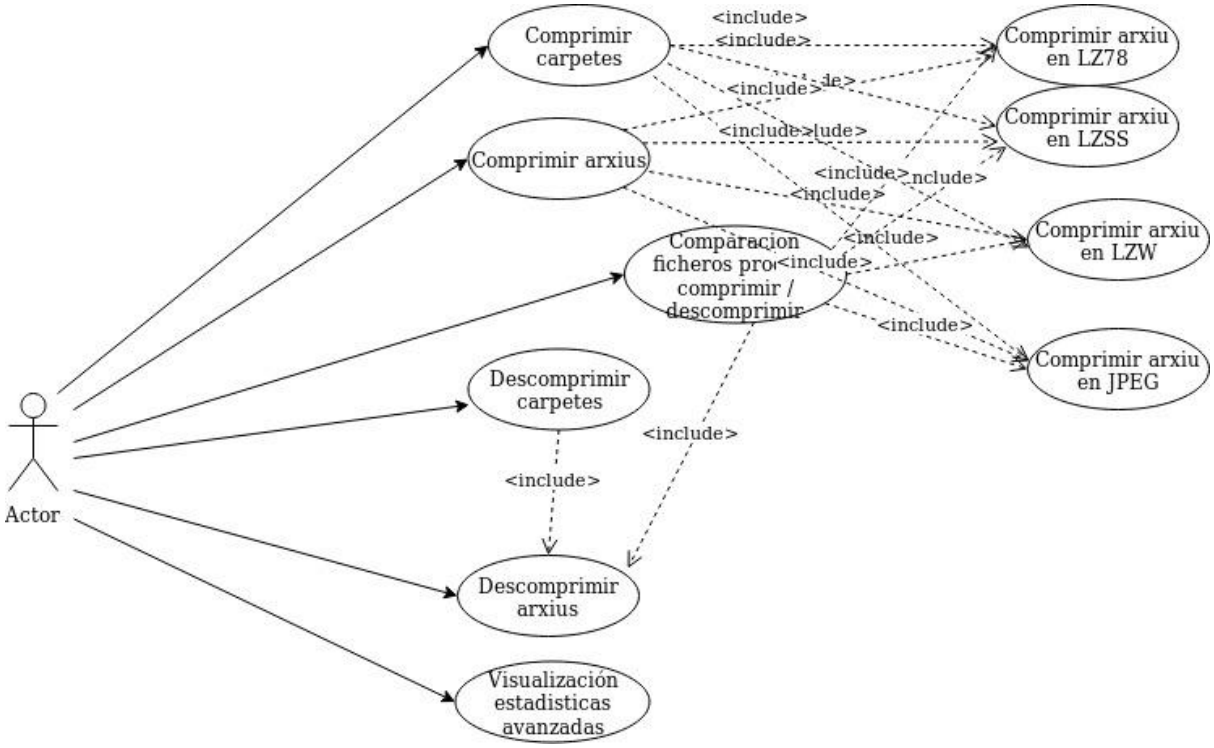
PROP

Entrega 1

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

Diagrama casos d’us



Lista de funcionalidades

Funcionalidades principales

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

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

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

Comprimir archivos en JPEG: Se comprime un archivo con el algoritmo JPEG. La extension 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 estadisticas basicas (tiempo de compresión y porcentaje de compresión).

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

Mostrar información estadistica de los procesos de compresion/descompresion: Una vez acabado el proceso de compresion/descompresion, el usuario puede ver una serie de estadisticas. Estas son: la velocidad de compresion/descompresion, tiempo que ha tardado, percentatge de compressio/descompressio i velocitat de compressio/descompressio.

Comprimir carpetas: el usuario pasa como input una carpeta y como output sale la misma carpeta con todo lo que contenia comprimido. El resultado es un unico archivo comprimido que dentro tiene todos los archivos de la carpeta.El usuario tiene que escoger con que algoritmo comprimir los archivos de texto y las imagenes. Puede elegir entre los siguientes algoritmos: LZ78, LZSS, LZW, JPEG. La extension 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 un fichero, el programa muestra por pantalla lo que tiene ese fichero por dentro y, una vez el usuario haya elegido el algoritmo, el programa muestra por pantalla el mismo fichero despues de haber pasado por su correspondiente proceso de compresion y descompresion.(+ Visualizar estadísticas + que algoritmos se pueden usar).

