Documentació UML

Diagrama de classes

La descripció de cada classe amb tots els seus atributs i mètodes es troba al DOCS/javadoc.
Els diagramas de clases de les tres capes diferents es troben a DOCS/

Relació classes implementades

Dominio	Adrián Álvarez Martínez	Jaume Bernaus Serra	Pol Monroig Company	David Santos Plana
LZ78	-	-	Sí	-
LZW	-	Sí	-	-
LZSS	-	-	-	Sí
JPEG	Sí	-	-	-
Utils	Sí	-	-	-
Stats	-	-	Sí	-
GlobalStats	-	-	-	Sí
Algorithm	-	-	-	Sí
Huffman	Sí	-	-	-
DomainCtrl	Sí	-	-	-
Tree	-	-	-	Sí
AutoAlgorithm	-	-	-	Sí
PhysicalFile	-	-	Sí	-
AlgorithmSet	-	-	Sí	-

Persistència	Adrián Álvarez Martínez	Jaume Bernaus Serra	Pol Monroig Company	David Santos Plana
DataCtrl	-	-	Sí	-

Presentacion	Adrián Álvarez Martínez	Jaume Bernaus Serra	Pol Monroig Company	David Santos Plana
ButtonsPanel	-	-	-	Sí
ConnectedButton	-	Sí	-	-
ContentDecorator	-	Sí	-	-
ContentInterface	Sí	-	-	-
Content	Sí	-	-	-
ContentPanel	-	-	Sí	-
CustomButton	-	-	-	Sí
ExitButton	Sí	-	-	-
FileChooser	-	-	Sí	-
FileSelector	-	-	Sí	-
ImagePanel	Sí	-	-	-
OptionSelector	-	Sí	-	-
VariableLabel	-	-	-	Sí
View	-	-	Sí	-
PresentationCtrl	Sí	-	-	-

Descripció algorismes implementats

LZ78:

La idea de tots els algoritmes Lempel-Ziv és que si un text no és uniformement aleatori llavors una paraula que ja hem vist és més probable que aparegui que una que no hem vist mai. Específicament LZ78 funciona construint un diccionari de paraules que s'anomenen frases, una frase és una concatenació de caràcters de mida més gran o igual a un. El diccionari que s'ha utilitzat en aquest versió del algoritme un SortedMap, es podia haver implementat amb un HashTable però donades una quantitat exhaustiva de proves empíriques s'ha concluit que l'eficiència era la mateixa per els dos. L'algoritme comença agafant la frase més petita del text que no haguem vist abans. Com que inicialment el diccionari està buit, la primera frase serà el primer caràcter. Seguidament agafem la següent frase que no haguem vist, de tal manera que si ja hem vist una frase el que fem és afegir el següent caràcter, fins que aquesta no estigui present al diccionari. Per cada frase **w** que no haguem vist la podem subdividir en dues sub-frases tal que **w=xy** i **|y|= 1**. Això significa que si **|w|= 1**, una frase format per un sol caràcter, tenim que **x** serà la paraula buida, que per precondició és una frase que sempre hem vist. Seguidament per cada paraula **w** que guardem al diccionari guardarem el índex de la posició de la paraula **x**. Quan acabem, la compressió representarà una sèrie d'índexs i caràcters. Per descomprimir simplement haurem de reconstruir el diccionari i les frases de manera recursiva. S'ha utilitzat un diccionari perquè hem de poder buscar i guardar frases úniques. EL algoritme implementat es subdivideix en varies parts. Primer es construeix el diccionari i al mateix temps es va guardan la codificació en forma de caràcters. Per últim per aprofitar la compressió al mateix es transformen els caràcters codificats en un binary string i per últim es transforma en un array de bytes per poder escriure a disc.

LZW:

Com la majoria dels mètodes de compressió l'algoritme LZW identifica les cadenes de caràcters repetides per tal de crear una taula d'equivalències i pode'ls-hi assignar codis més breus.

L'algoritme del mètode LZW en una sola passada pel text pot crear el diccionari i codificar el text. A més no és necessari passar aquest diccionari junt amb la codificació ja que el diccionari es reconstrueix durant el procés de descompressió.

