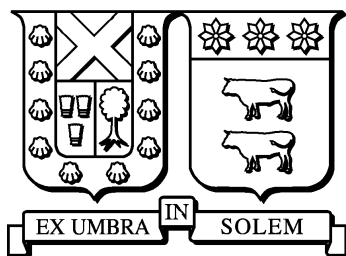


UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA

DEPARTAMENTO DE INFORMÁTICA
SANTIAGO – CHILE



“IMPLEMENTACIÓN DE ÁRBOLES CARDINALES COMPACTOS”

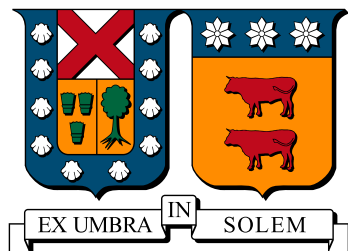
NICOLÁS ANDRÉS GONZÁLEZ GÓMEZ

MEMORIA DE TITULACIÓN PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL INFORMÁTICO

PROFESOR GUÍA: DIEGO ARROYUELO B.

AGOSTO 2017

UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE INFORMÁTICA
SANTIAGO – CHILE



**“IMPLEMENTACIÓN DE ÁRBOLES
CARDINALES COMPACTOS”**

NICOLÁS ANDRÉS GONZÁLEZ GÓMEZ

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

PROFESOR GUÍA: DIEGO ARROYUELO B.
PROFESOR CORREFERENTE: PEDRO GODOY B.

AGOSTO 2017

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

Resumen

Esta memoria aborda el área de las estructuras de datos compactas, teniendo como objetivo principal construir un árbol cardinal compacto. Para ello se utilizan secuencias generales que soportan operaciones rank y select. Las estructuras que se estudiarán, con el fin de construir un árbol cardinal compacto son: **Golynski, Alphabet Partitioning, Wavelet Tree, Huffman Shaped Wavelet Tree**. Además de estas estructuras, se utilizarán algoritmos de búsqueda como búsqueda lineal y búsqueda binaria. De esta manera se pretende determinar la mejor manera de construir árboles cardinales compactos.

Abstract

This work is related to succinct data structures, having as principal goal to build a cardinal succinct tree. To make this possible, are used general sequences, that support operations like rank and select. The structures that are will be study, with the goal to build a cardinal succinct tree are: **Golynski**, **Alphabet Partitioning**, **Wavelet Tree**, **Huffman Shaped Wavelet Tree**. Besides this structures, are will be used search algorithms like linear search and binary search. In this way, it is intended to determine the best way to build a cardinal succinct tree.

Índice de Contenidos

Resumen	III
Abstract	IV
Índice de Contenidos	V
Lista de Tablas	VII
Lista de Figuras	X
Glosario	XII
1. Introducción	1
Introducción	1
1.1. Definición del Problema	2
1.1.1. Objetivos	2
1.2. Estado del Arte	3
1.2.1. Marco Teórico	3
1.2.2. Árboles	3
1.2.3. Árboles Cardinales	3
1.2.4. Estructuras De Datos Compactas/Sucintas	4
1.2.5. Conceptos Preliminares	5
1.2.6. Árboles Cardinales Compactos	7
1.2.7. Árboles Generales	8
1.2.8. Representación De Árboles Generales	8
1.2.9. Wavelet Trees	9
1.2.10. Otras Estructuras	12

1.2.11. Entropía de Orden 0 - H_0	13
1.2.12. Tasa De Compresión	13
2. Propuesta	15
2.1. Implementación	17
2.1.1. Consideraciones Previas	17
2.1.2. Implementación de Árboles Cardinales	18
2.1.3. Implementación de Operaciones	19
2.1.4. Operaciones Básicas Sobre Estructura de Paréntesis Balanceados	19
2.1.5. Operaciones Sobre Estructura De Paréntesis Balanceados	20
2.1.6. Operaciones Básicas Sobre Estructura de Símbolos (Rank/Select)	25
2.1.7. Operaciones Sobre Estructura de Símbolos (Rank/Select)	26
2.1.8. Operaciones Sobre Secuencia de Símbolos sin Estructuras Adicionales	27
3. Experimentación	30
3.1. Sistema y Método de Experimentación	33
3.2. Resultados Experimentales	34
3.2.1. Árboles con Alfabetos Pequeños	34
3.2.2. Árboles con alfabetos grandes	41
Conclusiones	49
4. Apéndice	53
4.1. Uso de Espacio	53
4.2. Tablas De Rendimiento para Operación label-child	61
4.3. Gráficos Tiempo Vs Espacio	65
Bibliografía	72

Índice de cuadros

1.1. Operaciones objetivo a implementar	5
1.2. Resumen complejidades	13
2.1. Secuencia degree de ejemplo.	20
3.1. Descripción de los Árboles	31
3.2. Características de los Árboles.	32
3.3. Entropía H_0 de cada árbol.	33
3.4. Especificaciones del sistema utilizado.	33
3.5. El término teórico $2n$ en el uso de espacio equivale a 95.892.334 bits, mientras que $n \lg \sigma$ es 375.253.001 bits. Esto da un total teórico de 9,8265 bits por nodo. Por otro lado, $H_0 = 5,03$, por lo que $nH_0 = 241,305,387$. Esto equivale a 7,0328 bits por nodo en teoría.	35
3.6. El término teórico $2n$ en el uso de espacio equivale a 60.272.746 bits, mientras que $n \lg \sigma$ es 123.181.305 bits. Esto da un total teórico de 6,0875 bits por nodo. Por otro lado, $H_0 = 1,99$, por lo que $nH_0 = 60,120,256$. Esto equivale a 3,9949 bits por nodo en teoría.	36
3.7. El término teórico $2n$ en el uso de espacio equivale a 41.387.962 bits, mientras que $n \lg \sigma$ es 162.483.945 bits. Esto da un total teórico de 9,8517 bits por nodo. Por otro lado, $H_0 = 5,83$, por lo que $nH_0 = 120,576,171$. Esto equivale a 7,8266 bits por nodo en teoría.	37
3.8. El término teórico $2n$ en el uso de espacio equivale a 908.002.948 bits, mientras que $n \lg \sigma$ es 9.049.121.961 bits. Esto da un total teórico de 21,9319 bits por nodo. Por otro lado, $H_0 = 19,93$, por lo que $nH_0 = 9,049,021,179$. Esto equivale a 21,9317 bits por nodo en teoría.	42
3.9. El término teórico $2n$ en el uso de espacio equivale a 79.138.862 bits, mientras que $n \lg \sigma$ es 853.744.469 bits. Esto da un total teórico de 23,5759 bits por nodo. Por otro lado, $H_0 = 15,25$, por lo que $nH_0 = 603,520,875$. Esto equivale a 17,2522 bits por nodo teórico.	43

3.10.	El término teórico $2n$ en el uso de espacio equivale a 161.728.612 bits, mientras que $n \lg \sigma$ es 1.806.433.882 bits. Esto da un total teórico de 24,3391 bits por nodo. Por otro lado, $H_0 = 15,34$, por lo que $nH_0 = 1,240,450,368$. Esto equivale a 17,2522 bits por nodo teórico.	44
4.1.	El término teórico $2n$ en el uso de espacio equivale a 129.182.368 bits, mientras que $n \lg \sigma$ es 307.123.814 bits. Esto da un total teórico de 6,7549 bits por nodo. Por otro lado, $H_0 = 4,27$, por lo que $nH_0 = 276,029,133$. Esto equivale a 6.2735 bits por nodo en teoría.	53
4.2.	El término teórico $2n$ en el uso de espacio equivale a 95.892.334 bits, mientras que $n \lg \sigma$ es 375.253.001 bits. Esto da un total teórico de 9,8265 bits por nodo. Por otro lado, $H_0 = 5,03$, por lo que $nH_0 = 241,305,387$. Esto equivale a 7,0328 bits por nodo en teoría.	54
4.3.	El término teórico $2n$ en el uso de espacio equivale a 60.272.746 bits, mientras que $n \lg \sigma$ es 123.181.305 bits. Esto da un total teórico de 6,0875 bits por nodo. Por otro lado, $H_0 = 1,99$, por lo que $nH_0 = 60,120,256$. Esto equivale a 3,9949 bits por nodo en teoría.	55
4.4.	El término teórico $2n$ en el uso de espacio equivale a 41.387.962 bits, mientras que $n \lg \sigma$ es 162.483.945 bits. Esto da un total teórico de 9,8517 bits por nodo. Por otro lado, $H_0 = 5,83$, por lo que $nH_0 = 120,576,171$. Esto equivale a 7,8266 bits por nodo en teoría.	56
4.5.	El término teórico $2n$ en el uso de espacio equivale a 32.410.344 bits, mientras que $n \lg \sigma$ es 107.192.511 bits. Esto da un total teórico de 8,6147 bits por nodo. Por otro lado, $H_0 = 5,72$, por lo que $nH_0 = 92,616,933$. Esto equivale a 7,7153 bits por nodo en teoría.	57
4.6.	El término teórico $2n$ en el uso de espacio equivale a 17.470.206 bits, mientras que $n \lg \sigma$ es 61.628.619 bits. Esto da un total teórico de 9,0553 bits por nodo. Por otro lado, $H_0 = 5,76$, por lo que $nH_0 = 50,316,989$. Esto equivale a 7,7603 bits por nodo en teoría.	58
4.7.	El término teórico $2n$ en el uso de espacio equivale a 718.817.500 bits, mientras que $n \lg \sigma$ es 4.416.311.347 bits. Esto da un total teórico de 14,2877 bits por nodo. Por otro lado, $H_0 = 12,29$, por lo que $nH_0 = 4,415,983,430$. Esto equivale a 14,2868 bits por nodo en teoría. . .	58
4.8.	El término teórico $2n$ en el uso de espacio equivale a 662.825.598 bits, mientras que $n \lg \sigma$ es 5.173.234.641 bits. Esto da un total teórico de 17,6096 bits por nodo. Por otro lado, $H_0 = 15,60$, por lo que $nH_0 = 5,169,741,393$. Esto equivale a 17,5991 bits por nodo en teoría. . .	59
4.9.	El término teórico $2n$ en el uso de espacio equivale a 908.002.948 bits, mientras que $n \lg \sigma$ es 9.049.121.961 bits. Esto da un total teórico de 21,9319 bits por nodo. Por otro lado, $H_0 = 19,93$, por lo que $nH_0 = 9,049,021,179$. Esto equivale a 21,9317 bits por nodo en teoría. . .	59
4.10.	El término teórico $2n$ en el uso de espacio equivale a 9.711.312 bits, mientras que $n \lg \sigma$ es 93.902.790 bits. Esto da un total teórico de 21,3388 bits por nodo. Por otro lado, $H_0 = 14,83$, por lo que $nH_0 = 71,997,239$. Esto equivale a 16,8275 bits por nodo en teoría.	60

4.11. El término teórico $2n$ en el uso de espacio equivale a 79.138.862 bits, mientras que $n \lg \sigma$ es 853.744.469 bits. Esto da un total teórico de 23,5759 bits por nodo. Por otro lado, $H_0 = 15, 25$, por lo que $nH_0 = 603,520,875$. Esto equivale a 17,2522 bits por nodo en teoría. . . .	60
4.12. El término teórico $2n$ en el uso de espacio equivale a 161.728.612 bits, mientras que $n \lg \sigma$ es 1.806.433.882 bits. Esto da un total teórico de 24,3391 bits por nodo. Por otro lado, $H_0 = 15, 34$, por lo que $nH_0 = 1,240,450,368$. Esto equivale a 17,2522 bits por nodo en teoría. . .	61
4.13. Rendimiento árbol Proteins.	61
4.14. Rendimiento árbol English.	62
4.15. Rendimiento árbol DNA.	62
4.16. Rendimiento árbol Sources.	62
4.17. Rendimiento árbol XML.	63
4.18. Rendimiento árbol MIDI.	63
4.19. Rendimiento árbol Random5Kv1K.	63
4.20. Rendimiento árbol Random50Kv1K.	64
4.21. Rendimiento árbol Random1Mv200K.	64
4.22. Rendimiento árbol Wiki-50MB.	64
4.23. Rendimiento árbol Wiki-500MB.	65
4.24. Rendimiento árbol Wiki-1GB.	65

Índice de figuras

1.1. Árbol cardinal de ejemplo.	4
1.2. Árbol general de ejemplo.	8
1.3. Ejemplo de Wavelet Tree	10
2.1. Diagrama Árbol Cardinal.	16
2.2. Árbol cardinal de ejemplo.	17
3.1. Gráfico Tiempo vs Espacio - Árbol DNA.	38
3.2. Gráfico Tiempo vs Espacio - Árbol XML.	39
3.3. Gráfico Tiempo vs Espacio - Árbol Sources.	40
3.4. Gráfico Tiempo vs Espacio - Árbol Random1Mv200K. El punto referente a LS (34,5288; 241,2730), se excluyó debido a que escapa de las dimensiones del gráfico.	46
3.5. Gráfico Tiempo vs Espacio - Árbol Wikipedia-500MB. El punto referente a WTH (85,9200; 4,1112), se excluyó debido a que escapa de las dimensiones del gráfico.	46
3.6. Gráfico Tiempo vs Espacio - Árbol Wikipedia-1GB. El punto referente a WTH (75,7711; 5,1224), se excluyó debido a que escapa de las dimensiones del gráfico.	47
4.1. Gráfico Tiempo vs Espacio - Árbol Proteins.	66
4.2. Gráfico Tiempo vs Espacio - Árbol English.	66
4.3. Gráfico Tiempo vs Espacio - Árbol DNA.	67
4.4. Gráfico Tiempo vs Espacio - Árbol Sources.	67
4.5. Gráfico Tiempo vs Espacio - Árbol XML.	68
4.6. Gráfico Tiempo vs Espacio - Árbol MIDI.	68

4.7. Gráfico Tiempo vs Espacio - Árbol Random5Kv1K.	69
4.8. Gráfico Tiempo vs Espacio - Árbol Random50Kv1K.	69
4.9. Gráfico Tiempo vs Espacio - Árbol Random1Mv200K. El punto referente a LS (34,5288; 241,2730), se excluyó debido a que escapa de las dimensiones del gráfico.	70
4.10. Gráfico Tiempo vs Espacio - Árbol Wikipedia-50MB. Los puntos referentes a LS (34,5495; 323,3500) y WTH (129,5360;2,8792), se excluyeron debido a que escapan de las dimensiones del gráfico.	70
4.11. Gráfico Tiempo vs Espacio - Árbol Wikipedia-500MB. El punto referente a WTH (85,9200; 4,1112), se excluyó debido a que escapa de las dimensiones del gráfico.	71
4.12. Gráfico Tiempo vs Espacio - Árbol Wikipedia-1GB. El punto referente a WTH (75,7711; 5,1224), se excluyó debido a que escapa de las dimensiones del gráfico.	71

Glosario

- LS: Linear Search (Búsqueda Lineal).
- BS: Binary Search (Búsqueda Binaria).
- WT: Wavelet Trees.
- WTH: Huffman Shaped Wavelet Trees.
- GMR: Estructura de Golynski.
- AP: Alphabet Partitioning.

Capítulo 1

Introducción

En los últimos años el aumento de la cantidad de información ha crecido en forma exponencial. En un reporte de IBM el año 2011 se asegura que "Cada día, creamos 2.5 quintillones de bytes en datos. El 90 % de los datos en el mundo hoy, fueron creados solo en los últimos 2 años" [1].

Esto genera nuevos desafíos en el área de la informática, que busca poder procesar esta cantidad de información en tiempos que se consideren prácticos, además de lograr almacenarla en poco espacio, con el fin de hacerlas lo suficientemente pequeñas para procesarlas en lo más alto de la jerarquía de memorias.

Las estructuras de datos tienen un rol que es crucial en la forma en que se maneja esta información, teniendo una incidencia directa en qué tan rápido se puede acceder a ésta, o cuánto se logra disminuir el espacio utilizado por esta información.

La respuesta se halla en las estructuras de datos compactas, que presentan nuevos desafíos en cuanto a las estructuras de datos clásicas se refiere. Éstas persiguen modificar las estructuras clásicas para utilizar poco espacio, reteniendo su funcionalidad.

Para entrar en contexto y lograr dimensionar el correcto uso de la estructuras de datos y entender el porqué la necesidad de estar progresivamente mejorándolas, se presentará un ejemplo.

Es sabido que la web presenta un gran tamaño, y es representado mediante un grafo dirigido. Al año 2004 el grafo contenía 11.5 mil millones de nodos y 150 mil millones de links. Almacenar solo la estructura de este grafo, toma aproximadamente 600GB. Al utilizar estructuras de datos sucintas se logra almacenar utilizando apenas 100 GB [2].

En consecuencia, la relevancia de las estructuras de datos sucintas es cada vez mayor debido a la tasa de creación de datos del mundo.

Los árboles cardinales, en particular, se utilizan principalmente para realizar búsquedas. Una aplicación común de los árboles cardinales, es utilizarlos para almacenar texto predictivo o diccionarios de autocompletado, mediante la búsqueda de prefijos en el árbol. Además, los árboles cardinales son muy adecuados para

algoritmos de búsqueda aproximada en texto [3].

En las siguientes secciones de esta memoria se describirá el estudio realizado a los **árboles cardinales**, que son la estructura de datos sucinta objetivo en esta investigación.

1.1. Definición del Problema

Como se mencionó anteriormente, las estructuras de datos sucintas son un área de la informática, que busca representar la información de manera comprimida, pero que a la vez permita realizar operaciones de acceso a esta información en forma eficiente, es decir, sin perder su funcionalidad.

Se abordará lo que se denominan **árboles cardinales**, que será explicado posteriormente. Lo que se busca es realizar implementaciones de árboles cardinales, variando las estructuras de datos que se utilizan para implementar este árbol.

Aunque la representación de árboles de forma sucinta ya se ha abordado en distintos trabajos, en cuanto a la implementación de árboles cardinales no hay trabajos relacionados, además las operaciones que soportan los **árboles cardinales** son distintas a otros tipos de árboles generales, por ello el interés de representarlas y observar su comportamiento en la práctica.

1.1.1. Objetivos

El principal objetivo de esta memoria es comparar en cuanto a eficiencia, en espacio utilizado y tiempo de operación/respuesta, las distintas implementaciones sucintas que se proponen de árboles cardinales, en base a las estructuras utilizadas.

Objetivos Específicos

A continuación se descomponen los objetivos que se persigue cumplir en esta memoria:

- Realizar implementación de árbol cardinal en C++.
- Implementar las operaciones a soportar por el árbol cardinal (Ver Tabla 1.1).
- Realizar tests sobre la implementación de árbol cardinal instanciando las distintas estructuras soportadas.
 - Wavelet Trees.
 - Huffman Shaped Wavelet Trees.
 - Golynski et al.
 - Alphabet Partitioning.

- Búsqueda Lineal.
- Búsqueda Binaria.

1.2. Estado del Arte

1.2.1. Marco Teórico

En esta sección se dará la base teórica que es necesaria para comprender el trabajo realizado. En particular los conceptos, definiciones y estructuras que se utilizan para lograr la implementación de un árbol cardinal.

Se describirá qué es un árbol cardinal así como las formas que existen para representar árboles y estructuras sucintas como los diccionarios indexables, entre otros contenidos.

1.2.2. Árboles

Un árbol es una colección de elementos llamados *nodos*, uno de los cuales se diferencia como nodo *raíz*, junto con una relación (padre) que sitúa una estructura jerárquica en los nodos. Un nodo, puede ser del tipo de dato que se desee. Con frecuencia, los nodos corresponden a letras, string, o números con un círculo alrededor. (Ver Figura 1.2).

1.2.3. Árboles Cardinales

En la Figura 1.1 se muestra un ejemplo de un *árbol cardinal k-ario*, el cuál es un árbol con raíz tal que cumple las siguientes condiciones:

1. Cada nodo del árbol tiene grado máximo k (i.e., tiene como máximo k hijos).
2. Los hijos de un nodo están ordenados, por lo tanto se identifica el primer hijo, segundo hijo y así sucesivamente.
3. Cada arco del árbol está etiquetado con un único símbolo perteneciente al alfabeto $\Sigma = \{\alpha_1, \dots, \alpha_k\}$. Además, si se define $\mathbf{label}(x, i)$ como el rótulo del arco que conecta el nodo x con su i -ésimo hijo, se cumple además que:

$$\mathbf{label}(x, i) < \mathbf{label}(x, j) \iff i < j.$$

Es decir, las etiquetas de los enlaces a los hijos mantienen un orden como el descrito.

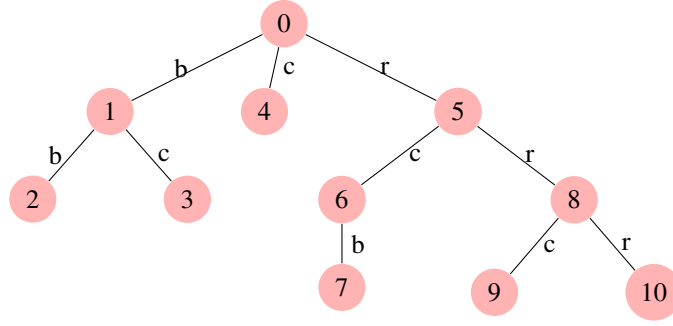


Figura 1.1: Árbol cardinal de ejemplo.

1.2.4. Estructuras De Datos Compactas/Sucintas

Una *estructura de datos sucinta* es una que requiere espacio cercano al *mínimo teórico requerido*. Para aclarar esta definición se tiene por ejemplo que hay $\frac{1}{n+1} \binom{2n}{n}$ árboles binarios distintos de n nodos. En consecuencia, se necesitan al menos $\log_2 \frac{1}{n+1} \binom{2n}{n} = 2n - O(\lg n)$ bits para diferenciar un árbol en particular de los demás.

Las representaciones sucintas, además de requerir un espacio reducido, en general mantienen la funcionalidad de operaciones tan eficientemente como sus contrapartes no-sucintas.

Dentro de las estructuras de datos sucintas existen 3 categorías en las cuales se pueden clasificar las representaciones logradas.

Considerando que Φ es el número de bits óptimo que es necesario para almacenar cierta cantidad de datos, se definen las siguientes categorías:

- Estructuras de Datos Implícitas: Éstas utilizan $\Phi + O(1)$ bits. Esto es, espacio óptimo más una cantidad de bits de orden $O(1)$. Es la más difícil de conseguir. Un ejemplo importante son los heaps o colas de prioridad.
- Estructuras de Datos Sucintas: Utilizan $\Phi + o(\Phi)$. Dentro de esta categoría entra la mayoría de estructuras sucintas desarrolladas.
- Estructuras de Datos Compactas: El espacio utilizado por estas estructuras es del orden de $O(\Phi)$ bits.

Por ejemplo, una estructura de datos que utiliza 2Φ bits de almacenamiento es compacta. Si utiliza $\Phi + \sqrt{\Phi}$ bits es sucinta, si utiliza $\Phi + \lg \Phi$ bits también es sucinta y por último, si utiliza $\Phi + 5$ bits es implícita.

En el caso de los árboles, se busca una representación que pueda ser navegable, y en este trabajo el interés se centra en las operaciones definidas en la Tabla 1.1.

Tabla 1.1: Operaciones objetivo a implementar

$\text{child}(x, i)$:	i -ésimo hijo del nodo x .
$\text{label-child}(x, \alpha)$:	Hijo del nodo x con el símbolo $\alpha \in \{1, \dots, k\}$.
$\text{parent}(x)$:	Padre del nodo x .
$\text{child-rank}(x)$:	Posición del nodo x con respecto a sus hermanos.
$\text{label}(x, i)$:	Etiqueta del i -ésimo hijo del nodo x .
$\text{degree}(x)$:	Número de hijos del nodo x .
$\text{subtree-size}(x)$:	Tamaño del sub-árbol con raíz en el nodo x .
$\text{preorder}(x)$:	Número en pre-orden del nodo x .
$\text{selectnode}(j)$:	Obtiene el nodo con número pre-orden j .
$\text{ancestor}(x, y)$:	Verdadero si el nodo x es ancestro del nodo y . Falso si no.
$\text{access-data}(x)$:	Obtiene los datos asociados al nodo x .

