Tanvir Hossain
Grupo 8.1

ALGORITMO LZ78

DESCRIPCIÓN

El algoritmo LZ78 es un algoritmo basado en la compresión y descompresión de textos sin pérdida de información. Una de las principales motivaciones detrás de LZ78 fue crear un algoritmo de compresión universal que no requiera ningún conocimiento sobre la entrada. El algoritmo es simple y tiene el potencial de un rendimiento muy alto en implementaciones de hardware. Es el algoritmo de la herramienta de compresión de archivos Unix, ampliamente utilizada, y se utiliza en el formato de imagen GIF.

PROCESO DE CODIFICACIÓN

Se inicializa un diccionario con un string vacío en la posición 0. Se leen los símbolos de entrada y se va añadiendo en el diccionario en la posición 1, 2, 3 , etc. Por cada símbolo X leido, se inspecciona en el diccionario para buscar una entrada que tenga X como primer símbolo de la secuencia. Puede ocurrir:

- Si no se encuentra ninguna coincidencia en el diccionario , X se agrega a la siguiente posición disponible en el diccionario y el token (0, 'X') se escribe en la salida.
- Si se encuentra el símbolo X en el diccionario, se lee el siguiente símbolo Y. Se concatena los símbolos X+Y y se vuelve a inspeccionar nuevamente para cada entrada que contenga estos símbolos concatenados. Este procedimiento se repite hasta que no encuentra una secuencia de símbolos que rompa la coincidencia. Por tanto, se volveríamos al punto anterior (es decir, esta secuencia se agrega a la siguiente posición disponible en el diccionario y se escribe un token en la salida).

Para tener claro este proceso a continuación explico el pseudocódigo de la implementación que he realizado:

diccionario := vacío; prefijo := vacío; índice := 1
mientras haya carácter en el flujo de entrada:

    carácter := carácter leído

    si prefijo+carácter existe en el diccionario:

```
            prefijo := prefijo+carácter
        sino:
            si prefijo es vacío: índicePrefijo := 0
            sino: índicePrefijo := coger el índice del prefijo
            escribir token(índicePrefijo, carácter)
            añadir en diccionario(índice, prefijo+carácter)
            ++índice
            prefijo = vacío
        fin
fin
si prefijo no es vacío:
        indicePrefijo := coger el índice del prefijo
        escribir token(índicePrefijo, )
fin
```

PROCESO DE DECODIFICACIÓN

El proceso de decodificación se realiza de la misma manera que el proceso de codificación. Se inicializa un diccionario vacío. En cada paso se lee un token de la entrada (I, C) = (índice, carácter). Puede ocurrir:

- Si el token es (nulo / 0, C), C se extrae del segundo campo, se escribe en la salida y el token se convierte en una entrada del diccionario.
- Si el token es (índice, secuencia), el decodificador extrae la secuencia del segundo campo y, con el índice, se sabe dónde está la referencia del siguiente token del que se extrae la secuencia para concatenar al anterior. La secuencia lograda se escribe en la salida y este último se agrega al diccionario como un nuevo token.

Para clarificar esta idea a continuación explico el pseudocódigo de la implementación que he realizado:

```
diccionario := vacío; índice := 1
mientras se pueda leer un token (palabraCodificada, C):
        i := palabraCodificada
```

carácter := C

si i = 0

string = vacio

sino

string := coger string de índice de palabraCodificada en diccionario

escribir en la salida: string+caracter

añadir en diccionario(índice, string+carácter)

++índice

VENTAJA

Las salidas generadas por el codificador se componen de dos campos: un puntero hacia el diccionario y un código de símbolo. Cada token corresponde a una secuencia de símbolos de entrada que se almacena en el diccionario solo después de que se haya escrito en el archivo de salida. Un token insertado en el diccionario no se puede eliminar y esto representa una ventaja ya que las secuencias futuras pueden comprimirse a partir de las secuencias más antiguas (prefijo).