Això passa degut a que el diccionari comença inicialitzat amb 256 entrades. Es van afegint codificacions per cada grup de caracteres , cada vegada més grans. Per l'output es treu el codi que se l'assigna al grup de caràcters. He decidit codificar els codis en 16 bits per tal de poder disposar de 65536 entrades en el diccionari.

Les estructures de dades per implementar l'algorisme han sigut un Map pel diccionari, ja que per cada entrada s'ha de guardar la cadena de caràcters, el seu codi i permet buscar fàcilment si el grup de caràcters ja té un codi assignat.

Un string per tal de guardar els codis en binari de l'output o el text descomprimit en el cas de la descompressió. Utilitza strings per a llegir els caràcters i operar amb ells creant les cadenes.

LZSS:

Aquest algoritme s'utilitza per a comprimir textos, la seva principal característica es la seva velocitat de compressió la qual és molt elevada.

El procés de compressió i descompressió es basa en un recorregut del fitxer de text de manera progressiva, de la qual cada nou caracter que es llegeix s'avalua respecte els 4096 últims caràcters

llegits, per tractar de trobar coincidències i poder emmagatzemar un conjunt de caràcters com a dos nombres, posició de coincidencia i longitud de coincidencia.
La manera la cual he implementat això consisteix en un buffer que emmagatzema els últims 4096 caràcters llegits del text, al afegir un nou caràcter, la estructura de dades que he creat quan inserto un nou caràcter en ella carrega les variables matchPosition i matchLength els màxims valors corresponents per al nou caràcter, si la mida de coincidencia que s'obté és menor que 3 es guarda el nou caràcter en el buffer de sortida, i el flag que el representa serà un 1, d'altra manera emmagatzemarà 2 bytes al output que indicaran la longitud de coincidencia i la posició on coincideix.

La complexitat temporal d'aquest algorisme consisteix en un recorregut lineal de l'endrada, la qual amb longitud n serà O(n), i per al cas pitjor seria que cada inserció que es realitza en el codi, una per cada valor del text recorri tot l'arbre sencer desde una única arrel, la qual es O(256*lg(256)) = O(1), després es fan algunes instruccions de cost constant, per tant el cost temporal del codi és O(n).

La complexitat espaial del codi sempre és la mateixa excepte contant la mida de la entrada, ja que les variables sempre ocupen lo mateix i els seus valors estan en constant canvi durant l'execució, per tant el cost espaial seria (tenint en compte els elements amb una mida considerable): 4KB del ringBuffer + 4*3KB de els arrays que utilitza el arbre + 0,256 KB dels caràcters que caben en un byte que es guarden en el rigthSon de l'arbre + (n/1000)*2 KB del text d'entrada, i el cas pitjor de la compressió que seria cap caràcter comprimit, per tant serien uns 16KB estàtics + O(n) de l'entrada.

Justificació de la estructura de dades: El fet de que al llegir un nou caràcter s'haguès de obtenir la màxima longitud dintre dels últims caràcters llegits m'ha fet decantar per un arbre, ja que sinó cada nova lletra hauria de recorrer tots els últims caràcters llegits, d'aquesta manera la estrategia que volia assolir era per cada posible combinació de un byte guardar tots els valors que han aparegut seguit d'aquest byte, per tant estaríem parlant d'un arbre per cada valor dintre d'un byte amb 256 fills per cada node, per tal de buscar totes les combinacions que han aparegut dels caràcters llegits anteriorment. Com un arbre dinamic fet amb punters podria arribar a ocupar un espai massa gros degut a l'arquitectura de 64 bits dels ordinadors actuals he preferit utilitzar un arbre estàtic format per arrays de shorts, ja que en aquests caben 4096 posicions de les que es basa el ringBuffer, amb 3 arrays, un pare i dos fills fent un arbre binari. M'ha semblat una de les millors opcions tant per el bon cost temporal que obtindrà al buscar coincidències, com el cost espaial que requereix crearlo, ja que estructures una mica més eficients poden ser molt mes grans.

