Evaluación del efecto del orden de procesamiento de datos en la compresión de imágenes en escala de grises por métodos sin pérdida

Soler Arrufat, Sergi

INS Jaume Vicens Vives, curso 2018-2019

Grupo: 2º de Bachillerato A

Tutoría: [redacted]

Número de palabras: 3996

Agradezco a la Fundació Privada Cellex el haberme permitido realizar una estancia sobre compresión de imágenes en el Instituto de Ciencias Fotónicas durante la cual aprendí muchos conceptos que me fueron útiles para la realización de esta monografía y donde encontré inspiración para hacer la investigación práctica.

Índice de contenidos

1. Introducción

Los avances en el campo de las tecnologías de la información hacen que cada vez sea más útil disponer de datos en formato digital y esto se aplica también a las imágenes. Aunque la capacidad de almacenar datos y transmitirlos ha aumentado[1] en los últimos años, el espacio que ocupan y la cantidad que se transmite también lo ha hecho[2], por lo que continúa resultando útil comprimirlos, es decir, hacer que requieran menos espacio y banda de ancho para almacenarlos y transmitirlos.

La compresión es especialmente importante en las imágenes debido a que tienen un gran tamaño comparado con los textos: generalmente, cada pixel ocupa el equivalente a tres caracteres de texto[3] si la imagen es en color o uno si es en escala de grises. Una imagen de 5MP (5 miliones de pixeles) sin comprimir ocupará 15MB, equivalente a quince millones de caracteres de texto. Además, el problema de la compresión de datos está aún abierto: no existe un algoritmo universal que pueda comprimir todos los tipos de datos de la forma que ocupen el menor espacio posible sin descartar información ni existe un algoritmo que permita suprimir toda la información que no percibimos por las limitaciones de nuestros sentidos de modo óptimo, es decir, reduciendo al máximo el tamaño sin afectar de modo alguno a la calidad percibida.

En esta monografía se estudiarán formas de comprimir imágenes en escala de grises sin pérdida de datos. El hecho de que estén comprimidas sin perdidas permite hacer una comparación sin tener en cuenta los efectos de la percepción humana[4] y el hecho de que las imágenes sean en escala de grises hace más fácil su tratamiento puesto que la imagen tiene un solo componente, la luminosidad, en vez de los tres de las imágenes en color. Como los algoritmos que se utilizarán no son de reciente invención y por lo tanto son muy conocidos, no se estudiará su efectividad sino como afecta el orden en el que se lee y procesa la imagen al tamaño que tiene una vez comprimida. La pregunta planteada es, por lo tanto: ¿Como afecta el orden en el que se procesa una imagen al espacio que ocupa una vez comprimida?

Para responder esta pregunta, se combinará una investigación teórica sobre compresión de imágenes y algoritmos con un experimento donde se compararán distintas formas de leer imágenes y el espacio que ocupan una vez comprimidas.

1 **Hilbert, M., Lopez, P. (2011). The World's Technological Capacity to Store, Communicate, and Compute Information.**

2 **Sumits, Arielle (2015). The History and Future of Internet Traffic.**

3 **Ver el anexo V para más detalle.**

4 **Ver el subapartado "compresión" del siguiente apartado para más detalles.**

2. Símbolos y compresión

2.1. Símbolos

Para poder explicar efectivamente que es la compresión de datos es necesario definir el concepto de "símbolo". En teoría de la información, un símbolo es un elemento que forma parte de un conjunto finito[5], o lo que es lo mismo, una pieza de información que puede tener un número finito de estados. Un mensaje es una secuencia ordenada de símbolos de un conjunto. Un ejemplo podría ser una frase en español, donde cada letra corresponde a un símbolo. Por razones que se explican en el anexo V, no se incluirán las tildes, las mayúsculas o los signos de puntuación. En ese caso habrá 28 símbolos distintos: las 27 letras y el espacio. Luego, la frase "hoy llueve" sería un mensaje, puesto que es una secuencia compuesta únicamente por símbolos del conjunto definido.

Para poder tratar una frase, un texto, una imagen... como un mensaje, es necesario que cualquier pieza de información se pueda expresar como un símbolo o una secuencia de estos. Por esta razón, los espacios en blanco o cambios de parágrafo deben también ser considerados como símbolos o se deben expresar de otro modo utilizando símbolos. Esto es especialmente importante al codificar en sistema binario como se verá en el apartado de codificación Huffman.

2.2. Compresión

La compresión de datos es un conjunto de procesos por el cual se representa una pieza de información utilizando un número menor de símbolos que en su representación original. Existen dos formas de conseguir este objetivo.

La primera forma consiste en "descartar" o "ignorar" partes del mensaje que tengan poca relevancia. Esto degrada el mensaje (puesto que se está eliminando información), pero si se hace con suficiente precaución —y si se tienen en cuenta las limitaciones de la percepción humana— se puede conseguir que sea muy difícil o prácticamente imposible que una persona se percate de este hecho. Este tipo de compresión se llama *compresión con pérdidas* y es la que se utiliza como elemento principal[6], por ejemplo, en el formato de audio mp3[7] o el de imagen JPEG. Esta eliminación de datos hace que no sea adecuada para texto u otras situaciones en las que la integridad del mensaje es crítica, pero permite reducir el tamaño de los

5 Shannon, Claude E. (1948). A Mathematical Theory of Communication.

6 Se habla de "elemento principal" debido al hecho de que muchos algoritmos combinan diferentes tipos de compresión.

7 Técnicamente MPEG-1 Audio Layer III o MPEG-2 Audio Layer III.

archivos de forma drástica comparado con el otro tipo de compresión. Se utiliza en prácticamente todos los vídeos y en muchos formatos de audio e imagen, como los dos ya mencionados.

El segundo tipo es la llamada *compresión sin pérdidas* y, como su nombre indica, no causa ninguna pérdida de información. Los algoritmos que la utilizan tienen en cuenta la redundancia de los datos y la reducen, obteniendo así un mensaje más corto.

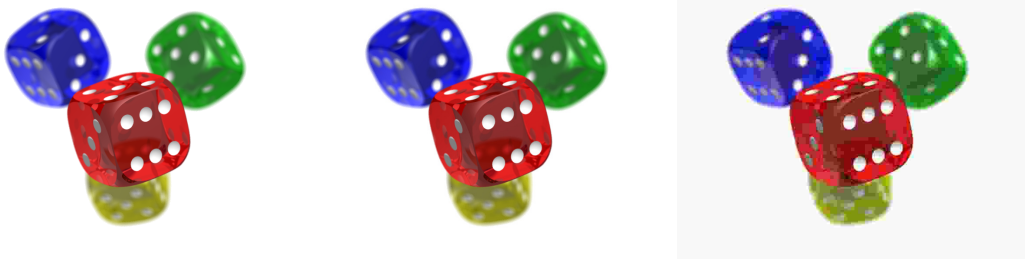


Figura 1: De izquierda a derecha: la misma imagen con compresión sin pérdidas, comprimida con pérdidas a alta calidad y comprimida con pérdidas a muy baja calidad para ilustrar el efecto de la compresión. Sus tamaños respectivos son de 212kB, 52kB y 5kB.
[CC-BY-SA 3.0 por ed_g2s]

La redundancia es una medida de la repetición de información en un mensaje. La mayoría de frases correctas y con significado en un lenguaje tienen una alta redundancia. Téngase en cuenta el siguiente fragmento de texto: "Ls montañs estban cubirtas de nie". Se trata de una frase a la cual se le han eliminado varias letras, pero aun así, es posible ver que originalmente decía "Las montañas estaban cubiertas de nieve". Esto ejemplifica como la redundancia da robustez a las transmisiones de información ya que permite recuperar el mensaje original aunque esté parcialmente dañado.

En un mensaje completamente aleatorio los datos no tienen ninguna redundancia, puesto que si se suprime o altera cualquiera de sus símbolos, no existe forma de recuperar el mensaje original. Cuanto menos "aleatorio" es el mensaje, mayor es su redundancia. La medida de la aleatoriedad se llama entropía y permite calcular el número mínimo de bits (u otra unidad de información) requeridos para transmitir un mensaje sin perder información. Técnicamente es la media ponderada (según la frecuencia con la que la fuente distribuye los símbolos) de la información que contiene cada símbolo. La "fuente" es todos los mensajes y submensajes posibles con sus correspondientes probabilidades de aparición. Si todos los símbolos del mensaje son independientes entre sí (es decir, la aparición de uno no cambia la probabilidad de aparición de los otros) la siguiente fórmula permite calcularla (en bits):

$$H(S)=\sum_{i=1}^{n}\left(p_i \log_2 \frac{1}{p_i}\right) \tag{2.1}$$

Donde p_i es la probabilidad de que el símbolo numero i aparezca y n es el total de símbolos.[8]

En la mayoría de los casos, sin embargo, los símbolos no son independientes entre sí. Continuando con ejemplos lingüísticos, es muy probable que después de una 'q' siga una 'u', y muy poco que aparezca una 'h'. La entropía real del sistema (que en este ejemplo sería el lenguaje español) siempre es igual o inferior a la que se obtiene utilizando esta fórmula. Por norma general, si se utilizan símbolos más largos y complejos —pares de letras o palabras enteras, por ejemplo— se obtiene una mejor aproximación para la entropía. Obtener una aproximación de la entropía de un sistema permite calcular el rendimiento de un algoritmo comparado con el óptimo.

3. Algoritmos de compresión sin pérdidas

Para poder entender porque algunos métodos de compresión son más eficientes que otros es necesario entender los mecanismos que utilizan. Cada algoritmo es más efectivo para un tipo específico de datos y es posible que ciertos procedimientos produzcan ficheros más grandes que los originales si se aplican a datos que no corresponden al tipo para el que son eficientes. Se han listado los métodos que se considera que tienen una complejidad que se encuentra dentro del alcance de esta monografía.

3.1. RLE (run-length encoding, codificación run-length)

El funcionamiento de RLE es muy simple: si un símbolo o secuencia de estos aparece de forma repetida y consecutiva en el mensaje, en vez de escribir la secuencia de repeticiones, se escribe la longitud de esta y el símbolo que contiene. Véase el siguiente ejemplo:

Secuencia original	(A,B,B,C,C,C,D,D,D,D,E,E,E,E,E)
Secuencia comprimida	(1,A,2,B,3,C,4,D,5,E)

Tabla 1: Ejemplo de codificación RLE

Este tipo de compresión puede implementarse de muchas formas distintas; según el tipo de datos a comprimir unas serán más eficientes que otras. Algunas asignan un 1 como número de repeticiones si no se encuentra repetición, mientras que otras no lo

[8] Shannon, Claude E. (1948). A Mathematical Theory of Communication.

hacen, por poner un ejemplo. Entre estas diferencias de implementación, una que es importante mencionar es que el numero de repeticiones puede almacenarse entre los datos (tabla 1) o como una secuencia a parte:

Secuencia original	(A,B,B,C,C,C,D,D,D,D,E,E,E,E,E)
Secuencias comprimidas	(A,B,C,D,E),(1,2,3,4,5)

Tabla 2: Ejemplo de codificación RLE con secuencias separadas para símbolos y repeticiones

Según como se haga, al aplicar otros algoritmos de compresión al resultado se podrán conseguir mejores o peores resultados.

