

UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE INFORMÁTICA
SANTIAGO - CHILE



“ESTRUCTURA DE DATOS COMPRIMIDAS PARA BÚSQUEDAS EN TEXTOS MUY REPETITIVOS Y ALFABETOS GRANDES”

FRANCISCO EDUARDO RIVEROS CORVALÁN

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN INFORMÁTICA

Profesor Guía: DIEGO ARROYUELO
Profesor Correferente: JOSÉ LUIS MARTI

Octubre - 2022

DEDICATORIA

Dedicado a mi familia por su apoyo incondicional durante todo el proceso de este trabajo.

RESUMEN

El manejo de textos o secuencias hoy en día es fundamental en diversas áreas. Estos pueden variar en sus características, ya sea en extensión, repetitividad, tamaño de su alfabeto, etc. En relación con lo anterior, se propone una estructura de datos que permita trabajar con textos que posean alfabetos grandes y sean repetitivos de manera eficiente. La estructura propuesta utiliza técnicas ya conocidas como la BWT y ASAP, esta última es intervenida para que sea eficiente trabajando secuencias de runs. La estructura es comparada con otras representaciones ya conocidas que permiten trabajar textos con los atributos propuestos. Los resultados de los experimentos realizados indican que la solución logra disminuir los tiempos de búsqueda a cambio de utilizar un leve cantidad extra de memoria.

Palabras Clave— Rank; Select; ASAP; Bit vector; Wavelet Tree.

ABSTRACT

The management of texts or sequences today is essential in various areas. These can vary in their characteristics, whether in length, repeatability, size of their alphabet, etc. In relation to the above, a data structure is proposed that allows working with texts that have large alphabets and are repetitive efficiently. The proposed structure uses already known techniques such as BWT and ASAP, the latter is intervened to be efficient working sequences of runs. The structure is compared with other already known representations that allow texts to be worked with the proposed attributes. The results of the experiments carried out indicate that the solution manages to reduce search times in exchange for using a slight extra amount of memory.

Keywords— Rank; Select; ASAP; Bit vector; Wavelet Tree.

ÍNDICE DE CONTENIDOS

RESUMEN	III
ABSTRACT	III
ÍNDICE DE FIGURAS	VI
ÍNDICE DE TABLAS	VII
CAPÍTULO 1: DEFINICIÓN DEL PROBLEMA	1
1.1 INTRODUCCIÓN	1
1.2 CONTEXTO Y SITUACIÓN ACTUAL	1
1.3 OBJETIVOS	2
1.3.1 OBJETIVO GENERAL	2
1.3.2 OBJETIVOS ESPECÍFICOS	2
CAPÍTULO 2: MARCO CONCEPTUAL	3
2.1 OPERACIONES DE BÚSQUEDA RANK/SELECT/ACCESS	3
2.1.1 RANK	3
2.1.2 SELECT	3
2.1.3 ACCESS	4
2.2 ENTROPÍA H_0	4
2.3 TRANSFORMACIÓN DE BURROWS-WHEELER (BWT)	4
2.4 WAVELET TREES (WT)	6
2.4.1 WT - RANK	6
2.4.2 WT - SELECT	7
2.4.3 WT - ACCESS	8
2.5 ALGORITMO FM-COUNT	9
2.6 ALPHABET PARTITIONING (AP)	13
2.6.1 AP - RANK	15
2.6.2 AP - SELECT	15
2.6.3 AP - ACCESS	15
2.7 A PRACTICAL ALPHABET-PARTITIONING RANK/SELECT DATA STRUCTURE (ASAP)	16
2.7.1 ASAP - RANK	17
2.7.2 ASAP - SELECT	17
2.7.3 ASAP - ACCESS	17
2.8 RUN LENGHT ENCONDING	18
2.9 ELIAS-FANO	18
2.10 PARTITIONED ELIAS-FANO	19
2.11 S18 VECTOR	20
2.12 RLRUNS	20

CAPÍTULO 3: PROPUESTA DE SOLUCIÓN	22
3.1 PREPARACIÓN Y PROCESAMIENTO DE TEXTOS	22
3.2 CREACIÓN DE AR-RLFM	23
3.3 AR-RLFM - OPERACIÓN COUNT	24
3.4 AR-RLFM - IMPLEMENTACIONES	24
CAPÍTULO 4: VALIDACIÓN DE LA SOLUCIÓN	26
4.1 ESTRUCTURAS A COMPARAR CON AR-RLFM	26
4.2 ENTORNO DE EJECUCIÓN Y COMPILACIÓN	26
4.3 S18 VECTOR Y $\vec{b}$	26
4.4 EXPERIMENTOS	27
4.4.1 ALBERT EINSTEIN INGLÉS	28
4.4.2 ALBERT EINSTEIN ALEMÁN	30
4.4.3 COREUTILS	33
4.4.4 WORLD LEADERS	35
4.4.5 ENGLISH	38
CAPÍTULO 5: CONCLUSIONES	41
ANEXOS	43
REFERENCIAS BIBLIOGRÁFICAS	53

ÍNDICE DE FIGURAS

1	Ejemplo operación <i>Rank</i> en string s	3
2	Ejemplo operación <i>Select</i> en string s	3
3	Ejemplo operación <i>Access</i> en string s	4
4	Ejemplo aplicación BWT en string T	5
5	Wavelet Tree de la palabra <i>abracadabra</i>	6
6	Ejemplo operación <i>Rank</i> aplicado en Wavelet Tree	7
7	Ejemplo operación <i>Select</i> aplicado en Wavelet Tree	8
8	Ejemplo operación <i>Access</i> aplicado en Wavelet Tree	9
9	BWT en string $T = mississippi$	10
10	Ejemplo Count patrón " <i>iss</i> " en string T - Paso 1	11
11	Ejemplo Count patrón " <i>iss</i> " en string T - Paso 2	12
12	Ejemplo Count patrón " <i>iss</i> " en string T - Paso 3	13
13	Resultados de experimento con AR-RLFM utilizando S18 vector para encontrar parámetro b ideal, utilizando el texto Einstein versión alemán con patrones de largo 16.	27
14	Resultados texto Einstein versión inglés $m = 4$	28
15	Resultados texto Einstein versión inglés $m = 8$	29
16	Resultados texto Einstein versión inglés $m = 16$	29
17	Resultados texto Einstein versión alemán $m = 4$	31
18	Resultados texto Einstein versión alemán $m = 8$	31
19	Resultados texto Einstein versión alemán $m = 16$	32
20	Resultados texto Coreutils $m = 4$	33
21	Resultados texto Coreutils $m = 8$	34

22 Resultados texto Coreutils m = 16	34
23 Resultados texto World Leaders m = 4	36
24 Resultados texto World Leaders m = 8	36
25 Resultados texto World Leaders m = 16	37
26 Resultados texto English.001 m = 4	38
27 Resultados texto English.001 m = 8	39
28 Resultados texto English.001 m = 16	39

ÍNDICE DE TABLAS

1 Ejemplo Elias-Fano, $\lceil \log u \rceil - l$ bits más significativos y l bits menos significativos para cada elemento de S	18
2 Ejemplo Elias-Fano, Cardinalidades de cada grupo de $\lceil \log u \rceil - l$ bits.	19
3 Versiones de AR-RLFM generadas.	25
4 Versiones Bases con las cuales sera comparado AR-RLFM.	26
5 Albert Einstein versión ingles Información	28
6 Albert Einstein versión alemán Información	30
7 Coreutils información.	33
8 World Leaders Información	35
9 English Información	38
10 Memoria utilizada por cada una de las estructuras en el texto einstein.en.txt	43
11 Tiempos promedios obtenidos por cada una de las estructuras para el texto einstein.en.txt	44
12 Memoria utilizada por cada una de las estructuras en el texto einstein.de.txt	45
13 Tiempos promedios obtenidos por cada una de las estructuras para el texto einstein.de.txt	46
14 Memoria utilizada por cada una de las estructuras en el texto coreutils	47

15	Tiempos promedios obtenidos por cada una de las estructuras para el texto coreutils	48
16	Memoria utilizada por cada una de las estructuras en el texto english.001.2 . .	49
17	Tiempos promedios obtenidos por cada una de las estructuras para el texto english.001.2	50
18	Memoria utilizada por cada una de las estructuras en el texto world_leaders . .	51
19	Tiempos promedios obtenidos por cada una de las estructuras para el texto world_leaders	52

CAPÍTULO 1

DEFINICIÓN DEL PROBLEMA

1.1. INTRODUCCIÓN

En la actualidad, existen diversas áreas en las que se requiere del manejo de altos volúmenes de datos. La memoria a utilizar juega un rol fundamental, considerando además el tiempo que demora en realizar tareas o operaciones a dichos datos para obtener información de ellos. Es por esto que surgen diversas estructuras de datos que buscan optimizar los recursos o procesos. Este trabajo se enfoca específicamente en la construcción de una estructura de datos que permita la manipulación de textos repetitivos y que posean alfabetos grandes. Además de permitir el uso de la operación *Count* que permitirá obtener la cantidad de ocurrencias de alguna frase o palabra dentro del texto. Se utilizará como base la estructura ASAP [Arroyuelo y Sepúlveda, 2019], la cual será intervenida para lograr mejores resultados en el contexto propuesto.

El documento comienza con el capítulo 1 en el que se introducirá a detalle el contexto de este trabajo. En el capítulo 2 se presentan conceptos o estructuras que serán necesarias para la construcción de la estructura a desarrollar. El capítulo 3 contiene la propuesta de solución al problema, y se especifica la implementación de la estructura junto con las diversas versiones que serán puestas a prueba. En este se define la operación *Count* a implementar. En el capítulo 4 se presentan las estructuras utilizadas como base para realizar la comparación de rendimiento con la estructura a construir, y se explica el proceso experimental a realizar junto con sus resultados. Además, se detallan los criterios para la medición del uso de memoria y tiempo utilizado en realizar las operaciones de búsqueda. Por último, el capítulo 5 contiene las conclusiones en base a los resultados obtenidos y se discute sobre el trabajo futuro.

1.2. CONTEXTO Y SITUACIÓN ACTUAL

La problemática por abordar consiste en crear una estructura de datos que permita comprimir textos grandes y muy repetitivos. Se estudiarán aquellos textos que posean un alfabeto considerable que soporte las operaciones *Rank*, *Select* y *Access* para realizar búsquedas. Estos escritos pueden requerir una cantidad enorme de memoria, además de que al realizar búsquedas, de alguna frase o carácter dentro de un texto muy largo, puede demorar. A su vez se pueden presentar varias ocurrencias dentro del texto.

Actualmente, existen una serie de textos que reflejan la problemática mencionada. Estos se pueden encontrar en repositorios de códigos que permiten el control de versiones de programas, aplicaciones u otros [Stopak, 2019]. Un ejemplo de esto, en el presente, es

el uso de Github que almacena diversos proyectos. Generalmente, en los repositorios se observa que las distintas versiones almacenadas contienen una gran cantidad de fragmentos sin alterar, lo que genera una base de datos con textos repetidos. Estos últimos, al ser concatenados producen un texto grande y repetitivo tal como podría ser el caso de la concatenación de las diversas versiones de Wikipedia, la cual permite realizar comparaciones entre distintas versiones.

Dada las problemáticas anteriores es necesaria la creación de una estructura de datos eficiente en el uso de memoria y el tiempo que requieren las operaciones elementales de búsqueda. Se define $T[1..n]$ como el texto que es preprocesado para la construcción de su estructura de datos. Con dicha estructura es posible realizar búsquedas dentro del texto, tales como el rastreo de las posiciones o las ocurrencias de algún patrón $P[1..m]$. La investigación actual se enfoca principalmente de esta última.

Por otro lado, la búsqueda en textos ha generado diversas técnicas. Por ejemplo, entre las soluciones clásicas se encuentra el uso de arreglos de sufijos. No obstante, esta técnica generalmente requiere de una gran cantidad de memoria. Es por esto que surge la propuesta de estructuras comprimidas, entre ellas están los arreglos de sufijos comprimidos. A su vez, surgen los FM-indexes basados en la transformación de Burrows-Wheeler del texto.

En relación a esto, se propone la implementación de una estructura de datos FM-index, en la que el objetivo es sacar provecho de la repetitividad de los textos con alfabetos grandes, para lograr que los tiempos de búsqueda sean más eficientes que las alternativas que existen en la actualidad.

1.3. OBJETIVOS

1.3.1. OBJETIVO GENERAL

- Implementar y evaluar una estructura de datos comprimida para búsqueda en textos muy repetitivos y con alfabetos grandes.

1.3.2. OBJETIVOS ESPECÍFICOS

- Modificar la estructura de datos ASAP [Arroyuelo y Sepúlveda, 2019] para permitir la compresión de *runs* de símbolos iguales en las subsecuencias de las particiones y runs de 0s y 1s en los vectores de bits de dichas particiones generadas por la estructura.
- Implementar el algoritmo de conteo de patrones de Ferragina y Manzini sobre la estructura del ítem anterior.
- Evaluar experimentalmente el prototipo.

CAPÍTULO 2

MARCO CONCEPTUAL

Para lograr llevar a cabo la estructura de datos se estudia a detalle los siguientes conceptos, teniendo en consideración la definición de un *string* $s[1..n]$ que posee un alfabeto $\Sigma = \{0, \dots, \sigma - 1\}$ y un símbolo $c \in \Sigma$ de s .

2.1. OPERACIONES DE BÚSQUEDA RANK/SELECT/ACCESS

Se definen las siguientes operaciones elementales de búsqueda que son aplicables en un string, sea de ejemplo $s = \text{estructura}$.

2.1.1. RANK

La operación $s.\text{Rank}_c(i)$ computa la cantidad de ocurrencias de un símbolo c dentro de un string s hasta la posición i . Por ejemplo, en la Figura 1 si se quiere obtener la cantidad de ocurrencias de la letra u en el string s hasta la posición 8 se obtiene $s.\text{Rank}_u(8) = 2$.

e	s	t	r	u	c	t	u	r	a
0	1	2	3	4	5	6	7	8	9

Figura 1: Ejemplo operación *Rank* en string s .

2.1.2. SELECT

La operación $s.\text{Select}_c(i)$ computa la posición que almacena la i -ésima ocurrencia del símbolo c en s . Por ejemplo, en la Figura 2 si se quiere obtener la posición de la segunda ocurrencia de la letra u en el string s se obtiene $s.\text{Select}_u(2) = 7$.

e	s	t	r	u	c	t	u	r	a
0	1	2	3	4	5	6	7	8	9

Figura 2: Ejemplo operación *Select* en string s .

2.1.3. ACCESS

La operación $s.Access(i)$ entrega el símbolo c de la posición i del string s . Por ejemplo, en la Figura 3 si se quiere obtener el símbolo de la posición 5 en el string s se obtiene $s.Access(5) = c$.

e	s	t	r	u	c	t	u	r	a
0	1	2	3	4	5	6	7	8	9

Figura 3: Ejemplo operación *Access* en string s .

2.2. ENTROPÍA H_0

Se define como H_0 a la entropía de orden cero, cuantifica qué tanto se puede comprimir el string s , dado que cada símbolo $c \in \Sigma$ ocurre n_c en un string de largo n , y se calcula de la siguiente manera:

$$H_0 = \sum_{c \in \Sigma} \frac{n_c}{n} \lg \frac{n}{n_c}$$

Se asume $n_c > 0$ dado que la suma solo considera los símbolos con ocurrencias en s . Esta medida conocida como entropía de compresión indica cuantos bits se deben utilizar en promedio por carácter c cuando se codifique el string s , basado en las frecuencias de cada carácter del alfabeto.

Mientras más sesgada la distribución de las frecuencias de c , el valor de H_0 se minimizará. De forma análoga, entre menos sesgada la distribución de frecuencias, H_0 tomará un valor más grande. Por ejemplo, en el caso de que todos los caracteres posean la misma cantidad de frecuencias, H_0 tomará su valor máximo $\lg \sigma$. Cabe destacar que entre más pequeño el valor de H_0 , más se logra comprimir el string s .

2.3. TRANSFORMACIÓN DE BURROWS-WHEELER (BWT)

En 1994 Michael Burrows y David Wheeler desarrollan la transformación "Burrows-Wheeler", la cual permite realizar permutaciones entre los caracteres de un texto. De esta manera, si el texto posee subcadenas con varias ocurrencias, el resultado final tendrá posiciones donde estas subcadenas o caracteres se repiten múltiples veces, pero utilizando el mismo espacio del texto original. Por lo que entre más repeticiones posea el texto la BWT nos permite comprimir de manera más eficiente [Burrows y Wheeler, 1994].