Para los árboles cardinales la operación más típica corresponde a *label-child*, dado que los árboles cardinales se caracterizan por esta operación.

1.2.5. Conceptos Preliminares

Lo que se intenta encontrar es una manera óptima de representar la estructura de árbol y la estructura de símbolos que está asociada a los árboles cardinales.

Dado que los árboles cardinales tienen además de la representación de la estructura del árbol en sí, una estructura de símbolos, se busca poder representarlas mediante una estructura que soporta operaciones rank y select.

Más tarde, en la sección de implementación se podrá ver que la resolución de las operaciones objetivos de esta memoria, se basan en combinaciones de estas operaciones. A continuación se describirán algunas de éstas que se considera son las más relevantes.

Diccionarios Indexables - *Indexable Dictionaries*

Sea $S \subset U = \{1, \dots, u\}$ un subconjunto ordenado de tamaño n de un universo de tamaño u . Un **diccionario indexable (ID)** es una estructura de datos que almacena un conjunto S y soporta las siguientes operaciones:

- **Rank**(S, x): Para un $x \in U$ retorna -1 si $x \notin S$; Por otra parte, retorna $|\{s_i \in S \mid s_i < x\}|$.
- **Select**(S, i): Para un $i \in \{1, \dots, n\}$, retorna el i -ésimo elemento más pequeño en S .

Pero la representación de interés es la de **Diccionarios Indexables Compactos** (*Succinct Indexable Dictionaries*). Dado que hay $\binom{u}{n}$ conjuntos de n elementos de un universo de tamaño u , una representación compacta

de un diccionario indexable requiere **al menos** $\mathcal{B}(n, u) = \lceil \lg \binom{u}{n} \rceil$ bits de espacio [4]. Lo cual corresponde a:

$$\begin{aligned}\mathcal{B}(n, u) &= n \lg \frac{u}{n} + n \lg e - O(\lg n) - \Theta\left(\frac{n^2}{u}\right) \\ &= n \lg \frac{u}{n} + 1,44n - O(\lg n) - \Theta\left(\frac{n^2}{u}\right)\end{aligned}\tag{1.1}$$

Por otra parte, un **Diccionario Indexable Completo** (*Fully-Indexable Dictionary (FID)*), es una estructura de datos que soporta las operaciones **Rank**(S, x), **Select**(S, i), **Rank**($\bar{S}, x$) y **Select**($\bar{S}, i$), donde $\bar{S}$ es el complemento del conjunto S .

Los *fully-indexable dictionary* relacionan a S con una representación de bits:

- Sea $B_S[0 \dots u-1]$ un vector de bits tal que $B[i] = 1 \Leftrightarrow i \in S$, de otra manera $B[i] = 0$. En consecuencia, **Rank**(S, x) es la cantidad de 1s en $B_S[0 \dots i-1]$, **Select**(S, i) es la posición del i -ésimo 1 en B_S , **Rank**($\bar{S}, x$) es la cantidad de 0s en $B_S[0 \dots i-1]$, y por último **Select**($\bar{S}, i$) es la posición del i -ésimo 0 en B_S .

Secuencias Sucintas con Operaciones *rank* y *select*

Dada una secuencia $S[1 \dots n]$ de símbolos sobre un alfabeto $\Sigma = \{0, \dots, \sigma-1\}$ (es decir, un *string*) y dado cualquier $c \in \Sigma$, se definen las siguientes operaciones:

- **rank_c**(S, i): Obtiene la cantidad de ocurrencias de c en $S[1 \dots i]$.
- **select_c**(S, j): Obtiene la posición del j -ésimo c en S .
- **access**(S, i): Obtiene el i -ésimo elemento de la secuencia, es decir, $S[i]$.

Para $\sigma = 2$ (secuencias binarias), estas operaciones están soportadas en tiempo constante y usando $n + o(n)$ bits [5], o incluso $nH_0(B) + o(n)$ bits [6], donde $H_0 \leq 1$ corresponde a la entropía de orden 0 de B .

Si $\sigma = O(\text{polylog}(n))$, la solución de Ferragina et al. [7] soporta las operaciones descritas en tiempo constante y requiriendo $nH_0(S) + o(n)$ bits de espacio, donde $H_0(S) \leq \lg \sigma$, es la entropía empírica de orden cero de S [8].

Además, en general el tiempo es del orden $O\left(\frac{\lg \sigma}{\lg \lg n}\right)$ y el espacio utilizado del orden de $nH_0(S) + o(n \lg \sigma)$ bits.

Por último, la representación de Golynski et al. [9] requiere $n(\lg \sigma + o(\lg \sigma))$ bits de espacio [10], soportando las operaciones **select_c**(S, j) en tiempo de orden $O(1)$, y las operaciones **rank_c**(S, i) y **access**(S, i) en tiempo $O(\lg \lg \sigma)$.

Paréntesis Balanceados

El problema de representar una secuencia balanceada de paréntesis está relacionado a la representación sucinta de árboles.

Dada una secuencia P de $2n$ paréntesis balanceados, las operaciones que se busca soportar son las siguientes:

- **findclose**(P, i): Para un $P[i] = '('$, obtiene la posición del cierra-paréntesis correspondiente.
- **findopen**(P, j): Para un $P[j] = ')'$, obtiene la posición del abre-paréntesis correspondiente.
- **excess**(P, i): Obtiene la diferencia entre la cantidad de abre-paréntesis y cierra-paréntesis hasta la posición i en P .
- **enclose**(P, i): Dado un par de paréntesis tal que su abre-paréntesis se encuentra en la posición i , **enclose**, obtiene la posición del abre-paréntesis más cercano que contiene a i .

Hay $\frac{1}{n+1} \binom{2n}{n}$ secuencias distintas con n pares de paréntesis balanceados [11], por lo tanto, el mínimo de bits para representar esta secuencia es $\left\lceil \lg \frac{1}{n+1} \binom{2n}{n} \right\rceil = 2n - O(\lg n)$ bits.

La estructura de paréntesis balanceados de interés para este trabajo es la presentada por Navarro y Sadakane [12]. Esta estructura utiliza un índice de $o(n)$ bits por sobre la secuencia P , además de los $2n$ bits requeridos por la secuencia P . Con esta estructura las operaciones descritas, son soportadas en tiempo constante.

1.2.6. Árboles Cardinales Compactos

El número de árboles cardinales k -arios diferentes con n nodos es $\frac{1}{kn+1} \binom{kn+1}{n}$ [11], en base a esto la cota mínima de teoría de la información para la cantidad de bits necesarios para representar un árbol cardinal es:

$$C(n, k) = \left\lceil \lg \frac{1}{kn+1} \binom{kn+1}{n} \right\rceil. \quad (1.2)$$

Asumiendo que k es una función de n , se tiene que $C(n, k) \approx n(\lg k + \lg e) - o(n \lg k) = 1,44n + n \lg k - o(n \lg k)$ bits.

Trabajos Previos

Con respecto a los desarrollos prácticos, cabe mencionar que no hay muchas implementaciones propuestas para árboles cardinales compactos. Las investigaciones no han salido aún de los márgenes teóricos. Por otra parte, las implementaciones de árboles compactos sí tienen resultados teóricos y prácticos en la literatura. En el trabajo de Arroyuelo et al. [13] se implementan y realizan comparaciones para varias técnicas de representar árboles generales de forma sucinta. Sin embargo, en dicho trabajo se dejan de lado los árboles cardinales.

1.2.7. Árboles Generales

Formalmente, un árbol general se define de la siguiente manera:

- Un solo nodo, es un árbol por sí mismo. Este nodo es también la raíz del árbol.
- Suponer que n es un nodo y $T_1, T_2, \dots, T_k$ son árboles con raíces $n_1, n_2, \dots, n_k$ respectivamente. Se puede construir un nuevo árbol haciendo que n sea el padre de los nodos $n_1, n_2, \dots, n_k$. En este árbol n es la raíz y $T_1, T_2, \dots, T_k$ son subárboles de la raíz. Los nodos $n_1, n_2, \dots, n_k$ son llamados los *hijos* del nodo n .

Para denotar un árbol nulo, es decir, un “árbol” sin nodos, se utiliza el símbolo Λ para representarlo.

En la Figura 1.1 se muestra un ejemplo de un árbol general.

Dentro de las operaciones más importantes soportadas por un árbol general, se encuentran las siguientes:

1. *first-child*(x): Obtiene el primer hijo del nodo x .
2. *next-sibling*(x): Obtiene el siguiente hermano del nodo x .

1.2.8. Representación De Árboles Generales

Un punto importante a conocer es cómo representar un árbol general, sin recurrir a la manera clásica que usa punteros provocando un significativo uso de espacio, sólo para almacenar direcciones de memoria.

En estructuras de datos sucintas, lo que se hace por lo general, es representar los árboles como secuencias de paréntesis balanceados, o secuencias binarias balanceadas. Dentro de las más reconocidas se encuentran 3: **BP**, **DFUDS** y **LOUDS**.

LOUDS

Level Order Unary Degree Sequence fue propuesta por Jacobson [14]. Ésta representa a cada nodo escribiendo en unario el grado de éste, y realizando un recorrido por niveles sobre el árbol. **LOUDS** utiliza una representación de bits, por ejemplo un nodo de grado 3 es escrito '1110', mientras que una hoja se representa con un '0'.

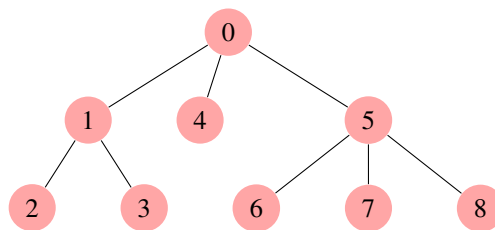


Figura 1.2: Árbol general de ejemplo.

Utilizando la Figura 1.2 como referencia, su representación **LOUDS** es como sigue:

$$\begin{array}{cccccccccccc} 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \underbrace{\hspace{1.5cm}}_0 & \underbrace{\hspace{1.5cm}}_1 & \underbrace{\hspace{1.5cm}}_4 & \underbrace{\hspace{1.5cm}}_5 & \underbrace{\hspace{1.5cm}}_2 & \underbrace{\hspace{1.5cm}}_3 & \underbrace{\hspace{1.5cm}}_6 & \underbrace{\hspace{1.5cm}}_7 & \underbrace{\hspace{1.5cm}}_8 \end{array}$$

Donde los números bajo la secuencia binaria (0, 1, 4,..., 8), indican qué nodo está siendo representado a lo largo de la secuencia.

Paréntesis Balanceados - BP

La representación de paréntesis balanceados (*balanced parentheses*) se crea realizando un recorrido *preorder* (primero en profundidad) del árbol, escribiendo un abre paréntesis cada vez que se llega a un nodo, y escribiendo un cierra paréntesis cuando se termina de recorrer todo el subárbol de un nodo.

Tomando como ejemplo el árbol de la Figura 1.2, la secuencia de paréntesis balanceados que lo representa es la siguiente:

$$\begin{array}{lcl} \text{secuencia:} & (& (& (&) & (&) &) & (&) & (& (&) & (&) & (&) &) &) \\ \text{nodo:} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \end{array}$$

DFUDS

Depth-First Unary Degree Sequence fue propuesta por Benoit et al. [15]. Esta representación de un árbol realiza un recorrido primero en profundidad, tal como **BP**, pero esta vez escribiendo en unario el grado del árbol, al igual que lo hacía **LOUDS**, pero esta vez utilizando paréntesis. Los nodos se identifican por el índice donde empiezan sus $d + 1$ paréntesis. Tomando como ejemplo el árbol de la Figura 1.2, la secuencia de paréntesis balanceados que lo representa es la que se muestra a continuación:

$$\begin{array}{cccccccccccc} (& (& (& (& (&) &) &) &) & (& (& (&) &) &) &) &) &) \\ \underbrace{\hspace{1.5cm}}_{\text{Dummy}} & \underbrace{\hspace{1.5cm}}_0 & \underbrace{\hspace{1.5cm}}_1 & \underbrace{\hspace{1.5cm}}_2 & \underbrace{\hspace{1.5cm}}_3 & \underbrace{\hspace{1.5cm}}_4 & \underbrace{\hspace{1.5cm}}_5 & \underbrace{\hspace{1.5cm}}_6 & \underbrace{\hspace{1.5cm}}_7 & \underbrace{\hspace{1.5cm}}_8 \end{array}$$

Se observa que al inicio de la secuencia de paréntesis, se agrega un abre paréntesis al que se denomina *dummy*. La adición de este paréntesis se explica debido a la necesidad de balancear la secuencia de paréntesis. Con esto es posible utilizar la representación DFUDS con estructuras de datos que soportan paréntesis balanceados.

1.2.9. Wavelet Trees

Los *wavelet trees* fueron introducidos por Grossi, Gupta y Vitter en su trabajo *High-order entropy-compressed text indexes* [16]. Los *wavelet trees* son una estructura de datos sucinta, que organizan las secuencias de símbolos en una jerarquía de *bit vectors*, para responder consultas sobre grandes alfabetos.

A continuación se detallará cómo se construyen y cómo se realizan consultas sobre esta estructura de datos.

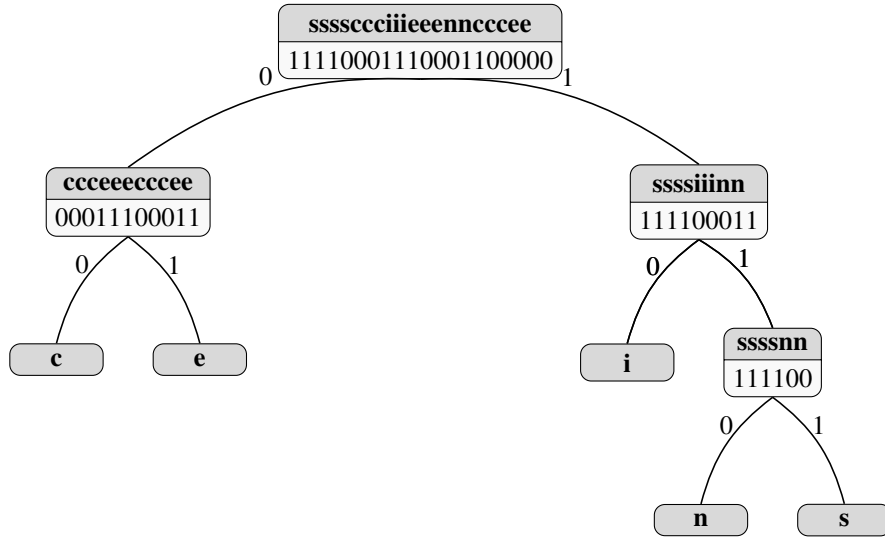


Figura 1.3: Ejemplo de Wavelet Tree

Construcción de Wavelet Trees

Para ilustrar cómo se construyen los *wavelet trees*, se utilizará como ejemplo la cadena "ssssccciieeennccee"

En primer lugar, se define el alfabeto utilizado para describir la cadena, en este caso se tiene que $\Sigma = \{c, e, i, n, s\}$. Con esto se procede a mapear estos valores con 0 y 1's con el fin de poder utilizar las operaciones rank/select de los *diccionarios indexables* que fueron mencionados anteriormente. El alfabeto queda como sigue:

$$\Sigma = \{c, e, i, n, s\}$$

$$\mathcal{B} = \{0, 0, 1, 1, 1\}$$

En la Figura 1.3, el nodo raíz corresponde a la representación hecha hasta el momento.

Luego el siguiente nivel del árbol, corresponde a separar los símbolos a los que les fue asignado cero por un lado, y a los que les fue asignado uno por otro lado.

Nuevamente en estos nodos se realiza el reconocimiento del alfabeto y se vuelve a realizar su codificación como *bit-vector*.

Por ejemplo, el nodo que agrupa los símbolos asignados con cero, queda de la siguiente manera.

$$\Sigma = \{c, e\}$$

$$\mathcal{B} = \{0, 1\}$$

Por último, se vuelve a separar este nodo, lo que lo lleva a los nodos terminales dado que éstas contienen solo un símbolo.

Luego se realiza el mismo procedimiento por el otro lado del árbol, resultando el *wavelet tree* que se muestra en la figura 1.3. Cabe agregar también, que las secuencias de símbolos en el *wavelet tree* son de referencia y en la estructura resultante solo se almacenan los vectores de bits.

Resolviendo Consultas Sobre el Wavelet Tree

Resolviendo una consulta del tipo Rank

Para explicar el procedimiento se resolverá la operación $rank_c(S, 6)$ que se interpreta como la cantidad de símbolos 'c' hasta la posición 6 de la secuencia S .

En primera instancia se busca saber cómo se representa el símbolo 'c' en este nivel, que en este caso resulta ser un cero.

Con esto se realiza un $rank_0(NodoRoot, 6)$ hasta la posición en cuestión. En este caso da 2, el cual corresponde a la *nueva posición* buscada.

Una vez hecho esto se desciende en el árbol por la rama que representa a las 'c', es decir, la de etiqueta '0', estando situados sobre el 'NuevoNodo'.

Lo que sigue es repetir lo realizado hasta ahora, a lo largo del árbol, es decir, se vuelve a consultar cual es la representación del símbolo 'c' en este nivel. Nuevamente es '0'. Por lo que se realiza un $rank_0(NuevoNodo, x)$ hasta la nueva posición definida en el nivel superior, es decir, hasta la posición 2, $rank_0(NuevoNodo, 2)$. El resultado es 2, por lo que esta es la nueva posición.

Ahora al descender en el árbol, por la rama etiquetada con '0', se llega a la hoja 'c'. Esto indica que el valor de la última posición guardada es la cantidad de 'c's que existen hasta la posición 6, en nuestro ejemplo.

Por último se tiene: $rank_5(6, 'c') = 2$.

Resolviendo consulta del tipo Access

Análogo al caso de $Rank_c(S, 6)$ en este caso se resolverá $Access(S, 8)$, siendo S la secuencia de símbolos y 8 la posición que se desea consultar.

Para conocer el símbolo dada una posición el procedimiento es el siguiente:

En primer lugar se realiza un $Access(NodoRoot, 8)$ y se obtiene un '1'. Con esto se sabe que se desciende por la rama etiquetada con '1'. Además se actualiza la posición buscada realizando un $Rank_1$ dado que se obtuvo un '1'. La nueva posición es $pos = Rank_1(NodoRoot, '1')$, que en este caso resulta ser '5'. Por último se desciende por el árbol, por la rama con etiqueta '1' al que se llamará 'NuevoNodo'.

Lo que resta es realizar el procedimiento antes descrito para cada nodo hasta llegar a un nodo hoja.

A continuación, se realiza un $Access(NuevoNodo, 5)$ que retorna '0'. Nuevamente se actualiza la posición realizando un $Rank_0(NuevoNodo, 5)$ que resulta ser '2', y se desciende por la rama etiquetada de '0'.

Al descender se llega al nodo hoja 'i', lo que implica que la posición 8 de la secuencia contiene una 'i'.

En consecuencia, se tiene que: $Access(S, 8) = 'i'$

Resolviendo consulta del tipo Select

Otra de las operaciones de un wavelet tree es $select(S, i)$.

Para ejemplificar la explicación se tomará el caso $Select_n(S, 1)$, es decir, obtener la posición la primera aparición de 'n' en la secuencia.

Para resolver esta consulta, se comienza desde los nodos hojas del árbol. Dado que se busca el símbolo 'n', se accede a la hoja que contiene una 'n'. Se sube por esta rama reconociendo la etiqueta del enlace, en este caso corresponde a un **cero**.

Como lo buscado es la primera ocurrencia del símbolo 'n', se realiza un $Select_0(S, 1)$, con el fin de encontrar la posición del primer 0 dentro del nodo del árbol. Esto da como resultado la quinta posición del nodo.

Nuevamente se vuelve a subir por el árbol, ahora teniendo en el enlace un **uno**. En consecuencia, se busca la posición del quinto uno dentro del nodo. Es decir, un $Select_1(S, 5)$, que da como resultado la octava posición.

Por último, se sube al nodo raíz, por el enlace etiquetado con un **uno**, y se busca el octavo uno dentro del nodo, esto es, $Select_1(S, 8)$, que resulta en la **décimo cuarta posición**.

En consecuencia, la **décimo cuarta posición** es la respuesta a la consulta *select* realizada, y el procedimiento es el anteriormente descrito.

1.2.10. Otras Estructuras

A continuación se presenta la Tabla 1.2, en la que se describe un resumen con respecto al espacio utilizado y el tiempo de respuesta de las estructuras de datos que se utilizarán para realizar los experimentos. Se incluyen también la búsqueda lineal y búsqueda binaria, que también forma parte de los experimentos de interés para realizar las pruebas sobre los árboles cardinales. Aunque no son estructuras de datos, permiten establecer puntos de comparación ya que no utilizan espacio adicional, creando tablas de búsqueda como las estructuras **rank/select**

Todas las estructuras a continuación se basan en la idea de los *diccionarios indexables*, que utilizan las operaciones **Rank**(S, x) y **Rank**(S, i) para navegar la secuencia de símbolos.

Tabla 1.2: Resumen complejidades

Propuesta	Espacio en bits	Tiempo de acceso
Wavelet Trees [16]	$2n + n \lg \sigma + o(n \lg \sigma)$	$\lg \sigma$
Huffman Shaped Wavelet Trees [17]	$2n + nH_0 + o(nH_0); H_0 \leq \lg \sigma$	$\lg \sigma$
Golynski et al.[9]	$2n + n \lg \sigma + o(n \lg \sigma)$	$\lg \lg \sigma$
Alphabet Partitioning [18]	$2n + nH_0 + o(nH_0)$	$\lg \lg \sigma$
Búsqueda Lineal	$2n + n \lg \sigma + o(n)$	σ
Búsqueda Binaria	$2n + n \lg \sigma + o(n)$	$\lg \sigma$

Referente a la notación utilizada se tiene:

- H_0 , corresponde a la entropía de orden cero (Ver Sección 1.2.11).
- σ , es el tamaño del vocabulario.
- n , es la cantidad de nodos del árbol.

1.2.11. Entropía de Orden 0 - H_0

La entropía de orden cero es una medida relacionada a la compresión de texto y se utiliza para modelar la compresibilidad de un texto.

Definición 2.1.: Dado un texto $T[1..u]$ sobre un alfabeto σ , la entropía empírica de orden cero de T se define como:

$$H_0 = \sum_{c \in \Sigma} \frac{n_c}{u} \log_2 \frac{u}{n_c}, \quad (1.3)$$

donde n_c es el número de ocurrencias del símbolo c en T . La sumatoria incluye solo aquellos símbolos que sí ocurren en T , por lo que $n_c > 0$.

Propiedad 2.1.: Dado un texto T sobre un alfabeto de tamaño σ , este cumple que $0 \leq H_0(T) \leq \lg k$.

1.2.12. Tasa De Compresión

Un concepto que también se considera necesario, tiene que ver con la compresión de datos. Los conceptos asociados a esta materia, entregan herramientas útiles para comparar algoritmos de compresión. Los principales conceptos que se utilizarán posteriormente para comparar resultados, son: **tasa de compresión** y **ahorro de espacio**. A continuación, se muestran sus definiciones:

- **Tasa de compresión:** Se define como el cociente entre el tamaño de los datos sin comprimir y el

tamaño de los datos comprimidos.

$$\text{Tasa de Compresión} = \frac{\text{Tamaño sin comprimir}}{\text{Tamaño comprimido}}. \quad (1.4)$$

- **Ahorro de espacio:** En ocasiones, se utiliza este concepto, el cual se define como la reducción conseguida con respecto al espacio utilizado sin comprimir.

$$\text{Ahorro de espacio \%} = \left(1 - \frac{\text{Tamaño comprimido}}{\text{Tamaño sin comprimir}}\right) \times 100. \quad (1.5)$$

Capítulo 2

Propuesta

Esta memoria propone descomponer un árbol cardinal en una estructura que represente su topología de árbol, por una parte, y por otra parte una componente que represente los símbolos de los arcos del árbol. Con estas dos estructuras se soportan todas las operaciones de interés para un árbol cardinal (ver Tabla 1.1).

