Syddansk Universitet

Institut for matematik of Datalogi

Algoritmer og Datastrukturer
Hold S17

Projekt del 2

Studerende:
Aslak Johannes Vainio-Pedersen
Mads Petersen

Instruktor:
Mathias (Teo) Wulff Svendsen

23. maj 2014

University of Southern Denmark

Indhold

1 Introduktion

I Vores projekts anden del skal vi bruge de klasser vi lavede i den første del til at implementere en komprimerings-metode. Som vil blive sagt senere foregår det ved Huffmann komprimering i java. Modsat projektets første del har vi i anden del ved afstemning med vægtning afgjort at skrive rapporten på dansk (under protest). I det følgende vil vi beskrive de to dele af programmet, den første del der komprimerer en fil, og den anden del der dekomprimerer en fil komprimeret med programmet. Som det vil ses stødte vi på hårdknuder men fandt løsninger.

2 Encoding

Der skal i java implementeres en huffman algoritme til at komprimere filer, dette skal gøres ved hjælp af PQHeap og DictBinTree fra del I af projektet. Dette gøres ved hjælp af pseudo koden fra side 431 i kapitel 16 i Introduction to algorithms:

```
HUFFMAN
1 n = |C|
2 Q = C
3 for i = 1 to n - 1
4 allocate a new node z.
5 z.left = x = EXTRACT-MIN(Q)
6 z.right = y = EXTRACT-MIN(Q)
7 z.freq = x.freq + y.freq
8 INSERT(Q,z)
9 return EXTRACT-MIN(Q)
```

Til dette formål skal der læses fra en fil og skrives til en fil, til dette formål bruges FileOuputStream og FileInputStream i forbindelse med de udleverede BitStream klasser. Efter disse streams er oprettet og der kan læses fra dem bliver der lavet et array over de tegn der indgår i input filen, dette gøres med et simpelt while-loop:

```
while((input=in.read()) != -1)
        {
            alphabet[input]++;
            inBytes++;
        }
```

samtidig bliver antallet af input bytes også optalt da der bliver læst en byte med read metoden kan der bare lægges en til inBytes variablen for hvert gennemløb af løkken. Her efter bliver der, via et gennemløb af arrayet alphabet,

optalt hvor mange af de 256 mulige tegn der bliver brugt, i.e. har en værdi der er større end nul i arrayet, denne optælling bliver brugt til at initialiserer en PQHeap:

```
for(int i = 0; i<256 ;i++)
        {
            if(alphabet[i]>0)
                n++;
        }

        myQ = new PQHeap(n);
```

Der bliver herefter lavet et nyt element for hver værdi i alphabet variablen der er større end 0. Disse elementer for en key = frekvensen af et tegn, og med data lig et nyt træ-objekt med en enkelt node indsat der indeholder både den byte(repæsenterert af en int med værdi mellem 0-255) der repæsenterer ASCII tegenet og den tilhørende frekvens af dette tegn, og disse elementer bliver indsat i en PQHeap:

```
for(int i = 0; i<256 ;i++)
        {
            if(alphabet[i]>0)
            {
                DictBinTree tree = new DictBinTree();
                tree.insert(alphabet[i], i);
                Element ele = new Element(alphabet[i],tree);
                myQ.insert(ele);
            }
        }
```

Efter dette bliver pseudo koden der er nævnt i starten brugt, koden tager de 2 Elementer i PQHeapen der har de mindste keys, de 2 træ-objekter der er tilknyttet disse 2 elementer bliver så indsat i et nyt træ, der har en knude med key = deres sammenlagte frekvens, og de 2 knuder bliver indsat som højre og venstre barn til denne knude og dette nye træ bliver så indsat i Heapen. Dette fortsætter n-1 gange hvor n er antallet af forskellige tegn der indgår 1 eller flere gange i input filen:

```
for (int i = 1; i<n; i++)
        {
            DictBinTree left = (DictBinTree)myQ.extractMin().data;
            DictBinTree right = (DictBinTree)myQ.extractMin().data;
            DictBinTree tree = new DictBinTree();
            tree.insert(left.root().returnKey()+right.root().returnKey());
            tree.root().setLeft(left.root());
            tree.setNodes(tree.nodes()+left.nodes());
            tree.root().setRight(right.root());
            tree.setNodes(tree.nodes()+right.nodes());
            myQ.insert(new Element(left.root().returnKey()+right.root().returnKey(),tree))
        }
```

Variablen der holder styr på antallet af noder i træerne bliver holdt ved lige fordi så kan orderedTraversal andvendes til at printe alle noderne i et givent træ. Et

String array der bruges til at oversætte tegn i inputfilen til deres forkortede binære repræsentation bliver lavet ved hjælp af det fremtstillede træ, og en let modificeret treewalk metode der tillægger 1 eller 0 til en streng hvis der bliver gået mod venstre eller højre respektivt:

```
public String[] huffWalk()
    {
        theHuff = new String[256];
        String[] result = huffHelper(root,"");
        return result;
    }

    private String[] huffHelper(DictBinTreeNode start,String init)
    {
        String str = init;
        if(start != null)
        {
            huffHelper(start.left,str + "1");
            if(start.character != -1)
            theHuff[start.character] = str;
            huffHelper(start.right,str + "0");
        }
        return theHuff;
    }
```

