ALGORITMIA PARA PROBLEMAS DIFÍCILES

MEMORIA PRÁCTICA 2

Curso 2019-2020

Víctor Peñasco Estívalez 741294
Rubén Rodríguez Esteban 737215

Tabla de contenidos

Introducción

En esta práctica se han implementado una serie de algoritmos para crear un compresor similar a bzip2. Los algoritmos implementados son: creación de un vector de sufijos, transformada de Burrows-Wheeler y Move to front, además de algunos procedimientos inversos para el proceso de descompresión.

Para llevar a cabo esta implementación se ha utilizado el lenguaje de programación Java, debido a que ofrece una gran variedad de estructuras de datos en sus principales librerías, además de tratarse de un lenguaje compilado, lo cual supone una gran disminución del tiempo de ejecución en comparación con lenguajes interpretados como Python.

Diseño e implementación

Como punto de partida, se ha introducido un carácter especial al final del texto que indica el final de la de entrada. Este carácter elegido ha sido el carácter de control ETX cuyo código ASCII es 0x03. Las razones por las que se ha escogido dicho carácter estriban en que dicho carácter tiene como significado literal final del texto, y además, que el código ASCII del carácter toma un valor muy bajo, haciendo que posteriormente en la ordenación siempre sea el carácter con menor valor de la cadena.

Vectores de sufijos

Para la construcción del vector de sufijos, primeramente se realizó una implementación ingenua que ofrecía una complejidad temporal de orden $O(n^2 \log n)$, siendo n el número de caracteres de la entrada (por ejemplo en un fichero de texto). Ante la notable complejidad del algoritmo, tras realizar diversas pruebas empíricas, se optó por modificar el algoritmo en cuestión con el fin de lograr reducir el coste temporal. Dicho algoritmo se denomina algoritmo de Skew, que ofrece una complejidad temporal para formar el vector de sufijos del orden $O(n)$.

Este algoritmo comienza creando 3 grupos de sufijos: S0, S1 y S2, de manera que cada sufijo en el grupo Sk empieza en el índice $3x + k$, para algún valor x.

Se construye S12, que se compone de trigramas (los 3 primeros caracteres) de los sufijos de los grupos 1 y 2. Este vector S12 se ordena en tiempo lineal utilizando el algoritmo radix sort.

Cada trigrama recibe como nombre su rango tras la ordenación. Si hay algún rango repetido (había algún trigrama repetido), entonces se vuelve a aplicar el algoritmo entero sobre la cadena conformada por los nombres (rangos de los bloques). Si los rangos de los bloques originales eran [4, 6, 2, 0, 3, 5, 1], la nueva cadena sobre la que aplicar recursivamente el algoritmo es “4620351”. Esta recursión hace que la cadena sea cada vez más pequeña (en el caso de que siguiera habiendo duplicados), y tal como se especifica al final de esta explicación, conserva la complejidad lineal del algoritmo.

Una vez S12 está ordenado, se ordena S0. Para esto se crea un tupla de dos elementos, conformada por la primera letra del sufijo y el rango del resto del sufijo. Estos rangos se pueden obtener directamente de los rangos de los bloques de S12. Una vez los sufijos se ha mapeado a estas tuplas de dos elementos, se pueden ordenar en tiempo lineal.

En este momento se encuentran los grupos S1 y S2 ordenados en una misma lista y el grupo S3 ordenado. El objetivo es hacer la mezcla de las listas en tiempo lineal.

Para esto se establece para cada sufijo (tanto en S12 y S0) pares y/o triples. Los pares se conforman concatenando el primer carácter del sufijo con el rango del resto del sufijo, y los triples concatenando los dos primeros caracteres con el rango del resto del sufijo. Que un sufijo posea par, triple o ambos depende de lo siguiente:

- Un sufijo de S0 puede formar un par tomando un carácter y un rango de S1, y un triple tomando dos caracteres y un rango de S2.
- Un sufijo de S1 puede formar un par tomando un carácter y un rango de S2.
- Un sufijo de S2 puede formar un triple tomando dos caracteres y un rango de S1.