Sea $T = \text{mississippi\$}$, utilizando $\$$ con el objetivo de denotar el termino del string,

para obtener $BWT(T)$ primero se lista los n shift cíclicos de s y luego se aplica un sort con la intención de realizar un orden lexicográfico de las filas, para esto, se considera el carácter especial \$ como inicial.

		<i>SA</i>	<i>F</i>	<i>L</i>	<i>L</i>
0	mississippi\$				
1	ississippi\$m	11	\$mississippi		<i>i</i>
2	ssissippi\$mi	10	i\$mississipp		<i>p</i>
3	sissippi\$mis	7	ippi\$mississ		<i>s</i>
4	issippi\$miss	4	issippi\$miss		<i>s</i>
5	sippi\$missi	1	ississippi\$m		<i>m</i>
6	sippi\$missis	0	mississippi\$		\$
7	ippi\$mississ	9	pi\$mississip		<i>p</i>
8	ppi\$mississi	8	ppi\$mississi		<i>i</i>
9	pi\$mississip	6	sippi\$missis		<i>s</i>
10	i\$mississipp	3	sissippi\$mis		<i>s</i>
11	\$mississippi	5	ssippi\$missi		<i>i</i>
		2	ssissippi\$mi		<i>i</i>

Figura 4: Ejemplo aplicación BWT en string T

La $BWT(T)$ corresponde a la última columna de esta listas ordenadas denotada como L , quedando:

$$L = BWT(T) = ipssm$piissii$$

En la Figura 4 se observa que la primera columna denotada por F posee un orden lexicográfico lo que genera intervalos de caracteres repetidos. Por otro lado, Suffix-Array (SA), a diferencia de F , almacena la posición de los caracteres resultantes luego del orden lexicográfico. Al observar a F es posible apreciar las primeras ocurrencias de cada carácter. Esta información es almacenada en lo que se denomina arreglo C . Cada uno de los valores del arreglo mencionado corresponde a un índice en el que la columna F cambia al siguiente símbolo en el orden lexicográfico, para el ejemplo propuesto se tiene que $C = [0, 1, 5, 6, 8, 12]$. Por último, en L se encuentran todos aquellos predecesores relacionados a cada carácter de F , es por esto que tienden a generarse repeticiones en la columna L .

Además, se tiene el arreglo de sufijos $SA = [11, 10, 7, 4, 1, 0, 9, 8, 6, 3, 5, 2]$, que posee la siguiente relación con T y L :

$$L[i] = T[SA[i] - 1] \quad (1)$$

Utilizando esta relación podemos obtener la $BWT(T)$ en tiempo $O(n)$, dado que basta con recorrer el arreglo SA , y aplicar la relación para obtener cada valor de L o T en caso de estar realizando el proceso inverso.

2.4. WAVELET TREES (WT)

Publicado en 2003 por Grossi, Gupta y Vitter , corresponde a una estructura fundamental tipo árbol capaz de comprimir textos. El árbol va dividiendo el alfabeto recursivamente en cada nivel de éste, hasta llegar a las hojas que corresponde a un símbolo individual. El espacio utilizado por un texto s en esta estructura es $n \lg \sigma$ bits. Por otra parte, permite aplicar funciones *Rank*, *Select* y *Access* en tiempo $O(\lg \sigma)$, sin embargo, esta estructura se vuelve más ineficiente al representar textos con alfabetos grandes [Navarro, 2014].

Sea $s = abracadabra$, para obtener el Wavelet Tree respectivo, el árbol irá dividiendo el alfabeto Σ de tal manera que, en cada nivel del árbol, aquellos caracteres que son repartidos hacia el hijo izquierdo serán denotados por un 0 y los caracteres que serán repartidos hacia el hijo derecho con un 1, por lo que nos queda un árbol como se observa en la Figura 5.

Cada hoja es un símbolo dado que divide el alfabeto original hasta llegar a un carácter único, obteniendo σ hojas y altura $\lg \sigma$. Por otra parte, en cada nivel del árbol se utiliza una cantidad igual o inferior a los n bits del texto original, es por esto que la memoria utilizada por los Wavelet Tree corresponde a $n \lg \sigma$ bits.

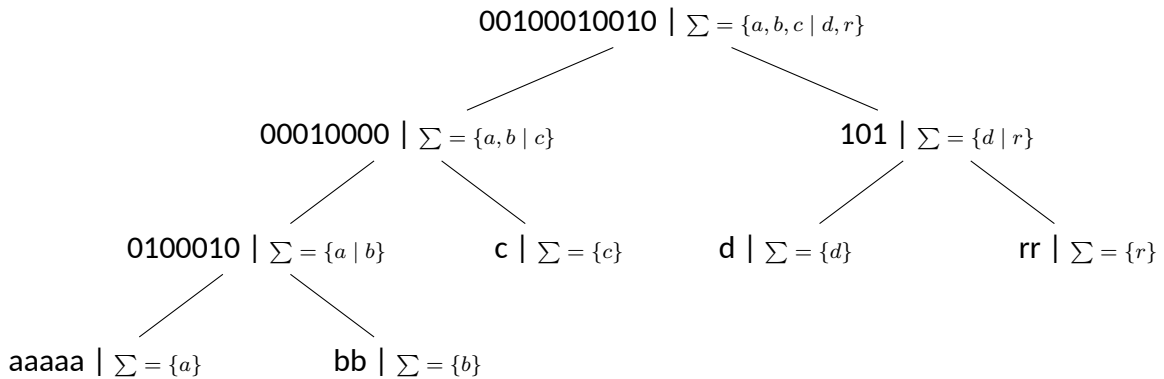


Figura 5: Wavelet Tree de la palabra *abracadabra*

2.4.1. WT - RANK

Para implementar la operación de búsqueda elemental *Rank* usando esta estructura, se comienza desde la raíz y se baja siguiendo el camino hasta llegar a la hoja del carácter que

se está buscando, es por esto que el tiempo de esta operación de búsqueda corresponde a $O(\lg \sigma)$.

Por ejemplo, en la Figura 6 si se quiere obtener $s.Rank_a(7)$ se consideran los primeros 7 caracteres, dado que a , en el nodo raíz corresponde a un 0, se debe contar la cantidad de 0's y continuar al siguiente nodo para obtener la nueva posición de término. En este caso la nueva posición de término del nodo hijo corresponde a 5, debido a que el nodo raíz posee 5 ceros hasta la posición 7. Este proceso se repite hasta llegar al problema trivial de las hojas, el que tiene un único símbolo y se puede aplicar la operación $Rank$ obteniendo el carácter directamente.

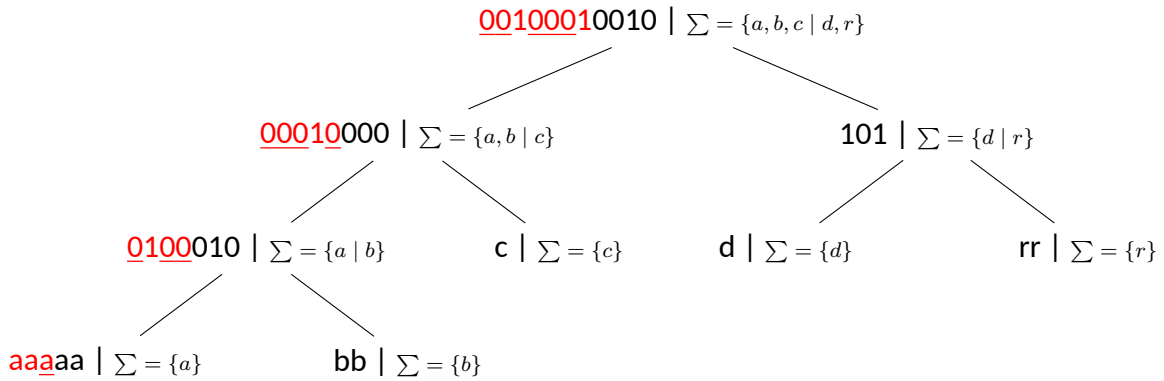


Figura 6: Ejemplo operación $Rank$ aplicado en Wavelet Tree

Como se observa, al repetir el proceso se llega al nodo hijo respectivo del carácter a en el que se calcula la cantidad de ocurrencias directamente, obteniendo $s.Rank_a(7) = 3$.

2.4.2. WT - SELECT

Para aplicar la operación de búsqueda elemental $Select$ a esta estructura se debe posicionar en la hoja del carácter que se busca y comenzar a subir niveles hasta llegar al nodo raíz; cada vez que se sube a un padre se actualiza la posición según corresponda.

Por ejemplo, como se observa en la Figura 7, si se quiere obtener $s.Select_a(4)$, el nodo hoja nos indica una posición inicial 4. Al subir al primer padre, dado que a corresponde a un 0 se debe actualizar la posición, según la cantidad de caracteres de la otra mitad del alfabeto denotados por 1, en este caso la posición se actualiza a 5 dado que se añade una b entre la cadena de a 's. Se repite el proceso hasta llegar al nodo raíz.

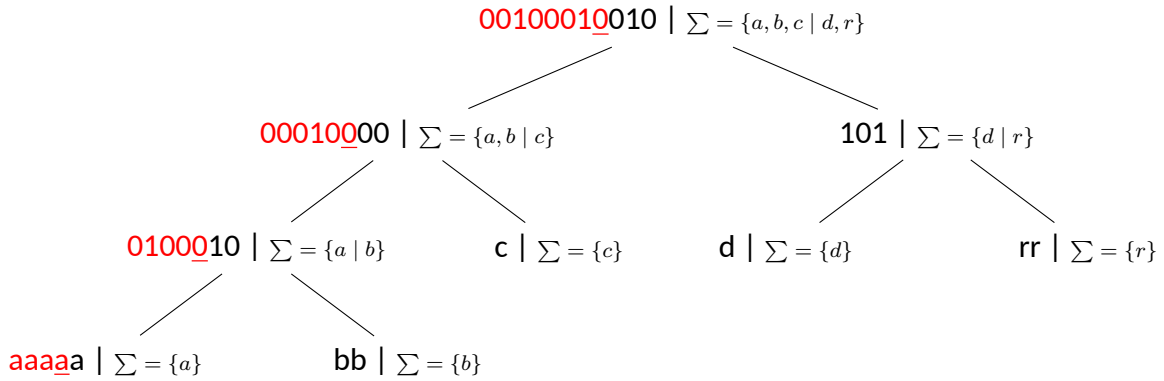


Figura 7: Ejemplo operación *Select* aplicado en Wavelet Tree

Como se observa, de manera análoga a la operación *Rank*, es necesario recorrer todos los niveles del árbol, por lo que también posee un tiempo de $O(\lg \sigma)$. Al finalizar se llega al nodo raíz con la posición del carácter deseado, obteniendo en este caso $s.Select_a(4) = 8$.

2.4.3. WT - ACCESS

Para aplicar la operación de búsqueda elemental *Access* a esta estructura, dado una posición inicial en el nodo raíz, debemos comprobar si dicha posición es un 0 o un 1 para saber a qué división del alfabeto pertenece y descender al siguiente nodo. Además, se debe utilizar la operación *Rank* con este valor para obtener la posición a trabajar en el siguiente nodo. La operación se aplica de manera análoga hasta llegar al nodo hoja.

Por ejemplo, como se observa en la Figura 8, si se quiere obtener $s.Access(7)$ se tiene que el nodo raíz en dicha posición posee un 1, luego aplica la operación *Rank* para calcular a qué 1 corresponde, se obtiene que es el segundo. Por lo que la posición queda en el segundo valor del nodo hijo de la derecha, el cual corresponde a un 0 que al aplicar *Rank* nos indica que es el primero. Por último, se descende al nodo hoja izquierdo de valor "d", y al finalizar el proceso se obtiene $s.Access(7) = d$.

Dado que también es necesario recorrer el árbol de raíz a hoja, la operación *Access* en esta estructura toma tiempo $O(\lg \sigma)$.

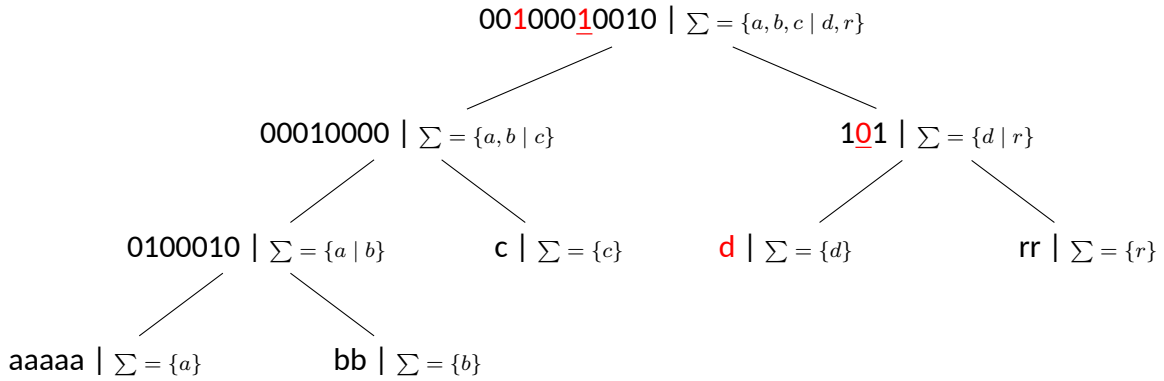


Figura 8: Ejemplo operación Access aplicado en Wavelet Tree

2.5. ALGORITMO FM-COUNT

El algoritmo *Count* basado en el procedimiento *backward-search* [Ferragina, 2005], retorna la cantidad de ocurrencias de un patrón P en un texto T , este utiliza la transformación de Burrows-Wheeler junto con sus componentes y comienza la búsqueda desde el final del patrón hasta su inicio.

Algorithm 1 FM-Count

```

1: procedure FM-Count( $P[0..m-1]$ )
2:    $b \leftarrow 0$ 
3:    $e \leftarrow n$ 
4:   for  $i \leftarrow m - 1$  downto 0 do
5:      $c \leftarrow P[i]$ 
6:      $b \leftarrow C[c] + \text{rank}_L(c, b)$ .
7:      $e \leftarrow C[c] + \text{rank}_L(c, e)$ .
   return  $e - b$ 

```

Sea el caso $T = \text{mississippi}$, en la Figura 9 se observa el resultado al aplicar la Transformación de Burrows-Wheeler.

F	L
\$	mississippi
i	\$mississipp
ippi	\$mississ
issippi	\$miss
issippi	\$m
mississippi	\$
pi	\$mississip
ppi	\$mississi
sippi	\$missis
sissippi	\$mis
ssippi	\$missi
ssissippi	\$mi

Figura 9: BWT en string $T = \text{mississippi}$

.

Se obtiene el arreglo $C = [0, 1, 5, 6, 8, 12]$.

Al evaluar $FM\text{-}Count(P)$ considerando el patrón $P = iss$ y el texto T , se comienza buscando el último carácter del patrón, el cual corresponde a s , y su primera ocurrencia dentro de L es $C[s] = 8$. Se tiene inicialmente $b = 0$ y $e = 12$.

F	L	
0	\$mississippi	$\rightarrow rank_L(s, 0) = 0$
1	i\$mississipp	
2	ippi\$missis	s
3	issippi\$mis	s
4	ississippi\$m	
5	mississippi\$	
6	pi\$mississip	
7	ppi\$mississi	
8	sippi\$missi	s
9	sissippi\$mi	s
10	ssippi\$missi	
11	ssissippi\$mi	
12		$\rightarrow rank_L(s, 12) = 4$

Figura 10: Ejemplo Count patrón "iss" en string T - Paso 1

Resultando:

$$b = C[s] + rank_L(s, 0) = 8 + 0 = 8$$

$$e = C[s] + rank_L(s, 12) = 8 + 4 = 12$$

Por lo tanto, existen $e - b = 4$ ocurrencias del carácter s en T , se puede observar en la Figura 10 que en la columna F dentro del intervalo $[8, 12)$ encontramos los patrones que comienzan con s . Para el siguiente paso, se busca el siguiente carácter del patrón el cual corresponde nuevamente a una s , pero utilizando los b y e resultantes en el paso 1, esto posiciona los índices b y e de manera tal que sabremos la cantidad de ocurrencias del patrón ss .

