PROP
Compressor d'arxius

Estructura de Dades i Algorismes

Identificador de Grup: grup-6.4

Joan Bosch Pons
Joan Salvador Magrans
Clara Sánchez Flaquer
Daniela Zhao

ÍNDEX

LZ78

L’algorisme LZ78, és un algorisme de compressió de fitxers plans de caràcters on no hi ha pèrdua de contingut.

L’algorisme LZ78 utilitza dues estratègies diferents, un per a realitzar la compressió i un altre per realitzar la descompressió.

El mètode utilitzat per fer la compressió utilitza un diccionari (HashMap, ja que la implementació d’aquest tipus de diccionari el cost d’afegir elements, eliminar elements o comprovar l’existència d’un element és constant) on la clau del diccionari és un ArrayList<Byte> i el valor obtingut és un Integer. S’inicia la compressió inicialitzant algunes variables auxiliars que ens ajuden a realitzar la compressió; el contingut de sortida (ArrayList<Byte>, on es construirà el contingut de sortida), un índex que ens indicarà quants elements s’han introduït al diccionari, la variable w que ens ajudarà a saber les coincidències trobades entre el diccionari i el text d’entrada i finalment inicialitzem el diccionari amb el caràcter buit.

Llavors, per cada caràcter del text d’entrada es mira si la concatenació d’aquest caràcter amb la variable auxiliar w es troba al diccionari o no. En cas de trobar-se al diccionari, la variable w passaria a valer el valor posterior de w amb la nova lletra. En el cas de no trobar-se la variable auxiliar amb el caràcter al diccionari, s’afegeix una nova entrada al diccionari amb clau aquesta variable i contingut l'índex del diccionari. A més també, s'afegeix a la llista de sortida, el bytes associats a la posició de la llista de Bytes que si es trobava al diccionari abans d’afegir-li el nou caràcter i també afegirem el caràcter que ha causat la nova entrada al diccionari.

A més d'això, el meu algorisme també conté un control de nombre màxim d’entrades al diccionari. Això és degut a la forma que s’ha implementat la codificació del nombre que referència a l’entrada del diccionari a l’output de la compressió, ja que,

de la forma implementada ens limita a 65.500 entrades (en comparació a la primera entrega que era de 55.000). En el cas de compressió de textos grans, aquesta barrera és superada, i és a aquests casos on s'aplica una metodologia diferent.

El mètode a seguir és semblant a l'anteriorment explicat, però ara, un cop trobem un sub-llista de bytes que no pertany al diccionari, el que fem és retrocedir dos bytes sobre la llista, obtenir la posició del diccionari que ocupa i s'afegeix a l'output, aquesta posició (obtenint els 2 primers bytes que codifiquen la posició de l'enter associat a la posició) i el segon caràcter que hem passat inicialment. Per tal de mantenir el text complet, en aquest cas, cal tornar a gestionar l'últim byte del contingut d'entrada, és per això que es torna una iteració endarrera al bucle del recorregut del contingut d'entrada.

Finalment, també introduirem el contingut que ens hagi quedat dins la variable auxiliar a l'ArrayList<Byte> de sortida per no perdre el contingut final de text d'entrada.

En acabar aquest procediment, el que obtindrem serà una llista de Bytes que contindrà els bytes corresponents a la compressió del contingut inicialment introduït.

Pseudo-codi comprimir:

```
MAX_LEN = maximum_dicc_lenght
w = Nothing
output = Nothing
dictionary = null_value

for each b in input_text do
      key = w + b
      if dictionary.contains(key) then
            w = key
      else
            if dictionary.size < MAX_LEN then
                  output += dictionary.value(w) + b
```

```
                dictionary.put(key)
            else if w.lenght < 1 then
                o = (w.size-1, w.size)
                u = w(0, w.size-1)
                output += dictionary.value(u) + o
                recompute b.
            else
                output += dictionary.value(null) + b
            end if
            w = null
      end if
next

for each b in w do
      output += dictionary.value(null) + b
next
```

En el cas de la descompressió, el procediment és l'invers al realitzat durant el procés de compressió. També s'utilitza un diccionari (HashMap) però en aquest cas tindrem com a clau un enter i com a contingut un ArrayList<Byte>.

