ALGORITMIA PARA PROBLEMAS DIFÍCILES

Práctica 2

Santiago Buey: 747827

David Solanas: 738630

1. Introducción

En esta práctica se nos piden que implementemos una estructura de datos, como es un vector de sufijos, y los algoritmos Burrows Wheeler, Move To Front y Huffman para la compresión. Con todos estos ingredientes, se creará un compresor similar a bzip2. Se trata de un compresor de ficheros por ordenación de bloques, el cual obtiene un porcentaje de compresión que por lo general es bastante superior al de otros compresores que se basan en los algoritmos LZ77 y LZ78, aunque depende del contenido del fichero a comprimir.

La forma de utilizarlo será por línea de comandos:

./bzip2 -c nombreFicheroAComprimir

./bzip2 -d nombreFicheroADescomprimir

La extensión del fichero comprimido devuelto será .bz2.

2. Estructuras y algoritmos implementados

2.1. Vector de sufijos

Para la creación de la estructura de datos de vectores de sufijos se han seguido las pautas encontradas en [1] y [2]. Tras leer el fichero de entrada y almacenarlo en un vector de *int* lo primero que se ha de hacer es construir el vector de sufijos A^{12}, para ello se han de llevar a cabo los siguientes pasos:

- Genera las posiciones no múltiplos de 3 de las tripletas (subcadena de tamaño 3) de la entrada *s*: Almacena en un vector la posición (todas las posiciones menos las múltiplo de 3, i mod 3 != 0) de todas las tripletas del texto. *O(n)*.
- Realizar tres pasadas del algoritmo Radix Sort [2] para ordenar los sufijos lexicográficamente. *O*((n + k)d + log n), con n tamaño de la entrada, k tamaño del alfabeto, d longitud de las cadenas (3).
- Calcular el rango (rank) de los sufijos: Ir tripleta a tripleta asignando un rank empezando por 0, si la tripleta *i* es distinta de la tripleta *i-1* entonces el rank de la tripleta *i* será el (rank de *i-1*) + 1. Si son iguales se le asigna el mismo rank que la tripleta *i-1*. *O(n).*
- Construir la cadena *T*: Es la concatenación de *t1* y *t2*, donde:
  - $t1 = r_1\ r_4 \dots r_{1+n}$
  - $t2 = r_2\ r_5 \dots r_{2+n}$
    r_i = i-ésimo rank
    Este paso se realiza al mismo tiempo que se calculan los rank para reducir el coste temporal del algoritmo. *O(n)*.
- Construir el vector de sufijos A^{12} a partir de *T*: Si todos los rank r_i son únicos se puede obtener el vector de sufijos A^{12} directamente de *T* O(n). Si por el contrario hubiera algún r_i repetido se debería repetir el proceso anterior recursivamente, nótese que si hubiera demasiadas llamadas recursivas habría que complicar el algoritmo para mantener el tiempo de ejecución.

Una vez obtenido A^{12} el siguiente paso es calcular A^0, para ello se ordenan los sufijos de posición múltiplo de 3 de A^{12} por su primer carácter con el algoritmo Radix Sort [2]. Finalmente se mezclan A^0 y A^{12} (ambos ordenados) recorriendo ambos vectores de forma simultánea y comparando los sufijos de A^0 y A^{12} y ordenando de menor a mayor tal y como se explica en [1] (*O(n)*).

Con esta implementación del algoritmo *Skew* para la construcción de vectores de sufijos [1][2], se ha obtenido una complejidad temporal *O(n)* si se consideran las comparaciones entre números *O(1)*, si no la complejidad temporal del algoritmo implementado es $O(n \cdot log\ n)$ con n el tamaño de la entrada.

2.2. Transformada de Burrows-Wheeler

La transformada de Burrows-Wheeler es un algoritmo que reestructura la cadena que toma como entrada de forma que el mensaje transformado resultante sea más simple de comprimir, ya que tendrá más agrupados sus caracteres que sean iguales.

