CS337 - Programming Assignment # 1
Experiments in Compression

TA: Lara Schmidt (lschmidt.cs.utexas.edu)

Updated on: January 31, 2012

1 Introduction

Data compression is both useful and/or necessary in a variety of contexts. Storage and transmission of data in a compressed form typically requires fewer bits, thus utilizing resources more efficiently. In this assignment you will experiment with different compression techniques covered in class. Your task has two parts. First, implement a program to evaluate the amount of entropy in a text file and to produce an optimal Huffman code. Second, implement LZ compression and decompression. The input will be text files containing ASCII characters, without unicode characters. Java source code for reading and writing to files is provided on the class web site; you are not required to implement the Java code for file IO. You may develop your own interface for file IO if you wish to do so, provided it conforms with the specification described below.

2 Part 1 - Entropy and Huffman Coding

2.1 Description

Given a text file, it is possible to compute the probability of occurrence of each symbol in that file according to the first order model. From these probabilities the entropy of the characters in the file can be computed and an optimal Huffman code for the characters in the file produced. The entropy estimates the average number of bits required to encode each character in the text file. It is then possible to estimate the amount of compression achievable by that model for the input text file.

2.2 Implementation

Write an algorithm estimating the amount of compression achievable by a first order model given a text file. This algorithm should compute character probabilities and design an optimal Huffman code by constructing the best Huffman tree. Given the Huffman code, the actual length of the Huffman coded file can

Table 1: Example code

Symbol	Code
c	1
g	01
t	001
a	000

be calculated although you are not required to output the Huffman coded file. For example, consider the following Huffman code in Table 1. The number of bits needed for the compressed string "cccg" is 3*1+1*2 = 5.

3 Deliverables

Your program (call the source code "Huff.java") should be invokable with a file name (e.g., tmp) as a command line argument, as follows:

```
rivera.cs.utexas.edu$$ java Huff tmp
```

The expected output should be as follows:

```
Actual length of the file by Huffman coding is
45
and minimum achievable is
42.93
```

Note that both the actual output length with huffman coding and minimum achievable for the file are in number of bits.

One final remark: include whitespaces, commas, matching case, etc. in your frequency calculation. Automatic grading will be used, and there will be trouble if your output doesn't match exactly. Also, you can use the method giveArray() in the IO.Compress class to read characters from the file into an array; see the last section below.

4 Part 2- LZ Compression

4.1 Description

The Lempel-Ziv compression algorithm given in the course packet accepts a string of text and produces a sequence of pairs (n; c), where n is the index into the dictionary, and c is a character. The decompression algorithm reconstructs the dictionary using the pairs that are retrieved from a compressed file. During decompression each pair in the sequence causes a new entry to be added into the dictionary. For example, when the decompression algorithm processes the pair (n; c), it is known that the index n refers to a string that is already contained

in the dictionary. Therefore the new entry to be added into the dictionary is obtained by concatenating the string corresponding to n with the character represented by c. You are required to implement algorithms for compression and decompression using this method.

4.2 Implementation - Compression

Write a program that is invoked with the name of a text file. This program should create a compressed file and write it to the disk. Your implementation should use the trie based approach (see Implementation of the Dictionary, page 21 of handbook). The name of the compressed file is obtained by concatenating the name of the input file with the string ".myZ".

4.3 Implementation - Decompression

The decompression algorithm should be invokable with the name of a compressed file, and it should write a decompressed file to the disk. The name of the decompressed file is determined by concatenating the name of the input file with the string ".unZ".

4.4 Deliverables

You are required to submit a file called "Comp.java", which takes two command line arguments, the first argument is either the character 'c'(for compression) or 'd'(for decompression). The second argument is the name of the file to be compressed or decompressed. An example run of your program might look like this -

```
rivera.cs.utexas.edu$$ java Comp c tmp
rivera.cs.utexas.edu$$ ls -l tmp*
-rw-r--r-- 1 ankur grad 5029 Mar 25 22:26 tmp
-rw-r--r-- 1 ankur grad 5600 Mar 26 11:06 tmp.myZ
rivera.cs.utexas.edu$$ java Comp d tmp.myZ
rivera.cs.utexas.edu$$ ls -l tmp*
-rw-r--r-- 1 ankur grad 5029 Mar 25 22:26 tmp
-rw-r--r-- 1 ankur grad 5600 Mar 26 11:06 tmp.myZ
-rw-r--r-- 1 ankur grad 5029 Mar 26 11:06 tmp.myZ.unZ
rivera.cs.utexas.edu$$ cmp tmp tmp.myZ.unZ
rivera.cs.utexas.edu$$
```

4.4.1 Remarks

In some cases, you may find that the size of the ".myZ" file is actually larger than the input file. We have observed that it is so for small files. Explain why this happens in your readme file. Along with the Comp.java file you are also required to submit two files for which your program indeed does some

compression (i.e. the size of ".myZ" file is smaller than the original file). Name these files "smaller1.txt" and "smaller2.txt". You are also required to submit a text file for which your compression algorithm actually increases the size of the file. Name this file "larger.txt". Make sure the file names match exactly, because automatic grading will be used.

