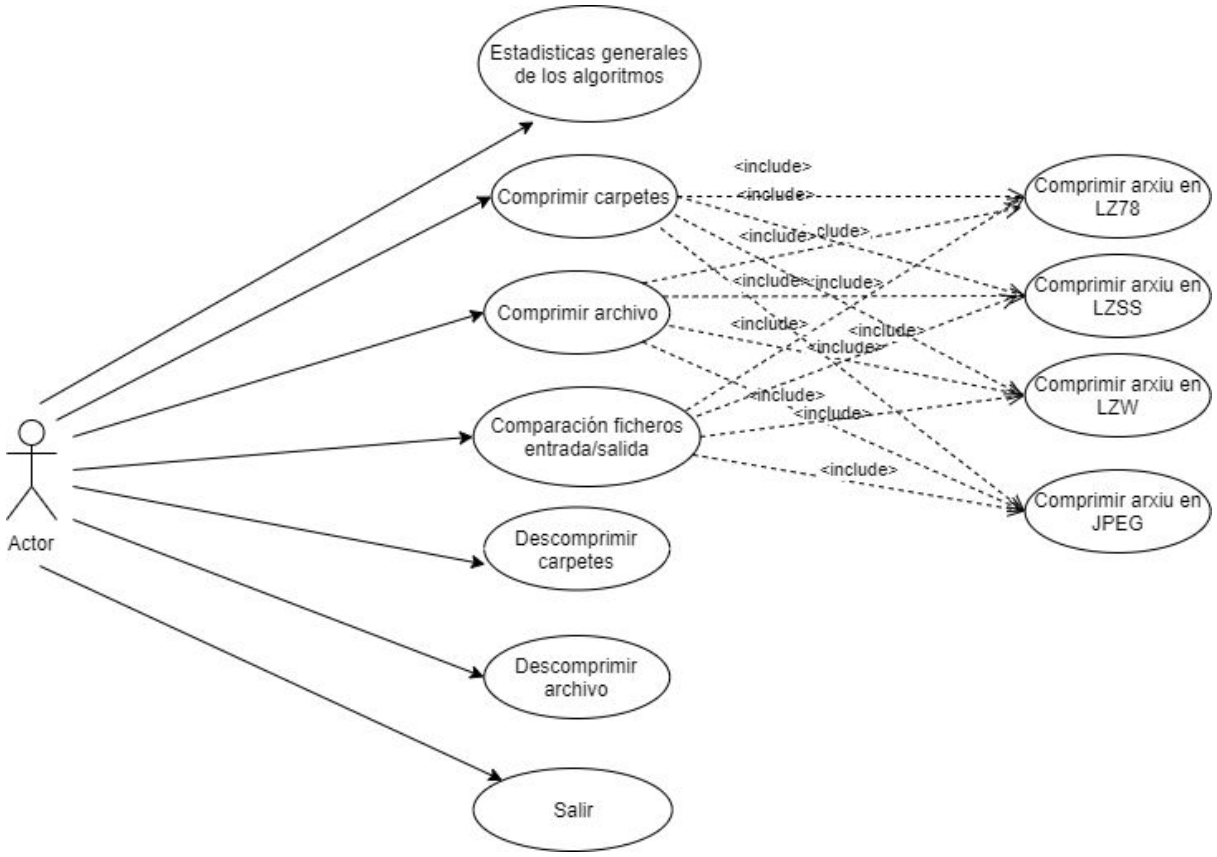
PROP

Entrega 2

García Sánchez, Ivan
Murciano Julia, Pau
Navarro Albarracin,Juan Jose
Pinilla Sanchez,Lucas

Diagrama casos d'ús



Lista de funcionalidades

Funcionalidades principales

Comprimir archivos en LZ78: Se comprime un archivo con el algoritmo LZ78 que. La extensión resultante será xx.lz78.

Comprimir archivos en LZSS: Se comprime un archivo con el algoritmo LZSS. La extensión resultante será xx.lzss.

Comprimir archivos en LZW: Se comprime un archivo con el algoritmo LZW. La extensión resultante será xx.lzw.

Comprimir archivos en JPEG: Se comprime un archivo con el algoritmo JPEG. La extensión resultante será xx.jimg.

Comprimir archivo: El usuario pasa como input el path, escoge un algoritmo y indica si quiere guardarlo. Si el fichero no es xx.txt o xx.ppm, salta una excepcion. Si se ha seleccionado guardar, lo guarda en el sistema. Como output se muestran unas estadísticas básicas (tiempo de compresión, porcentaje de compresión y tiempo tardado).

Descomprimir archivos: El usuario pasa como input un path de un fichero comprimido y el sistema automáticamente detecta el algoritmo con el que fue comprimido y lo descomprime. Si el archivo tiene una extensión que no es lz78, lzss, lzw o jimg salta un error. Como output salen las estadisticas del fichero descomprimido y el archivo descomprimido se ubica en el mismo directorio que el comprimido.

Comprimir carpetas: el usuario pasa como input una carpeta y como output sale la misma carpeta con todo lo que contenía comprimido. El resultado es un único archivo comprimido que dentro tiene todos los archivos de la carpeta.El usuario tiene que escoger con qué algoritmo comprimir los archivos de texto y las imágenes. Puede elegir entre los siguientes algoritmos: LZ78, LZSS, LZW, JPEG. La extensión resultante sera xx.cpta.

Descomprimir carpetas: el usuario pasa como input un archivo comprimido de tipo carpeta (xx.cpta) y si no es de este tipo salta un error.Como output sale la carpeta descomprimido.

Comparacion fichero entrada/salida: El usuario inserta el path que quiere procesar, intoduce el algoritmo con el cual comprimirlo/descomprimirlo i el programa muestra por pantalla si el original y el procesado son iguales.

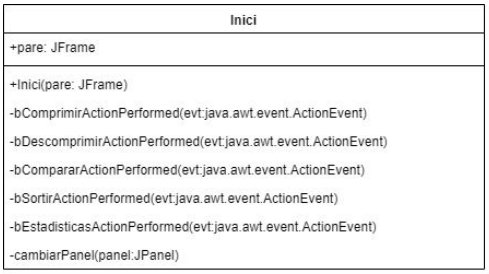
Salir: Se cierra el programa.

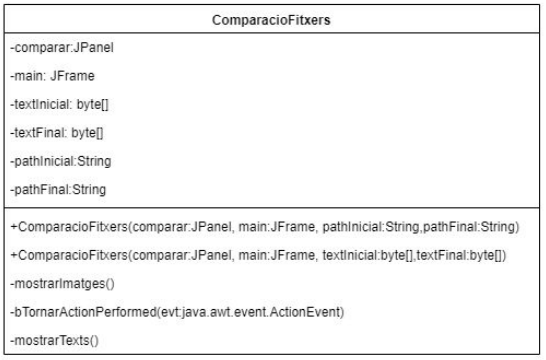
Funcionalidades secundarias

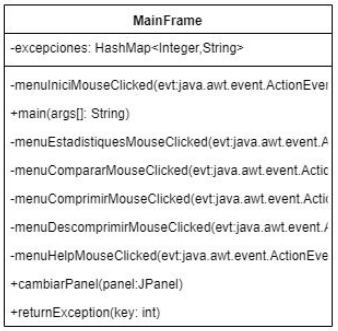
Estadísticas generales de los algoritmos: El usuario accede a la información estadística de todos los algoritmos segun lo que haya comprimido/descomprimido desde el inicio hasta el momento.