Para la estructura de árbol, se abordaron varias representaciones posibles como DFUDS, LOUDS y BP. En base a esto es que para la topología del árbol se propone utilizar la representación DFUDS, sobre la estructura de paréntesis balanceados propuesta por Navarro y Sadakane [12].

Para los símbolos del árbol, se propone utilizar alguna estructura para secuencias generales que soporte las operaciones *rank* y *select*. Dentro de éstas hay varias propuestas, cada una con sus ventajas y desventajas en cuanto a tiempo y espacio. Las que se utilizarán en este trabajo, corresponden a:

- *Wavelet Trees* [16].
- *Huffman Shaped Wavelet Trees* [17].
- *Golynski et al.* [9].
- *Alphabet Partitioning* [18].

Además de utilizar estas estructuras, se utilizarán búsqueda lineal y búsqueda binaria sobre el conjunto de símbolos. Esto dará un punto de comparación, ya que estos algoritmos no utilizan espacio adicional por sobre la secuencia.

Cabe mencionar que la representación utilizada para la topología del árbol no centra la atención de este trabajo, ya que principalmente se busca observar el comportamiento de las estructuras para *rank/select* en el manejo de los símbolos del árbol, propiedad que es característica de los árboles cardinales.

En el diagrama de la Figura 2.1 se unen conceptos que están relacionados, con el fin de dar una relación visual a todos los conceptos introducidos en el estado del arte y la propuesta que se llevará a cabo. Cabe

mencionar que la representación de símbolos usando secuencias binarias (IDs y FIDs) no será abordada en esta memoria.

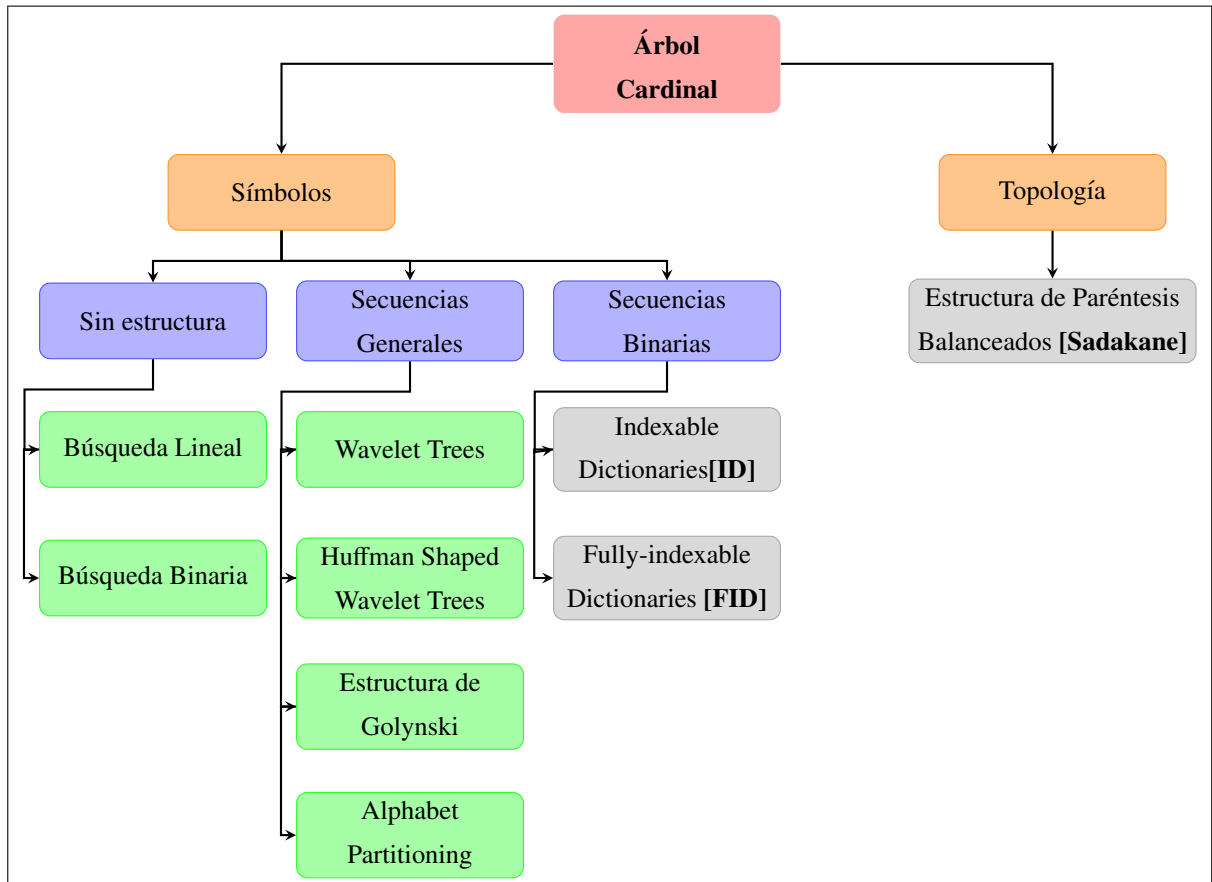


Figura 2.1: Diagrama Árbol Cardinal.

2.1. Implementación

2.1.1. Consideraciones Previas

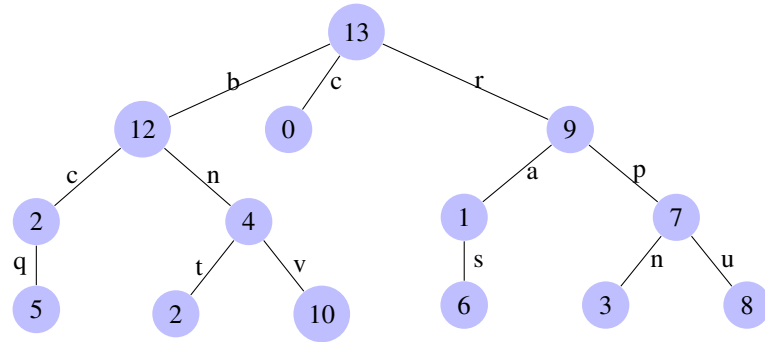


Figura 2.2: Árbol cardinal de ejemplo.

Arreglo de Símbolos

Un árbol cardinal contiene símbolos en sus arcos. Para almacenarlos, se define el arreglo *letts*, donde los símbolos son almacenados según aparecen al realizar un recorrido en preorden. Por cada nodo alcanzado durante este recorrido, se almacenan los símbolos de sus hijos de forma consecutiva en *letts*.

Considerando el árbol de la Figura 2.2, al realizar el recorrido en preorden e ir agregando las etiquetas de los nodos en el orden en que son visitados, el arreglo *letts* final es el siguiente:

letts:	b	c	r	c	n	q	t	v	a	p	s	n	u
índice:	0	1	2	3	4	5	6	7	8	9	10	11	12

Arreglo de Datos

Para almacenar los datos de los nodos del árbol, se define el arreglo *data*. Los datos de cada nodo se encuentran en la posición correspondiente al número preorden del nodo. Esto quiere decir, que el nodo con preorden 5 tendrá sus datos correspondientes almacenados en la posición 5 del arreglo de datos.

El arreglo *data* del ejemplo de la Figura 2.2 es el siguiente:

data:	13	12	2	5	4	2	10	0	9	1	6	7	3	8
índice:	0	1	2	3	4	5	6	7	8	9	10	11	12	13

2.1.2. Implementación de Árboles Cardinales

Nuestra implementación de árboles cardinales, se compone de tres elementos principales que son: una secuencia de símbolos, una secuencia de paréntesis, y un arreglo de datos.

Dado que se utilizarán varias estructuras para la secuencia de símbolos, se utiliza una clase parametrizada en C++ para implementarla, que permita elegir de forma simple la estructura de símbolos a usar. La estructura para representar la topología del árbol no varía, por lo que se define directamente. Por último, se utiliza un vector de enteros para almacenar los datos de cada nodo. En el Código 2.1 se observa la clase de árbol cardinal implementada.

```
1 // size_type: tamaño de los símbolos. (8 bits, 32 bits, 64 bits).
2 // seq_type: estructura de símbolos a utilizar (golynski, alphabet partitioning,
3 //           wavelet trees, etc).
4 template <class seq_type, typename size_type>
5 class cardinal_tree
6 {
7     private:
8         // Secuencia de símbolos.
9         seq_type *letts;
10
11         // Topología del árbol.
12         bp_support_sada<256, 32, rank_support_v5<1>, bit_vector::
13             select_0_type> *tree;
14
15         // Datos del árbol.
16         vector<int> * data;
17
18     public:
19         cardinal_tree(int_vector<> seq_, bit_vector * bp, vector<int> *
20             dat) {
21             // Inicializar secuencia de símbolos.
22             letts = new seq_type();
23             construct_im(*letts, seq_, 0);
24
25             // Inicializar topología del árbol.
26             tree = new bp_support_sada<256, 32, rank_support_v5<1>,
27                 bit_vector::select_0_type>(bp);
28
29             // Inicializar vector de datos.
30             data = dat;
31         }
32 }
```

```

28
29         // ... Operaciones árbol ...
30     };

```

Código 2.1: Implementación de la clase árbol cardinal.

2.1.3. Implementación de Operaciones

El trabajo de esta memoria está basado en la librería *Succinct Data Structure Library* (sdsl) [19], la cual cuenta con la implementación de diversas estructuras sucintas, basados en el trabajo de aproximadamente 40 artículos de investigación relacionadas con las estructuras de datos sucintas.

2.1.4. Operaciones Básicas Sobre Estructura de Paréntesis Balanceados

Las primeras implementaciones que se describen a continuación, corresponden a las operaciones elementales a partir de las cuales se construyen el resto de las operaciones. Éstas son las operaciones *rank* y *select* que actúan sobre la secuencia de paréntesis, y también sobre la secuencia de símbolos.

Operaciones Rank

La definición formal se ha abordado previamente en la Sección 1.2.5 pero, en breves palabras, la operación $rank_i(S, x)$ entrega la cantidad de ocurrencias de ceros (cierra-paréntesis) hasta la posición x . Es decir, actúa sobre el intervalo $[0, x]$ con 0 y x inclusivos.

A continuación (ver Código 2.2), se muestra la implementación del método *tree_rank0*.

```

1  size_t tree_rank0(size_t x) {
2      return x - tree->rank(x) + 1;
3  }

```

Código 2.2: Implementación del método *tree_rank0*.

La definición de la operación *tree_rank1* (ver Código 2.3) realiza la operación $rank_1(S, x)$. Dado que es conocida la cantidad de 0's hasta la posición x , la cantidad de 1's será la posición actual menos la cantidad de 0's hasta x . Dado que las posiciones inician en 0, se suma 1.

```

1  size_t tree_rank1(size_t x) {
2      return tree->rank(x);
3  }

```

Código 2.3: Implementación del método *tree_rank1*.

Operaciones Select

El método *tree_select0* (ver Código 2.4, es análoga a la definición de las operaciones *tree_rank*. La operación retorna la posición del i -ésimo 0 dentro de la secuencia de paréntesis balanceados.

```

1 size_t tree_select0(size_t x) {
2     return tree->select(x);
3 }
```

Código 2.4: Implementación del método *tree_select0*.

Como se mencionó en secciones anteriores, la operación $Select_1(S, x)$ no es necesaria para realizar las operaciones de navegación dentro del árbol, lo que permite ahorrar espacio.

2.1.5. Operaciones Sobre Estructura De Paréntesis Balanceados

En esta sección se describirán las definiciones formales e implementaciones realizadas para las operaciones objetivos.

Operación Degree

La operación degree entrega la cantidad de hijos de un nodo x . Se define formalmente de la siguiente manera:

$$degree(x) \equiv select_y(T, rank_y(T, x - 1) + 1) - x. \quad (2.1)$$

En general, lo que se necesita es contabilizar la cantidad de abre-paréntesis desde la posición x , hasta la posición y (ver Cuadro 2.1).

...	)	(	(	)	...
		x		y	

Tabla 2.1: Secuencia degree de ejemplo.

Al realizar *tree_rank0(x - 1)* se consigue la cantidad de cierra paréntesis, que preceden al nodo en cuestión. Al realizar *tree_select0* de esta cantidad más 1, se obtiene la posición de y . Teniendo esto, solo basta restar y

con x , para obtener la cantidad de abre-paréntesis.

```
1 size_t degree(size_t x) {  
2     return tree_select0(tree_rank0(x - 1) + 1) - x;  
3 }
```

Código 2.5: Implementación de la operación *degree*.

Operación Parent

La operación *parent* retorna la posición del padre de un nodo x . Los abre-paréntesis que indican el grado de un nodo, están conectados a sus hijos de cierta forma. Dado un nodo x , $findopen(T, x-1)$ devuelve el abre paréntesis correspondiente al nodo x dentro de la representación del nodo padre de x . A partir de allí, el primer abre paréntesis del padre de x se obtiene utilizando $rank_i$ y $select_i$ como se muestra a continuación:

$$parent(x) \equiv select_i(rank_i(T, findopen(T, x - 1))) + 1. \quad (2.2)$$

Como se observa en Código 2.6, si la cantidad predecesora de cierra-paréntesis es 0, entonces se retorna 1, que corresponde al nodo raíz.

```
1 size_t parent(size_t x) {  
2     size_t aux = tree_rank0(tree->find_open(x-1));  
3     if (aux == 0) return 1;  
4     return tree_select0(aux) + 1;  
5 }
```

Código 2.6: Implementación de la operación *parent*.

Operación Child-Rank

Esta operación retorna la posición de un nodo con respecto a sus hermanos. El valor retornado va de 1 hasta σ . Se calcula de la siguiente manera:

$$child_rank(x) \equiv select_i(rank_i(findopen(x - 1)) + 1) - findopen(x - 1). \quad (2.3)$$

Al realizar $findopen(x - 1)$ lo que se logra es obtener la posición de uno de los abre-paréntesis que describen el grado del nodo padre.

Lo que se necesita es encontrar en qué posición se encuentra el abre-paréntesis del nodo en la secuencia que indica el grado del nodo padre. Para ello se realiza un $select_i(rank_i(findopen(x - 1)) + 1)$ con el cual

se obtiene la posición x del paréntesis que cierra al nodo padre. Al restar ambos valores, se obtiene en qué posición está el paréntesis con respecto al cierra paréntesis.

Cabe destacar que los abre-paréntesis del nodo padre y los hijos de éste se relacionan de la siguiente manera: El primer paréntesis del nodo padre se relaciona con el último hijo de éste. El segundo paréntesis se relaciona con el penúltimo hijo del nodo, y así sucesivamente. Es decir, existe una relación inversa entre la posición del paréntesis y su ranking con respecto a sus hermanos. La implementación se puede ver en el Código 2.7.

```

1  size_t child_rank(size_t x) {
2      return tree_select0(tree_rank0(tree->find_open(x - 1)) + 1) -
          tree->find_open(x - 1);
3  }

```

Código 2.7: Implementación de la operación *child_rank*.

Operación Child

La operación *child*, entrega la posición del i -ésimo hijo de un nodo x . En la Sección 2.1.5, se discutió cómo se comportan las posiciones de los hijos de un nodo con respecto a las posiciones de los abre-paréntesis que definen el grado del nodo padre. Se concluyó que la relación es inversa. En base a esto, la operación *child*(x, i) se define formalmente como sigue:

$$child(x, i) \equiv findclose(T, select_i(T, rank_j(x) + 1) - i) + 1. \quad (2.4)$$

Para lograr obtener el i -ésimo hijo de un nodo, se ha de encontrar en primer lugar, el i -ésimo paréntesis con respecto al cierra-paréntesis que delimita al nodo x . Para encontrarlo, se contabiliza la cantidad de cierra-paréntesis hasta la posición x . Con esto se tiene el cierra paréntesis del nodo anterior en la secuencia de paréntesis. Luego se selecciona la cantidad obtenida más 1, con esto se obtiene el cierra-paréntesis respectivo al nodo actual x . Por último, queda restar i a la posición del cierra-paréntesis y realizar un *findclose* a esta posición y sumar 1 para llegar al nodo correspondiente. A continuación en el Código 2.8, se encuentra la implementación de esta operación:

```

1  size_t child(size_t x, size_t i) {
2      return tree->find_close(tree_select0(tree_rank0(x) + 1) - i) +
          1;
3  }

```

Código 2.8: Implementación de la operación *child*.

Operación Preorder

La operación preorder recibe la posición de un nodo x , y retorna la posición del nodo en un recorrido en preorden del árbol. La definición formal de la operación es la siguiente:

$$preorder(x) \equiv rank(T, x - 1). \quad (2.5)$$

En la secuencia de paréntesis, un cierra-paréntesis indica el fin de un nodo. Por lo tanto, hay que contar el total de cierra-paréntesis predecesores. Dado que en *DFUDS* los nodos se almacenan en preorden, ese número es el *preorder* del nodo. Ver Código 2.9.

```
1 size_t preorder(size_t x) {  
2     return tree_rank0(x - 1);  
3 }
```

Código 2.9: Implementación de la operación *preorder*.

Operación Select-Node

Esta operación es inversa a la operación preorder, descrita anteriormente. Select-Node recibe un valor j , y retorna la posición x del nodo con preorden j . Se logra de la siguiente manera:

$$select_node \equiv select(T, j) + 1. \quad (2.6)$$

Seleccionando la posición del j -ésimo cierra paréntesis y luego se suma 1 para obtener el nodo buscado. La implementación se muestra en Código 2.10:

```
1 size_t select_node(size_t j) {  
2     if (j == 0) return 1;  
3     return tree_select0(j) + 1;  
4 }
```

Código 2.10: Implementación de la operación *select_node*.

Operación Subtree-Size

La operación *subtree_size*(x) retorna la cantidad de nodos que contiene el sub-árbol con raíz en el nodo x . Se define de la siguiente manera:

$$subtree(x) \equiv findclose(T, enclose(T, x) - x)/2 + 1. \quad (2.7)$$

Para saber cuál es el tamaño del subárbol correspondiente, es necesario saber dónde inicia y dónde termina. Dónde inicia es un valor conocido, ya que es un parámetro de la operación. Para saber dónde termina, se realiza la búsqueda de aquellos paréntesis que contienen al nodo x . Con la operación $enclose(x)$ se obtiene el abre-paréntesis que contiene al nodo x . Realizando un $findclose$ de esta posición, es posible determinar la posición donde termina el sub-árbol, que tiene como raíz al nodo x . Una de las opciones es contar cuantos cierra-paréntesis hay dentro de la sub-secuencia. Sin embargo, dado que corresponde a un sub-árbol, la sub-secuencia se encontrará casi balanceada sólo le faltará un abre-paréntesis para estar balanceada. En consecuencia, se restan las posiciones final e inicial, se dividen por 2 y se suma 1 para obtener la cantidad de nodos totales a los que representa la subsecuencia. De esta manera, no se realizan más operaciones rank/select y se obtiene el resultado de forma directa. Ver Código 2.11

```

1 size_t subtree_size(size_t x) {
2     return (tree->find_close(tree->enclose(x)) - x)/2 + 1;
3 }
```

Código 2.11: Implementación de la operación *subtree_size*.

Operación Ancestor

La operación $ancestor(x, y)$ recibe dos nodos x e y , y retorna verdadero si x es un ancestro de y en el árbol. Se define formalmente como sigue:

$$ancestor(x, y) \equiv findclose(T, enclose(T, x)) \geq y. \quad (2.8)$$

Para saber si el nodo x es un ancestro del nodo y , lo que se chequea es que la posición de y sea menor a la posición donde termina el sub-árbol de x , cómo se hizo previamente en la sección 2.1.5. En el Código 2.12 se muestra la implementación de la operación *ancestor*.

```

1 bool ancestor(size_t x, size_t y) {
2     if (x > y) return false;
3     return tree->find_close(tree->enclose(x)) >= y;
4 }
```

Código 2.12: Implementación de la operación *ancestor*.

Operación Access-Data

La operación $access_data(x)$ entrega la información almacenada en el nodo x del árbol. Dado que los datos son almacenados en preorden dentro del arreglo *data*, la posición del arreglo que contiene los datos del nodo

x , están en la posición que corresponde al valor en preorden del nodo x . Por lo que la operación *access_data* se define formalmente como sigue:

$$\text{access_data}(x) \equiv \text{data}[\text{rank}_\text{T}(T, x - 1)]. \quad (2.9)$$

La implementación de la operación se muestra en el Código 2.13.

```

1 int access_data(size_t x) {
2     return (*info)[tree_rank0(x-1)];
3 }

```

Código 2.13: Implementación de la operación *access_data*.

2.1.6. Operaciones Básicas Sobre Estructura de Símbolos (Rank/Select)

Como ya se mencionó, se utilizan estructuras adicionales que soportan operaciones rank/select, para almacenar el conjunto de símbolos y realizar consultas de manera eficiente sobre ésta.

Operación Label-Rank

El método *label_rank* (ver Código 2.14) implementa la operación $\text{rank}_c(S, x)$, que retorna la cantidad de ocurrencias del símbolo c , hasta la posición x .

```

1 size_t label_rank(size_t x, uint8_t s) {
2     return letts->rank(x, s);
3 }

```

Código 2.14: Implementación de la operación *label_rank*.

Operación Label-Select

El método *label_select* implementa $\text{select}_c(S, i)$ (Ver Código 2.15, operación que retorna la posición de la i -ésima ocurrencia del símbolo c).

```

1 size_t label_select(size_t x, uint8_t s) {
2     return letts->select(x, s);
3 }

```

Código 2.15: Implementación de la operación *label_select*.

2.1.7. Operaciones Sobre Estructura de Símbolos (Rank/Select)

Operación Label

La operación $label(x, i)$ entrega el valor de la etiqueta del i -ésimo hijo del nodo x . Como ya se mencionó en la Sección 2.1.1, los símbolos son almacenados en un arreglo, siguiendo un recorrido en preorden.

La definición formal de esta operación se muestra a continuación:

$$label \equiv letts[rank_1(S, x - 1) + i - 2]. \quad (2.10)$$

Para obtener la etiqueta del i -ésimo hijo del nodo x , lo primero es realizar un conteo de los símbolos predecesores. Esto se consigue realizando $rank_1(S, x)$, esto ya que cada paréntesis se corresponde con la ocurrencia de un símbolo en el árbol. Con esto se tiene la posición donde inician los símbolos de los hijos del nodo x dentro del arreglo de símbolos. Luego se suma i , para obtener la posición donde se almacena el símbolo (hay que restar 1 ya que las posiciones en los arreglos comienzan desde cero). Por último, hay que restar nuevamente 1, debido al paréntesis ficticio que se agrega en la construcción de la secuencia de paréntesis balanceado, el cual no tiene un símbolo asociado. El Código 2.16 muestra la implementación realizada de esta operación.

```
1 size_type label(size_t x, size_t i) {  
2     return (*letts)[tree_rank1(x - 1) + i - 2] ;  
3 }
```

Código 2.16: Implementación de la operación $label_select$.

Operación Label-Child

La operación $label_child(x, \alpha)$ entrega la posición del hijo del nodo x etiquetado con el símbolo α . La definición formal se muestra a continuación.