JPEG:
Aquest algorisme consisteix en un algorisme de compressió d'imatges lossy, és a dir, que part de la informació de la imatge es perd en el procés, però és tan insignificant la pèrdua que els nostres ulls no la detecten. Principalment l'algoritme està basat en matrius, ja que la imatge en si és una matriu, i serà molt més fàcil treballar amb aquesta estructura de dades. Llavors, el que primer fa l'algoritme és llegir el ppm (és la versió P6), el que hem llegit són els colors dels pixels en RGB. En una imatge, hi existeix molta correlació entre aquests components de color, i nosaltres volem reduir la grandària de la imatge que és el principal objectiu, per tant haurem de fer una color space transform a YCbCr, que

representen lluminositat(Y), el color cromàtic blau (Cb) i el color cromàtic vermell (Cr). La lluminositat descriu la brillantor de la imatge, mentrestant els altres dos components tenen com informació la tonalitat de la imatge. Així que ara tindrem 3 components per treballar. Ara tractarem individualment les components per a després agrupar-les un altre cop. Seguidament, per poder continuar amb l'algorisme, haurem de dividir els color en blocs de 8x8. A continuació, farem el que es coneix com The Discrete Cosine Transform què és el cor del JPEG, on la DCT es tracta d'expressar una seqüència finita de números com a resultat de la suma de diferents senyals sinusoidals.

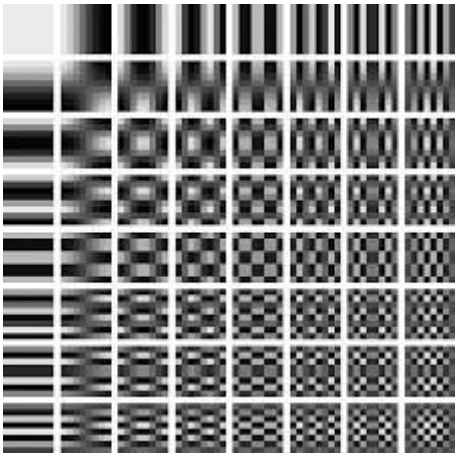

Figura 1: Seqüència de senyals de la DCT

Com podem observar en la Figura 1, tenim el bloc 8x8 amb les senyals pertinents de la DCT, i com la interpretem? Doncs si observem el punt 0.0, el que és totalment blanc, hi existeix un coeficient molt gran, vol dir que aquesta senyal apareix molt per tal de aconseguir aquesta senyal sinusoidal, si ens fixem, ens donarem compte que contra més complexa és un senyal el coeficient serà molt més petit. Una cosa molt important la DCT treballa en un rang de 1 a -1, ja que així aquesta forma la funció del sinus, llavors per tal de poder aplicar correctament la DCT haurem de normalitzar els components i ho farem restant la 128 (perquè és la meitat del color màxim de la imatge), i així aconseguirem normalitzar els components.
Com podem observar els coeficients són nombres reals que seran guardats com enters, això significa que haurem d'arrodonir els coeficients, així que primerament dividirem els coeficients pels factors de quantització i després arrodonirem: round(Fw,u / Qw,u). Això ens permet poder ressaltar els senyals més presents per tal de formar aquest senyal sinusoidal. A més, aquesta part efectua la qualitat de la imatge JPEG.
Seguidament, per poder codificar correctament, haurem d'ordenar els coeficients degudament, ja que si anem cap a la dreta, incrementarà el senyal horitzontal, i si ens movem cap a baix el senyal vertical. Llavors ens haurem de moure en zig-zag per ordenar els coeficients.

