienzan con ‘**ar**’.

2.3.2. Espacio Experimental del Indice

Como ya hemos visto el LZ-índice logra comprimir el texto, y su representación final requiere poco espacio, pero en el proceso de indexación para lograr construirlo se requiere de mucho espacio. Esto se debe principalmente a los punteros utilizados en la construcción de los tries. En un resultado experimental que se describe en [Nav08], sobre un texto en Ingles y de una secuencia de ADN, el mayor espacio utilizado para la construcción, como era de esperar, fue debido al *LZtrie* el cual utilizó 1,7 – 2,0 veces el tamaño del texto.

Analizando además el espacio utilizado por el *RevTrie* es en algunos casos 4 veces el tamaño del texto. Es el doble que *LZtrie* debido a los nodos vacíos que se deben agregar. Por lo tanto, en un análisis asintótico, este es el espacio que se toma en cuenta para obtener el espacio de indexación máximo del algoritmo. Para el caso del texto en Ingles el espacio de indexación fue de 4,8 – 5,8 veces el tamaño del texto, mientras que para la secuencia de ADN fue de 3,4 – 3,7 veces el tamaño del texto. A modo de comparación, la construcción de un suffix array sin ni una otra estructura de datos requiere de 5 veces el tamaño del texto [MF04].

2.3.3. El Algoritmo de Búsqueda NAV-LZI

En esta sección se detalla el algoritmo de búsqueda de NAV-LZI para un patrón $P[1..m]$ que se describe en [Nav04]. Como ya se ha visto anteriormente existen 3 tipos de ocurrencias, por lo que existe un algoritmo particular para cada una de las búsquedas que son soportadas por las estructuras de datos señaladas en la Sección 2.3.1.

Ocurrencias de Tipo 1. Dadas las propiedades de la compresión del LZ78, cada frase B_i que contiene a P es formada por otra frase más corta B_ℓ concatenada a un símbolo c . Si P no ocurre al final de B_i , entonces B_ℓ contiene también a P . Lo que se quiere buscar es la frase más corta posible B_i en la descomposición del LZ78 entre todas las frases B_i que contenga la ocurrencia P .

Dado que con el *LZTrie* solo se puede buscar prefijos, y dado que la frase B_i puede tener el string P como un sufijo, no se puede utilizar el *LZTrie* para encontrar B_i . Sin embargo, si P^r es el prefijo de B_i^r , este puede ser encontrado buscando P^r en el *RevTrie* en un tiempo de $O(m^2 \log \sigma)$.

Por lo tanto, las Ocurrencias de Tipo 1 son encontradas en un tiempo total de $O(m^2 \log \sigma + occ_1)$, esto ya que encontrar las ocurrencias en el *LZTrie* toman un tiempo constante.

Ocurrencias de Tipo 2. Estas son las ocurrencias que componen dos frases consecutivas de la descomposición LZ78, B_i y B_{i+1} , donde el patrón de búsqueda $P[1..m]$ coincide con el sufijo de B_i y el sufijo de $P[i + 1..m]$ coincide con el prefijo de B_{i+1} . La dificultad está en que la separación de P puede estar en cualquier posición, por lo que se debe buscar por todas las posibles separaciones. El procedimiento para las búsquedas de este tipo de ocurrencias es separar P en todas las posibles formas, y se buscan en el *RevTrie* el prefijo $P^r[i, 1]$ (obteniendo el nodo v_r) y se busca el sufijo $P[i + 1..m]$ en el *LZTrie* (obteniendo el nodo v_{lz}).

Dado que en el trie todos los strings representados en un subtree forman un intervalo en preorden, se tienen dos intervalos en preorden donde se encuentra el patrón a buscar: el primero está compuesto por las frases invertidas (frases que terminan con $P[1..i]$) y el segundo con las frases normales (frases que comienzan con $P[i + 1..m]$), por lo que se necesita encontrar el par de frases $(t, t + 1)$ tal que t se encuentra en el algún intervalo en preorden del

RevTrie mientras que $t + 1$ se encuentra en algún intervalo en preorden del *LZTrie*. En conclusión, si se define $p_{l_z} = \text{preorder}(v_{l_z})$ y $p_r = \text{preorder}(v_r)$, en la estructura de datos *Range* se debe buscar por:

$$[p_{l_z}..p_{l_z} + \text{subtreesize}(v_{l_z}) - 1] \times [p_r..p_r + \text{subtreesize}(v_r) - 1]$$

Las Ocurrencias de Tipo 2 pueden ser encontradas en un tiempo de $O(m^3 \log \sigma + (m + \text{occ}_2) \log n)$. El primer término proviene de la búsqueda que se debe realizar en los tries (en particular la búsqueda de las $O(m)$ particiones de P en el *RevTrie*) y el segundo término es por las $m - 1$ búsquedas realizadas en la estructura de datos *Range*.

Ocurrencias de Tipo 3. Este tipo de ocurrencias es cuando el patrón a buscar se encuentra en tres o más frases consecutivas, $B_{t-1}, \dots, B_{\ell+1}$, esto quiere decir que $P[i..j] = B_t \dots B_{\ell}$, por lo tanto: $P[1..i - 1]$ coincide con el sufijo de B_{t-1} y $P[j + 1..m]$ coincide con el prefijo de $B_{\ell+1}$. Este tipo de ocurrencia es la más compleja de buscar, por lo que es necesario aclarar ciertas observaciones: Dado que el algoritmo LZ78 garantiza que todas las frases representan un string distinto, existe como máximo una frase que coincida con $P[i..j]$, para cualquier i, j . Por lo que si se crean particiones de P en más de dos substrings consecutivos existe como máximo una ocurrencia de ese patrón para esta partición, lo que genera que las occ_3 tiendan a $O(m^2)$, las que corresponden a las distintas particiones de P .

Para encontrar este tipo de ocurrencias, Navarro define una matriz $C_{l_z}[1..m, 1..m]$ y arreglos A_i , para $1 \leq i \leq m$, los que guardan información de la búsqueda. Primero se identifica la única posible frase que coincide con cada substring de $P[i..j]$. Esto se realiza buscando todos los patrones del substring $P[i..j]$ en el *LZTrie*, incrementando i y por cada i se incrementa j . De esta manera, se realiza solo una búsqueda en el trie para cada i . Se agrega en $C_{l_z}[i, j]$ el nodo del *LZTrie* correspondiente a $P[i..j]$, y se agrega el par (id, j) al final de A_i , donde id es el identificador de la frase del nodo correspondiente a $P[i..j]$. Notar que dado que en la búsqueda por $P[i..j]$ se va incrementando j , se obtiene también los id en orden incremental, esto ya que el identificador de una frase es siempre mayor a la del nodo padre. Este proceso toma un tiempo de $O(m^2 \log \sigma)$.

Luego se buscan las $O(m^2)$ máximas concatenaciones de las frases sucesivas que coinciden

con patrones de los substrings continuos. Para $1 \leq i \leq j \leq m$, incrementando j , se intenta extender la coincidencia de $P[i..j]$ a la derecha. Si el id es el identificador de la frase para el nodo $C_{lz}[i, j]$, entonces se debe buscar por $(id + 1, r)$ en el arreglo A_{j+1} para algún r . Esta búsqueda puede ser binaria dado que el arreglo está ordenado. Si se encuentra $(id + 1, r)$ en A_{j+1} , esto quiere decir que $B_{id} = P[i..j]$ y $B_{id+1} = P[j + 1..r]$, lo que también significa que la concatenación de las frases $B_{id}B_{id+1}$ es igual a $P[i..r]$. Se repite el proceso para $j = r$, y este proceso se detiene cuando el par $(id + 1, r)$ no se encuentra en el arreglo correspondiente (esto quiere decir que la concatenación de las frases no se puede seguir extendiendo más, por lo que la actual es máxima).

Como se deben realizar $O(m^2)$ búsquedas binarias en los arreglos de tamaño $O(m)$, este procedimiento toma un tiempo en el peor caso de $O(m^2 \log m)$. En la práctica, la búsqueda binaria puede ser remplazada por *hashing*, lo que toma un tiempo de $O(m^2)$ en caso promedio.

Si $P[i..j] = B_t...B_\ell$ es una concatenación máxima, entonces se busca si la frase $B_{\ell+1}$ comienza con $P[j + 1..m]$, se comprueba si $Node[\ell + 1]$ desciende por el nodo $C_{lz}[j + 1, m]$, en tiempo constante por concatenación máxima. Finalmente se revisa si la frase B_{t-1} termina con $P[1..i - 1]$, comenzando con $Node[i - 1]$ en el *LZTrie* y sucesivamente consultando con el padre para verificar que los últimos $i - 1$ símbolos son iguales a $P'[1..i - 1]$, logrando esto en un tiempo de $O(m \log \sigma)$ por concatenación máxima. Si todas estas condiciones se cumplen, entonces se marca una ocurrencia $[t - 1, i - 1]$. En conclusión, las ocurrencias de tipo 3 son encontradas en un tiempo de $O(m^3 \log \sigma)$.

2.3.4. Solución de NAV-LZI

La solución que logra NAV-LZI es que dado un texto $T[1..u]$ sobre un alfabeto σ y con entropía empírica de $H_k T$, requiere un espacio de $4uH_k(T) + o(u \log \sigma)$ bits, para cualquier $k = o(\log_\sigma u)$. Dado un patrón de búsqueda $P[1..m]$, este índice es capaz de:

1. Ubicar la occ ocurrencias del patrón P en el texto T en un tiempo de $O(m^3 + \log \sigma + (m + occ) \log u)$.

2. Contar el número de ocurrencias del patrón en un tiempo $O(m^3 \log \sigma + m \log u + occ)$.
3. Determinar si el patrón P existe en T en un tiempo de $O(m^3 \log \sigma + m \log u)$.
4. Extraer cualquier substring de largo ℓ en un tiempo de $O(\ell)$.

En el Capítulo 3, se verá una alternativa para implementar NAV-LZI, ya que como se menciona anteriormente este es el índice en el cual se basa este trabajo. Principalmente se presentan las *Estructuras de Datos Sucintas*, con las cuales es posible lograr la misma eficiencia que las presentadas recientemente pero utilizan un menor espacio. En el trabajo [Nav04] describe que las cuatro estructuras de datos Originales del NAV-LZI en su formato sucinto utilizan un espacio de $n \log n + o(u \log \sigma)$.

2.4. LZ-índice de Russo y Oliveira (ILZI)

Russo y Oliveira [RO07] definen el *índice Lempel-Ziv invertido*. Descartaron la original descomposición LZ78, y definieron una nueva forma llamada *maximal parsing* de un texto T .

Para construir el ILZI, primero se construye la descomposición del LZ78 pero de $T^r = Z_1, \dots, Z_n$ y luego junto a otras estructuras de datos se construye el *suffix tree* $\mathcal{T}_{78}$ para el conjunto de strings $\{Z_1^r, \dots, Z_n^r\}$. Russo y Oliveira probaron que la cantidad de nodos del $\mathcal{T}_{78}$ es $O(2n)$, donde n es la cantidad de frases de la descomposición del LZ78 de T .

Para buscar un patrón, las ocurrencias de tipo 1 pueden ser encontradas al igual que en NAV-LZI en un tiempo de $O(m + occ_1)$. Para encontrar las ocurrencias de tipo 2 y 3 en este índice se logra un tiempo de $O(m)$ de una manera interesante. En vez de buscar por todas los posibles patrones, como se hace en NAV-LZI para ocurrencias de tipo 3, Russo y Oliveira dividen el patrón en substrings máximos. Esto quiere decir los subtrings de P que son nodos de $\mathcal{T}_{78}$ y no son prefijos de otro substring de P que también son nodos de $\mathcal{T}_{78}$. Este proceso para dividir los substrings máximos es logrado con un algoritmo de programación dinámica, y se utilizan algunas de las propiedades de los *suffix tree*. Por lo tanto, como se puede suponer

esta división de los substring máximos no puede ser mayor a $O(m)$ por lo que en un peor caso (poco probable) tiene la misma complejidad que NAV-LZI pero en un caso promedio, que es lo que ocurre en la práctica los tiempo de respuesta son mejores.

