UNIVERSIDAD DE CONCEPCIÓN
DEPARTAMENTO INGENIERÍA INFORMÁTICA Y
CIENCIAS DE LA COMPUTACION
FACULTAD DE INGENIERÍA

CALCULO EFICIENTE EN ESPACIO DE LA TRANSFORMADA DE BURROWS-WHEELER

Por

Agustín Felipe Peña Peñaranda

Memoria presentada para la obtención del título de
Ingeniero Civil Informático

Abril 2022
Concepción, Chile

Profesor Guía: José Fuentes Sepúlveda

© 2022, Agustin Felipe Peña Peñaranda
Se autoriza la reproducción total o parcial, con fines académicos, por cualquier medio o procedimiento, incluyendo la cita bibliográfica del documento

AGRADECIMIENTOS

A mis padres por el apoyo que me han dado.
Al profesor Roberto, por mostrarme la belleza de la informatica.
A Nicolas y Carla, por acompañarme durante este proceso.

Resumen

La compresión es uno de los principales retos de la informática moderna. Es crucial ser eficiente en el espacio, no sólo en el resultado sino también en el proceso de compresión. La transformada Burrows-Wheeler (BWT) es una herramienta esencial para esta tarea, pero el cálculo de esta transformada no es trivial y puede consumir mucho espacio. Uno de los algoritmos del estado del arte para calcular la BWT es el de Okanohara y Sadakane que utiliza la ordenación inducida. Este algoritmo calcula la BWT para su posterior compresión. Tomando como base el trabajo de Okanohara y Sadakane, se desarrolló el presente trabajo, donde se comprime la BWT durante su construcción, reduciendo así el peak de espacio utilizado en este proceso.

Índice general

Índice de cuadros

Índice de figuras

Capítulo 1

Introducción

Uno de los principales problemas de la informática moderna es el manejo de grandes volúmenes de datos. A su vez, una de las principales maneras de tratar este problema es la compresión, la cual busca representar en espacio reducido grandes colecciones de datos. Es en este contexto en el que aparece la *Transformada de Burrows-Wheeler (BWT)* [2], la cual es parte esencial en algoritmos de compresión como Bzip2 [10]. Esta memoria de título se centrará en el cálculo eficiente en espacio de la BWT

La BWT es una herramienta esencial para la compresión de datos. Es excepcionalmente eficaz para textos muy repetitivos, como los repositorios de códigos o la secuenciación genómica de individuos de una misma especie. Esta transformación no sólo ayuda a la compresión los datos, sino que también permite realizar búsquedas sin necesidad de revertir la compresión como es en el caso de FM-index[3].

Okanohara y Sadakane [9] introdujeron un algoritmo del estado del arte que utiliza la ordenación inducida para calcular el BTW. Su algoritmo calcula el BTW con una complejidad de temporal $O(n)$ y complejidad espacial de $O(n \log \sigma \log \log_\sigma n)$. En este trabajo buscaremos reducir el peak de espacio utilizado por el algoritmo de Okanohara y Sadakane, aplicando la técnica de compresión run-length [6] durante el cálculo de la BWT.

Capítulo 2

Marco Teórico

2.1. Transformada de Burrows-Wheeler

La transformada Burrows-Wheeler (BWT) fue inventada en 1994 por Michael Burrows y David Wheeler en su artículo "A Block-Sorting Lossless Data Compression Algorithm" [2]. Dada una secuencia $S[1..n]$ y un arreglo de sufijos SA[1..n] de S [7], la BWT $B[1..n]$ de S se define como $B[i] = S[SA[i] - 1]$, siendo $S[0] = S[n]$. Obsérvese que, en un alfabeto de tamaño σ, tanto S como B pueden almacenarse en $n \lg \sigma$ bits (lg es el logaritmo con base 2 por defecto), mientras que SA requiere $n \lg n$ bits en formato plano. Los índices de texto compactos utilizan B en sustitución de S y SA, lo que reduce considerablemente los requisitos de espacio.

La transformación permuta el orden de la secuencia S. Si la cadena original contiene muchas subcadenas que aparecen a menudo, entonces la cadena transformada contendrá posiciones consecutivas en las que un mismo carácter estará repetido varias veces. Esto es útil para la compresión, ya que tiende a ser fácil comprimir una cadena que contiene secuencias de caracteres repetidos con técnicas como move-to-front transform y run-length encoding [6].

La idea principal de esta transformada es ordenar lexicográficamente todos los sufijos de S, guardando un caracter especial \$ el que se usa como notación para representar el final del texto y permite la recuperación del texto original. Por definición, \$ es menor que todos los símbolos de la cadena S.

El cómputo de esta se puede ejemplificar a partir de rotaciones del texto y luego se ordenan estas rotaciones, obteniendo el BWT como último carácter de cada rotación.

La Figura 2.1.1 muestra un ejemplo de la palabra mississippi$, donde su BWT es ipssm$pissii.

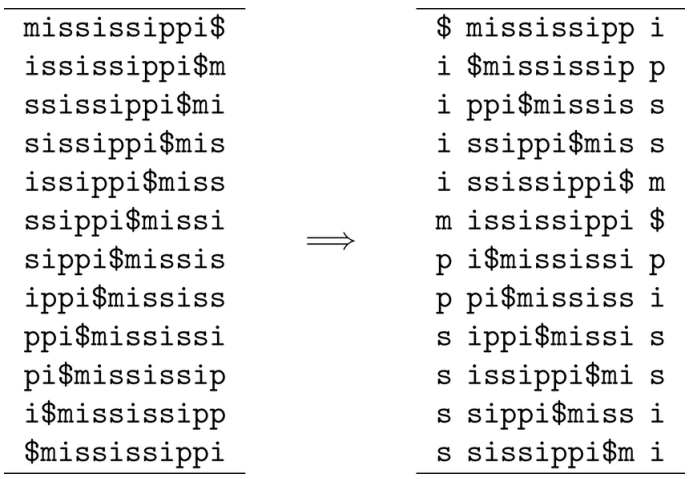

Figura 2.1.1: Construcción por rotación

2.2. Codificación Run-length

La codificación RLE o Run-length encoding [6] es una forma muy simple de compresión de datos en la que subsecuencias conformadas de mismo valor repetido son almacenadas como un único valor más su recuento. Esto es más útil en datos que contienen muchas de estas subsecuencias; por ejemplo, gráficos sencillos con áreas de color plano, como íconos, logotipos y la BWT de textos altamente repetitivos.
Un ejemplo de esta codificación es $10B1N8B3N12B2N$, la cual representa a la secuencia $BBBBBBBBBBNBBBBBBBBBNNNBBBBBBBBBBBBNN$.

Otro ejemplo es la BWT de un texto repetitivo. Por ejemplo, considerando el texto T: CCGTTTCTAACGCCCGTTTCTAACGCCCGTTTCTAACGCCCGTTTCTAA$ obtenemos la siguiente BWT y su correspondiente compresión RLE:

BWT: AATTTTAAAGGGCCC$AAACCCCTTTTCCCCCCCCCCCTTTTTTTTGGGG
RLE: 2A4T3A3G3C1$3A4C4T11C8T4G

2.3. Ordenamiento Inducido

El ordenamiento inducido [8] es un ordenamiento estable que tiene una complejidad temporal de $O(n)$.

El ordenamiento inducido funciona separando la entrada en bloques, ordenando estos bloques para luego utilizar esa información e inducir el orden de los elementos anteriores a cada bloque y estos elementos inducidos para inducir los elementos anteriores a ellos, ordenando en base a información ya computada.

La idea principal es clasificar los elementos del texto para encontrar los mínimos locales dentro del mismo, creando así los bloques o subcadenas. Para encontrar estos bloques, se clasifican los caracteres ,con una lectura inversa del texto, en tipo L, S y S*, donde este último representa un mínimo local. Para la clasificación se sigue el siguiente criterio, donde T_i representa al sufijo de T que comienza en la posición i.

$$\begin{aligned} L = &\quad \{i \in [0,n], \quad T_i > T_{i+1}\} \\ S = &\quad \{i \in [0,n], \quad T_i < T_{i+1}\} \\ S^* = &\quad \{i \in S, \quad i-1 \in L\} \end{aligned}$$

Por ejemplo, la clasificación de la palabra mmissssiippii$ se puede apreciar en la Figura 2.3.1.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
T =	m	m	i	s	s	i	s	s	i	i	p	p	i	i	$
type	L	L	S*	L	L	S*	L	L	S*	S	L	L	L	L	S*

Figura 2.3.1: Clasificación de los sufijos de la palabra mmissssiippii$ en L, S y S*

Un bloque o palabra S* va de un carácter S* al siguiente, incluyendo ambos caracteres S*. Las palabras S* de nuestro ejemplo serían las siguientes, donde el caracter $ representa el fin del texto, siendo el caracter con el valor mas pequeño.