Esto es así porque no se poseen rangos de S0, por lo que los sufijos de S1 solo pueden mirar en S2 para conocer rangos, y viceversa.

La complejidad temporal de este algoritmo, teniendo en cuenta la recursión, se rige por la siguiente ecuación: T(n) = T(2n/3) + O(n). Esta complejidad se traduce en O(n).

Para la implementación de este complejo algoritmo se ha utilizado código de una librería pública en Java, con ligeras modificaciones, la cual se puede encontrar en el apartado de referencias.

Transformada de Burrows-Wheeler

Para llevar a cabo la transformada de Burrows Wheeler se ha procedido tal y como se explica a continuación(véase figura 1). Siendo BWT la cadena que almacena la transformada, S el vector de índices de sufijos y X la cadena de entrada, el i-ésimo carácter, BWT[i], se corresponde con el carácter de la cadena original que se encuentra en la posición indicada por el sufijo *i -1* del vector de sufijos, es decir, BWT[i] = X[S[i - 1]].

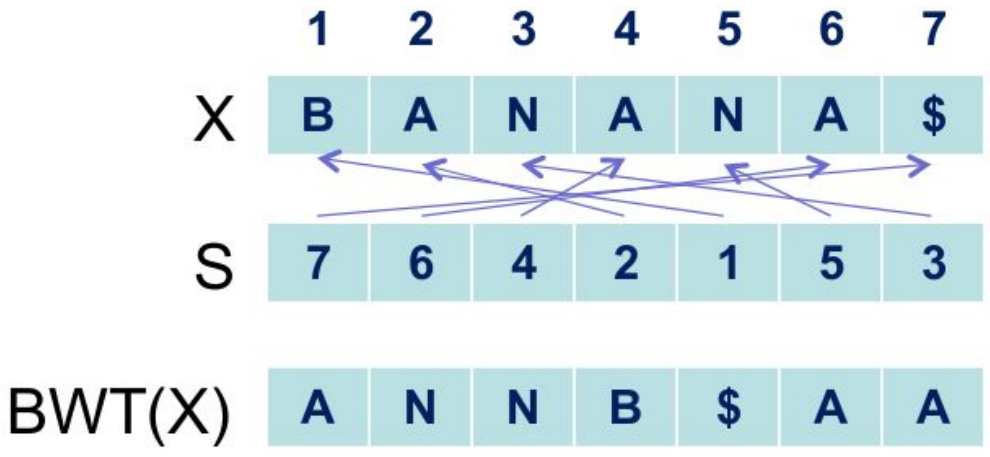

Figura 1
Esquema del algoritmo para la transformada de Burrows-Wheeler

Move to front

El algoritmo Move to front (véase figura 2) recibe como entrada una secuencia de caracteres, concretamente la transformada de Burrows-Wheeler, calculada tal y como se ha descrito en el apartado anterior. El algoritmo cuenta de entrada con un alfabeto que recoge todos los símbolos que pueden ser representados, siendo dicho alfabeto, el conjunto de todos los caracteres del código ASCII.

Como salida el algoritmo devuelve una secuencia de valores numéricos que representan las posiciones de los caracteres de entrada en el alfabeto predefinido. En otras palabras, para cada uno de los caracteres de la entrada, se obtiene su posición en el alfabeto, la cual, se añade a la salida. El carácter en el alfabeto se mueve al frente de éste. La secuencia de enteros retornada por el algoritmo ha sido guardada en un fichero de texto, de manera que los distintos enteros que constituyen dicha secuencia se hallan separados por espacios en blanco.