F	L	
0	\$mississippi	
1	i\$mississipp	
2	ippi\$missis <u>s</u>	
3	issippi\$mis <u>s</u>	
4	ississippi\$m	
5	mississippi\$	
6	pi\$mississip	
7	ppi\$mississi	
8	sippi\$missi <u>s</u>	$\rightarrow rank_L(s, 8) = 2$
9	sissippi\$mi <u>s</u>	
10	ssippi\$missi	
11	ssissippi\$mi	
12		$\rightarrow rank_L(s, 12) = 4$

Figura 11: Ejemplo Count patrón "iss" en string T - Paso 2

Resultando:

$$b = C[s] + rank_L(s, 0) = 8 + 2 = 10$$

$$e = C[s] + rank_L(s, 12) = 8 + 4 = 12$$

Por lo que, el patrón ss posee $e - b = 2$ ocurrencias en T , se puede observar en la Figura 11 que en la columna F el intervalo $[10,12)$ posee aquellas s que son antecidas por otro carácter s . Para el siguiente y último paso se realiza la búsqueda del carácter i con los índices b y e obtenidos.

F	L	
0	\$mississipp <i>i</i>	
1	i\$mississipp	
2	ippi\$mississ	
3	issippi\$miss	
4	ississippi\$m	
5	mississippi\$	
6	pi\$mississip	
7	ppi\$mississ <i>i</i>	
8	sippi\$missis	
9	sissippi\$mis	
10	ssippi\$miss <i>i</i>	$\rightarrow rank_L(i, 10) = 2$
11	ssissippi\$m <i>i</i>	
12		$\rightarrow rank_L(i, 12) = 4$

Figura 12: Ejemplo Count patron "iss" en string T - Paso 3

Resultando:

$$b = C[i] + rank_L(s, 0) = 1 + 2 = 3$$

$$e = C[i] + rank_L(s, 12) = 1 + 4 = 5$$

Dado que era el final del patrón se tiene que $e - b = 2$ corresponde a $FM-Count(P) = 2$. En la Figura 12 se puede observar que en el intervalo [3,5) las filas inician con el patrón $P = iss$.

Finalmente, la operación $FM-Count$ posee un orden de tiempo $O(m \log \sigma)$, dado que por cada símbolo del patrón $P[1..m]$ se debe realizar una búsqueda en la BWT.

2.6. ALPHABET PARTITIONING (AP)

Publicado en 2014, consiste en dividir el alfabeto original en subalfabetos de manera tal que los caracteres solo pertenezcan a uno de ellos, de esta manera el problema se divide en particiones que se pueden utilizar para realizar búsquedas y obtener la información del texto original [Barbay et al., 2014]. La cantidad de espacio utilizado por esta estructura esta definido por $nH_0(s) + o(nH_0(s))$. En cuanto a las operaciones Rank y Access son logradas en tiempo $O(\lg \lg \sigma)$, en cambio la operación Select logra realizarlo en tiempo $O(1)$.

Dado un alfabeto Σ , la idea es dividirlo en p subalfabetos $\Sigma_1, \dots, \Sigma_p$, de tal manera que:

- $\cup_{i=1}^p \sum_i = \sum$
- $\sum_i \cap \sum_j = \emptyset, \forall i \neq j$

La unión de todos estos alfabetos entrega el alfabeto original y la intersección de a pares entre los subalfabetos es vacía, es decir, no comparten caracteres entre ellos y de esta manera se logra particionar el alfabeto.

Sea el string $s = alabar_a_la_alabarda$, se tiene que $\sum = \{a, l, _, b, r, d\}$ y su particionamiento queda de la siguiente manera:

$$\sum_1 = \{a\}, \sum_2 = \{l, _ \}, \sum_3 = \{b, r\}, \sum_4 = \{d\}$$

Luego, se tiene un arreglo t del mismo largo del string s en el que por cada carácter se anota el número de la partición a la que pertenece, por ejemplo, para la primera letra se tiene que $a \in \sum_1$. Al repetir este proceso con cada uno de los caracteres se obtiene:

$t = 1 \ 2 \ 1 \ 3 \ 1 \ 3 \ 2 \ 1 \ 2 \ 2 \ 1 \ 2 \ 1 \ 2 \ 1 \ 3 \ 1 \ 3 \ 4 \ 1$

Posteriormente, se generan las siguientes s_i componentes según cada partición, por cada carácter de s se almacena al s_i respectivo relacionado al subalfabeto que contenga al carácter, respetando el mismo orden del texto original. Lo que resulta en:

s_1	s_2	s_3	s_4
aaaaaaaaa	l_l_l	brbr	d

Para determinar a qué subalfabeto corresponde cada carácter se utiliza un arreglo $m[1..\sigma]$, tal que $m[i] = j$, lo que indica a qué alfabeto $\sum_j$ ha sido mapeado i . Además dentro de $\sum_j$ los símbolos deben ser enumerados de 0 a $|\sum_j| - 1$. Para esto se utiliza como criterio que al existir k símbolos más pequeños que i asignados a $\sum_j$ se debe codificar i como k en $\sum_j$. Un método conocido para realizar estas particiones llamado **sparse**, se define formalmente:

$$m[\alpha] = \lceil \lg \left(\frac{n}{n_\alpha} \lg n \right) \rceil \quad (2)$$

En la que $\alpha \in \sum$, en el cual ocurre n_α veces en s , considerando que $m[\alpha] \leq \lceil \lg n^2 \rceil$

Los componentes t y s_i pueden ser representados por Wavelet Trees (*WT*), por lo tanto al dividir el problema en vez de trabajar con un único *WT* nos enfrentamos a aplicar con una mayor cantidad de estas estructuras. Sin embargo, no es un problema, dado que se particiona y se generan alfabetos pequeños, por lo que, los *WT* no serán árboles de gran altura y podrán funcionar de manera efectiva, sacando provecho a contextos en los cuales el alfabeto es muy grande.

De este modo dividimos el problema original. El ejemplo queda con 4 particiones que poseen sus strings y alfabetos respectivos, la idea es solo utilizar t y las particiones, dejando de lado el string original s , pero aún soportando las operaciones de búsquedas.

2.6.1. AP - RANK

Se requiere saber la cantidad de caracteres l que contiene el string s hasta la posición 16, es decir, $s.Rank_l(16)$. Primero se debe reconocer a qué alfabeto pertenece el carácter l , para este ejemplo se tiene $l \in \sum_2 = \{l, _ \}$, por lo tanto, se realiza un conteo de los valores 2 en el arreglo t hasta la posición 16, denotado por $t.Rank_2(16) = 6$. Este nuevo índice debe ser utilizado en el string s_2 dado que corresponde al substring de la partición respectiva al alfabeto al cual pertenece l , quedando $s_2.Rank_l(6) = 3$, por lo tanto:

$$s.Rank_l(16) = s_2.Rank_l(6) = 3$$

Formalmente definido por:

$$s.Rank_\alpha(i) = s_l.Rank_c(t.Rank_l(i)) \quad (3)$$

En el que $c = m.Rank_l(\alpha)$, $l = m.access(\alpha)$ y $\alpha \in \sum$. El tiempo de operación total corresponde a $O(\lg \lg \sigma)$.

2.6.2. AP - SELECT

Se requiere obtener la posición de la segunda l que contiene el string s , es decir, $s.Select_l(2)$. Al no poseer acceso a s , se realiza un select en s_2 . Por otro lado, como ya se tiene conocimiento de que $l \in \sum_2$ se busca la segunda l en s_2 , es decir, $s_2.Select_l(2) = 4$. Luego se busca el índice del cuarto valor 2 en t , quedando $t.Select_2(4) = 10$. Finalmente, se tiene que la posición del segundo carácter l en s es 10.

$$s.Select_l(2) = t.Select_2(4) = 10$$

Formalmente definido por:

$$s.select_\alpha(j) = t.select_l(s_l.select_c(j)) \quad (4)$$

Con $l = m.access(\alpha)$ y $\alpha \in \sum$. El tiempo para esta operación es de $O(1)$

2.6.3. AP - ACCESS

Se requiere obtener el carácter respectivo a la posición 5 del string s , es decir, $s.Access(5)$. Para obtener el resultado de la operación basta con utilizar el valor de la posición en $t[5] = 1$.

Luego, se debe contar cuantos 1 existen hasta esa posición, siendo $t.Rank_1(5) = 3$. Por último, este valor lo utilizaremos en el string de su partición respectiva s_1 , quedando:

$$s.Access(5) = s_1.Access(3) = a$$

Formalmente definido por:

$$s.access(i) = m.select_l(s_l.access(t.rank_l(i))) \quad (5)$$

En el que $l = t[i]$ y $\alpha \in \sum$. El tiempo total utilizado corresponde a $O(\lg \lg \sigma)$

2.7. A PRACTICAL ALPHABET-PARTITIONING RANK/SELECT DATA STRUCTURE (ASAP)

La estructura ASAP es una intervención de la estructura AP, la cual utiliza vectores de bits logrando mejorar los resultados, utilizando un poco más de memoria [Arroyuelo y Sepúlveda, 2019].

Utilizando de base el mismo ejemplo que en AP, en el que se tiene el string $s = alabar_a_la_alabarda$, se tiene que $\sum = \{a, l, _, b, r, d\}$ y su particionamiento queda de la misma manera:

$$\sum_1 = \{a\}, \sum_2 = \{l, _ \}, \sum_3 = \{b, r\}, \sum_4 = \{d\}$$

Junto con los componentes s_i respectivos:

$$\begin{array}{cccc} s_1 & s_2 & s_3 & s_4 \\ aaaaaaaaa & l_l_l & brbr & d \end{array}$$

La principal diferencia es que ya no se utiliza el arreglo t , sino que es reemplazado por un arreglo de p vectores de bits B_i , correspondiendo cada uno a su partición respectiva.

Cada vector de bits marca con un 1 donde se encuentran los elementos de cada partición, es decir, $B_i[j] = 1$ si y solo si $s[j] \in \sum_i$, quedando para el ejemplo actual:

$$B_1 = 101010010010101001$$

$$B_2 = 01000010110101000000$$

$$B_3 = 00010100000000010100$$

$$B_4 = 00000000000000000010$$

Con estos vectores de bits se pueden realizar las mismas operaciones que se desarrollan con el arreglo t , la diferencia es que ya no se utiliza un WT , sino que utiliza un arreglo de vectores de bits. Lo cual es una ventaja en tiempos de ejecución, ya que antes se debía pasar por los WT 's de cada partición para transformar las variables locales a globales, en cambio con este método se logra al utilizar el vector de bits de la partición respectiva.

2.7.1. ASAP - RANK

Tomando el caso de ejemplo, si se desea saber cuantas l hay hasta la posición 10, es decir $s.Rank_l(10)$, primero se debe notar que $l \in \sum_2$, luego si se cuenta cuantos 1's hay en B_2 hasta la posición 10 se tiene que $B_2.Rank_1(10) = 4$, por último, se obtiene cuantas l posee s_2 hasta la posición 4, finalmente:

$$s.Rank_l(10) = s_2.Rank_l(4) = 2$$

Formalmente:

$$s.Rank_\alpha(i) = s_l.Rank_c(B_l.Rank_1(i)) \quad (6)$$

En el que $l = m.Access(\alpha)$, $c = m.Rank_l(\alpha)$ y $\alpha \in \sum$. El tiempo de ejecución para esta operación esta dado por $O(\lg n)$.

2.7.2. ASAP - SELECT

Se requiere obtener la posición de la segunda l que contiene el string s , es decir, $s.Select_l(2)$. Para esto se realiza un Select en s_2 , ya que se conoce que $l \in \sum_2$. Por lo tanto buscamos la segunda l en s_2 , es decir, $s_2.Select_l(2) = 4$, luego se busca el índice del cuarto valor 1 en B_2 , quedando $B_2.Select_1(4) = 10$, finalmente se tiene que la posición del segundo carácter l en s es 10.

$$s.Select_l(2) = t.Select_2(4) = 10$$

Formalmente:

$$s.Select_\alpha(j) = B_l.Select_1(s_l.Select_c(j)) \quad (7)$$

En el que $l = m.Access(\alpha)$, $c = m.Rank_l(\alpha)$ y $\alpha \in \sum$. El tiempo para esta operación es de $O(1)$.

2.7.3. ASAP - ACCESS

Se requiere obtener el carácter respectivo a la posición 5 del string s , es decir, $s.Access(5)$. Con esta operación existe una gran diferencia, dado que ASAP no conoce a qué partición pertenece el carácter, es por esto que esta obligado a chequear en cada una, hasta encontrar aquel vector de bits con un 1 en la posición respectiva. Luego se debe contar cuantos 1 existen hasta esa posición en el vector de bits de la partición respectiva, siendo $B_1.Rank_1(5) = 3$, por último, este valor lo utilizaremos en el string de su partición s_1 , quedando:

$$s.Access(5) = s_1.Access(3) = a$$

Formalmente definido por:

$$s.Access(i) = m.Select_l(s_l.Access(B_l.Rank_l(i))) \quad (8)$$

En la que $l = m.Access(\alpha)$, $c = m.Rank_l(\alpha)$ y $\alpha \in \Sigma$.

2.8. RUN LENGHT ENCONDING

Patentado en 1983 por Hitachi, es una forma simple de compresión de datos, en que las secuencias de substrings o caracteres del mismo valor son almacenados junto con su cantidad de repeticiones, por lo que son muy útiles en textos que posean una gran cantidad de subcadenas repetidas [Fuentes-Sepúlveda et al., 2018].

Por ejemplo, sea un string $s = AAAAA BBBBAAABBBBAAAAACCCCC$

Sea s_f que almacena el nuevo texto. Al momento de aplicar run lenght encoding se obtiene que la primera subcadena corresponde a la letra A con un recuento de 5 repeticiones, quedando $s_f = 5A$. Luego continua la letra B con 3 repeticiones, por lo que $s_f = 5A3B$. Se debe reiterar con todos los caracteres restantes quedando de la siguiente manera:

$$s_f = 4A3B3A4B5A5C$$

2.9. ELIAS-FANO

Elias-Fano consiste es una codificación utilizada para secuencias monótonas, logra una buena compresión de la secuencia a trabajar y permite realizar operaciones de búsqueda elementales *rank* y *select* en tiempos eficientes [Ottaviano y Venturini, 2014].

Sea una secuencia $S[0, m - 1]$, en la que cada uno de sus valores es positivo y considerando un universo u del cual son extraídos los elementos, para todo $0 \leq i \leq m$ se cumple que $S[i] \leq S[i + 1]$.

Dado un valor entero l , para cada elemento de S , se agrupan sus $\lceil \log(u) \rceil - l$ bits más significados. Por ejemplo sea $S = \{1, 5, 12, 15, 23\}$, $l = 2$ y $u = 24$, se obtiene la Tabla 1.

Valores S	1		5		12		15		23	
Codificación binario	000	01	001	01	011	00	011	11	101	11

Tabla 1: Ejemplo Elias-Fano, $\lceil \log u \rceil - l$ bits más significativos y l bits menos significativos para cada elemento de S .

bits más significativos	000	001	010	011	100	101	110	111
Cantidad	1	1	0	2	0	1	0	0
Codificación unario	1	1		11		1		

Tabla 2: Ejemplo Elias-Fano, Cardinalidades de cada grupo de $\lceil \log u \rceil - l$ bits.

Para cada grupo generado a partir de los $\lceil \log u \rceil - l$ bits se obtiene la cantidad de ocurrencias de cada uno, el valor será anotado con su codificación en unario. Para el ejemplo propuesto se tiene que:

Luego, la estructura utilizará un bit vector para almacenar la unión de todas las codificaciones unarias separadas por un valor 0. Al utilizar lo obtenido en la Tabla 2 se observa la secuencia 1010011001000, la cual requiere a lo más $m + \frac{u}{2^l}$ bits. Por otra parte, la estructura almacena en otro arreglo una secuencia con los l bits menos significantes de cada valor, continuando con el mismo ejemplo se obtiene 0101001111 y se ocupa ml bits.

Dado que el manejo de l es fundamental para conseguir minimizar el uso de memoria requerida, dicho valor se obtiene al calcular $\log \frac{u}{m}$. Por último, se tiene que la estructura utiliza $m \lceil \log u \rceil + 2m$ para la construcción de ambos arreglos generados.

2.10. PARTITIONED ELIAS-FANO

Partitioned Elias-Fano (PEF) es un método que busca comprimir la información en bloques, se tiene que cada bloque es representado ya sea por un bit vector, runs o con Elias-Fano. Además, cuenta con un arreglo L que posee el último elemento que contiene cada uno de los bloques [Ottaviano y Venturini, 2014].