Como se puede ver, la naturaleza de los algoritmos de este tipo hace que su uso sea eficiente en mensajes que contengan repeticiones de datos consecutivas. Algunas imágenes (especialmente las que tienen regiones contiguas de un mismo color) pueden ser ejemplos de estos tipos de datos, como también lo son ficheros de audio que contienen largos períodos de silencio total.

3.2. Codificación Huffman

Como ya se ha explicado, si un mensaje tiene una entropía baja, ciertos símbolos (o secuencias de estos) aparecen con más frecuencia que otros. Luego, para reducir el espacio que ocupa el mensaje, sería conveniente que los símbolos que aparecen con más frecuencia ocupasen menos espacio que los que aparecen de forma esporádica.

La codificación Huffman permite obtener la forma óptima de codificar cada símbolo individualmente en código binario. Es importante tener en cuenta que en código binario no hay nada para separar un conjunto de bits de otro. Si la longitud de un símbolo es siempre la misma, esto no supone ningún problema. Si, como en este caso, cada símbolo puede ocupar un numero de bits diferente, es necesario que se codifiquen como códigos prefijo. Esto quiere decir que la codificación de ningún símbolo puede tener como bits iniciales la codificación de otro símbolo más corto; de otro modo no hay forma de determinar cual de los dos símbolos representa el código binario.

A 00	A 0
B 01	B 01
C 100	C 10
D 101	D 11
E 11	E 111

Tabla 3: Ejemplo de codificación con código prefijo (izquierda) y uno que no lo es (derecha).

El código de prefijos se puede representar como un árbol binario con un nodo raíz donde bajar a la derecha es un 1 y a la izquierda un 0 (o viceversa). El algoritmo para generar el árbol de Huffman se encuentra en el anexo I.

A	00
B	01
C	100
D	101
E	11

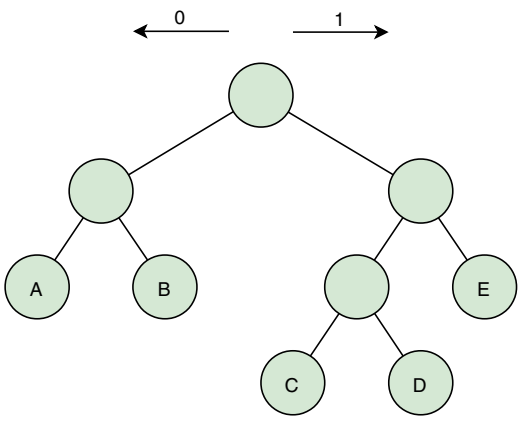


Figura 2: Código prefijo con su árbol correspondiente

Se puede generar un árbol para cada mensaje y almacenarlo junto a este o utilizar uno que se haya acordado con antelación. En el primer caso, los datos a transmitir ocuparán siempre menos espacio que los originales si no se tiene en cuenta el árbol. Solo en casos con entropía muy alta, el tamaño que ocupa el árbol será mayor que la diferencia entre los mensajes original y comprimido y resultará en un fichero de mayor tamaño. El segundo método —utilizar un árbol acordado— puede resultar en que algunos mensajes sean más largos que los originales, pero la longitud media comprimida será inferior a la sin comprimir. Este método requiere tener en cuenta la probabilidad de que aparezca un símbolo en cualquier mensaje posible para generar el árbol. Al ser imposible tener en cuenta todos los mensajes que pueden llegar a ser generados, el árbol solo se puede construir a partir de aproximaciones y puede no ser óptimo.

La codificación Huffman es una de las llamadas codificaciones de entropía, puesto que utiliza la distribución de probabilidad de los símbolos para comprimir el mensaje. Existen otras codificaciones de este tipo que resultan más eficientes que la codificación de Huffman, pero su funcionamiento es más complicado. La codificación aritmética es una de estas, pero no está tan extendida debido a que históricamente su uso estaba restringido por patentes.

3.3. Delta coding (Codificación de diferencias)

La codificación delta (o codificación de diferencias) no es de por si un algoritmo de compresión y genera un mensaje de longitud igual al original. Consiste en codificar

una secuencia de datos indicando el valor del primer símbolo y, a partir de ese momento, solo transmitiendo la diferencia entre un símbolo y el siguiente. Esta transformación del mensaje puede ser útil para ciertos tipos de mensaje si en una etapa posterior se aplica otro algoritmo de compresión como RLE o codificación Huffman, ya que puede cambiar la frecuencia de los símbolos o las repeticiones de modo que estos dos algoritmos puedan conseguir mejores resultados.

Un ejemplo de secuencia de dígitos donde esta codificación resultaría muy eficiente sería la secuencia (0,1,2,3,4,5,6,7,8,9,10). Ni RLE ni Huffman pueden comprimir esta secuencia, puesto que aparece aleatoria si se trata símbolo a símbolo. Sin embargo, si se aplica codificación delta, la secuencia que se obtiene es (0,1,1,1,1,1,1,1,1,1,1), secuencia que puede ser comprimida de forma muy eficiente por RLE y codificación Huffman.

4. Investigación práctica

Como ya se ha explicado, el objetivo de esta monografía es determinar si el orden en el que se procesan los pixeles de una imagen afecta al rendimiento de los algoritmos de compresión que se le apliquen, mediante una investigación teórica seguida de un experimento práctico. A continuación se explicará como se realizó esta segunda parte.

4.1. Planteamiento y proceso

La investigación teórica descubrió diferentes algoritmos de compresión sin pérdidas que serían los que se utilizarían en el experimento. Para realizar el experimento se requiere tener control y conocimiento sobre el método de compresión utilizado para cada caso, razón por la cual se optó por diseñar y programar un aplicativo propio que satisficiera estos requerimientos. El programa además debía permitir aplicar varios métodos de compresión de forma secuencial.

Una vez elaborado el programa de compresión, teniendo en cuenta que el experimento implicaría la realización de muchas pruebas distintas a distintos ficheros, se programó un segundo programa mucho más simple que permitiera automatizar el experimento siguiendo la metodología de este. El código de ambos programas se puede encontrar en el anexo VII.

4.2. Metodología experimental

El experimento que se realizó es una comparativa: se aplicaron diferentes métodos de compresión a diferentes imágenes y se comparó el espacio que ocupaba cada una según el método utilizado.

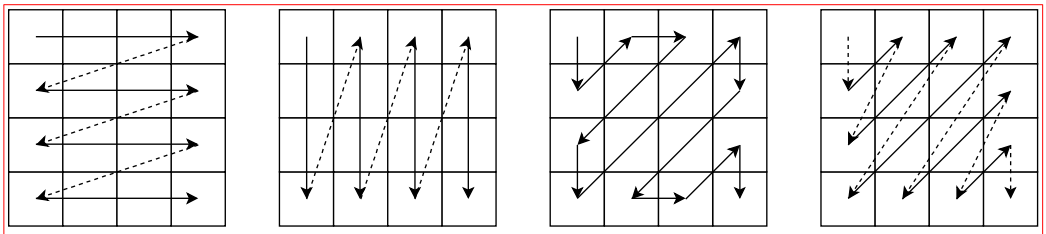


Figura 3: Los patrones utilizados. Las líneas continuas indican que los pixeles que traviesa son procesados. Las discontinuas indican que no se procesan. De izquierda a derecha, los números para referirse a cada patrón son 0, 1, 16 y 40.

Se utilizaron cuatro patrones u ordenes para procesar cada imagen. La figura 3 los muestra gráficamente. El primero consiste en leer la imagen como un texto, de izquierda a derecha y de arriba a abajo, y probablemente es el más utilizado cuando no se aplica compresión. El segundo consiste en visitar toda una columna antes de pasar a la fila siguiente. El tercero es similar al que se utiliza en el formato de imagen JPEG y el cuarto patrón consiste en recorrer la imagen en diagonal.

El programa de elaboración propia soporta activar o desactivar las codificaciones RLE y delta. Si se aplican las dos, primero se aplica delta y luego RLE. RLE utiliza dos listas separadas para los símbolos y las repeticiones. El resultado obtenido se puede (opcionalmente) codificar con Huffman, según tres modos, dos de ellos específicos a RLE. La siguiente tabla los explica con más detalle:

Modo	*Descripción*
0	No se aplica codificación Huffman.
1	Solo para RLE. Se aplica Huffman solo a la segunda lista (la del número de repeticiones).
2	Se aplica Huffman a todo el fichero con un solo árbol.
3	Solo para RLE. Se aplica Huffman a las dos listas (símbolos y número de repeticiones) con árboles diferentes para cada una.

Tabla 4: Los diferentes modos de aplicar Huffman que dispone el programa desarrollado

Se decidió que se utilizaría codificación Huffman en todos los casos, en modo 2 si no se aplica RLE y en modo 3 si sí se hace, puesto que en el peor de los casos, la codificación Huffman solo añade 264 bytes (o 528 bytes si se utiliza modo 3) de

tamaño al archivo y se espera que solo en casos muy particulares la entropía sea tan alta que su uso no permita reducir el tamaño esta cantidad de bytes. Lo mismo se aplica a la decisión de utilizar modo 3 cuando es posible en vez de 2.

El hecho de que se aplique (si se da el caso) codificación delta en primer lugar es debido a que, como ya se ha explicado, no es un algoritmo de compresión y por lo tanto debe aplicarse antes de los algoritmos que permiten reducir el tamaño. RLE es el siguiente método puesto que se espera que la lista de repeticiones tenga una entropía baja puesto que las frecuencias bajas normalmente son más frecuentes[9] que otras y por lo aplicar codificación Huffman podría reducir aún más el tamaño. Por esta razón, Huffman es el último algoritmo a aplicarse.

En total, se han utilizado 13 combinaciones de métodos y ordenes, explicadas en el anexo IV.

Las imágenes que se utilizaron se convirtieron a formato PNG y escala de grises. Luego, se comprimieron con cada uno de los trece métodos y se descomprimieron para comprobar que la compresión no las dañaba. Se registró el tamaño que ocupaba cada imagen en formato PNG, sin comprimir y con cada uno de los trece métodos.

Se seleccionaron 18 imágenes de diferentes categorías y tipos para tener una muestra de situaciones distintas. Estas, junto con una explicación más detallada sobre como se eligieron, se encuentran en el anexo II.

