Instituto Tecnológico y de Estudios Superiores de Monterrey
Campus Mty

Tecnológico de Monterrey

Materia

Análisis y diseño de algoritmos avanzados

Nombre del archivo

E2. Actividad Integradora 2

Cruz Daniel Pérez Jiménez - A01736214

Erwin Porras Guerra - A01734881

Marlon Yahir Martínez Chacón - A01424875

Grupo 601

14 de Noviembre del 2023

Para poder realizar las compresión de un archivo de texto se utilizaron los siguientes algoritmos

SA-IS:
El algoritmo lo que hace es tomar la cadena de texto que representa el contenido del archivo de texto y construir el suffix array mediante la combinación de distintas técnicas y estructuras de datos para que sea lo más óptimo posible, este algoritmo es una parte fundamental para poder obtener el Burrow-wheeler transform ya que a partir de lo que obtenemos de suffix array, que sería la columna First del BWT podemos obtener el Last al utilizar el siguiente algoritmo (bwt).
La complejidad temporal con la que cuenta este algoritmo es O(n), ya que cuenta con diversas optimizaciones y su complejidad espacial es de O(n+k), ya que depende del tamaño del alfabeto que sería k

Burrows-Wheeler Transform:

El algoritmo de burrows-wheeler toma como parámetro el suffix array que fue creado con anterioridad, la implementación que tenemos lo que hace es obtener el índice del carácter que está justo antes del inicio de este sufijo en la cadena original y se le agrega a la nueva lista del bwt que estamos creando la columna Last de nuestro bwt para poder utilizar más adelante para el move-to-front.

```
def bwtFunction(s, sa, bwt, secciones, occ):
    for i in range(len(s)):
        if i == 0 or s[sa[i-1]] != s[sa[i]]:
            secciones[s[sa[i]]] = i
        bwt[i] = s[sa[i]-1]

        for j in secciones.keys():
            if bwt[i] == j:
                occ[j].append(occ[j][i] + 1)
            else:
                occ[j].append(occ[j][i])
```

Al mismo tiempo se están obteniendo las estructuras de secciones y ocurrencias que nos ayudarán a poder hacer la transformada inversa del bwt, por lo tanto la complejidad temporal del bwt es de O(n+a) esto es debido a que se el ciclo for depende de la cantidad de elementos que tenga el texto original y del alfabeto para obtener las ocurrencias y su complejidad espacial también sería O(n+a).

Move-To-Front:

```
def moveToFront(index,alph):
    word=alph[index]
    alph2=alph[:]
    alph[1:index+1]=alph2[:index]
    alph[0]=word

def moveToFrontCoding(alphabet,T):
    alph2=alphabet[:]
    mtf=[]
    for i in range (len(T)):
        j=match(T[i],alph2)
        mtf.append(j)
        moveToFront(j,alph2)
    return mtf
```

Una vez que se obtiene el BWT el siguiente paso es aplicarle a ese resultado el algoritmo del move-to-front, que lo que hace básicamente es reorganizar dinámicamente el alfabeto o una lista de símbolos, de manera que los símbolos recientemente utilizados se muevan hacia el frente de la lista. Cuando se accede a un símbolo en particular, este símbolo se coloca al principio de la lista, lo que potencialmente mejora la compresión de datos si hay repetición de símbolos o patrones en la secuencia.

El algoritmo que se implementó lo que hizo fue que el carácter que se estaba comparando con el alfabeto obtenía su posición y lo agregaba a una lista y de ahí se movía el carácter al frente

del bwt, pero al momento de encontrar el mismo carácter en la siguiente ocurrencia, se añadía un 0, debido a que su posición era la número 0 porque estaba al frente de la lista, este uso de los ceros será útil más adelante para hacer uso del huffman y mejorar la compresión.
La complejidad temporal que se tiene de este algoritmo es de O(n*k) porque depende del número de elemento que se tenga de la cadena del bwt y también por las operaciones que se hacen para mover al carácter que se está utilizando al frente; y su complejidad espacial es de O(n+m) donde n es el número de caracteres del Bwt y m es el alfabeto que se esta utilizando.