INCONVENIENTE

Sin embargo, en el proceso de compresión el índice tiende a crecer y aumentar su tamaño rápidamente. El tamaño del diccionario está limitado a la mayor parte de la memoria completa de la máquina y esto implica un límite para el número de bits utilizados para el token del diccionario.

DIFICULTADES Y TOMA DE DECISIONES

En la implementación he usado HashMap como diccionario porque usa tablas de hash y es el almacenamiento de búsqueda más rápida.

Declaramos un diccionario HashMap<String,Integer>, donde el primer campo representa la secuencia de caracteres y el segundo el índice. Como he comentado anteriormente en cada iteración el índice va creciendo y el diccionario tendría un límite para representar un cierto índice, en este caso como máximo (2^32)-1. Pero también representa otro problema a la hora comprimir y escribir el token en un fichero

comprimido(si escribo en 1B, 2B...). Tendré que escribir un token (índice,carácter) cuando no encuentre coincidencia. El carácter podría codificar con un byte pero el indice puedo hacerlo con 2B o 4B, pero siempre tendre un limite para escribir ese índice “correcto” tanto en la salida como en el diccionario. Al principio decidí escribir el índice como un integer (4B), pero pago un coste muy alto (p.e: si tengo que escribir el índice 1 podré escribirlo con un bit, si decido escribir en 4B pago 31 bits de extra). Problema que he encontrado a la hora de implementar la compresión: nosotros suponíamos que nos llegaba una ruta inicial que representaba leer el contenido del archivo y una ruta final para dejar el resultado comprimido. FileReader no facilitaba la lectura y escritura de archivos. Tanto en caso de escribir un índice y como un carácter usaba el método write(). Cuando hacia write(índice) (donde índice es integer) internamente coge los 16 bits inferiores y los 16 bits de parte alta del índice ignora. Por tanto, si el índice supera (2^16) -1 habrá desbordamiento. Por otra parte, write(carácter), no lo escribe en 1B sino en 2B. Java considera un carácter como 2B. Resumiendo, la primera versión (LZ78_v1) que he implementado me funciona pero estoy teniendo problema de escribir el índice en 2B cuando queria imprimir en 4B y el carácter en 2B cuando se puede representar en 1B.
En la implementación de la segunda version (LZ78_v2) pudé superar los problemas anteriores. Gracias a RandomAccessFile se podía escribir en byte, short, integer o long. Esta vez, he tomado la decisión de comprimir o escribir el carácter (1B), el índice en short (2B) y no en 4B ya que he observado si el texto a comprimir no es muy grande (<500kB), al comprimir no se reduce mucho el tamaño e incluso a veces supera ese tamaño actual del fichero al comprimir. Debido a esto y considerando que estamos en un contexto académico donde los textos no serán tan grandes he decidido escribir los índices en 2B, al principio pagaré unos bytes pero luego los recuperaré. Con 2B la reducción de tamaño del fichero es notable. En esta versión he tenido la dificultad de pasar el código de la versión 1 y solucionar los errores encontrados (no me descomprimía bien) ya que tenía que conocer bien la nueva librería de RandomAccessFile y sus métodos. Por otra parte, debido la toma decision del grupo ahora el método comprimir(...) tenia que devolver un parámetro y en descomprimir(...) se recibira unos parametros adicionales. Lo cual he tenido que modificar un poco el código y me ha quedado compacto respecto a la versión 1.

Antes de la entrega, hemos sabido que las funciones del algoritmo no puede recibir unas rutas y acceder directamente al archivo ya que estábamos violando la arquitectura en tres capas. Por tanto, la estructura del programa en global ha cambiado un poco. Ahora el algoritmo recibirá un objeto file y otros parámetros, tendrá que devolver un estructura de datos (buffer, vector..) pero nunca escribir directamente en un fichero. Esto ha implicado que modificar un poco el código.