En aquest cas, abans d'iniciar la descompressió serà necessari el contingut comprimit prèviament per l'algorisme, inicialitzar la variable de sortida, el diccionari, i un índex que com abans ens ajudarà a controlar el número d'entrades sobre el diccionari.

S'iniciarà el bucle sobre el contingut d'entrada, i, en aquest cas, anirem llegint seqüències de 3 bytes consecutivament. Els dos byte, anàlogament al procés de compressió, codifiquen la clau del diccionari que ens servirà per obtenir la part inicial de la llista de bytes que hem d'introduir al contingut de sortida. En segon byte s'haurà d'afegir directament al contingut de sortida.

A més, per cada iteració de bucle, haurem d'afegir al diccionari la llista de bytes que acabem d'afegir al contingut de sortida, amb l'ajuda de la variable auxiliar que controla les entrades del diccionari.

En acabar aquest procés, el resultat final torna a ser el mateix contingut que s'havia introduït inicialment.

Pseudo-codi descompressió

```
dictionary = null_value
output = null

for each b1, b2 and b2 in input_text do
      key = integer_value_of(b1 & b2)
      dictionary.put(dictionary.value(key) + b2)
      output += dictionary.value(key) + b2
next
```

LZSS (Lempel-Ziv-Storer-Szymanski)

És un algorisme de compressió sense pèrdues (*lossless*), és a dir, que no omet cap informació del fitxer en comprimir-ho.
Aquest algorisme és una versió millorada de l'algorisme LZ77.

LZSS utilitza una tècnica de codificació amb diccionaris que intenta reduir la quantitat mitjana de bits necessaris per representar un caràcter/símbol. La compressió consisteix en substituir un conjunt de caràcters amb una referència d'una posició del diccionari pel mateix conjunt de caràcters. La descompressió consisteix en reconstruir el text a partir del text comprimit.

LZSS intenta utilitzar la informació que ja s'ha vist en el document per comprimir el contingut d'un fitxer. Per aconseguir això, s'utilitzen les finestres corredisses. S'utilitzen dos tipus, el *search buffer* i el *lookAhead buffer*.
En el *search buffer* es guarden els bytes que es troben abans de la posició actual al fitxer, és a dir, es guarden els caràcters ja tractats. Mentre que els caràcters no comprimits es guarden al *lookAhead buffer*.
Aquestes finestres tenen una mida fixa i es poden implementar amb diferents tamanys, però és lògic dir que com més grans siguin les finestres corredisses, més comprimirà l'algorisme. El *search buffer* normalment té una mida de 4096 (12 bits). El *lookAhead buffer* té el tamany de la màxima longitud de coincidències.

La sortida de la compressió té la següent estructura:

màscara1 8codificacions màscara2 8codificacions …

Utilitzo una màscara d' 1 byte (8 bits) seguida de 8 codificacions.

La màscara codifica el bit FLAG que utilitza l'algorisme per saber si el següent caràcter que llegeix està codificat o no. Si és 0 no està codificat i 1 si està codificat.

Un caràcter codificat consta de :

- posició(*offset*): quantitat de caràcters que ha de retrocedir per trobar el caràcter que substitueix.

Utilitzo 12 bits per l'*offset*, és a dir que té un tamany de (2^12) -1 = 4095, que és el típic utilitzat.

- longitud(*matchlength*): quantitat de caràcters que coincideixen a partir de la posició.

La longitud es codifica amb els 4 bits restants, això ens permet una coincidència de (2^4)-1 = 15 caràcters repetits. És un valor acceptable ja que en un text normal és inusual que hi hagi coincidències més llargues.
La meva implementació té en compte que :

- La key del HashMap es un caràcter i el value es un HashSet que guarda totes les posicions anteriors on s'ha trobat aquest caràcter/símbol.

He decidit utilitzar un HashMap per optimitzar la búsqueda del caràcter, tenim , ja que té cost constant O(1) per les funcions put(), remove() i contains().
He decidit utilitzar un HashSet per evitar tenir posicions repetides i perquè el cost de les funcions add(), remove() i contains() és constant O(1). Vaig dubtar en si utilitzar una LinkedList, ja que les funcions add() i remove() són constants.
Vaig fer una prova de comprimir amb LinkedLists però trigava una mica més que amb HashSet, així que ho he deixat amb HashSet.