Por otra parte, incluye un bit vector que tiene una cantidad de valores equivalente a la cantidad de bloques a generar, donde si el valor i del bit vector posee un 1 es por que el bloque i fue representado con Elias-Fano. Por otro lado, si el valor es 0 se debe corroborar si la representación del bloque no es nula; en caso de no ser nulo, corresponde a un bloque representado con un bit vector, en caso contrario, el bloque corresponde a un runs de valores 1.

Entre las versiones de PEF, se encuentra la versión *PEF uniform* y *PEF ϵ -opt*, en esta la principal diferencia está en el criterio utilizado para la construcción de bloques. Las operaciones de búsqueda *Rank* y *Select* dependerán directamente de cómo está implementado PEF.

2.11. S18 VECTOR

La estructura S18 vector, basada en el S9 vector, permite comprimir conjuntos de números ordenados [Arroyuelo y Weitzman, 2020], y hace uso principalmente de dos estructuras ya conocidas para llevar a cabo su objetivo. Por un lado, utiliza Runs-Lenght Encoding (RLE), ya que posee un gran rendimiento cuando se trabajan conjuntos que tienden a generar repeticiones; por otra parte, también hace uso de la estructura GAP, la cual posee un mejor rendimiento que RLE cuando se trata de conjuntos que poseen elementos solitarios o sin generar demasiados runs. De este modo, la estructura consigue optimizar el uso de ambas auxiliares logrando un resultado eficiente tanto en el uso de memoria como en el tiempo de ejecución de sus operaciones de búsqueda elementales.

Sea el conjunto $S = \{x_1, \dots, x_n\} \subseteq \{1, \dots, u\}$ y $G_s = \{g_1, \dots, g_n\}$ que corresponde a un conjunto de gaps, se tiene que $g_1 = x_1$ y $g_i = x_i - x_{i-1}$ para cada $i \in \{2, \dots, n\}$. La idea principal es utilizar el conjunto de gaps, en el que cada runs del conjunto S , que posea l elementos sucesivos, sean traducidos en l gaps, es decir, $g_i = 1, g_{i+1} = 1, \dots, g_{i+l-1} = 1$. Cuando se trata de un runs, S18 vector almacena directamente el valor l del largo respectivo, a diferencia de S9 que almacena los l 1's. Además, S18 agrega criterio extras al momento de codificar el conjunto de gaps.

La estructura posee dos fases. En primer lugar, utiliza la versión original para comprimir los gaps, en segundo lugar, trabaja las S9 words generadas en el paso previo y las reemplaza en base a los criterios de construcción de la estructura, utilizando $32s + O(\frac{s}{b} \lg u)$ bits.

Luego de realizar las dos fases, se obtiene un arreglo $W[1..s] = w_1, w_2, \dots, w_s$, que corresponde a la secuencia de S18 words, el cual luego es dividido en bloques de b S18 words por cada uno. Además, se almacena un arreglo $idx_univ[1..s/b]$ que corresponde al número de elementos cubiertos por el conjunto representado por $W[1..(i-1)b]$. Y el arreglo $idx_set[1..s/b]$ que almacena el número de elementos codificados del conjunto $W[1..(i-1)b]$.

En cuanto a las operaciones de búsqueda, permite el uso de *rank* y *select* de manera eficiente. Por un lado, la operación *rank* hace uso principalmente del arreglo $idx_univ[1..s/b]$, en el que para obtener el resultado de un elemento x se realiza una búsqueda del predecesor del x . Por otra parte, en cuanto a la operación *select* utiliza el arreglo $idx_univ[1..s/b]$ buscando el predecesor de x .

2.12. RL RUNS

Consiste en una estructura que permite comprimir secuencias de alfabetos grandes utilizando $r \log \frac{n\sigma}{r} + o(r \log \frac{n\sigma}{r})$ bits, además permite el uso de las operaciones *rank* y *select* en tiempos $O(\log \frac{\log(\frac{n\sigma}{r})}{\log \log n})$ y $O(\log \frac{\log(\frac{n}{r})}{\log \log n})$ respectivamente [Fuentes-Sepúlveda et al., 2018].

Considerando un string s de largo n que puede ser representado como una concatenación de r runs de letras dado un alfabeto de $[0..\sigma - 1]$ en el que $2 \leq \sigma \leq r$; se tiene la representación $n_0, n_1, \dots, n_{\sigma-1}$ que corresponde al número de runs para el carácter $0, 1, \dots, \sigma - 1$ respectivamente. Sea $c \in [0..\sigma - 1]$ y $i \in [1..n_c]$, se utiliza la representación $l_{c,i}$ para denotar el largo de cada run i de la letra c . La estructura codifica s utilizando las siguientes componentes:

- Un arreglo de bits $R[0..n - 1]$ tal que $R[i] = 1$ si y solo si $s[i] \neq s[i + 1]$.
- Un string $H[0..r - 1]$ tal que $H[i] = s[\text{select}_1(i + 1, R)]$.
- Un arreglo de bits $C[0..r + \sigma - 1]$ que corresponde a la concatenación de la codificación en unario de los números de runs de cada letra.
- Un arreglo de enteros $S[0..r - 1]$ que almacena los valores $l_{c,i}$ en el siguiente orden:
 $l_{0,1}, l_{0,2}, \dots, l_{0,n_0}, l_{1,1}, \dots, l_{1,n_1}, l_{\sigma-1,1}, \dots, l_{\sigma-1,n_{\sigma-1}}$.

De este modo la estructura logra con R identificar la última letra de cada run en s , luego con el string H almacena las letras en el orden correspondiente, con C modifica el número de corridas de cada letra, y finalmente con S almacena las longitudes de cada corrida agrupadas por letras.

CAPÍTULO 3

PROPUESTA DE SOLUCIÓN

Dado que el contexto propone un texto con alfabeto grande y repetitivo, se presenta la estructura de datos *AR-RLFM*, la cual hace uso de la estructura *ASAP*, interviniéndola de tal forma que posea un mejor rendimiento al manipular textos de secuencias repetitivas y alfabetos grandes. Por tanto, la estructura utilizará la BWT del texto para aumentar la cantidad de runs en la secuencia, la que será trabajada con *ASAP* y aumentará la cantidad de runs en cada partición generada, de este modo, se aplicarán estructuras que posean eficiencia en el manejo de runs en las componentes de *ASAP*.

La estructura *ASAP* posee un arreglo p que contiene los vectores de bits B_i para cada partición generada. La versión original hace uso de un *sd vector* de la *sdsI*, sin embargo, dado las características del contexto que se está abordando y que se está empleando la BWT del texto, los bitvectors generados tendrán un gran número de runs de 1's. Por lo que *AR-RLFM* será creado con un *ASAP* utilizando un *s18 vector* o *PEF*, en conjunto a una versión con *sd vector* para comparar los resultados obtenidos por cada una de las implementaciones. De este modo, se logra corroborar si el uso de estructuras que poseen un manejo eficiente con bitvectors con runs generan una mejora en el rendimiento o resultados, ya sea en memoria o tiempos en operaciones de búsqueda.

Por otra lado, la estructura *ASAP* posee sus componentes s_i , que representan los substrings de cada partición generada. Originalmente son manipulados con un WT de Huffman, por lo que se interviene esta componente de manera tal que quede como un parámetro. Esto otorga una mayor libertad a *AR-RLFM* para sustituir su componente *ASAP* con diversas versiones de Wavelet Trees o estructuras que posean los requisitos de implementación, como las operaciones *Rank* y *Select*. Dado que se trabaja con la BWT, los s_i generados tendrán runs de caracteres según su subalfabeto, en base a esto, se busca corroborar si el uso de una estructura que posea un manejo eficiente en cuanto a strings con runs genera un cambio en el rendimiento de la estructura *AR-RLFM* en su uso de memoria o tiempo en operaciones de búsqueda.

3.1. PREPARACIÓN Y PROCESAMIENTO DE TEXTOS

La estructura propuesta inicialmente procesa el texto a utilizar a partir de la aplicación de un filtro. Este último consiste en la enumeración de cada palabra del texto utilizando un identificador numérico, lo que produce un texto con una extensa secuencia de números.

En cuanto al tamaño del alfabeto del texto resultante es posible aseverar que será grande. Esto se debe a que cada palabra que conforma el texto se considera dentro del alfabeto con su respectivo identificador. Por ende, el filtro aplicado incrementa el tamaño final del

alfabeto.

La filtración del texto mencionada permite a la estructura reconocer runs de frases con facilidad y utilizar una menor cantidad de memoria. Ya que, un número ocupa menos bytes a diferencia de una palabra de largo promedio. Se considera T_s como el resultado del filtro aplicado al texto T original.

Los criterios utilizados para la lectura son los siguientes:

- Sensibilidad a las mayúsculas.
- Caracteres especiales, saltos de línea y espacios poseen símbolos particulares.
- Palabras que posean caracteres especiales en uno o ambos de sus extremos serán representados por separado.
- Palabras que contengan caracteres especiales entremedio serán considerados como una sola palabra y, por lo tanto, representados por un símbolo único.
- Se distinguen espacios múltiples según su largo, cada largo genera un símbolo respectivo.

3.2. CREACIÓN DE AR-RLFM

AR-RLFM hace uso de la estructura ASAP, pero a diferencia de trabajar con el texto original T , utilizará la BWT generada a partir de T_s y de este modo se busca sacar provecho de las características de ASAP y los runs generados por la BWT.

En la Figura 2 se observa que luego de aplicar el filtro al texto T y generado el texto de símbolos T_s , se utilizará la librería *sdsI* [Gog et al, 2019], la cual permite generar el *Suffix Array* (sa) a partir de T_s . Luego se recorre cada uno de sus elementos aplicando la Relación 1 para obtener la BWT de T_s . Además, durante el recorrido del sa , se crea el arreglo C relacionado a las primeras ocurrencias de cada símbolo, dado que una vez que cambia de símbolo debe almacenar el índice como inicio de un nuevo símbolo. Este arreglo será fundamental al momento de realizar la operación de búsqueda para la estructura AR-RLFM. Por último, la BWT generada es entregada a la estructura ASAP como parámetro.

Algorithm 2 AR-RLFM

```

1: procedure AR-RLFM( $T[0..n-1]$ )
2:    $T_s = \text{Filtro}(T)$ 
3:    $sa = \text{construct\_sa}(T_s)$ 
4:    $uniq\_symbols = 0$ 
5:   for  $i = 0$  to  $sa.size()$  do
6:      $bwt[i] = T_s[sa[i] - 1]$ 
7:     if  $T_s[sa[i]] \neq T_s[sa[i - 1]]$  then
8:        $C[uniq\_symbols] = i$ 
9:        $uniq\_symbols += 1$ 
10:   $C[uniq\_symbols] = T_s.size()$ 
11:   $ASAP(bwt)$ 

```

3.3. AR-RLFM - OPERACIÓN COUNT

Como se puede observar en el Algoritmo 1 *FM-Count* se utiliza directamente la *BWT* del texto para obtener los índices b y e . En cambio, como se observa en la Figura 3 *AR-RLFM* hace uso de la estructura *ASAP*, la cual provee la operación de búsqueda *Rank* necesaria y trabaja de igual forma con la *BWT* del texto T_s , pero bajo los criterios y características de la estructura *ASAP*.

Algorithm 3 AR-RLFM-Count

```

1: procedure AR-RLFM-Count( $P[0..m-1]$ )
2:    $b \leftarrow 0$ 
3:    $e \leftarrow n$ 
4:   for  $i \leftarrow m - 1$  downto  $0$  do
5:      $c \leftarrow P[i]$ 
6:      $b \leftarrow C[c] + rank_{ASAP}(c, b)$ .
7:      $e \leftarrow C[c] + rank_{ASAP}(c, e)$ .
   return  $e - b$ 

```

3.4. AR-RLFM - IMPLEMENTACIONES

En cuanto al arreglo de Bitvectors p , se implementarán tres estructuras:

- *Sparse bitvector* de la *sdsl*.
- PEF, utilizando la implementación de Gabriel Carmona [Carmona, 2022].
- Estructura *S18 Vector*, se considera la implementación de Héctor Larrañaga [Larrañaga, 2022].

Mientras que para el arreglo de substrings s_i se implementarán diversas estructuras, las cuales son:

- Wavelet Tree de la sdsI
 - WT INT
 - WT AP
 - WT RLMN parametrizado con un WT INT
 - WT RLMN parametrizado con un WT AP
- RL RUNS de Fuentes et al. [Fuentes, 2018]
 - RL RUNS AP
 - RL RUNS RLMN

Finalmente, cada una de las versiones de bitvectors a implementar para el arreglo p será combinada con cada una de las estructuras a utilizar en el arreglo de substrings s_i . En la Tabla 3 se tienen las implementaciones a realizar.

Versión [ASAP_BV_WT]	AR-RLFM	Tipo Bit vector p	Tipo Wavelet tree s_i
ASAP_SD_INT		Sparse bitvector	WT_INT
ASAP_SD_AP		Sparse bitvector	WT_AP con t_wt=WT_HUFF
ASAP_SD_RLMN		Sparse bitvector	WT_RLMN con t_wt = WT_INT
ASAP_SD_RLMN-AP		Sparse bitvector	WT_RLMN con t_wt = WT_AP
ASAP_SD_RUNS-AP		Sparse bitvector	RL_RUNS con t_wt = WT_AP
ASAP_SD_RUNS-RLMN		Sparse bitvector	RL_RUNS con t_wt = WT_RLMN
ASAP_PEF_INT		PEF	WT_INT
ASAP_PEF_AP		PEF	WT_AP con t_wt=WT_HUFF
ASAP_PEF_RLMN		PEF	WT_RLMN con t_wt = WT_INT
ASAP_PEF_RLMN-AP		PEF	WT_RLMN con t_wt = WT_AP
ASAP_PEF_RUNS-AP		PEF	RL_RUNS con t_wt = WT_AP
ASAP_PEF_RUNS-RLMN		PEF	RL_RUNS con t_wt = WT_RLMN
ASAP_S18_INT		S18 vector	WT_INT
ASAP_S18_AP		S18 vector	WT_AP con t_wt=WT_HUFF
ASAP_S18_RLMN		S18 vector	WT_RLMN con t_wt = WT_INT
ASAP_S18_RLMN-AP		S18 vector	WT_RLMN con t_wt = WT_AP
ASAP_S18_RUNS-AP		S18 vector	RL_RUNS con t_wt = WT_AP
ASAP_S18_RUNS-RLMN		S18 vector	RL_RUNS con t_wt = WT_RLMN

Tabla 3: Versiones de AR-RLFM generadas.

Fuente: Elaboración Propia.

CAPÍTULO 4

VALIDACIÓN DE LA SOLUCIÓN

4.1. ESTRUCTURAS A COMPARAR CON AR-RLFM

Las estructuras implementadas serán comparadas con una estructura Baseline que realizará un count idéntico a la estructura AR-RLFM, pero manipulando la BWT con estructuras ya conocidas, de este modo podremos comparar el rendimiento de las implementaciones tanto en tiempo como memoria utilizada.

Versión Baseline	Tipo Wavelet Tree
B_AP	WT_AP con: t_wt_byte=WT_HUFF , t_wt_int=WT_HUFF
B_AP-RLMN	WT_AP con: t_wt_byte=WT_RLMN , t_wt_int=WT_RLMN
B_RLMN	WT_RLMN
B_RUNS	RL_RUNS

Tabla 4: Versiones Bases con las cuales sera comparado AR-RLFM.
Fuente: Elaboración Propia.

4.2. ENTORNO DE EJECUCIÓN Y COMPILACIÓN

Se realizaron los experimentos en un servidor Intel(R) Xeon(R) CPU ES-2630 0 @ 2.30GHz, con el sistema operativo Linux versión 4.4.0,

Para compilar se utilizó la versión 7 de g++ junto con las siguientes flags: -std=c++17, -O3, -msse4.2, -DNDEBUG y -lstdc++fs.

4.3. S18 VECTOR Y b

Dado que una de las estructuras a implementar corresponde a S18 vector, se realiza un experimento previo para comprobar el valor ideal para el parámetro b utilizado por la estructura. Este parámetro se encuentra relacionado directamente a la división de bloques que genera. Por lo que se utilizó uno de los textos a trabajar en los experimentos aplicando un AR-RLFM con un S18 vector parametrizado con $b = \{8, 16, 32\}$.

Tal como se puede observar en la Figura 13, la relación entre b y el tiempo utilizado es directamente proporcional, mientras que, para el caso de la memoria necesaria, se tiene un comportamiento inversamente proporcional. Sin embargo, para $b = 8$, la cantidad de tiem-

po disminuye considerablemente en comparación a la cantidad de memoria extra que utiliza, por tanto, se utilizará las versiones de AR-RLFM con S18 vector con $b = 8$.

