Índice

Estructuras de Datos & Algoritmos

LZSS

Compresión

Estructuras de Datos En la primera entrega para implementar la ventana deslizante que contiene los últimos n caracteres leídos del fichero usábamos un ArrayList de `Character` ya que es modificable en tamaño y también permite recorrerlo para encontrar coincidencias.

Para guardar los caracteres que tienen coincidencia hasta el momento usábamos también un ArrayList de `Character` por las misma razones que la ventana deslizante.

Sin embargo, para la segunda entrega hemos cambiado las EDs que usamos. La implementación de la ventana deslizante ahora la hacemos con un vector de `byte` de tamaña fijo n. Lo hacemos así ya que usamos el vector de forma circular y por lo tanto no tenemos la necesidad de poder añadir o borrar elementos cambiando el tamaño del vector y también hemos cambiado el tipo del *array* de `Character` a `byte` para poder comprimir cualquier fichero. Para saber dónde empieza y donde acaba el contenido de la ventana, nos guardamos el índice que apunta al último elemento y un booleano que nos indica si la ventana está llena.

En cuanto a las EDs que empleamos en esta entrega para guardar los `bytes` en la coincidencia actual usamos el mismo `array` de `bytes` que para la ventana deslizante pero con un tamaño predefinido de m, donde m es la longitud máxima de coincidencia que permitimos.

Además también queremos remarcar, como ya hemos mencionado, que ahora la ventana y los bytes de la coincidencia actual los guardamos en un vector de `byte` y eso nos permite ahorrarnos cada iteración del algoritmo un par de conversiones con *casting* de `char` a `int` y vice versa que antes hacíamos.

Algoritmos El único algoritmo que usamos en esta implementación de LZSS es Knuth-Morris-Pratt ya que aumenta un poco la eficiencia para encontrar coincidencias de un patrón dentro de la ventana deslizante y es el mismo que usábamos en la entrega previa. Sin embargo lo hemos adaptado para que funcione con `array` de `byte` y no con `ArrayList` de `Character` que usábamos antes. Además, al igual que en la otra entrega, lo hemos modificado para que nos dé el *offset* hacia atrás des del final de la ventana hasta donde empieza la coincidencia y no el índice de la coincidencia.

Descompresión

Estructuras de Datos Respecto a la primera entrega, los cambios son los mismos que en la compresión ya que también usamos una ventana deslizante. Ahora usamos un `array` de `byte` circular y previamente habíamos usado un `ArrayList` de `Characters`.

Algoritmos Para la implementación de la descompresión no se usa ningún algoritmo auxiliar de importancia en ninguna de las dos entregas.

LZ78

Compresión

Estructuras de Datos En la primera entrega el algoritmo LZ78 solo podia comprimir archivos de texto y para ello se utilizaba un HashMap con key String y value Integer, actuando de diccionario y de manera que todos los caracteres que se leian del archivo input quedaban guardados en este. No habia manera de controlar eficientemente el overflow ni el fin de archivo.

En la primera entrega el algoritmo LZ78 solo podia comprimir archivos de texto y para ello se utilizaba un HashMap con key String y value Integer, actuando de diccionario y de manera que todos los caracteres que se leían del archivo input quedaban guardados en este. No había manera de controlar eficientemente el overflow ni el fin de archivo.

Ahora, para poder comprimir todo tipo de archivos de manera óptima con el algoritmo LZ78, hemos utilizado lectura de bytes y un Tree compuesto por Nodos. Cada Nodo tiene un índice int, que indica el orden de inclusión en el árbol y por lo tanto el orden de aparición en el archivo que se desea comprimir, y 256 hijos ya que al realizar un lectura de tipo byte hay 256 posibles valores. Para controlar el Overflow así como la detección del final del archivo nos hemos decantado por añadir un bit delante de cada array de bytes codificado, de manera que la composición de la codificación de un byte consta de:

- Un bit. 0 indica que se continua sin problemas, 1 que a continuación hay un overflow o fin del archivo. Si el bit anterior era 0, el indice del padre del byte codificado. Si era 1, en caso de overflow un 0 y en caso de fin del archivo un 1. El byte codificado.

Cada codificación se guarda en un ArrayList hasta que éste se llena o se acaba el archivo, entonces se recorre y se escribe en el fichero comprimido. En caso de overflow se crea un nuevo Tree y un nuevo ArrayList. ### Descompresión

Estructuras de Datos En la primera entrega, al poder comprimir solo archivos de texto, el HashMap de descompresión en lugar de tener value ArrayList tenia String.