- L'algorisme codificarà només quan la longitud de caràcters que coincideixen és superior a 3.

Segons la codificació explicada abans, en els dos casos (si es codifica o no) s'afegeixen 8 bits (1 byte) + 8 bits (1 byte) cada 8 bytes, per tant al principi vaig limitar la longitud mínima a 2, ja que seria el mateix posar 2 literals que afegir 2 codificacions i és més fàcil afegir els literals directament. Però vaig provar de posar la longitud mínima a 3 i per alguna raó que no acabo de comprendre, comprimeix més que posant la longitud mínima a 2.

- No implemento les finestres corredisses.

Amb la implementació del HashMap per optimitzar la búsqueda de coincidències vaig decidir que no feia falta crear unes finestres corredisses, ja que podria accedir directament al byte d'entrada a les posicions amb coincidèndies. Per controlar el tamany de la finestra corredissa, calculo l'offset de cada posició trobada del HashSet i si és més gran que la finestra corredissa l'elimino del HashSet perquè segur que ja no el tornaré a mirar més.

A continuació, es troben els pseudocodis tenint en compte alguns valors amb els quals he implementat:

Compressió:

```
while ( !EOF ) {
	for (i = 0; i<15; i++) {
		inicialitzar mascara;
		caracter = input[pos_actual];
		if( es troba al diccionari ) {
			for(totes les posicions trobades) {
				ml = buscar la coincidència més llarga;
				if( ml > mínim_match_length) {
					mascara[i] = 1;
					offset = pos_actual - posiciótrobada;
					matchlength = ml;
				}
			}
		}
		else {
			afegir caracter i posició al diccionari;
		}
		afegir posicions de caracters coincidents al map;
		afegir caracter o codificació a la sortida;
	}
}
```

Descompressió:

```
while ( !EOF ) {
	mascara = input[pos_actual];
	for(i=0; i<15; i++) {
	BitFlag = mascara[i];
		if(BitFlag == 1) {
			offset = bits offset de input[pos_actual];
			mlen = bits longitud de input[pos_actual];
			for(j = sortida.length-offset; j<mlen; j++) {
				concatenar a la sortida sortida[j];
			}
		}
```

```
            else {
                  concatenar a la sortida sortida[j];
            }
            pos_actual++;
            mascara >>> 1;
      }
}
```

LZW (Lempel-Ziv-Welch)

L' algorisme LZW és un algorisme de compressió sense pèrdues (*loseless*), és a dir, no perd informació al descomprimir un fitxer comprimit per aquest algorisme. És una versió millorada de l'algorisme LZ78.

Funcionament:

Aquest algorisme, per codificar, utilitza un diccionari que va omplint, de manera automàtica, a mesura que va llegint caràcters. És a dir, en comptes de fer dues passades al text que es vol comprimir, amb una ja en té suficient.
Aquest diccionari no es guarda al final la compressió, sinó que a l'hora de descomprimir, el diccionari es re-construeix utilitzant la mateixa lògica que ha utilitzat quan comprimia.
Al final de la descompressió, el diccionari tindrà sempre els mateixos caràcters que al final de la compressió. Seran idèntics.

Com he implementat jo l'algorisme:

L' algorisme LZW està implementat amb un diccionari HashMap.
Vaig decidir implementar-ho amb aquest tipus de diccionari ja que no necessitava cap tipus d'ordre a l'introduir els nous caràcters al diccionari per comprimir i descomprimir. Una altra raó per la qual vaig escollir aquesta estructura de dades és perquè el seu cost és constant quan utilitzo la funció put(), que a diferència del TreeMap té un cost logarítmic, log(n).
Aquest HashMap està inicialitzat amb tots els caràcters representables de la taula ASCII de 8 bits. Hi ha 32 caràcters que no representables que aquests no estan afegits al diccionari per estalviar iteracions amb caràcters que segur que no es trobaran a un text per comprimir.

Inicialitzar el diccionari amb aquests caràcters comporta que l'algorisme no pugui comprimir textos amb caràcters que es troben fora de la taula d'ASCII de 8 bits, és a dir, del 0 al 255 primers caràcters, ja que no trobarà els caràcters i donarà error.

