Programació declarativa. Aplicacions
Pràctica d'Scala

Luis Carlos Bautista - u1933558
Xavi Soto - u1918696

December 17, 2017

1 Estructura del codi font

1.1 CodeTree.scala

Conté la implementació de l'arbre de Huffman i totes les seves funcionalitats: crear, codificar, descodificar, etc. Hi ha dos versions: una per la primera part de la pràctica i l'altre per la segona.

1.2 ZipUnzip.scala

Conté les classes necessàries per comprimir -ZipperMaster i Slave- i descomprimir -UnzipperMaster i Slave- a més de declarar els case-class necessaris per l'intercanvi de missatges.

1.3 Serializer.scala

Conté els mètodes necessaris per a serialitzar els objectes que intervenen a la descompressió.

1.4 FileCompression.scala

Conté la implementació de la representació dels fitxers comprimits.

1.5 Utils.scala

Conté la implementació de les funcions auxiliars estàtiques que es fan servir per a comprimir, descomprimir, llegir i guardar fitxers.

1.6 BitVector.scala

Conté la implementació de la classe BitVector.

1.7 Exec.scala

Arxiu "main" de l'aplicació. Comprova que els paràmetres d'entrada siguin correctes i crida al Zipper/Unzipper segons el cas demanat per l'usuari.

2 Decisions preses: primera part

1. S'ha implementat la funció `mergeSort`, que com el seu nom indica, implementa l'algorisme de Merge Sort. Aquesta funció s'utilitza per implementar `makeOrderedLeafList`.
2. Les funcions: `decode` i `encode` són recursives utilitzant el mètode de la immersió.
3. La funció d'immersió de `decode` és especial ja que no recorre l'arbre i retorna el resultat, si no que un cop trobat el resultat torna a l'arrel per si mateixa i torna a començar. Aquest enfoc és realment dolent ja que és com si fos una recursió dins un altre recursió (i successivament) el que provocaria que el programa es quedes sense pila si el missatge a descodificar fos molt llarg (és a dir, recorre tantes vegades l'arbre com bytes s'haguessin de descodificar sense sortir de la primera recursió). Aquest disseny ha sigut corregit a la versió de la segona part de la pràctica.
4. Aquesta mateixa estratègia s'utilitza a la immersió del codificar.

3 Decisions preses: segona part

3.1 General

A les primeres versions funcionals de la compressió es van utilitzar les estructures de dades que es van donar a classe. Entre elles la més recurrent seria la `List`.

Si bé aquesta estructura és útil pel *pattern-matching*, no porta més avantatges i en canvi genera un seguit de problemes:

1. **És inmutable:** cada cop que vulguem modificar l'estructura haurem de generar innecessàriament una còpia.
2. **No és possible afegir naturalment un element al final de l'estructura (append):** Es pot fer creant una llista d'un element i concatenant-la a l'estructura on volem afegir el valor. Això però, no és molt net.

Aquests dos problemes s'arreglen si en comptes d'utilitzar una `List`, utilitzem un `ListBuffer`. Si bé perdem algunes característiques de la `List`, guanyem d'altres interessants:

1. És mutable: no crea còpies si no s'indica expressament.
2. Es poden afegir elements al final de l'estructura en temps constant.
3. Muta a `List` en temps constant.
4. Es perd la capacitat d'obtenir el cap en temps constant. Ara bé, aquesta característica és rarament utilitzada a la aplicació.

I per últim, els `MapReduce` que hi ha a la pràctica no són genèrics ja que impedia optimitzar el codi en parts crucials de l'execució.

3.2 CodeTree

La implementació de l'arbre de Huffman és genèrica. Això ens permet treballar amb arbres del tipus que vulguem: de caràcters, de nombres enters, de **bytes**...

La adaptació del codi per a que fos plenament funcional amb les classes de comprimir i descomprimir han distingit tres tipus de canvis:

3.2.1 modificació total de les funcions

S'ha fet ja que la seva implementació a la primera part era dolenta. La única funció refeta totalment és la immersió de la funció `decode`: ara només retorna el byte descodificat en comptes de tots els bytes, és a dir, la recursió acaba un cop arriba a qualsevol node fulla.

No s'ha canviat la implementació immersiva de la funció codificar. Si bé és el mateix enfoc que la del `decode` i que per tant s'allunya del que seria una funció recursiva ideal, la funció a la final acaba donant resultats satisfactoris.