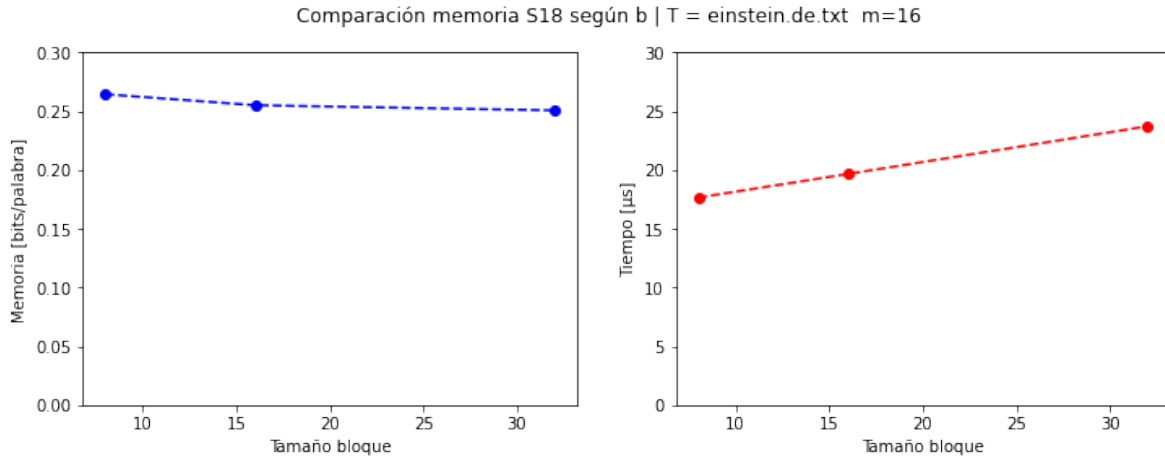


Figura 13: Resultados de experimento con AR-RLFM utilizando S18 vector para encontrar parámetro b ideal, utilizando el texto Einstein versión alemán con patrones de largo 16.

Fuente: Elaboración Propia.

4.4. EXPERIMENTOS

El contexto actual considera textos repetitivos y de alfabetos grandes, es por esto que se utilizó la biblioteca de Pizza&Chili Corpus [Ferragina y Navarro, 2005] que posee una colección de textos con diversas características ideales para los experimentos.

En cada uno de los textos se extrajeron fragmentos de distintos largos, los cuales serán utilizados como patrones a buscar en cada una de las versiones implementadas. Se realizaron 50 mil extracciones de largo $m = \{4, 8, 16\}$.

Se almacenará el tiempo que demora cada una de las implementaciones en realizar la función *Count* con cada uno de los patrones generados, los resultados serán promediados para obtener una estimación del tiempo utilizado por cada una. Por otra parte, en cuanto la memoria utilizada, se obtendrá la cantidad de bytes necesarias para construir cada una de las estructuras, multiplicadas por 8 para obtener la cantidad en bites y divididas en la cantidad total de símbolos del texto t , de este modo la memoria será interpretada en bits por palabras.

A continuación se presenta el detalle de cada texto junto con los resultados obtenidos por cada estructura.

4.4.1. ALBERT EINSTEIN INGLÉS

Este texto contiene la concatenación de las distintas versiones del artículo de Einstein, recopiladas desde Wikipedia en idioma inglés.

Información	Valor
Cantidad símbolos T_s	159.119.879
Símbolos distintos T_s	15.862
Cantidad runs $BWT(T_s)$	96.953
Largo promedio runs $BWT(T_s)$	1.640

Tabla 5: Albert Einstein versión ingles Información
Fuente: Elaboración Propia.

En la Tabla 5 se especifican las características del texto, por otro lado, en las Tablas 10 y 11(Anexo) se tienen los resultados de cada una de las estructuras, en las que aquella que utiliza menos memoria corresponde a la versión Baseline B_RUNS , con 0.0385[bits/palabra]. Mientras que en la versión AR-RLFM, la que utiliza menos memoria corresponde a $ASAP_S18_RLMN$ con 0.109[bits/palabra].

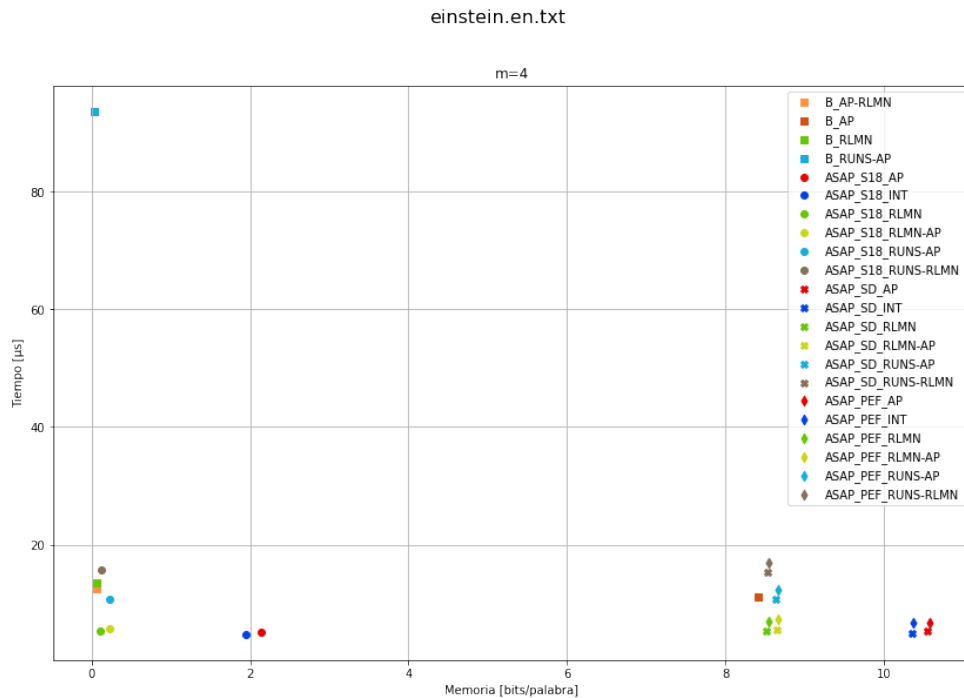


Figura 14: Resultados texto Einstein versión inglés m = 4
Fuente: Elaboración Propia.

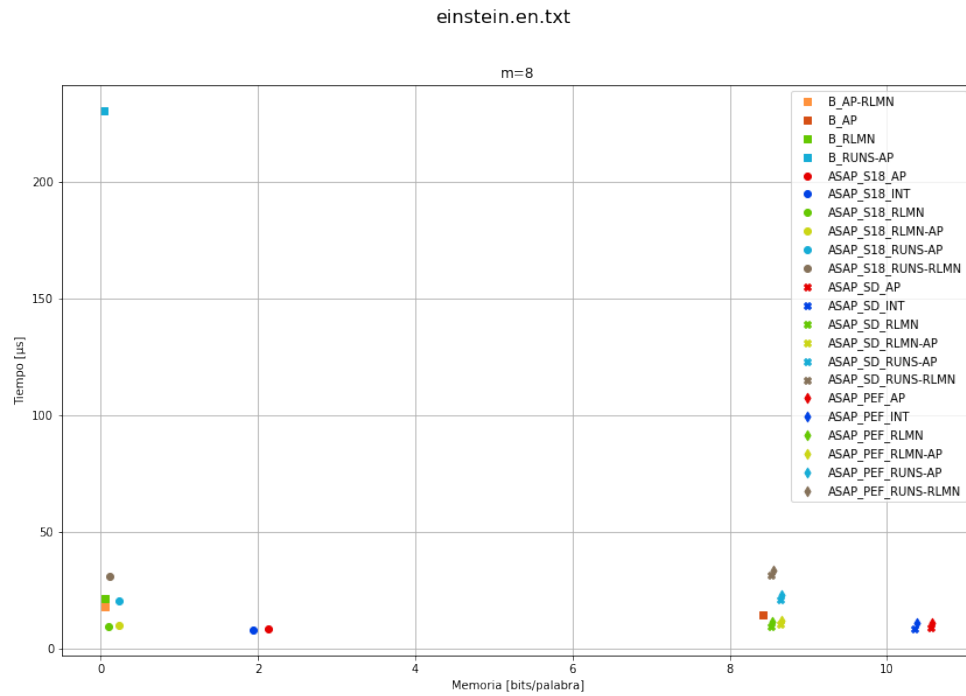


Figura 15: Resultados texto Einstein versión inglés m = 8
Fuente: Elaboración Propia.

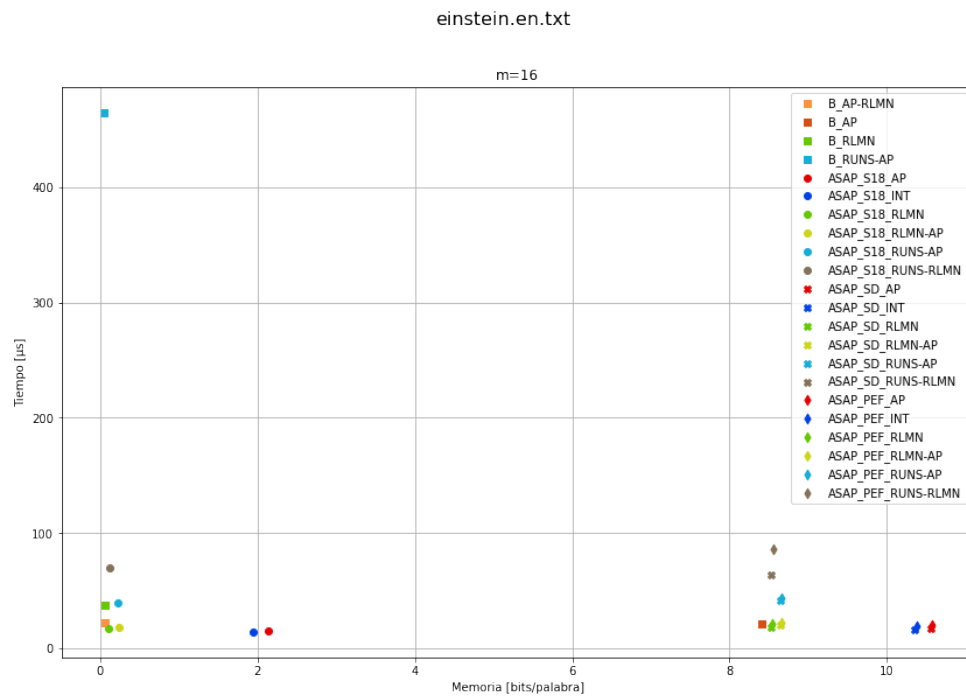


Figura 16: Resultados texto Einstein versión inglés m = 16
Fuente: Elaboración Propia.

Si bien la estructura Baseline B_RUNS fue la que utilizó menos memoria, se puede observar en las Figuras 14,15 y 16 que también es la estructura con el peor tiempo promedio. Por otro lado, si se compara con el Baseline versión $B_AP-RLMN$, aumenta el uso de memoria a 0.059[bits/palabra] y logra disminuir considerablemente el tiempo promedio.

En cuanto a AR-RLFM, podemos observar que para cualquier m , la versión $ASAP_S18_RLMN$ mantiene un tiempo promedio menor al Baseline $B_AP-RLMN$, a cambio de utilizar 0,05[bits/palabra] extras.

En relación con, $m = 4$, se tiene que la versión $B_AP-RLMN$ alcanza un tiempo promedio de 12.557[μs], en cambio, la versión $ASAP_S18_RLMN$ logra un tiempo promedio de 5.317[μs], disminuyendo en un 57.7 %.

Con $m = 8$, la versión $B_AP-RLMN$ posee un tiempo promedio de 18.038[μs] y la versión $ASAP_S18_RLMN$ un tiempo promedio de 9.203[μs], lo cual es un 49.0 % menor.

Por último con $m = 16$, la versión $B_AP-RLMN$ logra un tiempo promedio de 22.707[μs], mientras que la versión $ASAP_S18_RLMN$ logra un tiempo promedio de 17.155[μs], siendo 24,5 % menor.

4.4.2. ALBERT EINSTEIN ALEMÁN

Texto que contiene la concatenación de las distintas versiones del artículo de Einstein, recopiladas desde Wikipedia en idioma alemán.

Información	Valor
Cantidad símbolos T_s	29.970.946
Símbolos distintos T_s	7.105
Cantidad runs $BWT(T_s)$	40.349
Largo promedio runs $BWT(T_s)$	742

Tabla 6: Albert Einstein versión alemán Información
Fuente: Elaboración Propia.

En la Tabla 6 se especifican las características del texto, por otro lado, en las Tablas 12 y 13, se observa que la estructura que utiliza menos memoria es la versión Baseline B_RUNS , con 0.0741[bits/palabra], en cuanto a la versión AR-RLFM que utiliza menos memoria corresponde a $ASAP_S18_RLMN$ con 0.264[bits/palabra].

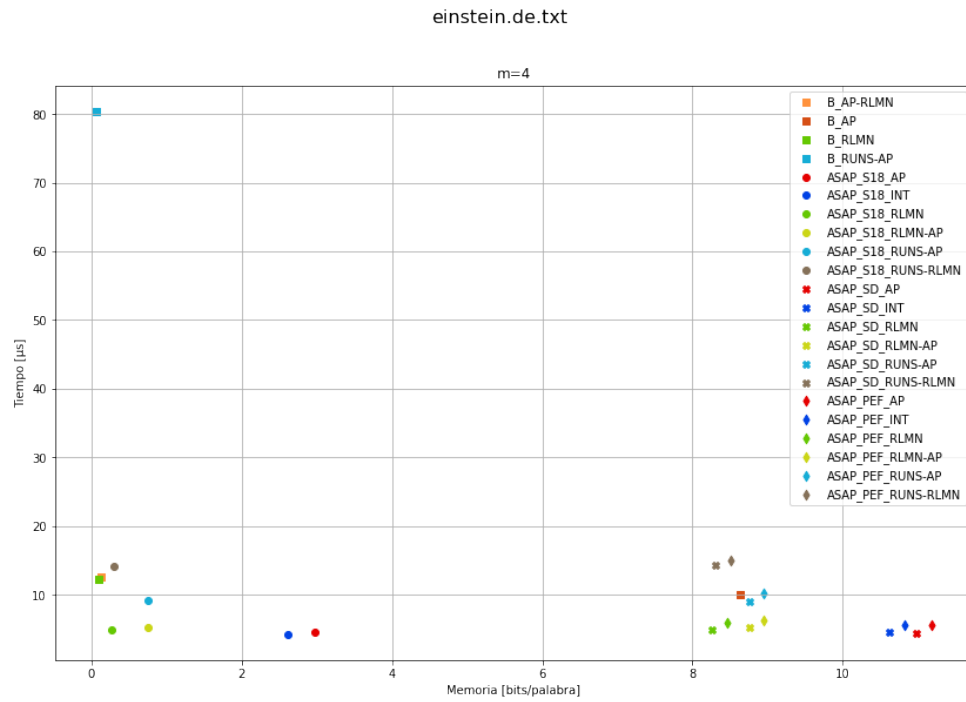


Figura 17: Resultados texto Einstein versión alemán m = 4
Fuente: Elaboración Propia.