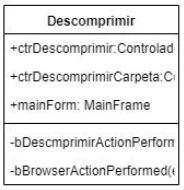
DIAGRAMA DE CLASES

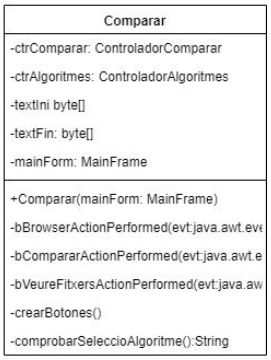
Capa de presentació

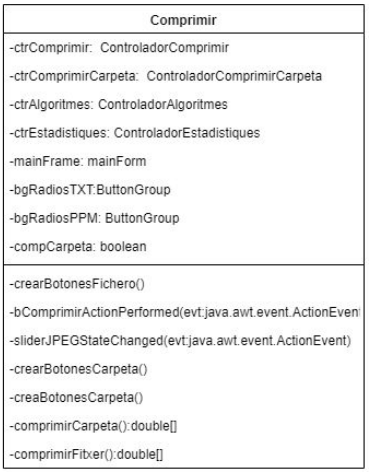




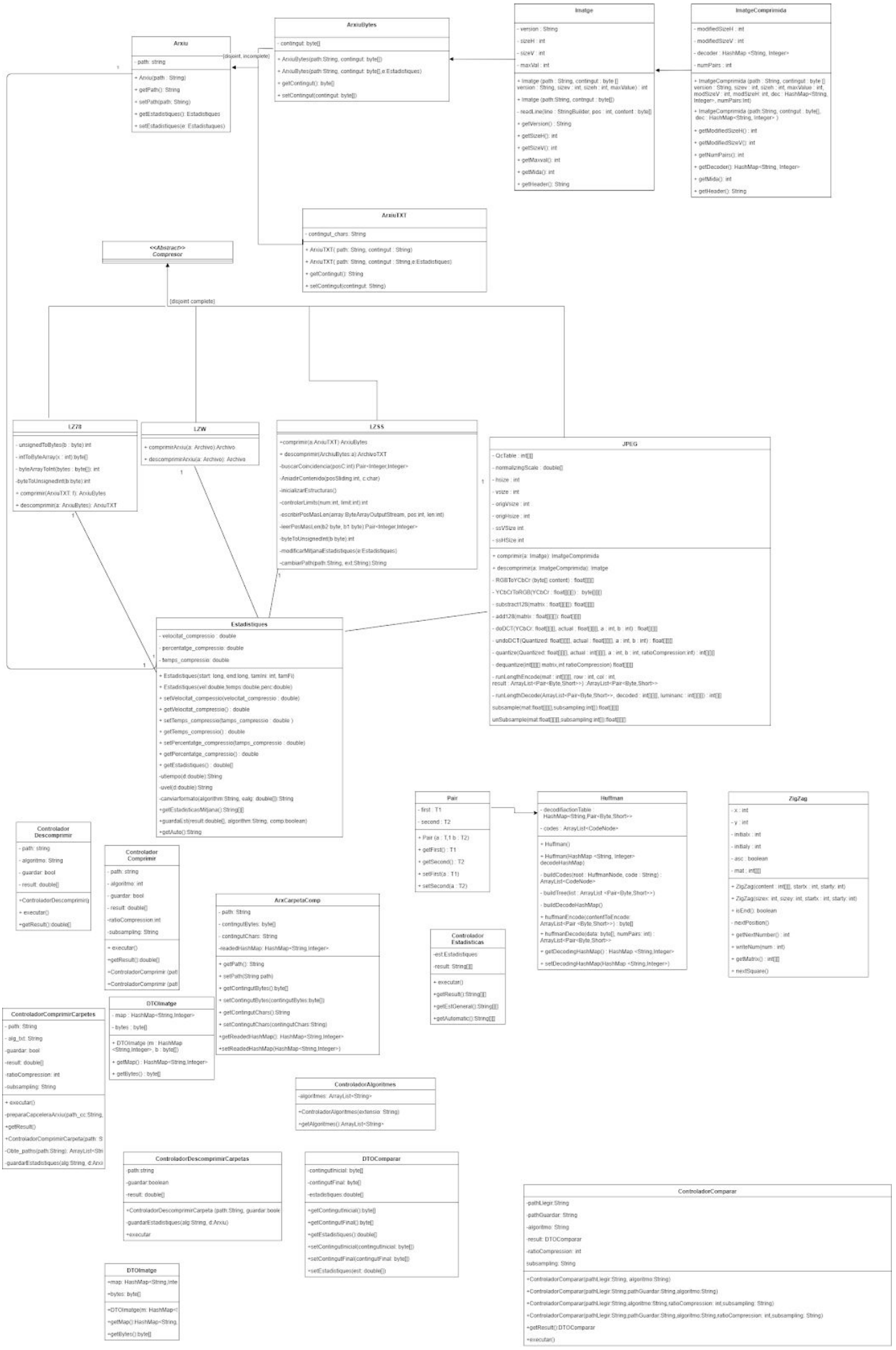




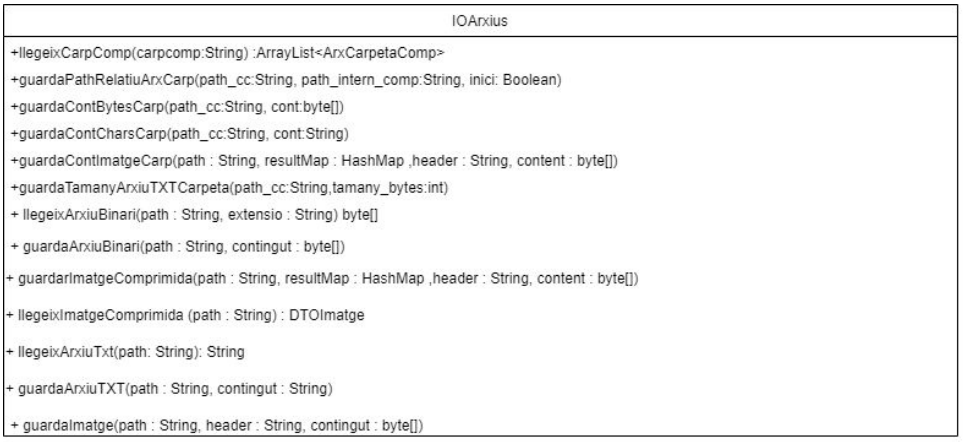


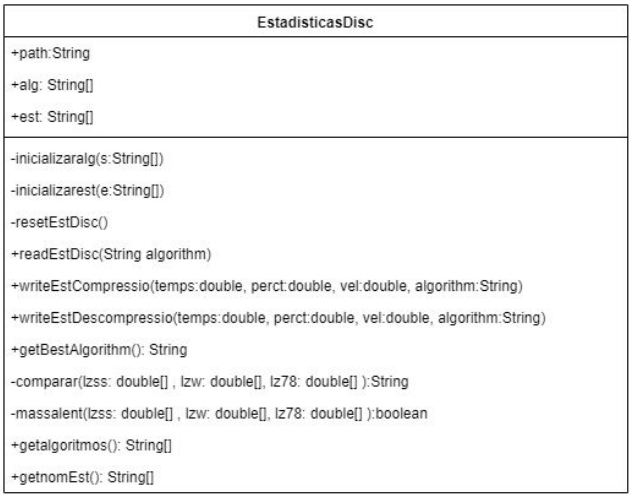


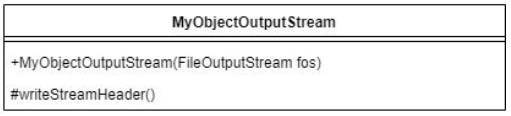
Capa de domini


Arxiu
- path: string
+ Arxiu(path : String)
+ getPath(): String
+ setPath(path: String)
+ getEstadistiques(): Estadistiques
+ setEstadistiques(e: Estadistiques)
[disjoint, incomplete]
ArxiuBytes
- contingut: byte[]
+ ArxiuBytes(path:String, contingut: byte[])
+ ArxiuBytes(path:String, contingut: byte[],e:Estadistiques)
+ getContingut(): byte[]
+ setContingut(contingut: byte[])
Imatge
- version : String
- sizeH : int
- sizeV : int
- maxVal : int
+ Imatge (path : String, contingut : byte [] version : String, sizev : int, sizeh : int, maxValue) : int
+ Imatge (path:String, contingut : byte[])
- readLine(line : StringBuilder, pos : int, content : byte[]
+ getVersion() : String
+ getSizeH(): int
+ getSizeV(): int
+ getMaxval(): int
+ getMida(): int
+ getHeader(): String
ImatgeComprimida
- modifiedSizeH : int
- modifiedSizeV : int
- decoder : HashMap <String, Integer>
- numPairs : int
+ ImatgeComprimida (path : String, contingut : byte [] version : String, sizev : int, sizeh : int, maxValue : int, modSizeV : int, modSizeH: int, dec : HashMap<String, Integer>, numPairs:int)
+ ImatgeComprimida (path:String, contingut : byte[], dec : HashMap<String, Integer>)
+ getModifiedSizeH() : int
+ getModifiedSizeV(): int
+ getNumPairs(): int
+ getDecoder(): HashMap<String, Integer>
+ getMida(): int
+ getHeader(): String
ArxiuTXT
- contingut_chars: String
+ ArxiuTXT(path: String, contingut : String)
+ ArxiuTXT(path: String, contingut : String,e:Estadistiques)
+ getContingut(): String
+ setContingut(contingut: String)
<<Abstract>>
Compresor
[disjoint complete]
LZ78
- unsignedToBytes(b : byte):int
- intToByteArray(x : int):byte[]
- byteArrayToInt(bytes : byte[]): int
-byteToUnsignedInt(b:byte):int
+ comprimir(ArxiuTXT, f): ArxiuBytes
+ descomprimir(a: ArxiuBytes): ArxiuTXT
LZW
+ comprimirArxiu(a: Archivo):Archivo
+ descomprimirArxiu(a: Archivo): Archivo
LZSS
+comprimir(a:ArxiuTXT):ArxiuBytes
+ descomprimir(ArchivBytes a):ArchivoTXT
-buscarCoincidencia(posC:int):Pair<Integer,Integer>
-AniadirContenido(posSliding:int, c:char)
-inicializarEstructuras()
-controlarLimits(num:int, limit:int):int
-escribirPosMasLen(array:ByteArrayOutputStream, pos:int, len:int)
-leerPosMasLen(b2 byte, b1 byte):Pair<Integer,Integer>
-byteToUnsignedInt(b byte):int
-modificarMitjanaEstadistiques(e:Estadistiques)
-cambiarPath(path:String, ext:String):String
JPEG
- QcTable : int[][]
- normalizingScale : double[]
- hsize : int
- vsize : int
- origVsize : int
- origHsize : int
- ssVSize int
- ssHSize int
+ comprimir(a: Imatge): ImatgeComprimida
+ descomprimir(a: ImatgeComprimida): Imatge
- RGBToYCbCr (byte[] content) : float[][][]
- YCbCrToRGB(YCbCr : float[][][]) : byte[][][]
- substract128(matrix : float[][][]): float[][][]
- add128(matrix : float[][][]): float[][][]
- doDCT(YCbCr: float[][][], actual : float[][][], a : int, b : int) : float[][][]
- undoDCT(Quantized: float[][][], actual : float[][][], a : int, b : int) : float[][][]
- quantize(Quantized: float[][][], actual : int[][][], a : int, b : int, ratioCompression:int) : int[][][]
- dequantize(int[][][] matrix,int ratioCompression) float[][][]
- runLengthEncode(mat : int[][], row : int, col : int, result : ArrayList<Pair<Byte,Short>>) :ArrayList<Pair<Byte,Short>>
- runLengthDecode(ArrayList<Pair<Byte,Short>>, decoded : int[][], luminanc : int[][]) : int[][]
subsample(mat:float[][],subsampling:int[]):float[][]
unSubsample(mat:float[][],subsampling:int[]):float[][]
Estadistiques
- velocitat_compressio : double
- percentatge_compressio: double
- temps_compressio: double
+ Estadistiques(start: long, end:long, tamIni: int, tamFi)
+ Estadistiques(vel:double,temps:double,perc:double)
+ setVelocitat_compressio(velocitat_compressio : double)
+ getVelocitat_compressio() : double