3.2.2 modificacions lleus dins les funcions

S'ha fet ja que el codi fet estava ben implementat però el canvi d'estructures comportava una millora exponencial del rendiment.

Això s'ha fet ja que es volia mantenir el codi el més semblant al de la primera part. Per exemple, la funció `encodeIn` de la primera part és:

```
1  def encodeIn(root: CodeTree, currentNode: CodeTree, text: List[Char], encoded: List[Bit]):
       List[Bit] = {
2    if(text.length > 0){
3      currentNode match{
4        case Leaf(c,w) => encodeIn(root,root,text.tail,encoded)
5        case Fork(l,r,c,w) =>
6          if (chars(l).contains(text(0)))
7                encodeIn(root,l,text,encoded:::List(0))
8            else
9              encodeIn(root,r,text,encoded:::List(1))
10     }
11   }
12   else
13     encoded
14 }
```

Si el comparem amb el de la segona part:

```
1  def encodeIn[T](root: CodeTree[T], currentNode: CodeTree[T], toCode: List[T], encoded: BitVector):
       BitVector = {
2    if(toCode.length > 0){
3      currentNode match{
4        case Leaf(t,w) => encodeIn(root,root,toCode.tail,encoded)
5        case Fork(l,r,lt,w) =>
6          if (content(l).contains(toCode(0)))
7            encodeIn(root,l,toCode,encoded.add(false))
8            else
9            encodeIn(root,r,toCode,encoded.add(true))
10     }
11   }
12   else
13     encoded
14 }
```

Observem que la manera de fer créixer l'element que volem retornar és diferent. A la primera versió es van concatenant llistes -que de fet només es fa per afegir elements al final de l'estructura-. Això és molt ineficient ja que a cada crida es crea una còpia.

En canvi, a la versió de la segona part sempre s'utilitza el mateix objecte i per tant no es creen còpies a mesura que es baixa per la recursió.

3.2.3 creació de noves funcions

La introducció de noves estructures de dades o el fet de que les classes de compressió i descompressió demanaven instruccions específiques van obligar la creació de noves funcions:

- `times`: És exactament la mateixa funció que la de la primera part. Només canvia la seva signatura i es que comptes de rebre una `List`, rep una `Array`. Aquest canvi s'ha fet per pura optimització ja que des de on es crida no cal fer un càsting de `List` a `Array` (aquesta operació és molt cara).
- `createCodeTreep`: L'antiga creava l'arbre a partir de la llista d'elements (la freqüència d'ocurrències dels bytes es calculava dins aquesta funció). Aquesta nova funció no la calcula dins i per tant ha de rebre la llista de freqüències.
- `UltimateCodeTable`: És una nova representació de la `CodeTable`. En comptes de guardar els registres de codificació en tuples dins una llista, s'ha implementat en forma de mapa on el byte a codificar és la clau i el contingut és la seva representació comprimida. Amb aquest canvi aconseguim fer consultes en temps quasi constant (els mapes a Scala estan implementats amb taules Hash).
- `quickEncode`: Aquesta funció no transforma el `CodeTree` i per tant ha de rebre la nova taula de codificació. També, en comptes de rebre una llista del contingut a codificar, rep un `Array`. Amb aquest canvis ens estalviem de crear la taula cada cop que volem codificar i la utilització de funcions del tipus `.toList` (a la majoria dels casos són còpies i sempre amb un rendiment lineal).

3.3 Compressió

3.3.1 Lectura de fitxers

La càrrega de fitxers ha memòria és paral·lela: el `MasterZipper` indica als `Slave` quins fitxers han de llegir sense cap criteri en especial. Així mateix, l'encarregat de guardar la informació és el `ZipperMaster` i no els `Slave`.

3.4 Càlcul de les ocurrències dels bytes

El `MasterZipper` reparteix els bytes els quals els `Slave` han de calcular les ocurrències. Aquest repartiment és transparent al fitxer i per tant els `Slave` sempre tindran una càrrega de treball quasi igual als seus germans, per exemple: Si tenim 999 bytes en total (no importa si hem llegit 1 o 10 fitxers) i tenim 3 workers, llavors cadascun s'encarregarà de 333 bytes.

3.4.1 Compressió dels fitxers

Per aprofitar al màxim els workers, la compressió d'un o diversos fitxers no es basa en els propis fitxers, sinó en blocs de bytes (*shards*).