2.4.1. Solución de ILZI

De esta manera, la solución que logra ILZI es que dado un texto $T[1..u]$ sobre un alfabeto σ y con entropía empírica de $H_k T$, requiere un espacio de $(5 + \epsilon)uH_k(T) + o(u \log \sigma)$ bits, para cualquier $0 < \epsilon < 1$. Dado un patrón de búsqueda $P[1..m]$, este índice es capaz de:

1. Ubicar la occ ocurrencias del patrón P en el texto T en un tiempo de $O((\frac{m}{\epsilon} + occ) \log u)$.
2. Contar el número de ocurrencias del patrón en un tiempo $O(\frac{m}{\epsilon} \log u + occ)$.
3. Determinar si el patrón P existe en T en un tiempo de $O(\frac{m}{\epsilon} \log u)$.
4. Extraer cualquier substring de largo ℓ en un tiempo de $O(\frac{\ell}{\epsilon \log_\sigma u})$.

2.5. LZ-índice de Arroyuelo, Navarro y Sadakane (ANS-LZI)

Este índice sigue el mismo principio que el construido por Navarro en [Nav08]. La diferencia está en la forma de construcción del índice el que logra una mejora de espacio cambiando la manera en como se navegan las estructuras de datos. El detalle de este índice se puede ver [Bil09].

Como se puede ver en la Figura 2.4, la cual es la utilizada en NAV-LZI, esta utiliza los *ids* para luego encontrar el nodo correspondiente en el LZTrie o en el RevTrie a través de los punteros a los nodos con las estructuras de *Node* y *RNode*.

En cambio ANS-LZI utiliza la navegación de la Figura 2.5. Como se puede ver en este esquema, desaparecen las estructuras de datos *Node* y *RNode*. La idea de esto es que *ids*

y $rids$ son permutaciones inversas que logran descifrar los nodos de los Tries dado un id gracias a ids^{-1} . Cada par de ids y su correspondiente ids^{-1} puede ser representado en un espacio de $1 + \epsilon$. Por lo tanto, en total queda $2 + \epsilon$. El parámetro ϵ permite reducir el espacio del índice, a cambio de mayor tiempo de búsqueda.

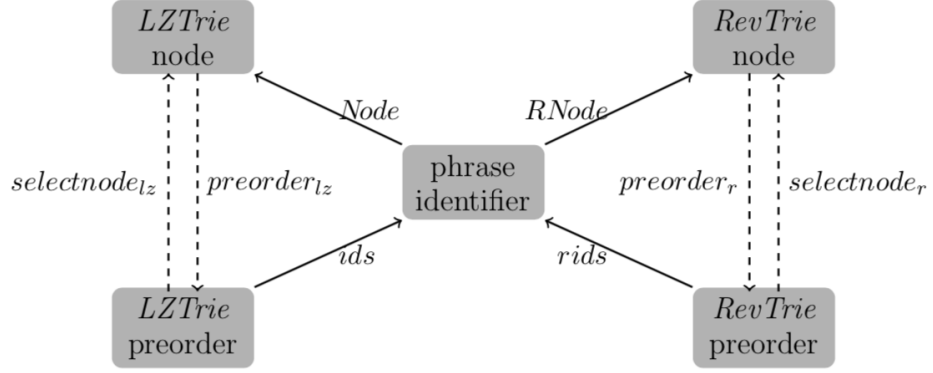


Figura 2.4: Esquema original de NAV-LZI.

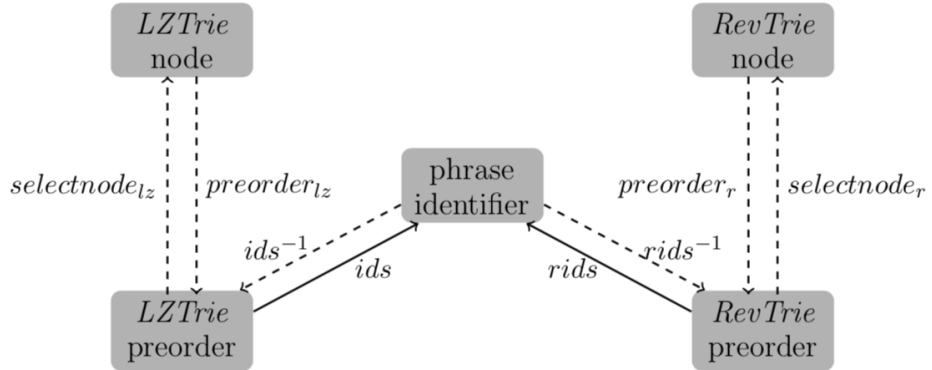


Figura 2.5: Esquema de ANS-LZI que logra una mejora en la utilización de espacio.

2.5.1. Solución de ANS-LZI

De esta manera, la solución que logra ANS-LZI es que dado un texto $T[1..u]$ sobre un alfabeto σ y con entropía empírica de $H_k T$, requiere un espacio de $(2 + \epsilon)uH_k(T) + o(u \log \sigma)$

bits, para cualquier $0 < \epsilon < 1$. Dado un patrón de búsqueda $P[1..m]$, este índice es capaz de:

1. Ubicar la ocurrencia del patrón P en el texto T en un tiempo de $O(m^2 + (m + occ) \log u)$.
2. Contar el número de ocurrencias del patrón en un tiempo $O(m^2 + m \log u + occ)$.
3. Determinar si el patrón P existe en T en un tiempo de $O(m^2 + m \log u)$.
4. Extraer cualquier substring de largo ℓ en un tiempo de $O(\ell / \log_\sigma u)$ (el substring tiene que existir).

2.6. Resumen de los LZ-índices

En los siguientes cuadros se puede ver un resumen de los LZ-índices vistos en esta sección. En el Cuadro 2.1, se puede ver la diferencia entre los espacios que utiliza cada uno de los índices y en el Cuadro 2.2 se hace un comparativo entre las principales operaciones.

Cuadro 2.1: Comparación de uso de espacio en bits que requiere cada índice.

Índice	Espacio en bits
Kärkkäinen y Ukkonen (KU-LZI)	$O(uH_k(T)) + u \log \sigma + o(u \log \sigma)$
Ferragina y Manzinini (FM-LZI)	$O(uH_k(T) \log^\gamma u) + o(u \log \sigma \log^\gamma u)$
Navarro (NAV-LZI)	$4uH_k(T) + o(u \log \sigma)$
Russo y Oliveira (ILZI)	$(5 + \epsilon)uH_k(T) + o(u \log \sigma)$
Arroyuelo, Navarro y Sadakane (ANS-LZI)	$(2 + \epsilon)uH_k(T) + o(u \log \sigma)$

Cuadro 2.2: Comparación entre los distintos índices para las operaciones básicas sobre un texto T , dado un patrón de búsqueda $P[1..m]$.

Indice	ubicar	contar	dse	extraer
KU-LZI	$O(m^2 + (m + occ) \log u)$	$O(m^2 + m \log u + occ)$	$O(m^2 + m \log u)$	$O(\ell / \log_{\sigma} u)$
FM-LZI	$O(m + occ)$	$O(m)$	$O(m)$	$O(\ell)$
NAV-LZI	$O(m^3 + \log \sigma + (m + occ) \log u)$	$O(m^3 \log \sigma + m \log u + occ)$	$O(\frac{m}{\epsilon} \log u)$	$O(\ell)$
ILZI	$O((\frac{m}{\epsilon} + occ) \log u)$	$O(\frac{m}{\epsilon} \log u + occ)$	$O(\frac{m}{\epsilon} \log u)$	$O(\frac{\ell}{\epsilon \log_{\sigma} u})$
ANS-LZI	$O(m^2 + (m + occ) \log u)$	$O(m^2 + m \log u + occ)$	$O(m^2 + m \log u)$	$O(\ell / \log_{\sigma} u)$

Capítulo 3

Nueva Implementación de LZ-index

Esta memoria toma como base la implementación del LZ-index de Arroyuelo [Bil09], en el formato propuesto del proyecto *Pizza&Chili* que implementa la solución de Navarro [Nav08] explicada en la Sección 2.3. Este LZ-index utiliza la descomposición en frases del LZ78 y sus estructuras de datos principales para hacer las búsquedas de las ocurrencias son el *LZTrie*, *RevTrie*, *Nodemap* y *Rnodemap*.

Para la implementación de estas estructura de datos, se emplea una alternativa que son las estructuras de datos sucintas, que tienen como finalidad utilizar menor espacio pero sin perder eficiencia. Esta misma implementación es seguida luego de lo realizado por Arroyuelo.

En esta sección, se detallarán las principales diferencias de esta implementación con la versión anterior de [Bil09], donde lo principal radica en que este trabajo está realizado en C++ 11, con lo que se agrega la funcionalidad de *templates*. De esta manera, el índice puede ser parametrizado según necesite el usuario. Además, debido a la actualización del lenguaje de programación, es posible incorporar a la implementación la librería SDSL, la que es la más utilizada para incorporar estructuras de datos sucintas.

3.1. Estructuras de Datos Sucintas

Las *estructura de datos sucinta* son estructuras de datos que usan una cantidad de espacio cercana a la cota inferior de la teoría de la información, pero, a diferencia de otras representaciones comprimidas, ésta permite realizar las mismas operaciones que en las estructuras de datos tradicionales sin perder eficiencia, al menos asintóticamente. En este trabajo se utiliza particularmente los paréntesis balanceados, ya que gracias a estos es posible construir árboles de forma sucinta.

3.1.1. Paréntesis Balanceados

Una estructura de datos sucinta conocida y utilizada en esta memoria corresponde a una representación estática de n pares de paréntesis balanceados, como se ven en el Cuadro 3.1. Existen operaciones que se pueden realizar a este tipo de paréntesis, tales como: Encontrar el paréntesis que cierra o abre a cierto paréntesis, o encontrar el par de paréntesis que encierra más cercanamente a otro. Los paréntesis balanceados serán de utilidad para representar los tries del índice de forma sucinta.

Los paréntesis balanceados son básicamente una secuencia de bits (o vector de bits), en donde un 1 corresponde a '(', mientras que un 0 es ')'. Existen autores donde consideran esta nomenclatura al revés, pero en esta memoria se sigue este formato ya que es como está diseñada la librería utilizada SDSL.

Las operaciones principales que se puede realizar sobre un bitvector son las de Rank y Select, las que serán de utilidad para recorrer este arreglo de manera eficiente.

Estructuras de Datos para Rank y Select. Dado un bitvector $A[1..n]$ la operación $rank_1(i)$ en su formato por defecto encuentra la cantidad de 1's hasta la posición i -ésima incluyendo esta. Mientras que la operación $select_1(i)$ por defecto retorna la posición del i -ésimo 1. Por ejemplo, si se tiene el siguiente paréntesis balanceado con su respectivo *bitvector*

Cuadro 3.1: Ejemplo de paréntesis balanceado con su respectiva representación de bitvector

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
(	(	(	)	(	)	)	(	)	(	(	(	)	(	)	)	(	)	)	)
1	1	1	0	1	0	0	1	0	1	1	1	0	1	0	0	1	0	0	0

Si se realiza operación $rank_1(10)$ este retorna 7. Mientras que $select_1(10)$ retorna 16.

Estas dos operaciones se dicen que son por defecto, ya que buscan los abre paréntesis (1's), pero también pueden ser aplicadas de la misma manera para los cierra paréntesis (0's). A estas se les llaman $rank_0(i)$ y $select_0(i)$.

Operaciones Básicas de los Paréntesis Balanceados. Si se tiene una secuencia de $2n$ paréntesis balanceados, las operaciones básicas que se desean soportar sobre estos son:

- $findclose(i)$: Dado un abre paréntesis en la posición i retorna la posición de su cierra paréntesis.
- $findopen(i)$: Dado un cierra paréntesis en la posición i retorna la posición de su abre paréntesis.
- $enclose(i)$: Dado un paréntesis abierto en la posición i retorna la posición del abre paréntesis del más cercano que encierra al de la posición i .
- $excess(i)$: Calcula la diferencia entre los paréntesis abiertos y cerrados hasta la posición i .