+ setTemps_compressio(tamps_compressio : double)
+ getTemps_compressio() : double
+ setPercentatge_compressio(tamps_compressio : double)
+ getPercentatge_compressio() : double
+ getEstadistiques() : double[]
-utemps(d:double):String
-uvel(d:double):String
-canviarformato(algorithm:String, ealg: double[]):String
+getEstadisticasMitjana():String[][]
+guardaEst(result:double[], algorithm:String, comp:boolean)
+getAuto():String
Pair
- first : T1
- second : T2
+ Pair (a : T,1 b : T2)
+ getFirst() : T1
+ getSecond() : T2
+ setFirst(a : T1)
+ setSecond(a : T2)
Huffman
- decodifiactionTable : HashMap<String,Pair<Byte,Short>>
- codes : ArrayList<CodeNode>
+ Huffman()
+ Huffman(HashMap <String, Integer> decodeHashMap)
- buildCodes(root : HuffmanNode, code : String) : ArrayList<CodeNode>
- buildTree(list : ArrayList <Pair<Byte,Short>>)
- buildDecodeHashMap()
+ huffmanEncode(contentToEncode: ArrayList<Pair <Byte,Short>>) : byte[]
+ huffmanDecode(data: byte[], numPairs: int) : ArrayList<Pair<Byte,Short>>
+ getDecodingHashMap() : HashMap <String,Integer>
+ setDecodingHashMap(HashMap <String,Integer>)
ZigZag
- x : int
- y : int
- initialx : int
- initialy : int
- asc : boolean
- mat : int[][]
+ ZigZag(content : int[][], startx : int, starty: int)
+ ZigZag(sizex: int, sizey: int, startx : int, starty: int)
+ isEnd(): boolean
- nextPosition()
+ getNextNumber() : int
+ writeNum(num : int)
+ getMatrix() : int[][]
+ nextSquare()
Controlador Descomprimir
- path: string
- algoritmo: String
- guardar: bool
- result: double[]
+ControladorDescomprimir()
+ executar()
+getResult():double[]
Controlador Comprimir
- path: string
- algoritmo: int
- guardar: bool
- result: double[]
-ratioCompression:int
-subsampling: String
+ executar()
+getResult():double[]
+ControladorComprimir (path
+ControladorComprimir (path
ArxCarpetaComp
- path: String
- contingutBytes: byte[]
- contingutChars: String
-readedHashMap: HashMap<String,Integer>
+ getPath(): String
+ setPath(String path)
+ getContingutBytes():byte[]
+ setContingutBytes(contingutBytes:byte[])
+ getContingutChars():String
+ setContingutChars(contingutChars:String)
+getReadedHashMap(): HashMap<String,Integer>
+setReadedHashMap(HashMap<String,Integer>)
Controlador Estadisticas
-est:Estadistiques
-result: String[][]
+ executar()
+getResult():String[][]
+getEstGeneral():String[][]
+getAutomatic():String[][]
ControladorComprimirCarpetes
- path: String
- alg_txt: String
-guardar: bool
-result: double[]
-ratioCompression: int
-subsampling: String
+ executar()
-preparaCapceleraArxiu(path_cc:String,
+getResult()
+ControladorComprimirCarpeta(path: S
-Obte_paths(path:String): ArrayList<Str
-guardarEstadistiques(alg:String, d:Arxi
DTOImatge
- map : HashMap<String,Integer>
- bytes : byte[]
+ DTOImatge (m : HashMap <String,Integer>, b : byte[])
+ getMap() : HashMap<String,Integer>
+ getBytes() : byte[]
ControladorAlgoritmes
-algoritmes: ArrayList<String>
+ControladorAlgoritmes(extensio: String)
+getAlgoritmes():ArrayList<String>
ControladorDescomprimirCarpetas
-path:string
-guardar:boolean
-result: double[]
+ControladorDescomprimirCarpeta (path:String, guardar:boole
-guardarEstadistiques(alg:String, d:Arxiu)
+executar
DTOComparar
-contingutInicial: byte[]
-contingutFinal: byte[]
-estadistiques:double[]
+getContingutInicial():byte[]
+getContingutFinal():byte[]
+getEstadistiques():double[]
+setContingutInicial(contingutInicial: byte[])
+setContingutFinal(contingutFinal: byte[])
+setEstadistiques(est: double[])
DTOImatge
-map: HashMap<String,Inte
+bytes: byte[]
+DTOImatge(m: HashMap<
+getMap():HashMap<String,
+getBytes():byte[]
ControladorComparar
-pathLlegir:String
-pathGuardar: String
-algoritmo: String
-result: DTOComparar
-ratioCompression: int
subsampling: String
+ControladorComparar(pathLlegir:String, algoritmo:String)
+ControladorComparar(pathLlegir:String,pathGuardar:String,algoritmo:String)
+ControladorComparar(pathLlegir:String,algoritmo:String,ratioCompression: int,subsampling: String)
+ControladorComparar(pathLlegir:String,pathGuardar:String,algoritmo:String,ratioCompression: int,subsampling: String)
+getResult():DTOComparar
+executar()