Nu bliver input filen gennemgået anden gang og bytes bliver indlæst en efter en og oversat til deres forkortede(i hvertfald for de tegn der indgår hyppigst) repræsentation vha. det frestillede String array, først bliver de 256 ints der repræsenterer hyppigheden af mulige tegn skrevet vha. et for-loop og alphabet variablen, her efter inBytes der fortæller hvor mange bytes der var i den oprindelige input fil:

```
for(int i = 0; i<256 ;i++)
        {
            out.writeInt(alphabet[i]);
        }
        //int compBits = 0;
        out.writeInt(inBytes);
        while((input=in.read()) != -1)
        {
            int count = 0;
            String temp = theHuff[input];
            while(count < temp.length())
                    {
                        if(temp.charAt(count) == '1')
                        {
                            out.writeBit(1);
                            //compBits ++;
                        }
                        else
                        {
```

```
                    out.writeBit(0);
                    //compBits ++;
                }
                count++;
            }
        }
```

3 Decoding

I klassen Decode har vi metoden decompress der udfører det tunge arbejde med at dekomprimere filen, den tager argumenter for inputfil og outputfil. Der er tre interessante ting i starten af metoden;

```
FileInputStream infile = new FileInputStream(inFile);
```

som vi bruger, er en standard java-funktionalitet brugt til at læse filer.

```
BitInputStream in = new BitInputStream(infile);
```

er en java-klasse som er os givet specifikt til dette projekt.

```
FileOutputStream out = new FileOutputStream(outFile);
```

igen en standard java-klasse brugt til at outputte en fil.

Herefter kommer vi ind i den egentlige dekomprimering. Meget af funktionaliteten er den samme (det fagspecifikke ordvalg er "copypasta") som vi havde i encoding-delen.

```
for(int i = 0 ; i < 256 ; i++)
{
    input = in.readInt();
    alphabet[i] = input;
}
```

Først går vi igennem inputfilen tegn for tegn, indtil vi har læst de første 256 integers, som er første del af headeren, den del der beskriver hvor mange af hvert tegn der var i source-filen. Herefter gemmer vi i `inBytes` længden i bytes af source-filen, altså længden af hvad det der følger efter headeren skal oversættes til.
Den her del:

```
for(int i = 0; i<256 ;i++)
{
    if(alphabet[i]>0)
        n++;
}

myQ = new PQHeap(n);
```

finder antallet af brugte tegn i alfabetet og laver en PQHeap af den størrelse til senere brug.

De næste par afsnit er beskrevet dybere i encode-delen af denne rapport, og der henvises hertil. Kort kan dog siges at disse afsnit behandler opbygningen og udfyldelsen af strukturerne som vi implementerede i projektets del 1. Det vigtigste at notere er at vi i `myT` gemmer huffmann træet der indeholder alle 256 tegns komprimerings-signatur.

I det sidste while-loop sker transformeringen.

```
while(input != -1)
{
    DictBinTreeNode current = myT.root();
    while((current.returnChar() == -1))
    {
        input = in.readBit();
        if (input == 1)
            current = current.left();
        else
            current = current.right();
    }
    out.write(current.returnChar());
}
```

Så længe `input` ikke er -1 (som den kun er hvis vi er nået til EOF) vil vi analysere knuderne i `myT` en efter en for at finde sekvensen af $0-$ og $1-$taller som det aktuelle tegn svarer til. Det er hvad det indre while-loop hjælper med, så længe vi ikke er nået til et blad i træet vil det indre while-loop aktivere og vi læser bits i den komprimerede fil en af gangen for at finde ud af om næste binære tal i signaturen er 1 eller 0. Når loopet når et blad kan i bladets data aflæses det tegn som bladet repræsenterer med metoden `returnChar()`.

På den måde får man læst hele den komprimerede fil bit for bit og oversat den til de oprindelige source-tegn. En egenskab af den bagvedliggende algoritme betyder at kortere bit-strenge ikke kan forveksles når man er igang med at læse en længere bit-streng, så der er ikke nogen separator mellem de forskellige tegn, hvilken man ellers kunne forvente ville være et krav.

Til sidst har vi bare tilbage at lukke output-filen og input-filen så de ikke længere er aktive, og output filen vil være at finde i samme mappe.

4 Test

Programmet har været testet på en gratis bog "Little Peter : A Christmas Morality for Children of any Age"fundet på gutenberg.org. Inkluderet i afleverings mappen ved navn "pg45666.txt"på 175 KB. Når filen komprimeres fylder den 102 KB. Når den dekomprimeres igen er der et ekstra 's' i slutningen, dette sker fordi der bliver skrevet mindre end en fuld byte og derfor bliver der padded med 0'er, disse nuller når de bliver dekomprimeret svarer til et s. Dette

kan undgås enten ved at inkluderer størrelsen på den komprimerede fil, så det vides hvor mange bit der skal dekodes. Dette er test ved at tælle præcis hvor mange bit der bliver skrevet i det sidste while-loop og kun læse det antal når der dekodes igen. Denne fejl bliver størrer og størrer jo flere gange den samme fil komprimeres og de-komprimeres.