En primer lugar se definen ciertos valores, que se utilizarán para facilitar el cálculo de la operación:

$$\begin{aligned} position_symbols_begin &= rank_1(x - 1) - 1 \\ alpha_previous_count &= rank_\alpha(position_symbols_begin - 1, \alpha) \\ position_next_alpha &= select_\alpha(alpha_previous_count + 1, \alpha) \\ i &= position_next_alpha - position_symbols_begin + 1 \end{aligned} \quad (2.11)$$

Una vez realizadas estas definiciones, el cálculo de la operación $label_child(x, i)$ se resume a lo siguiente:

$$label_child \equiv child(x, i). \quad (2.12)$$

El objetivo es determinar cuál es la posición i del hijo con etiqueta α , ya que con esto, la operación se resuelve realizando $child(x, i)$. Para lograr esto, se obtiene en que posición del arreglo de símbolos comienzan los símbolos del nodo x . Se realiza para esto $rank_l(x - 1) - 1$. Se contabiliza la cantidad de ocurrencias del símbolo α hasta la posición anterior a los símbolos de los hijos del nodo x , $rank_\alpha(S, rank_l(x - 1) - 1)$. De esta manera, al realizar $select_\alpha(S, rank_\alpha(S, rank_l(x - 1) - 1) + 1)$, se tendrá la posición del símbolo α buscado. Por último, queda restar la posición del símbolo buscado, con la posición donde los símbolos comienzan, para obtener la posición del símbolo dentro de sus símbolos hermanos y realizar la operación $child(x, i)$.

A continuación, se muestra la implementación realizada (Código 2.17):

```

1  size_t label_child(size_t x, size_type alpha) {
2      size_t position_symbols_begin;
3      if (x == 1) {
4          position_symbols_begin = 0;
5      }
6      else position_symbols_begin = tree_rank1(x - 1) - 1;
7      size_t alpha_previous_count;
8      if (position_symbols_begin == 0) alpha_previous_count = 0;
9      else alpha_previous_count = label_rank(position_symbols_begin,
10         alpha);
11     size_t position_alpha = label_select(alpha_previous_count + 1,
12         alpha);
13     int position_symbols_end = position_symbols_begin + degree(x) -
14         1;
15     if (!(position_alpha >= position_symbols_begin && position_alpha
16         <= position_symbols_end)) return 0;
17     size_t i = position_alpha - position_symbols_begin + 1;
18     return child(x, i);
19 }
```

Código 2.17: Implementación de la operación *label.child*.

2.1.8. Operaciones Sobre Secuencia de Símbolos sin Estructuras Adicionales

En esta sección se describen los símbolos almacenados sin una estructura de datos optimizada para resolver consultas rápidamente, a diferencia de las utilizadas para las secciones anteriores que soportan operaciones *rank/select*. Lo que se utiliza aquí es un arreglo que contiene todos los símbolos del árbol y se realizan búsquedas sobre éste. Para realizar las búsquedas sobre el arreglo de símbolos se realizará búsqueda lineal y búsqueda binaria.

Operación Label-Child con Búsqueda Lineal

Dado que *letts* es un arreglo sin estructura, se busca el símbolo α entre los símbolos de los hijos del nodo x , usando búsqueda lineal. En el Código 2.18 se ve la implementación utilizada.

```
1 int linear_search(int_vector<> *seq, int left, int right, uint8_t alpha)
    {
2     for (size_t i=left; i<right+1; i++) {
3         if ((*seq)[i] == alpha) return i;
4     }
5     return -1;
6 }
```

Código 2.18: Implementación de la operación *linear_search*.

La implementación de *label_child* es similar a la descrita previamente en la Subsección 2.1.7. Se obtienen los índices en los que están contenidos los símbolos del nodo x , y luego se realiza búsqueda lineal sobre el tramo definido. La nueva implementación de *label_child*(x, α) se muestra a continuación (Código 2.19):

```
1 size_t label_child(size_t x, size_type alpha) {
2     size_t position_symbols_begin;
3     if (x == 1) {
4         position_symbols_begin = 0;
5     }
6     else position_symbols_begin = tree_rank1(x - 1) - 1;
7     size_t position_symbols_end;
8     position_symbols_end = position_symbols_begin + degree(x) - 1;
9     int i = linear_search(letts, position_symbols_begin,
10        position_symbols_end, alpha);
11     if (i == -1) return 0;
12     i = i-position_symbols_begin+1;
13     return child(x, i);
14 }
```

Código 2.19: Implementación de la operación *label_child* usando búsqueda lineal.

Operación Label-Child usando Búsqueda Binaria

Similar a la búsqueda lineal, lo que se busca soportar es la operación *label_child*(x, i), pero esta vez usando búsqueda binaria. El primer paso, como ya se explicó anteriormente en la sección 2.1.7, es encontrar el intervalo donde se encuentran los símbolos del nodo x .

Una vez realizado esto, solo basta invocar la función descrita en Código 2.20, donde se implementa la búsqueda binaria utilizada.

```
1 int binary_search(int_vector<> *seq, size_t left, size_t right,
   size_type alpha) {
2     size_t mid;
3     while (left <= right) {
4         mid = (left+right)/2;
5         if ((*seq)[mid] < alpha) left = mid+1;
6         else if ((*seq)[mid] > alpha) right = mid-1;
7         else return mid;
8     }
9     return -1;
10 }
```

Código 2.20: Implementación de la operación *binary_search*.

A continuación, en Código 2.21 se muestra la implementación para soportar la operación *label_child(x,i)* utilizando búsqueda binaria.

```
1 size_t label_child(size_t x, uint8_t alpha) {
2     size_t position_symbols_begin;
3     if (x == 1) position_symbols_begin = 0;
4     else position_symbols_begin = tree_rank1(x - 1) - 1;
5     size_t position_symbols_end;
6     position_symbols_end = tree_rank1(tree_select0(tree_rank0(x -
       1) + 1)) - 2;
7     size_t i = binary_search(letts, position_symbols_begin,
       position_symbols_end, alpha);
8     if (i == -1) return 0;
9     i = i-position_symbols_begin+1;
10    return child(x, i);
11 }
```

Código 2.21: Implementación de la operación *label_child* usando búsqueda binaria.

Capítulo 3

Experimentación

En esta sección se describirán los experimentos realizados, con el fin de comparar las estructuras utilizadas para soportar los símbolos del árbol cardinal.

Como ya se mencionó anteriormente, las estructuras en estudio para almacenar los símbolos del árbol son las siguientes:

- | | |
|--|---|
| 1. Balanced Wavelet Trees | 4. Alphabet Partitioning |
| 2. Huffman Shaped Wavelet Trees | 5. Búsqueda Lineal (Sin estructura) |
| 3. Golynski et al. | 6. Búsqueda Binaria (Sin estructura) |

Con respecto a los árboles utilizados para realizar los experimentos, estos son 12, de los cuales y 9 de ellos corresponden a árboles cardinales generados por el algoritmo LZ78 sobre distintos datasets. De esos datasets 6 provienen del sitio **Pizza&Chili Corpus** [20], y 3 de los árboles fueron generados a partir de una versión de una Wikipedia. Por último, 3 árboles fueron generados aleatoriamente.

Una descripción de los árboles se muestra a continuación:

1. **Proteins:** Árbol LZ78 sobre archivo de proteínas de 477MB.
2. **English:** Árbol LZ78 sobre texto en inglés de 477MB.
3. **DNA:** Árbol LZ78 sobre texto de cadenas de ADN de 386MB. Mayoritariamente conformado por los símbolos A, C, G, T.
4. **Sources:** Árbol LZ78 sobre texto de códigos fuente de 202MB.
5. **XML:** Árbol LZ78 sobre archivos XML de 283MB.
6. **MIDI:** Árbol LZ78 sobre archivos del tipo MIDI de 54MB.

7. **Wiki50MB**: Árbol LZ78 generado a partir de prefijo de la Wikipedia de 50MB.
8. **Wiki500MB**: Árbol LZ78 generado a partir de prefijo de la Wikipedia de 500MB.
9. **Wiki1GB**: Árbol LZ78 generado a partir de prefijo de la Wikipedia de 1GB.

Al generar los árboles aleatorios, se centró la atención en el tamaño del alfabeto utilizado. Los nombres de los árboles, poseen información respecto a su forma, utilizando la letra “v” como separador. El valor que precede a “v” corresponde al tamaño del alfabeto, y el valor que sucede a “v”, corresponde a la aridad promedio del árbol. Una descripción de los árboles aleatorios se muestra a continuación:

1. **Random5Kv1K**: Árbol aleatorio con $\sigma = 5,000$ y aridad promedio 1,000.
2. **Random50Kv1K**: Árbol aleatorio con $\sigma = 50,000$ y aridad promedio 1,000.
3. **Random1Mv200K**: Árbol aleatorio con $\sigma = 1,000,000$ y aridad promedio 200,000.

En la Tabla 3.1 se presentan además algunas características para describir los árboles utilizados, como lo son el tamaño del alfabeto y la cantidad de nodos presentes.

Tabla 3.1: Descripción de los Árboles

Árbol	Cantidad de Nodos	Tamaño Vocabulario
Proteins	64.591.184	27
English	47.946.167	227
DNA	30.136.373	17
Sources	20.693.981	231
XML	16.205.172	98
MIDI	8.735.103	133
Random5K1K	359.408.750	5.000
Random50Kv1K	331.412.799	50.000
Random1Mv200K	454.001.474	1.000.245
Wikipedia-50MB	4.855.656	663.091
Wikipedia-500MB	39.569.431	3.125.941
Wikipedia-1GB	80.864.306	5.305.569

También, además de la cantidad de nodos y símbolos para describir los árboles utilizados, se muestra en la Tabla 3.2 información sobre la forma de estos árboles. Se muestran características tales como: la cantidad promedio de hijos de cada nodo, la profundidad promedio de las hojas, entre otras.

Tabla 3.2: Características de los Árboles.

Árbol	Aridad Máxima	Aridad Promedio	Altura Máxima	Altura Promedio
Proteins	26	2,39	193	7,28
English	210	2,16	91	10,79
DNA	16	2,01	72	13,76
Sources	209	1,90	288	10,00
XML	97	1,72	165	16,67
MIDI	133	2,43	670	6,17
Random5Kv1K	999	948,98	3	3,00
Random50Kv1K	999	948,96	3	3,00
Random1Mv200K	200.049	200.000,65	2	2,00
Wikipedia-50MB	374.744	5,81	36	2,77
Wikipedia-500MB	1.596.882	5,44	93	3,32
Wikipedia-1GB	2.712.105	5,51	93	3,36

Dentro de los datos presentes en la Tabla 3.2, se muestra la aridez promedio que indica el grado promedio de los nodos de los árboles. Se observa que todos los árboles que no fueron generados aleatoriamente, poseen en promedio alrededor de 2 nodos hijos por nodo. Esto se explica ya que los árboles fueron generados por el algoritmo de compresión LZ78. En estos árboles es menos probable que nodos distantes de la raíz posean una gran cantidad de hijos.

Se observa también que existe disparidad entre la altura máxima de estos árboles (no aleatorios) y la altura promedio de los nodos hojas de los árboles. Esto indica que son pocos los nodos hojas situados en estas posiciones tan lejanas del nodo raíz. Quiere decir que los nodos por lo general están cercanos al nodo raíz, y solo algunos se alejan en mayor medida de éste.

Con respecto a los árboles generados aleatoriamente, son bastante más grande que los demás árboles. Estos se generaron con la intención de que sus alfabetos fueran más grandes, en consecuencia, son éstos los árboles con alfabetos de mayor tamaño. Son árboles más bajos, y casi todos sus nodos hojas alcanzan la altura máxima del árbol.

Por último, se observa en la Tabla 3.3 la entropía de orden cero de cada árbol. Además como punto de comparación se muestra $\lceil \log \sigma \rceil$, que corresponde a la cota superior de H_0 .

Tabla 3.3: Entropía H_0 de cada árbol.

Árbol	Entropía H_0	$\lceil \log \sigma \rceil$
Prots	4,27348	5
English	5,03284	8
DNA	1,99494	5
Sources	5,82663	8
XML	5,71527	7
MIDI	5,76032	8
Random5K1K	12,2868	13
Random50Kv1K	15,5991	16
Random1Mv200K	19,9317	20
Wikipedia-50MB	14,8275	20
Wikipedia-500MB	15,2522	22
Wikipedia-1GB	15,3399	23

3.1. Sistema y Método de Experimentación

Con respecto al sistema utilizado para realizar las pruebas, éstas se realizaron en una máquina con las especificaciones mostradas en la Tabla 3.4:

Tabla 3.4: Especificaciones del sistema utilizado.

Componente	Descripción
Nombre CPU	Intel(R) Xeon(R) CPU E5-2630 @2.30GHz
CPU(s)	12
Caché	15 MB
RAM	48 GB
S.O.	Ubuntu Server 16.04 (Linux versión 4.4.0-51)

Como se ha mencionado en este trabajo, la operación principal a estudiar es $label_child(x, \alpha)$: dado un nodo y un símbolo, retornar el nodo hijo de x identificado con el símbolo α .

Las pruebas se realizaron al ejecutar la operación $label_child$ 100.000 veces sobre nodos aleatorios dentro del árbol. El procedimiento es el siguiente.

1. Se escoge un símbolo al azar dentro de los hijos del nodo raíz.
2. Se desciende por el símbolo utilizando $label_child$ contabilizando el tiempo que toma encontrar el nodo.

3. Se vuelve a realizar el proceso, hasta llegar a un nodo hoja. En tal caso se vuelve al nodo raíz y se repite el proceso.

Cabe agregar que la función *random* fue utilizada con la misma semilla generadora de números aleatorios. Esto permite realizar el mismo recorrido en los árboles cada vez, para las distintas estructuras probadas. De esta manera no existen diferencias en los nodos recorridos del árbol y se minimizan los sesgos.

3.2. Resultados Experimentales

En esta sección se presentarán los resultados experimentales obtenidos a partir de los experimentos planteados en el capítulo anterior.

3.2.1. Árboles con Alfabetos Pequeños

Comparación Teórico / Práctica

En esta sección se analizarán los resultados obtenidos al realizar el contraste entre la cantidad de bits utilizados en los experimentos por los árboles, y lo que indican los márgenes teóricos.

En primera instancia, se utilizaron los árboles *Proteins*, *English*, *DNA*, *Sources*, *XML*, *MIDI*, en los cuáles se ejecutaron los experimentos utilizando las estructuras BS, LS, WT, WTH, GMR, AP.

La Tabla 4.1 muestra los resultados obtenidos con el árbol *Proteins*. Se muestra en las tablas el espacio teórico utilizado por las estructuras, sin considerar los términos $o(\cdot)$ de orden inferior. Junto a estos, se muestra el espacio utilizado por los árboles cardinales compactos en la práctica. Y por último, se indica en las descripciones el tamaño utilizado por las componentes teóricas $2n$, $n \log \sigma$ y nH_o .

Además de las que se presentan en este capítulo, en el Apéndice 4.1 se muestran tablas similares para los restantes conjuntos de datos.

Un factor importante que ayuda a determinar el grado de compresión final de la secuencia de símbolos de un árbol cardinal, es la entropía de orden cero. Este factor considera la frecuencia de cada símbolo y el valor resultante se puede considerar como la cantidad de bits promedio de información que contiene cada símbolo. De esta manera un símbolo muy repetido (es decir, aporta poca información) se codificará con una menor cantidad de bits, y para un símbolo poco repetido se utilizarán más bits en su representación. Dado que su frecuencia es baja, no impacta mayormente en el espacio utilizado. Esto es válido para las estructuras que comprimen, como lo son *Huffman*, *Shaped Wavelet Tree* y *Alphabet Partitioning*.

Por otra parte, esto induce a pensar que debido a estos resultados aún existe margen para mejorar, por ejemplo, reduciendo las constantes teóricas, y obtener de esta manera estructuras aún más pequeñas. Aunque esto es cierto, además se tiene que considerar el rendimiento de la estructura con respecto al tiempo de acceso, y

Tabla 3.5: El término teórico $2n$ en el uso de espacio equivale a 95.892.334 bits, mientras que $n \lg \sigma$ es 375.253.001 bits. Esto da un total teórico de 9,8265 bits por nodo. Por otro lado, $H_0 = 5,03$, por lo que $nH_0 = 241,305,387$. Esto equivale a 7,0328 bits por nodo en teoría.

English	Bits por Nodo teórico	Bits por Nodo Práctico
Estructuras	(No incluye términos $o(\cdot)$ de orden inferior)	(Incluye términos de orden inferior)
LS	9,8265	10,5357
BS	9,8265	10,5357
WT	9,8265	14,4304
GMR	9,8265	17,0103

English	Bits por Nodo teórico	Bits por Nodo Práctico
Estructuras	(No incluye términos $o(\cdot)$ de orden inferior)	(Incluye términos de orden inferior)
WTH	7,0328	10,0662
AP	7,0328	9,5267

lograr el balance óptimo. Cabe agregar que para la implementación de los árboles cardinales propuestos, la operación *access* no se utiliza, por lo que su rendimiento no influye en los resultados.

Los árboles con alfabetos más pequeños ($\sigma = 30$, como el caso de *Proteins* y *DNA* (ver Tabla 3.6)), del orden de las decenas, son los que logran las mejores tasas de compresión, alcanzando un ahorro de un 48,49 %. El árbol English es la excepción, dónde se logra un ahorro de un 9,57 %, con un tamaño de alfabeto de $\sigma = 227$. Esto se contrasta con que en árboles con alfabetos que superan los $\sigma = 130$ símbolos, las estructuras utilizan más espacio del que utilizan los algoritmos basados en Búsqueda Lineal y búsqueda binaria, debido al **overhead** existente. Esto da como resultado que las estructuras que debiesen compactar e incluso comprimir los datos, en realidad utilizan más espacio. En consecuencia, se obtiene tanto redundancia como compresión, en árboles con alfabetos en el orden de las decenas como de las centenas.

Además, se observa que la diferencia existente entre los valores teóricos y prácticos varían de manera similar en todos los árboles. Por lo tanto, no se puede concluir que algún tamaño de alfabeto, o cantidad de nodos, afecte de manera significativa en la cantidad de bits utilizada por nodo.

De los experimentos se tiene que la estructuras que comprimen como Alphabet Partitioning y Huffman Shaped Wavelet Tree, son las que alcanzan un mayor ahorro promedio en cuánto a espacio de los datos con un 11,65 % y un 7,72 % de ahorro respectivamente. Esto considerando todos los árboles, de los cuáles solo en 3 de los 6 utilizados se logra reducir el espacio utilizado. Wavelet Tree sólo logra reducir el espacio utilizado en 2 árboles (DNA y Proteins). Además en promedio, utiliza un 14,74 % más de espacio que los algoritmos de búsqueda, los cuales no comprimen o compactan los símbolos.

Tabla 3.6: El término teórico $2n$ en el uso de espacio equivale a 60.272.746 bits, mientras que $n \lg \sigma$ es 123.181.305 bits. Esto da un total teórico de 6,0875 bits por nodo. Por otro lado, $H_0 = 1,99$, por lo que $nH_0 = 60,120,256$. Esto equivale a 3,9949 bits por nodo en teoría.

DNA	Bits por Nodo teórico	Bits por Nodo Práctico
Estructuras	(No incluye términos $o(\cdot)$ de orden inferior)	(Incluye términos de orden inferior)
LS	6,0875	10,5324
BS	6,0875	10,5324
WT	6,0875	8,4767
GMR	6,0875	15,2027

DNA	Bits por Nodo teórico	Bits por Nodo Práctico
Estructuras	(No incluye términos $o(\cdot)$ de orden inferior)	(Incluye términos de orden inferior)
WTH	3,9949	5,8404
AP	3,9949	5,4249

Se observa también, resultados no esperados en los árboles *Sources* (ver Tabla 3.7), *XML* y *MIDI*, al utilizar los algoritmos Búsqueda Lineal y búsqueda binaria. Estos algoritmos, los cuales no utilizan compresión, obtienen un espacio menor que algunas de las estructuras construidas para almacenar los símbolos en menor espacio. Uno de los factores de este hecho, es la redundancia que se genera al intentar comprimir un alfabeto pequeño (200 símbolos). En estos casos, es posible que la estructura generada para almacenar los símbolos de forma eficiente, utilice más espacio del que se logra comprimir o compactar.

Por último, en cuanto a espacio, la estructura GMR es la que obtiene, a lo largo de todos los árboles utilizados en esta primera parte, los peores resultados, utilizando en promedio un 54, 12 % más de espacio. Es la estructura que más redundancia genera, utilizando más bits por nodo, que cualquiera de las otras estructuras utilizadas. Cabe agregar que no logra ahorro en espacio con ningún árbol, en esta primera parte.

Respecto al rendimiento

Es esta sección de discutirán los resultados obtenidos, teniendo como objetivo el rendimiento alcanzado por las estructuras.

Utilizando el algoritmo de búsqueda binaria, en general se obtienen los mejores resultados. Se sobrepone a las demás estructuras y algoritmos, como la opción más rápida. Aunque tiene en contra el hecho de que no comprime los datos, ésta es la que obtiene el mejor rendimiento en los 6 árboles utilizados en esta primera parte.

Tabla 3.7: El término teórico $2n$ en el uso de espacio equivale a 41.387.962 bits, mientras que $n \lg \sigma$ es 162.483.945 bits. Esto da un total teórico de 9,8517 bits por nodo. Por otro lado, $H_0 = 5,83$, por lo que $nH_0 = 120,576,171$. Esto equivale a 7,8266 bits por nodo en teoría.

Sources	Bits por Nodo teórico	Bits por Nodo Práctico
Estructuras	(No incluye términos $o(\cdot)$ de orden inferior)	(Incluye términos de orden inferior)
LS	9,8517	10,5396
BS	9,8517	10,5396
WT	9,8517	14,4435
GMR	9,8517	17,0151

Sources	Bits por Nodo teórico	Bits por Nodo Práctico
Estructuras	(No incluye términos $o(\cdot)$ de orden inferior)	(Incluye términos de orden inferior)
WTH	7,8266	11,2594
AP	7,8266	10,8414

En general, todas las estructuras son entre un 85,20 % y un 125,50 % más lentas que el rendimiento alcanzado por Búsqueda Binaria. Es por esto que búsqueda binaria se toma como la estructura principal de comparación. Luego del algoritmo de búsqueda binaria, el algoritmo que le sigue en rendimiento es Búsqueda Lineal, que obtiene resultados comparables y muy similares a búsqueda binaria. A priori, se considera inesperado este resultado, debido a que este algoritmo no utiliza ninguna estructura especializada para buscar los símbolos. Sin embargo, logra mejor rendimiento que las estructuras que utilizan índices y espacio adicional para realizar las búsquedas.

Esto, sin embargo, se explica debido a la baja aridez promedio de los árboles. Esto significa que la búsqueda que se realiza para lograr la posición del nodo dado un símbolo, se realiza buscando sólo entre 2 o 3 símbolos en la mayoría de casos en los experimentos realizados. Además, Búsqueda Lineal en comparación a búsqueda binaria, en una lista de 2 o 3 elementos logra en muchos casos un mejor desempeño. Esto da como resultado lo que se observa en los árboles: el rendimiento de búsqueda binaria no es notoriamente mayor que el rendimiento de Búsqueda Lineal.

Dentro de las estructuras de datos compactas, las que le siguen en rendimiento son Huffman Shaped Wavelet Tree y GMR. Un factor que explica estos resultados, es la cantidad de operaciones que aún perteneciendo a $O(1)$, generan suficiente overhead y terminan siendo menos eficientes en tiempo que Búsqueda Lineal o búsqueda binaria. Es decir, el tiempo que se ocupa en procesamiento (acceso a los índices), es superior al tiempo que toma buscar el símbolo requerido.

Se observa que en promedio, Huffman Shaped Wavelet Tree es un 85,20 % más lento que búsqueda binaria, y GMR un 89,24 %. Éstas son las estructuras que mejor se comportan en cuánto a rendimiento.