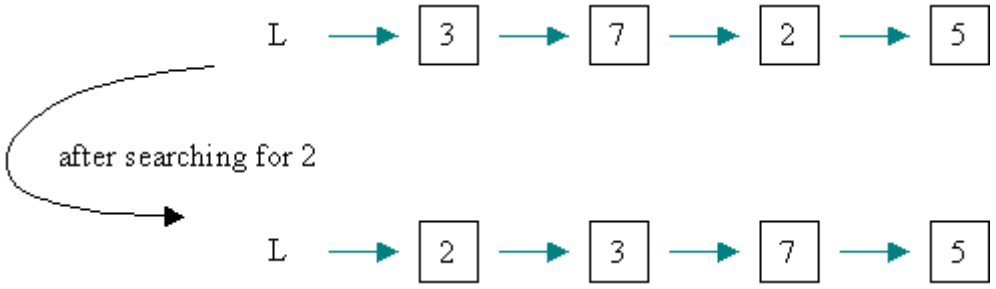

Figura 2
Esquema del algoritmo Move to front

Huffman

Con el objetivo de llevar a cabo los procesos de compresión y descompresión de ficheros se ha empleado el algoritmo basado en códigos Huffman. Para llevar a cabo la compresión de archivos Huffman recibe como entrada un fichero de texto donde se almacena la salida proporcionada por el algoritmo Move to front descrito en el apartado anterior, generando así un fichero comprimido.

Para la descompresión, Huffman recibe como parámetro el fichero que se desea descomprimir, proporcionando como salida el fichero resultante a la salida de Move to Front.

Como algoritmo de Huffman (véanse referencias) se ha escogido una implementación escrita en lenguaje C++ desarrollada para la asignatura de *Algoritmia Básica 2018-19* cuyos autores fueron José María Vallejo y Rubén Rodríguez. Sobre este programa se han realizado una serie de pequeñas modificaciones para adaptarlo a esta práctica. Este

algoritmo está implementado en el lenguaje C++. La integración con el código en Java para formar el compresor y descompresor se describe en el apartado de integración.

Move to front inverso

El algoritmo de Move to front inverso sigue un procedimiento muy similar al original. En primer lugar se ha inicializado el alfabeto con todos los caracteres del código ASCII. Posteriormente se procede a leer cada código numérico del fichero de texto, generado tras efectuar la descompresión con el algoritmo de Huffman, de manera que se obtiene el carácter del alfabeto almacenado en la posición indicada por el carácter leído. Tras este proceso, se desplaza el carácter leído al frente del alfabeto.

Los correspondientes caracteres obtenidos tras aplicar el proceso anterior son almacenados en un fichero de texto de forma que dicha cadena de texto resultante se corresponde con la transformada de Burrows-Wheeler del texto original.

Transformada inversa de Burrows-Wheeler

Para recuperar el texto original a partir de la transformada de Burrows-Wheeler se ha seguido el algoritmo (véase figura 3) explicado a continuación.

Recorriendo la cadena transformada, se guarda utilizando un vector (del tamaño del alfabeto) el número de ocurrencias de cada carácter. Al mismo tiempo, en otro vector (indexI), del tamaño de la cadena, se almacena la j-ésima ocurrencia del carácter i, es decir, cuántas veces ha aparecido ya el carácter en la posición i de la transformada.

Luego, recorriendo el vector del tamaño del alfabeto, se almacena en cada posición cuántas ocurrencias de caracteres más pequeños existen.

Se vuelve a recorrer la transformada creando un vector (C), que guarda en la posición i el número de caracteres más pequeños hay que el carácter en la posición i de la transformada. Esto se hace de forma directa ya que la información ya está almacenada en el vector creado en el paso anterior.

Se crea un último vector (LF) del tamaño de la transformada, que almacena en la posición i la suma del entero en la posición i del vector C y el entero en la posición i del vector indexI. El resultado de esta suma se decrementa en una unidad para adaptarse al hecho de que la posición inicial de un vector sea la posición 0.