La key i el value del HashMap és diferent depenent de si comprimim o descomprimim un fitxer. Quan comprimim la key és un ArrayList<Byte> i el value un integer, en canvi quan descomprimim la key és un integer i el value és un ArrayList<Byte>. Aquest canvi el faig perquè quan comprimeixo, el contingut de sortida serà un conjunt de integers traduïts a bytes i per tant quan descomprimeixi hauré de buscar a través dels bytes traduïts a integers i no de bytes com quan comprimeixo.

El fet de que el contingut de sortida d'un fitxer comprimit sigui així és perquè facilita la lectura a l'hora de descomprimir i també perquè comprimir en bytes fa una major compressió que comprimir en caràcters. Tot i així, la compressió en bytes, quan s'utilitzen fitxers petits, aquest algorisme afegeix 2 bytes per cada integer del diccionari, això provoca que en fitxers molt petits, la mida del fitxer comprimit sigui major que la del fitxer original. Per tant, aquest algorisme és molt eficient per fitxers molt extensos però per fitxers petits no.

Pseudo-codi comprimir:

Inicialitzar el diccionari HashMap<ArrayList<Byte>, Integer> amb cadenes de longitud 1 que seràn els 256 primers caràcters de la taula ASCII de 8 bits.

```
w = null;
while ( !EOF ) {
	k = read input;
	if ( wk is in HashMap ) {
		w = wk;
	}
	else {
		output += (char) get value from key w (HashMap);
```

```
            add wk in the HashMap;
            w = k;
      }
}
print output;
```

El output serà un conjunt de bytes que representen els integers de les diferents posicions del diccionari.

Pseudo-codi descomprimir:

Inicialitzar el diccionari HashMap<Integer, ArrayList<Byte>> amb cadenes de longitud 1 que seràn els 256 primers caràcters de la taula ASCII de 8 bits.

```
read (int) input_vell;
caracter = get value from key input_vell (HashMap);
output = caracter;
while ( !EOF ) {
      read (int) input_nou;
      if ( input_nou not in the HashMap ) {
            cadena =  get value from key input_vell (HashMap);
            cadena += caracter;
      }
      else {
            cadena =  get value from key input_nou (HashMap);
      }
      output += cadena;
      caracter = first caracter de cadena;
      add HashMap (get value from key input_vell (HashMap)+caracter);
      input_vell = input_nou;
}
print output;
```

El output serà un conjunt de bytes que contindrà el mateix contingut que el fitxer d'entrada abans de comprimir el contingut. Aquest després es tradueix a strings i el contingut de sortida és el mateix que el contingut del fitxer original.

JPEG

L'algorisme JPEG (Joint Photographic Experts Group) és un algorisme de compressió d'imatges amb pèrdua. Si es realitza amb la configuració adequada aquesta pèrdua és quasi imperceptible, però sinó es veurà de forma clara una reducció de la qualitat.

Funcionament:

Per comprimir una imatge cal començar amb una representació en RGB de cada píxel. El primer pas és transformar els píxels a l'espai de color YCbCr, on Y representa la brillantor, Cb la crominància en component blau i Cr la crominància en component vermell. Aquest canvi és útil per al següent pas.

En segon lloc es fa un mostreig de les components Cb i Cr, de forma que aprofitem la menor sensibilitat de l'ull humà a canvis en aquestes components per eliminar informació que ens permet obtenir una major compressió. La component Y no la modifiquem ja que l'ull és bastant sensible als canvis en aquest cas. Les opcions més habituals són 4:4:4, on no s'aplica cap tipus de mostreig, 4:2:2, on s'agafa un de cada dos píxels en horitzontal i 4:2:0, on s'agafa un únic píxel per cada bloc de 2x2 píxels.

Abans de continuar cal preparar les matrius de forma que es puguin dividir en blocs de 8x8 píxels. Cal afegir píxels redundants als blocs dels extrems que no queden complets.

Després es fa la transformació discreta del cosinus, que consisteix en aplicar una operació matemàtica sobre blocs de 8x8 píxels per tal de concentrar la informació a les posicions de dalt a l'esquerra del bloc.