Capa de persistència







En primer lloc vam considerar que els compressors eren una part molt important del nostre sistema així que vam decidir fer una classe per a cada algorisme a representar. Aquestes són les classes: JPEG, LZSS, LZW, LZ78. Aquestes classes només tenen dos mètodes públics: comprimir i descomprimir, i per tant només elles tenen la responsabilitat de fer-ho. Per el procés de JPEG hem hagut de definir tres classes extres, que són Huffman, la qual s'encarrega de fer la codificació huffman de l'algorisme, la classe ZigZag, la qual és una classe auxiliar per al procés de RunLengthEncoding i la classe Pair la qual representa un parell de dades. La classe pair és usada també per l'algorisme LZSS.
En segon lloc vam decidir agrupar-les totes en una superclasse, la qual vam anomenar Compressor.
En tercer lloc, vam crear una classe que representés les estadístiques dels algoritmes. Aquesta classe la hem anomenat estadístiques i és la responsable de guardar les estadístiques de tots els algoritmes. Aquestes estadístiques son: velocitat de compressió (en bytes), percentatge de compressió i el temps mig de compressió.
En quart lloc hem representat a tots el fitxers d'un sistema de fitxers. En primer lloc hem decidit fer una classe que engloba a tots el fitxers. Aquesta classe l'hem anomenat Arxiu i només té el path del arxiu. Un cop definit aquest concepte en el sistema hem separat en dos

tipus de fitxers: ArxiuBytes, el qual representa un arxiu, i el seu contingut està en un array de bytes i ArxiuTXT el qual representa un arxiu, però el seu contingut està representat en una String. Aquestes dues classes hereden de Arxiu ja que representen un fitxer del sistema. Un cop definits aquests conceptes, hem decidit representar les imatges, en una subclasse anomenada Imatge. Aquesta té el header de les imatges PPM i hereda de l'arxiu bytes ja que el seu contingut el representem en un array de bytes. En últim lloc, l'última classe de fitxer que tenim és la imatge comprimida. Aquesta representa una imatge quan ja ha passat per el procés de compressió de JPEG, i té uns paràmetres extres que no té la imatge normal. Aquesta classe, al ser una extensió de la classe Imatge, hem fet que heredi d'ella.

Algorisme LZW

Fet per Ivan Garcia Sanchez

Introducció

Aquest algoritme es un algoritme de compressio sense perdua creat al 1984 per Abraham Lempel, Jacob Ziv i Terry Welch. Es un algoritme molt utilitzat en la compressio d'arxius en Unix i per comprimir imatges en format GIF.

Descripcio de l'algoritme

Proces de compressió

L'algoritme tracta de generar un diccionari de manera dinamica segons les repeticions que troba al text i les assigna un valor de menys tamany que identificara aquesta cadena al text comprimit.

Primer genera un diccionari amb 256 entrades, una per cada caracter ASCII.