[issi] [issi] [iippii$] [$mmi]

A continuación se ordenan estos bloques por orden lexicográfico, considerando

cada bloque como un elemento. Esto permite un ordenamiento general con muchas menos comparaciones, siendo el siguiente el orden del ejemplo

[$mmi] [iippii$] [issi] [issi]

Al hacer esto, tendremos el orden relativo de todos los bloques S*, los que se utilizarán para inducir la posición de los caracteres L.

Teniendo los caracteres L con su posición ya inducida, se utiliza esta información para inducir la posición de los caracteres S; dentro de los caracteres S, se consideran los S*.

Una vez que tenemos todas las posiciones establecidas, se obtiene la secuencia ordenada.

El algoritmo se aplica de manera recursiva, considerando cada palabra S* como un elemento y representando a estos como una nueva secuencia. El caso base corresponde a una secuencia donde todos sus símbolos son distintos. Esta idea está basada en un algoritmo llamado suffix array induced sorting (SAIS), el cual es utilizado para calcular el arreglo de sufijos de una secuencia.

2.4. Algoritmo de Okanohara y Sadakane

El algoritmo de Okanohara y Sadakane [9] es uno de los algoritmos más competitivos para el cálculo de la BWT, el cual ofrece la mejor relación entre espacio y tiempo, según el estudio realizado en “Parallel computation of the Burrows Wheeler Transform in compact space”[5]. Este algoritmo es representado como **os** en la Figura 2.4.1, donde se aprecia que este algoritmo es el que obtiene el mejor equilibrio entre el tiempo de ejecución y la utilización del espacio, siendo el más cercano al punto 0,0 de la figura.

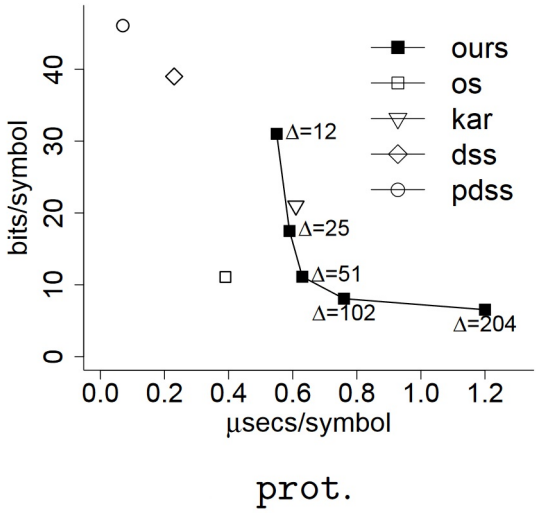

Figura 2.4.1: Tiempo y uso de memoria en la construcción de la BWT para una secuencia de proteínas. El algoritmo de Okanohara y Sadakane es representado como **os**. La figura fue obtenida de "Parallel computation of the Burrows Wheeler Transform in compact space"[5].

El algoritmo propuesto calcula la BWT utilizando la técnica de ordenación inducida modificada. La modificación realizada es que la función de comparación al hacer la ordenación inicial de las palabras S* se modifica para generar una BWT de estas palabras S*.

Después de hacer esta modificación, el ordenamiento inducido continúa en su forma tradicional, siendo el resultado final el BWT inducido.

El algoritmo de Okanohara y Sadakane no considera el cálculo de la BWT para textos altamente repetitivos. Por lo tanto, puede ser modificado para que use espacio proporcional a la BWT representada con run-length.

2.4.1. Algoritmos

El algoritmo diseñado por Okanohara y Sadakane se divide en 2 partes, el algoritmo 1 y el algoritmo 2.

2.4.1.1. Algoritmo 1

Este algoritmo clasifica el texto según lo requerido para hacer el ordenamiento inducido. Con esa clasificación se separa el texto en substrings $S*$, para luego

```
Input : T[1, n] : texto de entrada
n     : tamaño de entrada
k     : tamaño del alfabeto
1  Escanear T para clasificar todos los caracteres de T como de tipo S , S* o L
2  Descomponer T en substrings R[1, ...n'] (R[i] ∈ {1, ..., k}*)
3  Nombra los substrings S* utilizando el resultado de Induce(R), y obtiene un nuevo
     string acortado T1[1, n'], T1[i] ∈ {1, ..., k'}
4  if Cada caracter de T1 es único then
5  |   Calcula directamente B1 de T1
6  else
7  |   B1 ← Algorithm1(T1, n', k') // llamada recursiva
8  |
9  Decodificar S* cadenas de B1 en R'[1, ..., n']
10 B ← Induce(R')// Algoritmo 2
11
12 return B
```

Algoritmo 1: Cálculo de la BWT de Okanohara y Sadakane [9].

crear un arreglo de sufijos de estos substrings. Este arreglo de sufijos es procesado para obtener la BWT de los substring S*.

El cálculo de la BWT del arreglo de sufijos se realiza usando SAIS el cual funciona de forma recursiva, como fue mencionado anteriormente. Al final de este algoritmo se llama al algoritmo 2, el cual recibe el arreglo de substring S* ordenado según la BWT.

2.4.1.2. Algoritmo 2

El algoritmo 2 transforma la BWT parcial, que recibe en forma de substrings S*, a la BWT del texto original. Esto lo realiza mediante 3 ciclos for principales.

El primero, que va de la línea 1 a la 6, tiene la función de pasar la información recibida al formato que se va a utilizar, manteniendo los strings en queues, guardando un sentinela al final de cada string para señalar el final de estos, así como ordenarlos según su último elemento, el cual luego de una rotación pasa a ser el primero. Esto último es de suma importancia, ya que es ese primer elemento que nos permitirá hacer las primeras inducciones.

En el segundo ciclo for, que comienza en la línea 7 y termina en la línea 21, se utiliza la información anterior para poder inducir el orden de los caracteres tipo L. Ya que en el ciclo anterior se invirtieron los substrings S*, esta inducción se realiza desde el final de cada substring, hasta el principio, permitiendo así que

T_i utilize la información que le proporciona T_{i+1} para inducir su posición. Este ciclo induce la posición de los caracteres L en base a los caracteres S*, para luego inducir los L restantes de los L ya inducidos.

El tercer ciclo for, que comienza en la línea 23 y termina en la línea 38, realiza un proceso parecido al ciclo for anterior, donde se inducen los caracteres tipo S en base a los caracteres tipo L, para luego inducir los caracteres tipo S restantes en base a los ya inducidos. Los caracteres S* son parte de los tipo S por lo que también son inducidos en este for.

Finalmente, el ciclo for que comienza en la línea 39 invierte el resultado del ciclo anterior, para luego ir concatenando el resultado de la inducción tipo L y la inducción tipo S, donde el resultado sera la BWT del texto original.

2.4.2. Dificultades

La documentación realizada en el trabajo de Okanohara y Sadakane es imprecisa y no se corresponde totalmente con el código implementado. El código implementado no utilizaba nombres de variables autodescriptivas. Esto implicó un esfuerzo más importante para entender el funcionamiento de su implementación y hacer coincidir este con lo dicho en el paper publicado. Los comentarios del código estaban en japonés, lo que dificultó la lectura de estos, ya que los editores de texto no los interpretan como caracteres en japonés.