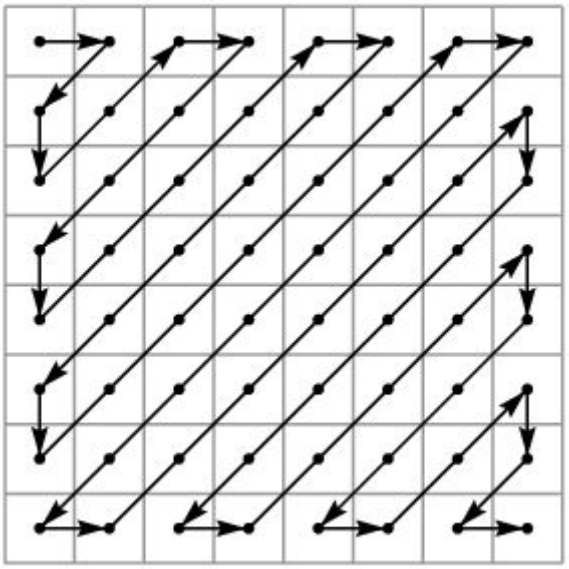

Figura 2: Recorregut en zig-zag

Així, ara, ja tenim preparat per poder codificar la imatge amb Huffman (explicat més endevant), on compte el nombre de freqüències dels coeficients, és a dir, quantes vegades apareixen, per tal d'ordenar-les en un arbre binari per aconseguir un codi binari, on contra la freqüència sigui més alta, més curt serà el codi i viceversa. Finalment, quan aconseguim els 3 components codificats ho escriurem en l'arxiu JPEG, i ja podrem donar per finalitzada la fase de compressió.

Per altra part per fer la descompressió, haurem de fer tot a l'inversa, és a dir haurem de desfer Huffman seguint l'arbre creat, després haurem de tornar a recórrer en zig-zag el bloc de 8x8, i desquantitzar amb el mateixos factors (multiplicar en comptes de dividir), i haurem d'aplicar la Inverse Discrete Cousin Transform on bàsicament desfarem el DCT abans fet, i normalitzarem a 0..255, sumant-li 128 als coeficients. Una vegada fet això faltaria l'últim pas, passar de YCbCr a RGB, i escriure en el fitxer .ppm.

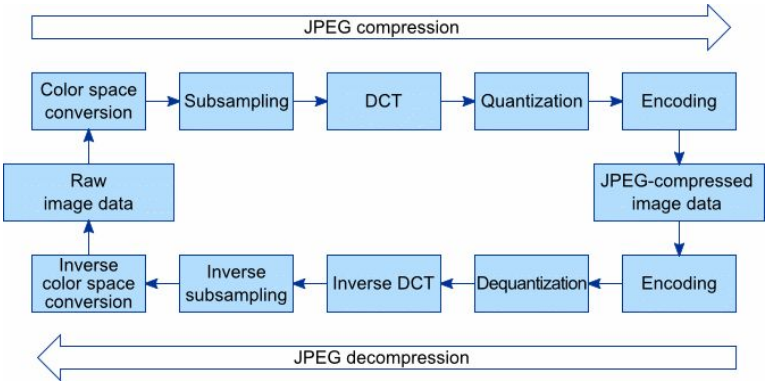

Figura 3: Esquema de la compressió i descompressió del JPEG

HUFFMAN:

La principal idea de Huffman és la descompressió i compressió de fitxers d'imatge, en aquest cas l'hem utilitzat per l'algorisme de JPEG.
Principalment, l'algorisme de Huffman és compon de diferents parts, la compressió i la descompressió. La primera part, es basa en una llista de freqüències, on contem cada vegada les freqüències dels números, és a dir, quantes vegades apareix en una llista de integers. Després, amb aquesta llista feta, continuarem creant l'arbre de Huffman, que es binari, i posarem les freqüències més altes a dalt de tot de l'arbre, sense ocupar mai l'arrel. Llavors d'aquesta manera, les freqüències més baixes quedaran a les fulles. Una vegada tenim l'arbre, podem passar a la següent i última fase, on crearem la codificació que recorrerem l'arbre de dalt a baix, així podrem comprimir molt eficientment, ja que si baixem per l'arrel cap a la dreta tindrem un 1 en codificació i si anem cap a l'esquerra un 0, per tant les freqüències més altes tindran un codi més petit, per tant les repeticions son molt més petites.
Ara parlem de la segona part, la descompressió, on primerament haurem d'aconseguir les freqüències de la imatge compressa, quan tinguem aquestes frequencies, haurem de crear l'arbre de Huffman (abans descrit), i una vegada aconseguit això podrem descodificar l'imatge. Agafarem el primer número de la imatge, si es un 0 anem cap a l'esquerra en l'arbre, i si es un 1 cap a la dreta. Fem això fins que arribem a una fulla, i així obtindrem el número descodificat (ja que la fulla de l'arbre tindrà el número).