Un cop inicialitzat el diccionari, va buscant repeticions de text que ja havien aparegut abans. El metode és el següent:
Llegeix un caracter i el fusiona amb el que havia anteriorment. Aquesta fusio mira si ja te una entrada al diccionari. Si ja la te, aquesta fusio pasa a ser el que havia anteriorment per a que sigui fusionat amb el seguent caracter. Altrament, afegeix una entrada nova al diccionari amb aquesta fusio assignant-li un nou valor diferents a la resta de entrades, afegeix al resultat de la compressio la traduccio del que havia anteriorment i aquest caracter pasa a ser el que hi havia anteriorment.

Aquest diccionari tindra les entrades limitades, perque sino hi haura errors a l'hora d'escriure el contingut en un fitxer.

Quan ja no hi ha mes caracters al contingut, afegeix al resultat la traducció del que havia anteriorment.

Procés de descompressió

El proces de descompressio es generar el diccionari tal i com s'havia generat en el proces de compressio.

Inicialment s'inicialitza el diccionari amb 256 entrades, una per cada caracter ASCII.

Despres llegeix el primer codi de la traduccio, posa al resultat la seva traduccio i guarda el codi en codi antic.

A partir d'aqui ja itera sobre tots els codis de traduccio del contingut. Si el codi nou esta al diccionari, es fica al resultat la seva traduccio i fiquem al diccionari una entrada nova amb un nou codi de compressio i com a contingut la traduccio del codi antic fusionada amb el primer caracter de la traduccio del codi nou. Altrament, l'unic que cambia es que al resultat fiquem la fusio de la traduccio del codi antic i el primer caracter de la traduccio del codi nou. Finalment posem codi nou en codi antic.

Estructura de dades utilitzades

Les estructures de dades que he utilitzat són les seguents:

- **HashMap**: He utilitzat aquest tipus d'estructura de dades per a que faci el paper de diccionari de traduccions als processos de compressio i descompressio.
  He utilitzat el HashMap perque el cost de totes les crides son de cost O(1), aportant-me major velocitat a l'algoritme.
  Podia haver utilitzat TreeMap pero aquesta estructura de dades es mes lenta en les operacions de get i put (O(log(n)) i a mes, no necesito el map ordenat per les claus.
  També podria haber utilitzat TrieMap, però en JDK8, el seu rendiment està per sota del HashMap i, a més, l'avantatge que té aquest sobre el HashMap no la necessito (accedir a tots els elements alhora). A més, com nosaltres fem accessos aleatoris, el HashMap té cost constant i el Trie cost logaritmic.

Pseudocodis

Comprimir

```
1 Initialize table with single character strings
2     P = first input character
3      WHILE not end of input stream
4           C = next input character
5           IF P + C is in the string table
6             P = P + C
```

```
7           ELSE
8             output the code for P
9           add P + C to the string table
10            P = C
11          END WHILE
12      output code for P
```

1- Inicialitzem el diccionari amb una entrada per cada caracter ASCII.
2- Llegim el primer caracter del contingut.
3- Iterem per tot el contingut.
4-Llegim el seguent caracter del contingut.
5-Mires si ja s'havia trobat abans la combinacio de caracters anteriors concatenat amb el caracter actual.
6- la combinacio de caracters anteriors passa a ser la que hi havia concatenada amb el caracter actual.
7- Tractament si no es troba al diccionari la combinacio de caracters anteriors concatenat amb el caracter actual.
8-Fiques la traduccio de la concatenacio de caracters anteriors al resultat.
9-Afegeixes a una nova entrada del diccionari la combinacio de caracters anteriors concatenat amb el caracter actual amb un codi diferent a la resta.
10- Posem com a combinacio de caracters anteriors el caracter actual.
11-Tornem al 3 per seguir iterant per tots els caracters del contingut.
12- Un cop ja hem recorregut tot el contingut, fiquem la traducció de la concatenacio de caracters anteriors al resultat.

<u>Descomprimir</u>

```
1 Initialize table with single character strings
2     OLD = first input code
3     output translation of OLD
4     WHILE not end of input stream
5         NEW = next input code
6         IF NEW is not in the string table
7               S = translation of OLD
8               S = S + C
9         ELSE
10              S = translation of NEW
11        output S
12        C = first character of S
13        OLD + C to the string table
14        OLD = NEW
15    END WHILE
```

1- Inicialitzem el diccionari amb una entrada per cada caracter ASCII.
2-Llegim el primer codi del contingut comprimit.
3-Afegim al resultat la traduccio d'aquest codi, es a dir, el contingut original.
4-Iterem per cada codi del contingut comprimit.

5-Llegim el nou codi del contingut.
6- Mirem si el nou codi ja esta al diccionari.
7- Fiquem a una variable S la traduccio corresponent al codi antic.
8- Concatenem la traduccio del codi antic amb el primer caracter de la traducció de codi antic.
9-Mirem si el codi nou no esta al diccionari.
10-Fiquem a una variable S la traduccio corresponent al codi nou.
11-Afegim al resultat el conjunt de caracters de la variable S.
12-Conseguim primer caracter que s'ha afegit al resultat.
13-Fiquem en una entrada nova del diccionari amb la traduccio del codi vell mes el caracter aconseguit a la linea 12.
14-Assignem el codi nou a codi vell.
15- Repetim per a tots els codis del contingut comprimit.

Algorisme LZ78

Fet per Lucas Pinilla Sánchez

Introducció

El LZ78 es un algorisme de compressio de dades sense perduda basat en el LZ77. El seu funcionament es basa guardar una sequencia de caracters que apareixen en un text en una estructura de dades que es crea tant en el metode comprimir com descomprimir i es simetrica.

Descripcio del algorisme

Compressio

Aquest algorisme recorre el text i genera un diccionari que va omplint a mesura que es troba amb una sequencia de caracters no reconeguda previament.
Si hi ha un caracter que no es troba al diccionari l'emagatzema i es s'escriu al arxiu comprimit amb la posició 0. En cas contrari, es guarda la posicio del diccionari on es troba i s'accedeix al seguent caracter, així repetidament fins que la sequencia no es troba dins del diccionari, llavors enmagatzema la ultima lletra conjuntament amb la posicio on es troba la resta de la sequencia.

Per optimitzar el codi s'ha fet servir un byte anomenat flag o bandera que ens permet saber si la posicio es pot codificar amb un sol byte o si pel contrari el map te mes de 255 entrades i ens calen dos bytes. Aquest byte indica amb un 0 si es pot codifar amb un o un 1 en cas contrari.

Descompressio

Primer de tot es llegueix el byte de flag que ens permet saber com hem de lleguir les seguents 8 parelles de posicio i caracter. Es llegueix 8 parelles calculant la posicio segons el bit de flag que va de major a menor pes.
Si la posicio es 0 s'escriu el caracter i s'afegeix al diccionari.
Si la posicio no es 0, es busca al diccionari la sequencia corresponent a la posicio i se li suma el caracter que l'acompanya, s'escriu i s'afegeix al diccionari la nova sequencia.

El diccionari va creixent a mesura que es llegueix un parell de l'arxiu comprimit i es simetric al diccionari que es crea al proces de compressió.