Input : $R[1, n']$: Lista de substrings S^*.

```
1  for i ← 1 to n' do
2  |  U ← Reverse(R_i)
3  |  U.push_back(k + 1) // centinela
4  |  c ← U.pop_front()
5  |  S*_c.push_back(U)
6  end
7  for i ← 1 to k do
8  |  while U ← L_i.pop_front() do
9  |  |  c ← U.pop_front()
10 |  |  if c < i then LS_i.push_back(c + U)
11 |  |  else
12 |  |  |  BL_i.push_back(c)
13 |  |  |  L_c.push_back(U)
14 |  end
15 |  LS_i ← Reverse(LS_i)
16 |  while U ← S*_i.pop_front() do
17 |  |  c ← U.pop_front()
18 |  |  E.push_back(c)
19 |  |  L_c.push_back(U)
20 |  end
21 end
22 E ← Reverse(E)
23 for i ← k to 1 do
24 |  while U ← S_i.pop_front() do
25 |  |  c ← U.pop_front()
26 |  |  if c < i then
27 |  |  |  BS_i.push_back(c)
28 |  |  |  S_c.push_back(U)
29 |  |  else
30 |  |  |  c2 ← E_c.pop_front() // alcanza el centinela
31 |  |  |  BS_i.push_back(c2)
32 |  end
33 |  while U ← LS_i.pop_front() do
34 |  |  c ← U.pop_front()
35 |  |  BS_i.push_back(c)
36 |  |  S_c.push_back(U)
37 |  end
38 end
39 for i ← 1 to k do
40 |  BS_i ← Reverse(BS_i)
41 |  B ← B + BL_i + BS_i
42 end
43 return B
```

Algoritmo 2: Induce(R): Algoritmo para inducir BWT a partir de las subcadenas S^* de Okanohara y Sadakane [9].

Capítulo 3

Metodología

3.1. Comprimir durante el procesamiento

La BWT se utiliza principalmente para la compresión de datos, por lo que el uso de espacio durante el procesamiento es primordial. El peak de espacio utilizado corresponde a la ejecución del algoritmo 2 de Okanahara y Sadakane, donde gran parte de este espacio es utilizado para almacenar la BWT.

Las cualidades de la BWT permiten su compresión, en especial para textos repetitivos. La idea es que durante el procesamiento se almacene la BWT en su forma comprimida, ahorrando así espacio en el punto donde se encuentra el peak de memoria. Esto permite a su vez poder obtener directamente la versión comprimida de la BWT, como también calcular la BWT en su versión original, pero reduciendo el peak de memoria utilizado durante el procesamiento.

La compresión de la BWT está basada en la codificación Run-length. Para lograr la compresión durante el procesamiento se implementó una estructura de datos compacta donde se almacenará la BWT. Esta estructura es una queue modificada, la cual remplazaría a las queue utilizadas en el algoritmo 2, estas son BL que aparece en la línea 12, BS que es mayormente utilizada en el ciclo for de la línea 23 y la queue E.

3.2. Queue Modificada

Se elegió una modificación de una queue que está implementada con una lista enlazada unidireccional, como se muestra en la Figura 3.2.1, porque cumple con las características necesarias para el adecuado funcionamiento del algoritmo. Las características que se necesitan son tener un comportamiento FIFO, es decir el primero en entrar es el primero en salir, ser eficiente con el uso de espacio y permitir la operación *Reverse*. La idea de esta lista enlazada es que tenga las funciones que tendría una queue normal. Estas son *Enqueue*, *Dequeue*, *Top*, *Reverse*, *isEmpty*, *Size* aparte de un constructor y un destructor.

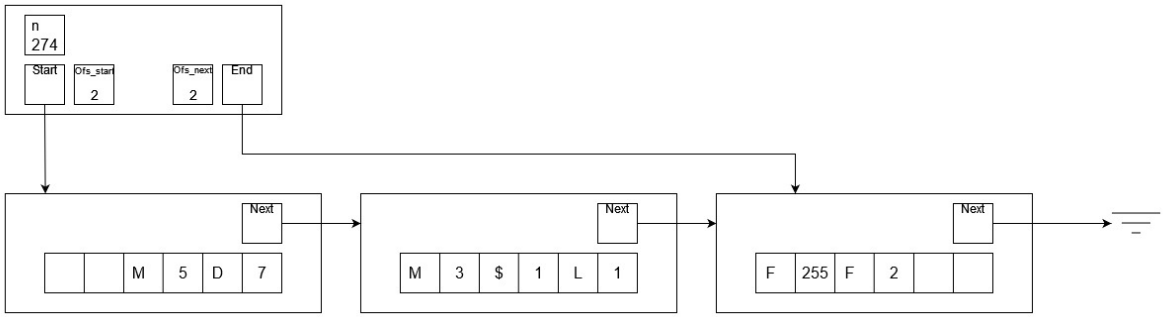

Figura 3.2.1: Diagrama queue modificada.Para texto en formato RLE, M5D7M3$1L1F257

Para un óptimo funcionamiento de esta estructura, se tienen 2 subestructuras, la cabeza de la lista, elemento superior de la Figura 3.2.1, y los nodos de la lista, elementos inferiores de la Figura 3.2.1. En la cabeza de la lista se guardará la cantidad de elementos ingresados (n), un puntero al nodo inicial ($Start$)y al nodo final (End), además de 2 valores de offset de desplazamiento, uno en relación al primero nodo (Ofs_start) y el otro en relación al último nodo (Ofs_end). Como vemos en la Figura 3.2.1, Ofs_start indica que el primer elemento del nodo inicial está en la 2da posición y Ofs_end que indica que el último elemento del último nodo está en la segunda posición.

Cada nodo tiene un puntero al nodo siguiente y un arreglo de caracteres donde se almacenará la información del nodo, como se ve en los elementos inferiores de la Figura 3.2.1. Es importante que la estructura del nodo sea lo más simple posible, ya que un pequeño aumento en su tamaño tendría un gran impacto en la memoria utilizada por el algoritmo en general. En el arreglo de caracteres no se

guardan solamente caracteres si no también enteros de 1 byte, donde los elementos que estén en una posición par serán caracteres y el siguiente elemento, el cual corresponde a un índice impar, contendrá un contador de repeticiones, esto para que la estructura del nodo sea simple y eficiente con el espacio que utiliza.

El crecimiento de esta lista enlazada tiene un comportamiento semi-dinámico. Si la lista tuviera un comportamiento donde crea un nodo nuevo por cada elemento, el espacio ocupado por los punteros sería mayor que el tamaño total del texto, por lo que no tendría sentido usarlo en este caso. También se descartó el uso de una lista dinámica, la cual comienza con un tamaño fijo y una vez que se llena se dobla el espacio de esta. Esto podría dar el caso de que el último elemento ingresado sea el que genera que se doble el espacio de la lista, utilizando el doble del espacio necesario.

El comportamiento semi-dinámico consiste en que el nodo contiene un arreglo de un tamaño fijo, evitando de esta forma tener un puntero por cada elemento, una vez que se llena un nodo se crea otro, lo que soluciona el problema que generaría un arreglo dinámico.

La compresión realizada en esta estructura es del tipo RLE, cuando ingresa un elemento se revisa si el último elemento ingresado es igual al que se está ingresando. De ser ese el caso, simplemente se aumenta en 1 el contador de caracteres consecutivos. En caso contrario, se agrega como un nuevo elemento teniendo este el contador en 1. Debido a que el contador usa 1 byte puede llegar hasta 255. Cuando se tiene una secuencia de caracteres iguales más larga que 255, el próximo elemento ingresado se procesará como si los caracteres sean distintos, a pesar de no ser así como vemos en la figura 3.2.1 en el tercer nodo. Esto modifica la forma de la RLE pero es solucionado cuando se implementa la operación dequeue, simulando como si fuera un solo par.

Capítulo 4

Experimentación

Para la experimentación, compararemos el peak de consumo de memoria utilizado para construir el BWT de ambos algoritmos, es decir, del algoritmo de Okanohara y Sadakane con el propuesto en esta memoria de título.

También se comparará el peak de memoria utilizado en las diferentes etapas de procesamiento para identificar más claramente las diferencias de ambos algoritmos.

4.1. Estudio experimental

Para medir el espacio utilizado se utiliza malloc_count [1], este nos permite medir el pico de espacio de forma general así como por secciones. Esto se logra generando un bypass en la funcion malloc y en la funcion free para poder llevar un registro del espacio utilizado.

Las secciones a medir son la lectura del texto (**Lectura**), la clasificación de los sufijos del texto en tipo S, S* y L (**Clasificar**), la ejecución del algoritmo SAIS (**SAIS**), hasta el punto en que llama al algoritmo 2 (**Post-size**). Dentro del algoritmo 2 se medirá el rendimiento de cada uno de los tres bucles for (**S*, L, S**), el tamaño de la estructura que almacena el resultado final (**BWT-size**) y el espacio ocupado al teminar el algoritmo (**Final**).