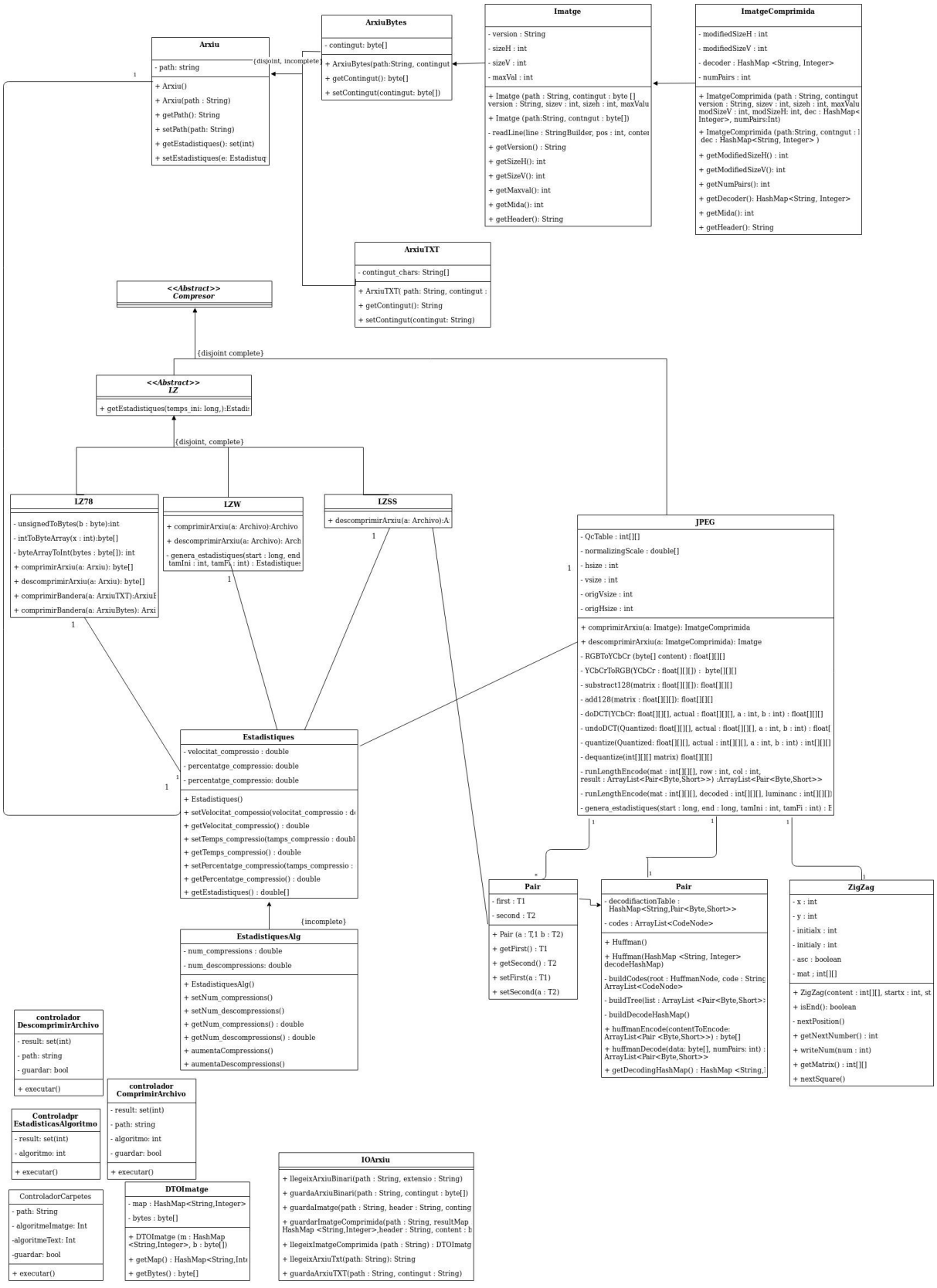
Guardar archivo a disco: Una vez se ha comprimido o descomprimido el archivo, el usuario puede guardar si quiere el resultado en disco.

Salir: Se cierra el programa.

Funcionalidades secundarias

Estadísticas generales de los algoritmos: El usuario elige uno de los cuatro algoritmos i como output le salen todas las estadísticas de ese algoritmo (porcentaje de compresion medio, velocidad media, porcentaje de perdida media). Si no se ha comprimido ningun archivo, sale todo ceros.