D'aquesta manera tots els fitxers es divideixen en blocs del mateix nombre de bytes adequadament identificats i per tant el `ZipperMaster` només haurà de repartir un nombre igual de blocs per a comprimir entre els `Slave` indiferentment si pertanyen a un fitxer o a un altre.

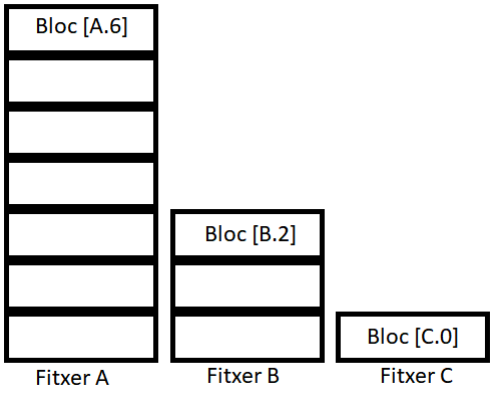

Figure 1: Exemple de la divisió de fitxers en blocs de bytes

3.4.2 Guardar dels comprimits

Els fitxers comprimits es guardarien de manera que un nom (ruta del fitxer) identifiqués tot el seu comprimit.

Guardar tot el fitxer comprimit en un `BitVector` implica que els blocs s'han de reduir. Aquesta tasca també s'ha paral·lelitzat de manera que cada `Slave` redueixi els blocs de `BitVector` de un mateix fitxer (sempre que hi hagi més fitxers que workers). A més a més, la càrrega de feina està repartida el més equitativament possible per evitar descompensacions entre workers.

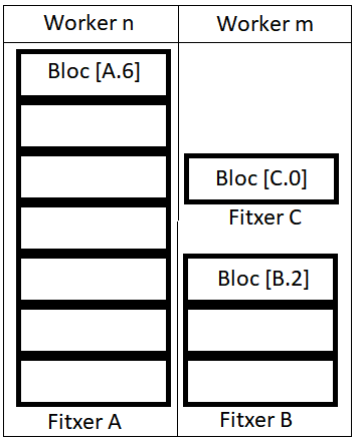

Figure 2: Exemple de la distribució de càrrega alhora de reduir els blocs comprimits

Per acabar, el `MasterZipper` només haurà de transformar els resultats dels workers a un `FileCompression` i enviar-ho, juntament amb l'arbre de Huffman, al main.

Per tant, el responsable d'escriure la compressió i serialitzar els Objectes necessaris per a la descompressió serà el main.

3.5 Descompressió

Aquesta classe és molt més simple que la de comprimir. Bàsicament es deu a que la manera en que hem guardat els fitxers comprimits impedeix un tractament avançat alhora de descomprimir.

Només s'ha pogut paral·lelitzar la descompressió a nivell de fitxer quan el ideal hauria haver estat a nivell de blocs de `BitVector`. La distribució de què fitxer ha de descomprimir quin `Slave` és equitativa seguint l'esquema de la figura 2

Ja que cada `Slave` descomprimeix un fitxer, aquest també s'encarrega de escriure'l a disc per evitar un viatge innecessari de missatges i dades entre actors.

4 Com s'han fet els MapReduce?

A la pràctica només s'han fet dos `MapReduce` complets i tots dos a la part de comprimir. Això és degut a que la forma d'emmagatzemar els comprimits impedeix aplicar `MapReduce`, ja que els `BitVector` corresponen a fitxers i no a blocs de fitxers. Aquesta manera de guardar-ho ens ha limitat degut a que és impossible determinar on es pot dividir un `BitVector` evitant la possible divisió (`MAP`) per a descomprimir i la seva posterior unió (`REDUCE`).

Així doncs, l'`UnzipperMaster` només determina quins workers i els fitxers que han de descomprimir i escriure.

A més, les comunicacions entre Master i Slave no sempre eren per realitzar un `MapReduce`, sinó que a vegades només es realitzava un `mapping` per operar un tipus d'informació ja que així s'agilitzava més el procés.

També cal esmentar que no s'ha implementat la versió genèrica del `MapReduce` ja que impedia l'optimització d'algunes parts del codi. Això ho veurem clarament quan expliquem el segon `MapReduce`. Tot seguit els `MapReduce` que s'han fet:

1. `MapReduce` per l'obtenció de les freqüències dels arxius:

   Quan el `ZipperMaster` ha rebut la ruta de tots els arxius a comprimir, el primer que fa és repartir aquestes rutes entre tots els workers per obtenir els bytes que els componen.