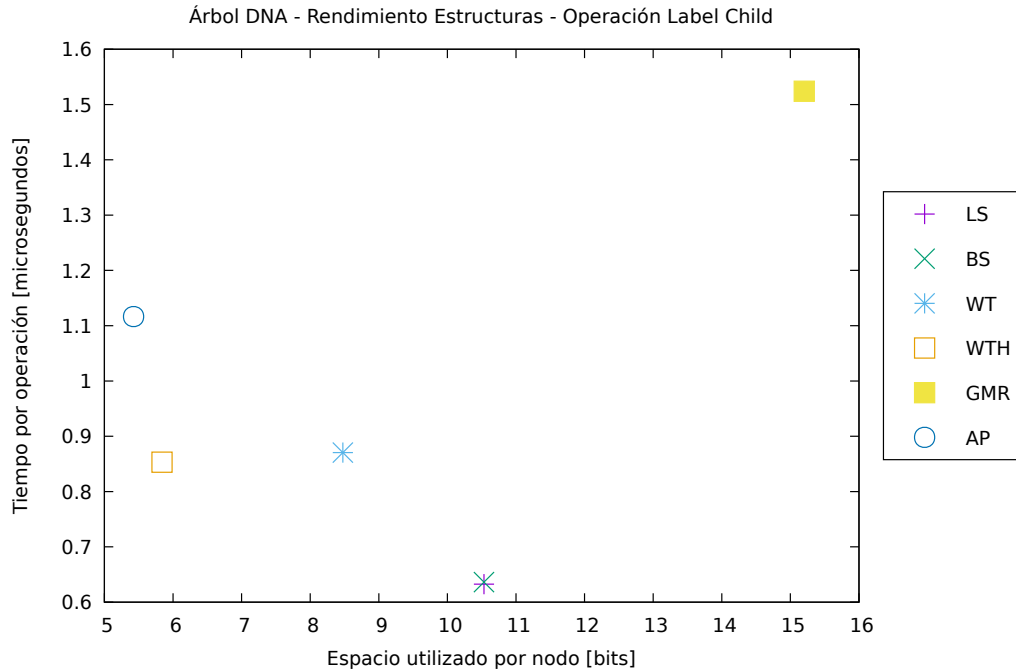


Figura 3.1: Gráfico Tiempo vs Espacio - Árbol DNA.

Se observa también que Alphabet Partitioning presenta los peores resultados. En promedio, es un 125,50 % más lento que búsqueda binaria, esto se debe a que este método es más eficiente para alfabetos grandes, lo cual no es el caso de estos primeros experimentos.

Por último, cabe agregar que Wavelet Tree no obtiene los mejores resultados, aunque tampoco los peores: su rendimiento es mejor que el de Alphabet Partitioning, pero no superior al de Huffman Shaped Wavelet Tree o GMR. Es un 96,32 % más lento que búsqueda binaria.

Análisis de Gráficos

En los siguientes gráficos se pueden apreciar los resultados obtenidos en cuanto a espacio y tiempo, de todas las estructuras utilizadas. Los gráficos se realizaron de tal manera que el eje x corresponde al espacio en bits que ocupa cada nodo. El eje y representa el tiempo que toma realizar la operación *label-child*. En resumen, un punto más a la izquierda utiliza menos espacio que uno más a la derecha y un punto más abajo tiene mejor rendimiento que un punto situado por sobre éste.

Se tienen distintos resultados, pero se agruparán según el desempeño en cuanto a tiempo. Se observa que los gráficos de los árboles Proteins y DNA (ver gráfico en la Figura 3.1), presentan resultados similares. El desempeño de las estructuras de la más veloz a la más lenta es el siguiente: BS, LS, WTH, WT, AP, GMR.

DNA utiliza mayoritariamente 4 caracteres, que son A,C,G,T, y se puede observar que el espacio utilizado por Alphabet Partitioning y Huffman Shaped Wavelet Tree, que son las estructuras que mejor se comportan en esta materia, utilizan menos de 6 bits por nodo. Esto indica que la compresión es importante.

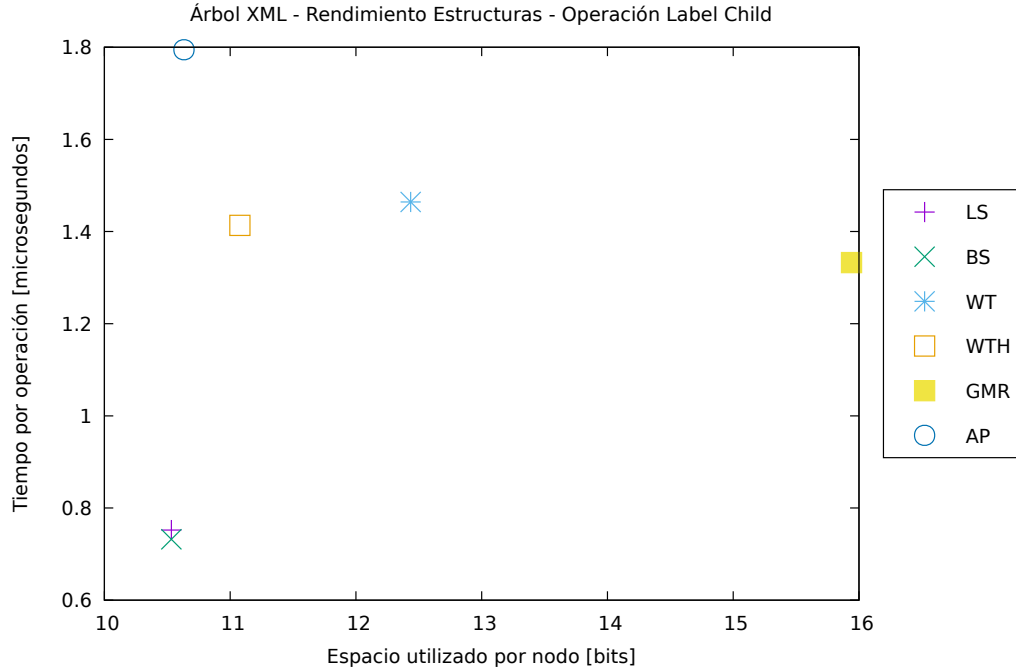


Figura 3.2: Gráfico Tiempo vs Espacio - Árbol XML.

Se pasa de utilizar 8 bits para codificar el alfabeto más los 2 bits pertenecientes a la topología del árbol, un total de 10 bits, a usar 6 bits para representar un nodo. La entropía de orden cero $H_0 = 1,99$ indica que a lo largo del árbol, la ocurrencia de cada símbolo es similar a si solo fuesen 4 símbolos. Pero es sabido que el tamaño del alfabeto es 17, por lo que el grado de compresión es alto. En este árbol, son búsqueda binaria y Búsqueda Lineal los algoritmos que destacan consiguiendo los mejores rendimientos en cuanto a tiempo.

Para dimensionar la utilización de espacio, se tiene el siguiente ejemplo. Para codificar los símbolos ASCII, se utilizan 8 bits, lo que considera un alfabeto de 256 símbolos. Pero el árbol Proteins tiene un alfabeto de 27 símbolos, por lo que la cantidad mínima de bits necesarios para representar cada símbolo inequívocamente es $\lceil \lg 27 \rceil = \lceil 4,75 \rceil$, es decir, 5 bits. Además, la cantidad de bits utilizados para representar la secuencia de paréntesis utiliza 2 bits por nodo. Por lo tanto, se tiene que para el árbol Proteins en particular, se requieren 7 bits para representar un nodo, o 10 bits si se utilizan 8 bits para representar los símbolos y no se compacta su representación. Esto es sin ninguna estructura de dato que optimice la utilización de operaciones como las listadas en la Tabla 1.1. Si se observan los resultados obtenidos, se tiene que con las estructuras Alphabet Partitioning y GMR se utilizan menos de 6 bits. De esta manera, es posible dimensionar los logros obtenidos en cuanto a eficiencia de estas estructuras. Se utilizan menos bits que el mínimo sin comprimir, dado que al menos se compacta la representación de los símbolos, y además se soporta gran cantidad de operaciones sobre el árbol.

En los árboles English, XML, MIDI, los mejores rendimientos se obtuvieron en el siguiente orden: BS, LS, GMR, WT, WTH y AP. En los árboles Sources, XML, MIDI (ver gráfico de la Figura 3.3 para Sources) se da el caso, donde los algoritmos BS y LS, utilizan menos espacio que las demás estructuras. Aunque los algoritmos de búsqueda bordean los 10 bits de espacio por nodo, las otras estructuras utilizan más bits, siendo

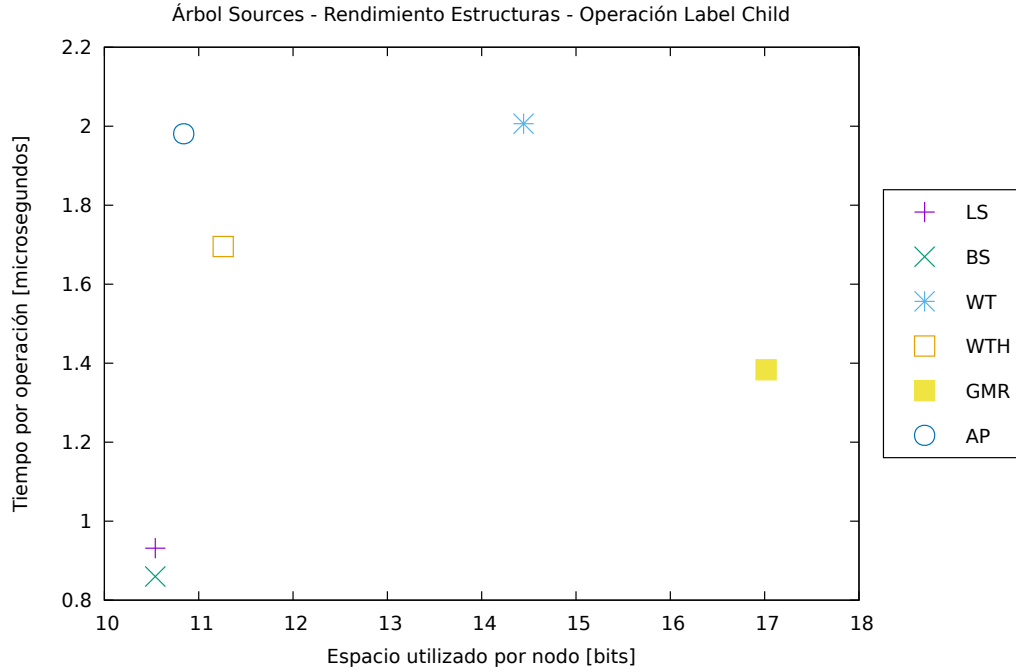


Figura 3.3: Gráfico Tiempo vs Espacio - Árbol Sources.

GMR la que más utiliza (15,94 en XML, 17,05 en MIDI y 17,01 en Sources). En estos árboles se observa, el fenómeno antes descrito, que debido al overhead, se utiliza más espacio que al almacenar los datos sin comprimir, como se hace con los algoritmos de búsqueda. En el árbol English, Alphabet Partitioning utiliza 9,52 bits por nodo, variando las demás estructuras entre 10,06 y 17,01. Logra comprimir como ninguna de las otras estructuras, pero su rendimiento en comparación a Búsqueda Binaria, un algoritmo que no comprime, es muy inferior (los tiempos de Alphabet Partitioning duplican los tiempos de Búsqueda Binaria). En este árbol, Alphabet Partitioning es la estructura que menos espacio utiliza, pero peor rendimiento obtiene.

Específicamente en el árbol Sources, como ya se comentó previamente, las estructuras se comportan muy mal en cuanto a espacio. Ninguna es capaz de superar el espacio utilizado por los algoritmos de búsqueda. Se observa que Alphabet Partitioning y Huffman Shaped Wavelet Tree son las estructuras que se acercan al espacio utilizado que al menos debiesen alcanzar, y también se observa que GMR utiliza casi el doble de espacio que los datos originales.

En general, en esta primera parte de los experimentos, se observa que el espacio utilizado por nodo por la estructura GMR es excesivo en comparación a las demás. Por otra parte, Búsqueda Binaria y lineal obtienen los mejores resultados en cuanto a rendimiento, pero no se tiene el beneficio de la compresión. Por último, los mejores resultados en cuanto a espacio utilizado, los obtiene Alphabet Partitioning, que alcanza valores cercanos a los teóricos, y además soporta las operaciones de árbol cardinal. Sin embargo, es la estructura que peor desempeño presenta.

En vista que los resultados obtenidos no son concluyentes, lo que se decidió realizar a continuación, es

generar los 3 árboles aleatorios que se encuentran descritos en la Tabla 3.2 y Tabla 3.1, además de utilizar árboles obtenidos a partir de los datos de la Wikipedia. Estos árboles cuentan con una aridad promedio mucho mayor en sus nodos, y alfabetos del orden de los miles. Con estos árboles se espera clarificar el rendimiento obtenido por las estructuras. También dado que los árboles poseen una baja aridad promedio, que es el factor que explica el buen rendimiento de *Búsqueda Lineal*, se podrá verificar si en realidad es una buena opción, por lo menos, en cuánto a rendimiento, cuando la aridad alcance un orden de magnitud mayor.

3.2.2. Árboles con alfabetos grandes

En esta sección se experimentará con árboles generados aleatoriamente, así como también con árboles generados al ejecutar el algoritmo de compresión LZ78 sobre una versión de la Wikipedia. Estos son árboles con gran cantidad de nodos, alfabetos grandes y aridad elevada. Se espera con esto conseguir las respuestas que con alfabetos pequeños quedaron inconclusas.

En los resultados obtenidos con los árboles generados, son varios los aspectos que se tienen que considerar. Al igual que para los alfabetos pequeños, primeramente se realizará una comparación teórico / práctica de los resultados obtenidos.

Comparación Teórico / Práctico

Nuevamente se contrastarán los resultados obtenidos, observando las cotas en espacio de los valores teóricos, y los resultados que se obtuvieron en la práctica.

Además de las tablas que se presentan en este capítulo, en el Apéndice 4.1 se muestran las tablas correspondientes a los restantes conjuntos de datos.

Las tablas construidas siguen el mismo formato que las utilizadas para describir los resultados de los árboles con alfabetos pequeños.

Primeramente, se observa de los resultados que la cantidad de bits utilizada por los algoritmos de *Búsqueda Binaria* y *Búsqueda Lineal*, superan ampliamente a los valores teóricos descritos en las tablas. Esto ocurre debido a que para estos algoritmos, los símbolos se representaron como un arreglo de valores enteros sin signo de 32 bits cada uno. No se utilizó ninguna forma de compresión o alguna forma de codificar el alfabeto con la cantidad de bits óptimo, que en este caso sería de $\lceil \lg \sigma \rceil = 23$ bits para el alfabeto más grande. Es decir, mientras que en el análisis teórico se considera un uso óptimo de bits para representar el alfabeto, $n \lg \sigma$, en la práctica se utilizan 32 bits para representarlos.

Se tiene también que el espacio utilizado por la estructura *Alphabet Partitioning* y *GMR*, son las que mejor se comportan, teniendo en cuenta la diferencia existente entre los **bits por nodo teórico** y los **bits por nodo práctico** descritos en las tablas. Son estas dos estructuras las que consiguen una menor diferencia, que varía alrededor de los 8 bits, y se comportan de manera homogénea dentro de los árboles generados. Las demás estructuras no mantienen su tasa de compresión, a medida que aumenta el tamaño del alfabeto.

Tabla 3.8: El término teórico $2n$ en el uso de espacio equivale a 908.002.948 bits, mientras que $n \lg \sigma$ es 9.049.121.961 bits. Esto da un total teórico de 21,9319 bits por nodo. Por otro lado, $H_0 = 19,93$, por lo que $nH_0 = 9,049,021,179$. Esto equivale a 21,9317 bits por nodo en teoría.

random1Mv200K	Bits por Nodo Teórico	Bits por Nodo Práctico
Estructuras	(No incluye términos $o(\cdot)$ de orden inferior)	(Incluye términos de orden inferior)
LS	21,9319	34,5288
BS	21,9319	34,5288
WT	21,9319	32,3474
GMR	21,9319	29,2489

random1Mv200K	Bits por Nodo Teórico	Bits por Nodo Práctico
Estructuras	(No incluye términos $o(\cdot)$ de orden inferior)	(Incluye términos de orden inferior)
WTH	21,9317	33,9678
AP	21,9317	32,3415

Cómo se vio con árboles de alfabetos pequeños, **Wavelet Tree** no se comporta de la mejor manera. De hecho, agrega redundancia, por lo que el espacio utilizado es mayor al de los datos originales. Sin embargo, en árboles donde el tamaño de los alfabetos es mucho más grande, se tiene que **Wavelet Tree** tiene un comportamiento diferente. Se observa que alcanza buenas tasas de compresión, ahorrando en promedio un 22,27 % de espacio en los árboles aleatorios con respecto a los datos originales. Sin embargo, en cuanto a los árboles de Wikipedia (ver Tablas 3.9 y 3.10), en promedio **Wavelet Tree** utiliza un 0,48 % extra de espacio. Se observa que este decaimiento en su capacidad de compresión se debe al tamaño del alfabeto. Cuando el tamaño del alfabeto alcanza los $\sigma = 663,091$, en el árbol wiki-50MB (ver Tabla 4.10), el espacio ahorrado cae de un 36,71 % y 23,79 %, que es lo conseguido en árboles con alfabeto hasta $\sigma = 50,000$ a un 6,60 %. Luego cuando el alfabeto aumenta a $\sigma = 1,000,245$ decae nuevamente a un 6,31 %. Por último, cuando el tamaño del alfabeto supera los $\sigma = 3,125,941$, esta estructura comienza a utilizar más espacio, que los algoritmos de búsqueda.

La estructura GMR es la que obtiene las mejores tasas de compresión promedio. Para los árboles aleatorios, donde los tamaños de los alfabetos varían de $\sigma = 5,000$ hasta $\sigma = 1,000,245$ el ahorro de espacio promedio conseguido es de 27,54 %. Superando a las demás estructuras por casi un 5,0 %. Otro aspecto importante que se observa, es que el porcentaje de espacio ahorrado en el árbol Random1Mv200k (ver Tabla 3.8), con alfabeto mayor a un millón, es 15,29 % similar al espacio promedio ahorrado con los árboles Wikipedia, donde los alfabetos van de los $\sigma = 600,000$ hasta un poco más de los $\sigma = 5,000,000$, donde se consiguen un 11,90 % de ahorro promedio. Con alfabetos de estos órdenes de magnitud (en el orden de los miles, hasta los cientos de miles) es GMR la estructura que demuestra superioridad en cuanto a ahorro de espacio. Se observó que su rendimiento en alfabetos pequeños, es muy inferior a las demás estructuras. Por otra parte, cabe destacar la tendencia de esta estructura a disminuir su rendimiento en cuanto a espacio. Se pasa de conseguir

Tabla 3.9: El término teórico $2n$ en el uso de espacio equivale a 79.138.862 bits, mientras que $n \lg \sigma$ es 853.744.469 bits. Esto da un total teórico de 23,5759 bits por nodo. Por otro lado, $H_0 = 15,25$, por lo que $nH_0 = 603,520,875$. Esto equivale a 17,2522 bits por nodo teórico.

Wikipedia-500MB	Bits por Nodo Teórico	Bits por Nodo Práctico
Estructuras	(No incluye términos $o(\cdot)$ de orden inferior)	(Incluye términos de orden inferior)
LS	23,5759	34,5371
BS	23,5759	34,5371
WT	23,5759	35,1892
GMR	23,5759	30,9387

Wikipedia-500MB	Bits por Nodo Teórico	Bits por Nodo Práctico
Estructuras	(No incluye términos $o(\cdot)$ de orden inferior)	(Incluye términos de orden inferior)
WTH	17,2522	85,9200
AP	17,2522	25,6496

un 27,54 % de ahorro promedio de espacio, a un 11,90 %, al aumentar el tamaño del alfabeto al tamaño de los millones. Es prudente aseverar que el desempeño en cuanto a espacio de GMR, tiende a disminuir al seguir aumentando el tamaño del alfabeto.

En cuánto a *Huffman Shaped Wavelet Tree*, se observa que su rendimiento respecto a espacio, con alfabetos en el orden de los miles, presenta un ahorro promedio de espacio de un 22,02 %, y es en este rango, donde obtiene su mejor desempeño en cuánto a espacio. Por otra parte, cabe agregar que esta estructura, al trabajar sobre árboles en los que el tamaño del vocabulario asciende por encima de los $\sigma = 600,000$ y el millón de símbolos, su desempeño en cuánto a espacio decae considerablemente. Su desempeño como se observa en los árboles Wikipedia, no es competitivo, llegando a utilizar más espacio que los algoritmos de búsqueda Lineal y Binaria. En promedio utiliza un 181,03 % extra de espacio lo cual es inaceptable.

La última estructura que queda por abordar es *Alphabet Partitioning*. Se observó que para alfabetos pequeños lograba ahorrar en promedio un 11,65 % de espacio. Para los árboles aleatorios, el ahorro de espacio aumenta a 23,26 %, estando solo por debajo del rendimiento de GMR. La principal ventaja de *Alphabet Partitioning* sobre las demás estructuras, es que es la única que muestra una tendencia de mejorar su tasa de compresión. En los árboles Wikipedia, alcanza un ahorro de espacio promedio de un 25,50 %, el más alto conseguido comparado con todas las estructuras en cuestión. Además es la única que logra su mejor rendimiento en cuánto a espacio, en los árboles que poseen los alfabetos más grandes testeados. Esto induce a conjeturar que aún existe un margen de ganancia en cuánto a espacio, para *Alphabet Partitioning*.

Otra observación importante que no ocurre en los alfabetos pequeños, es que la cantidad de espacio utilizado por las estructuras compactas, no superan la cantidad de espacio utilizada por los algoritmos de búsqueda (Binaria y Lineal) en mayor cantidad de casos. De esta manera, se puede ver cómo la capacidad de

Tabla 3.10: El término teórico $2n$ en el uso de espacio equivale a 161.728.612 bits, mientras que $n \lg \sigma$ es 1.806.433.882 bits. Esto da un total teórico de 24,3391 bits por nodo. Por otro lado, $H_0 = 15,34$, por lo que $nH_0 = 1,240,450,368$. Esto equivale a 17,2522 bits por nodo teórico.

Wikipedia-1GB	Bits por Nodo Teórico	Bits por Nodo Práctico
Estructuras	(No incluye términos $o(\cdot)$ de orden inferior)	(Incluye términos de orden inferior)
LS	24,3391	34,5369
BS	24,3391	34,5369
WT	24,3391	36,6745
GMR	24,3391	31,6471

Wikipedia-1GB	Bits por Nodo Teórico	Bits por Nodo Práctico
Estructuras	(No incluye términos $o(\cdot)$ de orden inferior)	(Incluye términos de orden inferior)
WTH	17,2522	75,7711
AP	17,2522	25,7046

compresión de las estructuras aumenta a medida que aumenta el tamaño del alfabeto, y los resultados y estructuras se comportan de mejor manera. Las estructuras que fueron creadas para utilizar el tamaño óptimo para representar los alfabetos e incluso comprimirlos, ahorran aproximadamente un 23,77 % de espacio en promedio del que utilizan los algoritmos de búsqueda (binaria, lineal), los cuáles no comprimen el árbol. Esto dentro del rango en el que las estructuras se comportan de mejor manera, es decir, cuando el tamaño del alfabeto es del orden de los miles.

Al experimentar con un alfabeto que se encuentra en el orden de los millones, solo AP mantuvo e incluso mejoró su tasa de compresión. Por otra parte, la capacidad de compresión de las demás estructuras decayó, y algunas como Huffman Shaped Wavelet Tree lo hicieron considerablemente.

Respecto al Rendimiento

En relación al rendimiento conseguido por las estructuras sobre árboles con alfabetos grandes, la tendencia de Búsqueda Binaria a ser el algoritmo más rápido, se mantiene y se afianza, al superar a las demás estructuras ahora con alfabetos mucho más grandes. Se observa también que a medida que aumenta el tamaño del alfabeto, la ventaja que obtiene por sobre las demás estructuras, se hace aún mayor.