La manera clásica de hacer la transformada Burrows-Wheeler de una cadena sería, en primer lugar, obtener sus rotaciones, tantas como caracteres tenga. Después se ordenan estas rotaciones, y la última columna resultante que forman todas ellas será la transformada. El proceso, aplicado sobre la cadena "^BANANA|" se ilustra en la siguiente figura[4] :

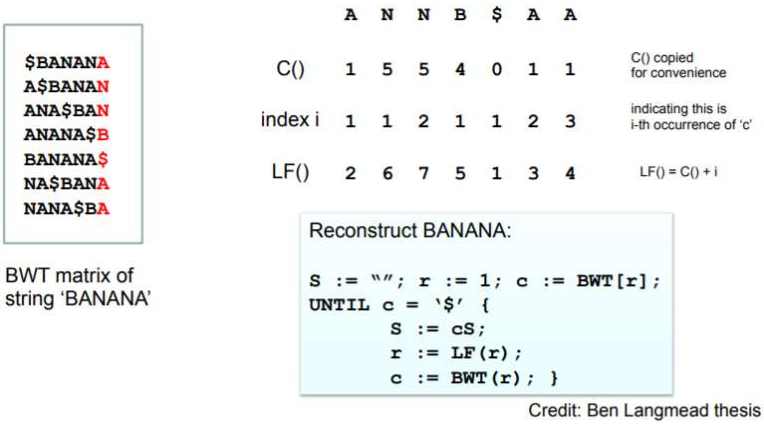
Transformation				
1. Input	2. All rotations	3. Sort into lexical order	4. Take the last column	5. Output
^BANANA|	^BANANA| |^BANANA A|^BANAN NA|^BANA ANA|^BAN NANA|^BA ANANA|^B BANANA|^	ANANA|^B ANA|^BAN A|^BANAN BANANA|^ NANA|^BA NA|^BANA ^BANANA| |^BANANA	ANANA|^B ANA|^BAN A|^BANAN BANANA|^ NANA|^BA NA|^BANA ^BANANA| |^BANANA	BNN^AA|A

Sin embargo, se puede de una forma más simple, ya que es posible calcular el vector de sufijos del texto de entrada, como se ha explicado en el apartado anterior. Tomando como entrada la cadena a transformar y su vector de sufijos, se ha implementado la función *transformadaBW().* Esta función recorre la cadena de entrada y para cada una de sus iteraciones ejecuta el siguiente cálculo para obtener el último carácter de cada rotación:

```
transformada[i] = input[(suffix_array[i] - 1 + tam) % tam];
```

siendo input la cadena de entrada, suffix_array el vector de sufijos, y tam su dimensión.

Esta transformada es inversible, lo que quiere decir que a partir de ella podremos obtener la cadena de texto original, como se ha implementado en la función *inversaBW()*.



En esta implementación [6] mostrada en la imagen anterior se obtiene en primer lugar el vector de enteros c[ ], que tendrá tantos componentes como caracteres tenga la cadena de entrada. Cada componente contendrá la cuenta de caracteres menores al correspondiente en la posición.

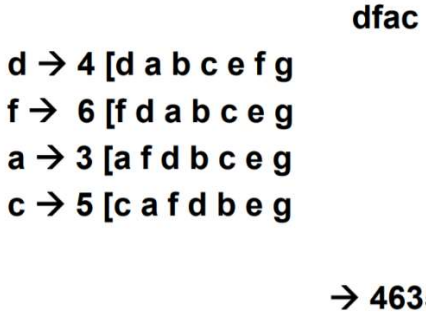
A continuación se genera el vector aparicionesC[ ], cuyo tamaño será igual al del diccionario de caracteres, almacenará el total de veces que ha aparecido el carácter correspondiente a su posición. Este vector es necesario para crear el vector index[ ], que almacena el número de ocurrencias hasta el momento del correspondiente carácter i. El vector LF[ ] es igual a la suma de index y c (LF=c + index), y con los enteros que en el se han ido calculando se podrá finalmente decodificar la transformada de Burrows-Wheeler y obtener el texto original.