DIAGRAMA DE CLASES



En primer lloc, hem creat una classe Arxiu per a identificar cada fitxer diferent amb el seu path al sistema. D'aquesta classe hem fet herencia a dues subclases, segons si el contingut ens interessa tenir-lo en bytes (Arxiu Bytes) o en una string (arxiuTXT). Dins el sistema hem hagut de diferenciar les fotos, ja que tenen diversos parametres que no tenen els arxius de text. Aquest son la versio de ppm, alçada de la foto, amplada de la foto i el valor maxim que te el ppm. La clase Imatge hereda de ArxiuBytes ja que tindra el contingut en bytes.
A mes, hem hagut de diferenciar una imatge amb una imatge comprimida, ja que aquesta a mes de tenir els valors d'una imatge, ha de guardar el tamany modificat, ja que ha de ser multiple de 8, i tambe ha de guardar el traductor de la compressio per utilitzar-lo a l'hora de descomprimir.

Tambe hem creat la classe Estadistiques, ja que cada arxiu tindra les seves un cop hagin acabat de ser comprimits o descomprimits. Les estadistiques que mostrarem per aquesta primera entrega seran la velocitat de compressio, el percentatge que ha comprimit/descomprimit i el temps que ha trigat en fer-ho.

Un altre classe que hem fet ha estat la classe compresor, classe a la qual delegarem la responsabilitat de comprimir/descomprimir. Com a fills tindra un per cada algoritme (LZ78, LZSS, LZW i JPEG). Tots aquests tindran unes estadistiques generals segons les mitjanes de totes les compresions i descompresions realitzades per aquests. Aquesta funcionalitat la donarem de cara a la segona entrega.

Hem creat una classe encarregada d'accedir als arxius de disc, ja sigui llegir-los o escriure el contingut en un. Aquesta classe simulara la capa de persistencia.

I per ultim, tenim els controladors que serveixen per connectar la capa de presentacio amb la capa de domini.

ALGORITME 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.

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.

Descomprimir

```
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.

Algoritme 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.
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

```
1  Inicialitzem totes les estructures i omplim el buffer d'anticipació
2       PA = BC
3       WHILE TC > 0
4           IF PA.D < MIN_C
5                   E = E+C
6                   F = F << 1
7           ELSE
8                   F = (F << 1 OR 1)
9                   E += PA.P + PA.D
10          END IF
11          CB = CB + 1
12          WHILE i < PA.D AND PL< TC
13                  DIC = DIC + C
14                  SW = SW + C
15                  LA = LA + NEW(C)
16            END WHILE
17            WHILE i < PA.D
18                  DIC = DIC + C
19                  SW = SW + C
20                  TC = TC - 1
21          END WHILE
22          IF CB >= 8 OR TC == 0
23                  S = S + E
24                  CB = 0
25          END IF
26          IF TC > 0
27                  P = BC
28          END IF
```

```
29        IF PC BETWEEN (PA.P AND PA.P+PA.D)
30              PA.D= PC - PA.P
31        END IF
32   END WHILE
33   EST = CE
```

1. Inicialitzem totes les estructures de nou i SlidingWinidow ho omplim amb un caracter indicatiu (fara de controlador aquest caracter)
2. Realitzem la busqueda de coincidencia i el resultat ho guardem a un par
3. Mentre el tamany llegit sigui major a 0
4. Si el desplazament del par es menor que el minim per codificar
5. Guardem a la variable escriure el caracter a imprimir
6. Fem un shift left amb el byte flag
8. Desplazem el byte Flag a l'esquerra i fem una or amb un 1 per posar el bit a 1
9. Afegim a escriure, els bytes de posició+desplazament
11. Aumentem el contador de bits en 1 i posem la i a 0
12. Mentre que la i(variable per bucles) sigui menor que el desplazament i la posició per llegir menor que el final del contingut entra al bucle
13. Afegim el caracter codificat al diccionari
14. Afegim el caracter al buffer de cerca(SW)
15. Afegim al buffer d'anticipacio(LA) una nova lletra per a codificar del contingut
17. Mentre i sigui menor al desplazament (vol dir que hem arribat al final del fitxer)
18. Afegim el caracter codificat al diccionari
19. Afegim el caracter codificat al buffer de cerca
20. Disminuim el tamany a codificar en 1
22. Si el contador de bits es 8 o mes i el tamany a codificar no es 0 entra
23. Escribim a la sortida el byte flag seguit del contingut a escriure
24. Resetegem el contador de bits i el byte flag
26. Si el tamany a codificar es > 0 entra
27. Busquem la seguent coincidencia i ho guardem al par P
29. Si la posicio de guardar al buffer de cerca es troba entre la posicio de la coincidencia i la posicio de la coincidencia+desplazament
30. Cambiem el desplazament a la (posicio contingut - posicio inicial repeticio)

Descompressor