Un cop fet això es fa la quantització, que consisteix en dividir els valors de cada 8x8 per una matriu determinada. En funció del nivell de pèrdua que volguem haurem de triar una matriu o una altra. Aquesta fase aprofita que a l'ull humà li costa apreciar diferències en les altes freqüències de brillantor, d'aquesta manera es pot aplicar la divisió sense que la imatge pateixi pèrdues bastant visibles.

Per acabar, es passen els valors de les matrius a un format lineal mitjançant el mètode de zig-zag. A la vegada s'intenta reduir al màxim el nombre de zeros consecutius per millorar la compressió.

L'últim pas és aplicar l'algorisme de Huffman, que consisteix en definir un codi per cada valor d'entrada, que és més curt si el valor és més freqüent. Un cop tenim els codis assignats, passem els diferents valors d'entada als seus codis corresponents.

Per descomprimir només caldrà seguir aquests passos a la inversa. Cal tenir en compte que al desfer la quantització no serà possible recuperar els valors amb exactitud. Aquest pas és el principal responsable de la pèrdua de qualitat de la imatge.

Implementació:

Primer de tot es llegeixen els valors de l'string d'entrada que conté els caràcters d'un fitxer .ppm amb codificació P6 (binària) i es desen en matrius.

Per la transformació de l'espai de color he utilitzat una fórmula que a partir dels valors de R, G i B d'un píxel calcula els valors de Y, Cb i Cr.

Per fer el mostreig he afegit una opció per triar entre 4:4:4 i 4:2:0. En el primer cas, no cal fer res i en el segon, s'agafen els valors de Cb i Cr necessaris i es crea la matriu de cada component només amb aquests.

A l'afegir píxels redundants he fet que es copiï l'últim píxel existent per omplir els espais buits, ja que si ho hagués fet amb un color preestablert podrien aparèixer distorsions en el color dels píxels més propers als extrems.

A la transformació discreta del cosinus he implementat un codi que realitzi aquesta operació.

La matriu de quantització es pot triar entre una que conserva bastant bé la qualitat de la imatge i una altra que permet una major compressió, però que provoca que la pèrdua de qualitat sigui visible.

El zig-zag simplement recorre les matrius Y, Cb i Cr i aplica el zig-zag sobre cada bloc 8x8, de manera que podem guardar les matrius en un array. Per tal d'aprofitar la gran quantitat de zeros que es generen a la quantització subsitueixo els zeros seguits per un caràcter reservat i a continuació el nombre de zeros que haurà d'anar allà quan es llegeixi al descomprimir.

L'algorisme de Huffman primer crea un arbre binari per tal de generar els codis i després fa la compressió, substituint cada valor pel seu codi corresponent. El resultat s'escriu en un array de bytes juntament amb l'arbre per tal de poder fer després la descompressió.

La descompressió està implementada de la mateixa forma, però seguint els passos a la inversa. Primer es llegeix l'arbre binari de Huffman de l'array corresponent al fitxer comprimit i després es fa la descompressió convertint els valors codificats als originals, en segon lloc es desfà el zig-zag per obtenir les matrius, a continuació es desfà la quantització, després s'aplica l'operació inversa a la transformació discreta del cosinus, s'eliminen els píxels redundants i es copien els valors de Cb i Cr a tots els píxels de cada bloc 2x2 si al comprimir es va aplicar el mostreig, es transforma l'espai de color a RGB i es crea un string que representa una imatge en format .ppm amb codificació P6.

Estructures de dades utilitzades:

Per emmagatzemar els valors dels píxels de la imatge a tractar he utilitzat tres matrius d'enters, una per cada component del color. Quan estan en RGB tinc una matriu per R, una per G i una per B. De la mateixa manera, quan els píxels estan en YCbCr, cada una de les matrius correspon a una de les components.

Per generar els strings que contenen els valors obtinguts després de la compressió/descompressió, utilitzo *StringBuilder* en compres de *String*, ja que quan es fa una concatenació amb *String*, se'n crea un de nou copiant l'antic i afegint el text nou a continuació, cosa que pot ser molt ineficient quan estic treballant amb imatges grans. Amb *StringBuilder* puc fer servir el mètode append, que fa una concatenació de forma molt més ràpida ja que no fa una còpia de l'string original.