PROBLEMA

El algoritmo implementado presenta un problema. Optar por un índice que se pueda representar en 2B implica que no se puede comprimir bien los textos grandes. Por tanto, cuando supere el índice 65535 y tenga que escribirlo en short habrá desbordamiento y el índice no será correcto. Esto se puede solucionar cogiendo índice como int o long pero tamaño final del comprimido será más grande. Pero estos también tienen sus límites para representar. De cara a la segunda entrega tengo pensado a escribir el índice como integer y intentaré solucionar el problema de escritura en la salida que no ocupe mucho.

SOLUCIONES POSIBLES

1. Guardar el contexto del diccionario o lo que voy a escribir en la salida (congelar) y usar siempre las mismas entradas (diccionario estático).
2. Eliminar todo el diccionario (restablecer) y comenzar con un nuevo diccionario.
3. Elimine las entradas utilizadas menos recientemente (LRU) para insertar otras nuevas entradas.

Investigando las diferentes soluciones propuestas por distintos autores. He elegido implementar la segunda solución. Entonces cuando el índice supera su límite (65535) vuelvo a redefinir el diccionario. Con esta idea he podido resolver el problema que tuve en la primera entrega. Por otra parte, también mantengo la idea de escribir el índice como un short y no como un int, por razones que he explicado previamente. Sin embargo, si quisiera imprimir el índice como un int entonces el diccionario habría que redefinir a partir del valor 55300. He investigado para este caso y es muy peculiar que tenga que restablecer el diccionario con un valor inferior al anterior.

Por otra parte cabe mencionar que el código se ha tenido que adaptar a una nueva clase que hemos definido, MyByteCollection. Es decir, en la primera entrega teníamos una clase Pareja y en esta entrega hemos sustituido el funcionamiento de algunas funciones y adaptarlo en la nueva clase.

DESCRIPCIÓN DE LOS TEST REALIZADOS

1) Jp1.txt

Objetivo: Se comprueba un texto con pocos caracteres (simple)

Resultados: El ratio de compresión es de 2.33 ya que el tamaño final(42B) es más grande que inicial(18B) ya que hay pocos caracteres sin coincidencias.

2) Jp2.txt

Objetivo: Se comprueba con un texto bastante grande.

Resultados: El texto descomprimido es igual al original (antes fallaba, ahora no). El ratio de compresión es de 0.52. Tamaño inicial: 384013B Tamaño final:188954B

3) Jp3.txt

Objetivo: Comprobar que el algoritmo soporta la compresión de textos vacíos.

Resultados: El algoritmo funciona con normalidad. Escribe el último carácter del algoritmo ‘8’.

4) Jp4.txt

Objetivo: Comprobar que el algoritmo es capaz de comprimir y descomprimir textos simples escritos en codificación ASCII.

Resultados: El texto descomprimido es igual al original.

5) Jp5.txt

Objetivo: Comprobar que el algoritmo es capaz de comprimir y descomprimir textos con caracteres especiales(Ñ,ó,etc) de codificaciones distintas de ASCII (UTF-8,UTF-16,etc).

Resultados: El texto descomprimido es igual al original.

6) Jp6.txt

Objetivo: Comprobar que con textos que tienen poca repetición, el ratio de compresión es grande.

Resultados: El ratio de compresión es 0.96.

7) Jp7.txt

Objetivo: Comprobar que con textos que tienen mucha repetición, el ratio de compresión es pequeño:

Resultados: El ratio de compresión es 0.15.

RECURSOS UTILIZADOS

[1]https://en.wikipedia.org/wiki/LZ77_and_LZ78#LZ78
[2]http://www.stringology.org/DataCompression/lz78/index_en.html
[3]https://towardsdatascience.com/how-data-compression-works-exploring-lz78-87f44b487d92
[4]https://www.slideshare.net/DavideNardone/lz78-58791779
[5]https://w3.ual.es/~vruiz/Docencia/Apuntes/Coding/Text/02-string_encoding/03-LZ78/index.html
[6]http://defeo.lu/in420/dm-lz78
[7]http://compressions.sourceforge.net/LempelZiv.html
[8]http://voho.eu/wiki/algoritmus-lz78/