2.3. Algoritmo Move To Front

Move to front es un algoritmo que transforma cada mensaje en una secuencia de enteros. Toma como parámetros de entrada la cadena a transformar, y el diccionario de los símbolos que se usan en la cadena, que deberá estar ordenado. Nosotros, en nuestro caso, usaremos como diccionario los 256 caracteres de la tabla ASCII, de forma que se contienen los caracteres que pueden aparecer en la mayoría de textos. Además, generar el diccionario será tan simple como realizar 256 iteraciones:

```
for (i = 0; i < 256; i++)
{
    diccionario[i] = (unsigned char)i;
}
```

El algoritmo consiste en que para cada uno de los caracteres de la cadena, se devolverá su posición en el diccionario, y se desplazará al principio de este, de ahí el nombre de move to front. Se puede ver intuitivamente en la siguiente figura[3], cuya entrada es la cadena “dfac” y su salida la secuencia de enteros 4635:

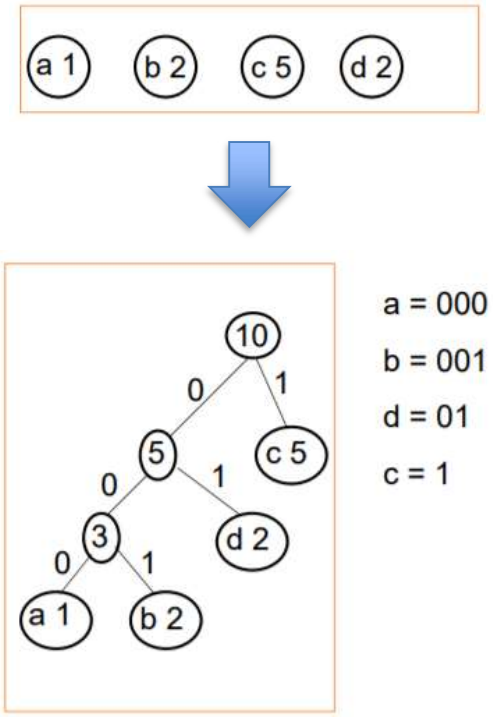


A la hora de implementar este algoritmo (*moveToFront()*), el proceso ha consistido en recorrer la cadena de entrada, y para cada uno de sus caracteres, buscar su posición en el diccionario que se pasa como parámetro a la función. Una vez encontrada (si por alguna razón no se encontrara daría error y finalizaría el programa) se añade a la secuencia de enteros a devolver el índice que representa dicha posición. Finalmente, se traslada el carácter desde el índice encontrado a la posición 0 del diccionario.

Para realizar la operación inversa (*moveToFrontInverso()*), es decir, a partir de una secuencia de índices obtener la cadena original, también se necesitará pasar como parámetro el diccionario original ordenado (tabla ASCII). Se recorrerá la secuencia de índices, y para cada uno de ellos se obtendrá el caracter del diccionario que indexan, añadiéndolo a la cadena de salida. Cada carácter será después desplazado a la primera posición del diccionario. De esta forma, una vez recorrida por completo la secuencia de índices se habrá obtenido el texto inicial. Esta operación se utilizará para descomprimir los ficheros .bz2.

2.4. Huffman

El algoritmo de Huffman es un algoritmo para la construcción de códigos de Huffman, el cual toma un alfabeto de símbolos, con sus frecuencias de aparición asociadas, y produce códigos Huffman para ese alfabeto y frecuencias concretas. Estos códigos son generados mediante la creación de un árbol, y son códigos libres de prefijo, de forma que se elimina la posibilidad de que ocurra ambigüedades a la hora de representar un determinado carácter. A continuación se muestra cómo sería el árbol generado y los códigos resultantes a partir de 4 caracteres con sus respectivas frecuencias de aparición[3] :



Como ya se hizo una implementación del código de Huffman anteriormente en las prácticas de la asignatura Algoritmia Básica, se ha reutilizado el código entregado el año pasado por el integrante

del grupo David Solanas. Este código se supone fiable, ya que fue sometido a numerosas pruebas en su día y obtuvo la máxima calificación.