En alfabetos pequeños, la ventaja obtenida por Búsqueda Binaria con respecto a las demás estructuras era considerable. Se tenía que estas estructuras eran entre un 85,20 % y un 125,50 % más lentas que Búsqueda Binaria. Pero cuando se refiere a los alfabetos grandes, esta diferencia es aún mayor. Las estructuras son entre un 52,91 % y un 249,30 % más lentas.

Con respecto a **Búsqueda Lineal**, depende mucho de la aridez de los nodos del árbol. En general, no es una buena opción. Es la estructura más lenta y no ofrece ahorrar espacio en los nodos del árbol.

En promedio, la estructura más rápida y con una gran ventaja con respecto a las demás estructuras es **GMR**. En promedio, resultó ser solo un 52,91 % más lenta que **Búsqueda Binaria**. En cuanto al tamaño del alfabeto, se comporta de mejor manera cuando se trabaja con alfabetos en los órdenes de los miles y millones.

La estructura que sigue en rendimiento a **GMR** es **Huffman Shaped Wavelet Tree**. Esta estructura es un 228,53 % más lenta que **Búsqueda Binaria**. Aunque la tendencia no es acentuada, se observa que a medida que aumenta el tamaño del alfabeto, el rendimiento disminuye. Un antecedente de esta aseveración, es que en los árboles con alfabeto pequeño antes vistos, esta estructura es solo un 85,20 % más lenta que **Búsqueda Binaria**.

Alphabet Partitioning es en promedio un 268,41 % más lenta que **Búsqueda Binaria**. Con respecto a los árboles con alfabetos pequeños, mejoró su rendimiento en comparación a las otras estructuras. Pasó de ser la estructura más lenta, a subir una posición con respecto a las demás estructuras.

Luego se tiene que **Wavelet Tree**, en promedio es un 296,30 % más lenta que **Búsqueda Binaria**. Se observa que su rendimiento decae, cuando se aumenta el tamaño del alfabeto σ . Resulta ser la estructura más lenta, en árboles con alfabetos más grandes. Con respecto a los árboles con alfabetos pequeños, bajó una posición que antes era de **Alphabet Partitioning**.

Cabe agregar, que el rendimiento de todas las estructuras y algoritmos en estudio, disminuye cuando se someten a árboles con alfabetos más grandes.

Análisis de Gráficos

Los gráficos generados, en los cuáles se registra el espacio y tiempo para cada estructura, facilitan en gran manera el determinar cómo se comportan las estructuras y el determinar las diferencias entre ellas. A diferencia de los árboles con alfabetos pequeños, estos gráficos se distinguen en mayor medida, siendo las diferencias en cuanto a las métricas, mucho más evidentes.

Lo que primeramente se observa en estos árboles, es que la capacidad de compresión se comporta de mejor manera que para los alfabetos pequeños. Todas las estructuras tienden a comportarse de mejor manera a medida que se aumenta el tamaño del alfabeto. Pero también se observan sus limitancias, dado que al aumentar demasiado el tamaño del alfabeto, las estructuras pierden su tasa de compresión y los beneficios que presentan, dejando que los algoritmos de búsqueda se observen mejor situados en los gráficos.

Se observa claramente, que los algoritmos de búsqueda están muy por debajo en cuanto a espacio utilizado para representar un nodo del árbol, pero el desempeño que obtiene **Búsqueda Binaria** en específico, no se puede comparar al rendimiento de las demás estructuras. Su comportamiento es superior y los resultados lo demuestran categóricamente.

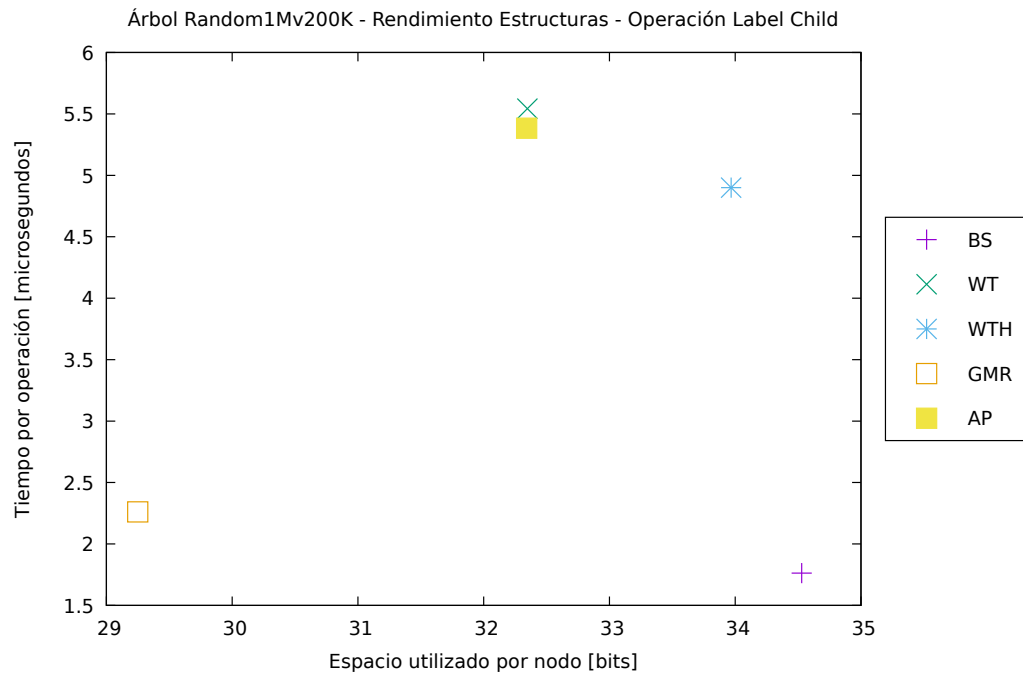


Figura 3.4: Gráfico Tiempo vs Espacio - Árbol Random1Mv200K. El punto referente a LS (34, 5288; 241, 2730), se excluyó debido a que escapa de las dimensiones del gráfico.

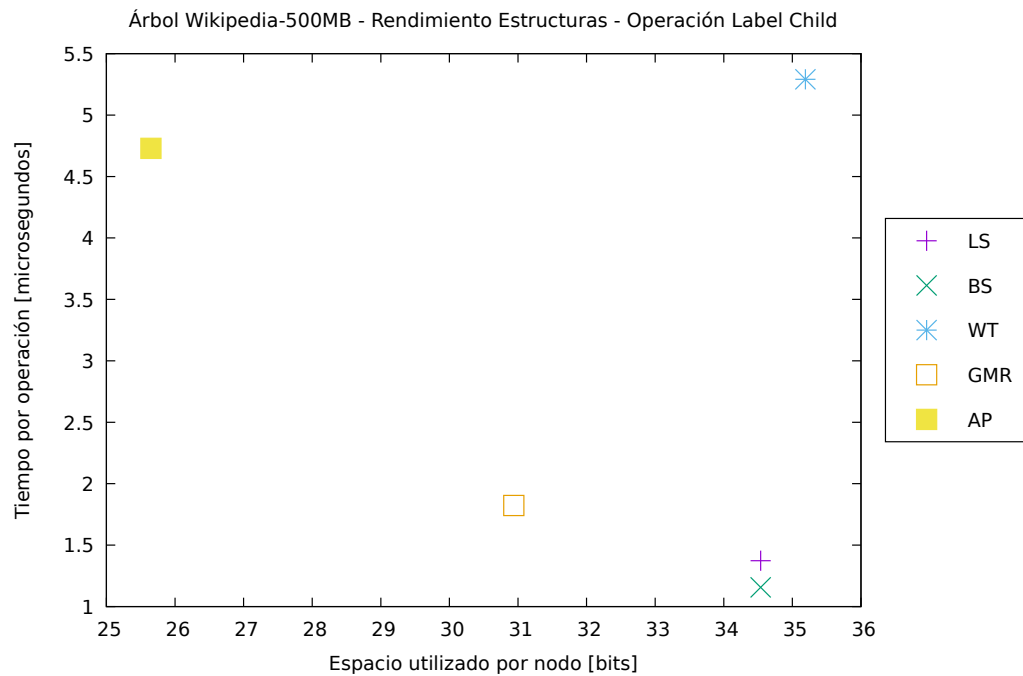


Figura 3.5: Gráfico Tiempo vs Espacio - Árbol Wikipedia-500MB. El punto referente a WTH (85, 9200; 4, 1112), se excluyó debido a que escapa de las dimensiones del gráfico.

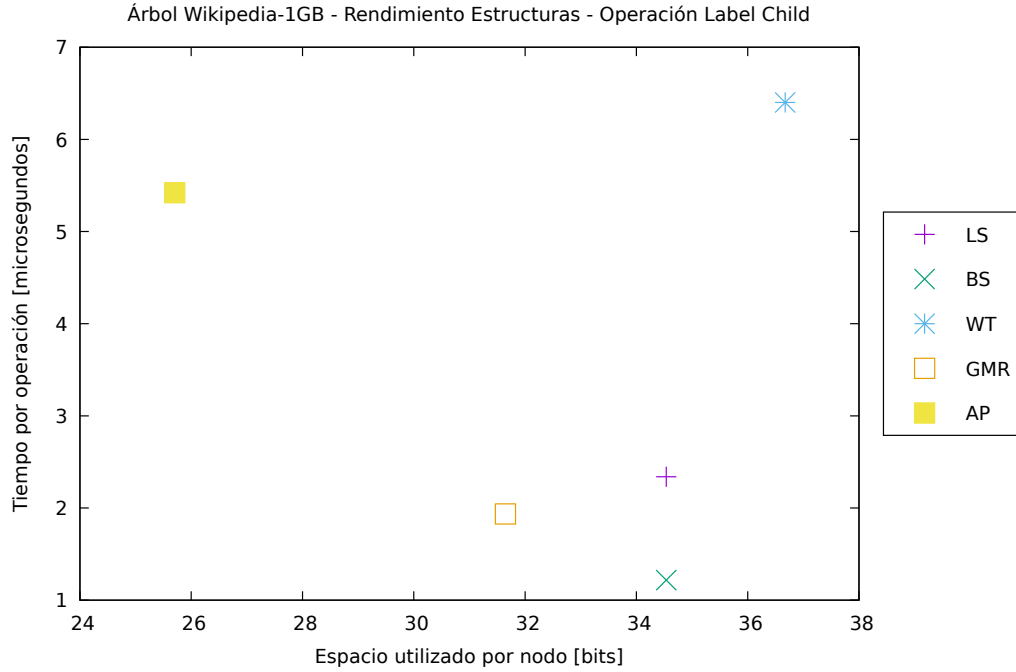


Figura 3.6: Gráfico Tiempo vs Espacio - Árbol Wikipedia-1GB. El punto referente a WTH (75,7711; 5,1224), se excluyó debido a que escapa de las dimensiones del gráfico.

En cuanto a **Búsqueda Lineal**, sus resultados varían mucho. Se observa que su rendimiento está directamente relacionado a la aridad promedio del árbol. En algunos gráficos como el de **Wikipedia-500MB** (ver Figura 3.5) o el de **Random5Kv1K** (ver Figura 4.7) **Búsqueda Lineal** se comporta muy bien, en cuanto a rendimiento, encontrándose dentro de las opciones más rápidas. Esto se explica debido a que la aridad de estos árboles no es tan elevada, por lo que el tiempo de búsqueda, aunque crece linealmente, no impacta mayormente. En el árbol **Random1Mv200K** la aridad promedio es de 200,000, por lo que realizar **Búsqueda Lineal** cada vez, para encontrar un símbolo no pasa desapercibido, e impacta en el rendimiento como se observa en el Gráfico 3.4.

En el árbol Wikipedia-1GB y Wikipedia-50MB ocurre algo distinto. El rendimiento de **Búsqueda Lineal** decae, siendo que la aridad promedio de este árbol es de 5,51 nodos. Lo que explica en este árbol el pésimo rendimiento de **Búsqueda Lineal**, es la altura promedio y la aridad del nodo padre. En las pruebas se recorre el árbol hasta un nodo hoja y luego se vuelve al nodo padre. Este árbol tiene altura promedio 3, por lo que frecuentemente se está realizando búsquedas en el nodo padre, que presenta la mayor aridad dentro del árbol con 2,712,105 nodos hijos. Es por esto que en este caso la aridad promedio no es un antecedente para predecir el mal rendimiento de **Búsqueda Lineal**. En el árbol Wikipedia-50MB ocurre exactamente lo mismo.

Con respecto a las estructuras predominantes en cuanto a espacio, se observa que la presencia de **Alphabet Partitioning** y **GMR** ocurren en mayor medida. No obstante, la presencia de otras estructuras como **Huffman Shaped Wavelet Tree** también muestra su notoriedad, debido a que en ocasiones, se comporta de manera similar a las estructuras antes mencionadas, y en otros casos su desempeño en cuanto espacio es inesperado

e indeseado, cómo es el caso del árbol Wikipedia-1GB.

Huffman Shaped Wavelet Tree en los árboles Wikipedia, muestra un decremento considerable en cuanto al espacio utilizado. Una posible explicación a este fenómeno es la procedencia de los árboles. Al ser obtenidos al utilizar el algoritmo de compresión LZ78, es posible que al utilizar además el algoritmo de Huffman, se genere redundancia en vez de comprimir aún más los símbolos. Un escenario similar ocurre en los sistemas de archivos, que al intentar comprimir un conjunto de datos ya comprimido, resulta en un archivo de tamaño final que utiliza más espacio que el archivo original, en la gran mayoría de ocasiones.

Se observa en los gráficos en el Apéndice 4.1, que la cantidad de espacio que se utiliza varía en alrededor de 32 bits, lo cual tiene sentido dado que se utilizan 32 bits para representar estos alfabetos. Huffman Shaped Wavelet Tree destaca en los gráficos, debido a su excesivo uso de espacio.

Las estructuras de datos compactas, obtienen en general mejores resultados, logrando mayor compresión. Comparado con los algoritmos de búsqueda, es evidente su superioridad. Por otra parte, los algoritmos de búsqueda, en particular Búsqueda Binaria, es la que obtiene los mejores resultados en cuanto a rendimiento, en todos los árboles.

Por último, la estructura que utiliza la menor cantidad de bits promedio para representar un nodo del árbol cardinal, es Alphabet Partitioning con 26,11 bits. Es seguido por GMR con 27,72, aunque ya es sabido que actúa bien solo en un rango de tamaños de vocabularios. Luego Wavelet Tree que utiliza 30,77 y por último Huffman Shaped Wavelet Tree con 62,00 bits, que es la estructura que más acotado rango tiene, con respecto a tamaños de vocabulario, para un correcto funcionamiento.

Conclusiones

En base a los experimentos realizados y el análisis provisto en la sección anterior, a continuación se presentan las conclusiones de los resultados obtenidos con el fin de determinar cuáles son las mejores estructuras de datos para la construcción de árboles cardinales compactos.

Como primer apronte, se tiene que el comportamiento de las estructuras en la práctica no es superior al desempeño teórico. Como ya se ha mencionado, el análisis de cotas de algoritmos deja fuera aspectos inherentes a la computación, lo que da una visión simplificada del comportamiento de los algoritmos en general y los resultados en estos experimentos no son la excepción. Con esto se apunta a que en general las estructuras poseen la misma cota de crecimiento en cuanto a complejidad en tiempo, pero aún así presentan diferentes resultados.

Los resultados en cuánto a espacio en alfabetos pequeños no fueron muy alentadores. Se observa que existió redundancia, con algunas estructuras donde la utilización de espacio estuvo lejos de ser la óptima. En cuánto al rendimiento conseguido en estos árboles, es búsqueda binaria el algoritmo de búsqueda que supera con creces el rendimiento de las estructuras de datos compactas.

Las estructuras de datos compactas que muestran los mejores resultados al trabajar con alfabetos pequeños son *Huffman Shaped Wavelet Tree* y *Alphabet Partitioning*. Dado que éstas son las estructuras que consiguen utilizar espacio comprimido. Comprimen un 7,72 % y un 11,65 % respectivamente. Sin embargo este es el desempeño promedio en cuánto a espacio. También ocurre, al utilizar estas estructuras, que se utiliza más espacio que el utilizado por los algoritmos de búsqueda, es decir, se obtiene redundancia de datos, una consecuencia que no es buscada. Donde realmente se obtiene ventaja al utilizar estas estructuras es en árboles con alfabetos en el orden de las decenas, donde en ningún caso se obtuvo redundancia. Sin embargo, estos alfabetos son muy pequeños como para lograr tener una aplicación real, pero este es el comportamiento que presentan al menos en árboles con alfabetos pequeños.

En cuánto a *GMR*, su utilización en alfabetos pequeños no se recomienda. Esta estructura en todos los árboles utiliza espacio extra. Es la estructura que peor se comporta en cuánto a espacio, utilizando siempre más espacio del necesario. En promedio utiliza un 54,12 % extra de espacio. También se tiene que *Wavelet Tree* utiliza un 14,74 % extra de espacio. Este comportamiento descarta a estas dos estructuras de ser opciones a construir árboles cardinales, al menos con alfabetos pequeños.

Una vez sabido cómo se comportan en cuánto a espacio, solo resta saber cual de éstas consigue mejor rendimiento. Es sabido, de la sección de resultados, que **Alphabet Partitioning** es la estructura que peor desempeño consigue en tiempo de operación. Dentro de las estructuras que son las mejores opciones para construir un árbol cardinal compacto debido a que destacan en el ahorro de espacio, se tiene que **Huffman Shaped Wavelet Tree** es un 85,20 % más lento que **Búsqueda Binaria**, y **Alphabet Partitioning** un 125,50 % más lento.

En cuánto a los resultados de **Búsqueda Binaria**, no se recomienda su uso. Su rendimiento depende directamente de la aridez promedio del árbol, teniendo baja aridez su rendimiento es ligeramente inferior al de **Búsqueda Binaria**. Pero si la aridez comienza a aumentar, rápidamente su rendimiento decae por debajo de todas las demás estructuras.

Por lo tanto, de los resultados obtenidos, se tiene que para árboles con alfabeto pequeño, las mejores opciones dependerán del contexto y las especificaciones buscadas. Dados los antecedentes descritos, si no se necesita compresión, lo cual en la práctica difícilmente ocurrirá este caso, el algoritmo de **Búsqueda Binaria** es una gran opción, dado que supera con creces el rendimiento de cualquier otra estructura. Al utilizar alfabetos pequeños, no más allá del orden de las centenas, lo adecuado es utilizar un **árbol cardinal compacto con Huffman Shaped Wavelet Tree** como estructura de datos que maneje los símbolos. Logra los mejores resultados en comparación a las demás estructuras.

Al utilizar las estructuras de datos compactas con alfabetos grandes, se hace evidente que el comportamiento de éstas, está condicionado al tamaño del alfabeto. Es cuando los alfabetos crecen cuando se pueden determinar los umbrales en los que las estructuras alcanzan sus mejores resultados.

En particular, los algoritmos de **Búsqueda Binaria** y binaria no utilizan compresión ni compactan la representación del alfabeto, por lo que es normal que obtengan los peores resultados en esta materia, como se ve en los resultados. **Búsqueda Binaria**, es el algoritmo que obtiene los mejores resultados en cuánto a rendimiento y supera a las demás estructuras que en promedio son entre 85,20 % y un 125,50 % más lentas que **Búsqueda Binaria**. Además, sus resultados en comparación a las estructuras de datos compactas, se vuelven superiores a medida que el tamaño del alfabeto aumenta. Por lo que su uso es recomendado cuando la utilización de espacio no es prioridad, lo cual como ya se mencionó, no es una situación realista.

Con respecto a **Búsqueda Binaria**, las conclusiones para alfabetos pequeños se mantienen. Su rendimiento depende directamente de la aridez promedio del árbol, en consecuencia, su uso no se recomienda, dado que **Búsqueda Binaria** es una mejor opción en cualquier contexto comparado a **Búsqueda Binaria**.

Con respecto a la capacidad de compresión y ahorro de espacio, hay rangos en los que ciertas estructuras se comportan mejor que otras. Por lo que para poder concluir cual es mejor, hace falta distinguir estos rangos para el tamaño del alfabeto.

Los mejores desempeños en cuánto a ahorro de espacio, se observan cuando el tamaño del alfabeto varía entre $\sigma = 5,000$ y $\sigma = 1,000,000$. Sin embargo, se observa que al llegar al valor de $\sigma = 1,000,000$, el ahorro de espacio ya es mínimo, por lo que seguir aumentando el tamaño del alfabeto provocará que las estructuras utilicen espacio extra.

Dentro de este rango de tamaños de alfabetos, la estructura de GMR consigue un 27,54 % de ahorro de espacio. Es la estructura de datos compacta que mayor espacio consigue ahorrar en este rango. Además de ser la mejor estructura en cuanto a ahorro de espacio, al comparar su velocidad con las demás estructuras, consigue ser la más rápida siendo solo un 52,91 % más lenta que Búsqueda Binaria. Es **la mejor estructura** para este rango de tamaños de alfabetos, con una amplia ventaja.

Alphabet Partitioning es la estructura que sigue a GMR en ahorro de espacio con un 23,26 % de ahorro en promedio. Claramente es inferior que GMR y esta estructura es además un 268,41 % más lenta que Búsqueda Binaria. Por lo que, es evidente la superioridad de GMR.

Las estructuras que le siguen en ahorro de espacio a Alphabet Partitioning son Wavelet Tree y Huffman Shaped Wavelet Tree, que ahorran un 22,27 % y un 22,02 % respectivamente. Como se ve, estas estructuras se comportan de manera similar. En cuanto a rendimiento son un 296,99 % y un 228,53 % más lentas que Búsqueda Binaria, respectivamente. Al ver los datos en conjunto es sencillo determinar cuál de estas estructuras es más conveniente utilizar para construir árboles cardinales. Lo conveniente es hacerlo utilizando Huffman Shaped Wavelet Tree, pero como ya se vio anteriormente los resultados de GMR son superiores.

Se observa que Wavelet Tree es la estructura más lenta, y los beneficios que pudiera ofrecer ya los poseen las demás estructuras, en consecuencia, su uso no se recomienda.

Cuando se utilizan alfabetos con $\sigma > 1,000,000$ los resultados de las estructuras quedan en entredicho. Se observa que el comportamiento está lejos de ser el esperado o buscado, aunque no ocurre con todas las estructuras.

Wavelet Tree y Huffman Shaped Wavelet Tree comienzan a utilizar espacio extra, un 0,48 % y un 181,03 % extra respectivamente. Esto las deshabilita como opciones para construir un árbol cardinal compacto.

Por otra parte, se tiene que GMR presenta una disminución de su tasa de compresión, haciendo evidente que se ha sobrepasado el punto óptimo en que GMR presenta su mejor desempeño. Cuando el tamaño del alfabeto asciende a los millones de símbolos, GMR ahorra en promedio un 11,90 % de espacio.

La mejor opción es Alphabet Partitioning. En este contexto alcanza un ahorro de espacio de un 25,50 % con respecto a los datos sin comprimir. Es la única estructura que vislumbra un margen de crecimiento, ya que a medida que aumenta el tamaño del alfabeto, es mayor el ahorro de espacio que consigue.

En cuanto a rendimiento, se mantienen los resultados previos, de mejor desempeño a peor desempeño el orden de las estructuras es el siguiente: GMR, Huffman Shaped Wavelet Tree, Alphabet Partitioning, Wavelet Tree.

Cabe agregar también, que el tamaño del árbol, cantidad de nodos y altura no incide mayormente en el rendimiento de las estructuras, más si, la aridez y el tamaño del alfabeto. Sin embargo, todas las estructuras utilizadas tardan más tiempo en resolver *label-child* con árboles más grandes.

