Algoritmia para problemas difíciles

Práctica 2

Informe de resultados

Andrés Gavín Murillo 716358

Darío Ferrer Chueca 721132

21 de enero de 2020

Introducción

El objetivo de la práctica consiste en desarrollar un programa de compresión y descompresión utilizando la transformada de Burrows-Wheeler con vectores de sufijos, Move-To-Front y Huffman.

La práctica se ha desarrollado en C++, donde se explican todos los pasos realizados, y se compila y ejecuta de la siguiente manera (realizado para Hendrix):

make

Generar datos de prueba: **./practica2 [-c | -d] <file>**

-c Crea el archivo comprimido <file>.bwh usando BWT, Move-To-Front y Huffman.

-d Crea el archivo descomprimido <file> usando BWT, Move-To-Front y Huffman.

El archivo a descomprimir debe tener extensión .bwh

Detalles de implementación

Creación de vector de sufijos

Hay diferentes aproximaciones para implementar este algoritmo. La más obvia es la opción con coste $O(n^2 log(n))$, pero se puede mejorar notablemente. La implementación realizada está basada en las explicaciones y algoritmo de la siguiente página:
https://www.hackerrank.com/topics/suffix-array

La idea sobre la que se apoya el siguiente algoritmo es que no se están ordenando strings independientes, si no que todos son parte de un único string. De esta manera, se ordena primeramente por el primer carácter de los sufijos, posteriormente para los siguientes dos, posteriormente cuatro y así sucesivamente, llegando a hacer únicamente log(n) ordenaciones, no n como en el caso del algoritmo inicial.

Si se han ordenado los sufijos teniendo en cuenta los 2^i primeros caracteres, la ordenación según los 2^{i+1} se puede hacer en tiempo O(nlog(n)). Para ello, es necesario que se puedan comparar los sufijos en tiempo constante O(1). Por este motivo se ha creado una estructura de datos que representa un sufijo, que consta de 3 elementos. En primer lugar, su índice, y además de éste, 2 valores, ranking primero y ranking siguiente. Estos dos valores guardan el ranking de la primera mitad del sufijo y de la segunda mitad respectivamente.

Mediante estos valores se puede saber si un sufijo es mayor o menor que otro en orden alfabético, y al ser dos enteros, se puede hacer la comparación en tiempo constante. Dados dos sufijos, se compara el valor de su ranking primero, si son distintos el que tenga menor valor es menor. Si son iguales se compara el segundo, siguiendo el mismo criterio que en el caso del primer ranking.

Para la ejecución del algoritmo es necesario el uso de dos estructuras, un vector de Sufijos, con la estructura explicada anteriormente, el cual se usa para realizar las ordenaciones, y un vector de enteros, el cual guarda los rankings para todos los sufijos en cada iteración. De esta manera, se comienza rellenando el vector de enteros, con el valor ASCII de la letra inicial de cada sufijo, que hace las veces de primer ranking. Posteriormente, se calcula el número de iteraciones, que corresponde con el logaritmo en base 2 de N por arriba, siendo N el número total de caracteres de la cadena.

El siguiente paso es realizar el bucle para el número de iteraciones calculado. En cada paso, se genera la lista de los N sufijos, con el índice y su ranking inicial correspondiente, tomado de la lista de rankings. Además de esto, se debe calcular el ranking segundo, que hace referencia a la segunda mitad del sufijo. Para ello, se debe tomar como ranking segundo el ranking del sufijo a distancia: $sufijo_i + 2^{num\_iteracion}$, siendo i el sufijo que se está evaluando. Esto se debe a que, dado un sufijo i, si se ordena por los first primeros caracteres, $suffix_i$ = primeros first/2 caracteres de $suffix_i$ + first/2 caracteres de $suffix_{i+2^{n\_iteracion}}$.

Como ejemplo, para el caso inicial, al evaluar el sufijo 0, se toma su ranking y se calcula el ranking de $0+2^0$, lo que es igual a 1. Para la segunda iteración, el ranking actual hace referencia a la ordenación según dos caracteres, y se busca ordenar según 4 caracteres. Para ello, en el sufijo primero(índice 0) se toma como segundo ranking el ranking del sufijo número($0+2^1=2$).