   `Master envia String -> Slave retorna Array[Byte]`

   Un cop els `Slaves` han retornat tots els bytes, el `ZipperMaster` agafa tots aquests bytes i els fragmenta en grups de bytes (*shards*) per a que els workers calculin les freqüències de cada un dels bytes rebuts

   `(MAP) Master envia Array[Byte] -> Slave retorna List[(Byte,Int)]`

   Un cop rebudes les llistes de tots els `Slaves`, es crea un diccionari (`map`) on cada element d'aquest diccionari tindrà com a clau el Byte corresponent, i com a valor una llista amb les freqüències calculades de tots els `Slaves` sobre aquell byte. Amb el diccionari fet, es tornen a distribuir a cada esclau cada una de les parelles (Byte,List[Int]) als esclaus per a que calculin el total final de les freqüències de cada byte rebut.

   `(REDUCE) Master envia List[(Byte,List[Int])] -> Slave retorna ListBuffer[(Byte,Int)]`

   Finalment, un cop hem realitzat aquest `MapReduce` ja podem crear l'arbre de Huffman amb les dades obtingudes.

2. `MapReduce` per la compressió dels arxius:

   Tenint l'arbre de Huffman creat i obtenint el `UltimateCodeTable`, es procedeix a fragmentar la informació dels fitxers en *shards* i, un cop fragmentats, s'obtenen grups de *shards* per distribuir-los als `Slaves` per a que puguin comprimir els bytes.

   `(MAP) Master envia UltimateCodeTable i ListBuffer[((String,Int), Array[Byte])] -> Slave retorna ListBuffer[((String,Int),BitVector)]`

   Un cop comprimida la informació de cada fitxer, creem un diccionari per agrupar els *shards* en funció del fitxer al que corresponen, de manera que obtenim `Map[String, TreeSet(Int,BitVector)]` on:

   - String : Nom del fitxer
   - TreeSet(Int,BitVector) : Estructura on Int representa el nº del shard del fitxer i BitVector és l'informació comprimida d'aquell trós de fitxer

En aquest procés de construcció del diccionari es pot comprendre per què no s'ha fet el `MapReduce` genèric i es que el contigut d'aquest no és una `List`, sinó un `TreeSet`. Això es degut a que aquesta estructura de dades ens permet mantenir l'ordre automàticament i ens estalvia de fer una ordenació quan els workers estiguin fent el (REDUCE). Passem de una complexitat `O(n log n)` a una `O(log n)`.

Un cop construït el diccionari, calculem com s'hauran de distribuir els fitxers de manera que la feina quedi el més balancejada possible i un cop fet això, enviem als `Slaves` els noms dels fitxers i els shards dels fitxers que els hi pertoca per la seva unió.

`(REDUCE) Master envia String i TreeSet(Int,BitVector) -> Slave retorna String i BitVector`

5 Com s'ha modificat la classe BitVector?

Per tal de fer la pràctica, el `BitVector` hauria de ser capaç de créixer dinàmicament. Com que la seva implementació té com a base una `Array`, l'única opció era la de crear un nou `Array` més gran i copiar en aquesta el contingut de l'actual.

Hi ha diverses estratègies i, de entre totes elles hem escollit la mateixa que ha escollit la classe Vector de `C++`: doblar l'espai si l'`Array` es queda petita. Això comporta un seguit de problemes:

- L'`Array` creix sense supervisió
- No és molt eficient si parlem d'espai ja que, en moltes ocasions, el final de l'`Array` estarà buit
- A més a més, si es serialitza (com és el nostre cas) i no es fa res, es guarda un espai sense utilitzar.

Aquest últim problema s'ha solucionat amb la funció `fit`, que serà explicada més endavant.

5.1 Mètodes crucials

5.1.1 `resize`

Donat la posició d'un bit, si aquesta posició és més petita que la mida real de l'Array llavors només s'actualitza la mida virtual.