Todas estas operaciones son posibles resolverlas en tiempo constante, tal como se muestra en [MR01].

3.1.2. Representación Sucinta de Árboles

Dado un árbol con n nodos existe una representación por Munro y Raman en [MR01] que plantea una representación en formato sucinto con paréntesis balanceados. Este árbol ocupa un espacio de $2n + o(n)$ bits, lo que es cercano a la cota inferior teórica de $2n - \Theta(\log n)$ bits.

Para la construcción del árbol con paréntesis balanceados se realiza un recorrido en preorden, donde se escribe un abre paréntesis cada vez que haya un nodo y un cierra paréntesis significa que se debe subir un nivel para seguir construyendo en preorden. El subárbol de un nodo i queda representado por todos los paréntesis que se encuentran dentro del respectivo abre y cierra paréntesis de i . Por lo tanto, en esta representación, una hoja queda representada por “()” dado que no encierra a ni un otro subárbol.

Continuando con el mismo ejemplo mostrado en secciones anteriores de [Bil09], en las Figuras 3.1, 3.2 se puede ver esta representación.

```

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35
par: ( ( ( ) ( ( ) ) ( ) ( ( ) ( ) ( ) ) ( ( ) ) ) ( ( ( ) ) ) ( ( ( ) ) ) )
ids:  1 17  3 15  14  4 12  10  16  6 11  2 7 9  5 8 13
letts: a $  b r  l  r a  d  l  - p  l a b  - a p

```

Figura 3.1: Paréntesis balanceado del árbol mostrado en 3.2

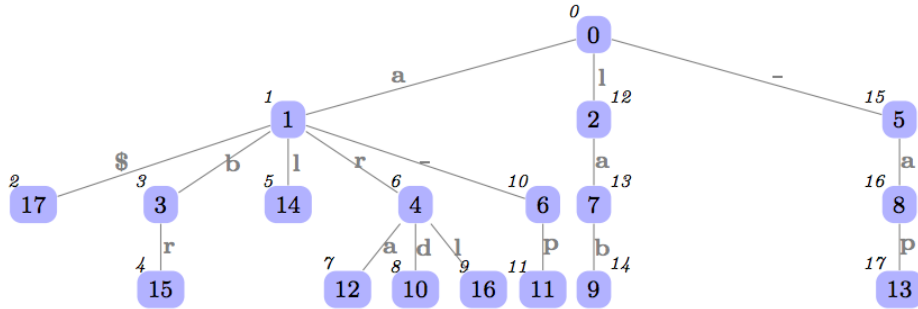


Figura 3.2: Árbol correspondiente a paréntesis balanceados de la Figura 3.1

Con esta representación y las operaciones básicas descritas en la sección anterior de los paréntesis balanceados, es posible cubrir las siguientes funciones dado un árbol en su formato sucinto en tiempo constante.

- $parent(i)$: Retorna el nodo padre de i .
- $subtreesize(i)$: Retorna el tamaño del subárbol del nodo i incluyéndolo.
- $depth(i)$: Retorna la altura del nodo i en el árbol.
- $firstchild(i)$: Retorna el primer hijo del nodo i .
- $nextsibling(i)$: Retorna el próximo hermano del nodo i .
- $ancestor(i, j)$: Indica si el nodo i es antecesor del nodo j .

La implementación de estas funciones con las operaciones básicas de los paréntesis balanceados queda entonces de la siguiente manera:

$$parent(i) = enclose(i)$$

$$subtreesize(i) = (findclose(i) - i + 1)/2$$

$$depth(i) = excess(i)$$

$$firstchild(i) = i + 1$$

$$nextsibling(i) = findclose(i) + 1$$

$$ancestor(i, j) = i \leq j \leq findclose(i)$$

El detalle de como se implementan en la práctica todas las funciones del *LZTrie* y *RevTrie* utilizadas en este trabajo se detallan en las secciones 3.3 y 3.4 respectivamente.

3.2. Implementación del Nuevo LZ-index

En la implementación del nuevo *LZ-index* se utiliza la librería **SDSL - Succinct Data Structure Library** [Gog]. Esta librería trae consigo todas las operaciones básicas de los paréntesis balanceados creados desde un vector de bits representando los paréntesis.

3.2.1. Definición de la Clase

La siguiente definición corresponde a la clase principal que implementa el nuevo LZ-Index:

```
1  template<class bp, class alphabet>
2  class LZindex {
3  private:
4      LZtrie<bp, alphabet> *LZT; //LZTrie
5      Revtrie<bp, alphabet> *RVT; //RevTrie
6      int_vector<> node; // Vectors of pointers to node in LZTrie
7      int_vector<> rnode; // Vectors of pointers to node in RevTrie
8  }
```

En las líneas 4, 5, 6 y 7 se pueden ver las estructuras de datos que representan a este índice, tal como se explica en la Sección 2.3.

El primer aporte de esta implementación, en comparación en su versión anterior, se puede ver en la línea 1, donde se crea un *template* (funcionalidad de C++) que permite ingresar como parámetros qué tipo de estructura de paréntesis balanceado se desea utilizar y el tamaño del alfabeto del texto. En las versiones anteriores, estos parámetros eran fijos a un cierto tipo de paréntesis balanceados y alfabeto de tipo `char`. Esto último solo permite textos con alfabetos de tamaño 255, dado que `char` es representado por solo 1 byte (8 bits). En la nueva implementación, en cambio, se puede utilizar alfabetos más grandes con tan solo declarar las clases que representan 8, 16, 32 o 64 bits. Los tamaños de los posibles alfabetos que se pueden usar se puede ver en el Cuadro 3.2. Se puede apreciar cómo con la clase correspondiente a 16 bits bastaría para representar, por ejemplo, textos escritos en la lengua China, que en los caracteres de Han cuenta actualmente con un número cercano a los 2500 símbolos. También queda a disposición del usuario qué tipo de paréntesis balanceado se desea ocupar abriendo la posibilidad de utilizar otras versiones implementadas en la librería SDSL, las que se pueden ver en el Cuadro 3.3.

Cuadro 3.2: Tipos de alfabeto con los que se puede parametrizar el LZ-index.

Clase	Largo Alfabeto
uint8_t	255
uint16_t	65.535
uint32_t	4.294.967.295
uint64_t	18.446.744.073.709.551.615

Cuadro 3.3: Tipos de paréntesis balanceados, conocidos en la librería SDSL como BPS (balanced parentheses support), con los que se puede parametrizar el LZ-index

BPS	Descripción
bps_support_g<>	Pioneer structure (2 levels).
bp_support_gg<>	Pioneer structure ($\log_B n$ levels).
bp_support_sada<>	Min-Max-Tree approach.

3.2.2. Principales Métodos

1. count(): Retorna la cantidad de ocurrencias encontradas en el texto del patrón pattern.

```

1  int count(alphabet *pattern, uint64 length, uint64* numocc)
2  {
3      trieNode *fwd,*bwd;
4      uint64 *fwdid;
5      Count = 0;
6      fwd = fwdmatrix(pattern,length,&fwdid);
7      bwd = bwdmatrix(pattern,length,fwdid);
8      reportType1(fwd,bwd,fwdid,length);
9      if (length > 1) reportType2(fwd,bwd,length);
10     if (length > 2) reportType3(fwd,fwdid,bwd,pattern,length);
11
12     delete [] fwd;
```

```

13     delete [] bwd;
14     delete [] fwdid;
15     *numocc      = Count;
16     return 0;
17 }

```

Funcionamiento: Se crean las matrices fwd, bwd y fwdid a partir de las operaciones del *LZTrie* y *RevTrie* y apoyadas también por *Node* y *Rnode*. Estas matrices ayudan en la búsqueda de los nodos de cierta posición en los bit vectors. Luego, se busca el patrón en las ocurrencias de tipo 1, 2 y 3. Notar que si el patrón es de largo 1 solo puede estar en una ocurrencia de tipo 1 y si es de largo 2, solo puede estar en ocurrencias de tipo 1 o 2, es decir, se ahorra la búsqueda de ocurrencias si no son posibles de ocurrir.

2. size_bytes(): Retorna el tamaño en bytes del índice.

```

1  uint64 size_bytes(){
2      uint64 size = 0;
3      size += RVT->size();
4      size += LZT->size();
5      size += size_in_bytes(node);
6      size += size_in_bytes(rnode);
7      return size;
8  }

```

Funcionamiento: Se suman los tamaños de todos los atributos de la clase en bytes.

3.3. Implementación del LZTrie

El *LZTrie* es la primera estructura de datos y se revisará sus principales atributos que permiten diferenciar esta versión del *LZ-index* con sus antecesores.

3.3.1. Definición de la Clase

La siguiente definición corresponde a la clase LZTrie:

```
1  template<class bp, class alphabet>
2  class LZtrie {
3  private:
4      bp bps; // initialization of balance parentheses
5      bit_vector bv; // bit vector (parentheses)
6      vector<alphabet> letters; // vector with the letters
7      uint64 n; // number of nodes
8      uint64 nbits; // log n
9      int_vector<> id; // ids of the trie
10     vector<uint64> boost; // direct pointer to first children
11     uint64 alphabetSize; // alphabet size
12 }
```

La línea 4 y 5 definen el tipo de paréntesis balanceado que se ocupará y el vector de bits que contiene los paréntesis balanceados. El atributo `bps` permite utilizar las operaciones principales sobre los paréntesis balanceados definidos previamente en la librería `sds1`.

La línea 6 y 9 corresponde al vector de letras y vector con ids, como el representado en el ejemplo de la Figura 3.1. Una observación que se debe realizar es que en ambos vectores la primera posición es vacía y se marca con un 0 (`id[0] = letts[0] = 0`), esto dado a que el primero nodo (nodo raíz) no tiene ninguna letra por la que descienda.

La línea 7 y 8 simplemente definen la cantidad de nodos que tiene el *LZtrie* y el logaritmo de este respectivamente. `nbits` es útil para crear el vector `id` con exactamente los bits necesario y no desperdiciar espacio.

La línea 10, correspondiente al vector `boost` guarda todos los hijos de la raíz para poder acceder más rápidamente a estos.

Por último, la línea 11 es el largo del alfabeto.

3.3.2. Principales Métodos

Los métodos que componen el *LZTrie* utilizan principalmente las operaciones básicas de los paréntesis balanceados lo que ayuda a recorrer el árbol.

1. letter(): Retorna la letra por la cual desciende el nodo *i*.

```
1 alphabet letter(trieNode i) {  
2     return letters[bps.rank(i - 1)];  
3 }
```

Funcionamiento: rank retorna la cantidad de abre paréntesis que hay hasta cierta posición *i*. La implementación de este método en sds1 retorna incluyendo la posición en la que está, es por esto que se realiza *i-1*. Con la cantidad de nodos que hay hasta esa posición, se puede saber que letra desciende por el nodo *i* con el vector de letras.

2. child(): Si es posible descender por la letra *c* desde el nodo *i*, retorna el nodo *j* que corresponde al nodo de la letra *c*.

```
1 trieNode child(trieNode i, alphabet c) {  
2     trieNode j;  
3     alphabet tc;  
4     if (i == ROOT) return boost[c];  
5     // Next child  
6     j = i + 1;  
7     while (bv[j] == OPEN_P) { // There is a child  
8         tc = letters[bps.rank(j - 1)];  
9  
10        // shortcut for leftrankLZTrie: is a child by letter c?  
11        if (tc == c) return j;  
12        j = bps.find_close(j) + 1;  
13    }
```

```
14
15     return NULLT;
16 }
```

Funcionamiento: El primer `if` hace uso del vector `boost` donde si detecta que el nodo `i` corresponde a la raíz, utiliza el puntero que se había almacenado anteriormente. Luego se recorre todos los hijos del nodo `i` en preorden y si se encuentra la letra `c` se retorna el nodo `j` asociado.

3. parent(): Retorna el padre del nodo `i`.

```
1  trieNode parent(trieNode i) {
2      if (i == ROOT) return NULLT;
3      if (boost[letters[bps.rank(i - 1)]] == i) return ROOT;
4      return bps.enclose(i);
5  }
```

Funcionamiento: En la línea 3 se verifica si el nodo corresponde a un nodo hijo de la raíz para responder con una mayor eficiencia que el nodo es la raíz. Si no lo es, retorna `enclose(i)` que significa el paréntesis que encierra a `i` que corresponde a su padre.