Run length Encoding:

Una vez que se obtiene una lista con las posiciones de los números en el move-to-front, el run length comprime más esa información tomando los 0's que existen después de algún valor, los cuenta y regresa una nueva lista con el valor de la posición del carácter del move-to-front, el 0 y el número de ocurrencias del 0.

```
def runLengthEncoding(moveToFront):
    encoded = []
    zeroCounter = 0
    for i in moveToFront:
        if i == 0:
            zeroCounter += 1
            if zeroCounter == 255:
                encoded.append(0)
                encoded.append(255)
                zeroCounter = 0
        else:
            if zeroCounter > 0:
                encoded.append(0)
                encoded.append(zeroCounter)
                zeroCounter = 0
            encoded.append(i)
    if zeroCounter > 0:
        encoded.append(0)
        encoded.append(zeroCounter)
    return encoded
```

En el código lo que se hace es utilizar una nueva lista, primero se utiliza un ciclo for para iterar por cada uno de los elementos del move-to-front y por cada elemento comprueba si es que es un 0, en caso que sea un 0 se le suma uno al contador de ceros, y una vez que ya el número no es un 0 se le agrega a la lista el número 0 y el contador, además en caso de superar los 255 ocurrencias del 0, agrega el 255 y el 0 a la nueva lista y empieza a contar otra vez para tener todas las ocurrencias

Por ejemplo si tenemos la siguiente cadena que recibimos del move-to-front: 1,3,4,0,0,0,4,3,1, el algoritmo de run length encoding regresará como resultado 1,3,4 0, 3, 4, 3,1; como se puede observar en el resultado regresa una lista igual al move-to-front sólo que se comprime al contar las ocurrencias que tienen los 0's (los cuales simbolizan el mismo número) en la cadena.
La complejidad temporal con la que cuenta este algoritmo es O(n), ya que se utiliza un ciclo for para poder recorrer todo la lista que se obtiene del move-to-front y procesar cada uno de los elementos para poder contar los 0's, así mismo su complejidad espacial sería de O(n), por la misma razón que depende del número de elementos que tenga la lista inicial

Huffman:

Antes de empezar este algoritmo necesitamos primero cierta información. Esta parte del programa lo que hace es tomar el archivo generado por el lengthRun y lo interpreta como un vector, como tal el tamaño de este será el que se va a comprimir, esta parte se hace en tiempo O(n) y espacio O(n). Ya teniendo el vector se recorre en su totalidad contando las frecuencias de cada número, tomando en cuenta que el número después de cada 0 significa las instancias que se le deben sumar a 0, esta parte igual es en tiempo O(n). Cuando obtenemos las frecuencias de cada número, podemos comenzar con la creación del árbol de huffman.

Para crear el árbol de huffman tenemos que ir representando cada tipo de dato como un nodo, comenzamos con los datos con menor frecuencia, les asignamos un nodo rama, a este nodo rama se le asigna el valor de las suma de las frecuencias de sus hijos, y se repite el proceso con el nodo de frecuencia más próxima. Se repite este proceso hasta tener un árbol completo y podemos proceder a asignar códigos. La creación del árbol tiene una complejidad temporal de O(nlogn) y espacial O(n)

```
HuffmanNode* buildHuffmanTree(const map<int, int>& charFrequency) {
    priority_queue<HuffmanNode*, vector<HuffmanNode*>, CompareNodes> minHeap;
    // Create a leaf node for each character and add it to the min heap
    for (const auto& pair : charFrequency) {
        minHeap.push(new HuffmanNode(pair.first, pair.second));
    }
    // Build the Huffman tree
    while (minHeap.size() > 1) {
        HuffmanNode* left = minHeap.top();
        minHeap.pop();
        HuffmanNode* right = minHeap.top();
        minHeap.pop();

        HuffmanNode* newNode = new HuffmanNode('ä', left->frequency + right->frequency);
        newNode->left = left;
        newNode->right = right;

        minHeap.push(newNode);
    }
    return minHeap.top();
}
```

La creación de los códigos de Huffman es bastante simple, solamente se asigna un 0 cada vez que se recorre hacia la izquierda, y un 1 cada vez que se recorre hacia la derecha, se van contando estos valores hasta que se llegue a un nodo hoja y se le asigna este código.

```
void generateHuffmanCodes(HuffmanNode* root, const string& code, map<int, string>& huffmanCodes) {
    if (root == nullptr) {
        return;
    }
    if (root->data != 'ä') {
        huffmanCodes[root->data] = code;
    }
    generateHuffmanCodes(root->left, code + "0", huffmanCodes);
    generateHuffmanCodes(root->right, code + "1", huffmanCodes);
}
```