En caso de que ese número de sufijo supere el total de sufijos, se coloca -1. Una vez generados ambos rankings, se hace la ordenación de los sufijos y una vez realizada ésta, se actualiza la lista de enteros con los rankings de cada sufijo. Se toma la lista de sufijos ordenada, se comienza poniendo el ranking del menor como 0, y a partir de aquí se aumenta el número de ranking si los rankings del sufijo aumentan. Finalmente, se devuelve el orden según han quedado los elementos en la lista L.

El coste del algoritmo es $O(nlog(n)^2)$, puesto que la ordenación se llama log(n) veces y el coste de la ordenación es O(nlog(n)). Como algoritmo de sort se ha elegido el std::sort de c++. Se podría mejorar el algoritmo utilizando un radix sort, el cual puede ordenar con coste O(n), quedando finalmente un coste total O(nlog(n)), sin embargo, aunque el coste asintótico es mejor, el sort de C++ está muy optimizado y sería difícil mejorarlo con una implementación normal de radix sort.

Transformada de Burrows-Wheeler

Las implementaciones han sido desarrolladas siguiendo el documento publicado por Burrows y Wheeler en May 10, 1994: https://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-124.pdf.

Para calcular la transformada de Burrows-Wheeler dada una cadena se obtiene el vector de sufijos de la cadena duplicada, es decir, dada una cadena c1, c2, c3 … cn, se obtiene el

vector de sufijos de la cadena c1, c2, c3 … cn, c1, c2, c3 … cn. Esto se realiza para que al ordenarse los sufijos se tenga en cuenta el contexto, ya que por ejemplo en la cadena "decode" si no se tiene en cuenta el contexto, el sufijo "e" iría antes que el sufijo "ecode".
A continuación, sólo se consideran los sufijos pertenecientes a la cadena original, es decir, aquellos cuyo índice es menor a la longitud de la cadena. Finalmente, mediante un bucle que recorre los índices del vector de sufijos, se va asignando a cada elemento de la transformada el carácter de la cadena original cuya posición es el índice del sufijo actual - 1. Para el caso cuyo índice es 0 se asigna a la posición actual de la transformada el último carácter de la cadena original, y se guarda la posición del vector de sufijos como valor I. Por ejemplo, para la cadena "decode" y el sufijo "code" se asignaría en la posición actual de la transformada el carácter "e" porque es el carácter anterior a "code".

El coste del algoritmo para calcular la transformada es $O(2n\log(2n)^2)$ del cálculo del vector de sufijos + $O(2n)$ para el bucle. Es decir, un coste de $O(n\log(n)^2)$.

Para realizar la inversa de la transformada se ha utilizado el algoritmo proporcionado por Burrows-Wheeler que deshace los pasos anteriores para obtener la transformada. Este algoritmo tiene coste $O(n)$.

Move-To-Front

Para realizar el algoritmo Move-To-Front se ha seguido lo detallado en Wikipedia: https://en.wikipedia.org/wiki/Move-to-front_transform.

Este algoritmo emplea un diccionario con todos los posibles valores del carácter (256 ya que es 1 byte) y recorre mediante un bucle la cadena de entrada, guarda la posición del diccionario donde está almacenado el carácter, y elimina del diccionario dicho carácter para introducirlo en la última posición. Finalmente devuelve las posiciones que se han ido obteniendo. El coste de este algoritmo es $O(n)$.

Para obtener la cadena original, deshace las operaciones realizadas en el algoritmo anterior mediante otro bucle y utilizando un método de diccionario similar. Este algoritmo también tiene coste $O(n)$.

Huffman

Se ha utilizado una implementación del algoritmo de Huffman desarrollada en el curso pasado en la asignatura Algoritmia Básica, en la que mediante un Montículo y una tabla de frecuencias devuelve la cadena comprimida. El coste de esta implementación es $O(n\log(n))$.