5. subtreesize(): Retorna la cantidad de nodos del subárbol del nodo `i` inclusive.

```
1  uint64_t subtreesize(trieNode i) {
2      trieNode j;
3      j = bps.find_close(i);
4      return (j - i + 1) / 2;
5  }
```

Funcionamiento: En la línea 3 se busca por el paréntesis que cierra al nodo `i`, y luego se hace la operación de la línea 4 que retorna la cantidad de nodos incluyendo al nodo `i`.

6. ancestor(): Determina si el nodo `i` es ancestro del nodo `j`, retorna verdadero o falso según corresponda.

```

1 bool ancestor(trieNode i, trieNode j) {
2     return (i <= j) && (j < bps.find_close(i));
3 }

```

Funcionamiento: Si el nodo *i* es menor *j* significa que el preorden de este nodo se encuentra antes por lo que podría ser un ancestro. Si además *j* es menor que el cierra paréntesis de *i* quiere decir que necesariamente es un hijo de *i*. La operación *i <= j* realiza una primera verificación y ahorra el llamado a `find_close(i)` en caso de no ser necesario.

7. next(): Retorna el siguiente nodo en preorden de *i*.

```

1 trieNode next(trieNode i) {
2     i++;
3     while(bv[i] == CLOSE_P) {
4         if(i < 2*n) i++;
5         else return 2*n;
6     }
7     return i;
8 }

```

Funcionamiento: Se suma uno a *i*, si este es un abre paréntesis (nodo) se retorna, si no, entra en un ciclo hasta encontrar el siguiente nodo para retornarlo.

3.4. Implementación del Revtrie

La implementación del *RevTrie* sigue la misma estructura del *LZTrie*, es por esto que los métodos son exactamente iguales a como se ve en la sección anterior con excepción a la función `child`, esto debido a que el *RevTrie* no guarda los nodos vacíos (optimización de espacio), por lo que puede existir un hijo que está oculto que esta función debe identificar.

3.4.1. Definición de la Clase

La siguiente definición corresponde a la clase RevTrie:

```
1  template<class bp, class alphabet>
2  class Revtrie {
3  private:
4      bp bps;
5      bit_vector bv;
6      uint64 n;
7      uint64 nbits;
8      int_vector<> id;
9      int_vector<> nodemap;
10     LZtrie<bp, alphabet> *LZT;
11 }
```

La diferencia con la Definición de la Clase del *LZTrie* radica en una primera instancia en la línea 9, la cual es el misma estructura de datos utilizada en la Clase *LZ-index*, este guarda punteros la posición de los nodos del *LZTrie*. Además, como se puede ver en la línea 10 esta clase también tiene un puntero al *LZTrie*, este es de utilidad para descubrir los nodos vacíos del *RevTrie* y también las letras correspondientes a los nodos.

3.4.2. Principales Métodos

7. child(): Si es posible descender por la letra *c* desde el nodo *i*, situa punteros *lzl* y *lzt* en las correspondientes posiciones del *LZTrie* y retorna nodo *j*.

```
1  trieNode child(trieNode i, uint64 depth, alphabet c,
2    trieNode *lzl, trieNode *lzt) {
3      trieNode j;
4      trieNode jlz;
5      uint64 d, pos, jid;
```

```

6     alphabet tc;
7     j = i + 1;
8     while (bv[j] == OPEN_P) { // There is a child
9         pos = leftrank(j);
10        jid = rth(pos);
11        jlz = nodemap[jid];
12        for (d = depth; d; d--) {
13            jlz = LZT->parent(jlz);
14        }
15        tc = LZT->letter(jlz);
16        if (tc > c) break;
17        if (tc == c) {
18            *lzl = jlz;
19            if (pos < n-1 && id[pos+1] == jid) { //empty node
20                pos = rightrank(j);
21                jlz = nodemap[rth(pos)];
22                for (d = depth; d; d--) jlz = LZT->parent(jlz);
23                *lzl = jlz;
24            }
25            else *lzl = NULLT;
26            return j;
27        }
28        j = bps.find_close(j) + 1;
29    }
30
31    *lzl = *lzl = NULLT;
32    return NULLT; // no child to go down by c
33
34 };

```

Funcionamiento: En esta función es necesario utilizar el *LZTrie* por dos razones principales. La primera es que al no tener una estructura de datos con las letras del *RevTrie* es necesario consultarlo con en el LZT. La segunda es debido a los nodos vacíos, este árbol si se incluyen los nodos vacíos utiliza un espacio de $O(2n)$, por lo que para ahorrar este espacio se eliminan los nodos vacíos, pero igual se deben tomar en cuenta para saber si existen hijos por la letra c. Esto se puede ver en la línea 12 donde se hace un `for` que sube hasta el nodo correspondiente en el LZT y luego en la línea 15 se pregunta por la letra que baja por ese nodo. Esa letra puede ser que no se esté viendo por los nodos vacíos del *RevTrie*. Además en la línea 19 se verifica si este nodo es vacío o no, con lo que se dejan los punteros `lzl` y `lzh` apuntando a los respectivos nodos del hijo por la letra c (nodo de más a la izquierda y el de más a la derecha).

Capítulo 4

Experimentos

En esta sección se detallan todos los experimentos realizados con el fin de comprobar la eficiencia de este nuevo *LZ-index*. Se compara estos resultados con respecto a versión anterior. Se mide el espacio utilizado por el índice y tiempo de respuesta. El tiempo de construcción no es analizado ni comparado con la versión anterior ya que el objetivo de este trabajo no es obtener una mejora en esta área.

4.1. Configuración Experimental Inicial

Para los experimentos se utiliza un Servidor en la nube de DigitalOcean. Este cuenta con Sistema Operativo Ubuntu 18.02.1 x64bits, 12 vCPU, 48GB Memoria Principal, 960GB SSD DISK. Se compila el programa con g++ 4.2.1 y optimización completa.

Colección de Textos. Los textos utilizados son los que provee el repositorios de índices comprimidos *Pizza&Chili* [piz], que permiten generar distintos escenarios reales.

1. **English Text:** Este es un texto clásico con palabras en ingles. Representa un escenario real de una persona buscando por términos dentro del texto. Se utiliza la versión de 200 MB. Este puede ser encontrado en <http://pizzachili.dcc.uchile.cl/texts/nlang/english.200MB.gz>

2. **Secuencia de ADN:** En la bioinformática se utilizan textos indexados, dada su representación y gran tamaño. Para este caso se utiliza el archivo de 200 MB <http://pizzachili.dcc.uchile.cl/texts/dna/dna.200MB.gz>.
3. **Secuencia MIDI:** Se analiza también un archivo con secuencias MIDI para el procesamiento musical. En este caso se utiliza un archivo de 50 MB. <http://pizzachili.dcc.uchile.cl/texts/music/pitches.gz>.
4. **Archivo XML:** Estos Archivos se han convertido en un cierto estándar para representar documentos semi estructurados, por lo tanto es fundamental poder manejar estos archivos eficientemente. Dado que los XML son usualmente requeridos por otras aplicaciones, poder comprimir y buscar rápidamente en ellos se vuelve básico. En este caso se utiliza el archivo de 200 MB <http://pizzachili.dcc.uchile.cl/texts/xml/dblp.xml.200MB.gz>.
5. **Proteínas:** Este archivo también proviene de aplicaciones biológicas y es de gran utilidad poder indexar, para luego buscar rápidamente. Se utiliza el archivo de 200 MB <http://pizzachili.dcc.uchile.cl/texts/protein/proteins.200MB.gz>.
6. **Códigos Fuente:** Se utilizan códigos fuentes, los cuales usualmente están poco estructurados por lo que será interesante comprobar la indexación y búsqueda dentro de estos. Se utiliza la versión de 200 MB <http://pizzachili.dcc.uchile.cl/texts/code/sources.200MB.gz>.

Comparación con LZ-index Original. Se compara el Nuevo LZ-index con el creado por Arroyuelo y Navarro, mas detalle de este se encuentra en el trabajo [Bil09] que sigue el formato de NAV-LZI. Este se puede encontrar en <http://pizzachili.dcc.uchile.cl/indexes/LZ-index/LZ-index.tar.gz>. En esta memoria, solo se compara consultas de tipo count.

Para la búsqueda de patrones, se generaron con la función `genpatterns` de *Pizza&Chili*. Esta crea patrones aleatorios que existan en el texto. Se utilizaron estos mismo patrones para el LZ-index original (Arroyuelo y Navarro) como para el nuevo LZ-index (este trabajo). Se crearon cuatro archivos por texto con patrones de largo 5, 8, 12 y 20. Cada archivo contiene 1000 patrones.

4.2. English Text

Espacio utilizado. Como se puede ver en la Figura 4.1, el nuevo índice utiliza 43 MB menos que el índice original. Por lo tanto, hay un 13 % de mejora con respecto al tamaño de la antigua implementación. Esto es logrado gracias a las estructuras de datos de la librería SDSL, y permite trabajar estos índices en memoria en computadores sin requerimientos técnicos tan avanzados. Para ser construido, el nuevo índice toma 334,77, segundos mientras que el índice original toma 151,88 segundos.

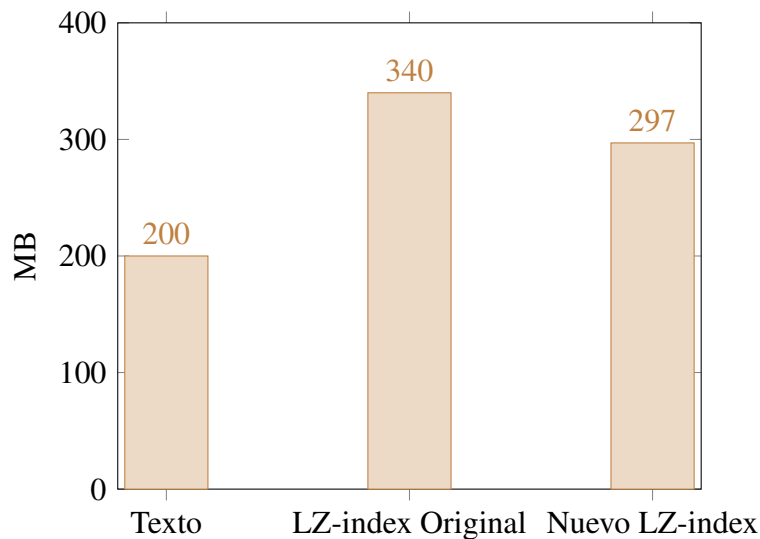


Figura 4.1: Tamaño en MB del texto English, LZ-index original y el nuevo LZ-index desarrollado en este trabajo.

Además, es interesante analizar la información del Cuadro 4.1, donde está la cantidad de nodos que componen al LZTrie y al RevTrie. Se comprueba que en la práctica sigue lo expuesto en la teoría, donde la cantidad de nodos del RevTrie es mayor. Es importante remarcar que es mucho menor a los $2n$ del peor caso. Como se puede ver, en este caso, el RevTrie cuenta con solo un 3 % más de nodos.

Tiempo de Respuesta. Para analizar el tiempo de respuesta del nuevo índice, se comparan los métodos count de ambos índices. Estos cuentan la cantidad de ocurrencias de los patrones dentro del texto. En la Figura 4.2 se puede ver un comparativo entre los dos índices en cuestión. Para los patrones de largo 5 el nuevo índice logra una mejora de un 40 % en el

Cuadro 4.1: Cantidad de nodos en LZTrie y cantidad de nodos en RevTrie para el texto English.

Nodos LZTrie	21.411.876
Nodos RevTrie	21.999.087

tiempo de búsqueda, lo cual es notable. También se logra una mejora para los patrones de largo 8 con un 24 %. Sin embargo, para los textos de largo 12 y 20, el índice original obtiene mejores tiempos de respuesta, 49 % y 37 % respectivamente. Esto se debe a que en estos patrones existen más ocurrencias de tipo 2 y 3, por lo que se deben ejecutar una mayor cantidad de operaciones `rank` y `select`. Esto ocurre ya que el índice tiene estructuras de datos que requieren menos espacio, pero tienen la desventaja de perder en tiempo de respuesta. Esto se comprueba en el Cuadro 4.3, donde en los patrones de largo 12 y 20 se empieza a ver más ocurrencias de tipo 2 y 3. En el Cuadro 4.2, se puede ver la cantidad de ocurrencias encontradas.