Estructura de dades utilitzada

Per fer el algorme s'ha ulitzat un hashmap per enmagatzemar tant al proces de compressio com descompressio les sequencies de caracters ja acccedides previament.
He fet servir aquesta estructura de dades ja que he considerat que era que millor s'adapta a les necessitats del algorisme.
També podria haber utilitzat TrieMap, però en JDK8, el seu rendiment està per sota del HashMap i, a més, l'avantatge que té aquest sobre el HashMap no la necessito (accedir a tots els elements alhora). A més, com nosaltres fem accessos aleatoris, el HashMap té cost constant i el Trie cost logaritmic.
En el proces de compressio he fet servir una llista, degut a que la clau del map es la sequencia de caracters i que es interessant poder borrar elements del map d'una posicio determinada per temes d'eficiencia. La llista permet mantenir una relacio durant tot el proces de compressio de la posicio amb la clau del map.

PSEUDOCODI

Compressio

```
begin
        ini diccionario
        while(data not null)
        begin
                actual = inicial +a
                if act in map then
                        ini = a; pos = map.get(pos)
                else map.put(act), write(pos,a)
        end
end
```

Descompresssio

```
begin
        Inicializar diccionario
        while(data not empty)
```

```
        begin
            read pos and character
            insertar nueva frase en diccionario con map(pos) + character
            escribir en el resultado nueva palabra
        end
end
```

Algoritme LZSS

Fet per Juan Jose Navarro Albarracin

Introducció

L'algotitme LZSS (Lempel-Ziv-Storer-Szymanski) es un algoritme de compresio sense perdues derivat del LZ77 i creat en 1982 por James Storer y Thomas Szymanski. Els compressors basats en algorismes sense perdua s'utilitzen quan la informacio a comprimir es critica i no es pot perdre informacio.

Descripcio de l'algorisme

Compressio

Aquest algorisme utilitza una finestra de desplaçament que es divideix en dos parts: un buffer d'anticipacio, que guardara els caracters que volem codificar i un buffer de cerca, que es la finestra on buscarem les repeticions iguals a les del buffer d'anticipacia.
Per a comprimir, codifiquem el text de la seguent manera. Busquem la mateixa sequencia que tenim al buffer d'anticipacio, al buffer de busqueda. Si la trobem codifiquem 2 bytes que contindran la posicio d'inici de la repeticio i la longitud d'aquesta. Si no la trobem, codifiquem un caracter normal, igual que el llegit.
A part d'aquesta codificacio utilitzarem un byte anomenat flag, on guardarem un 0 si no hem codificat el caràcter i un 1 si hem codificat posicio i desplaçament. Com un byte conte 8 bits, podrem codificar 8 repeticions/caracters en cada byte.

Descompressió

Per a descomprimir nomes utilitzarem un buffer de busqueda on anirem buscant les repeticions codificades. Primer es llegeix el byte bandera. Agafem bit a bit per a sapiguer si hem de decodificar una posició+desplaçament o nomes un caracter.
Si el bit es 0 guardarem el caracter al buffer de cerca.
Si el bit es 1 llegim la posicio i el desplaçament. Un cop tenim la posicio anem al buffer de cerca a la posició indicada i llegim tants caracters com indica el desplaçament.

Estructures de dades utilitzades

En aquest algorisme hem utilitzat les seguents estructures de dades:

- **Diccionari:** Hem utilitzat un diccionari per a guardar els diferents caracters que anem guardant al buffer de cerca. L'implementacio d'aquest diccionari es amb apuntadors als caracters seguents, per tant te una estructura d'arbre. Hem fet servir aquesta estructura perque la cerca de la clau te un cost molt petit.

- **Arbre:** Hem utilitzat aquesta estructura per a implementar el diccionari. D'aquesta forma tenim un diccionari estructurat caracter a caracter on cada node es un sol caracter i esta conectat amb el seu consecutiu. Hem fet servir aquesta estructura perque la cerca es fa molt més sencilla i rapida.

- **Llista:** Estructura utilitzada per a implementar els nodes que te cada pare de l'arbre, es a dir, per a saber amb quins caracters es conecta cada caracter. Hem utilitzat llistes perque per a afegir es més rapid que no un array.

- **Array:** Hem utilitzat aquesta diverses vegades amb diferents tipus de dades. Un tipus de dades son char, on guardem els caracters que hem codificat o els que volem codificar. Altre tipus són byte, ja que l'utilitzarem per a guardar tots els bytes que formaran el contingut del fitxer comprimit.

Pseudocodi

Comprimir

```
Begin
Ini vectorSinCodificar;
While(llegit>0)
        If secuencia < MIN_COINCIDENCIA
                Guardo bit carácter
                Guardo carácter
        ELSE
                Guardo bit paraula
```

```
            Guardo paraulaCoincidencia
        ENDIF
        WHILE (leemosCaracteres que hemos leído)
            Añadimos contenido al bufferBusqueda
            Añadimos contenido al windowLeido
        END WHILE
        WHILE (SI hemos llegado al final del fichero)
            Añadimos contenido a búsqueda
        END WHILE
        Escribimos byte bandera
        Escribimos palabras escritas
END WHILE
```

Descompressor

```
Begin
        Inicializamos estructuras
        WHILE (Contenido no se haya leído)
            Cogemos bit bandera
            IF bit == 0
                Escribimos carácter
            ELSE
                Escribimos palabra
            END IF
            IF (Hemos leído 7 bits)
                Cogemos otro byte bandera
            END IF
ENDWHILE END
```

Algoritme JPEG

Fet per Pau Murciano Julia

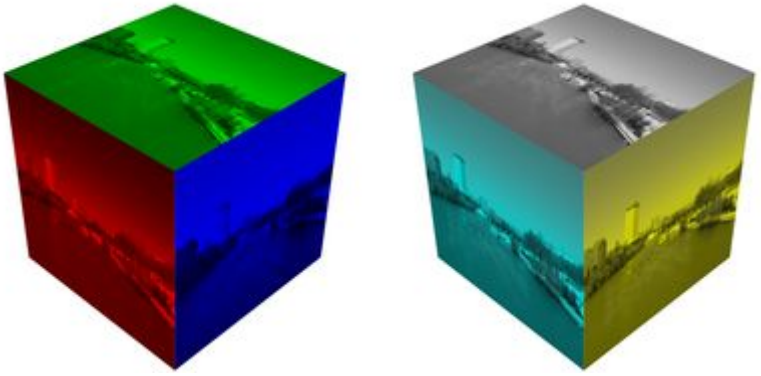
Introducció

L'algorisme JPEG es un algorisme de compresio d'imatges *lossy* (amb perdues) creat per el Joint Photographic Experts Group (JPEG) a l'any 1992. Actualment es fa servir per a poder transportar qualsevol tipus d'imatge amb una mida molt més petita.