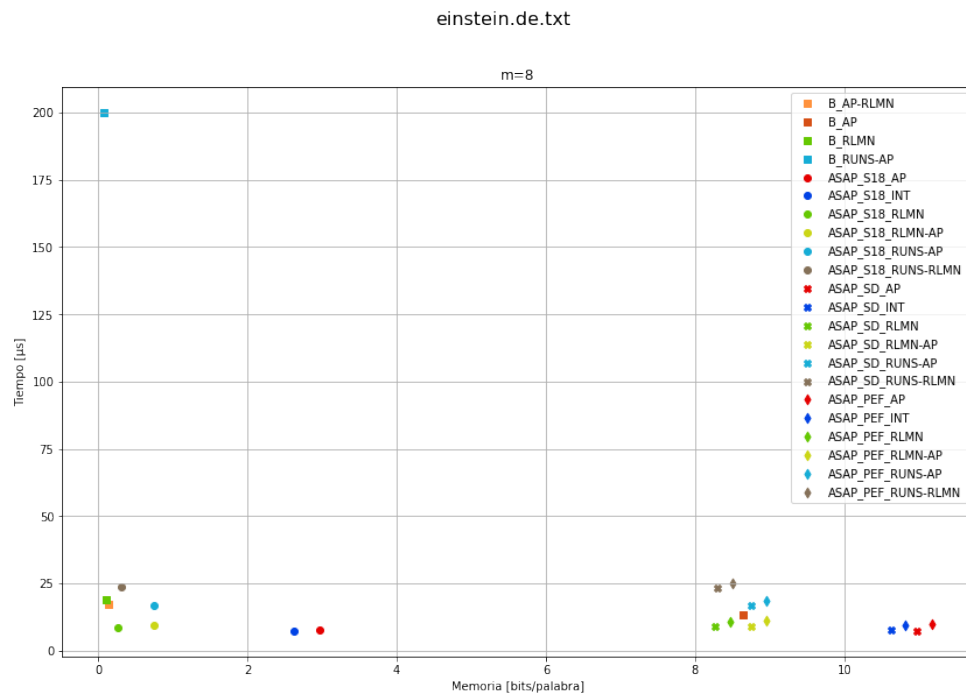


Figura 18: Resultados texto Einstein versión alemán m = 8
Fuente: Elaboración Propia.

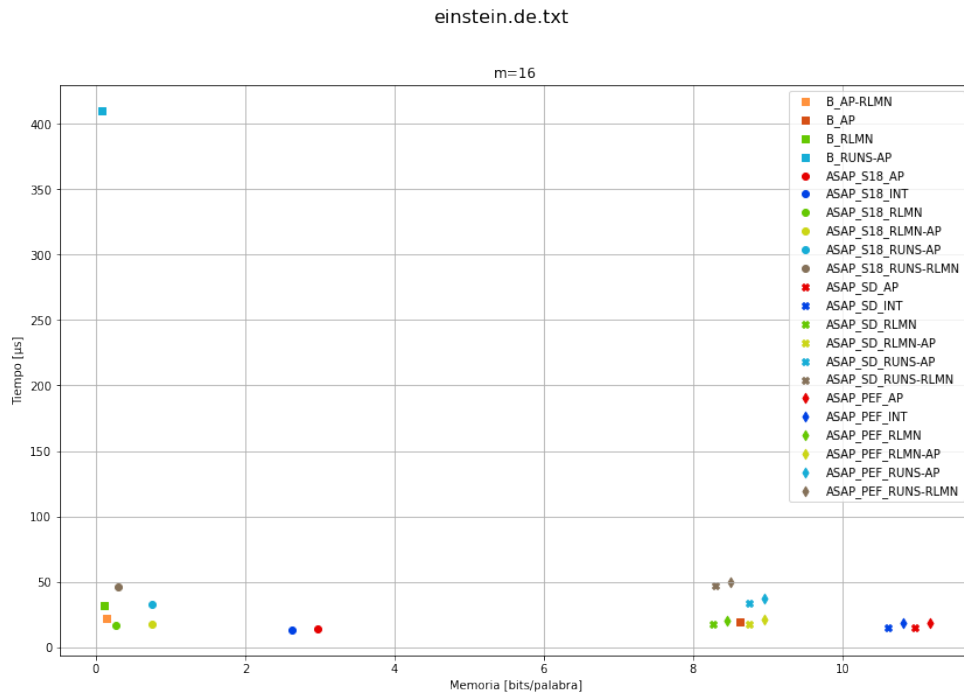


Figura 19: Resultados texto Einstein versión alemán $m = 16$

Fuente: Elaboración Propia.

La estructura Baseline B_RUNS nuevamente fue la que utilizó menos memoria, sin embargo, se puede observar en las Figuras 17, 18 y 19 que es la estructura con el peor tiempo promedio.

Con $m = 4$ como se puede observar en la Figura 17, se tiene que la versión Baseline que posee el menor tiempo promedio es B_RLMN , utilizando 0.104[bits/palabra] alcanza un tiempo promedio de 12.312[μs], sin tomar en cuenta la versión B_AP dado que su uso de memoria es considerablemente mayor.

Por otra lado, la versión AR-RLFM $ASAP_S18_RLMN$ posee un tiempo promedio de 4.84[μs], por lo que comparando con la versión Baseline B_RLMN disminuye en un 60.7 % el tiempo promedio utilizando 0.1604[bits/palabra] de memoria extra.

En cuanto a los resultados con $m = 8$, esta vez el Baseline $B_AP-RLMN$ es quien posee un tiempo promedio menor, con 17.346[μs] utilizando 0.139[bits/palabra].

La versión AR-RLFM $ASAP_S18_RLMN$ obtuvo un tiempo promedio de 8.637[μs], lo cual comparando con el Baseline $B_AP-RLMN$ logra disminuir en 50,2 % el tiempo promedio a cambio de utilizar 0.125[bits/palabra] de memoria extra.

Para $m = 16$ se tiene que nuevamente el Baseline que logra un menor tiempo promedio es $B_AP-RLMN$ con 22.396[μs], por otra parte, se tiene que la versión AR-RLFM $ASAP_S18_RLMN$ posee un tiempo promedio de 16.307[μs], lo cual es un 27,2 % menor uti-

lizando 0.126[bits/palabra] extra.

4.4.3. COREUTILS

Este texto contiene la concatenación de códigos fuentes, principalmente la recopilación de 9 versiones del paquete Coreutils5 sin considerar archivos binarios, junto con versiones 1.0.x y 1.1.x de Linux Kernel 6 ordenadas por fechas de creación.

Información	Valor
Cantidad símbolos T_s	93.319.681
Símbolos distintos T_s	148.654
Cantidad runs $BWT(T_s)$	2.540.091
Largo promedio runs $BWT(T_s)$	36

Tabla 7: Coreutils información.
Fuente: Elaboración Propia.

En la Tabla 7 se especifican las características del texto, por otro lado, en los resultados de las Tablas 14 y 15, se puede observar que la estructura que utiliza menos memoria nuevamente es la versión B_RUNS con 0.872[bits/palabra], en cuanto a la versión AR-RLFM que utiliza menos memoria corresponde a la versión $ASAP_S18_RLMN$ con 2.618[bits/palabra].

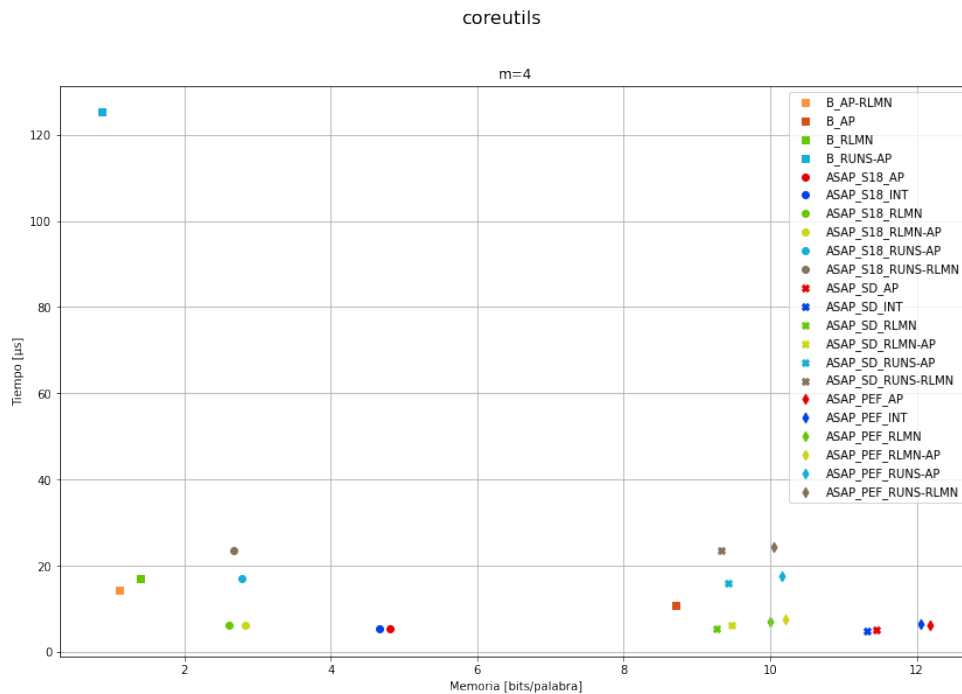


Figura 20: Resultados texto Coreutils m = 4
Fuente: Elaboración Propia.

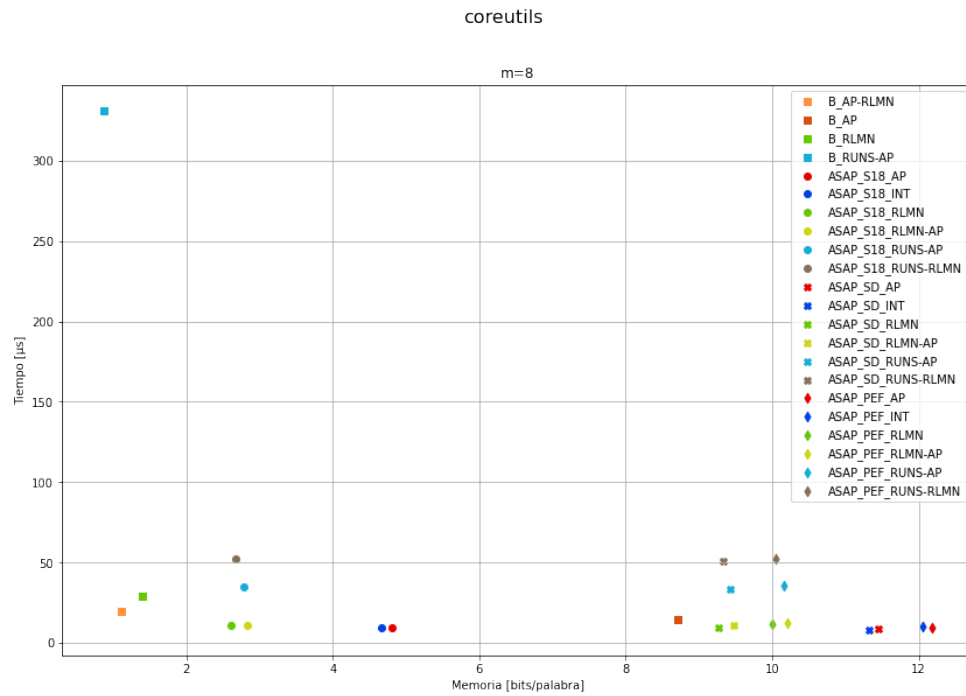


Figura 21: Resultados texto Coreutils m = 8
Fuente: Elaboración Propia.

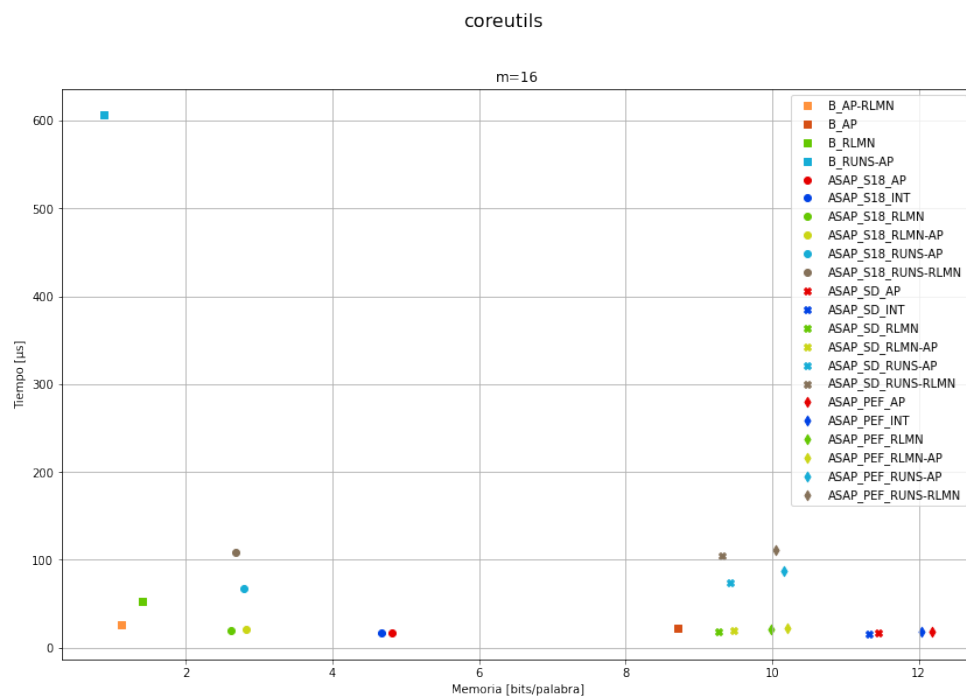


Figura 22: Resultados texto Coreutils m = 16
Fuente: Elaboración Propia.

De manera análoga a los textos anteriores, observando las Figuras 20,21 y 22, la versión Baseline B_RUNS a pesar de utilizar la menor cantidad de memoria, es quien posee los tiempos promedios más altos.

En cuanto a los patrones de largo $m = 4$, sin considerar la versión B_AP dado su alto uso de memoria, se tiene que la versión B_AP_RLMN es quien posee el menor tiempo promedio con 14.372[μs] utilizando 1.122[bits/palabra].

Por otra parte, la versión AR-RLFM con el menor tiempo promedio es la versión $ASAP_S18_RLMN$, con un promedio de 6.168[μs], si comparamos con la versión Baseline B_AP_RLMN , disminuye en un 57.1 % el tiempo promedio, utilizando 1.496 [bits/palabras] adicionales.

Para patrones de largo $m = 8$ se tiene que la estructura B_AP_RLMN aumenta a 19.975[μs], en cuanto a $ASAP_S18_RLMN$ obtiene un tiempo promedio de 9.707[μs], siendo un 51.4 % menor.

Para $m = 16$ se tiene que la versión Baseline B_AP_RLMN obtiene un tiempo promedio de 25.509[μs], en comparación a la versión AR-RLFM $ASAP_S18_RLMN$ que alcanza un tiempo promedio de 19.356[μs], lo cual es un 24.1 % menor.

4.4.4. WORLD LEADERS

Este texto contiene la concatenación de todos los documentos PDF de la CIA desde el año 2006 al 2009, archivos que fueron transformados a textos con el software *pdftotext*.

Información	Valor
Cantidad símbolos T_s	31.203.286
Símbolos distintos T_s	29.176
Cantidad runs $BWT(T_s)$	206.296
Largo promedio runs $BWT(T_s)$	150

Tabla 8: World Leaders Información
Fuente: Elaboración Propia.

En la Tabla 8 se especifican las características del texto, por otro lado, en los resultados de las Tablas 18 y 19, la versión que utiliza menos memoria corresponde a B_RUNS con 0.718[bits/palabra], sin embargo, para cualquier m posee el tiempo promedio más elevado. Por otra parte, la versión AR-RLFM que utiliza menos memoria es la versión $ASAP_S18_RLMN$ con 2.24[bits/palabra].

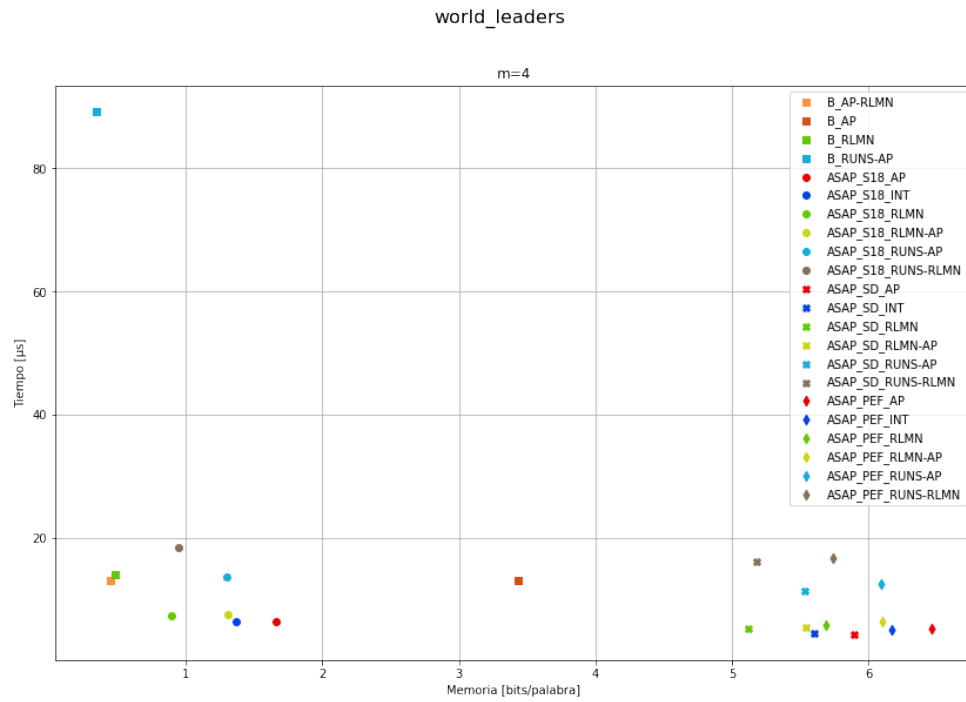


Figura 23: Resultados texto World Leaders m = 4
Fuente: Elaboración Propia.