Cuadro 4.2: Cantidad de ocurrencias encontradas para los 1000 patrones de cada largo, para el texto English.

Largo Patrón	Cantidad de Ocurrencias
5	55.410.345
8	3.776.085
12	251.338
20	9.163

Cuadro 4.3: Porcentaje de tipo de ocurrencias para patrones de distinto largo para el texto English.

Largo Patrones	Occ Tipo 1	Occ Tipo 2	Occ Tipo 3
5	72.78 %	27.22 %	0.00 %
8	62.94 %	37.05 %	0.02 %
12	69.26 %	30.26 %	0.48 %
20	42.06 %	39.56 %	18.38 %

Si se analiza en promedio cuánto se demora el algoritmo en encontrar un patrón en el texto, simplemente se dividen los valores del gráfico por los 1000 patrones, con lo que queda, por ejemplo, para un patrón de largo 5 solo 11,8 milisegundos. Mientras que para un patrón de largo 20 solo toma 1 milisegundo por consulta.

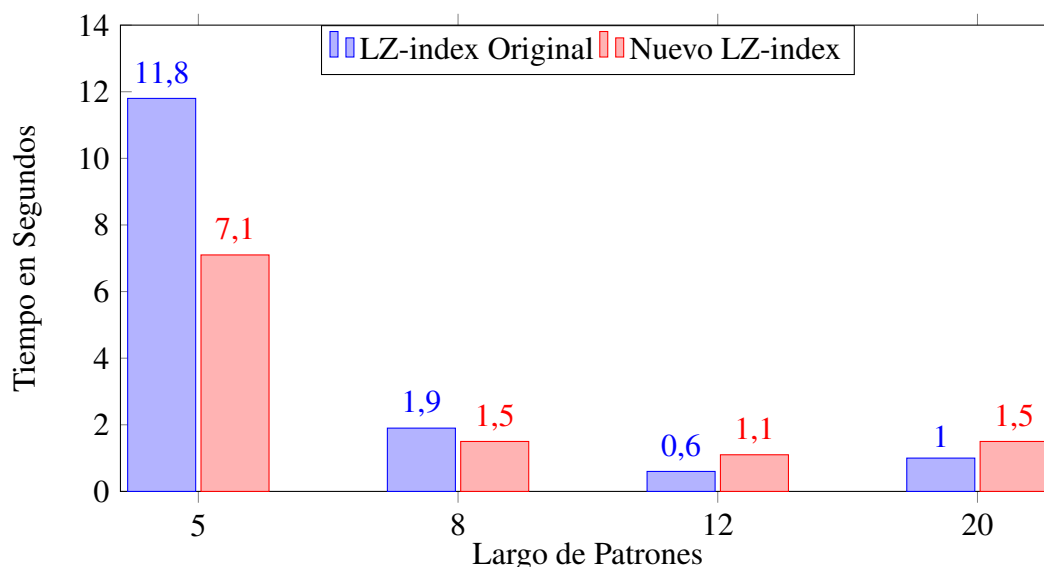


Figura 4.2: Tiempo requerido en contar 1000 patrones de largo 5, 8, 12 y 20 en texto English.200MB con el LZ-index original y versión nueva.

4.3. Secuencia ADN

Espacio utilizado. Para este caso, se puede ver en la Figura 4.3 cómo la compresión del nuevo índice es mejor a la del índice original, logrando una mejora de un 13 % en términos de espacio. Notar que el tamaño utilizado por el índice es cercano al del texto, esto quiere decir que no se necesita de un gran espacio extra y se puede generar búsquedas de patrones de manera rápidas con un LZ-index. Para ser construido, este índice toma 194,55 segundos, mientras que el índice original toma 90,44 segundos.

La información en el Cuadro 4.4, nos indica la cantidad de nodos que ocupa cada uno de los árboles que componen el LZ-index. El RevTrie utiliza solo un 2 % más de nodos que el LZTrie.

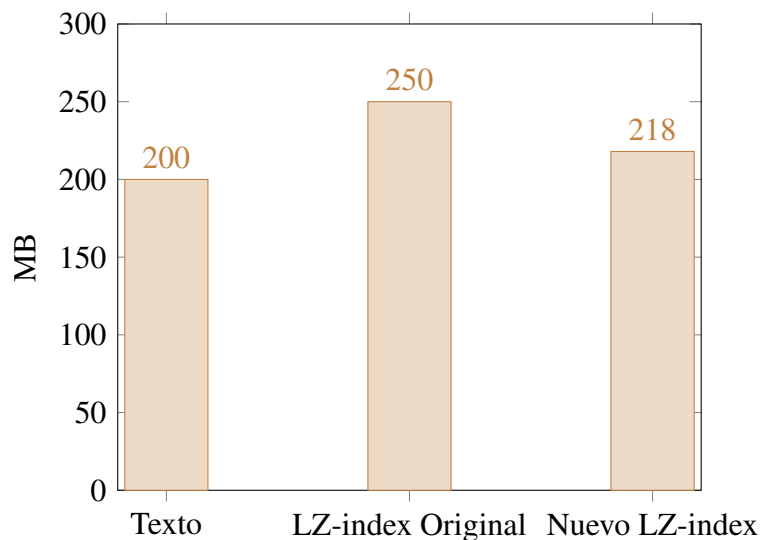


Figura 4.3: Tamaño en MB que utiliza el texto Secuencia de ADN, LZ-index original y el nuevo LZ-index desarrollado en este trabajo.

Cuadro 4.4: Cantidad de nodos en LZTrie y cantidad de nodos en RevTrie para el texto secuencia ADN.

Nodos LZTrie	16.373.738
Nodos RevTrie	16.648.285

Tiempo de Respuesta. La comparación de los tiempos de respuesta de los índices para el caso de este texto se encuentra en la Figura 4.4. El tiempo tomado para encontrar los patrones de largo 5 es 56 % menor con el nuevo índice por lo tanto se logra una mejora considerable para estos patrones. En los patrones de largo 8 y 12 también existe una mejora, la que es de un 60 % y 45 % respectivamente. Los patrones de largo 20 son los únicos donde el índice original es mejor, con un 41 % respecto al nuevo. Los resultados obtenidos son explicados con el Cuadro 4.6, ya que para los patrones de largo 5, 8 y 12 están principalmente concentrados en ocurrencias de tipo 1 y 2. Mientras que para los patrones de largo 20, el 1,65 % son ocurrencias de tipo 3. Estas ocurrencias utilizan dentro de un ciclo la función `ancestor`, que es poco eficiente al ir subiendo un nivel a la vez con la función `find_close`.

Llama la atención la diferencia de tiempo que toma este texto en comparación con el texto English para los patrones de largo 5. El texto English toma en total 11,8 segundos, mientras

que el texto actual toma 148,28 segundos. Esto se debe a la gran diferencia entre las ocurrencias encontradas entre estos dos textos, esta información está en los Cuadros 4.5 y 4.2 donde para los patrones de largo 5, el texto English tiene solo un 18 % de los que tiene el texto Secuencia ADN.

Cuadro 4.5: Cantidad de ocurrencias encontradas para los 1000 patrones de cada largo, para el texto Secuencia ADN.

Largo Patrón	Cantidad de Ocurrencias
5	315.572.958
8	7.931.575
12	471.728
20	108.676

Cuadro 4.6: Porcentaje de tipo de ocurrencias para patrones de distinto largo, para el texto Secuencia ADN.

Largo Patrones	Occ Tipo 1	Occ Tipo 2	Occ Tipo 3
5	70.18 %	29.82 %	0.00 %
8	51.79 %	48.21 %	0.02 %
12	57.41 %	42.51 %	0.08 %
20	50.32 %	48.04 %	1.65 %

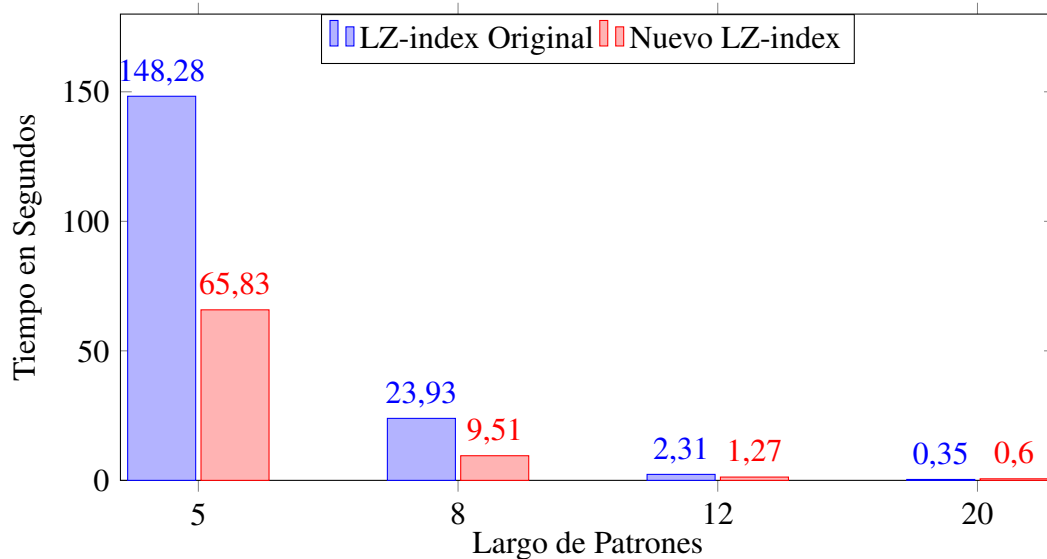


Figura 4.4: Tiempo requerido en contar 1000 patrones de largo 5, 8, 12 y 20 en texto Secuencia ADN con el *LZ-index* original y versión nueva.

4.4. Secuencia MIDI

Espacio utilizado. El espacio utilizado por este texto y los índices se puede ver en la Figura 4.5. El nuevo índice utiliza menos espacio que la versión original y se logra una mejora de un 17 % con respecto al índice original. Notar que en este caso, a diferencia con los textos analizados anteriormente, el nuevo índice utiliza más del doble de espacio que el texto (117 % más) esto significa que la composición de este texto es poco estructurado y tiene una baja repetición de frases del LZ78. Para ser construido, este índice toma 80,29 segundos, mientras que el índice original toma 28,66 segundos.

En el Cuadro 4.7 se encuentra la cantidad de Nodos que componen los Tries de este índice. Sigue la lógica presentada en textos anteriores donde el RevTrie no alcanza a ser su peor caso teórico de $O(2n)$ y solo tiene un 1 % más que el LZTrie.

Tiempo de Respuesta. Los resultados de los experimentos para este texto se puede ver en la Figura 4.6. En este caso el nuevo índice solo logra superar al original para los patrones

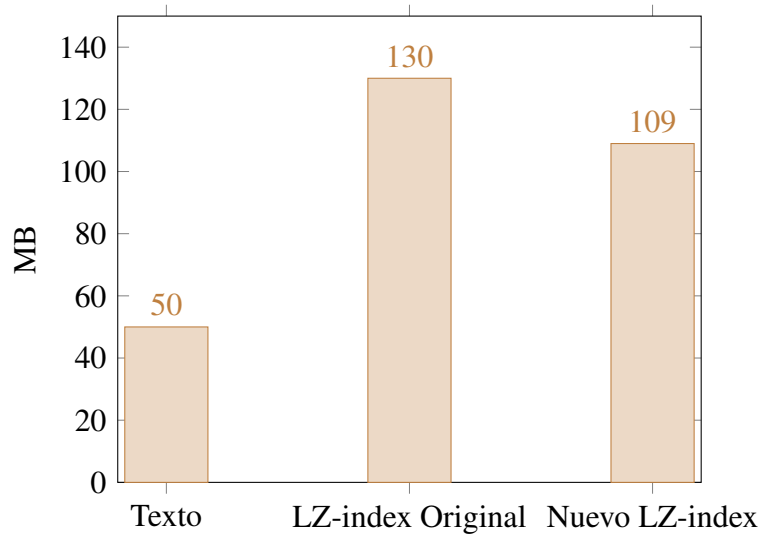


Figura 4.5: Tamaño en MB que utiliza el texto Secuencia MIDI, LZ-index original y el nuevo LZ-index desarrollado en este trabajo.