Idealmente, las imágenes a utilizar no tendrían que haber estado comprimidas con pérdidas en ningún momento. Esto requeriría disponer de equipo fotográfico que permita tomar imágenes en un formato sin pérdidas y todas las imágenes tendrían que ser propias. Como en el momento de realizar esta monografía no se disponía de equipo de este tipo, esta condición no se pudo dar.

5. Resultados

Los resultados en bruto que se han obtenido se pueden encontrar en el anexo III.

Se decidió que la diferencia de tamaño de una imagen utilizando los mismos algoritmos y variando solo el orden de procesamiento sería el valor que se utilizaría principalmente para responder a la cuestión inicial. Se utilizó un valor para cada combinación de algoritmos e imagen, la diferencia entre la imagen de menor tamaño y la de mayor tamaño para dicha combinación de algoritmos.[10] Para poder comparar

9 **Blelloch, Guy E. (2013). Introduction to Data Compressión.**

10 **Como resulta evidente, este proceso no se siguió en la primera combinación de algoritmos (solo Huffman) debido a su independencia del orden de procesamiento.**

los valores obtenidos con las diferentes imágenes, se pasó esta diferencia a porcentaje respecto al tamaño mínimo, de modo que un valor de 20 significaría que la peor imagen es un 20% más grande que la mejor. Estos resultados se representaron en el gráfico de la figura 4.

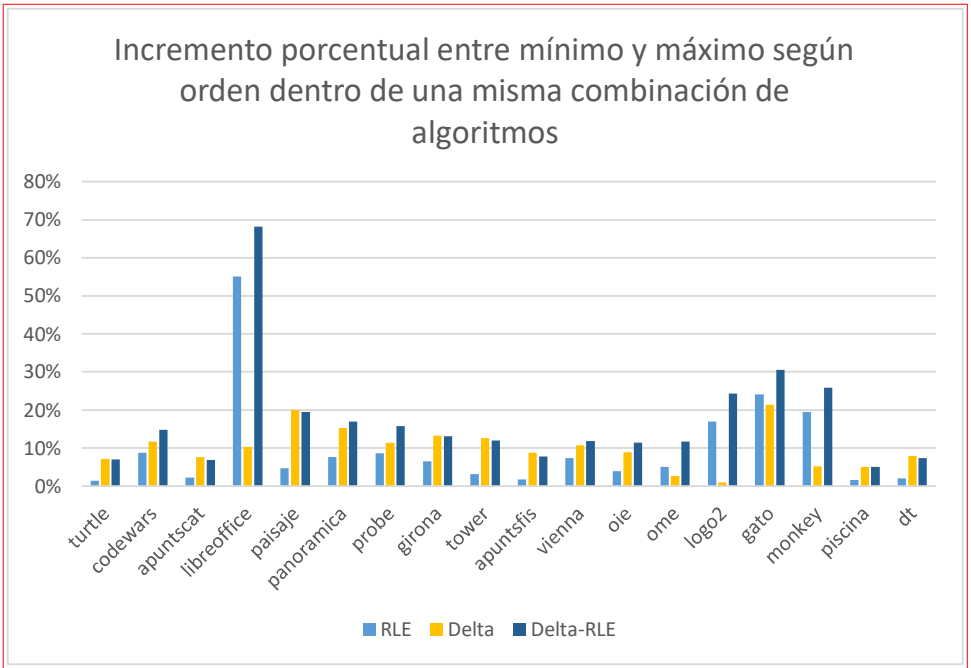


Figura 4

Este gráfico permite ver que en la mayoría de los casos, la diferencia de tamaño es inferior al 20%, aunque hay alguna excepción. El gráfico se ha hecho a partir de la diferencia máxima entre el mejor orden y el peor para cada algoritmo. Si los cuatro ordenes tuvieran un rendimiento general similar, este gráfico sería significativo. Sin embargo no hay ninguna imagen en la que el mejor resultado sea uno que utilice uno de los patrones diagonales. De hecho, estos dos patrones siempre han dado peores resultados que los otros dos y por lo tanto es conveniente dejar de considerarlos. Teniendo en cuenta este hecho, se realizó el gráfico de la figura 5, que relaciona el tamaño del archivo comprimido utilizando el patrón horizontal respecto al vertical. Los datos son el resultado de dividir el tamaño horizontal por el vertical y posteriormente restar 1 (o 100%). Así, un valor de un 20% significa que el archivo vertical es un 20% mayor que el horizontal y un -20% un 20% menor.

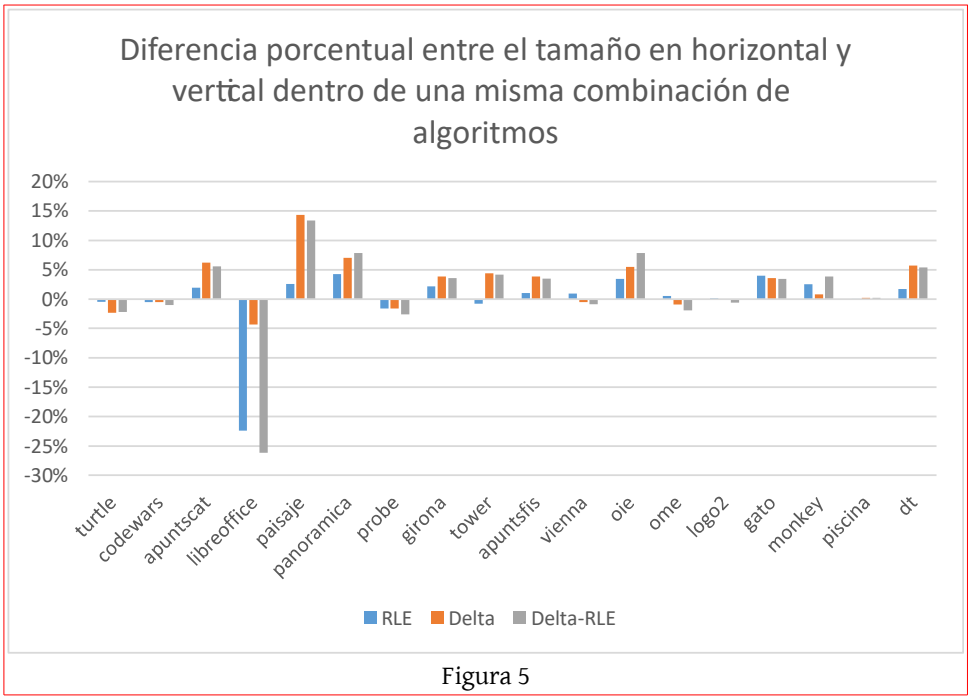


Figura 5

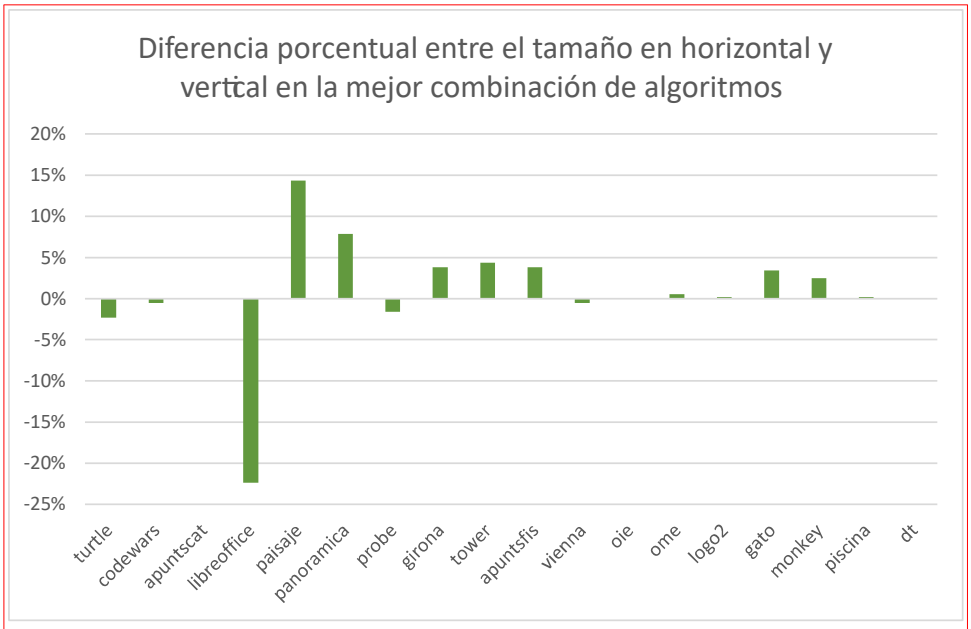


Figura 6

Finalmente, para determinar la consecuencia real de considerar ambos patrones, se realizó un último gráfico (figura 6), donde para cada imagen solo se muestra la diferencia (de la misma forma que en el anterior) para el algoritmo que da mejores resultados. En algunos casos, el mejor algoritmo es aplicar únicamente codificación Huffman, por lo que en esos casos el valor es 0, ya que esta codificación no depende de la dirección ni del orden en el que procesa los datos.

El primer hecho a destacar a partir de los gráficos, aunque no el más evidente, es que aparentemente la codificación delta es más sensible al cambio de dirección de lectura que la codificación run-length. Observando los datos utilizados para generar los gráficos 2 y 3, de las 18 imágenes, 5 tenían más variación de tamaño al comprimir con RLE y 13 con delta, cosa que respalda esta teoría.

El segundo hecho a destacar es que al pasar de considerar los patrones diagonales a no hacerlo, la diferencia de tamaño se ha reducido considerablemente. Esto indica que estos patrones no solo son peores en todos los casos, sino que en algunas imágenes existe un gran margen entre la eficiencia de estos y la de los otros dos.

En tercer lugar, se puede ver que en el caso de la imagen "libreoffice" la diferencia entre patrones es muy drástica. Observando los datos en bruto (anexo III) se puede ver que la diferencia entre el tamaño de la imagen comprimida con RLE y con codificación delta pero sin RLE es la más grande, suponiendo el uso de solo delta un incremento del 200% aún considerando los casos con menor diferencia —en otros el incremento llega a más del 300%—. Probablemente el bajo numero de colores de la imagen, posible debido a que procede de un formato comprimido sin pérdidas[11] permite que RLE sea tan eficiente. La diferencia entre direcciones de lectura puede ser debida a la direccionalidad del texto: aunque no se encuentra dentro del alcance de esta monografía investigar tal hecho, se teoriza que el alfabeto latín contiene más rayas verticales que horizontales, haciendo que RLE sea más eficiente al leer de forma vertical y lo sea poco al leer en horizontal.