2.5. Obtención del compresor

Finalmente, se deben juntar todos los elementos descritos hasta ahora para obtener un compresor similar a bzip2. Su utilización será por línea de comandos, invocándose de las formas:

./bzip2 -c nombreFichero → Para comprimir el fichero de nombre *nombreFichero*
./bzip2 -d nombreFichero → Para descomprimir el fichero de nombre *nombreFichero*

El orden de utilización de los elementos anteriores para el compresor será el siguiente:
Comprimir:
Crear vector de sufijos → Calcular la transformada Burrows Wheeler con el vector de sufijos y el texto a comprimir como entradas → Hacer move to front de la BWT calculada → Aplicar Huffman → Imprimir en fichero .bz2

Descomprimir:
Descompresión de Huffman → Move To Front inverso → Burrows-Wheeler inverso → Imprimir el fichero descomprimido

3. Pruebas realizadas

En las fases iniciales, como la creación del vector de sufijos, se tomaban como entrada textos grandes y ejemplos de internet, para probar que se creaban correctamente.
Para probar la correcta implementación de los algoritmos que se iban implementando, se utilizaban como entrada cadenas simples como “banana” o “abracadabra” , para comprobar que la transformada Burrows-Wheeler, move to front, etc. se calculaban correctamente.

Input: *text = “banana$”*
Output: *Burrows-Wheeler Transform = “annb$aa”*

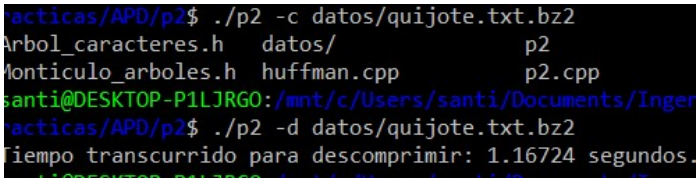
Input: *text = “abracadabra$”*
Output: *Burrows-Wheeler Transform = “ard$rcaaaabb”* [5]

Al ser la transformada de Burrows-Wheeler y Move To Front algoritmos inversibles, se probaba que la cadena inicial y la resultante de aplicar un algoritmo y su inversa fuesen iguales.

Más adelante, con la compresión y descompresión ya implementadas, se probaba que el fichero antes y después de comprimir fuesen iguales, mediante el comando diff:

```
racticas/APD/p2$ ./p2 -c datos/pract1_APD.cpp
Tiempo transcurrido para comprimir: 0.0156908 segundos.
santi@DESKTOP-P1LJRGO:/mnt/c/Users/santi/Documents/Ingenieria_Informatica_7o
racticas/APD/p2$ ./p2 -d datos/pract1_APD.cpp.bz2
Tiempo transcurrido para descomprimir: 0.514922 segundos.
santi@DESKTOP-P1LJRGO:/mnt/c/Users/santi/Documents/Ingenieria_Informatica_7o
racticas/APD/p2$ diff datos/pract1_APD
pract1_APD.cpp          pract1_APD.cpp.bz2      pract1_APD_original.cpp
santi@DESKTOP-P1LJRGO:/mnt/c/Users/santi/Documents/Ingenieria_Informatica_7o
racticas/APD/p2$ diff datos/pract1_APD.cpp datos/pract1_APD_original.cpp
santi@DESKTOP-P1LJRGO:/mnt/c/Users/santi/Documents/Ingenieria_Informatica_7o
racticas/APD/p2$
```

Se comprimían mplo el txt del quijote:



4. Reparto del trabajo

El trabajo ha sido realizado conjuntamente entre los dos miembros del grupo, tanto en la fase de diseño de los conjuntos de datos de prueba, algoritmo, memoria, etc. como en la fase de implementación. Además cada miembro ha realizado su propio conjunto de pruebas y propuesto posibles mejoras y cambios al encontrar posibles fallos.

5. Referencias

[1] - *David Weese (April 4, 2015, 13:02). Linear time suffix array construction* http://www.mi.fu-berlin.de/wiki/pub/ABI/SS13Lecture3Materials/script.pdf

[2] - *String Matching (28-11-19).* http://webdiis.unizar.es/asignaturas/APD/wp/wp-content/uploads/2013/09/191125StringMatching1.pdf

[3] - *Algoritmos de compresión de datos (16-12-19).* http://webdiis.unizar.es/asignaturas/APD/wp/wp-content/uploads/2013/09/191204compression2.pdf

[4] – *Wikipedia: Burrows-Wheeler transform* https://en.wikipedia.org/wiki/Burrows%E2%80%93Wheeler_transform

[5] – *GeeksForGeeks: Burrows-Wheeler data transform algorithm* https://www.geeksforgeeks.org/burrows-wheeler-data-transform-algorithm/

[6] – *Inversa Burrows-Wheeler* https://web.stanford.edu/class/cs262/presentations/lecture4.pdf