A l'algorisme de Huffman, per generar el codi corresponent a cada valor utilitzo arbres binaris, ja que d'aquesta manera, un cop construït l'arbre, puc obtenir els codis mirant si un node és fill dret o esquerre.

Per la compressió amb Huffman, faig servir un HashMap per emmagatzemar els codis. D'aquesta manera puc obtenir el codi corresponent a cada valor de forma ràpida i sense haver de recórrer constantment l'arbre.

Algorisme de Compressió de Carpetes

En aquest apartat volem donar constància de l'algorisme que hem utilitzat per a fer la compressió de les carpetes a fi de poder facilitar la comprensió del mateix.

El primer que cal comentar, és que aquest algorisme utilitza tots els quatre algorismes que s'han exposat anteriorment en aquest document, per tant, totes les explicacions, objeccions i observacions que es fan als apartats anteriors són també rellevants durant l'execució de l'algorisme de compressió de carpetes.

A mode de resum, podem dir que aquest algorisme funciona utilitzant una serie de *headers* els quals són els que indiquen al sistema quines són les accions que ha de dur a terme en tot moment.

Definim dos tipus de header:

- Header d'una carpeta: És el header que codifica tot el necessàri per tal de codificar una carpeta comprimida. Té el següent format:

  ```
  'c' + num_bytes_nom_carpeta + nom_carpeta + num_fitxers_carpeta (4 bytes) + contingut_carpeta_comprimida
  ```

- Header d'un fitxer: És a header que codifica tot el necessari per tal de codificar un fitxer comprimit dins d'una carpeta comprimida. Té el següent format:

  ```
  'f' + num_bytes_nom_fitxer (1 Byte) + nom_fitxer + byte_mida_nom_algorisme (1 Byte)+ nom_algorisme_fitxer + llargada_del_contingut_del_fitxer_comprimit (4 bytes) + contingut_fitxer
  ```

També és convenient recordar que les estrures de dades utilitzades per emmagatzemar el resultat de la compressió de les carpetes i arxius que conté són les mateixes que s'utilitzen en els algorismes, és a dir llistes i vectors de bytes.

A causa de la repetició d'algunes accions dins l'algorisme, s'han creat funcions auxiliars tal i com es pot observar al diagrama de classes, aquestes funcions auxiliars (tals com: `Concat(byte[], byte[]):byte[]`, o `getSubArrayByte(byte[], int, int):byte[]`) són funcions implementades al

ControladorArxius i que es poden utilitzarar al pseudo-codi per il·lustrar de forma eficient el comportament de l'algorisme.

Aleshores, podem definir la funció principal de compressió i descompressió d'una carpeta de la següent forma:

Comprimir carpeta:

```
Array<String> paths = getAllPaths(root_path)
Array<Byte> header;

header.add('c')
header.add(name_folder.lenght)
header.add(name_folder)
header.add(num_files_folder)

for each file in folder.files do
      Array<Byte> header_file;
      if(file.is_folder) result = compress_folder(file)
      else {
            header_file.add('f')
            header_file.add(file.name.lenght)
            header_file.add(file.name)

            result = compress_file_(file)

            header_file.add(result.algorism_name.lenght)
            header_file.add(result.algorism_name)
      }

      header_file.add(result.content.lenght)
      header_file.add(result.content)
      herader.add(header_file)
next

return header
```

Descomprimir carpeta:

```
int folder = input.getNumBytes(1)
int size_folder_name = input.getNumBytes(1)
String folder_name = input.getNumBytes(size_folder_name)
int num_files = input.getNumBytes(4)
byte[] content = input.getAllLastBytes()

for each n in num_files do
      if (content.getNumByte(1) == 'c'){

            decompress_folder(content)
      }
      else {
            int size_file_name = content.getNumByte(1)
            String file_name = content.getNumByte(size_file_name)
            int size_alg_name = content.getNumByte(1)
            String alg_name = content.getNumByte(size_alg_name)

            int content_size = content.getNumByte(4)

            Byte[] content_file = content.getSubArrayByte(content, n, n+content_size)

            decompress_file(content_file, alg_name)
      }
next
```