Por último, se ha calculado la media y mediana de la variación porcentual de tamaño en valor absoluto. Son las siguientes:

[11] **Ver anexo II.**

Todos los patrones		
RLE	**Delta**	**Delta-RLE**
Mediana: 5,75%	Mediana: 9,57%	Mediana: 12,53%
Media: 10,02%	Media: 10,04%	Media: 17,21%

Tabla 5: Estadísticos del primer gráfico

Horizontal-vertical		
RLE	**Delta**	**Delta-RLE**
Mediana: 1,64%	Mediana: 3,69%	Mediana: 3,52%
Media: 2,84%	Media: 2,50%	Media: 5,22%

Tabla 6: Estadísticos de los datos del segundo gráfico en valor absoluto

Mejor algoritmo
Mediana: 1,97%
Media: 3,80%

Tabla 7: Estadísticos para el mejor algoritmo

6. Conclusiones

A partir de los datos obtenidos y respondiendo a la pregunta de investigación, se puede concluir que el orden en el que se lee una imagen afecta de forma importante el espacio que ocupa una vez comprimida. Este hecho ha sido demostrado por la ineficiencia del uso de los patrones diagonales frente a los patrones horizontal y vertical. También se ha llegado al resultado leer la imagen de izquierda a derecha y de arriba abajo (patrón 0) es el orden más eficiente en la mayoría de las circunstancias y en los pocos casos que no es así, la diferencia es pequeña. A partir de estos dos hechos, se puede concluir que en la mayoría de los casos no es importante determinar que orden es el mejor para una imagen en concreto puesto que utilizar el patrón número 0 acostumbra a dar buenos resultados, pero el uso de un patrón inadecuado para la imagen puede implicar un incremento de espacio considerable.

Se ha determinado también que existen excepciones a esta regla —se identificó la imagen “libreoffice”, que no cumple esta tendencia— y se ha teorizado que el alfabeto latino puede tener cierta direccionalidad que puede favorecer la compresión si se lee la imagen de forma vertical. Las imágenes de textos escaneados no han mostrado este fenómeno, por lo que se supone que bien el proceso de escaneado o la densidad del texto influyen de forma notable en el proceso de

compresión. A fin de determinar la razón de este valor anómalo, sería útil repetir esta investigación utilizando imágenes seleccionadas y creadas específicamente para probar o desmentir esta teoría.

La longitud de este texto no permite estudiar a fondo las diferencias según tipo de imagen, hecho que permitiría obtener conclusiones más específicas sobre que factores de la imagen afectan a la diferencia de tamaño al cambiar el orden de lectura. Tampoco ha sido posible realizar un tratamiento estadístico avanzado de los resultados por esta misma razón. Además de corregir estos dos defectos, los siguientes pasos para mejorar esta investigación serían aumentar el número de imágenes utilizadas para poder hacer tal clasificación y obtener datos más significativos, aumentar el número de patrones y hacer una investigación a fondo sobre diferentes formas de procesar los datos y, finalmente, añadir otros algoritmos de compresión como LZ77, que no pudo ser incluido por su complejidad y la falta de espacio.

En conclusión, ha sido posible responder a la pregunta inicial a partir de los datos obtenidos en el apartado práctico. Aunque se encontraron algunas dificultades —el algoritmo de Huffman resultó ser más difícil de implementar de lo que se creía— fue posible terminar el trabajo como se había planeado y de forma que las únicas modificaciones que se harían serían las mejoras descritas en el párrafo anterior.

7. Bibliografía

BLELLOCH, GUY E. (2013). *Introduction to Data Compressión*, **http://www.cs.cmu.edu/afs/cs/project/pscico-guyb/realworld/www/compression.pdf** [consulta de 23 de junio de 2018]

FELDSPAR, ANTAEUS (2002). *An Explanation of the 'Deflate' Algorithm*, **https://zlib.net/feldspar.html** [consulta de 18 de agosto de 2018]

HILBERT, M., LOPEZ, P. (2011). *The World's Technological Capacity to Store, Communicate, and Compute Information,* doi:10.1126/science.1200970

HUFFMAN, DAVID A. (1952). *A Method for the Construction of Minimum-Redundancy Codes*, doi:10.1109/JRPROC.1952.273898

SALOMON, DAVID (2007). *Data Compression: The Complete Reference*. 4ª ed., Londres: Springer.

SHANNON, CLAUDE E. (1948). *A Mathematical Theory of Communication*, doi:10.1002/j.1538-7305.1948.tb01338.x

SMITH, STEVEN W. (1998). *The Scientist and Engineer's Guide to Digital Signal Processing,* **http://www.dspguide.com** [consulta de 20 de agosto de 2018]

SUMITS, ARIELLE (2015). *The History and Future of Internet Traffic,* **https://blogs.cisco.com/sp/the-history-and-future-of-internet-traffic** [consulta de 7 de octubre de 2018]

WAHBA, W. Z., MAGHARI, A. Y. A. (2016) *Lossless Image Compression Techniques Comparative,* Study International Research Journal of Engineering and Technology (IRJET), volumen 3.

Anexo I: Algoritmo de Huffman y árbol canónico de Huffman

El algoritmo de Huffman genera el árbol de la siguiente forma:

1. Primero, se cuantifica la frecuencia de aparición de cada símbolo y se añaden a una 'cola'.
2. Mientras haya más de un elemento (nodo) en la cola:
    1. Los dos elementos con menor frecuencia pasan a ser hijos de un nuevo nodo que tiene como frecuencia la suma de las dos frecuencias.
    2. Se suprimen los dos elementos de la cola y se añade el nuevo nodo a esta.
3. El nodo resultante es la "raíz" del árbol.

Como la longitud del código correspondiente a un símbolo es la parte importante del algoritmo, es posible "cambiar nodos de sitio" si no se cambia su longitud. Es posible reordenar el árbol de modo que el resultado sea un árbol "canónico". Este árbol cumple la propiedad de dados cualquier par símbolos y sendas codificaciones, de longitudes n y k, donde $n>k$, el numero binario formado por los k primeros bits de la codificación de longitud n será siempre mayor que el número formado por el código más corto. Esto permite codificar el árbol con el número de códigos de longitud 1,2,3..., n seguido de los códigos.

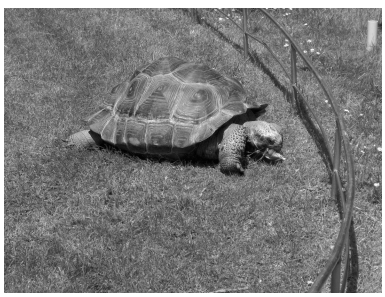
Anexo II: Imágenes utilizadas en el experimento

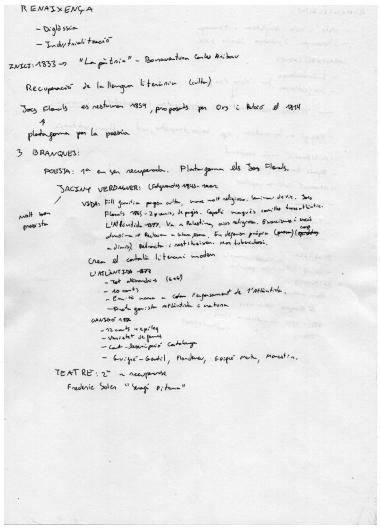
Se decidió que las imágenes utilizadas en el experimento deberían ser de tipos varios a fin de hacer que los resultados fueran significativos. Por esta razón, se decidió considerar tres categorías: fotografía, texto y artificial. Estas tres categorías se corresponden con tres formas de crear imágenes: por medio de capturar escenas u objetos reales, por escaneo de documentos o por diseño digital. La primera categoría comprende las imágenes resultado de realizar una fotografía – paisajes, personas, objetos, etc. No se incluyen las fotografías de textos o cualquier otro material impreso, puesto que estas caen dentro de la segunda. Esta categoría, texto, incluye las imágenes resultado de escanear un documento. Finalmente, la última categoría incluye las imágenes de creación digital. Dentro de cada categoría se han buscado imágenes sustancialmente diferentes entre sí a fin de ampliar los escenarios considerados.

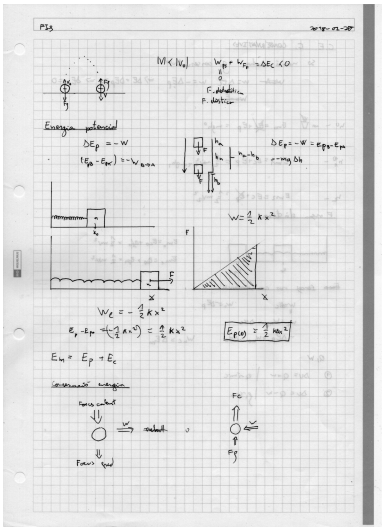
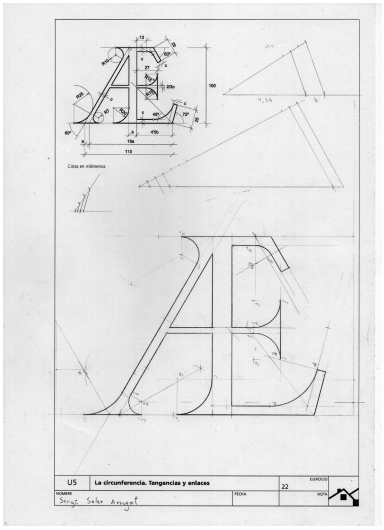
El formato de las imágenes se encuentra en la tabla. Si la imagen procede de otra en otro formato, el formato de la imagen original se encuentra entre paréntesis y el de la imagen utilizada fuera. El formato PNG aplica compresión sin pérdidas. JPEG, como ya se ha mencionado, aplica compresión con pérdidas. SVG es un formato de gráficos vectoriales, es decir, no almacena una lista de valores para los colores (o intensidad luminosa) de los pixeles, sino que almacena "las instrucciones" para crear el gráfico: posición de elementos como rectas, curvas; tamaño, etc. Se considera que son imágenes sin pérdida puesto que representan exactamente el gráfico original.