Finalmente, para recuperar el texto original, se utiliza un bucle con n - 1 iteraciones. Esto se hace para no recuperar el último carácter de la cadena original, el cual se corresponde con el carácter de control ETX que fue añadido en el proceso de compresión. En cada iteración, se añade a la cadena resultante el carácter que se encuentra en la posición r de la transformada. Esta r se inicializa a 0 y tras cada iteración toma el valor LF(r).

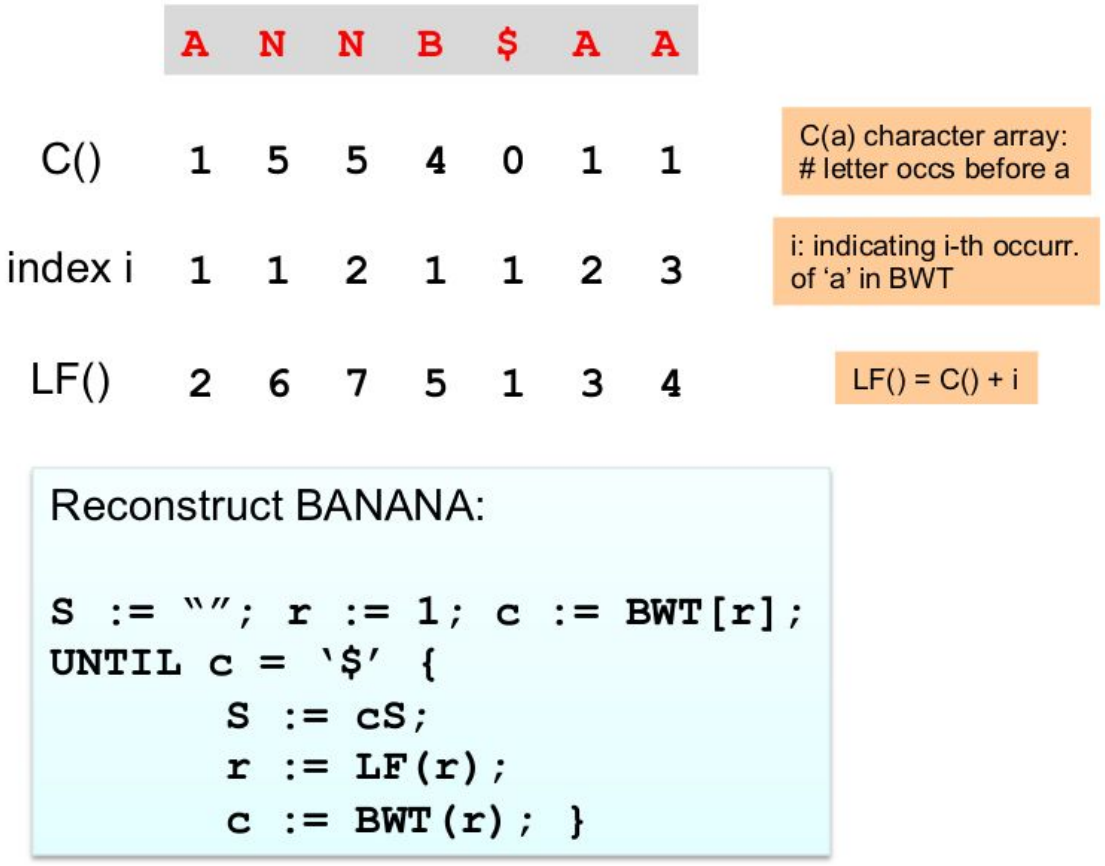

Figura 3
Esquema del algoritmo para la transformada de Burrows-Wheeler inversa

Integración: bzip3

El objetivo final de la práctica es crear un compresor similar a bzip2 que utilice los algoritmos de BWT, Move to front y Huffman. Como se ha utilizado código en Java para la realización de los dos primeros algoritmos, y código legado en C++ para Huffman, se ha utilizado un script en bash para la integración de ambos códigos.