Aunque el objetivo principal es medir la eficiencia del espacio, la velocidad sigue siendo esencial. Para realizar la medición del tiempo, se utilizó la biblioteca time.h del lenguaje C, pidiendo el tiempo antes y después de la ejecución del

programa. Durante esta ejecución se omite la medición del espacio para no afectar al resultado. Por cada experimento se realizaron 30 repeticiones para luego promediar el resultado, obteniendo así un resultado más preciso.

El compilador utilizado es gcc (Debian 10.2.1-6) 10.2.1 20210110, juntos a los siguientes flags de compilacion: -m64 -D_LARGEFILE64_SOURCE=1 -D_FILE_OFFSET_BITS=64 -O9 -ldl

Los experimentos se llevaron a cabo en un servidor NUMA (Non-Uniform Memory Access) con dos nodos. Cada nodo cuenta con un procesador Intel® Xeon® Gold 5320T, con 20 cores físicos de 2.3GHz. El servidor corre Linux 5.10.0-13-amd64 con compatibilidad de 64 bits. Por cada core, la máquina cuenta con una caché L1 de datos, L1 de instrucciones y L2 de 48KB, 32KB y 1.25MB, respectivamente. La caché L3 es compartida por los 20 cores de un nodo, con un tamaño de 30MB. Finalmente, la memoria RAM disponible es de 252GB (126GB por cada nodo). El servidor contaba con el modo Hyperthreading habilitado, lo que da un total de 80 cores lógicos de ejecución. Sólo se ejecutará un programa a la vez, evitando así que los programas que se ejecutan en segundo plano afecten a las mediciones.

4.2. Datasets a utilizar

El datasets que se utilizó para textos no repetitivos[1] es Pizza&Chili [4]. En este repositorio se encuentran diferentes archivos de texto, con diferentes características, como el tamaño del texto, el tamaño del alfabeto y la naturaleza del texto. Estos son:

[1] http://pizzachili.dcc.uchile.cl/texts.html

Archivo	Tamaño del alfabeto	Tamaño del archivo(B)
SOURCES	230	210.866.607
PITCHES	133	55.832.855
PROTEINS	27	1.184.051.855
DNA	16	403.927.746
ENGLISH	239	2.210.395.553
XML	97	294.724.056

Cuadro 4.2.1: Textos no repetitivos

Se tendrá un segundo dataset que consiste en textos repetitivos[2], se espera que la BWT final comprimida con RLE sea más corta, ya que estos son sumamente compresibles.

Archivo	Tamaño del alfabeto	Tamaño del archivo(B)
CERE	5	461.286.644
COREUTILS	236	205.281.778
EINSTEIN.EN.TXT	139	467.626.544
ESCHERICHIA_COLI	15	112.689.515
INFLUENZA	15	154.808.555
KERNEL	160	257.961.616
PARA	5	429.265.758
WORLD_LEADERS	89	46.968.181

Cuadro 4.2.2: Textos repetitivos

Como último experimento se utilizó el mismo archivo pero este estará truncado para obtener archivos de distinto tamaño, para así poder ver el comportamiento del algoritmo donde una variable que cambia es el tamaño del archivo.

Esto es realizó con 2 textos: uno es un texto en inglés, el cual no es repetitivo, y el historial de las ultimas 1000 versiones de las 200 páginas más grandes de Wikipedia, el cual es altamente repetitivo.

[2]http://pizzachili.dcc.uchile.cl/repcorpus/real/

Archivo	Tamaño del alfabeto	Tamaño del archivo(B)
50Wiki.xml	98	50.000.000
100Wiki.xml	98	100.000.000
200Wiki.xml	98	200.000.000
500Wiki.xml	98	500.000.000
10000Wiki.xml	98	1.000.000.000
22000Wiki.xml	98	2.210.595.110

Cuadro 4.2.3: Textos de Wikipedia

Archivo	Tamaño del alfabeto	Tamaño del archivo(B)
eng.50MB	239	52.428.800
eng.100MB	239	104.857.600
eng.200MB	239	209.715.200
eng.1024MB	239	1.073.741.824
eng.2200MB	239	2.210.395.553

Cuadro 4.2.4: Textos en ingles

4.3. Resultados y análisis

En los apéndices A1-A4 se muestran tablas con un mayor nivel de detalle de los resultados obtenidos.

En esta sección nos vamos a referir al algoritmo de Okanohara y Sadakane [9] como **O&S**, mientras que el trabajo de esta memoria será referido como **MT**

4.3.1. Textos no repetitivos

4.3.1.1. Resultados

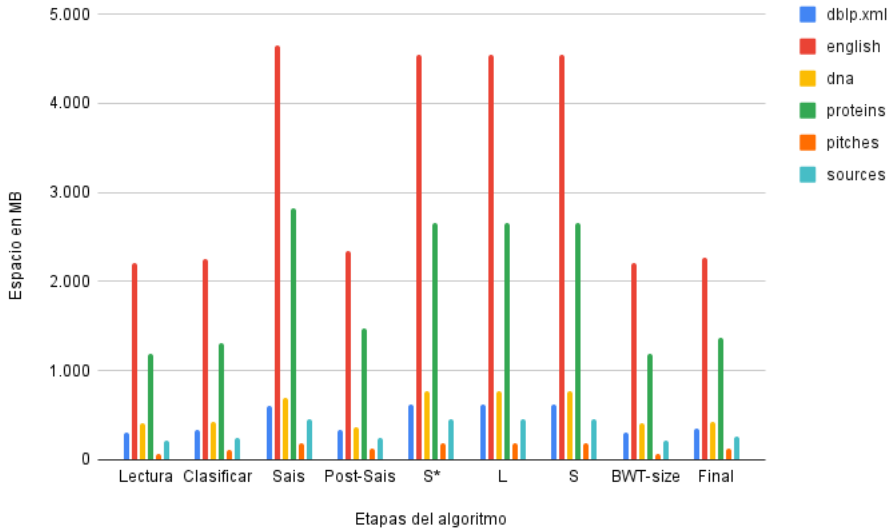

Figura 4.3.1: Uso de memoria O&S para textos no repetitivos

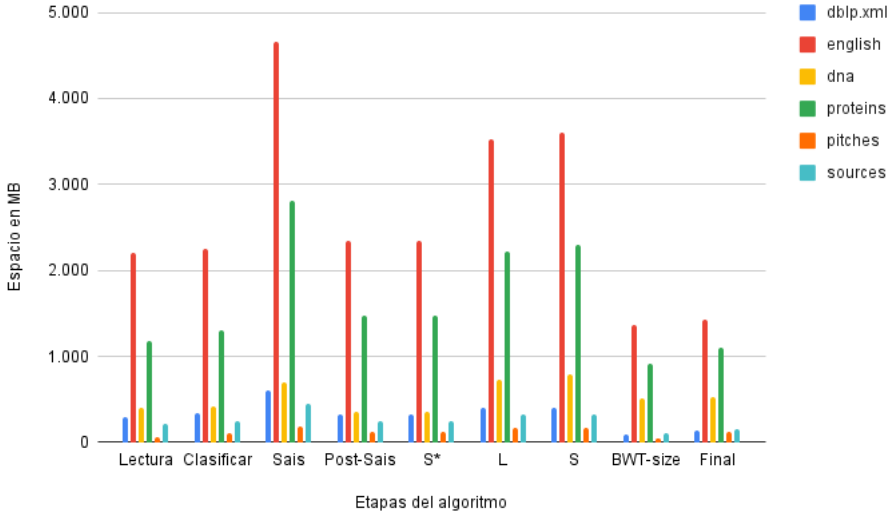

Figura 4.3.2: Uso de memoria MT para textos no repetitivos

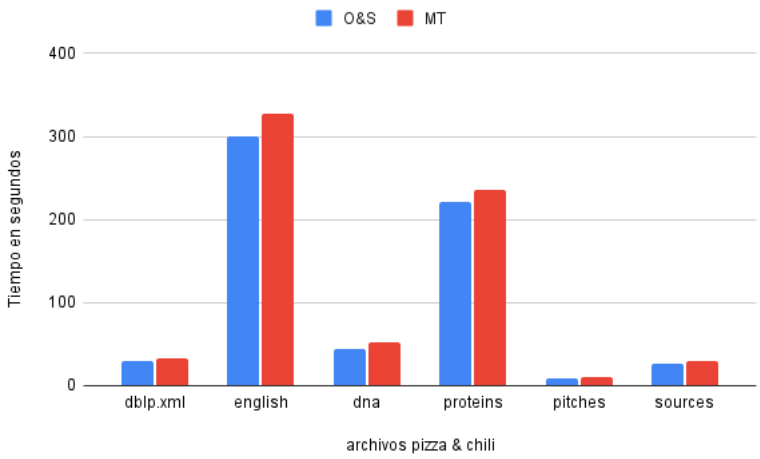

Figura 4.3.3: Tiempo de ejecución para textos no repetitivos

4.3.1.2. Análisis

Recordemos que la colección de textos de esta prueba contiene textos de naturaleza distinta, variando el tamaño de los textos y del alfabeto.