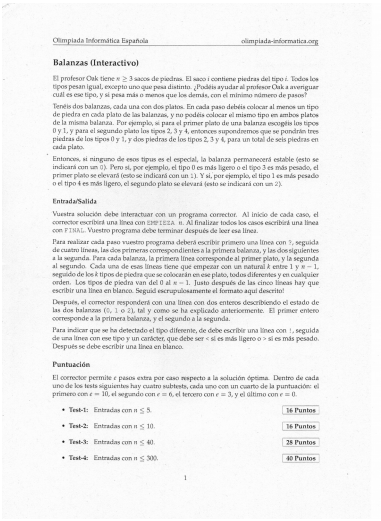
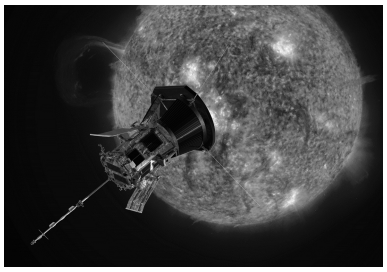
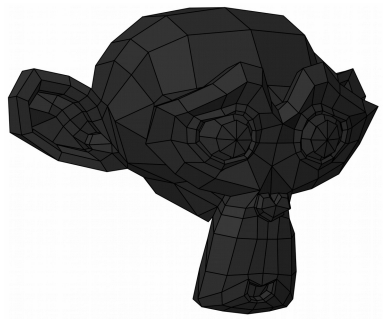
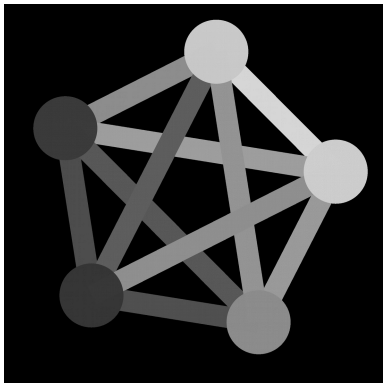
El nombre que se ha asignado a las imágenes no corresponde con su título, si lo tuvieran, ni es necesariamente una palabra en español.

Formato original	*Nombre*	*Imagen*	*Tipo*	*Autoría*
PNG	vienna		Fotografía – estructura	Robert F. Tobler (CC-BY-SA 4.0)

PNG	tower		Fotografía – estructura	Robert F. Tobler (CC-BY-SA 4.0)
JPEG	girona		Fotografía – estructura	Propia
JPEG	turtle		Fotografía – natura	Propia

JPEG	paisaje		Fotografía – paisaje	Propia
JPEG	panoramica		Fotografía – panorámica	Propia
JPEG	piscina		Fotografía – otros (material de piscina)	Propia
JPEG	codewars	REMOVED	Fotografía – gente	Desconocido, imagen pública
PNG	apuntscat		Texto – apuntes manuscritos	Propia

PNG	apuntsfis		Texto – apuntes manuscritos	Propia
PNG	dt		Texto – dibujo técnico	Propia
PNG	ome		Texto – texto con iconos poco denso	Olimpíada Matemática Española

PNG	oie		Texto – texto denso en papel reciclado	Cesc Folch Aldehuelo / Olimpíada Informática Española
PNG (SVG)	libreoffice		Artificial – logotipo	Christoph Noack (CC-BY-SA 3.0)
JPEG	probe		Artificial – imagen 3D	NASA/Johns Hopkins APL/Steve Gribben
PNG (SVG)	monkey		Artificial – imagen 3D	Inductiveload
PNG (SVG)	logo2		Artificial – logotipo	Eukombos

PNG (JPEG)	gato		Artificial – imagen pixelada parcialmente	Michael Gäbler/ AzaToth (CC-BY 3.0)

Anexo III: Resultados

Los resultados del experimento (los tamaños de cada fichero). Todos los valores son medidos en bytes. Los valores de cada fila se han coloreado con un degradado de verde (fichero más pequeño) a rojo. El nombre de las columnas indica el modo de la siguiente forma: D indica codificación delta, R indica codificación run-length, H*n* indica que se ha utilizado el modo *n* de Huffman y el número final indica el modo de procesar la imagen.

Imagen	PNG	pgm	H2_0	RH3_0	RH3_1	RH3_16	RH3_40	DH2_0	DH2_1	DH2_16	DH2_40	DRH3_0	DRH3_1	DRH3_16	DRH3_40
turtle	3697579	5059601	4573955	4993235	4972447	5039156	5038976	3787526	**3701420**	3963062	3962011	4310777	4220110	4513128	4511971
codewars	294472	699408	680069	596385	593374	645746	645321	351485	**349611**	390327	388750	373656	369877	424444	422686
apuntscat	5514146	8681489	**4957663**	5529601	5640714	5652903	5651385	5401221	5759342	5814491	5811631	6323969	6693414	6755448	6751026
libreoffice	33511	942016	170214	36242	**29619**	45865	45915	145665	139660	153498	153994	48246	38229	63772	64313
paisaje	2656515	5059601	4604086	4572941	4694553	4786976	4786404	**2653447**	3097623	3183347	3178929	3056725	3527047	3651323	3646254
panoramica	1435215	2975248	2862240	2457556	2566116	2645125	2643311	**1602982**	1724466	1847141	1842899	1722262	1869062	2013322	2008452
probe	6709275	29254517	24234659	13423283	13211420	14352628	14337828	8558504	8422920	9377953	9368573	7818835	**7619446**	8817720	8798180
girona	4247987	9980945	9738274	8606909	8800331	9164476	9162802	**4367237**	4540787	4943897	4938458	4959687	5143759	5612599	5603954
tower	3481694	7372817	6826683	5755924	5711747	5894546	5892840	**4118713**	4307243	4637212	4633804	4589013	4786685	5135024	5130612
apuntsfis	5275769	8721681	5454325	6026647	6091087	6134856	6134527	**5417756**	5632210	5894754	5892939	6356581	6584832	6853200	6851678
vienna	13755980	35389457	33564944	27055368	27317830	29039955	29035833	15237833	**15154891**	16776939	16766647	17321743	17173477	19197362	19184789
oie	4749770	8759361	**4178277**	4478718	4638550	4657176	4654920	4987969	5276287	5430825	5430526	5420270	5880287	6040260	6037370
ome	459365	8759361	1434060	**456677**	459151	479614	479552	1471988	1458006	1497453	1497478	502487	492934	550904	550863
logo2	61477	4000017	1205500	**43629**	43702	51024	51017	522106	521852	526850	527123	51649	51334	63530	63811
gato	87788	311415	274899	153433	159854	190349	189375	131228	136077	159370	158337	**123517**	127918	161225	159522
monkey	204061	3334017	1457610	**171262**	175627	204563	204681	537688	541814	565972	565994	210201	218492	264491	264543
piscina	2715525	5059601	3310740	3604431	3606625	3663954	3663713	**2758479**	2765368	2896770	2895884	3233898	3240885	3398440	3397544
dt	5441040	8721681	**5174776**	5773577	5872357	5887983	5887760	5436589	5766110	5866909	5866064	6365985	6729398	6834346	6833413

Anexo IV: Combinaciones de métodos de compresión utilizadas

Hay dos factores a considerar para comprimir una imagen en el experimento: el orden y los algoritmos a utilizar. Como siempre se utiliza Huffman (se ha justificado el porque en el cuerpo del trabajo) y el modo en que se utiliza viene determinado por si se ha utilizado o no RLE, no es necesario tenerlo en cuenta en el cómputo de posibilidades. Luego, es posible utilizar o no RLE y codificación delta. Esto genera cuatro posibilidades: no utilizar ninguno, utilizar solo RLE, utilizar solo delta o utilizar ambos. Si para cada patrón de lectura se aplicasen los cuatro métodos, el resultado sería que habría 16 formas diferentes de comprimir cada imagen. Sin embargo, cuando no se aplica ni RLE ni delta, solo se aplica Huffman, método que es independiente del orden de los datos de entrada. Por esta razón, si no se aplica RLE ni delta, solo se utiliza un patrón. En total, pues hay 13 formas de comprimir cada imagen: todas las combinaciones menos tres en el caso de solo Huffman. La tabla de resultados los contiene todos así como su designación en código.

Anexo V: Codificación de texto

Existen muchos sistemas de codificación de texto diferentes, pero para esta monografía solo se considerará un sistema: UTF-8. Hay tres razones para esta decisión: en primer lugar, el sistema UTF-8 es el sistema de codificación de texto más utilizado, estando un 92,3% de las páginas web codificadas con este; en segundo lugar, es capaz de codificar cualquier carácter Unicode[12], razón por la cual puede ser utilizado en casi cualquier situación comunicativa; en tercer lugar, es compatible con el antiguo sistema de codificación ASCII, siendo este un subconjunto de UTF-8.

UTF-8 es un sistema de codificación de texto de longitud variable, es decir, no todos los caracteres tienen la misma longitud una vez codificados. Los caracteres pertenecientes a ASCII ocupan un solo byte, mientras que los emoticonos, por ejemplo, ocupan cuatro.

Para el cálculo de la introducción, se ha considerado que todos los caracteres de texto eran de un byte de longitud, es decir, se encontraban del subconjunto de ASCII. Esto sería cierto para la mayoría de textos en inglés. En español, los caracteres con tilde o la eñe no forman parte de este conjunto. Sin embargo, el resto de letras del alfabeto si que pertenecen a él, por lo que se considera que la aproximación de un byte por palabra es suficientemente precisa, sobretodo teniendo en cuenta que se utilizó para hacer una comparación.

Para los ejemplos utilizados en esta monografía, hay un aspecto que es importante mencionar sobre Unicode (y por lo tanto también para UTF-8, aunque la codificación no es necesaria para entender los ejemplos): hay elementos de texto que se pueden codificar de más de una forma: las letras con acento se pueden codificarse como un carácter de acento seguido del carácter al que se aplican o como el carácter con el acento incorporado. Seria posible hacer que las mayúsculas se codificaran como un carácter modificador (que haría que el siguiente carácter fuera una mayúscula) y la letra a la que modifican.

Es por esta razón que en los ejemplos lingüísticos no se utilizan mayúsculas ni caracteres acentuados ni signos de puntuación, puesto que sería posible confundir el concepto de símbolo con el de codificación. Aunque las codificaciones sirven para referirse a símbolos, un símbolo puede ser codificado de diferentes formas. Estas diferentes formas de codificar podrían causar confusión en los ejemplos: un carácter acentuado podría parecer un símbolo para un lector o dos para otros, dependiendo en que términos pensaran. Como la longitud del trabajo no permite hacer el tipo de

[12] **Ver definición en el anexo VI para más información.**

explicación necesario evitar este problema, se ha decidido utilizar un subconjunto de caracteres que no causen confusión en los lectores y que los reconozcan como un símbolo cada uno.

La omisión de los signos de puntuación se ha hecho para evitar tener que delimitar el concepto de signo de puntuación y/o tener que definir un número concreto de símbolos. Los ejemplos no los requieren y se considera que una discusión sobre los signos de puntuación del español no aportaría ningún contenido al trabajo (no es relevante) mientras que la enumeración de estos sin explicación alguna podría resultar confusa.

Anexo VI: Definiciones

Bit: un dígito en código binario. Desde el punto de vista de la teoría de la información, un símbolo que solo puede tomar valores "0" y "1".

Byte: unidad de información correspondiente a ocho bits, que puede ser la representación en código binario de un número. Desde el punto de vista de la teoría de la información, un símbolo que puede tomar 256 (2^8) valores distintos.

Pixel: unidad mínima de información en una imagen digital. Representa un color o una intensidad luminosa y normalmente tiene un tamaño de 3 bytes en el primer caso o 1 byte en el segundo.