Finalmente la parte de compresión se realiza al volver a interpretar el vector inicial con los valores utilizando los códigos de huffman y se insertan en un archivo binario, como lo fue mencionado anteriormente el tamaño del vector será el que se comprimirá, utilizando los algoritmos previos con el texto de “Drácula” para obtener este vector, podemos observar que el tamaño de este es de 2452316 bytes, o 2.45 megabytes. El tamaño del archivo binario donde se comprimió la información termina siendo en este caso de 2.17 megabytes, logrando una compresión de 0.31 megabytes.

```
cout << "Bytes of int vector: "
<< sizeof(vector<int>) + (sizeof(int) * frequencies.size()) << endl;
```

```
Bytes of int vector: 2452316
```

output.bin	11/14/2023 9:32 PM	BIN File	2,174 KB

El proceso de descompresión también es bastante simple, para este se empieza a leer cada bit en el archivo binario, y se va recorriendo el árbol de huffman hasta llegar a un nodo hoja, cuando se llega a un nodo hoja se descubre el valor de este y se interpreta. Este proceso se realiza en tiempo O(n)

```cpp
vector<int> decode_file(struct HuffmanNode* root, string s)
{
    vector<int> ans;
    struct HuffmanNode* curr = root;
    for (int i = 0; i < s.size(); i++) {
        if (s[i] == '0')
            curr = curr->left;
        else
            curr = curr->right;

        // reached leaf node
        if (curr->left == NULL and curr->right == NULL) {
            //cout << curr->data << endl;
            ans.push_back(curr->data);
            curr = root;
        }
    }
    // cout<<ans<<endl;
    return ans;
}
```

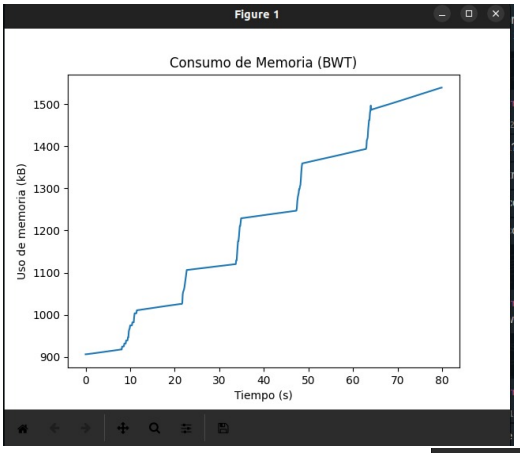
Estadísticas resultantes de la evaluación

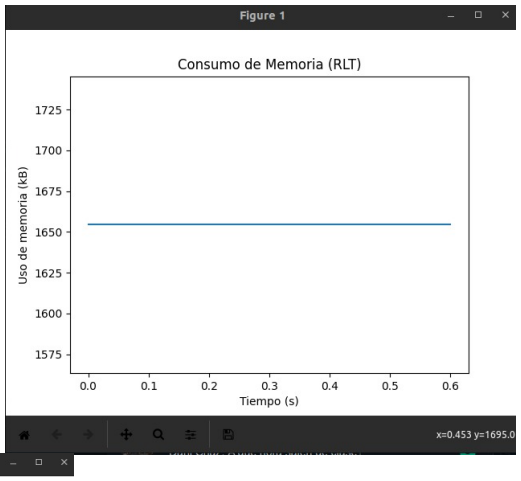
Las siguientes son los datos que se obtuvieron al hacer uso de los algoritmos con el archivo de Dracula.txt que tiene peso de 849 kb.