Descripcio de l'algorisme

Compressio

En l'estat inicial, es te les tres components originals de la imatge descomprimida (component vermella (R), component verda (G), i component blava (B)) per separat.
En primer lloc, el que es fa es transformar les tres components inicials en una altra base de colors, la base YCbCr (lluminancia (Y), diferencia de color blau(Cb) i diferencia de color vermell(Cr)). Aixo es fa perquè l'ull huma es molt sensible als canvis de llum (Y) pero no als de color (Cb i Cr) aixi que podem comprimir aquestes dues ultimes components.



RGB YCbCr

$$\begin{pmatrix} Y \\ Cb \\ Cr \end{pmatrix} = \begin{pmatrix} 0.299 & 0.587 & 0.114 \\ -0.169 & -0.334 & 0.500 \\ 0.500 & -0.419 & -0.081 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix} + \begin{pmatrix} 0 \\ 128 \\ 128 \end{pmatrix}$$

Fórmula per passar de RGB a YCbCr

En segon lloc, per a poder fer canvis que no noti l'ull huma el que es fa dividir la imatge en matrius de 8 pixels x 8 pixels i mitjançant la DCT (Transformacio Discreta del Cosinus), interpretar cada matriu com una ona.

$$G_{u,v} = \frac{1}{4}\alpha(u)\alpha(v)\sum_{x=0}^{7}\sum_{y=0}^{7} g_{x,y}\cos\left[\frac{(2x+1)u\pi}{16}\right]\cos\left[\frac{(2y+1)v\pi}{16}\right] \qquad \alpha(u) = \begin{cases} \frac{1}{\sqrt{2}}, & \text{if } u = 0 \\ 1, & \text{otherwise} \end{cases}$$

Fórmula de DCT

En tercer lloc, el que es fa es quantitzar el resultat de la matriu, es a dir, eliminar les frequencies altes de cada ona, ja que son les que l'ull huma menys nota. Per fer-ho es divideix cada pixel de les matrius de 8x8 entre el seu pixel corresponent a la matriu de quantitzacio.

$$Q_C = \begin{pmatrix} 17 & 18 & 24 & 47 & 99 & 99 & 99 & 99 \\ 18 & 21 & 26 & 66 & 99 & 99 & 99 & 99 \\ 24 & 26 & 56 & 99 & 99 & 99 & 99 & 99 \\ 47 & 66 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \end{pmatrix}$$

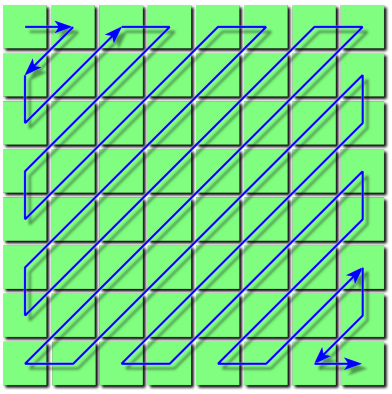
Matriu de quantització

$$F(u,v)_{Quantization} = round\left(\frac{F(u,v)}{Q(u,v)}\right)$$

Fórmula de quantització

En quart i ultim lloc, es fa una codificacio entropica de cada matriu, la qual es divideix en dues parts, el RLE (Run Length Encoding) i aplicar codificació huffman al resultat.

Per aplicar poder fer el Run Length Encoding, primer hem de recorrer la matriu de forma en la que puguem treure mes zeros seguits possibles, ja que eliminarem els 0s. Aquesta forma consisteix en recorrer la matriu en zig-zag de la seguent forma:

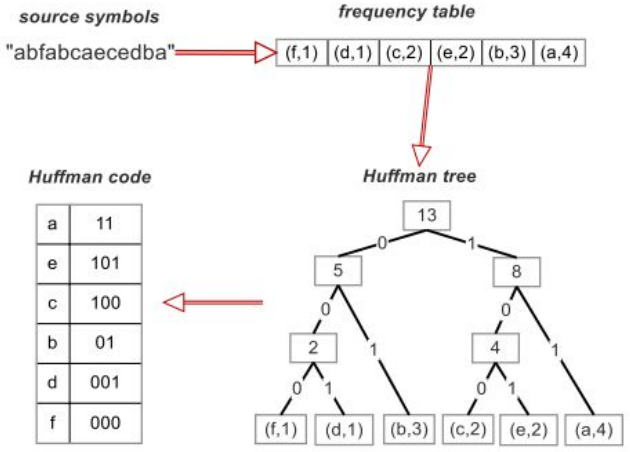


Recorregut en zig-zag d'una matriu

Un cop haguem fet aixo, tindrem una tira de pixels amb uns valors que com mes al final estiguem mes zeros hi haura, aixi que per cada valor que trobem diferent de 0, treurem el seguent parell de valors (Numero de 0s abans del valor, valor). Quan s'arriba al final de la matriu, es posa un parell especial, que es el parell (0,0), que serveix per indicar fi de matriu.

Un cop es te tota la tira de parells es pot aplicar la codificacio huffman al resultat. Per fer-ho el que es fa es comptar quantes vegades apareix cada parell de valors en tota la tira de parells i ordenar per menor frequencia. Quan ja es te ordenat per menor frequencia, es

construeix un arbre de la seguent forma: Cada parell de valors amb la seva frequencia es un node, i mentre hi hagi nodes disponibles i tots no formin un arbre, s'agafen els dos nodes amb menor frequencia i s'ajunten en un que la seva frequencia es la suma dels seus dos nodes fills.
Una vegada ja es te l'arbre format, es crea el codi de codificacio per a cada parell de valors de la seguent forma: Començem al node arrel, i si anem cap a l'esquerra posem un 0 si anem a l'esquerra o un 1 si anem a la dreta. Despres per a cada node fill, es fa el mateix, i s'afegeix cada numero a la dreta de l'anterior. Quan ja es te tots els codis, es posen en una taula i es procedeix a substituir cada parell de valors per el seu codi i es guarda.



Procés de codificació huffman

Descompressio

Per a descomprimir un arxiu només hem de desfer els passos que hem fer per a codificar-lo. En primer lloc, es desfa els codis huffman, es a dir, es va llegint els codis i per a cada codi es guarda el seu parell equivalent.

Un cop es tenen tots els parells, es desfa el RLE recorrent la matriu en zig zag i com que tenim totes les dades (el número de 0s abans de cada valor i el valor que no és 0), es reconstrueix cada matriu de 8 pixels x 8 pixels.