```
1  Inicialitzem totes les estructures
2  IF TAMC == 0 RETURN A
3  F = C[0]
3  WHILE i < TAMC
4         B = F >> PF
4           IF B == 0
5                 CH = C[i]
6                 S += CH
7                 SW += CH
```

```
8               i++
7       ELSE
8               P = leer(C[i], C[i+1])
9               FOR j < P.D
10                      S += SW[P.P]
11                      SW[c] = SW[P.P]
12              END FOR
13              i+=2
14      END IF
15      IF PF < 0 AND i<TAMC
16              F = C[i]
17      END IF
18  RETURN (path, S)
```

```
1  Inicialitzem totes les estructures noves per a netejar les posibles dades
2  Si el tamany del contingut es 0, retornem un arxiu buit
3  Agafem el flag que es el primer byte del arxiu
3  Mentre la i sigui menor al tamany del contingut
4  Agafem el bit de la posicio que necesitem i ho guardem a bandera
5  Si aquesta bandera es 0
6  Guardem a caracter el caracter del contingut[i]
7  Escribim a la sortida el caracter
8  Guardem al buffer de cerca el caracter
9  Aumentem la i en 1
11 Guardem el resultat de llegir els 2 bytes posicio i desplazament al par
12 si la j es més petita que el desplazament
13 Guardem a la sortida el contingut de la posicio del par
14 Guardem el contingut de la posicio del par al buffer de cerca
16 Aumentem en 2 la i
18 Si la posicio del flag es mes petit que 0 i la i es mes petita que el tamany del contingut
19 Guardem al flag el contingut de la posicio i
21  Retornem un arxiu amb el path cambiat i el contingut de sortida
```

Algoritme JPEG

Fet per Pau Murciano Julia

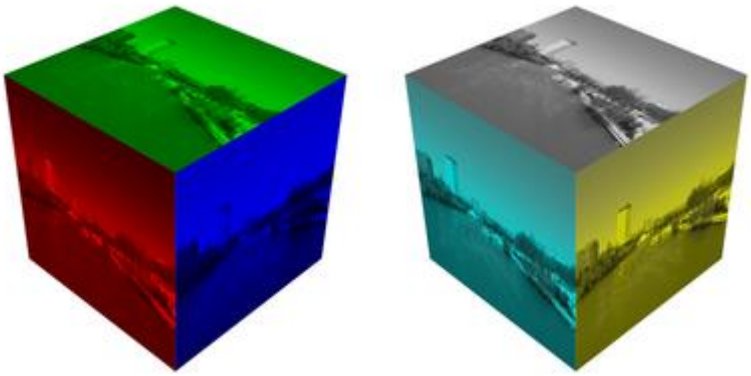
Introducció

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.

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

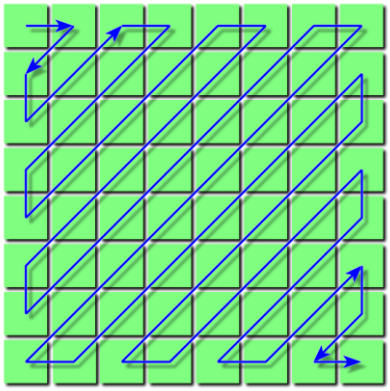
Matriu de quantització

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

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:

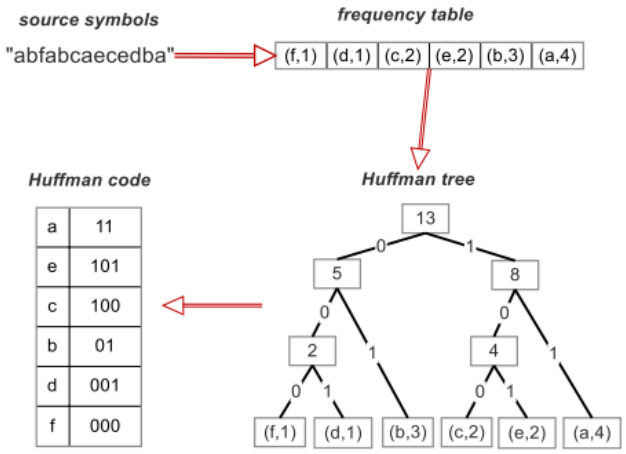


Recorregut en zig-zag d'una matriu

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.
- 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.*