En conclusión, no se puede aseverar que dada una construcción de árbol cardinal con ciertas estructuras de datos compactas, sea la mejor opción en cualquier contexto. Sin embargo, se puede construir un árbol cardinal para que se adapte a los datos con los cuales se construirá, y también lograr que su desempeño en cuanto a rendimiento y espacio, sea consecuente con lo que se pretende conseguir. Conocer ciertos atributos del árbol, tales como el tamaño del alfabeto, su aridad promedio, altura, entre otros, es de utilidad para determinar que estructura de datos compacta utilizar.

Capítulo 4

Apéndice

4.1. Uso de Espacio

Tabla 4.1: El término teórico $2n$ en el uso de espacio equivale a 129.182.368 bits, mientras que $n \lg \sigma$ es 307.123.814 bits. Esto da un total teórico de 6,7549 bits por nodo. Por otro lado, $H_0 = 4,27$, por lo que $nH_0 = 276,029,133$. Esto equivale a 6.2735 bits por nodo en teoría.

Proteins	Bits por Nodo teórico	Bits por Nodo Práctico
Estructuras	(No incluye términos $o(\cdot)$ de orden inferior)	(Incluye términos de orden inferior)
LS	6,7549	10,53046
BS	6,7549	10,53046
WT	6,7549	9,64028
GMR	6,7549	8,92853

Proteins	Bits por Nodo teórico	Bits por Nodo Práctico
Estructuras	(No incluye términos $o(\cdot)$ de orden inferior)	(Incluye términos de orden inferior)
WTH	6,2735	15,2100
AP	6,2735	8,6125

Tabla 4.2: El término teórico $2n$ en el uso de espacio equivale a 95.892.334 bits, mientras que $n \lg \sigma$ es 375.253.001 bits. Esto da un total teórico de 9,8265 bits por nodo. Por otro lado, $H_0 = 5,03$, por lo que $nH_0 = 241,305,387$. Esto equivale a 7,0328 bits por nodo en teoría.

English	Bits por Nodo teórico	Bits por Nodo Práctico
Estructuras	(No incluye términos $o(\cdot)$ de orden inferior)	(Incluye términos de orden inferior)
LS	9,8265	10,5357
BS	9,8265	10,5357
WT	9,8265	14,4304
GMR	9,8265	17,0103

English	Bits por Nodo teórico	Bits por Nodo Práctico
Estructuras	(No incluye términos $o(\cdot)$ de orden inferior)	(Incluye términos de orden inferior)
WTH	7,0328	10,0662
AP	7,0328	9,5267

Tabla 4.3: El término teórico $2n$ en el uso de espacio equivale a 60.272.746 bits, mientras que $n \lg \sigma$ es 123.181.305 bits. Esto da un total teórico de 6,0875 bits por nodo. Por otro lado, $H_0 = 1,99$, por lo que $nH_0 = 60,120,256$. Esto equivale a 3,9949 bits por nodo en teoría.

DNA	Bits por Nodo teórico	Bits por Nodo Práctico
Estructuras	(No incluye términos $o(\cdot)$ de orden inferior)	(Incluye términos de orden inferior)
LS	6,0875	10,5324
BS	6,0875	10,5324
WT	6,0875	8,4767
GMR	6,0875	15,2027

DNA	Bits por Nodo teórico	Bits por Nodo Práctico
Estructuras	(No incluye términos $o(\cdot)$ de orden inferior)	(Incluye términos de orden inferior)
WTH	3,9949	5,8404
AP	3,9949	5,4249

Tabla 4.4: El término teórico $2n$ en el uso de espacio equivale a 41.387.962 bits, mientras que $n \lg \sigma$ es 162.483.945 bits. Esto da un total teórico de 9,8517 bits por nodo. Por otro lado, $H_0 = 5,83$, por lo que $nH_0 = 120,576,171$. Esto equivale a 7,8266 bits por nodo en teoría.

Sources	Bits por Nodo teórico	Bits por Nodo Práctico
Estructuras	(No incluye términos $o(\cdot)$ de orden inferior)	(Incluye términos de orden inferior)
LS	9,8517	10,5396
BS	9,8517	10,5396
WT	9,8517	14,4435
GMR	9,8517	17,0151

Sources	Bits por Nodo teórico	Bits por Nodo Práctico
Estructuras	(No incluye términos $o(\cdot)$ de orden inferior)	(Incluye términos de orden inferior)
WTH	7,8266	11,2594
AP	7,8266	10,8414

Tabla 4.5: El término teórico $2n$ en el uso de espacio equivale a 32.410.344 bits, mientras que $n \lg \sigma$ es 107.192.511 bits. Esto da un total teórico de 8,6147 bits por nodo. Por otro lado, $H_0 = 5,72$, por lo que $nH_0 = 92,616,933$. Esto equivale a 7,7153 bits por nodo en teoría.

XML	Bits por Nodo Teórico	Bits por Nodo Práctico
Estructuras	(No incluye términos $o(\cdot)$ de orden inferior)	(Incluye términos de orden inferior)
LS	8,6147	10,5321
BS	8,6147	10,5321
WT	8,6147	12,4353
GMR	8,6147	15,9406

XML	Bits por Nodo Teórico	Bits por Nodo Práctico
Estructuras	(No incluye términos $o(\cdot)$ de orden inferior)	(Incluye términos de orden inferior)
WTH	7,7153	11,0769
AP	7,7153	10,6330

Tabla 4.6: El término teórico $2n$ en el uso de espacio equivale a 17.470.206 bits, mientras que $n \lg \sigma$ es 61.628.619 bits. Esto da un total teórico de 9,0553 bits por nodo. Por otro lado, $H_0 = 5,76$, por lo que $nH_0 = 50,316,989$. Esto equivale a 7,7603 bits por nodo en teoría.

MIDI	Bits por Nodo Teórico	Bits por Nodo Práctico
Estructuras	(No incluye términos $o(\cdot)$ de orden inferior)	(Incluye términos de orden inferior)
LS	9,0553	10,5491
BS	9,0553	10,5491
WT	9,0553	13,1190
GMR	9,0553	17,0576

MIDI	Bits por Nodo Teórico	Bits por Nodo Práctico
Estructuras	(No incluye términos $o(\cdot)$ de orden inferior)	(Incluye términos de orden inferior)
WTH	7,7603	11,1706
AP	7,7603	10,8142

Tabla 4.7: El término teórico $2n$ en el uso de espacio equivale a 718.817.500 bits, mientras que $n \lg \sigma$ es 4.416.311.347 bits. Esto da un total teórico de 14,2877 bits por nodo. Por otro lado, $H_0 = 12,29$, por lo que $nH_0 = 4,415,983,430$. Esto equivale a 14,2868 bits por nodo en teoría.

random5Kv1K	Bits por Nodo Teórico	Bits por Nodo Práctico
Estructuras	(No incluye términos $o(\cdot)$ de orden inferior)	(Incluye términos de orden inferior)
LS	14,2877	34,5369
BS	14,2877	34,5369
WT	14,2877	21,8580
GMR	14,2877	21,1725

random5Kv1K	Bits por Nodo Teórico	Bits por Nodo Práctico
Estructuras	(No incluye términos $o(\cdot)$ de orden inferior)	(Incluye términos de orden inferior)
WTH	14,2868	20,9019
AP	14,2868	20,9398

Tabla 4.8: El término teórico $2n$ en el uso de espacio equivale a 662.825.598 bits, mientras que $n \lg \sigma$ es 5.173.234.641 bits. Esto da un total teórico de 17,6096 bits por nodo. Por otro lado, $H_0 = 15, 60$, por lo que $nH_0 = 5,169,741,393$. Esto equivale a 17,5991 bits por nodo en teoría.

random50Kv1K	Bits por Nodo Teórico	Bits por Nodo Práctico
Estructuras	(No incluye términos $o(\cdot)$ de orden inferior)	(Incluye términos de orden inferior)
LS	17,6096	34,5388
BS	17,6096	34,5388
WT	17,6096	26,3205
GMR	17,6096	24,6457

random50Kv1K	Bits por Nodo Teórico	Bits por Nodo Práctico
Estructuras	(No incluye términos $o(\cdot)$ de orden inferior)	(Incluye términos de orden inferior)
WTH	17,5991	25,9122
AP	17,5991	26,2152

Tabla 4.9: El término teórico $2n$ en el uso de espacio equivale a 908.002.948 bits, mientras que $n \lg \sigma$ es 9.049.121.961 bits. Esto da un total teórico de 21,9319 bits por nodo. Por otro lado, $H_0 = 19, 93$, por lo que $nH_0 = 9,049,021,179$. Esto equivale a 21,9317 bits por nodo en teoría.

random1Mv200K	Bits por Nodo Teórico	Bits por Nodo Práctico
Estructuras	(No incluye términos $o(\cdot)$ de orden inferior)	(Incluye términos de orden inferior)
LS	21,9319	34,5288
BS	21,9319	34,5288
WT	21,9319	32,3474
GMR	21,9319	29,2489

random1Mv200K	Bits por Nodo Teórico	Bits por Nodo Práctico
Estructuras	(No incluye términos $o(\cdot)$ de orden inferior)	(Incluye términos de orden inferior)
WTH	21,9317	33,9678
AP	21,9317	32,3415

Tabla 4.10: El término teórico $2n$ en el uso de espacio equivale a 9.711.312 bits, mientras que $n \lg \sigma$ es 93.902.790 bits. Esto da un total teórico de 21,3388 bits por nodo. Por otro lado, $H_0 = 14, 83$, por lo que $nH_0 = 71,997,239$. Esto equivale a 16,8275 bits por nodo en teoría.

Wikipedia-50MB	Bits por Nodo Teórico	Bits por Nodo Práctico
Estructuras	(No incluye términos $o(\cdot)$ de orden inferior)	(Incluye términos de orden inferior)
LS	21,3388	34,5495
BS	21,3388	34,5495
WT	21,3388	32,2666
GMR	21,3388	28,7013

Wikipedia-50MB	Bits por Nodo Teórico	Bits por Nodo Práctico
Estructuras	(No incluye términos $o(\cdot)$ de orden inferior)	(Incluye términos de orden inferior)
WTH	16,8275	129,5360
AP	16,8275	25,8430

Tabla 4.11: El término teórico $2n$ en el uso de espacio equivale a 79.138.862 bits, mientras que $n \lg \sigma$ es 853.744.469 bits. Esto da un total teórico de 23,5759 bits por nodo. Por otro lado, $H_0 = 15, 25$, por lo que $nH_0 = 603,520,875$. Esto equivale a 17,2522 bits por nodo en teoría.

Wikipedia-500MB	Bits por Nodo Teórico	Bits por Nodo Práctico
Estructuras	(No incluye términos $o(\cdot)$ de orden inferior)	(Incluye términos de orden inferior)
LS	23,5759	34,5371
BS	23,5759	34,5371
WT	23,5759	35,1892
GMR	23,5759	30,9387

Wikipedia-500MB	Bits por Nodo Teórico	Bits por Nodo Práctico
Estructuras	(No incluye términos $o(\cdot)$ de orden inferior)	(Incluye términos de orden inferior)
WTH	17,2522	85,9200
AP	17,2522	25,6496

Tabla 4.12: El término teórico $2n$ en el uso de espacio equivale a 161.728.612 bits, mientras que $n \lg \sigma$ es 1.806.433.882 bits. Esto da un total teórico de 24,3391 bits por nodo. Por otro lado, $H_0 = 15, 34$, por lo que $nH_0 = 1,240,450,368$. Esto equivale a 17,2522 bits por nodo en teoría.

Wikipedia-1GB	Bits por Nodo Teórico	Bits por Nodo Práctico
Estructuras	(No incluye términos $o(\cdot)$ de orden inferior)	(Incluye términos de orden inferior)
LS	24,3391	34,5369
BS	24,3391	34,5369
WT	24,3391	36,6745
GMR	24,3391	31,6471

Wikipedia-1GB	Bits por Nodo Teórico	Bits por Nodo Práctico
Estructuras	(No incluye términos $o(\cdot)$ de orden inferior)	(Incluye términos de orden inferior)
WTH	17,2522	75,7711
AP	17,2522	25,7046

4.2. Tablas De Rendimiento para Operación label-child

Proteins	Espacio [bits]	Tiempo [μ S]
LS	8,53046	0,90297
BS	8,53046	0,89804
WT	7,64028	1,74931
WTH	6,92853	1,54408
GMR	13,21000	1,91398
AP	6,61251	1,87061

Tabla 4.13: Rendimiento árbol Proteins.

English	Espacio [bits]	Tiempo [μ S]
LS	8,53569	0,86752
BS	8,53569	0,80898
WT	12,4304	1,8562
WTH	8,0662	1,75049
GMR	15,0103	1,38904
AP	7,52667	2,00121

Tabla 4.14: Rendimiento árbol English.

DNA	Espacio [bits]	Tiempo [μ S]
LS	8,53237	0,63253
BS	8,53237	0,63615
WT	6,47672	0,87056
WTH	3,84041	0,85311
GMR	13,2027	1,52403
AP	3,42493	1,11641

Tabla 4.15: Rendimiento árbol DNA.

Sources	Espacio [bits]	Tiempo [μ S]
LS	8,53963	0,93147
BS	8,53963	0,85988
WT	12,4435	2,00648
WTH	9,25937	1,69545
GMR	15,0151	1,38336
AP	8,8414	1,98083

Tabla 4.16: Rendimiento árbol Sources.

XML	Espacio [bits]	Tiempo [μS]
LS	8,53207	0,75212
BS	8,53207	0,73219
WT	10,4353	1,46406
WTH	9,07687	1,41344
GMR	13,9406	1,33273
AP	8,6330	1,79440

Tabla 4.17: Rendimiento árbol XML.

MIDI	Espacio [bits]	Tiempo [μS]
LS	8,54914	0,82495
BS	8,54914	0,78469
WT	11,119	1,44305
WTH	9,17059	1,55849
GMR	15,0576	1,31919
AP	8,81421	1,93348

Tabla 4.18: Rendimiento árbol MIDI.

R5Kv1K	Espacio [bits]	Tiempo [μS]
LS	32,53690	2,48927
BS	32,53690	1,41458
WT	19,85800	4,94126
WTH	18,90190	4,03088
GMR	19,17250	2,32992
AP	18,93980	4,63190

Tabla 4.19: Rendimiento árbol Random5Kv1K.

R50Kv1K	Espacio [bits]	Tiempo [μS]
LS	32,53880	2,49112
BS	32,53880	1,38536
WT	24,32050	5,75986
WTH	23,91220	5,04267
GMR	22,64570	2,43087
AP	24,21520	5,77469

Tabla 4.20: Rendimiento árbol Random50Kv1K.

R1Mv200K	Espacio [bits]	Tiempo [μS]
LS	32,5288	241,2730
BS	32,5288	1,76304
WT	30,3474	5,54278
WTH	31,9678	4,90000
GMR	27,2489	2,26143
AP	30,3415	5,38323

Tabla 4.21: Rendimiento árbol Random1Mv200K.

Wiki-50MB	Espacio [bits]	Tiempo [μS]
LS	32,54950	323,3500
BS	32,54950	1,07278
WT	30,26660	3,42887
WTH	127,53600	2,87921
GMR	26,70130	1,42202
AP	23,84300	3,29255

Tabla 4.22: Rendimiento árbol Wiki-50MB.

Wiki-500MB	Espacio [bits]	Tiempo [μS]
LS	32,53710	1.373,3400
BS	32,53710	1,15671
WT	33,18920	5,29084
WTH	83,92000	4,11125
GMR	28,93870	1,82292
AP	23,64960	4,72932

Tabla 4.23: Rendimiento árbol Wiki-500MB.

Wiki-1GB	Espacio [bits]	Tiempo [μS]
LS	32,5369	2.339,6800
BS	32,5369	1,21817
WT	34,6745	6,40096
WTH	73,7711	5,12249
GMR	29,6471	1,93575
AP	23,7046	5,42182

Tabla 4.24: Rendimiento árbol Wiki-1GB.

4.3. Gráficos Tiempo Vs Espacio

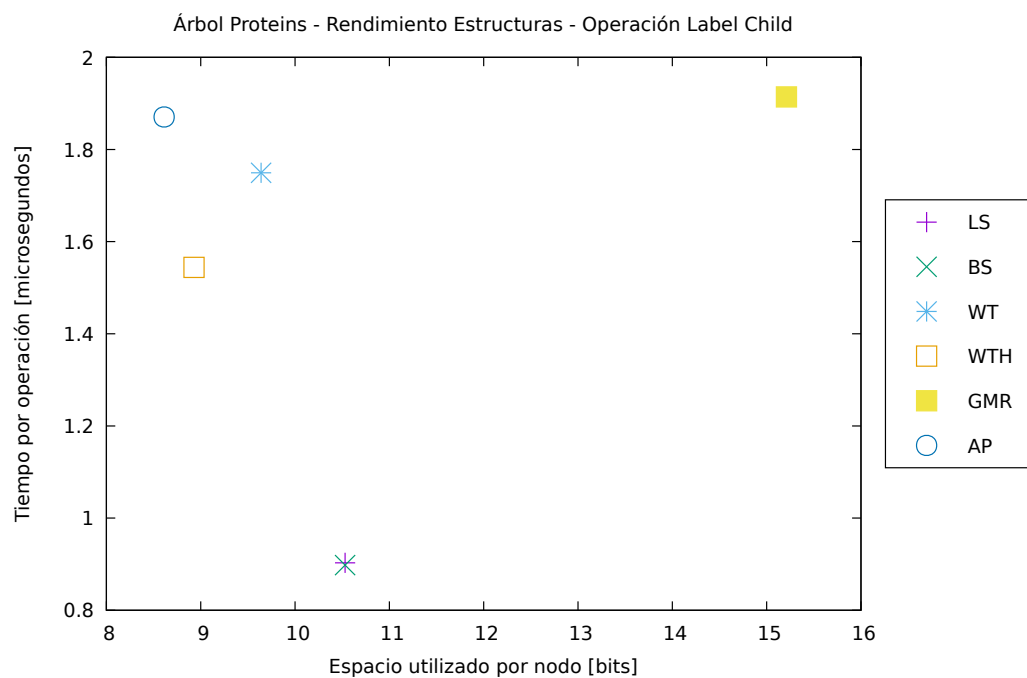


Figura 4.1: Gráfico Tiempo vs Espacio - Árbol Proteins.

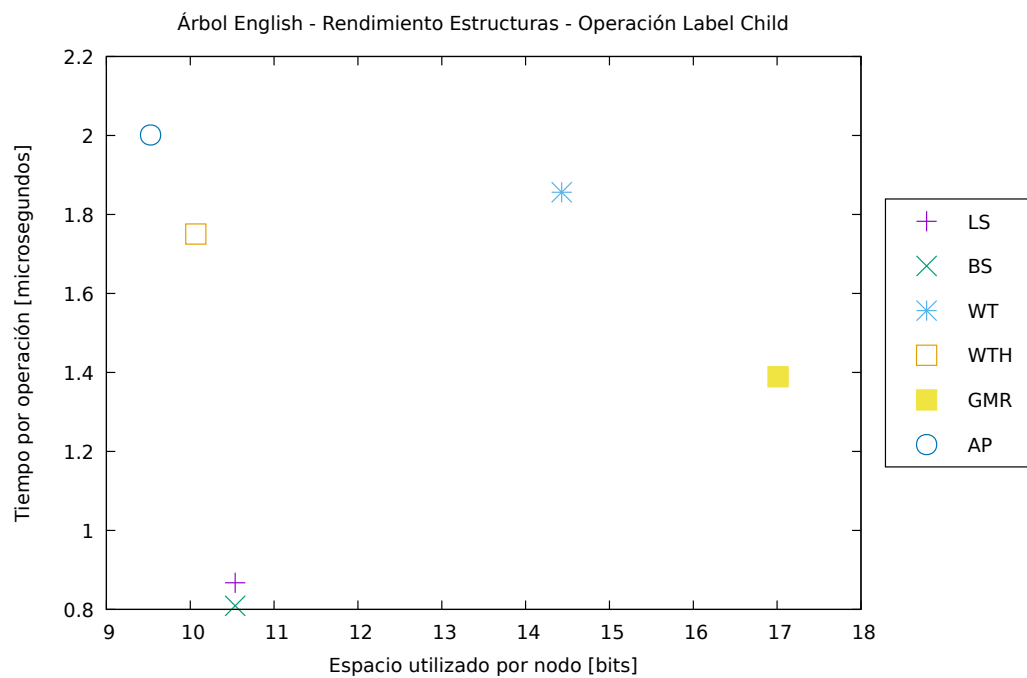


Figura 4.2: Gráfico Tiempo vs Espacio - Árbol English.

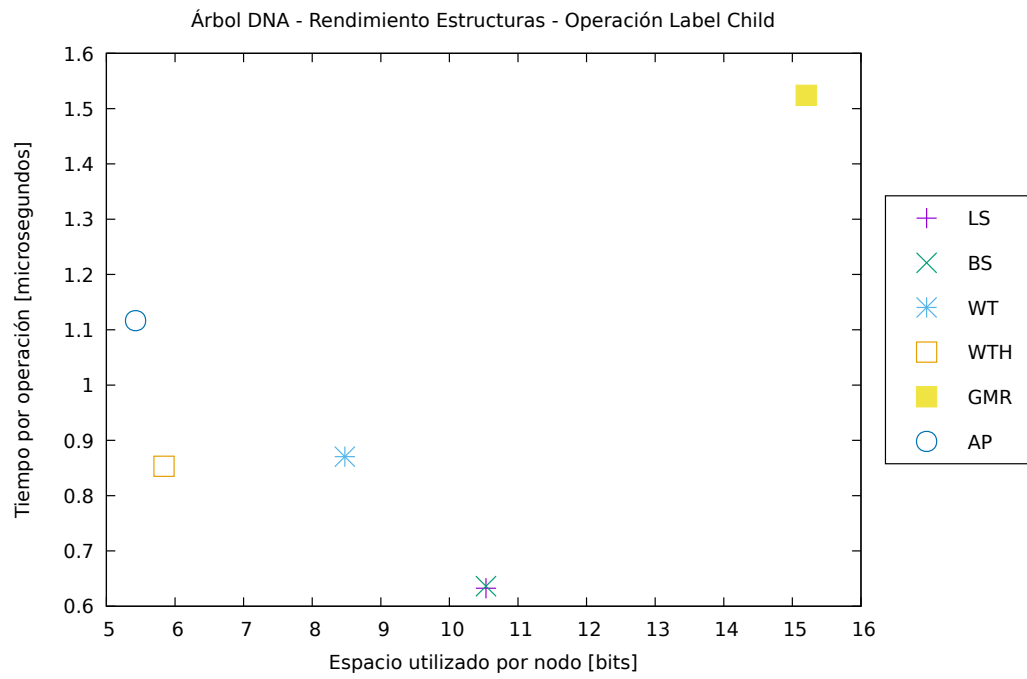


Figura 4.3: Gráfico Tiempo vs Espacio - Árbol DNA.

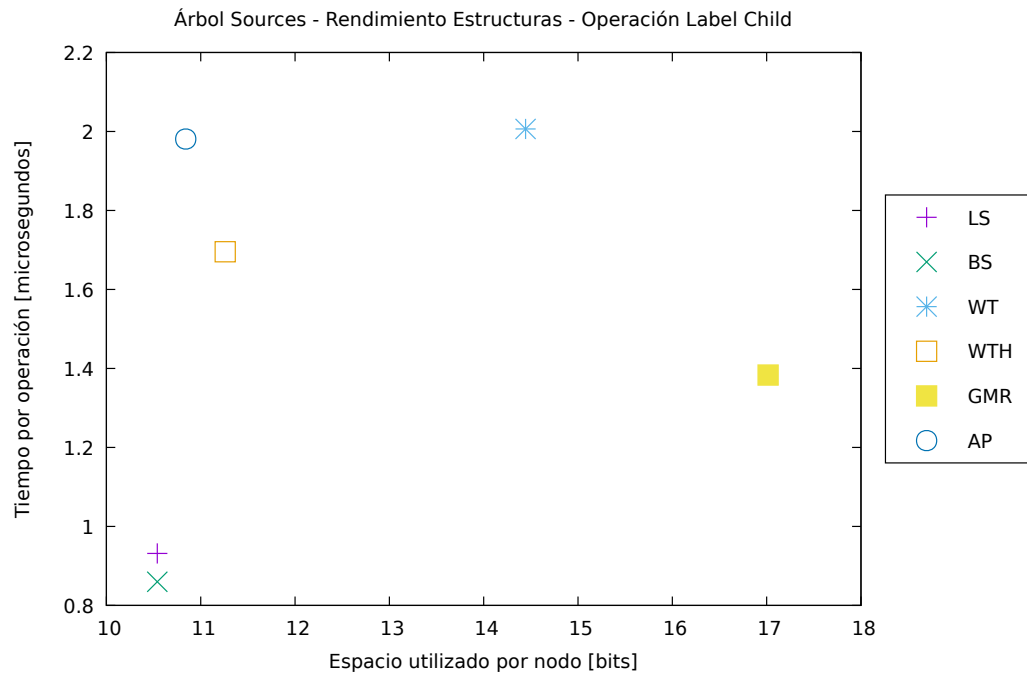


Figura 4.4: Gráfico Tiempo vs Espacio - Árbol Sources.