Quan ja es te cada matriu de 8x8, es desquantitza i per tant es multiplica cada numero de la matriu per el seu equivalent a la matriu de quantificaco i es recupera una aproximacio del seu valor real.

$$F(u,v)_{deQ} = F(u,v)_{Quantization} \times Q(u,v)$$

Formula de desquantitzacio

Una vegada es te la imatge desquantitzada, es desfa la DCT que s'ha fet al codificar i s'obte les components YCbCr.

$$f_{x,y} = \frac{1}{4}\sum_{u=0}^{7}\sum_{v=0}^{7}\alpha(u)\alpha(v)F_{u,v}\cos\left[\frac{(2x+1)u\pi}{16}\right]\cos\left[\frac{(2y+1)v\pi}{16}\right]$$

Formula inversa de DCT

Finalment, es passa de YCbCr a RGB i ja es te la imatge descomprimida.

$$\begin{aligned} R &= Y & &+ 1.402 & \cdot(C_R - 128) \\ G &= Y - 0.34414 & \cdot(C_B - 128) &- 0.71414 & \cdot(C_R - 128) \\ B &= Y + 1.772 & \cdot(C_B - 128) & & \end{aligned}$$

Formula per passar de YCbCr a RGB

Estructures de dades utilitzades

Les estructures de dades que he fet servir son les seguents:

- HashMap: L'he fet servir per a guardar la associació dels codis huffman amb el parell de valors. L'he triat perque el que vull fer es moltes cerques de valors dins de l'estructura de dades sense importar l'ordre que tenen els valors. Per aquesta tasca la millor estructura de dades es una taula de Hash, ja que el cost mig de les cerques es constant (O(1))i l'estructura que ho representa a Java es el HashMap.
  També podria haber utilitzat TrieMap, però en JDK8, el seu rendiment està per sota del HashMap i, a més, l'avantatge que té aquest sobre el HashMap no la necessito (accedir a tots els elements alhora). A més, com nosaltres fem accessos aleatoris, el HashMap té cost constant i el Trie cost logaritmic.
- PriorityQueue: Aquesta estructura de dades s'ha fet servir per muntar l'arbre en el proces de creacio dels codis huffman. L'he triat perque necessito tenir en tot moment els valors dels nodes ordenats, i la priority queue, que s'implementa amb un heap (en aquest cas es un min-heap), es el que millor va. Si fos una altra estructura, quan vulguessim inserir un nou node, hauriem de reordenar la estructura sencera (cost O(n*log(n))) o mirar d'inserir-la en un lloc correcte (cost O(n)), en vers al cost d'inserir un valor nou a un heap (O(log(n)).
- Arbre: En el cas d'aquesta estructura de dades, s'ha triat perque es la que es fa servir per a muntar els codis en el proces de codificació huffman.
- ArrayList: Aquesta estructura de dades l'he fet servir quan he de tenir una llista de elements pero no se quants elements seran, i per tant necessitava que pogues creixer.
- Array: L'he fet servir per tractar les matrius de dades de la imatge i en tots els casos en els que he hagut de guardar elements quan si que sabia el total d'elements que hi hauria.

Pseudocod

RunLengthEncoding

Encode

```
1 FUNCTION RLEEncode (matrix)
2       resultat = cjt buit
3       CONTADOR = 0
4       WHILE NOT end of matrix reached
5               newValue = nextZigZagValue();
6               IF newValue = 0
7                       contador = contador + 1
8               ELSE
9                       tuple = {contador,newValue}
10                      resultat.add(tuple)
11                      contador = 0
12              END IF
13      ENDWHILE
14      resultat.add({0,0})
14 RETURN resultat
```

El que fa aquesta funcio es per una matriu de 8x8, la recorre fent zig zag (tal i com es mostra a la foto de l'explicacio del run length encoding, i si el valor es zero, incrementa el comptador, sino, guarda en un conjunt, un parell de valors que es el numero de zeros que es portava i el valor diferent de zero. Al final fiquem un parell amb dos zeros per a marcar el final de la matriu.

Decode

```
1 FUNCTION RLEDecode(data)
2       resultat = matrix
3       FOR p IN data
4               IF p == {0,0}
5                       resultat.acabaSubMatriu()
6               ELSE
7                       resultat.writeInZigZag(p.numZeros)
8                       resultat.write(p.numDifZero)
9       ENDFOR
10 RETURN resultat
```

Aquesta funcio el que fa es per a cada parell, de que li entra, si es el parell {0,0}, escriu amb zeros tot el que queda de la submatriu. Si no, escriu tants zeros en zig zag com la primera

component del parell i despres escriu el numero diferent de zero a la seguent posicio del zig zag.

Huffman

Creació arbre

```
1 FUNCTION CrearArbre (data)
2       contats = cjt buit
3       FOR d IN data
4               IF contats.contains(d)
5                       contats[d] = contats[d].frequencia + 1
6               ELSE
7                       contats.add(d,0)
8               ENDIF
9       ENFOR
10      PrioQueue = contats
11      WHILE PrioQueue .size() is not equal to 1
12              node = new node()
13              node.left = PrioQueue .pop()
14              node.right = PrioQueue .pop()
15              node.frequencia = node.left.frequencia + node.right.frequencia
16              PrioQueue.push(node)
17      ENDWHILE
18 RETURN PrioQueue.pop()
```

Aquesta funcio el que fa es en primer lloc, comptar quants parells hi ha de cada parell en totes les dades i despres es construeix l'arbre de huffman. Per construir-lo, agafa els dos nodes amb menys frequencia i els uneix en un que te la suma de les dues frequencies i el torna a ficar al conjunt de nodes. Un cop nomes queda un node, l'arbre ja esta construit.

Creació codis

```
1 FUNCTION crearCodis (nodeArbre,codi)
2       IF nodeArbre.left == empty AND nodeArbre.right == empty
3               RETURN result = codi
4       ELSE
5               codisLeft = crearCodis(nodeArbre.left,codi+"0")
6               codisRight = crearCodis(nodeArbre.right,codi+"1")
7               RETURN UNION(codisLeft,CodisRight)
8       ENDIF
9 END
```

Aquesta funcio el que fa es crear els codis de huffman per a codificar de la seguent forma. Començant des de l'arrel, es mira si te fills. Si no en te, es retorna el codi construit fins a ell. Si en te, es construeix els codis dels fills, ficant hi un zero o un u a la dreta segons si es el

fill esquerra o dret, respectivament. Un cop es tenen els codis dels fills, s'ajunten els sets de codis i es retornen.

Codificar contingut

```
1 FUNCTION buildCodes (codifier, data)
2       resultat = cjt buit
3       FOR d IN data
4               resultat.append(codifier[data])
5 RETURN resultat
```

Per a codificar el contingut, el que es fa es per a cada parell de dades a data, es busca el seu codi al codifier i s'afegeix al resultat. Un cop s'han codificat tots els resultats, es retorna el resultat amb tot codificat.

Decodificar contingut

```
1 FUNCTION decode(decodifier,data)
2       resultat = cjt buit
3       codi = buit
4       FOR bit IN data
5               codi.append(bit)
6               IF decodifier.conte(codi)
7                       resultat.add(decodifier[codi])
8                       codi = buit
9               ENDIF
10      ENDFOR
11 RETURN resultat
```

Per a decodificar les dades, es va construint un codi amb cada bit que entra per les dades, i si existeix el codi al decodificador, s'afegeix el parell decodificat al resultat i es reinicia el codi construit. Es fa aixo fins que no queden bits i es retorna el contingut decodificat.

**No s'ha inclos el pseudocodi de les operacions de passar de RGB a YCbCr i el seu pas invers, ni tampoc les operacions per fer la DCT i la seva inversa perque son formules.*