Unicode: estándar de codificación de caracteres que establece un conjunto de símbolos para el almacenamiento y transmisión de texto, pero no la forma de codificar los símbolos. Incluye la practica totalidad de los alfabetos utilizados hoy en día, además de símbolos matemáticos, musicales, fonéticos, emoticonos, etc.

Anexo VII: Código desarrollado

Una versión digital del código se puede encontrar en el siguiente enlace:

https://gitlab.com/fraret/frimco

En la versión pdf de la monografía, se encuentra incrustado aquí

El código en C++ se ha compilado con g++ 8.2.0 en un sistema linux. El código python se ha utilizado con la version 3.7.

Programa principal

binaryfileif.cpp

```
#include "binaryfileif.h"


BinaryFileIf::BinaryFileIf(const std::string& filename){
    instream.open(filename, std::ios_base::binary);
    next_bit=0;
    next_bit_remaining=0;
}

bool BinaryFileIf::GetBit(){
    if(next_bit_remaining==0){
        next_bit=instream.get();
        next_bit_remaining=8;
    }
    bool result= (next_bit & 0x80)>>7;
    next_bit=next_bit<<1;
    --next_bit_remaining;
    return result;
}

uint8_t BinaryFileIf::get(){
    next_bit_remaining=0;
    next_bit=0;
    if(instream.peek() != EOF) return instream.get();
    else throw("EOF Error");
}


void BinaryFileIf::close(){
    instream.close();
}

index BinaryFileIf::GetTextIndex(){
    next_bit_remaining=0;
    next_bit=0;
    index i;
    instream>>i;
    instream.get();
```

```
    return i;
}
```

binaryfileif.h

```
#ifndef BINARYFILEIF_H
#define BINARYFILEIF_H


#include <fstream>
#include <string>

typedef int index;

class BinaryFileIf{
    private:
        uint8_t next_bit=0;
        uint8_t next_bit_remaining=0;
        std::ifstream instream;

    public:
        bool GetBit();
        uint8_t get();
        BinaryFileIf(const std::string& filename);
        void close();
        index GetTextIndex();
};

#endif // BINARYFILEIF_H
```

binaryfileof.cpp

```
#include "binaryfileof.h"

void BinaryFileOf::PutBit(bool bit){
    next_bit=next_bit<<1;
    next_bit+=bit;
    --next_bit_remaining;
    if(not next_bit_remaining){
        outstream.put(next_bit);
        next_bit=0;
        next_bit_remaining=8;
    }

}

BinaryFileOf::BinaryFileOf(const std::string& filename){
    outstream.open(filename, std::ios_base::binary);
    next_bit=0;
    next_bit_remaining=8;
}
```

```
void BinaryFileOf::put(uint8_t char_f){
    if(next_bit_remaining==8){
        outstream.put(char_f);
    }else{
        next_bit=next_bit<<next_bit_remaining;
        outstream.put(next_bit);
        next_bit=0;
        next_bit_remaining=8;
        outstream.put(char_f);
    }

}

void BinaryFileOf::close(){
    if(next_bit_remaining==8){
        outstream.close();
    }else{
        next_bit=next_bit<<next_bit_remaining;
        outstream.put(next_bit);
        next_bit=0;
        next_bit_remaining=8;
        outstream.close();
    }
}
```

binaryfileof.h

```
#ifndef BINARYFILEOF_H
#define BINARYFILEOF_H



#include <fstream>
#include <string>

class BinaryFileOf{
    private:
        uint8_t next_bit=0;
        uint8_t next_bit_remaining=8;
        std::ofstream outstream;

    public:
        void PutBit(bool bit);
        void put(uint8_t char_f);
        BinaryFileOf(const std::string& filename);
        void close();

};

#endif // BINARYFILEOF_H
```

huff.cpp

```
#include <queue>
#include <map>
#include <fstream>
#include <iostream>
#include <algorithm>

#include "binaryfileof.h"
#include "binaryfileif.h"
#include "huff.h"


struct node{
    node *left, *right;
    huff_data value;
    index freq,age;
    bool leaf;
    node(huff_data data,index freq_f, index age_f){
        left=NULL;
        right=NULL;
        age=age_f;
        value=data;
        freq=freq_f;
        leaf=true;
    }
    node(node *left_f, node *right_f,index age_f){
        left=left_f;
        right=right_f;
        age=age_f;
        freq=left->freq+right->freq;
        value=huff_null;
        leaf=false;
    }
    node(){
        left=NULL;
        right=NULL;
        value=huff_null;
        leaf=false;
    }
    node(huff_data data){
        left=NULL;
        right=NULL;
        value=data;
        leaf=true;
    }
};


struct more_frequent{
```

```
    bool operator()(node* a, node* b){
        if((a->freq)>(b->freq)) return true;
        else if((a->freq)==(b->freq)){
            if ((a->age)>(b->age)) return true;
            else return false;
        }else return false;
    }
};


bool CodedFile::isRLE(){
    return mode & 0x02;
}
bool CodedFile::isDelta(){
    return mode & 0x01;
}


std::vector<std::pair<huff_data,index>>
DataToDataAndFreqs(const std::vector<huff_data> & data){
    std::vector<index> freqs(256*sizeof(huff_data),0);
    for(const huff_data & value:data){
        ++freqs[value];
    }
    std::vector<std::pair<huff_data,index>> answer;
    for(index i=0;i<freqs.size();++i){
        if(freqs[i]!=0){
            answer.push_back({i,freqs[i]});
        }
    }
    return answer;
}


node* CreateHuffTree(const
std::vector<std::pair<huff_data,index>> & data_and_freqs){

std::priority_queue<node*,std::vector<node*>,more_frequent
> p_queue;
    for(auto & p: data_and_freqs){
        p_queue.push(new node(p.first,p.second,0));
    }
    index i=1;
    while(p_queue.size()>1){
        node* first=p_queue.top();
        p_queue.pop();
        node* second=p_queue.top();

        node* temp;
        if(first->age>second->age){
            temp=second;
            second=first;
```

```
            first=temp;
        }
        p_queue.pop();
        p_queue.push(new node(first,second,i));
        ++i;
    }

    return p_queue.top();
}

node* CreateHuffTree(const std::vector<huff_data> & vector){
    return CreateHuffTree(DataToDataAndFreqs(vector));
}

bool canon(std::pair<huff_data,huff_data> & a,
std::pair<huff_data,huff_data> & b){
        if(a.second<b.second) return true;
        return false;
}

void increment(std::vector<bool> & vec){
    index last=vec.size()-1;
    while(vec[last]){
        vec[last]=false;
        --last;
    }
    vec[last]=true;

}

void HuffTreeToMap(node* tree, huff_data
current_code_length,
std::vector<std::pair<huff_data,huff_data>> & pairs){
    if(not (tree->leaf)){
        ++current_code_length;
        HuffTreeToMap(tree-
>left,current_code_length,pairs);
        HuffTreeToMap(tree-
>right,current_code_length,pairs);
        --current_code_length;
    }else{
        pairs.push_back({tree-
>value,current_code_length});
    }

}

std::pair<std::map<huff_data,std::vector<bool>>,std::vector
<std::pair<huff_data,huff_data>>> HuffTreeToMap(node* tree){

    huff_data temp=0;
```

```
    std::vector<std::pair<huff_data,huff_data>> pairs;
    HuffTreeToMap(tree,temp,pairs);

    std::map<huff_data,std::vector<bool>> answer;
    std::vector<bool> nums;


    std::sort(pairs.begin(),pairs.end(),canon);



    for(auto & p:pairs){
        if(nums.size()) increment(nums);
        while(p.second>nums.size()) nums.push_back(0);
        answer[p.first]=nums;


    }
    return {answer,pairs};
}



std::vector<huff_data> MapToVector(const
std::map<huff_data,std::vector<bool>> & map_f, const
std::vector<std::pair<huff_data,huff_data>> & pairs){
    /*
     *
     *
     */

    if(pairs.size()==1) throw ("Unary Huffman tree not
supported yet");

    std::vector<huff_data> freq;
    huff_data max_seen_length=0;
    for(const std::pair<const huff_data,const huff_data> &
p:pairs){
        const huff_data & length=p.second;
        while(max_seen_length<length){
            freq.push_back(0);
            ++max_seen_length;
        }
        ++freq[length-1];
    }


    for(auto & p:pairs){
        freq.push_back(p.first);

    }
```

```
    return freq;

}

std::vector<huff_data>
MapToVector(std::pair<std::map<huff_data,std::vector<bool>>
,std::vector<std::pair<huff_data,huff_data>>> & thing){
    return MapToVector(thing.first,thing.second);

}

huff_data ReadHuffNum(BinaryFileIf & infile){
    static_assert(sizeof(huff_data)==1, "Multi-byte huffman
not yet implemented");
    return infile.get();
}

void WriteHuffNum(huff_data num, BinaryFileOf & outfile){
    static_assert(sizeof(huff_data)==1, "Multi-byte huffman
not yet implemented");
    outfile.put(num);
    return;
}

void FillLayer(huff_data & nums, huff_data layer, node*
tree, BinaryFileIf & infile){
    if(tree->leaf) return;
    if(layer==0){
        if(nums>=2){
            tree->left=new node(ReadHuffNum(infile));
            tree->right=new node(ReadHuffNum(infile));
            nums-=2;
        }else if (nums==1){
            tree->left=new node(ReadHuffNum(infile));
            tree->right=new node();
            --nums;
        }else{
            tree->left=new node();
            tree->right=new node();
        }

    }else{
        FillLayer(nums,layer-1,tree->left,infile);
        FillLayer(nums,layer-1,tree->right,infile);
    }


}

node* TreeFromFile(BinaryFileIf & infile){
    std::vector<huff_data> lengths;
```

```
    huff_data remaining_codes=2;
    huff_data data_num=0;
    while(remaining_codes){
        huff_data next_num=ReadHuffNum(infile);
        lengths.push_back(next_num);
        remaining_codes-=next_num;
        data_num+=next_num;
        remaining_codes*=2;

    }


    node* tree = new node();
    huff_data layer=0;
    for(auto next_num:lengths){
        FillLayer(next_num,layer,tree,infile);
        ++layer;
    }


    return tree;
}



void VectorToFile(const std::vector<huff_data> &
vector_f,BinaryFileOf & outfile){
    for(const auto & num:vector_f){
        WriteHuffNum(num,outfile);
    }
}

void DataToFile(const std::vector<huff_data> & data,
std::map<huff_data,std::vector<bool>> & huff_map,
BinaryFileOf & outfile){
    for(const huff_data & value:data){
        const std::vector<bool> & code=huff_map[value];
        for(const bool & bit:code){
            outfile.PutBit(bit);
        }
    }

}

std::vector<huff_data> FileToData(BinaryFileIf & infile,
node* tree,index num){
    std::vector<huff_data> answer(num);
    node* current=tree;
    index i=0;
    while(i<num){
        if(current->leaf){
            answer[i]=current->value;
```

```
            current=tree;
            ++i;
        }else{
            bool bit=infile.GetBit();
            if(bit){
                current=current->right;
            }else{
                current=current->left;
            }
        }

    }
    return answer;
}


index ReadIndex(BinaryFileIf & infile){
    static_assert(sizeof(index)==4,"Maximum symbols is
fixed to uint32 at the moment");
    index datalength=0;
    datalength|= infile.get()<<24;
    datalength|= infile.get()<<16;
    datalength|= infile.get()<<8;
    datalength|= infile.get();
    return datalength;
}

void WriteIndex(index num, BinaryFileOf & outfile){
    static_assert(sizeof(index)==4,"Maximum symbols is
fixed to uint32 at the moment");
    outfile.put(num>>24);
    outfile.put(num>>16);
    outfile.put(num>>8);
    outfile.put(num);

}


void DestroyTree(node* tree){
    if(tree->leaf){
        delete tree;
    }else{
        DestroyTree(tree->left);
        DestroyTree(tree->right);
        delete tree;
    }
}

CodedFile ReadData(std::string & filename){
    CodedFile answer;
    BinaryFileIf infile(filename);
```

```
    if(infile.get()!='E' and infile.get()!='5'){
        std::cerr<<"File has not the magic
number"<<std::endl;
    }else infile.get();
    infile.get();
    answer.mode=infile.get();
    answer.order=infile.get();
    infile.get(); //0A
    answer.x_size=ReadIndex(infile);
    answer.y_size=ReadIndex(infile);

    node* tree=NULL;
    if((answer.mode & 0x80)){//Uses a single Huffman table
or two
        if(infile.get()!=0xFE) throw ("No huffman table");
        tree =TreeFromFile(infile);
    }
    uint8_t temp=infile.get();
    if(temp!=0xFF){
        std::cout<<(int)temp<<std::endl;

        throw ("No data");
    }

    int datalength=ReadIndex(infile);

    if((answer.mode & 0x80)){//Uses a single Huffman table
or two
        answer.data=FileToData(infile,tree,datalength);
    }else{
        answer.data.resize(datalength);
        for(index i=0;i<datalength;++i){
            answer.data[i]=infile.get();
        }
    }
    if(answer.isRLE()){
        if(answer.mode & 0x40){
            if(tree!=NULL) DestroyTree(tree);
            if(infile.get()!=0xFE) throw ("No second huffman
table");
            tree =TreeFromFile(infile);
        }

        if(infile.get()!=0xFD){
            throw ("No runs");
        }
        int runlength=ReadIndex(infile);


        if((answer.mode & 0x80 or answer.mode &
0x40)){//Uses huffman
            answer.coefs=FileToData(infile,tree,runlength);
```

```
            DestroyTree(tree);
        }else{
            answer.coefs.resize(runlength);
            for(index i=0;i<runlength;++i){
                answer.coefs[i]=infile.get();
            }
        }
    }
    infile.close();
    return answer;


}

void WriteData(CodedFile encoded_file, std::string &
filename){
    BinaryFileOf outfile(filename);
    outfile.put('E');outfile.put('5');outfile.put('\n');
    outfile.put(encoded_file.mode);
    outfile.put(encoded_file.order); outfile.put('\n');
    WriteIndex(encoded_file.x_size,outfile);
    WriteIndex(encoded_file.y_size,outfile);

    node* tree=NULL;

std::pair<std::map<huff_data,std::vector<bool>>,std::vector
<std::pair<huff_data,huff_data>>> thing;

    if((encoded_file.mode & 0x80)){//Uses a single Huffman
table or two

        if(encoded_file.mode & 0x40){//two huff
            tree=CreateHuffTree(encoded_file.data);
        }else{
            std::vector<huff_data>
quantify=encoded_file.data;

quantify.insert(quantify.end(),encoded_file.coefs.begin(),e
ncoded_file.coefs.end());
            tree=CreateHuffTree(quantify);
        }
        std::vector<bool> temp;
        thing=HuffTreeToMap(tree);

        outfile.put(0xFE);
        VectorToFile(MapToVector(thing),outfile);
    }
    outfile.put(0xFF);
    WriteIndex(encoded_file.data.size(),outfile);

    if((encoded_file.mode & 0x80)){//Uses a single Huffman
```

```
table or two
        DataToFile(encoded_file.data,thing.first,outfile);
    }else{
        for(auto & el : encoded_file.data){
            outfile.put(el);
        }
    }

    if(encoded_file.isRLE()){
        if(encoded_file.mode & 0x40){ //If This has a
second codification
            tree=CreateHuffTree(encoded_file.coefs);
            thing.first.clear();
            std::vector<bool> temp;
            thing=HuffTreeToMap(tree);
            outfile.put(0xFE);
            VectorToFile(MapToVector(thing),outfile);
        }
        outfile.put(0xFD);
        WriteIndex(encoded_file.coefs.size(),outfile);

        if((encoded_file.mode & 0x80) or (encoded_file.mode
& 0x40)){//Uses huffman

DataToFile(encoded_file.coefs,thing.first,outfile);
        }else{
            for(auto & el : encoded_file.coefs){
            outfile.put(el);
            }
        }
    }

    outfile.close();



}
```