Podemos ver que el peak de memoria para el algoritmo O&S (ver Figura 4.3.1), es el mismo que el de MT (ver Figura 4.3.2), siendo el punto donde más se usa memoria la ejecución de SAIS. Por otro lado, podemos ver una disminución considerable en el uso de memoria en las siguientes etapas del algoritmo para MT.

En la figura 4.3.3 podemos ver que MT se demora un 9 % más que O&S.

4.3.2. Textos repetitivos

4.3.2.1. Resultados

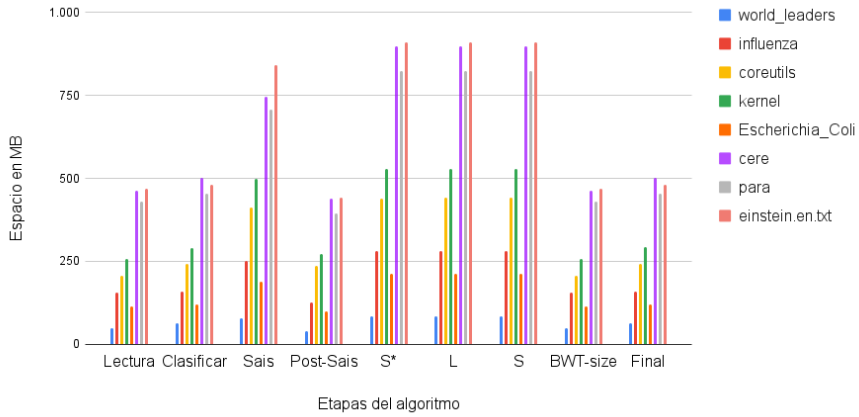

Figura 4.3.4: Uso de memoria para textos repetitivos O&S

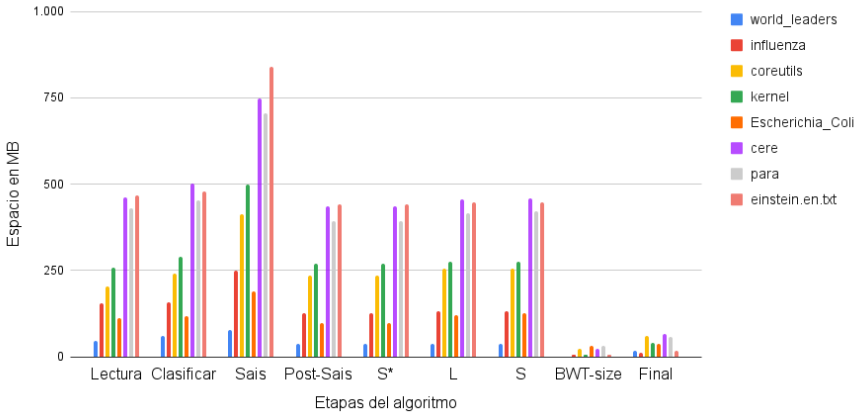

Figura 4.3.5: Uso de memoria para textos repetitivos MT

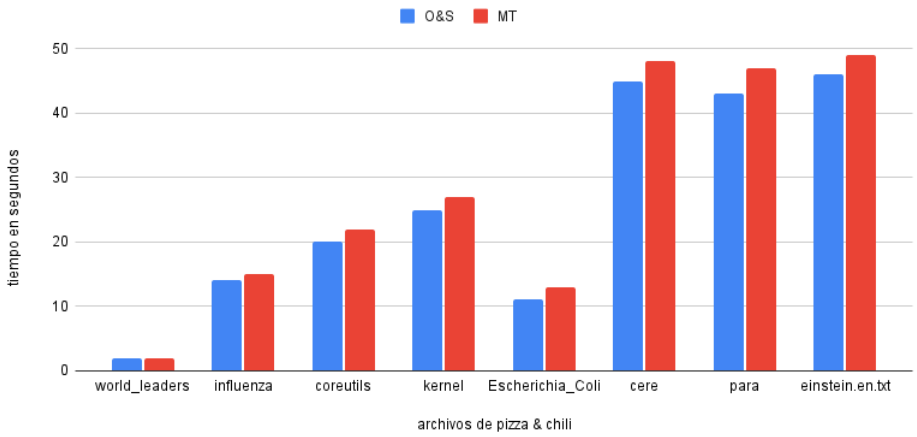

Figura 4.3.6: Tiempo de ejecución para textos repetitivos

4.3.2.2. Análisis

El comportamietno de O&S para textos repetitivos puede verse en la Figura 4.3.4, donde el peak de memoria se encuentra en las etapas de inducción de $S*$, L y S. Esto cambia para MT (ver Figura 4.3.5), donde el peak de memoria se encuentra en la etapa de SAIS, reduciendo así el peak de memoria de MT en un 10,2 % comparado con O&S.

Al igual que para los textos no repetitivos, MT es un 9 % más lento que O&S.