Cuadro 4.7: Cantidad de nodos en LZTrie y cantidad de nodos en RevTrie para el texto Secuencia MIDI.

Nodos LZTrie	8.293.533
Nodos RevTrie	8.407.649

de largo 5, en los que logra un tiempo 37 % mejor. Esto se debe a que los patrones de este largo no cuenta con una cantidad considerable de ocurrencias de tipo 3, como se puede ver en el Cuadro 4.9. Mientras tanto para el resto de los patrones las ocurrencias de tipo 3 empiezan aparecer paulatinamente y además como indica el Cuadro 4.8, a diferencia de los textos anteriores, la cantidad de ocurrencias encontradas para los distintos largos se mantiene, por lo tanto aunque sea bajo el porcentaje de ocurrencias de tipo 2 y 3, en realidad se está considerando una gran cantidad de ocurrencias.

Cuadro 4.8: Cantidad de ocurrencias encontradas para los 1000 patrones de cada largo, para el texto Secuencia MIDI.

Largo Patrón	Cantidad de Ocurrencias
5	5.612.491
8	5.336.238
12	6.935.597
20	5.975.833

Cuadro 4.9: Porcentaje de tipo de ocurrencias para patrones de distinto largo, para el texto Secuencia MIDI.

Largo Patrones	Occ Tipo 1	Occ Tipo 2	Occ Tipo 3
5	89.71 %	10.29 %	0.00 %
8	93.40 %	6.58 %	0.02 %
12	94.40 %	5.55 %	0.05 %
20	93.17 %	6.71 %	0.12 %

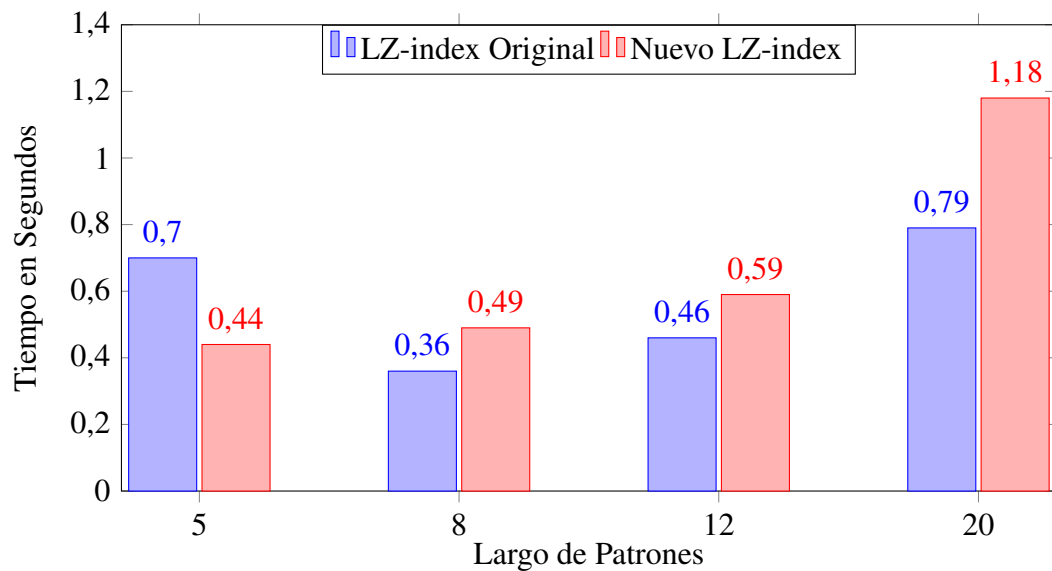


Figura 4.6: Tiempo requerido en Contar 1000 patrones de largo 5, 8, 12 y 20 en Texto Secuencia MIDI Pitches con el *LZ-index* original y versión nueva.

4.5. Archivo XML

Espacio utilizado. El espacio utilizado por este texto en comparación con los índices se puede ver en la Figura 4.7. Este texto es el único que se logra comprimir en un índice que ocupa menos espacio que el texto, esto es logrado tanto en el índice original como en el nuevo. El motivo de esto es debido a que los archivo XML son muy estructurados, lo que conlleva a la alta repetición de frases LZ78. En el Cuadro 4.11 se puede ver como existe una gran cantidad de ocurrencias en patrones de largo 5 y 8, en comparación con los otros textos, como por ejemplo el texto MIDI, que tiene solo un 2 % de la cantidad de ocurrencias de largo 5 que tiene este texto. El nuevo índice ocupa un menor espacio utilizando un 79 % del tamaño del texto, mientras que el índice original un 94 %, por lo tanto, se logra una mejora de un 16 % con el nuevo índice comparando con la implementación anterior. Para ser construido este índice toma 206,90 segundos, mientras que el índice original toma 91,45 segundos.

El RevTrie cuenta con un 3 % más de nodos que el LZTrie, por lo tanto sigue sin estar cercano al peor caso teórico, y está más cerca al límite inferior $\Omega(n)$. Esta información se puede ver en el Cuadro 4.10.

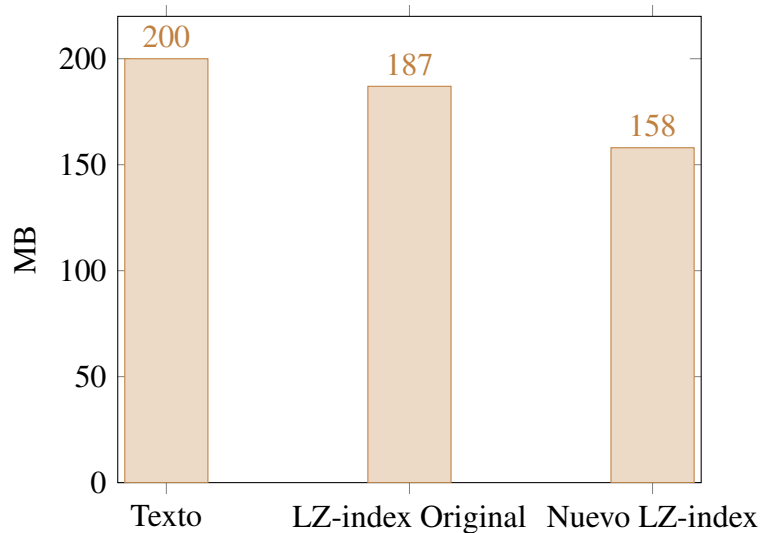


Figura 4.7: Tamaño en MB que utiliza el texto Archivo XML, LZ-index original y el nuevo LZ-index desarrollado en este trabajo.

Tiempo de Respuesta. Está información se encuentra en la Figura 4.8 y se puede observar

Cuadro 4.10: Cantidad de nodos en LZTrie y cantidad de nodos en RevTrie para el Archivo XML.

Nodos LZTrie	11.775.737
Nodos RevTrie	12.088.116

que para los patrones de distintos largos se tiene un mejor tiempo de respuesta con el nuevo índice. Esto se debe a que no hay una gran cantidad de ocurrencias de tipo 3, como se puede ver en el Cuadro 4.12, y además por la forma en como está constituido el texto, donde las frases son repetitivas, el nuevo índice no tiene mayores problemas con las ocurrencias de tipo 2. Esta misma razón es la causante del poco tiempo que demora en contar las ocurrencias a pesar de haber una gran cantidad encontradas (Cuadro 4.11). Por ejemplo, si se compara con el texto de secuencia de ADN existe un 28 % menos de frases del LZ78. Se gana un tiempo de 53 %, 50 % , 40 % y 26 % para los patrones de largo 5, 8, 12 y 20 respectivamente.

Cuadro 4.11: Cantidad de ocurrencias encontradas para los 1000 patrones de cada largo, para el Archivo XML.

Largo Patrón	Cantidad de Ocurrencias
5	295.936.760
8	153.209.461
12	75.581.590
20	18.782.903

Cuadro 4.12: Porcentaje de tipo de ocurrencias para patrones de distinto largo, para el Archivo XML.

Largo Patrones	Occ Tipo 1	Occ Tipo 2	Occ Tipo 3
5	91.68 %	8.32 %	0.00 %
8	85.97 %	14.02 %	0.00 %
12	81.23 %	18.76 %	0.01 %
20	82.18 %	17.72 %	0.10 %

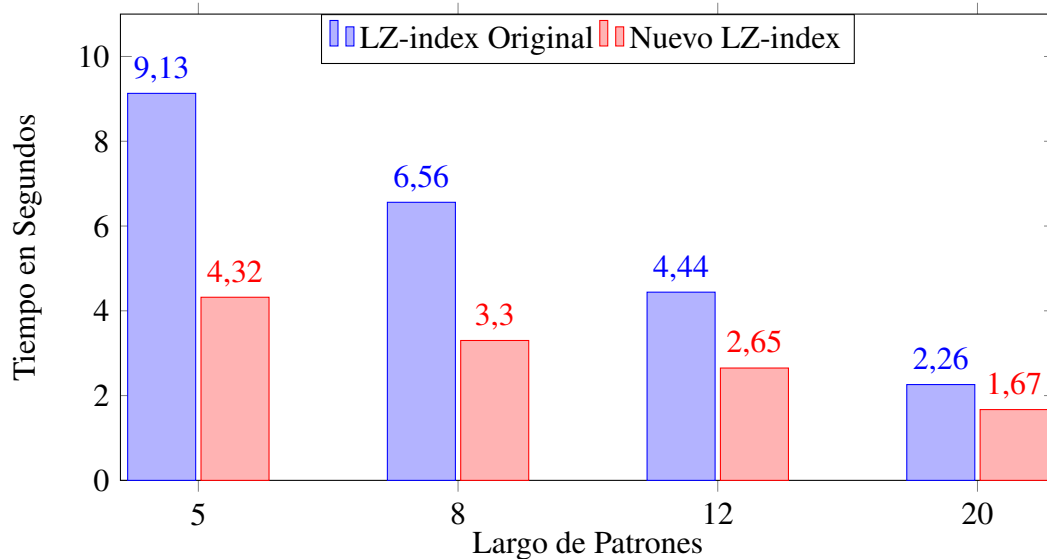


Figura 4.8: Tiempo requerido en contar 1000 patrones de largo 5, 8, 12 y 20 del Archivo XML con el *LZ-index* original y versión nueva.

4.6. Proteínas

Espacio utilizado. El espacio utilizado por este texto se puede ver en la Figura 4.9. A diferencia de los experimentos anteriores el espacio utilizado por los índices es bastante mayor al utilizado por el texto. El nuevo índice utiliza un 207 % de espacio en comparación con el texto. Este fenómeno se debe al contrario de lo que paso con el archivo XML donde existían pocas frases del LZ78, en este caso es el texto que tiene más frases. La cantidad de nodos del LZTrie que está en el Cuadro 4.13, indica también la cantidad de frases del LZ78. Si se compara con los otros textos ya analizados, su más cercano competidor es el texto English que solo cuenta con un 71 % de los nodos que tiene este texto. El texto English es el segundo que más frases tiene, por lo que coincide que es el segundo con índices de mayor tamaño. El nuevo índice sigue utilizando menos espacio con respecto al original, utilizando un 14 % menos. El tiempo de construcción de este índice es de 363,00 segundos, mientras que el índice original toma 130,91 segundos.

Con respecto al Cuadro 4.13 se puede ver nuevamente que la cantidad de nodos del RevTrie

no es ampliamente superior a los del LZTrie, donde solo se tiene un 1 % más.