5 Questions and Clarifications

Clarifications regarding this project will be posted to the piazza. Though changes are not expected, you are responsible for any modifications found there. Please also try to ask questions there so other students may benefit from your questions.

6 Deadline and Sumission Instructions

The project is due Feb. 13th 2012 11:59PM. No late submissions allowed. As usual, your readme file must include the name of both the partners and their EIDs, as well as their e-mail addresses. Adhere to the project protocol! To package your project into a jar file, type:
"`jar cvf project1.jar Huff.java Comp.java smaller1.txt smaller2.txt larger.txt readme.txt`"
You can include additional java source files if you need them. Just make sure they are all in your jar file. To actually turn in the project, type:

```
turnin --submit lschmidt project1 project1.jar
```

To confirm your submission:

```
turnin --list lschmidt project1
```

Only one submission per group.

7 Appendix - IO source code

We will provide a file named IO.java on the class web page. This file implements three java classes IO.pair, IO.Compress and IO.Decompress, which can be used without modifications to complete this assignment. *If you use this file, include it in the jar file you submit.* The class IO.Compress can be used for implementing both the parts of this assignment. The class IO.Decompress can be used for implementing the decompression algorithm.

7.1 IO.pair

This class is used for storing the tuple (index,character). This is the type that will be returned by the method decode(...) in the class IO.Decompress. If p is an object of the type IO.pair, then p.index refers to the index of an entry in the dictionary, and p.extension refers to the character that must be appended to this string to create a new dictionary entry. The pair class also contains a field named 'valid'. This is helpful to determine when there are no more pairs to be decoded.

7.2 IO.Compress

The constructor of this class does two tasks. First, it initializes its internal objects so that they are ready to read from the input file. Second, it determines the name of the output file, and initializes it, so that the next call to encode(...) writes to the output file. This class provides a method giveArray() for copying all the characters from a file to a character array. It also provides a method encode(int x, char a), where the parameter x refers to an index in the dictionary, and a is the character that must be appended to the corresponding string. This method should be called during compression when a new pair is added to the sequence. The method done() should be called at the end of the compression process. Note that the method encode will be called several times during compression. The following code shows some parts of an example compression program.

```
/*infile is the name of the file to be compressed*/
public static void compress(String infile) throws Exception
  {
    IO.Compressor io ;
    /* Initialize a IO.Compressor object so that it is ready to
    * read the input file, and to write the output file to the disk */
    io = new IO.Compressor(infile);
    /* Read all characters from the input file to a character array */
    char[] carray = io.giveArray();
    ...
    /* Perform compression on the array carray, *
    * this part may call io.encode(...)  several times */
    ...
    /* Close all relevant files */
    io.done();
  }
```

7.3 IO.Decompress

The constructor of this class does two tasks. First, it initializes its internal objects so that they are ready to read from the input file (i.e. compressed file). Second, it determines the name of the output file, and initializes it, so that the

next call to io.append(...) writes to the output file. This class provides a method decode() which returns the next unprocessed pair from the sequence contained in the compressed file (if no more pairs are available, then it returns a pair whose 'valid' field is set to false). This method will also be called several times during the decompression process (like the method encode in IO.Compress). It also provides a method append(String s), which appends the string s to the output generated so far. The method done should be called at the end of the decompression process. The following code shows some parts of an example decompression program.

```
/*infile is the name if the file to be decompressed*/
decompress(String infile) throws Exception
  {
    /* Initialize a IO.Decompressor object so that it is ready *
    * to read the sequence of pairs from infile, and to write *
    * the decompressed file to the disk */
    IO.Decompressor io = new IO.Decompressor(infile);
    while(...)
    {
      ....
      /* Every call to io.decode() returns the next unprocessed pair from the *
      * compressed file.  next.valid is false if no more pairs are left.  */
      IO.pair next = io.decode();
      ....
      /* output is the latest entry that was added to the *
      * dictionary, append it to the decompressed file */
      io.append(output);
      ....
    }
    /* Close all relevant files */
    io.done();
  }
```