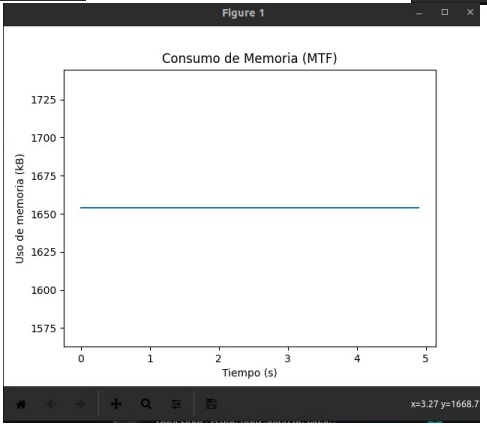
Tiempo de ejecución (Drácula)

```
BURROWS WHEELER
Tiempo de Ejecución: 7.076077222824097 s
MOVE TO FRONT
Tiempo de Ejecución: 0.5090298652648926 s
RUN-LENGTH
Tiempo de Ejecución: 0.04413270950317383 s
```

Uso de la memoria (Drácula)



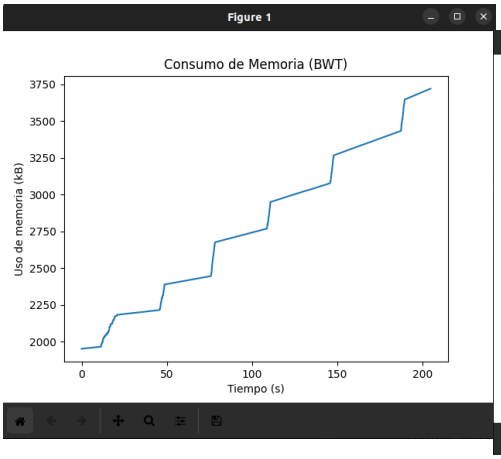


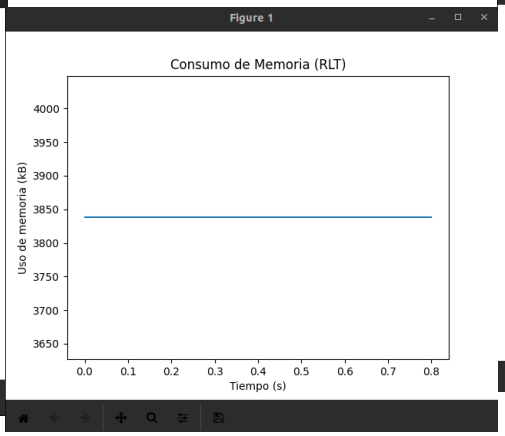


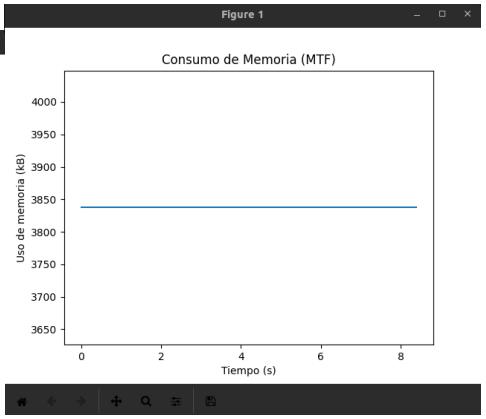
Tiempos de Ejecución (Iliada)

```
BURROWS WHEELER
Tiempo de Ejecución: 17.982341527938843 s
MOVE TO FRONT
Tiempo de Ejecución: 0.8446555137634277 s
RUN-LENGTH
Tiempo de Ejecución: 0.056574106216430664 s
```

Uso de Memoria (Iliada)