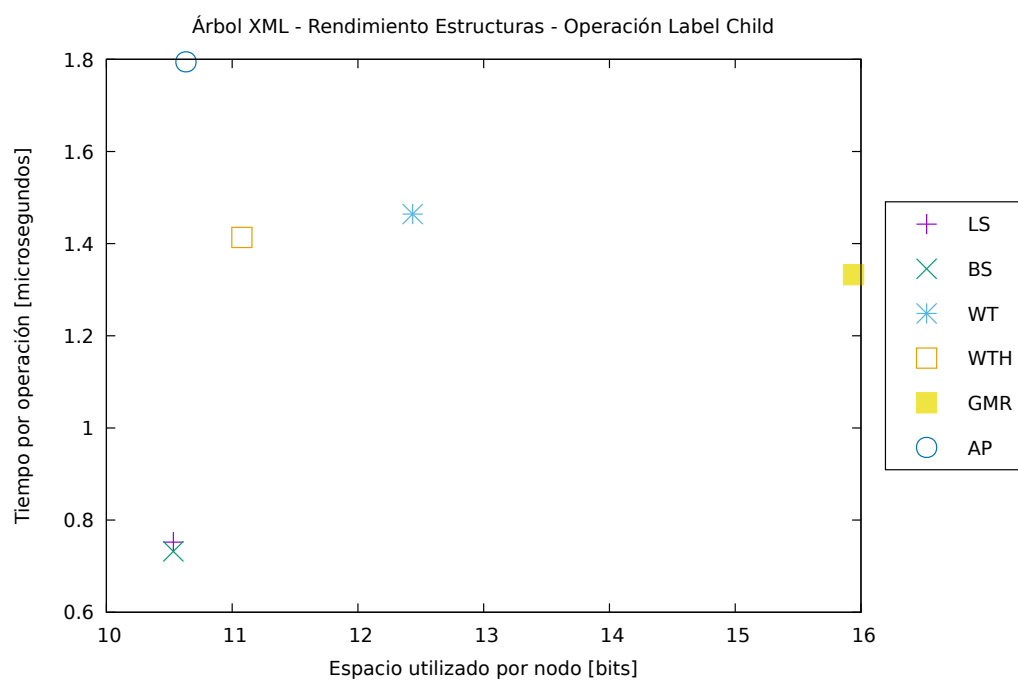


Figura 4.5: Gráfico Tiempo vs Espacio - Árbol XML.

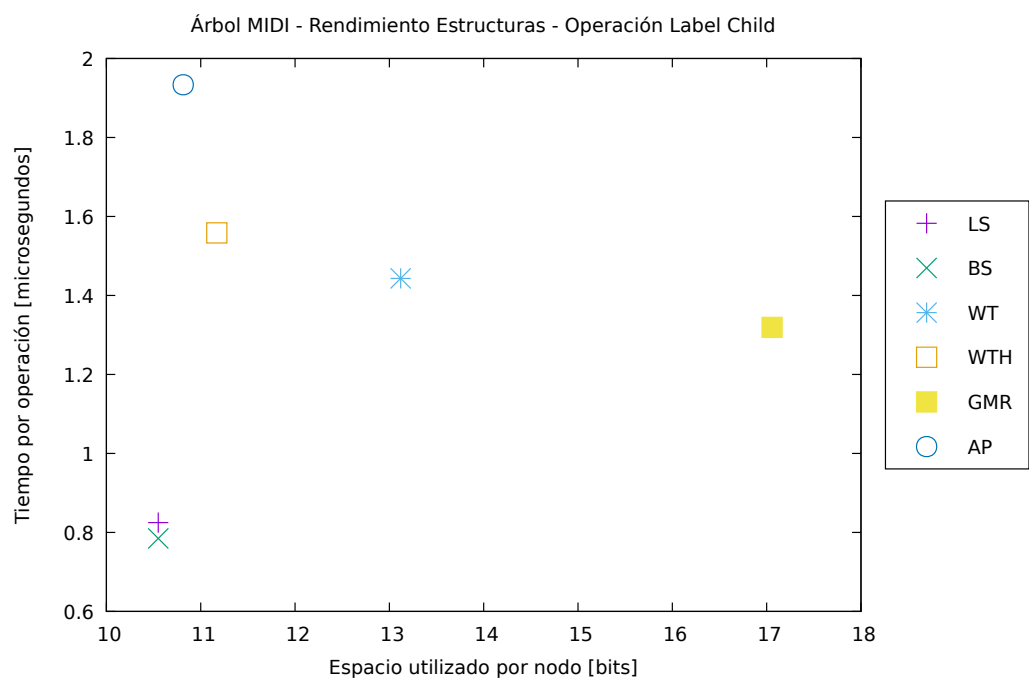


Figura 4.6: Gráfico Tiempo vs Espacio - Árbol MIDI.

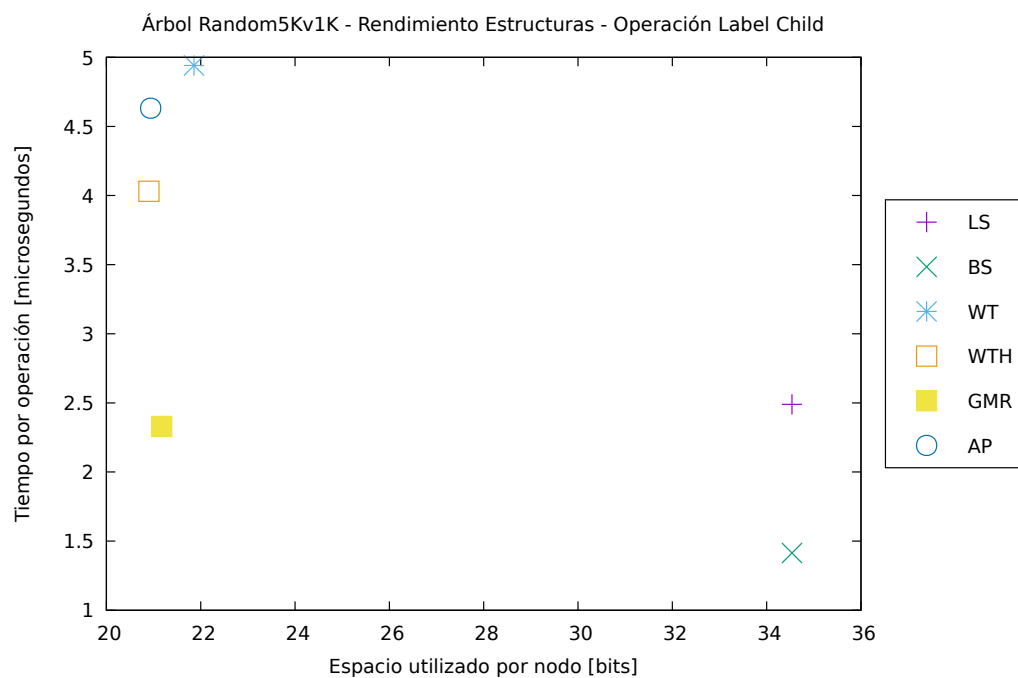


Figura 4.7: Gráfico Tiempo vs Espacio - Árbol Random5Kv1K.

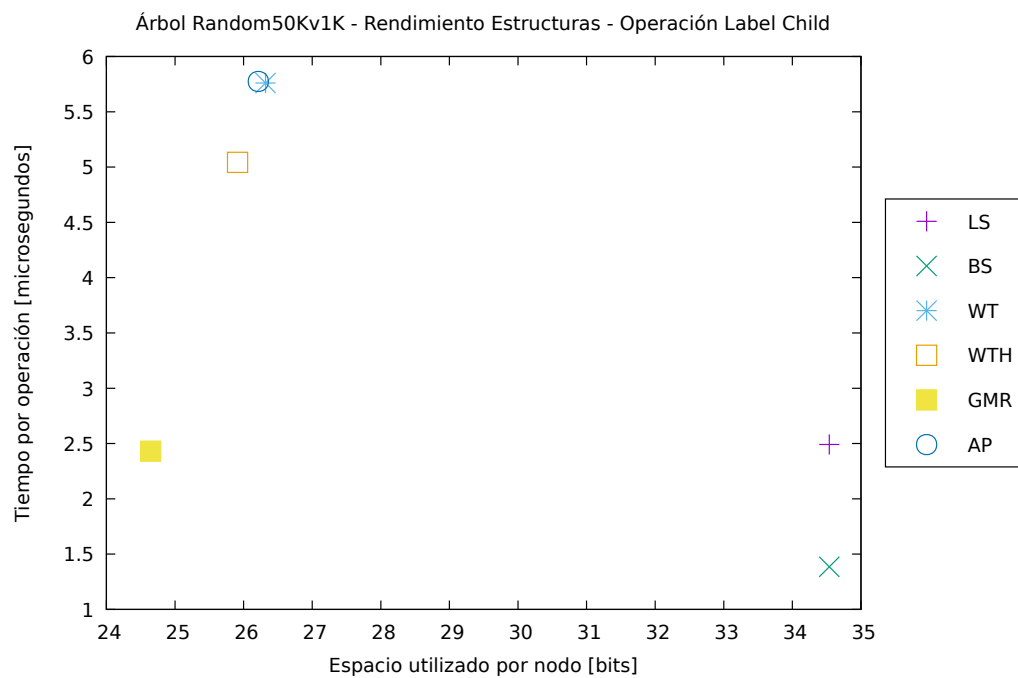


Figura 4.8: Gráfico Tiempo vs Espacio - Árbol Random50Kv1K.

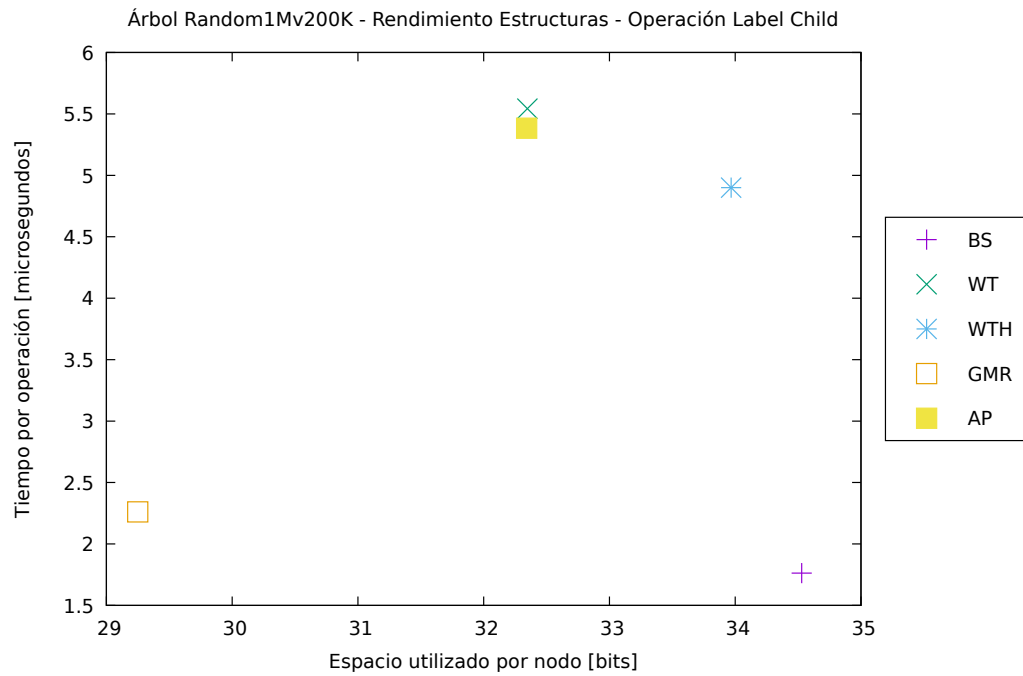


Figura 4.9: Gráfico Tiempo vs Espacio - Árbol Random1Mv200K. El punto referente a LS (34, 5288; 241, 2730), se excluyó debido a que escapa de las dimensiones del gráfico.

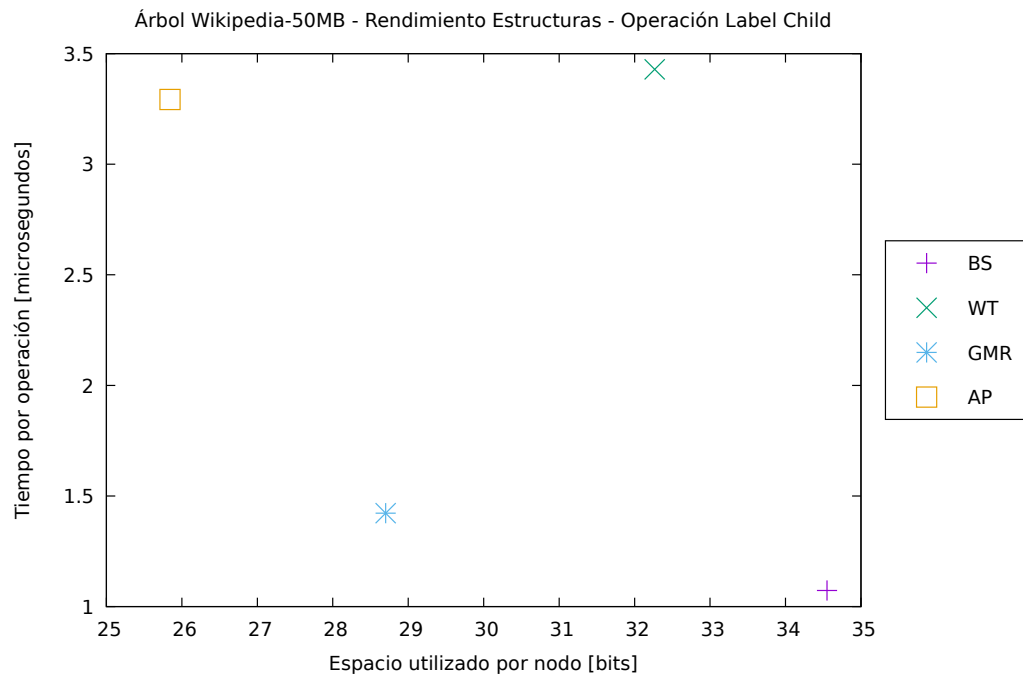


Figura 4.10: Gráfico Tiempo vs Espacio - Árbol Wikipedia-50MB. Los puntos referentes a LS (34, 5495; 323, 3500) y WTH (129, 5360; 2, 8792), se excluyeron debido a que escapan de las dimensiones del gráfico.

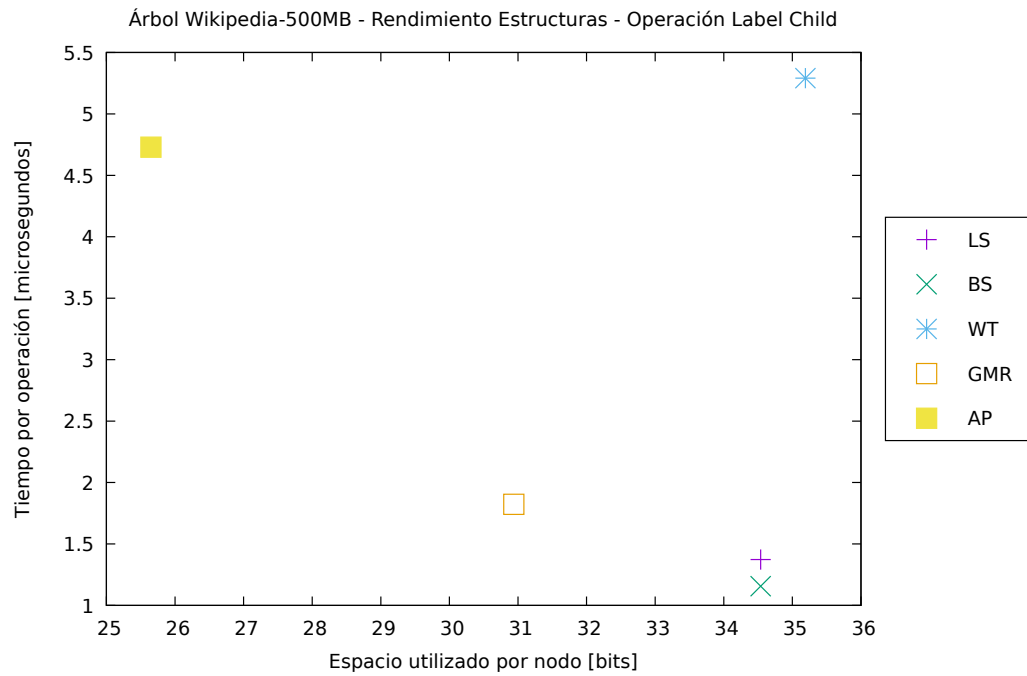


Figura 4.11: Gráfico Tiempo vs Espacio - Árbol Wikipedia-500MB. El punto referente a WTH (85,9200; 4,1112), se excluyó debido a que escapa de las dimensiones del gráfico.

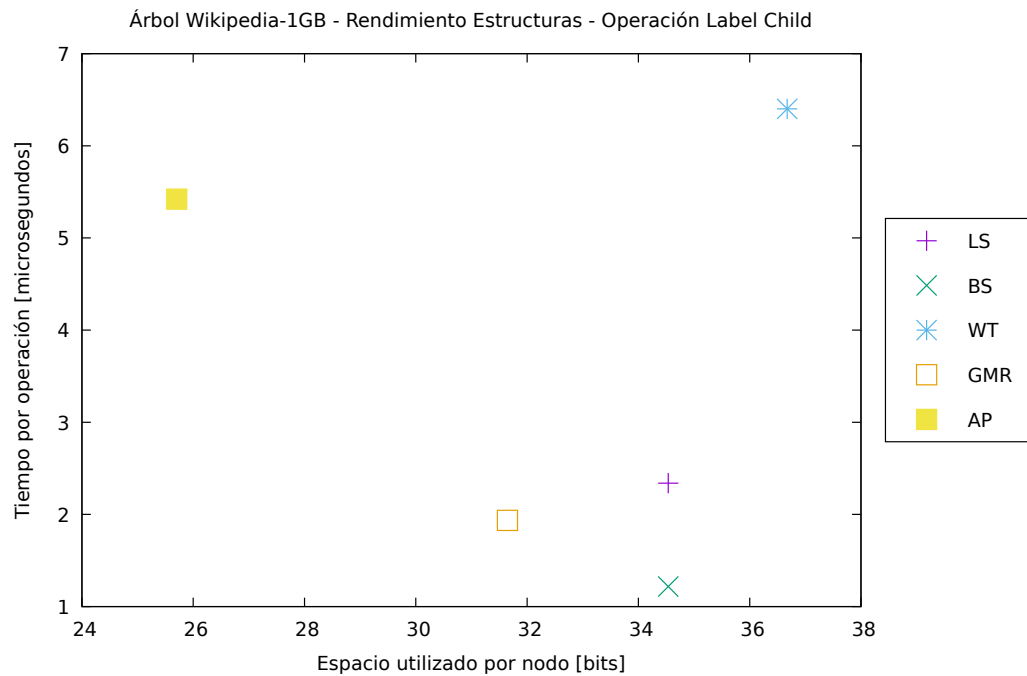


Figura 4.12: Gráfico Tiempo vs Espacio - Árbol Wikipedia-1GB. El punto referente a WTH (75,7711; 5,1224), se excluyó debido a que escapa de las dimensiones del gráfico.

Bibliografía

- [1] IBM. Big data success stories. Technical report, IBM, 2011.
- [2] G. Navarro. Estructuras de datos compactas. Technical report, 2007.
- [3] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: An aid to bibliographic search. *Commun. ACM*, 18(6):333–340, June 1975.
- [4] Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. Succinct indexable dictionaries with applications to encoding k -ary trees, prefix sums and multisets. *ACM Trans. Algorithms*, 3(4), 2007.
- [5] J. Ian Munro. Tables. In *Foundations of Software Technology and Theoretical Computer Science, 16th Conference, Hyderabad, India, December 18-20, 1996, Proceedings*, pages 37–42, 1996.
- [6] Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. Succinct indexable dictionaries with applications to encoding k -ary trees, prefix sums and multisets. *ACM Trans. Algorithms*, 3(4), 2007.
- [7] Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro. Compressed representations of sequences and full-text indexes. *ACM Trans. Algorithms*, 3(2), 2007.
- [8] Giovanni Manzini. An analysis of the burrows-wheeler transform. *J. ACM*, 48(3):407–430, 2001.
- [9] Alexander Golynski, J. Ian Munro, and S. Srinivasa Rao. Rank/select operations on large alphabets: a tool for text indexing. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2006, Miami, Florida, USA, January 22-26, 2006*, pages 368–373, 2006.
- [10] Jérémy Barbay, Meng He, J. Ian Munro, and S. Srinivasa Rao. Succinct indexes for strings, binary relations and multi-labeled trees. In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2007, New Orleans, Louisiana, USA, January 7-9, 2007*, pages 680–689, 2007.
- [11] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete mathematics - a foundation for computer science* (2. ed.). Addison-Wesley, 1994.
- [12] Gonzalo Navarro and Kunihiro Sadakane. Fully functional static and dynamic succinct trees. *ACM Trans. Algorithms*, 10(3):16:1–16:39, 2014.
- [13] Diego Arroyuelo, Rodrigo Cánovas, Gonzalo Navarro, and Kunihiro Sadakane. Succinct trees in practice. In *Proceedings of the Twelfth Workshop on Algorithm Engineering and Experiments, ALENEX 2010, Austin, Texas, USA, January 16, 2010*, pages 84–97, 2010.
- [14] Guy Jacobson. Space-efficient static trees and graphs. In *30th Annual Symposium on Foundations of Computer Science, Research Triangle Park, North Carolina, USA, 30 October - 1 November 1989*, pages 549–554. IEEE Computer Society, 1989.
- [15] David Benoit, Erik D. Demaine, J. Ian Munro, Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005.

- [16] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '03, pages 841–850, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics.
- [17] Veli Mäkinen and Gonzalo Navarro. Succinct suffix arrays based on run-length encoding. *Nord. J. Comput.*, 12(1):40–66, 2005.
- [18] Jérémy Barbay, Travis Gagie, Gonzalo Navarro, and Yakov Nekrich. Alphabet partitioning for compressed rank/select and applications. In Otfried Cheong, Kyung-Yong Chwa, and Kunsoo Park, editors, *Algorithms and Computation - 21st International Symposium, ISAAC 2010, Jeju Island, Korea, December 15-17, 2010, Proceedings, Part II*, volume 6507 of *Lecture Notes in Computer Science*, pages 315–326. Springer, 2010.
- [19] Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practice: Plug and play with succinct data structures. In *13th International Symposium on Experimental Algorithms, (SEA 2014)*, pages 326–337, 2014.
- [20] Paolo Ferragina and Gonzalo Navarro. *Pizza&chili corpus*, 2005.