4.3.3. Texto en inglés

4.3.3.1. Resultados

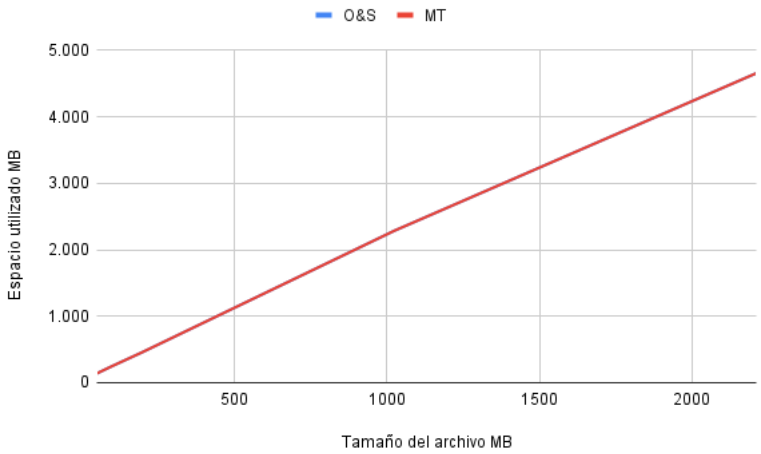

Figura 4.3.7: Peak de memoria para archivos en inglés

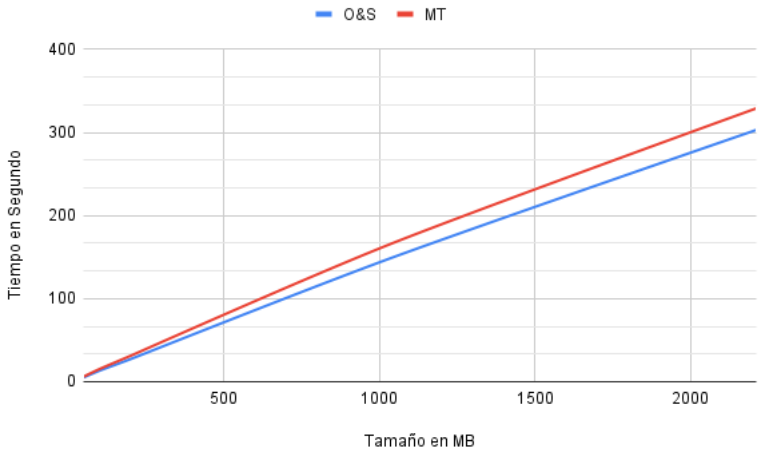

Figura 4.3.8: Tiempo de ejecución para texto en inglés

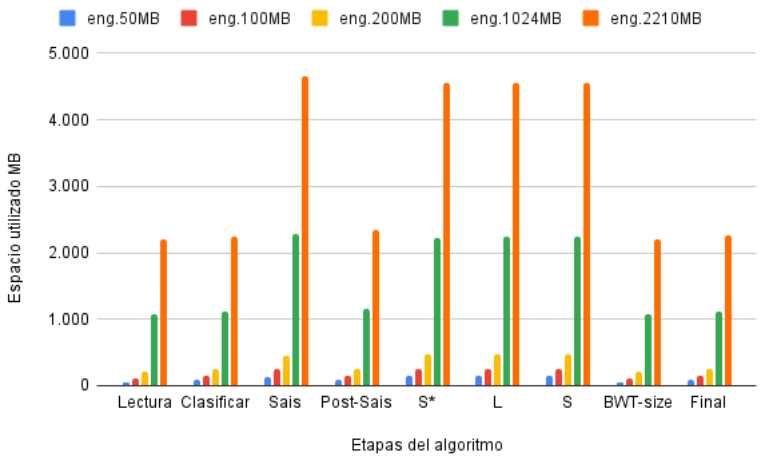

Figura 4.3.9: Uso de memoria para textos en inglés O&S

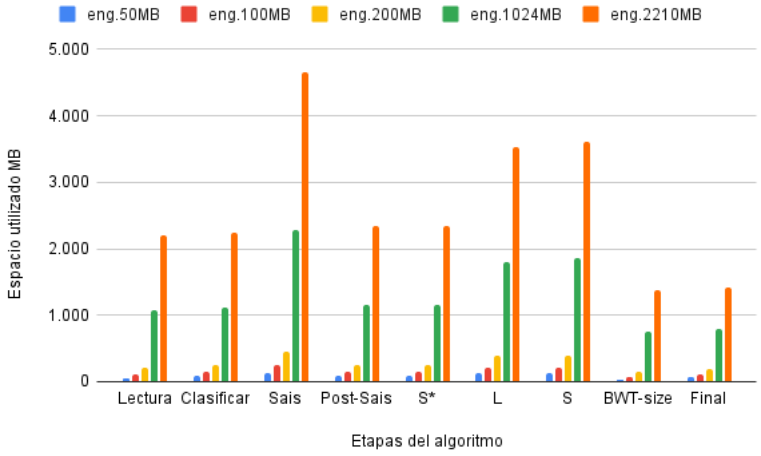

Figura 4.3.10: Uso de memoria para textos en inglés MT

4.3.3.2. Análisis

El texto en inglés utilizado entra en la categoría de texto no repetitivo. Vemos en la Figura 4.3.7 que las 2 líneas están superpuestas, mostrando que no se redujo el peak de memoria utilizada. En la Figura 4.3.8 vemos que MT es 9 % más lento que O&S.

En la figura 4.3.9 y la figura 4.3.10 vemos como a pesar de no reducir el peak de memoria en el algortimo, si se redujo el peak de memoria utilizada en las etapas desde inducción de S* hasta la etapa final.

4.3.4. Texto de wikipedia

4.3.4.1. Resultados

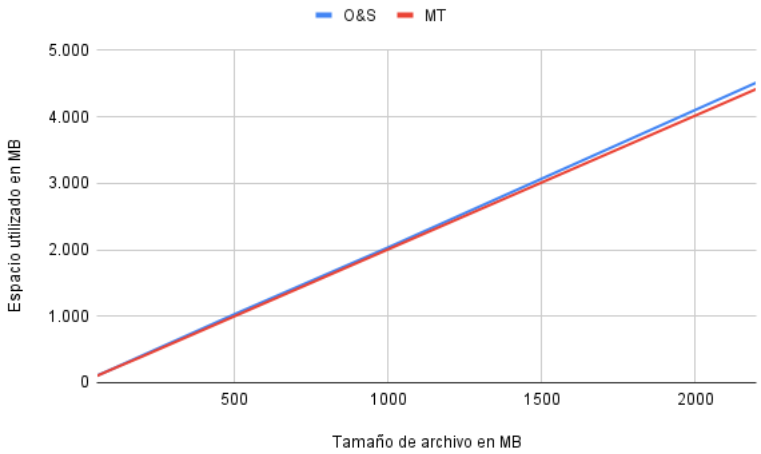

Figura 4.3.11: Peak de memoria para texto de wikipedia

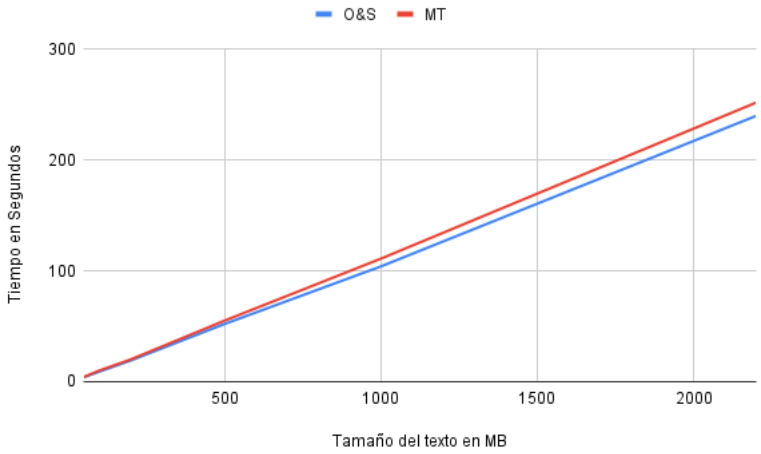

Figura 4.3.12: Tiempo de ejecución para texto de wikipedia

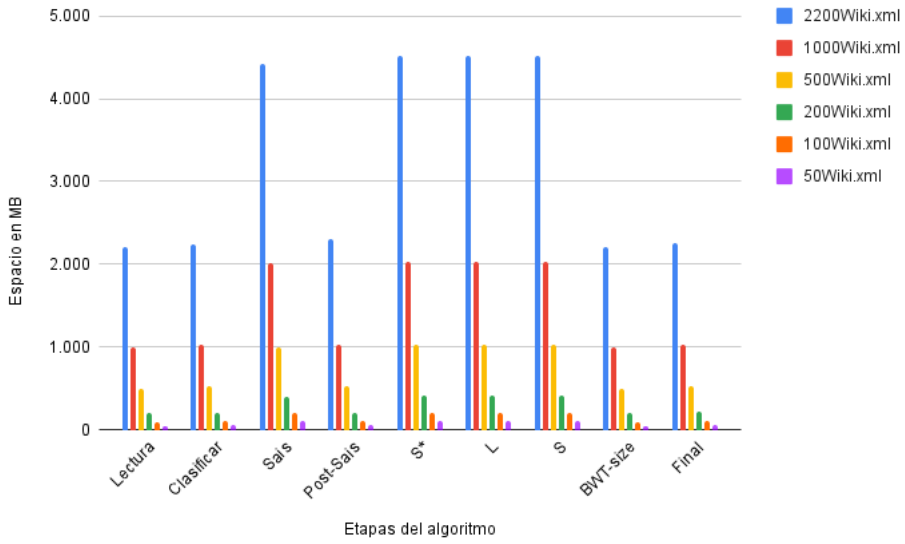

Figura 4.3.13: Uso de memoria para texto de wikipedia O&S

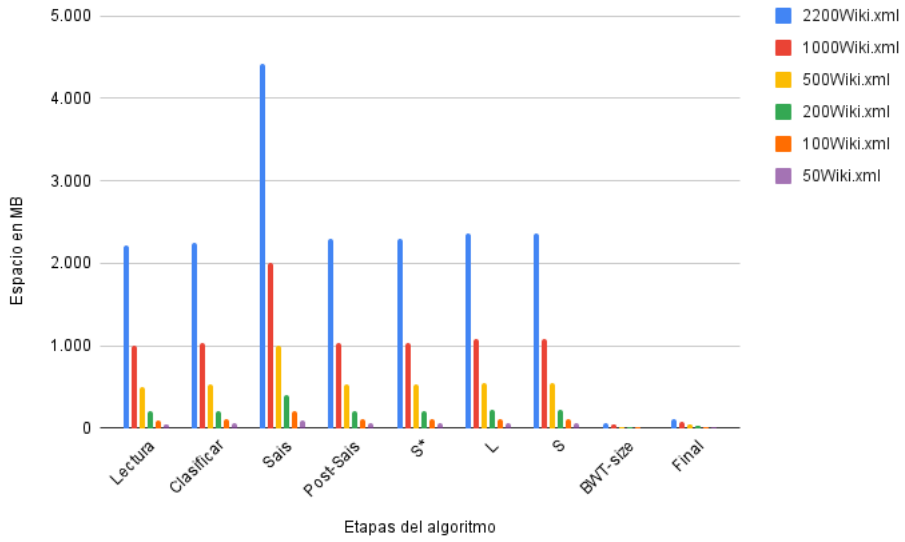

Figura 4.3.14: Uso de memoria para texto de wikipedia MT

4.3.4.2. Análisis

Los textos de Wikipedia son altamente repetitivos. Como vemos en la Figura 4.3.11, se logró reducir el peak de memoria de MT en relación a O&S.