Este script se ha denominado *bzip3.sh*. Para comprimir, recibe como entrada la ruta de fichero de texto, y genera un fichero comprimido del mismo nombre pero con extensión .bz3. Para la descompresión, se recibe un fichero .bz3 y recupera el archivo original.

Análisis del coste

Compresión

Para conocer el coste de la compresión de un fichero, primero hay que analizar la complejidad temporal de cada una de las partes del proceso por separado:

1. Crear vector de sufijos: al implementar el algoritmo de Skew, se obtiene un coste de orden O(n), siendo n el número de caracteres de la cadena de entrada.

2. Transformada de Burrows-Wheeler: al utilizar vectores de sufijos para la construcción de la transformada se consigue un coste de orden O(n).

3. Move to front: el algoritmo move to front, al utilizar un alfabeto en la que la búsqueda de un elemento en el alfabeto tiene caso peor coste O(m), supone un coste total en caso peor O(m * n), aunque en la práctica el coste es menor, ya que los caracteres más frecuentes se encuentran en las primeras posiciones de la lista enlazada sobre la que se realiza la búsqueda.

4. Compresión Huffman: la complejidad del algoritmo de Huffman se calcula en base a las operaciones mostradas a continuación.

    4.1. Leer fichero y contar para cada carácter el número de veces que aparece tiene un coste O(n) siendo n el número de caracteres del fichero.

    4.2. Crear un montículo con prioridad de máximos (los caracteres con mayor frecuencia al principio para asignar un código de compresión menor) tiene un coste O(m * log m), dado que es necesario recorrer el vector de todas las tuplas <carácter, frecuencia> e insertarlas en el montículo, siendo m el número de caracteres en el alfabeto, es decir, 256.

    4.3. La construcción del árbol tiene un coste O(m * log m) ya que se deben eliminar las dos tuplas <carácter, frecuencia> con mayor frecuencia del montículo para formar un nuevo nodo en el árbol y repetir este proceso hasta que el montículo quede vacío siendo m el número de elementos del montículo.

    4.4. Generar los códigos de codificación binarios para cada carácter tiene un coste de O(m).

    4.5. Guardar el árbol en el fichero comprimido con recorrido en preorden tiene un coste O(m).

    4.6. Guardar el contenido del fichero comprimido tiene un coste O(n).

La complejidad temporal del proceso de compresión en su conjunto tiene un coste de O(n + m log m).

Descompresión

Para conocer el coste de la descompresión de un fichero comprimido mediante bzip3, primero hay que analizar la complejidad temporal de cada una de las partes proceso por separado:

1. Descompresión Huffman: la complejidad del algoritmo de Huffman se calcula en base a las operaciones mostradas a continuación.

    a. Recuperación del árbol tiene un coste de O(m) siendo m el tamaño del alfabeto.

    b. Escribir el fichero descomprimido tiene un coste de O(n * log m) siendo n el número de caracteres del fichero y m el número de caracteres del alfabeto.

    La complejidad temporal del proceso de compresión en su conjunto tiene un coste de O(n * log m).

2. Move to front inverso: en este caso, el acceso a los caracteres de la lista enlazada que representa el alfabeto es constante, ya que se accede de forma directa conociendo la posición (entrada de este algoritmo). Por tanto este procedimiento tiene un coste de orden O(n).

3. Transformada inversa de Burrows-Wheeler: tal y como se ha explicado anteriormente, el algoritmo seguido para la recuperación del texto original a partir de la transformada realiza más de un bucle, pero todos ellos tienen coste lineal en el tamaño de la entrada o en el tamaño del alfabeto. Esto se traduce en que el coste de realizar la inversa de la transformada es de orden O(n + m).

La complejidad temporal del proceso de descompresión en su conjunto es O(m + n * log m).