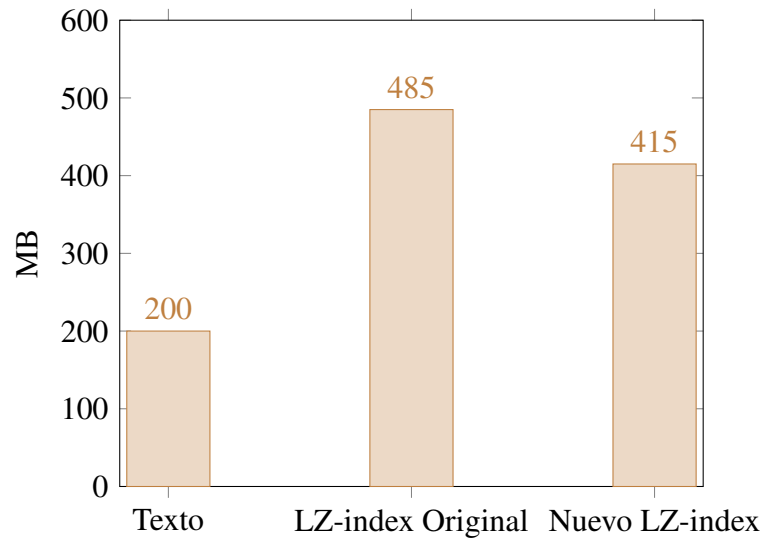


Figura 4.9: Tamaño en MB que utiliza el texto Proteínas, LZ-index original y el nuevo LZ-index desarrollado en este trabajo.

Cuadro 4.13: Cantidad de nodos en LZTrie y cantidad de nodos en RevTrie para el texto Proteínas.

Nodos LZTrie	30.080.187
Nodos RevTrie	30.380.816

Tiempo de Respuesta. El tiempo de respuesta de este texto se puede ver en la Figura 4.10. El nuevo índice solo tiene mejores resultados para los patrones de largo 5 con un 57 % de mejora, mientras que para los patrones de largo 8, 12 y 20 el índice original obtiene mejores resultados. Esto se explica con el Cuadro 4.15, ya que para los patrones donde no se tiene mejora empiezan a aparecer ocurrencias de tipo 3, que como se sabe, estas ocupan la función `ancestor`, que a su vez, utiliza repetidas veces la función `find_close` que para el caso de esta esta implementación se pierde eficiencia al utilizar menos espacio. En el Cuadro 4.14 está la cantidad de ocurrencias encontradas.

Cuadro 4.14: Cantidad de ocurrencias encontradas para los 1000 patrones de cada largo, para el texto Proteínas.

Largo Patrón	Cantidad de Ocurrencias
5	610.353
8	274.119
12	158.504
20	91.106

Cuadro 4.15: Porcentaje de tipo de ocurrencias para patrones de distinto largo, para el texto Proteínas.

Largo Patrones	Occ Tipo 1	Occ Tipo 2	Occ Tipo 3
5	68.55 %	31.45 %	0.00 %
8	75.97 %	23.83 %	0.19 %
12	64.08 %	33.94 %	1.97 %
20	47.68 %	44.32 %	8.00 %

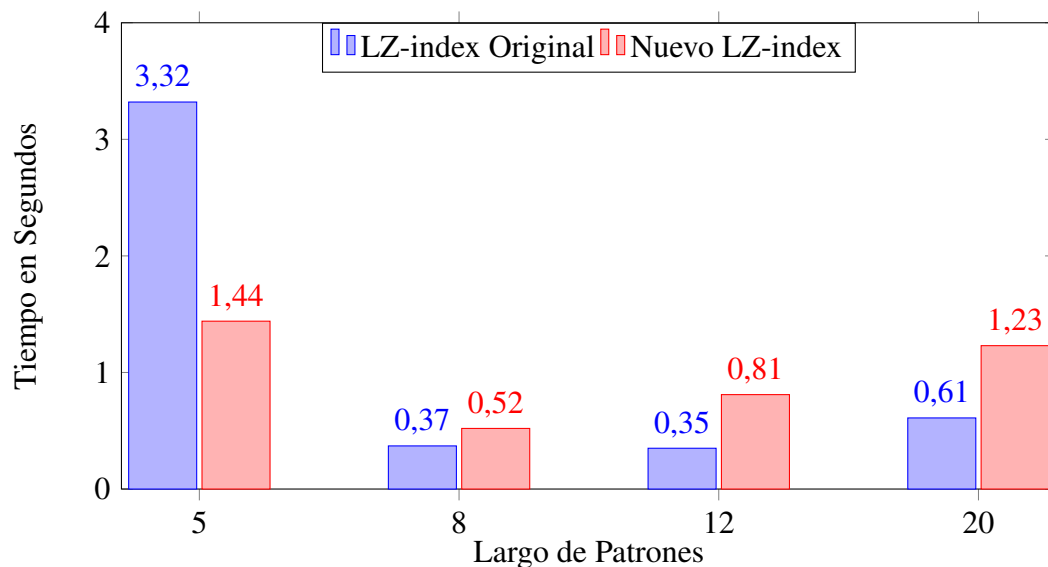


Figura 4.10: Tiempo requerido en contar 1000 patrones de largo 5, 8, 12 y 20 en texto Proteínas con el *LZ-index* original y versión nueva.

4.7. Sources

Espacio utilizado. El espacio utilizado por los índices y el texto se puede ver en la Figura 4.11. Este espacio es muy parecido al conseguido por el texto English en la Figura 4.1, esto se debe ya que la cantidad de frases de la descomposición del LZ78 es similar. Recordar que la cantidad de frases del LZ78 es lo mismo a la cantidad de nodos del LZTrie que se puede ver en el Cuadro 4.16. Para este último texto también se sigue la tendencia que el nuevo índice utiliza un menor espacio, el que es un 15 % menor en comparación con la implementación anterior. El tiempo de construcción de este índice toma 275,00 segundos, mientras que el índice original toma 120,43 segundos.

La cantidad de nodos utilizados por el LZTrie y el RevTrie, como se puede ver en el Cuadro 4.16, es bastante similar donde solo hay un 2 % más de nodos que utiliza el RevTrie.

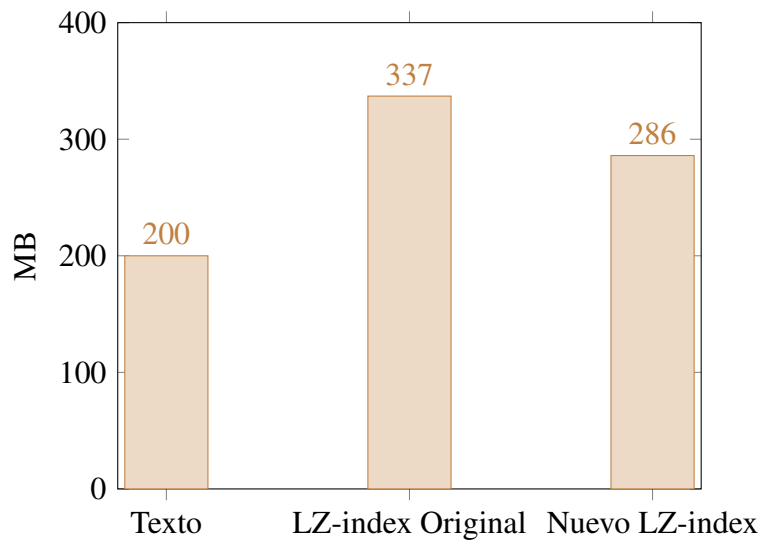


Figura 4.11: Tamaño en MB que utiliza el texto Sources, LZ-index original y el nuevo LZ-index desarrollado en este trabajo.

Cuadro 4.16: Cantidad de nodos en LZTrie y cantidad de nodos en RevTrie para el texto Sources.

Nodos LZTrie	20.587.824
Nodos RevTrie	21.095.589

Tiempo de Respuesta. El tiempo de respuesta se puede ver en el gráfico de la Figura 4.12. En este caso el nuevo índice logra mejores resultados para los patrones de largo 5, 8 y 12. Mientras que para los patrones de largo 20 se llegan a mejores resultados con el índice original. Estos resultados se explican dada la cantidad de ocurrencias que existen para los distintos patrones. En el Cuadro 4.18, se puede ver que solamente existe una cantidad significativa de ocurrencias tipo 3 para los patrones de largo 20, lo que coincide que para estas ocurrencias sean las mas lentas de encontrar con el nuevo índice, debido al alto uso del método ancestro. La cantidad de ocurrencias encontradas se puede ver en el Cuadro 4.17.

Cuadro 4.17: Cantidad de ocurrencias encontradas para los 1000 patrones de cada largo, para el texto Sources.

Largo Patrón	Cantidad de Ocurrencias
5	102.416.863
8	35.742.634
12	9.666.331
20	4.344.635

Cuadro 4.18: Porcentaje de tipo de ocurrencias para patrones de distinto largo, para el texto Sources.

Largo Patrones	Occ Tipo 1	Occ Tipo 2	Occ Tipo 3
5	84.15 %	15.85 %	0.00 %
8	82.54 %	17.46 %	0.00 %
12	84.64 %	15.33 %	0.03 %
20	79.74 %	20.09 %	0.17 %

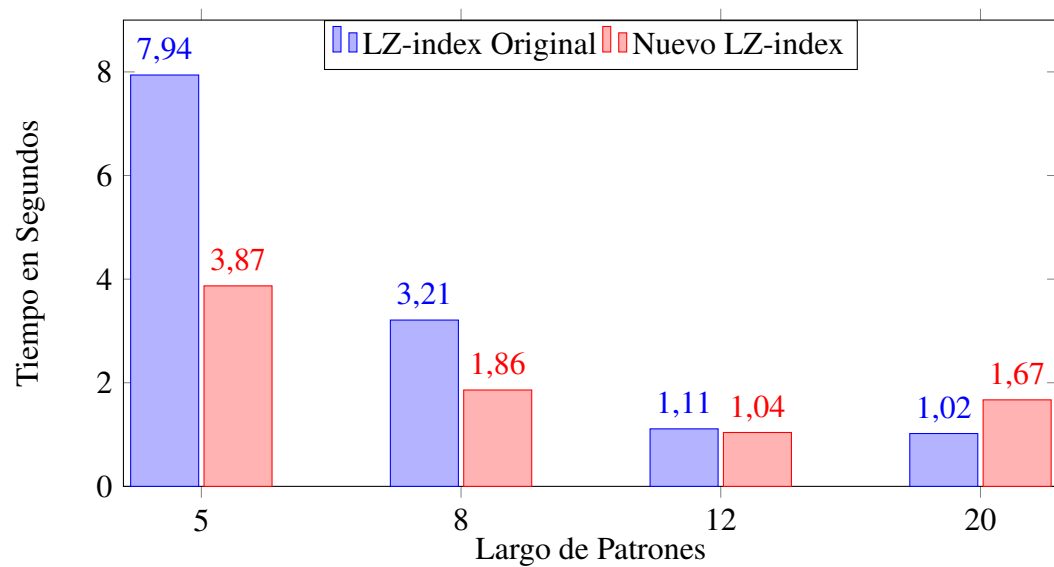


Figura 4.12: Tiempo requerido en contar 1000 patrones de largo 5, 8, 12 y 20 en texto Sources con el *LZ-index* original y versión nueva.

Conclusiones

En este trabajo se rediseña el LZ-index implementado por Navarro [Nav08]. Las principales características del rediseño consistieron en cambiar el lenguaje de programación en el que estaba escrito, evolucionando de C a C++11. Esto permite el uso de la librería SDSL la que permite crear paréntesis balanceados a partir de un *bitvector* y facilita todas las operaciones sobre estos. Una de las principales mejoras que se presentaron con este índice fue la incorporación de la funcionalidad *template* propia de C++. Estos incorporan dos grandes ventajas, primero permiten al usuario que utiliza el índice poder escoger que tipo de paréntesis desea ocupar, lo que es muy útil ya que hay paréntesis balanceados que ocupan un menor espacio a costa de un mayor tiempo de respuesta, como también hay otros que utilizan un mayor espacio pero toman menor tiempo de respuesta en promedio, por lo que queda a gusto del usuario según lo que busca. El segundo parámetro que lo determina el usuario es el largo del alfabeto, esto es otro gran avance dado que las versiones anteriores de este índice solo soportaban alfabetos de tamaño 256 (8 bytes), mientras que en este nuevo índice, este número puede ser modificable como guste el usuario (permite alfabetos de 8, 16, 32 o 64 bytes). Esto permite la incorporación de alfabetos de mayor tamaño como el lenguaje Chino.