La Figura 4.3.12 nos muestra que MT es un 9 % más lento que O&S.

En la Figura 4.3.13 vemos como O&S tiene 2 peaks en el uso de memoria, siendo

el primero la operación de SAIS y el segundo la inducción de $S*$, L y S. La Figura 4.3.14 nos muestra que MT, a diferencia de O&S, tiene solo un peak de memoria siendo este la etapa del SAIS.

4.4. Discusión

4.4.1. Repetitivos vs no repetitivos

A partir de los resultados obtenidos se observa la diferencia para MT en las dos categorías de texto: los repetitivos y los no repetitivos. A pesar de que en ambos casos se logra reducir la memoria utilizada en las etapas de $S*$ a $Final$, es claramente más eficiente para los textos repetitivos. En la etapa **BWT-size** se muestra el espacio ocupado por la transformada, donde se puede apreciar la diferencia con las distintas categorías de texto. El poco espacio utilizado por la BWT en los textos repetitivos se ve reflejado en el bajo peak de espacio al realizar la induccion en las etapas $S*$, L y S.

A pesar de ser una solución orientada mayormente para textos repetitivos, los textos no repetitivos igual mostraron un buen resultado, reduciendo la memoria utilizada en las etapas de inducción de $S*$, L y S. En estos textos no se logró reducir el peak de memoria, ya que este se encontraba durante la ejecución de SAIS. No obstante, las etapas de inducción tenían un uso de memoria muy cercano al peak, por lo que solo la mejora del peak durante SAIS no hubiera sido sufuciente para realizar una mejora significativa a este algoritmo.

Tanto para los textos repetitivos como para los textos no repetitivos, se logro mejorar el peak de memoria en las etapas de inducion, la cual era uno de los dos peaks de memoria de la ejecucion del programa. Esto permite que una mejora realizada en la etapa SAIS tenga un impacto directo en el peak de memoria del programa.

4.4.2. Tiempo

Como se observa en los gráficos de tiempo de la sección de resultados, para todos los datasets utilizados, MT es levemente más lento que O&S, tendiendo a ser hasta un 9 % más lento. El uso mas común para la BWT es la compresión de datos, por lo que luego de obtenerla, esta suele ser comprimida. O&S entrega la

BWT sin comprimir, mientras que MT ya realizó el proceso de compresión. Dado esto, dentro de un gran algoritmo de compresión esa diferencia del 9 % de tiempo sería amortizada, ya que MT puede entregar el texto comprimido, evitando tener que hacer una posterior codificación run-length.

En caso de necesitar la BWT sin comprimir, MT es capaz de entregarla. En ese caso esa diferencia del 9 % no sería compensada de forma temporal, pero sí de forma espacial por la reducción del peak de memoria para los textos altamente repetitivos.

Capítulo 5

Conclusión

La compresión es uno de los principales retos de la informática moderna. Es crucial ser eficiente en el espacio, no sólo en el resultado sino también en el proceso de compresión. La transformada Burrows-Wheeler (BWT) es una herramienta esencial para esta tarea, pero el cálculo de esta transformada no es trivial y puede consumir mucho espacio.

El algoritmo diseñado por Okanohara y Sadakane es uno de los algoritmos del estado del arte que tienen mejor rendimiento en relación al tiempo y el espacio que ocupa. Por esa razón se decidió realizar una mejora a este algoritmo, para así ser un aporte al estado del arte.

El algortimo de Okanohara y Sadakane utiliza ordenamiento inducido para poder calcular la BWT. Dentro de este algoritmo gran parte del espacio es usado para almacenar la BWT resultante. Una de las características de la BWT es su compresibilidad utilizando codificación run-length, en especial cuando se trata de textos altamente repetitivos.

Para poder disminuir el espacio utilizado, se modifica la construcción para que construya directamente la BWT comprimida, logrando reducir el espacio necesario para almacenarla. Esto es posible por que el ordenamiento inducido almacena su resultado de forma segmentada en distintas queues, segmentos que tienen las propiedades compresibles de la BWT, por lo que al modificar las queues estas almacenan los datos de forma comprimida por run-length, para luego concatenerlos y obtener la BWT, sea comprimida o no comprimida.

Esto fue probado tanto en textos altamente repetitivos, los que son altamente compresibles usando la combinación de BWT y run-length, como en textos no repetitivos, logrando reducir la memoria utilizada en las etapas de inducción, siendo notablemente eficiente en los textos repetitivos. Sin embargo, para realizar el ordenamiento inducido se necesita calcular el arreglo de sufijo de los substrings $S*$. El espacio utilizado por el arreglo de sufijos ocaciona un aumento en la memoria utilizada en las etapas previas a la inducción, siendo para los textos repetitivos este el punto donde se encuentra el peak de memoria. Para los textos no repetitivos, el peak de memoria era durante la inducción, por lo que se logra reducir este peak, pero la etapa del ordenamiento con arreglo de sufijos tenía un uso de memoria levemente menor, por lo que la mejora en la reducción del peak de memoria no es significativo.

A pesar de no lograr una mejora significativa en el peak de memoria del algoritmo global, se logró un avance importante, ya que este algoritmo tenía dos etapas con un alto uso de memoria, por lo que si se logra mejorar uno, no sería un gran avance debido a la existencia del otro.

El trabajo a seguir es realizar una mejora del uso de espacio en la etapa del arreglo de sufijos, siendo este donde está el peak de memoria. Por los avances hechos en esta memoria de título, cualquier mejora en esta etapa tendrá un impacto importante en el peak de memoria utilizada en el algoritmo. El arreglo de sufijos es una estructura de datos muy popular, por lo que está bien documentada, lo que facilitará la investigación para la optimización de esta.

Bibliografía

[1] Timo Bingmann. Malloc_count - tools for runtime memory usage analysis and profiling. https://panthema.net/2013/malloc_count/, Sep 2014.

[2] Michael Burrows and David Wheeler. A block-sorting lossless data compression algorithm. In *Digital SRC Research Report*. Citeseer, 1994.

[3] Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Proceedings 41st annual symposium on foundations of computer science*, pages 390–398. IEEE, 2000.

[4] Paolo Ferragina and Gonzalo Navarro. Pizza & chili corpus collection text. http://pizzachili.dcc.uchile.cl/, 2005.

[5] José Fuentes-Sepúlveda, Gonzalo Navarro, and Yakov Nekrich. Parallel computation of the burrows wheeler transform in compact space. *Theoretical Computer Science*, 812:123–136, 2020.

[6] Solomon Golomb. Run-length encodings (corresp.). *IEEE transactions on information theory*, 12(3):399–401, 1966.

[7] Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. Linear work suffix array construction. *Journal of the ACM (JACM)*, 53(6):918–936, 2006.

[8] Ge Nong, Sen Zhang, and Wai Hong Chan. Linear suffix array construction by almost pure induced-sorting. In *2009 data compression conference*, pages 193–202. IEEE, 2009.

[9] Daisuke Okanohara and Kunihiko Sadakane. A linear-time burrows-wheeler transform using induced sorting. In *International Symposium on String Processing and Information Retrieval*, pages 90–101. Springer, 2009.

[10] Ville Skyttä. Bzip2 / bzip2. https://gitlab.com/bzip2/bzip2/, 2014.