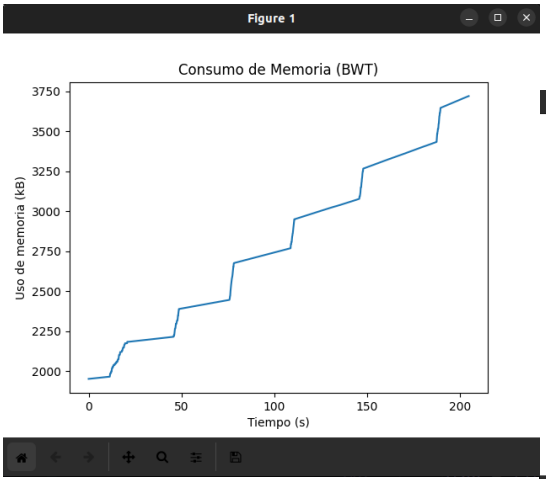


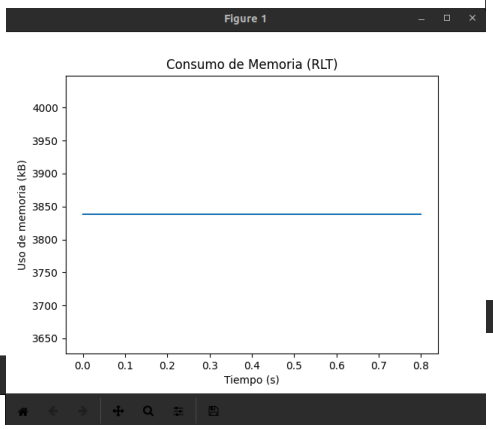


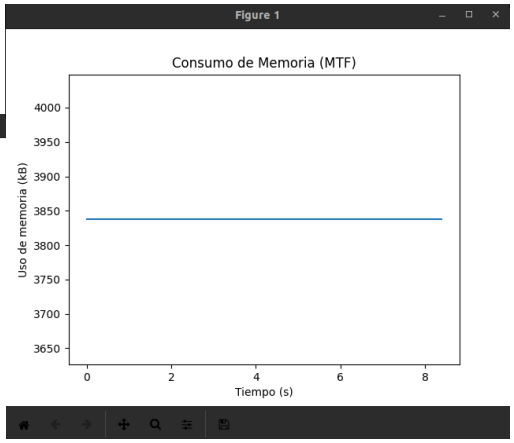
Tiempos de Ejecución (War)

```
BURROWS WHEELER
Tiempo de Ejecución: 32.03930616378784 s
MOVE TO FRONT
Tiempo de Ejecución: 1.866117000579834 s
RUN-LENGTH
Tiempo de Ejecución: 0.15441298484802246 s
```

Uso de memoria (War)







Reflexiones

Erwin Porras Guerra:

Para mi todo este proceso fue muy interesante, siempre me había preguntado cómo funcionaban los programas de compresión como 7zip o winrar y antes estos me parecían casi magia, ahora que entiendo como funcionan me fascina mucho este proceso y entiendo la importancia de crear este tipo de algoritmos. La importancia de este tipo de algoritmos es muy grande ya que se ocupan en prácticamente todas las áreas de la computación, y particularmente para el uso cotidiano nos ofrecen más opciones al momento de trabajar con grandes cantidades de datos.

Marlon Yahir Martínez Chacón:

Durante las tareas que he tenido a lo largo de mi vida estudiantil que involucran el uso y entrega de archivos a través de alguna plataforma de revisión, siempre he utilizado algún programa de compresión para poder mandarlos y que sean aceptados por la plataforma porque a veces tenían alguna limitante de tamaño, pero con la compresión si eran aceptados, por lo que siempre me había preguntado acerca de la ingeniería que hay detrás de las técnicas de compresión que nos entregan los archivos .rar, .zip, etc; por lo que el ver los algoritmos fue un proceso fue muy interesante y el uso de los archivos binarios para poder tener la compresión necesaria también fue un gran aprendizaje para mi. Con esto llegue a la conclusión de que a pesar de que puede ser abrumador el pensar que hay detrás de algunos programas que utilizamos en nuestras actividades por todo lo que se usa, una vez que entiendes la lógica es sencillo de utilizar e implementar.

Cruz Daniel Pérez:
Los archivos son fundamentales para la vida moderna y muy rara vez nos ponemos a pensar en todo lo que este conjunto de información conlleva, ya que si nos ponemos a pensar, la compresión de archivos es fundamental para almacenar información o enviarla en plataformas digitales, por ejemplo, resulta fascinante cómo es que se puede reducir el tamaño de un archivo y así ahorrar muchos recursos de almacenamiento, esto aunado con algoritmos cuya estructura está diseñada para procesar miles de datos de una manera eficaz, nos traen a la compresión de archivos que podrían parecer incompresibles. Durante esta actividad pude descubrir la lógica que hay detrás de la compresión y codificación de archivos, así como de los distintos algoritmos que realizan esta tarea, siendo así que en esta actividad logramos comprimir razonablemente grandes cantidades de información.

Link al repositorio:
https://github.com/MarlonY123/AlgoritmosE2.git