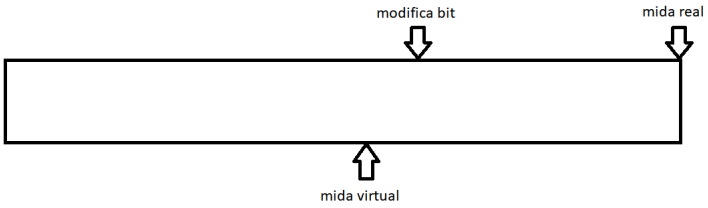

Figure 3: Abans de fer un resize simulat

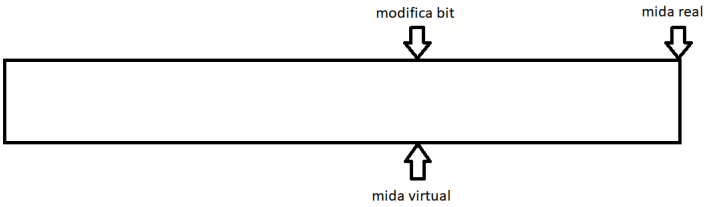

Figure 4: Després de fer un resize simulat

En canvi, si aquesta és més gran que l'`Array` llavors es crea un Array del doble de mida i es copia byte a byte de l'actual a la nova.

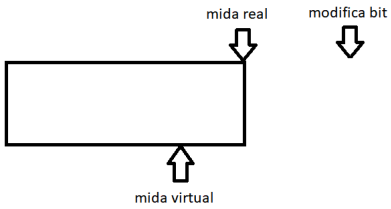

Figure 5: Abans de fer un resize real

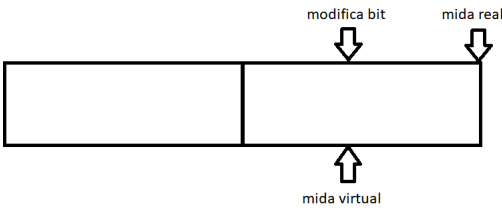

Figure 6: Després de fer un resize real

5.1.2 `set` i `clear`

S'han vist lleugerament modificats, abans de modificar el bit s'invoca la funció `resize`.

5.1.3 `get`

Ara llença una excepció cas que es vulgui accedir a un bit fora de rang.

5.1.4 `:::`

O concatenació, és de les funcions més importants si volem implementar un programa altament paral·lelitzable. Això és degut a que ens deixa concatenar dos `BitVector` (com el típic operador de `List`) i per tant, un fitxer pot se dividit per a comprimir per blocs i després ajuntar els comprimits en un sol `BitVector`: el fitxer comprimit.

5.1.5 `fit`

Permet ajustar la mida real del vector a la mida virtual. És a dir, l'`Array` del `BitVector` tindrà la mida mínima que permeti guardar tota la informació. Per exemple, si tenim el següent `BitVector`:

la funció ens retornaria el següent:

Aquest estalvi d'espai s'accentua a mida que el vector es fa més gran.

5.2 Altres mètodes

Els següents mètodes s'han creat i fet servir en versions anteriors del projecte però s'han deixat a la versió final per què son interessants.

5.2.1 `::`

O afegir al cap (en anglès *prepend*), permet afegir un bit a l'inici del `BitVector`. Era especialment útil pel *pattern-matching* però la seva ineficiència en un programa molt gran genera moltes copies i fa que el rendiment caigui ràpidament.

5.2.2 `tail`

Permet obtenir la cua del `BitVector`. No s'utilitza ja que genera una còpia cada vegada que s'invoca a més de tenir una eficiència baixa: linear en bits i no en bytes.

6 Desenvolupament de la pràctica

Primer de tot, vam realitzar la implementació del `CodeTree` en la seva forma bàsica donada inicialment, després del `BitVector`. Tota aquesta part no ens va portar molt problema al principi, fins que més tard ens vam donar compte que no estava gaire ben optimitzada...

Un cop vam llegir el segon apartat de la pràctica, vam veure que en comptes de guardar només caràcters com estava definit a l'inici de la pràctica, per a poder omplir l'arbre de Huffman amb qualsevol informació de qualsevol fitxer, no només havíem de guardar caràcters sinó que hauríem de guardar els bytes tal qual.

Per a això, en comptes de modificar els tipus `Char` dels valors que es guardaven, en comptes de intercanviar-los per `Byte`, vam decidir fer la classe `CodeTree` de tipus genèric per si ens era necessari en el futur.

Després vam dividir la feina en les següents parts:

- Creació de la classe que gestionaria l'estructura de directoris.
- Actors 'pare' que s'encarregarien de la gestió de la compressió i descompressió dels workers/slaves, i els propis slaves.
- Serialització i de-serialització de les classes i format de l'arxiu on s'emmagatzemaria la informació de la compressió.
- Diferents funcions necessàries per les diferents accions (classe `Utils`).
- Gestionar els paràmetres d'entrada i comprovar la seva correctesa.