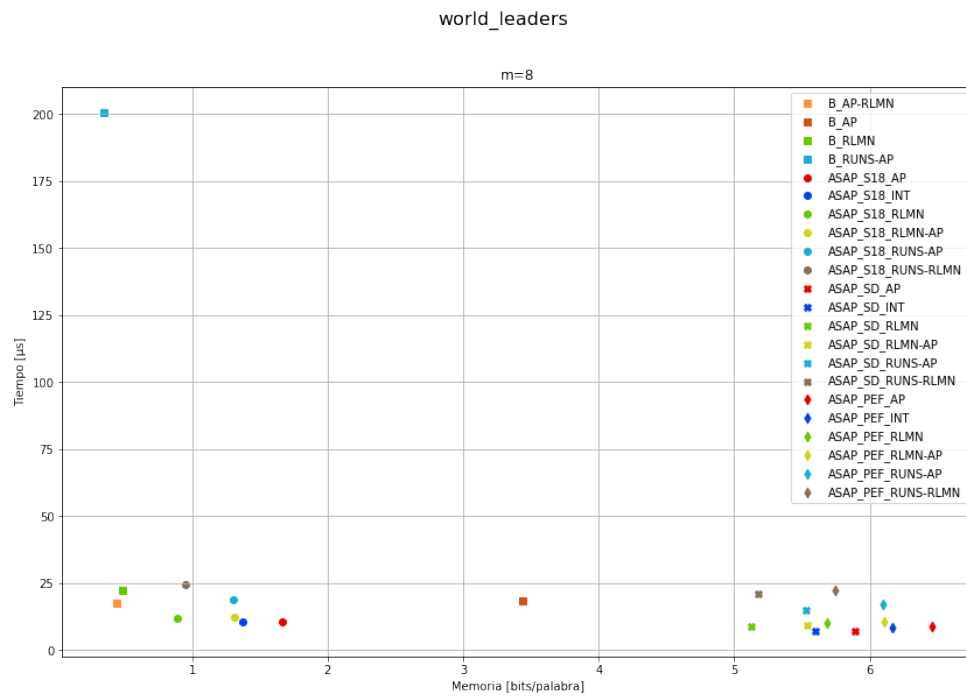


Figura 24: Resultados texto World Leaders m = 8
Fuente: Elaboración Propia.

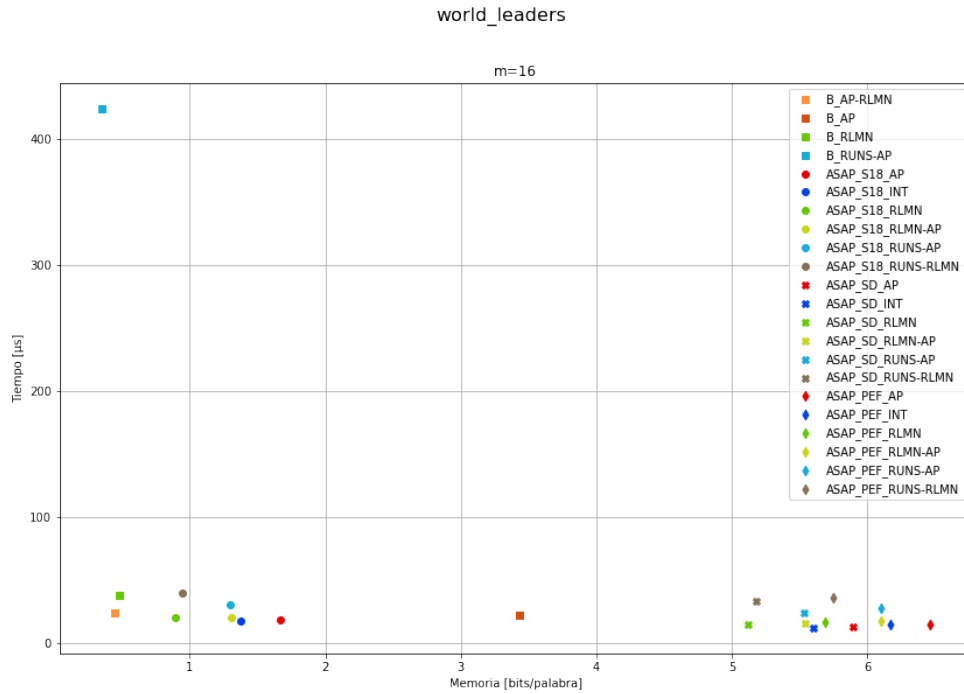


Figura 25: Resultados texto World Leaders $m = 16$
Fuente: Elaboración Propia.

De manera análoga a los textos anteriores, observando las Figuras 20,21 y 22, la versión con menos uso de memoria corresponde a $B_RUNS-AP$, pero posee los tiempos promedios mas altos con una gran diferencia.

En cuanto a $m = 4$ se tiene que la versión Baseline que posee el menor tiempo promedio es $B_AP-RLMN$ con $13.03[\mu s]$ utilizando $0.448[\text{bits/palabra}]$, sin considerar la versión B_AP dado su alto consumo de memoria, por otra parte, la versión AR-RLFM $ASAP_S18_RLMN$ posee un tiempo promedio de $7.416[\mu s]$, si comparamos con la versión Baseline $B_AP-RLMN$, se tiene que es un 43.1 % menor a cambio de $0.445 [\text{bits/palabra}]$ extra, diferencia de memoria que se mantiene para cada m .

Para $m = 8$ se tiene que la versión $B_AP-RLMN$ obtiene un tiempo promedio de $17.419[\mu s]$, en cuanto a la versión $ASAP_S18_RLMN$ se obtiene $12.089[\mu s]$, siendo un 30.6 % menor.

Por último, para $m = 16$ se tiene que la versión $B_AP-RLMN$ posee un tiempo promedio de $23.87[\mu s]$, en comparación a $ASAP_S18_RLMN$ con $19.976[\mu s]$, siendo un 16.3 % menor.

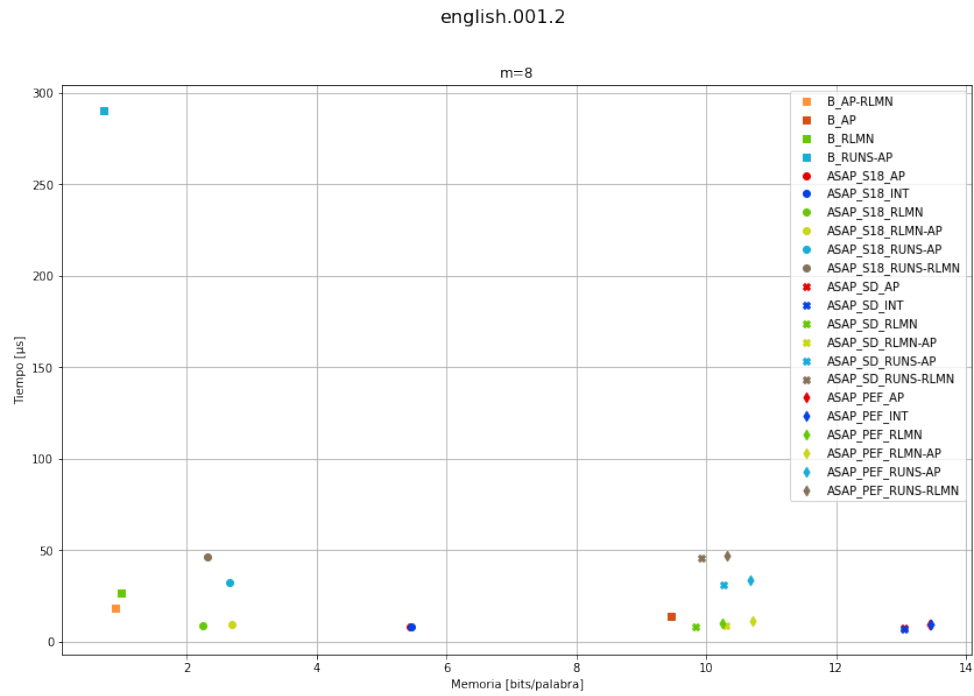


Figura 27: Resultados texto English.001 m = 8
Fuente: Elaboración Propia.

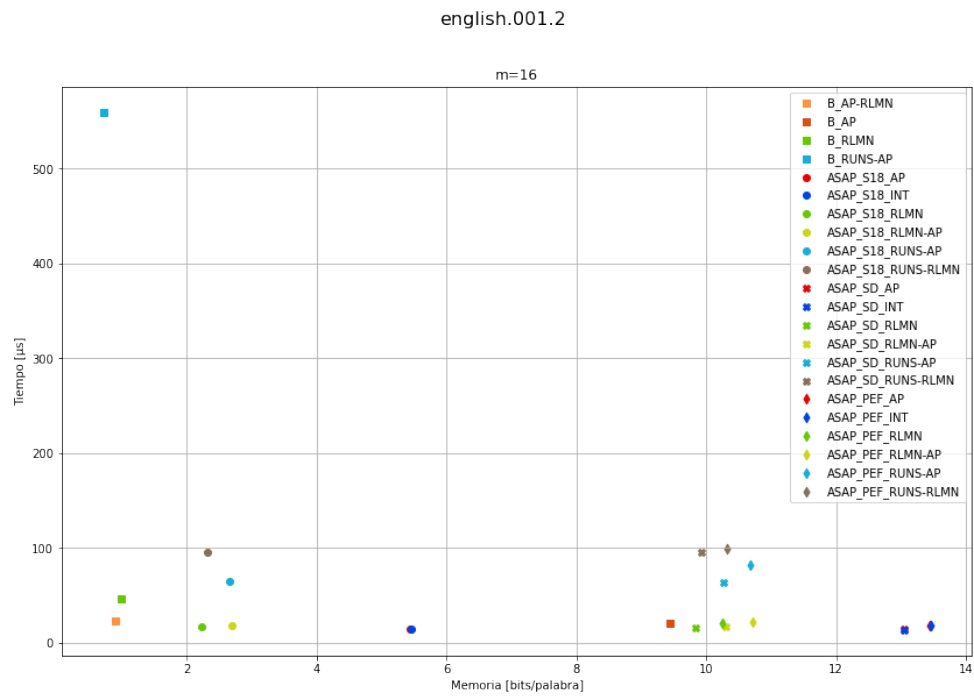


Figura 28: Resultados texto English.001 m = 16
Fuente: Elaboración Propia.

Observando las Figuras 26,27 y 28, nuevamente la versión con menos uso de memoria corresponde a $B_RUNS-AP$, a cambio de tiempos promedios mucho mayores altos.

Para $m = 4$ se tiene que la versión Baseline que posee el menor tiempo promedio es $B_AP-RLMN$ con $13.681[\mu s]$ utilizando $0.898[\text{bits/palabra}]$, sin considerar la versión B_AP dado su alto consumo de memoria. En cuanto a la versión $AR-RLFM_ASAP_S18_RLMN$, posee un tiempo promedio de $5.549[\mu s]$, siendo un 59.4 % menor con una diferencia de $1.342[\text{bits/palabras}]$.

Con $m = 8$ la versión Baseline $B_AP-RLMN$ posee un tiempo promedio de $18.701[\mu s]$, por otro lado, la versión $AR-RLFM_ASAP_S18_RLMN$, posee un tiempo promedio de $9.129[\mu s]$ disminuyendo un 51.2 %.

Finalmente, para $m = 16$ la versión Baseline $B_AP-RLMN$ tiene un tiempo promedio de $22.579[\mu s]$, en comparación a la versión $AR-RLFM_ASAP_S18_RLMN$ que obtuvo un tiempo promedio de $16.642[\mu s]$, lo cual es un 26.3 % menor.

CAPÍTULO 5

CONCLUSIONES

La estructura propuesta fue implementada con éxito, lográndose intervenir la estructura ASAP de manera tal que logre comprimir runs de símbolos iguales en sus substrings y vectores de bits de las particiones generadas. Además, se demuestra que nuestra solución posee mejores resultados en cuanto a tiempos de ejecución en el uso de la operación *Count*, comparando experimentalmente sus resultados con otras estructuras conocidas.

Al comparar entre las versiones AR-RLFM generadas, la versión AR-RLFM *ASAP_S18_RLMN* posee una gran diferencia en el uso de la memoria en relación con las versiones que no fueron implementadas con estructuras de datos con manejo eficiente de runs. Cabe destacar que en cada texto la versión que posee el menor tiempo promedio es, en su mayoría, la versión *ASAP_S18_INT*. Sin embargo, esto es a cambio de una gran cantidad de memoria extra, por tanto, se considera que la versión *ASAP_S18_RLMN* es la que obtuvo un mejor rendimiento en cada texto.

Comparando entre los bitvectors implementados para cada versión de AR-RLFM, para cada texto en los experimentos se obtuvo que las versiones de AR-RLFM generadas con S18 utilizaron una menor cantidad de memoria. Mientras que las versiones con SD vector y PEF utilizaron una cantidad similar de memoria. Las implementaciones que utilizaron PEF se vieron afectadas, ya que el contexto genera runs tanto de 1's como de 0's, y PEF funciona preferentemente con runs de 1's. Lo anterior, explica el porqué del alto consumo de memoria de la estructura.

En cuanto al uso de las estructuras para los substrings de cada partición generada por la estructura ASAP, se tiene que aquellas que poseen un manejo eficiente de runs obtuvieron mejor desempeño que aquellas que no. Esto se puede observar sin importar el tipo de bitvector a utilizar, dado que sucede tanto con el grupo de S18, PEF y SD vector. En estos, cada una de las versiones implementadas con un *WT_RLMN* y *RL_RUNS* logran utilizar una menor cantidad de memoria a cambio de aumentar el tiempo de búsqueda en cada texto.

Los resultados para la estructura AR-RLFM se vieron afectados según las características del texto. Al comparar aquellos textos que poseen runs de un largo promedio considerable, se obtiene un mejor rendimiento del uso de la memoria, específicamente en aquellas que contienen estructuras eficientes en el manejo de runs. Estas últimas fueron altamente competitivas al ser comparadas con los baselines propuestos.

No obstante, en los textos con runs que no tienen un largo considerable, se obtiene que aquellas estructuras que manipulan runs son afectadas. Lo anterior se puede observar en los textos *Coreutils*, *World Leaders* y *English*. En estos casos el uso de memoria es considerablemente mayor a pesar de no ser los textos más grandes. Esto se debe a que son aquellos que, poseen runs que no tienen un largo suficiente para permitir sacar provecho de las com-

ponentes implementadas en cada versión. Mientras que en los textos Einstein poseen runs de un largo tal que permite comprimir de mejor manera el texto a trabajar.

Por otra parte, los resultados también dependen del largo del patrón a buscar, si bien el uso de memoria se mantiene constante para cualquier largo, los tiempos de búsqueda son afectados a medida de que se va alargando el patrón. Esto se debe a que el desarrollo de la función *Count* aumentara el número de iteraciones que requiere utilizar la función *Rank*.

Si utilizamos la versión Baseline *B_AP-RLMN* como punto de comparación con la versión *AR-RLFM ASAP_S18_RLMN*, se genera un *trade off*, en el que se utiliza una pequeña cantidad extra de memoria. *AR-RLFM* logra disminuir de forma considerable el tiempo promedio, según el texto o el largo del patrón, esto puede variar como fue explicado anteriormente. No obstante, en todos los textos experimentados se dio que nuestra solución obtiene un tiempo promedio inferior a cambio de una cantidad extra de memoria.

A modo de conclusión, la solución presentada en esta investigación posee un resultado eficiente tanto en el uso de memoria como en el tiempo promedio. Esto se debe principalmente al uso de estructuras que generan y manejan runs. Por otra parte, a través de los experimentos se observa la importancia de conocer las variables o características de un texto al momento de comprimirlo y querer realizar operaciones de búsqueda. Sin embargo, .