Las pruebas se hicieron sobre 6 textos que se pueden encontrar en el repositorio de *Pizza&Chili*, estos textos contienen variadas estructuras por lo que es interesante comprobar como se comporta el índice ante distintos escenarios. El nuevo índice se compara contra el índice de NAV-LZI, ya que estos dos siguen el mismo formato de construcción y la diferencia está en el rediseño mencionado en el párrafo anterior.

En los experimentos, lo primero que se puede observar es la cantidad de nodos que utiliza el *RevTrie*, en teoría este en un peor caso podría llegar a utilizar $O(n^2)$, pero dado que se

eliminan los nodos vacíos su peor caso disminuye considerablemente llegando a $O(2n)$. Sin embargo, en la práctica se puede ver que la cantidad de nodos utilizados por el *RevTrie* no supera por más de un 3 % a los del *LZTrie* (esto quiere decir: $1,03n$). Este premisa es de vital importancia para la construcción de los LZ-index, ya que si el caso promedio de los nodos del *RevTrie* estuvieran más cercano a $O(2n)$ significaría un aumento de espacio considerable.

En términos de espacio, se comprueba que el nuevo índice obtiene una mejora, donde en todos los textos en que se prueba se tiene un espacio por debajo al índice original de NAV-LZI. Esto se debe, puesto que las estructuras de datos provenientes de la librería SDSL logran una disminución en el espacio. El texto en el que se logra una mayor disminución de espacio es en el de MIDI, donde se tiene una reducción de 17 % con respecto a la implementación anterior. Mientras que el texto que menor mejora se obtiene es el de ADN y English con un 13 % con respecto a la implementación anterior.

Siguiendo con el análisis de espacio utilizado, se puede ver que hay textos que se pueden comprimir más que otros. Está el caso de los archivos XML, donde el nuevo índice logra una compresión que utiliza 79 % del texto. Este es el único caso donde el índice utiliza un menor espacio que el texto. La razón de esto es por la estructura del texto que se traduce en la cantidad de frases formadas por el LZ78. Esto se comprueba con la cantidad de nodos creados por el *LZTrie*, que es lo mismo que la cantidad de frases, donde para el caso del archivo XML es el que menos frases contiene de los textos analizados. En caso contrario, el índice que más espacio utiliza es el texto Proteínas. Este texto tiene cerca de 3 veces más la cantidad de frases que las que tiene el archivo XML. El espacio que utiliza el índice para el texto Proteínas es 207 % en comparación con el texto.

Analizando el tiempo de respuesta de los experimentos, se puede ver que los patrones cortos de largo 5, en todos los textos se obtiene una mejora. Esto se debe a que el nuevo diseño del índice se comporta de una buena manera para las ocurrencias de tipo 1, que es donde se ubican en su mayoría los patrones cortos. Cuando existen ocurrencias de tipo 2, el tiempo entre el nuevo índice y el índice original comienzan a equipararse y para las ocurrencias de tipo 3 el nuevo índice se ve superado considerablemente. La razón de este fenómeno está en que para las ocurrencias de tipo 2 se empieza a utilizar con mayor frecuencia la operación `find.close`, y dado que esta bajo una estructura de la librería SDSL que utiliza

menos espacio conlleva a que tome un mayor tiempo de respuesta. Para las ocurrencias de tipo 3, se utiliza para el *LZTrie* y *RevTrie* la función *ancestor*. Esta función ocupa reiteradas veces la función *find_close*, por consecuencia el tiempo de respuesta del nuevo índice para este tipo de ocurrencias sea más lento. Esto se comprueba experimental con el texto XML, donde se obtienen mejores resultados para todos los patrones y coincide que el porcentaje de ocurrencias de tipo 3 es el menor con un 0,10 % . Mientras tanto, para todo el resto de los textos, las ocurrencias de tipo 3 son resueltas en menor tiempo por el índice original. Para el texto English, en los patrones de largo 20 existe un gran porcentaje de ocurrencias tipo 3 con un 18,38 %, lo que implica al índice original tener un 37 % de mejora en el tiempo de respuesta con respecto al nuevo para los patrones de este largo.

Se comprueba que el LZ-index es una herramienta poderosa para encontrar ocurrencias en un texto con un índice comprimido, donde buscar un patrón en promedio toma 2 milisegundos por consulta. Esto responde a la velocidad que buscan los usuarios donde la búsqueda del patrón pasa a ser casi imperceptible. Dado el tamaño de los textos que se tienen hoy en día digitalizados es importante seguir avanzando en esta tarea ya que el uso de estos índices son necesarios por todas las áreas de investigación.

Capítulo 5

Anexos

Cuadro 5.1: Ocurrencias encontradas para el texto English

Largo Patrón	Ocurrencias tipo 1	Ocurrencias tipo 2	Ocurrencias tipo 3	Total
5	40.327.470	15.082.677	198	55.410.345
8	2.376.546	1.398.910	629	3.776.085
12	174.070	76.050	1.218	251.338
20	3.854	3.625	1.684	9.163

Cuadro 5.2: Ocurrencias encontradas para el texto Secuencia de ADN

Largo Patrón	Ocurrencias tipo 1	Ocurrencias tipo 2	Ocurrencias tipo 3	Total
5	221.465.899	94.106.965	95	315.572.959
8	4.107.594	3.823.825	156	7.931.575
12	270.822	200.537	369	471.728
20	54.683	52.205	1.788	108.676

Cuadro 5.3: Ocurrencias encontradas para el texto Secuencia MIDI

Largo Patrón	Ocurrencias tipo 1	Ocurrencias tipo 2	Ocurrencias tipo 3	Total
5	5.034.919	577.535	37	5.612.491
8	4.984.200	351.053	985	5.336.238
12	6.547.435	384.908	3.254	6.935.597
20	5.567.860	400.819	7.154	5.975.833

Cuadro 5.4: Ocurrencias encontradas para Archivo XML

Largo Patrón	Ocurrencias tipo 1	Ocurrencias tipo 2	Ocurrencias tipo 3	Total
5	271.306.233	24.629.835	692	295.936.760
8	131.720.831	21.485.496	3.134	153.209.461
12	61.395.096	14.178.210	8.284	75.581.590
20	15.435.588	3.328.723	18.592	18.782.903

Cuadro 5.5: Ocurrencias encontradas para el texto Proteínas

Largo Patrón	Ocurrencias tipo 1	Ocurrencias tipo 2	Ocurrencias tipo 3	Total
5	418.412	191.936	5	610.353
8	208.261	65.329	529	274.119
12	101.573	53.803	3.128	158.504
20	43.439	40.377	7.290	91.106

Cuadro 5.6: Ocurrencias encontradas para el texto Sources

Largo Patrón	Ocurrencias tipo 1	Ocurrencias tipo 2	Ocurrencias tipo 3	Total
5	86.178.833	16.237.757	273	102.416.863
8	29.502.198	6.239.151	1.285	35.742.634
12	8.181.120	1.482.026	3.185	9.666.331
20	3.464.438	872.746	7.451	4.344.635

Bibliografía

- [Bil09] Diego Arroyuelo Billiardi. Autoíndices comprimidos para texto basados en lempel-ziv. *Universidad de Chile, Departamento de Ciencias de la Computación*, 2009.
- [BW94] M. Burrows and D. J. Wheeler. *A block-sorting lossless data compression algorithm*. Technical Report 124, Digital Equipment Corporation, 1994.
- [CM99] D. Clark and J. I. Munro. Efficient suffix trees on secondary storage. *Proc. 7th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 1999.
- [FG99] P. Ferragina and R. Grossi. The string b-tree: a new data structure for string search in external memory and its applications. *Journal of the ACM*, 1999.
- [FM05] P. Ferragina and G. Manzini. Indexing compressed text. *Journal of the ACM*, 54(4):552–581, 2005.
- [FM07] G. Franceschini and S. Muthukrishnan. In-place suffix sorting. In *Proc. 34th International Colloquium on Automata, Languages and Programming (ICALP)*, LNCS 4596, pages 533–546, 2007.
- [GN08] R. Gonzalez and G. Navarro. Rank/select on dynamic compressed sequences and applications. *Theoretical Computer Science*, 2008.
- [Gog] Simon Gog. <https://github.com/simongog/sdsl-lite>.
- [IWB99] A. Moffat I. Witten and T. Bell. Managing gigabytes. *Morgan Kaufmann Publishers, New York*, 1999.
- [KM99] R. Kosaraju and G. Manzini. Compression of low entropy strings with lempel-ziv algorithms. *SIAM Journal on Computing*, 29(3):893–911, 1999.
- [Knu73] D. Knuth. The art of computer programming – volume 3: Sorting and searching. *Addison-Wesley*, 1973.
- [KSY07] W.K. Sung K. Sadakane and M. Yiu. A space and time efficient algorithm for constructing compressed suffix arrays. *Algorithmica*, 2007.

- [KU96] J. Kärkkäinen and E. Ukkonen. Lempel-ziv parsing and sublinear-size index structures for string matching. *Proc. 3rd South American Workshop on String Processing (WSP)*, pages 141–155, 1996.
- [Kä99] J. Kärkkäinen. *Repetition-based text indexes*. PhD thesis, Department of Computer Science, University of Helsinki, Finland, 1999.
- [LZ76] A. Lempel and J. Ziv. On the complexity of finite sequences. *IEEE Transactions on Information Theory*, 22(1):75–81, 1976.
- [Man01] G. Manzini. An analysis of the burrows-wheeler transform. *Journal of the ACM*, 48(3):407–430, 2001.
- [McC76] Edward M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976.
- [MF04] G. Manzini and P. Ferragina. Engineering a lightweight suffix array construction algorithm. *Algorithmica*, 40(1):33–50, 2004.
- [MFS84] J. Komlos M. Fredman and E. Szemeréd. Storing a sparse table with $o(1)$ worst case access time. *Journal of the ACM*, 31(3):538–544, 1984.
- [MM93] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.
- [MR01] J. I. Munro and V. Raman. Succinct representation of balanced parentheses and static trees. *SIAM J. Computing*, page 762–776, 2001.
- [MS77] Boyer Moore and Robert Strother. A fast string searching algorithm. *Association for Computing Machinery*, 1977.
- [Nav04] G. Navarro. Indexing text using the ziv-lempel trie. *Journal of Discrete Algorithms (JDA)*, 2(1):87–114, 2004.
- [Nav08] G. Navarro. Implementing the lz-index: Theory versus practice. *ACM Journal of Experimental Algorithmics (JEA)*, 13:article 1.2, 49 pages, 2008.
- [NM07] G. Navarro and V. Makinen. Compressed full-text indexes. *ACM Computing Surveys*, 2007.
- [NP07] J. Na and K. Park. Alphabet-independent linear-time construction of compressed suffix arrays using $o(n \log n)$ -bit working space. *Theoretical Computer Science*, 2007.
- [piz] <http://pizzachili.dcc.uchile.cl/>.
- [Ram96] R. Raman. Priority queues: small, monotone and trans-dichotomous. *Proc. 4th European Symposium on Algorithms (ESA)*, page 121–137, 1996.

- [RO07] L. Russo and A. Oliveira. A compressed self-index using a ziv-lempel dictionary. *Journal of Information Retrieval*, 5(3):501–513, 2007.
- [TBW90] J. Cleary T. Bell and I. Witten. Text compression. *Prentice–Hall*, 1990.
- [TCS01] R. Rivest T. Cormen, C. Leiserson and C. Stein. Introduction to algorithms. *Prentice–Hall*, 2001.
- [Wei73] P. Weiner. Linear pattern matching algorithm. *14th Annual IEEE Symposium on Switching and Automata Theory*, pages 1–11, 1973.
- [WHS03] K. Sadakane W.K. Hon and W.K. Sung. Breaking a time-and-space barrier in constructing full-text indices. *Proc. 44th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, 2003.
- [ZL77] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 22(3):337–343, 1977.
- [ZL78] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.