huff.h

```
#ifndef HUFF
#define HUFF

#include <vector>


typedef uint8_t huff_data;
const huff_data huff_null=-1;
typedef int index;


struct CodedFile{
```

```
        uint8_t mode;
        uint8_t order;
        index x_size,y_size;
        std::vector<huff_data> data;
        std::vector<huff_data> coefs;
        bool isRLE();
        bool isDelta();

    };


    CodedFile ReadData(std::string & filename);
    void WriteData(CodedFile encoded_file, std::string &
    filename);


    #endif
```

main.cpp

```
    #include <iostream>
    #include <vector>
    #include <string>
    #include <fstream>
    #include <unistd.h>
    #include "huff.h"

    typedef int index;

    namespace modes{
        const int row_major=0;
        const int column_major=1;
        const int row_major_scan_1=8;
        const int row_major_scan_2=9;
        const int row_major_scan_3=10; //not yet implemented
        const int row_major_scan_4=11; //        ""
        const int column_major_scan_1=12;
        const int column_major_scan_2=13;
        const int column_major_scan_3=14; //not yet
    implemented
        const int column_major_scan_4=15; //        ""
        const int diag_uR_1=16;
        const int zig_zag_1=40;
        const int zig_zag_2=41;
    }

    class ImageOrderIterator{

        private:
            index x,y,x_size,y_size,i;
            uint8_t mode;
```

```
bool valid=true;
bool toggle;
bool IsTop(){ return y==0; }
bool IsBottom(){ return y==(y_size-1);}
bool IsLeft(){return x==0;}
bool IsRight(){return x==(x_size-1);}


void next_row_major(){
    ++i;
    x=i%x_size;
    y=i/x_size;
}
void next_col_major(){
    ++i;
    x=i/y_size;
    y=i%y_size;
}


bool & zig_zag_1_dir=toggle;
const bool DL=false;
const bool UR=true;

void nextZigZag1(){
    if(IsBottom() and IsRight()) valid=false;
    if(zig_zag_1_dir==DL){
        if(IsBottom()){
            ++x;
            zig_zag_1_dir=UR;
        }else if(IsLeft()){
            ++y;
            zig_zag_1_dir=UR;
        }else{
            --x;
            ++y;
        }
    }else{
        if(IsRight()){
            ++y;
            zig_zag_1_dir=DL;
        }else if(IsTop()){
            ++x;
            zig_zag_1_dir=DL;
        }else{
            ++x;
            --y;
        }

    }

}
```

```
const bool L=false;
const bool R=true;
const bool D=false;
const bool U=true;
bool & scan_dir=toggle;

void nextRowMajorScan1(){
    if(scan_dir==R){
        if(IsRight()){
            ++y;
            scan_dir=L;
        }else{
            ++x;
        }
    }else{
        if(IsLeft()){
            ++y;
            scan_dir=R;
        }else{
            --x;
        }

    }

}

void nextColMajorScan1(){
    if(scan_dir==U){
        if(IsTop()){
            ++x;
            scan_dir=D;
        }else{
            --y;
        }
    }else{
        if(IsBottom()){
            ++x;
            scan_dir=U;
        }else{
            ++y;
        }

    }

}

bool & diag_on_x=toggle;
index past_init_pos;
void nextDiagUR1(){
    if(not diag_on_x){
```

```
                    if(IsTop() or IsRight()){
                        x=0;
                        ++past_init_pos;
                        y=past_init_pos;

                        if(y==y_size){
                            --y;
                            ++x;
                            diag_on_x=true;
                            past_init_pos=1;
                        }
                    }else{
                        ++x;
                        --y;
                    }
                }else{
                    if(IsTop() or IsRight()){
                        y=y_size-1;
                        ++past_init_pos;
                        x=past_init_pos;

                    }else{
                        ++x;
                        --y;
                    }

                }
            }

        public:
            ImageOrderIterator(index y_size_s, index x_size_s,
    uint8_t mode_s){
                x_size=x_size_s;
                y_size=y_size_s;
                mode=mode_s;

                switch (mode){

                    case modes::row_major://row major
                        x=0;y=0;i=0;
                        break;

                    case modes::column_major: //col major
                        x=0;y=0;i=0;
                        break;

                    case modes::zig_zag_1:
                        x=0;y=0;zig_zag_1_dir=DL;
                        break;
                    case modes::zig_zag_2:
                        x=0;y=0;zig_zag_1_dir=UR;
                        break;
```

```
                    case modes::row_major_scan_1:
                        x=0;y=0;scan_dir=R;
                        break;

                    case modes::row_major_scan_2:
                        x=x_size-1;y=0;scan_dir=L;
                        break;

                    case modes::column_major_scan_1:
                        x=0;y=0;scan_dir=D;
                        break;

                    case modes::column_major_scan_2:
                        x=0;y=y_size-1;scan_dir=U;
                        break;

                    case modes::diag_uR_1:
                        x=0;y=0;
                        past_init_pos=0;
                        diag_on_x=false;
                        break;

                    default:
                        throw("Mode not implemented on init");

                }

            }

            index num(){
                if(valid) return x_size*y+x;
                else throw("Tried to access non existent
    iteration");

            }

            void next(){
                switch (mode){
                    case modes::row_major:
                        next_row_major();
                        break;

                    case modes::column_major:
                        next_col_major();
                        break;

                    case modes::zig_zag_1:
                    case modes::zig_zag_2:
                        nextZigZag1();
                        break;
```

```
                    case modes::row_major_scan_1:
                    case modes::row_major_scan_2:
                        nextRowMajorScan1();
                        break;

                    case modes::column_major_scan_1:
                    case modes::column_major_scan_2:
                        nextColMajorScan1();
                        break;

                    case modes::diag_uR_1:
                        nextDiagUR1();
                        break;

                    default:
                        throw("Mode not implemented on next");

                }

            }


};

struct picture{
    std::vector<uint8_t> bitmap;
    index x_size, y_size;
    inline const index size(){
        return x_size*y_size;
    }
};

picture ImageToReorder(picture & image, uint8_t mode){
    picture reordered_image;
    reordered_image.bitmap.resize(image.size());
    reordered_image.x_size=image.x_size;
    reordered_image.y_size=image.y_size;
    ImageOrderIterator
iterador(image.y_size,image.x_size,mode);
    for(int i=0;i<image.size();++i){

reordered_image.bitmap[i]=image.bitmap[iterador.num()];
        iterador.next();
    }
    return reordered_image;
}

picture ReorderToImage(picture & reordered_image, uint8_t
mode){
    picture image;
    image.bitmap.resize(reordered_image.size());
```

```
    image.x_size=reordered_image.x_size;
    image.y_size=reordered_image.y_size;
    ImageOrderIterator
iterador(reordered_image.y_size,reordered_image.x_size,mod
e);
    for(int i=0;i<(reordered_image.size());++i){

image.bitmap[iterador.num()]=reordered_image.bitmap[i];
        iterador.next();
    }
    return image;
}

picture FileToPicture(std::string filename){
    std::ifstream infile(filename,std::ios_base::binary);
    if(infile.get()!='P') throw("File is not a Netpbm
bitmap");
    if(infile.get()!='5') throw("File is not greyscale");
    picture image;
    int color_depth;
    infile>>image.x_size>>image.y_size>>color_depth;