Programa completo

El programa completo utiliza las técnicas anteriores para comprimir y descomprimir cualquier fichero. Para ello se descompone el fichero en bloques de 64KiB que se van copiando a memoria.

Para la compresión se calcula la transformada de Burrows-Wheeler en primer lugar para colocar los bytes idénticos en posiciones contiguas. A continuación, se emplea la técnica de Move-To-Front para convertir los bytes más repetidos en el mismo valor. De esta manera un algoritmo como Huffman que comprime en función de la frecuencia de cada byte tendrá mejores resultados y será capaz de comprimir mejor. Finalmente, se aplica Huffman para comprimir la cadena transformada.

El coste de la compresión es $O(nlog(n)^2)$.

Para la descompresión se realizan los pasos opuestos, es decir, primero se descomprime mediante Huffman. A continuación, se revierte el algoritmo Move-To-Front, y finalmente se deshace la transformada de Burrows-Wheeler obteniendo la cadena original.

El coste de la descompresión es $O(nlog(n))$.

Pruebas realizadas

Se ha comprobado que el programa funciona correctamente en Hendrix, pero los resultados que se muestran a continuación se han ejecutado en el sistema que se muestra a continuación:

- SO: x86_64 GNU/Linux 5.3.0-24-generic Ubuntu 19.10
- CPU: Intel® CoreTM i5-8250U
- gzip 1.10
- bzip2-1.0.6

10kb.txt	Tamaño (Bytes)	Tamaño Comprimido (Bytes)	Tiempo en comprimir (segundos)	Tiempo en descomprimir (segundos)	Ratio de compresión
Practica2	9510	3213	0,048	0,007	0,3379
Huffman	9510	6142	0,0093	0,007	0,6458
Gzip	9510	3035	0,004	0,003	0,3191
Bzip2	9510	2544	0,0076	0,004	**0,2675**

100kb.txt	Tamaño (Bytes)	Tamaño Comprimido (Bytes)	Tiempo en comprimir (segundos)	Tiempo en descomprimir (segundos)	Ratio de compresión

Practica2	102180	26843	0,468	0,015	0,2627
Huffman	102180	55881	0,025	0,013	0,5469
Gzip	102180	27036	0,02	0,006	0,2646
Bzip2	102180	14552	0,0236	0,003	**0,1424**

aa.txt	Tamaño (Bytes)	Tamaño Comprimido (Bytes)	Tiempo en comprimir (segundos)	Tiempo en descomprimir (segundos)	Ratio de compresión
Practica2	122242	15396	0,411	0,016	0,1259
Huffman	122242	15487	0,011	0,01	0,1267
Gzip	122242	602	0,006	0,005	0,0049
Bzip2	122242	70	0,008	0,0055	**0,0006**

pg45788.txt	Tamaño (Bytes)	Tamaño Comprimido (Bytes)	Tiempo en comprimir (segundos)	Tiempo en descomprimir (segundos)	Ratio de compresión
Practica2	1089647	416243	4,764	0,154	0,382
Huffman	1089647	623128	0,165	0,123	0,5719
Gzip	1089647	414880	0,0873	0,017	0,3807
Bzip2	1089647	312311	0,1306	0,1033	**0,2866**

quijote.txt	Tamaño (Bytes)	Tamaño Comprimido (Bytes)	Tiempo en comprimir (segundos)	Tiempo en descomprimir (segundos)	Ratio de compresión
Practica2	2198927	816623	9,437	0,283	0,3713
Huffman	2198927	1248343	0,3176	0,2596	0,5677
Gzip	2198927	813719	0,1946	0,027	0,3701
Bzip2	2198927	602700	0,243	0,1996	**0,2741**

Como se puede apreciar, la versión de Bzip2 comprime mejor que la desarrollada en esta práctica, pero utilizar sólo Huffman sin ninguna transformada es claramente peor. Por tanto si se realizaran más optimizaciones se conseguirían unos resultados más similares a las versiones comerciales.