En relación al trabajo que se proyecta para el futuro, el desarrollo y las conclusiones de esta investigación sugieren la continuación de la experimentación de esta solución en diversidades de textos. De esta manera se logra continuar con los aportes a la disciplina y a las mejores en tiempo y memoria.

ANEXOS

Tipo	Memoria [bits/palabra]
B_AP-RLMN	0.0594
B_AP	8.415
B_RLMN	0.0561
B_RUNS-AP	0.0385
ASAP_S18_AP	2.1395
ASAP_S18_INT	1.9434
ASAP_S18_RLMN	0.1086
ASAP_S18_RLMN-AP	0.2303
ASAP_S18_RUNS-AP	0.2291
ASAP_S18_RUNS-RLMN	0.1195
ASAP_SD_AP	10.5525
ASAP_SD_INT	10.3564
ASAP_SD_RLMN	8.5216
ASAP_SD_RLMN-AP	8.6433
ASAP_SD_RUNS-AP	8.6421
ASAP_SD_RUNS-RLMN	8.5325
ASAP_PEF_AP	10.5709
ASAP_PEF_INT	10.3748
ASAP_PEF_RLMN	8.54
ASAP_PEF_RLMN-AP	8.6618
ASAP_PEF_RUNS-AP	8.6605
ASAP_PEF_RUNS-RLMN	8.5509

Tabla 10: Memoria utilizada por cada una de las estructuras en el texto einstein.en.txt
Fuente: Elaboración Propia.

Versión	Tiempo [μ s]		
	m = 4	m = 8	m = 16
B_AP-RLMN	12.56	18.04	22.71
B_AP	11.11	14.18	20.81
B_RLMN	13.58	21.6	37.85
B_RUNS-AP	93.68	230.52	464.9
ASAP_S18_AP	5.06	8.2	14.99
ASAP_S18_INT	4.81	8.07	14.55
ASAP_S18_RLMN	5.32	9.2	17.16
ASAP_S18_RLMN-AP	5.74	10.0	18.2
ASAP_S18_RUNS-AP	10.71	20.58	39.65
ASAP_S18_RUNS-RLMN	15.64	30.96	69.59
ASAP_SD_AP	5.42	8.77	16.9
ASAP_SD_INT	4.94	8.58	16.62
ASAP_SD_RLMN	5.23	9.48	18.66
ASAP_SD_RLMN-AP	5.49	10.12	20.07
ASAP_SD_RUNS-AP	10.7	21.01	41.43
ASAP_SD_RUNS-RLMN	15.39	31.61	64.2
ASAP_PEF_AP	6.8	10.8	19.74
ASAP_PEF_INT	6.66	10.65	19.22
ASAP_PEF_RLMN	7.02	11.46	21.53
ASAP_PEF_RLMN-AP	7.27	12.09	22.66
ASAP_PEF_RUNS-AP	12.33	22.92	43.8
ASAP_PEF_RUNS-RLMN	16.86	33.21	86.08

Tabla 11: Tiempos promedios obtenidos por cada una de las estructuras para el texto einstein.en.txt

Fuente: Elaboración Propia.

Tipo	Memoria [bits/palabra]
B_AP-RLMN	0.1389
B_AP	8.6337
B_RLMN	0.1036
B_RUNS-AP	0.0741
ASAP_S18_AP	2.9751
ASAP_S18_INT	2.619
ASAP_S18_RLMN	0.2644
ASAP_S18_RLMN-AP	0.7514
ASAP_S18_RUNS-AP	0.7511
ASAP_S18_RUNS-RLMN	0.3041
ASAP_SD_AP	10.9753
ASAP_SD_INT	10.6192
ASAP_SD_RLMN	8.2646
ASAP_SD_RLMN-AP	8.7516
ASAP_SD_RUNS-AP	8.7513
ASAP_SD_RUNS-RLMN	8.3043
ASAP_PEF_AP	11.1729
ASAP_PEF_INT	10.8168
ASAP_PEF_RLMN	8.4621
ASAP_PEF_RLMN-AP	8.9492
ASAP_PEF_RUNS-AP	8.9488
ASAP_PEF_RUNS-RLMN	8.5018

Tabla 12: Memoria utilizada por cada una de las estructuras en el texto einstein.de.txt
Fuente: Elaboración Propia.

Versión	Tiempo [μ s]		
	m = 4	m = 8	m = 16
B_AP-RLMN	12.64	17.35	22.4
B_AP	10.03	13.13	19.06
B_RLMN	12.31	19.02	31.92
B_RUNS-AP	80.41	200.07	410.07
ASAP_S18_AP	4.48	7.52	13.85
ASAP_S18_INT	4.29	7.45	13.56
ASAP_S18_RLMN	4.84	8.64	16.31
ASAP_S18_RLMN-AP	5.26	9.27	17.36
ASAP_S18_RUNS-AP	9.16	16.68	32.35
ASAP_S18_RUNS-RLMN	14.06	23.62	46.18
ASAP_SD_AP	4.33	7.44	14.78
ASAP_SD_INT	4.54	7.48	14.57
ASAP_SD_RLMN	4.97	9.03	17.43
ASAP_SD_RLMN-AP	5.29	9.16	18.03
ASAP_SD_RUNS-AP	9.05	16.69	33.28
ASAP_SD_RUNS-RLMN	14.27	23.38	47.25
ASAP_PEF_AP	5.65	9.71	18.5
ASAP_PEF_INT	5.6	9.52	18.18
ASAP_PEF_RLMN	5.95	10.67	20.66
ASAP_PEF_RLMN-AP	6.33	10.99	21.32
ASAP_PEF_RUNS-AP	10.28	18.64	37.0
ASAP_PEF_RUNS-RLMN	14.91	25.09	49.88

Tabla 13: Tiempos promedios obtenidos por cada una de las estructuras para el texto einstein.de.txt

Fuente: Elaboración Propia.

Tipo	Memoria [bits/palabra]
B_AP-RLMN	1.1222
B_AP	8.7112
B_RLMN	1.4052
B_RUNS-AP	0.8716
ASAP_S18_AP	4.8043
ASAP_S18_INT	4.6721
ASAP_S18_RLMN	2.6184
ASAP_S18_RLMN-AP	2.8285
ASAP_S18_RUNS-AP	2.7839
ASAP_S18_RUNS-RLMN	2.6761
ASAP_SD_AP	11.4521
ASAP_SD_INT	11.3199
ASAP_SD_RLMN	9.2662
ASAP_SD_RLMN-AP	9.4763
ASAP_SD_RUNS-AP	9.4317
ASAP_SD_RUNS-RLMN	9.3239
ASAP_PEF_AP	12.1784
ASAP_PEF_INT	12.0461
ASAP_PEF_RLMN	9.9924
ASAP_PEF_RLMN-AP	10.2025
ASAP_PEF_RUNS-AP	10.1579
ASAP_PEF_RUNS-RLMN	10.0502

Tabla 14: Memoria utilizada por cada una de las estructuras en el texto coreutils
Fuente: Elaboración Propia.

Versión	Tiempo [μ s]		
	m = 4	m = 8	m = 16
B_AP-RLMN	14.37	19.98	25.51
B_AP	10.82	14.37	22.16
B_RLMN	17.11	29.3	52.26
B_RUNS-AP	125.37	331.22	606.84
ASAP_S18_AP	5.49	9.51	16.79
ASAP_S18_INT	5.39	9.3	16.43
ASAP_S18_RLMN	6.17	10.93	19.36
ASAP_S18_RLMN-AP	6.35	11.26	20.13
ASAP_S18_RUNS-AP	16.95	34.74	67.74
ASAP_S18_RUNS-RLMN	23.55	51.98	108.33
ASAP_SD_AP	5.05	8.54	16.51
ASAP_SD_INT	5.0	8.32	16.02
ASAP_SD_RLMN	5.29	9.71	18.35
ASAP_SD_RLMN-AP	6.1	10.58	19.77
ASAP_SD_RUNS-AP	16.1	33.27	73.42
ASAP_SD_RUNS-RLMN	23.48	51.05	103.78
ASAP_PEF_AP	6.33	9.69	17.78
ASAP_PEF_INT	6.4	10.02	17.7
ASAP_PEF_RLMN	7.09	11.49	20.7
ASAP_PEF_RLMN-AP	7.46	12.12	21.77
ASAP_PEF_RUNS-AP	17.61	35.3	86.71
ASAP_PEF_RUNS-RLMN	24.43	52.09	111.42

Tabla 15: Tiempos promedios obtenidos por cada una de las estructuras para el texto coreutils

Fuente: Elaboración Propia.

Tipo	Memoria [bits/palabra]
B_AP-RLMN	0.8985
B_AP	9.4502
B_RLMN	0.9896
B_RUNS-AP	0.7177
ASAP_S18_AP	5.443
ASAP_S18_INT	5.4516
ASAP_S18_RLMN	2.2403
ASAP_S18_RLMN-AP	2.704
ASAP_S18_RUNS-AP	2.6677
ASAP_S18_RUNS-RLMN	2.3227
ASAP_SD_AP	13.0453
ASAP_SD_INT	13.0539
ASAP_SD_RLMN	9.8426
ASAP_SD_RLMN-AP	10.3064
ASAP_SD_RUNS-AP	10.27
ASAP_SD_RUNS-RLMN	9.925
ASAP_PEF_AP	13.4477
ASAP_PEF_INT	13.4563
ASAP_PEF_RLMN	10.245
ASAP_PEF_RLMN-AP	10.7087
ASAP_PEF_RUNS-AP	10.6724
ASAP_PEF_RUNS-RLMN	10.3273

Tabla 16: Memoria utilizada por cada una de las estructuras en el texto english.001.2
Fuente: Elaboración Propia.

Versión	Tiempo [μ s]		
	m = 4	m = 8	m = 16
B_AP-RLMN	13.68	18.7	22.58
B_AP	10.68	14.1	20.67
B_RLMN	15.69	26.59	46.48
B_RUNS-AP	116.64	290.38	559.53
ASAP_S18_AP	4.96	7.9	14.31
ASAP_S18_INT	5.02	8.12	14.31
ASAP_S18_RLMN	5.55	9.13	16.64
ASAP_S18_RLMN-AP	5.99	9.77	18.0
ASAP_S18_RUNS-AP	16.64	32.12	64.53
ASAP_S18_RUNS-RLMN	23.85	46.58	95.13
ASAP_SD_AP	4.63	7.3	13.88
ASAP_SD_INT	4.69	7.22	13.53
ASAP_SD_RLMN	5.4	8.32	15.94
ASAP_SD_RLMN-AP	5.41	8.9	17.12
ASAP_SD_RUNS-AP	16.41	31.27	63.92
ASAP_SD_RUNS-RLMN	23.36	45.94	95.33
ASAP_PEF_AP	6.09	9.65	18.39
ASAP_PEF_INT	5.94	9.29	17.44
ASAP_PEF_RLMN	6.4	10.38	20.02
ASAP_PEF_RLMN-AP	7.03	11.44	21.98
ASAP_PEF_RUNS-AP	17.54	33.38	81.45
ASAP_PEF_RUNS-RLMN	24.0	46.97	98.42

Tabla 17: Tiempos promedios obtenidos por cada una de las estructuras para el texto english.001.2

Fuente: Elaboración Propia.

Tipo	Memoria [bits/palabra]
B_AP-RLMN	0.448
B_AP	3.4352
B_RLMN	0.4831
B_RUNS-AP	0.3479
ASAP_S18_AP	1.6652
ASAP_S18_INT	1.3727
ASAP_S18_RLMN	0.8932
ASAP_S18_RLMN-AP	1.31
ASAP_S18_RUNS-AP	1.3022
ASAP_S18_RUNS-RLMN	0.949
ASAP_SD_AP	5.8946
ASAP_SD_INT	5.6021
ASAP_SD_RLMN	5.1226
ASAP_SD_RLMN-AP	5.5394
ASAP_SD_RUNS-AP	5.5316
ASAP_SD_RUNS-RLMN	5.1784
ASAP_PEF_AP	6.4591
ASAP_PEF_INT	6.1666
ASAP_PEF_RLMN	5.6871
ASAP_PEF_RLMN-AP	6.1039
ASAP_PEF_RUNS-AP	6.0961
ASAP_PEF_RUNS-RLMN	5.7429

Tabla 18: Memoria utilizada por cada una de las estructuras en el texto world_leaders
Fuente: Elaboración Propia.

Versión	Tiempo [μ s]		
	m = 4	m = 8	m = 16
B_AP-RLMN	13.03	17.42	23.87
B_AP	13.18	18.24	21.74
B_RLMN	14.01	22.42	37.93
B_RUNS-AP	89.27	200.65	424.09
ASAP_S18_AP	6.35	10.62	17.68
ASAP_S18_INT	6.34	10.59	17.54
ASAP_S18_RLMN	7.42	12.09	19.98
ASAP_S18_RLMN-AP	7.53	12.43	20.34
ASAP_S18_RUNS-AP	13.76	18.8	29.76
ASAP_S18_RUNS-RLMN	18.37	24.71	39.19
ASAP_SD_AP	4.4	7.32	12.15
ASAP_SD_INT	4.44	7.32	12.01
ASAP_SD_RLMN	5.29	8.65	14.27
ASAP_SD_RLMN-AP	5.51	9.13	14.89
ASAP_SD_RUNS-AP	11.3	15.0	24.04
ASAP_SD_RUNS-RLMN	16.08	20.95	32.79
ASAP_PEF_AP	5.29	8.74	14.64
ASAP_PEF_INT	5.0	8.54	14.41
ASAP_PEF_RLMN	5.84	10.0	16.59
ASAP_PEF_RLMN-AP	6.41	10.46	17.38
ASAP_PEF_RUNS-AP	12.48	17.1	27.09
ASAP_PEF_RUNS-RLMN	16.79	22.4	35.77

Tabla 19: Tiempos promedios obtenidos por cada una de las estructuras para el texto world_leaders

Fuente: Elaboración Propia.

REFERENCIAS BIBLIOGRÁFICAS

- [Arroyuelo y Sepúlveda, 2019] Arroyuelo, D. y Sepúlveda, E. (2019). A practical alphabet-partitioning rank/select data structure. En Brisaboa, N. R. y Puglisi, S. J., editores, *String Processing and Information Retrieval - 26th International Symposium, SPIRE 2019, Segovia, Spain, October 7-9, 2019, Proceedings*, volumen 11811 de *Lecture Notes in Computer Science*, pp. 452–466. Springer.
- [Arroyuelo y Weitzman, 2020] Arroyuelo, D. y Weitzman, M. (2020). A hybrid compressed data structure supporting rank and select on bit sequences. pp. 1–8.
- [Barbay et al., 2014] Barbay, J., Claude, F., Gagie, T., Navarro, G., y Nekrich, Y. (2014). Efficient fully-compressed sequence representations. *Algorithmica*, 69(1):232–268.
- [Burrows y Wheeler, 1994] Burrows, M. y Wheeler, D. (1994). A block-sorting lossless data compression algorithm. En *Digital SRC Research Report*. Citeseer.
- [Carmona, 2022] Carmona, G. (2022). Pef library. <https://github.com/Yhatoh/PEF>.
- [Ferragina, 2005] Ferragina, Paolo y Manzini, G. (2005). Indización de texto comprimido.
- [Ferragina y Navarro, 2005] Ferragina, P. y Navarro, G. (2005). Pizza&chili corpus.
- [Fuentes, 2018] Fuentes, J. L. (2018). Rl runs library. <https://github.com/jfuentess/sdsl-lite>.
- [Fuentes-Sepúlveda et al., 2018] Fuentes-Sepúlveda, J., Kärkkäinen, J., Kosolobov, D., y Puglisi, S. (2018). Run compressed rank/select for large alphabets. En *2018 Data Compression Conference*, pp. 315–324.
- [Gog et al, 2019] Gog et al (2019). Succinct data structure library. <https://github.com/simongog/sdsl-lite>.
- [Larrañaga, 2022] Larrañaga, H. (2022). S18 library. <https://github.com/HectorxH/Modificacion-S18>.
- [Navarro, 2014] Navarro, G. (2014). Wavelet trees for all. *Journal of Discrete Algorithms*, 25:2–20.
- [Ottaviano y Venturini, 2014] Ottaviano, G. y Venturini, R. (2014). Partitioned elias-fano indexes. En *Proceedings of the 37th international ACM SIGIR conference on Research & development in information retrieval*, pp. 273–282.
- [Stopak, 2019] Stopak, J. (2019). Initial commit.