    infile.get(); //clear newline

    if(color_depth!=255) throw ("Wrong colordepth");
    image.bitmap.resize(image.x_size*image.y_size);
    for(int i=0;i<image.x_size*image.y_size;++i){
        image.bitmap[i]=infile.get();
    }
    infile.close();
    return image;
}

void PictureToFile(std::string filename, picture image){
    std::ofstream outfile(filename,std::ios_base::binary);
    outfile<<"P5"<<std::endl;
    outfile<<image.x_size<<' ';
    outfile<<image.y_size<<std::endl;
    outfile<<(int)255<<std::endl;
    for(int i=0;i<image.x_size*image.y_size;++i){
        outfile.put(image.bitmap[i]);
    }
    outfile.close();
}

void RLE_Encode(const std::vector<uint8_t> &
bitmap,std::vector<huff_data> & data,std::vector<huff_data>
& coefs){
    index max_pos=bitmap.size();
    index i=0;
    while(i<max_pos){
        huff_data count =0;
```

```
            huff_data value=bitmap[i];
            ++i;
            if(i==max_pos){
                goto past_while;
            }
            while(count <255 and i<max_pos and
    bitmap[i]==value){
                ++count;
                ++i;
            }

            past_while:

            coefs.push_back(count);
            data.push_back(value);
        }
    }

    std::vector<uint8_t> RLE_Decode(const
    std::vector<huff_data> & data,const std::vector<huff_data> &
    coefs,index imagesize){
        std::vector<uint8_t> answer;
        index i=0;
        if(coefs.size()!=data.size()){
            throw("Bad RLE Encoded");
        }
        int m=0;
        answer.resize(imagesize);
        while(i<data.size()){
            index j=coefs[i]+1;
            for(index k=0;k<j;++k){
                answer[m]=data[i];
                ++m;
            }
            ++i;

        }
        return answer;
    }

    std::vector<uint8_t> DeltaEncode(const
    std::vector<uint8_t> & bitmap){
        uint8_t prev_pixel=0;
        std::vector<uint8_t> answer(bitmap.size());
        for(int i=0;i<bitmap.size();++i){
            answer[i]=bitmap[i]-prev_pixel;
            prev_pixel=bitmap[i];
        }
        return answer;
    }

    std::vector<uint8_t> DeltaDecode(const
```

```
std::vector<uint8_t> & bitmap){
    uint8_t prev_pixel=0;
    std::vector<uint8_t> answer(bitmap.size());
    for(int i=0;i<bitmap.size();++i){
        answer[i]=bitmap[i]+prev_pixel;
        prev_pixel=answer[i];
    }
    return answer;
}


CodedFile Encode(picture image,uint8_t mode,uint8_t order)
{
    CodedFile answer;
    answer.mode=mode;
    answer.order=order;
    answer.x_size=image.x_size;
    answer.y_size=image.y_size;
    picture reordered_image=ImageToReorder(image,order);
    if(answer.isDelta()){
        image.bitmap=DeltaEncode(reordered_image.bitmap);
    }else{
        image=reordered_image;
    }
    if(answer.isRLE()){
        RLE_Encode(image.bitmap,answer.data,answer.coefs);
    }else{
        answer.data=image.bitmap;
    }
    return answer;
}

picture Decode(CodedFile coded_file){
    picture answer;
    answer.x_size=coded_file.x_size;
    answer.y_size=coded_file.y_size;
    std::vector<huff_data> temp;
    if(coded_file.isRLE()){

temp=RLE_Decode(coded_file.data,coded_file.coefs,answer.size
());
    }else{
        temp=coded_file.data;
    }
    if(coded_file.isDelta()){
        answer.bitmap=DeltaDecode(temp);
    }else{
        answer.bitmap=temp;
    }
    return ReorderToImage(answer,coded_file.order);
```

```
}


int main(int argc, char *argv[]){
    static const char *optString = "edRDH:p:";
    try{
        bool encode=false;
        if(argc>=3){
            int opt = getopt( argc, argv, optString );
            uint8_t mode=0x00;
            uint8_t order=0x00;
            uint8_t huffman=0x00;
            bool RLE=false;
            bool Delta=false;
            while(opt!=-1){
                switch(opt){
                    case 'e':
                        encode=true;
                        break;
                    case 'd':
                        break;
                    case 'R':
                        RLE=true;
                        break;
                    case 'D':
                        Delta=true;
                        break;
                    case 'H':
                        huffman=atoi(optarg);
                        break;
                    case 'p':
                        order=atoi(optarg);

                }

                opt = getopt( argc, argv, optString );
            }
            if(RLE) mode |= 0x02;
            if(Delta) mode |= 0x01;
            mode |= huffman<<6;

            if(optind==(argc-2)){
                std::string in_filename=argv[optind];
                std::string out_filename=argv[optind+1];
                if(encode){
                    CodedFile
c_file=Encode(FileToPicture(in_filename),mode,order);
                    WriteData(c_file,out_filename);
                }else{

PictureToFile(out_filename,Decode(ReadData(in_filename)));
                }
```

```
            }else{
                std::cout<<"Usage: program -[e|d] -(D|R|H)
(h_num)input_file output_file"<<std::endl;
            }


        }else{
            std::cout<<"Usage: program -[e|d] -(D|R|H)
(h_num)input_file output_file"<<std::endl;

        }
    }catch(char const * msg){
        std::cerr << msg << std::endl;
        return EXIT_FAILURE;
    }

    return EXIT_SUCCESS;
}
```

Programa secundario (tester.py)

```
import os
import subprocess

cases=((["-e","-H","2","-p","0"],"H2_0"),
           (["-e","-R","-H","3","-p","0"],"RH3_0"),(["-
e","-R","-H","3","-p","1"],"RH3_1"),(["-e","-R","-
H","3","-p","16"],"RH3_16"),(["-e","-R","-H","3","-
p","40"],"RH3_40"),
           (["-e","-D","-H","2","-p","0"],"DH2_0"),(["-
e","-D","-H","2","-p","1"],"DH2_1"),(["-e","-D","-
H","2","-p","16"],"DH2_16"),(["-e","-D","-H","2","-
p","40"],"DH2_40"),
           (["-e","-D","-R","-H","3","-p","0"],"DRH3_0"),
(["-e","-D","-R","-H","3","-p","1"],"DRH3_1"),(["-e","-
D","-R","-H","3","-p","16"],"DRH3_16"),(["-e","-D","-R","-
H","3","-p","40"],"DRH3_40"))


def do_cases(filename,dir_f):

    pgmrute="pgm"+"/"+dir_f+"/"+filename[:-4]+".pgm"

    rst=[]

    for case in cases:
        casename=filename[:-4]+case[1]
        print(casename)
        fmpgrute="fpgm"+"/"+dir_f+"/"+casename+".fpgm"
        cpgmrute="cpgm"+"/"+dir_f+"/"+casename+".pgm"
```

```
        subprocess.call(["./tdr2"]+case[0]+[pgmrute]+
[fmpgrute])
        subprocess.call(["./tdr2","-d"]+[fmpgrute]+
[cpgmrute])

hash1=int(subprocess.check_output(["sha1sum",cpgmrute]).de
code("utf-8").split()[0],16)

hash2=int(subprocess.check_output(["sha1sum",pgmrute]).dec
ode("utf-8").split()[0],16)

        if (hash1!=hash2):
            print("On file {} in case {} hash1={:x}
hash2={:x}".format(filename,case[1],hash1,hash2))

        rst.append(subprocess.check_output(["wc","-
c",fmpgrute]).decode("utf-8").split()[0])
    return rst



folders=["text","natureplaces","artificial"]

FNULL=open(os.devnull, 'w')

stats=open("stats.csv","w")

filestats=[]

stats.write("\t".join(["Filename","PNG","pgm"]+[ pair[1]
for pair in cases])+"\n")

subprocess.call(["rm","-r","pgm","fpgm","cpgm"])
subprocess.call(["mkdir","pgm","fpgm","cpgm"])

for fold in ["pgm","fpgm","cpgm"]:
    for dir_f in folders:
        subprocess.call(["mkdir",fold+"/"+dir_f])

for dir_f in folders:
    for filen in os.listdir(os.fsencode("png/"+dir_f)):
        filename = os.fsdecode(filen)
        currfilestat=[]
        if filename.endswith(".png"):
            subprocess.run(["ffmpeg","-
i","png"+"/"+dir_f+"/"+filename,"pgm"+"/"+dir_f+"/"+filename
[:-4]+".pgm"],capture_output=True)

            pngsize=subprocess.check_output(["wc","-
c","png"+"/"+dir_f+"/"+filename]).decode("utf-8").split()
```

```
[0]
            pgmsize=subprocess.check_output(["wc","-
c","pgm"+"/"+dir_f+"/"+filename[:-4]+".pgm"]).decode("utf-
8").split()[0]
            line=[filename,pngsize,pgmsize]
+do_cases(filename,dir_f)
            stats.write("\t".join(line)+"\n")
        else:
            continue

print("Pbm done")
```