Pruebas

Para comprobar la correcta implementación de los algoritmos, se ha creado un script denominado *ejecutar2.sh* que automatiza la ejecución de los casos de prueba. Cada caso de prueba ha consistido en comprimir un fichero y, acto seguido, descomprimirlo. Por último se verifica que el fichero original y el fichero resultante del proceso anterior son iguales. La siguiente tabla muestra todos los ficheros con los que se ha probado el compresor, mostrando tanto el tamaño original del fichero como el del fichero comprimido, el ratio de compresión, y los tiempos que ha tardado en efectuar la compresión y descompresión, respectivamente.

Caso de prueba	Fichero utilizado	Tamaño original	Tamaño comprimido	Ratio de compresión	Tiempo de compresión (en segundos)	Tiempo de descompresión (en segundos)
1	makefile	0.89 KB	0.5 KB	0.562	0.099 s	0.097 s
2	profitChecker.sh	1.9 KB	1.1 KB	0.579	0.101 s	0.095 s
3	timeChecker.py	3 KB	1.2 KB	0.4	0.100 s	0.095 s
4	fields.c	11 KB	6 KB	0.546	0.132 s	0.153 s
5	Compilador.java	79 KB	34 KB	0.43	0.288 s	0.278 s
6	lcet10.txt	426 KB	228 KB	0.535	0.563 s	0.465 s
7	plrabn12.txt	481 KB	282 KB	0.586	0.580 s	0.494 s
8	cornellBox.ppm	1917 KB	1013 KB	0.528	1.586 s	2.648 s
9	Shakespeare.txt	5458 KB	3000 KB	0.549	8.794 s	9.150 s
10	big.txt	6488 KB	3527 KB	0.544	11.920 s	10.889 s
11	caustica.ppm	8107 KB	3806 KB	0.469	9.949 s	11.209 s

Nota: las métricas anteriores han sido ejecutadas en local y no en hendrix. Si se ejecutan en Hendrix los tiempos saldrán superiores dada la calidad del servicio.

De manera adicional, también se ha optado por probar con ficheros de tamaños grandes. Sin embargo, el compresor diseñado puede fallar con este tipo de ficheros ya que dependiendo de la configuración de la Máquina Virtual de Java, el espacio de memoria que tiene Java asignado para el heap se puede sobrepasar. Como posible solución a este problema se había pensado en leer el fichero por bloques de tal forma que se evitaría desbordar la memoria del heap.

Organización del trabajo

Tarea realizada	**Horas dedicadas (por integrante de equipo)**	
	Víctor Peñasco	Rubén Rodríguez
Vectores de sufijos	5	5
Transformada de B-W	2	2
Move to front	0.5	0.5
Huffman	1	1
Move to front inverso	0.5	0.5
Transformada inversa de B-W	2.5	2.5
Integración	2	2
Pruebas	4	4
Memoria	5	5
Total	**22.5**	**22.5**

Referencias

Simple Linear Work Suffix Array Construction - https://www.cs.helsinki.fi/u/tpkarkka/publications/icalp03.pdf

Implementación Skew en Java - https://github.com/carrotsearch/jsuffixarrays/blob/master/src/main/java/org/jsuffixarrays/Skew.java

Move to Front transform - https://en.wikipedia.org/wiki/Move-to-front_transform

Algoritmos para strings - http://webdiis.unizar.es/asignaturas/Bio/wp-content/uploads/2015/05/4CStrings3.pdf

Algoritmos de Compresión - http://webdiis.unizar.es/asignaturas/APD/wp/wp-content/uploads/2013/09/191204compression2.pdf

Suffix Arrays and BWT - https://web.stanford.edu/class/cs262/presentations/lecture5.pdf

Prácticas de Algoritmia Básica - https://github.com/ZgzInfinity/AlgoritmiaBasica

The Skew (Linear Time) Algorithm for Suffix Array Construction - https://gist.github.com/markormesher/59b990fba09972b4737e7ed66912e044