En la primera entrega, al poder comprimir solo archivos de texto, el HashMap de descompresión en lugar de tener value ArrayList tenia String.

En la segunda entrega, para descomprimir hemos utilizado en este caso un HashMap con key Integer y value ArrayList actuando como diccionario. Al descomprimir se lee primero un bit, que indica si hay overflow o fin de archivo, un numero, que indica una entrada del diccionario y a continuación se lee el byte del que va acompañado dicho numero. Finalmente se escribe en el archivo de descompresión la concatenación del value correspondiente a la entrada del diccionario junto con el byte leído.

LZW

Compresión

Estructuras de Datos En la primera entrega el algoritmo LZW solo podia comprimir archivos de texto por lo que el HashMap tenia como key String.

En la primera entrega el algoritmo LZW solo podia comprimir archivos de texto por lo que el HashMap tenia cómo key String.

Para comprimir en el algoritmo LZW hemos utilizado un HashMap con key ArrayList y value Integer, actuando de diccionario y de manera que todos los bytes que se leen del archivo input quedan guardados en este. El diccionario se inicializa con los valores unitarios esperados en el archivo a comprimir y a medida que se vaya leyendo la entrada se irán añadiendo posibles combinaciones de éstos. En caso de overflow del diccionario, se codifica en el archivo comprimido un int indicativo, de manera que al descomprimir se pueda saber en qué momento se ha reiniciado el diccionario. A diferencia de la primer entrega, en la cual todas las codificaciones de integers se representaban en 32 bits, en esta segunda entrega cada vez que se codifica un int se escriben los n bits representativos precedidos por este entero n representado en 5 bits, diminuyendo considerablemente el tamaño de la compresión.

Descompresión

Estructuras de Datos En la primera entrega, al poder comprimir solo archivos de texto, el HashMap de descompresión en lugar de tener value ArrayList tenia String.

En la primera entrega, al poder comprimir solo archivos de texto, el HashMap de descompresión en lugar de tener value ArrayList tenia String.

Para descomprimir hemos utilizado en este caso un HashMap inverso al de compresión, con key Integer y value ArrayList, actuando también como diccionario. Al igual que en la compresión el diccionario se inicializa con los valores unitarios esperados en el archivo a descomprimir y a medida que se vaya leyendo la entrada se irán añadiendo posibles combinaciones de éstos. Al descomprimir un integer comprimido se leen previamente 5 bits que indican la longitud del integer codificado. En caso de lectura del integer qué indica overflow se reinicia el diccionario.

JPEG

Compresión

Estructuras de Datos Para la compresión se ha usado una tabla Huffman con valores predefinidos.

En la primera entrega, se leía la imagen entera en la memoria y luego se comprimía. En la segunda, se lee la imagen por hileras y se comprime en el mismo momento.

Algoritmos

- *DCT (Discrete Cosine Transform)* Transforma los datos de la imagen como sumas de cosenos bidimensionales.
- *RLE (Run Length Encoding)* Codifica los datos según el número de 0 que lo preceden (Run) y los bits necesarios para representar-lo (Length). Los valores de Run y length son los que se codifican con la tabla Huffman. Gracias a la DCT, hay muchos 0 lo que permite comprimir los datos con RLE.

Descompresión

Estructuras de Datos Para descomprimir el archivo, se ha usado la misma tabla de Huffman que en la compresión pero en formato de árbol.

Al igual que al comprimir, en la primera entrega, se creava la imagen entera en la memoria mientras se descomprimía y luego se escribía. En la segunda entrega, se escribe la imagen por hileras a medida que se va descomprimiendo.

Algoritmos

- *inverse DCT (Discrete Cosine Transform)* Deshace la DCT de la compresión
- *RLE (Run Length Encoding)* Deshace el RLE de la compresión.

Folder

Compresión

Para comprimir carpetas, recorremos el árbol de ficheros en orden y guardamos para cada fichero su *path* relativo en el árbol en formato string, seguido de un marcador para señalizar el fin de la String. Después de cada marcador, siguen los datos comprimidos con los algoritmos pertinentes (con su magic byte correspondiente). La selección de algoritmo de compresión es la misa que usamos para la compresión automática de ficheros. Usamos un marcador especial para distinguir entre carpetas vacías y archivos normales.

Descompresión

La descompresión es simplemente leer los caracteres hasta llegar al marcador de fin de String y descomprimir los datos que le siguen con el algoritmo indicado por el magic byte. Cuando se ha descomprimido el fichero, buscamos la siguiente String con marcador, repitiendo el proceso anterior hasta llegar al fin del fichero.