Apéndice A

Tablas

A1. Textos no repetitivos

Archivo (B)	Lectura (B)	Clasificar (B)	Sais (B)	Inicio mejora (B)	S* (B)	L (B)	S (B)	BWT-size (B)	Final (B)
dblp.xml	296146035	332165415	603407508	325925288	622061163	622398763	622398763	296135874	340446491
english	2210404690	2247148023	4655844587	2339443285	4549838839	4550649079	4550649079	2210395553	2261863991
dna	403936883	419933187	690680430	358754944	762682691	762735731	762735731	403927746	420303347
proteins	1184060992	1301676262	2815265766	1475838486	2659890342	2659984950	2659981446	1184051855	1362714678
pitches	55841992	104532618	189360468	123449994	179282850	179738610	179735234	55832855	126469826
sources	210875744	246949339	451649222	245747316	456613924	457785396	457785396	210866607	257701124
sources	210875744	246949339	451649222	245747316	456613924	457785396	457785396	210866607	257701124

Cuadro A1.1: Okanohara y Sadakane para textos no repetitivos

Archivo (B)	Lectura (B)	Clasificar (B)	Sais (B)	Inicio mejora (B)	S* (B)	L (B)	S (B)	BWT-size (B)	Final (B)
dblp.xml	296146035	332165415	603407508	325925288	325950680	398811456	402364096	86840484	131155840
english	2210404690	2247148023	4655844587	2339443285	2339468677	3522098677	3600520925	1370030700	1421503877
dna	403936883	419933187	690680430	358754944	358780336	733212568	797822920	506187852	522567928
proteins	1184060992	1301676262	2815265766	1475838486	1475863878	2216664870	2298995550	919164636	1097831934
pitches	55841992	104532618	189360468	123449994	123475386	161919690	162166562	48781980	119423690
sources	210875744	246949339	451649222	245747316	245772708	326044652	329827132	100013292	146852284
sources	210875744	246949339	451649222	245747316	245772708	326044652	329827132	100013292	146852284

Cuadro A1.2: Memoria de Título para textos no repetitivos

Archivo	O&S (S)	MT (S)
dblp.xml	30	33
english	300	328
dna	44	52
proteins	221	235
pitches	9	10
sources	26	29
sources	26	29

Cuadro A1.3: Tiempo para textos no repetitivos

A2. Textos repetitivos

Archivo (B)	Lectura (B)	Clasificar (B)	Sais (B)	Inicio mejora (B)	S* (B)	L (B)	S (B)	BWT-size (B)	Final (B)
world_leaders	46978342	61760771	76725857	37603257	84571439	84842879	84842879	46968181	62294735
influenza	154817692	158958609	250575497	126758669	281567225	281610185	281610185	154808555	159078745
coreutils	205290915	240589982	412294327	235039331	440321110	441094214	441094214	205281778	243430534
kernel	257970753	290843037	499814037	270345245	528306862	529027918	529027918	257961616	292486430
Escherichia_Coli	112698652	119387706	188880065	99374511	212064027	212102923	212099931	112689515	119556955
cere	461295781	502237106	747232826	437375706	898662351	898675855	898669103	461286644	502476367
para	429274895	454637047	706246017	393065473	822331232	822344224	822340976	429265758	454911584
einstein.en.txt	467635681	479451447	840408469	441774087	909400632	909896712	909896712	467626544	480013848

Cuadro A2.1: Okanohara y Sadakane para textos repetitivos

Archivo (B)	Lectura (B)	Clasificar (B)	Sais (B)	Inicio mejora (B)	S* (B)	L (B)	S (B)	BWT-size (B)	Final (B)
world_leaders	46978342	61760771	76725857	37603257	37628649	39258897	39258897	1562148	16893441
influenza	154817692	158958609	250575497	126758669	126784061	131735309	132636341	7078164	11352829
coreutils	205290915	240589982	412294327	235039331	235064723	255969675	255969675	22912092	61065323
kernel	257970753	290843037	499814037	270345245	270370637	276428717	276428717	6861684	41391237
Escherichia_Coli	112698652	119387706	188880065	99374511	99399903	122044239	125621991	31299372	38171551
cere	461295781	502237106	747232826	437375706	437401098	455164170	458297306	24355380	65549842
para	429274895	454637047	706246017	393065473	393090865	416908737	421076233	32660292	58310593
einstein.en.txt	467635681	479451447	840408469	441774087	441799479	446630175	446630175	5230692	17622735

Cuadro A2.2: Memoria de Título para textos repetitivos

Archivo	O&S (S)	MT (S)
world_leaders	2	2
influenza	14	15
coreutils	20	22
kernel	25	27
Escherichia_Coli	11	13
cere	45	48
para	43	47
einstein.en.txt	46	49

Cuadro A2.3: Tiempo para Texto retitivo

A3. Textos en inglés (no repetitivos)

Archivo (B)	Lectura (B)	Clasificar (B)	Sais (B)	Inicio mejora (B)	S* (B)	L (B)	S (B)	BWT-size (B)	Final (B)
eng.50MB	52438961	86660138	139225651	90192897	142621698	143401554	143401554	52428800	88630450
eng.100MB	104866737	140717436	249081019	147173899	252031500	252878876	252878876	104857600	144091772
eng.200MB	209724337	245775546	460871777	257087497	466802698	467646698	467646698	209715200	250331322
eng.1024MB	1073750961	1109813220	2285210581	1158262131	2232003956	2232834452	2232834452	1073741824	1120132980
eng.2210MB	2210404690	2247148023	4655844587	2339443285	4549838839	4550649079	4550649079	2210395553	2261863991

Cuadro A3.1: Okanohara y Sadakane para textos en inglés (no repetitivo)

Archivo (B)	Lectura (B)	Clasificar (B)	Sais (B)	Inicio mejora (B)	S* (B)	L (B)	S (B)	BWT-size (B)	Final (B)
eng.50MB	52438961	86660138	139225651	90192897	90218289	120795825	122296953	34701012	70907401
eng.100MB	104866737	140717436	249081019	147173899	147199291	213070131	217039163	75773340	115012251
eng.200MB	209724337	245775546	460871777	257087497	257112889	386822153	395210513	149936484	190557345
eng.1024MB	1073750961	1109813220	2285210581	1158262131	1158287523	1808550363	1852431755	753018348	799414243
eng.2210MB	2210404690	2247148023	4655844587	2339443285	2339468677	3522098677	3600520925	1370030700	1421503877

Cuadro A3.2: Memoria de Título para textos en inglés (no repetitivo)

Archivo	O&S (S)	MT (S)
eng.50MB	5	6
eng.100MB	13	15
eng.200MB	27	31
eng.1024MB	147	164
eng.2210MB	303	329

Cuadro A3.3: Tiempo para Texto en inglés (no repetitivo)

A4. Textos de Wikipedia (repetitivos)

Archivo (B)	Lectura (B)	Clasificar (B)	Sais (B)	Inicio mejora (B)	S* (B)	L (B)	S (B)	BWT-size (B)	Final (B)
2200Wiki.xml	2210604247	2246284580	4418160825	2303017455	4513612566	4514297894	4514297894	2210595110	2249374374
1000Wiki.xml	1000010161	1032229408	2006480191	1034746129	2034746130	2035340514	2035340514	1000000000	1033969730
500Wiki.xml	500009137	526804505	1001667044	532101306	1032101307	1032633979	1032633979	500000000	528064283
200Wiki.xml	200009137	213640527	398175634	213326160	413326161	413703681	413703681	200000000	214214097
100Wiki.xml	100009137	108310259	202698912	105198668	205198669	205491885	205491885	100000000	108736173
50Wiki.xml	50009137	56645719	101746030	55175656	105175657	105465881	105465881	50000000	57033961

Cuadro A4.1: Okanohara y Sadakane para textos de Wikipedia (repetitivo)

Archivo (B)	Lectura (B)	Clasificar (B)	Sais (B)	Inicio mejora (B)	S* (B)	L (B)	S (B)	BWT-size (B)	Final (B)
2200Wiki.xml	2210604247	2246284580	4418160825	2303017455	2303042847	2363918327	2363918327	64722036	103506039
1000Wiki.xml	1000010161	1032229408	2006480191	1034746129	1034771521	1077271681	1077271681	43857324	77831793
500Wiki.xml	500009137	526804505	1001667044	532101306	532126698	551822074	551822074	20139564	48208586
200Wiki.xml	200009137	213640527	398175634	213326160	213351552	224031408	224031408	10670412	24889248
100Wiki.xml	100009137	108310259	202698912	105198668	105224060	111840604	111840604	6522708	15263620
50Wiki.xml	50009137	56645719	101746030	55175656	55201048	58595648	58595648	3225348	10264048

Cuadro A4.2: Memoria de Título para textos de Wikipedia (repetitivo)

Archivo	O&S (S)	MT (S)
2200Wiki.xml	240	252
1000Wiki.xml	104	111
500Wiki.xml	52	55
200Wiki.xml	19	20
100Wiki.xml	9	10
50Wiki.xml	4	4

Cuadro A4.3: Tiempo para textos de Wikipedia (repetitivo)