6.1 Creació de la classe per estructura de directoris

Per a l'estructura de directoris, vam pensar crear una classe Contenidor, on un Contenidor podia ser un Arxiu o una Carpeta (no està totalment implementada):

- Una Carpeta es formava d'un `String` (nom) i una llista de Contenidors (els arxius o carpetes que contenia a dins).
- Un Arxiu només consta del nom que el defineix.

```
@SerialVersionUID(100L)
 class Contenidor extends Serializable

      case class Arxiu (
        name: String
      ) extends Contenidor

      case class Carpeta (
          name: String,
          contenidors: List[Contenidor]
      ) extends Contenidor

      def contenidors(elem: Contenidor): List[Contenidor] = elem match {
        case Arxiu(n) => List[Contenidor]()
        case Carpeta(n,c) => c
      }

      def isDirectory(elem: Contenidor): Boolean = elem match {
        case Arxiu(n) => false
        case Carpeta(n,c) => true
      }
```

Aquesta estructura al final no ens va caldre, ja que l'estructura dels directoris en comptes de definir-la mitjançant aquesta estructura, al guardar un `Map[String,BitVector]` on cada `String` era la ruta relativa de cada fitxer segons la ruta d'origen donat al fer la compressió.

6.2 Serialització i deserialització de la informació de la compressió

Primer de tot, vam fer que la classe que contenia l'arbre de Huffman (`CodeTree`) i la classe que contenia el map amb l'informació de cada arxiu (`FileCompression`) passessin a extendre el trait Serializable, juntament amb l'anotació `@SerialVersionUID(‘‘num’’L)`, on "num" és un número que servirà per identificar la classe a l'hora de deserialitzar.

Després, bàsicament tenim la classe `Serializer` que tindrà un mètode per serialitzar la informació de la compressió, on donats un *filepath*, un `FileCompression` i un `Huffman.CodeTree`, serialitzarà al fitxer especificat pel *filepath* l'arbre de Huffman i després la compressió del mapa.

Per deserialitzar el mateix, donat un filepath llegirà l'arbre de Huffman i la classe `FileCompression` i afegirem aquestes dues dades a una `Array[Any]` on un cop rebudes pel main farà un càsting al tipus corresponent.

6.3 Gestió paràmetres entrada

Per els paràmetres d'entrada s'han utilitzat una sèrie d'asserts de manera que es poguessin verificar fins a cert punt la correctesa d'aquests.

En el cas de la ruta destí però, s'ha deixat llibertat ja que és probable que s'insereixi una ruta on s'hagi d'emmagatzemar el arxiu de compressió que no existeixi, de la mateixa manera que la ruta on voldries descomprimir el/s arxiu/s.

7 Observacions i possibles problemes

- El programa no pot descomprimir fitxers més grans 60MB (al voltant de 65 milions de bytes). La causa és desconeguda.
- No s'ha contemplat la possibilitat de que alguns dels workers deixi de funcionar i per tant l'aplicació no és tolerable a errors.
- No s'ha considerat la possibilitat de que les diferencies entre fitxers sigui molt gran (es generaran moltíssims blocs, per exemple: un fitxer de 400KB i un altre de 2 bytes).
- El programa no pot guardar el comprimit a una ubicació que no existeix (és a dir, no crea la ubicació).
- A vegades la descompressió dona errors del tipus *bad coded file* del qual desconeixem la causa. Es podria pensar que és culpa de la concatenació de `BitVector` però a la majoria dels casos no dona problemes.

8 Exemples d'execució

Tots els fitxers que s'han posat a prova no superen els 60MB per tal de poder descomprimir i verificar que son iguals als originals. La comanda del shell de Windows utilitzada per comprovar-ho ha sigut la `FC`.

Tipus de fitxers	Nombre de fitxers	Bytes a comprimir	Ratio de compressió(%)	Pes del comprimit
jpg	25	57.497.497	99.93	57.481.908
flac	1	52.299.955	100.00	52.320.832
mp3	3	51.450.928	99.55	51.240.501
mp4	2	52.677.504	99.97	52.682.028
txt	4	1.421.312	53.60	761.856
xls	2	11.841.536	94.19	11.153.408
exe	1	25.632.768	100.00	25.653.248

Table 1: Exemples d'execució

El ratio de compressió és el resultat de fer la divisió del nombre de bytes resultants de la compressió del fitxer/s entre el seu pes original. D'aquesta manera si el ratio és més petit que 100 vol dir que ha comprimit.