Report on LZW Algorithm

By Yu Sheng (Julia)

2018/4/25

I chose Java as the language for the implementation of LZW algorithm. And I used LZW algorithm to compress an input of sequence of characters to a sequence of indices.

For the first step, we created a dictionary of characters using Hashtable where Hashtable’s keys are integers and values are strings. I set all the characters (in ASCII table) to the key and their corresponding ASCII code to be the value. After that we will have 256 characters (with ASCII code from 0-255) in the dictionary. So the size of the current dictionary is 256, and we will set the code of next new character to be 256. Initially, we set string *s* to the first character of the uncompressed string. And we created an ArrayList named *output* to record the compressed output.

In next step, we set character *c* to be the next character in input, and we add up *s* and *c* to be a new string named *combo*. If *combo* is already a key in dictionary, then we set *s* to *combo*. However, if *combo* is not in the dictionary, we would add the value of *s*in dictionary to Arraylist *output* and create a new key *combo* in dictionary with the current dictionary’s size as the corresponding value. And since we had a new key, we should add 1 to the size of dictionary. And set *c* to be the current value of *s*. We would keep repeating this step until we exhaust the whole input.

After we went over all characters from the input, then we would now add last character, which is the current *s* to *output*. And the output now is the compressed text in integers, and we assume each integer takes four bytes.

After we implemented the compression, we can now do the compression for different input by calling compress(input string). If the input is a file, then we should use read file commend provided in professor’s handouts, input will be read in as string.

Table 1

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Size of Text | Output of Compress(in bytes) | #Entries in Dictionary (compress) | Output of Decompress(in bytes) | #Entries in Dictionary (Decompress) |
| Sample Text(Page 1) | 52 | 156 | 294 | 52 | 294 |
| Anassa Kata(Input from file) | 223 | 608 | 407 | 223 | 407 |
| Moby Dick(Input from file) | 1243005 | 944488 | 236377 | 1243005 | 236377 |

In Table 1 shown above, we recorded the size of original text, the size of compressed output and decompressed output and the size of two dictionaries for three different inputs. We found that for all three inputs, size of decompressed output is equal to the size of original input, and two dictionaries (compress and decompress) have the same number of the entries. We could say when we are decompressing, we are actually producing back the exact same dictionary as the one produced in the compressing process except key and value are exchanged. (i.e. in the compressed dictionary key is string, value is index but in the decompressed dictionary, key is index but value is string). However, there are still some differences between compression small files and relatively large file (moby10b.txt). For both sample text and anassa.txt, we found that their size of compressed output is bigger than the original text, which means LZW does not actually reduce the size. But when file size is larger, like when there are 1243005 characters in moby10b.txt, LZW compression starts to show its benefit. In this case, we can see that the size of compressed output 944488 bytes is 3/4 of original text size 1243005 bytes. As mentioned in the textbook, compressing using LZW algorithm in longer texts tend to yield better results.

**Code**

import java.io.File;

import java.io.FileNotFoundException;

import java.util.ArrayList;

import java.util.HashMap;

import java.util.Map;

import java.util.Scanner;

public class lab13 {

/\*\* Compress a string to a list of output symbols. \*/

public static ArrayList<Integer> compress(String uncompressed) {

System.out.println("Compressing...");

// Build the dictionary.

int dictSize = 256;

Map<String,Integer> dict = new HashMap<String,Integer>();

for (int i = 0; i < 256; i++)

dict.put("" + (char)i,i);

char[]uc=uncompressed.toCharArray();

String s=uc[0]+"";

char c;

ArrayList<Integer> output=new ArrayList<Integer>();

for(int i=1;i<uc.length;i++){

c=uc[i];

String combo=s+c;

if(dict.containsKey(combo))

s=combo;

else{

output.add(dict.get(s));

dict.put(combo,dictSize);

dictSize++;

s=c+"";

}

}

System.out.println("Dictionary size: " + dict.size() + " entries.");

output.add(dict.get(s));

return output;

} // compress()

/\*\* Decompress a list of output indices from LZW compress() to a string. \*/

public static String decompress(ArrayList<Integer> indices) {

System.out.println("Decompressing...");

// Build the dictionary.

int dictSize = 256;

Map<Integer,String> codes = new HashMap<Integer,String>();

for (int i = 0; i < 256; i++)

codes.put(i, "" + (char)i);

//String previous = "" + (char)(int)indices.remove(0);

StringBuffer result = new StringBuffer();

String previous = codes.get(indices.remove(0));

result.append(previous);

for (int current : indices) {

String s;

if (codes.containsKey(current))

s = codes.get(current);

else if (current == dictSize)

s = previous + previous.charAt(0);

else

throw new IllegalArgumentException("Bad compressed current: " +

current);

result.append(s);

// Add w+entry[0] to the dictionary.

codes.put(dictSize++, previous + s.charAt(0));

previous = s;

}

System.out.println("Dictionary size: " + codes.size() + " entries.");

return result.toString();

} // decompress()

public static void main(String[] args) {

try {

//ArrayList<Integer> compressed = compress("It was the best of times, it was the worst of times.");

//String lines=new Scanner(new File("anassa.txt")).useDelimiter("\\Z").next();

// ArrayList<Integer> compressed = compress(lines);

String lines2=new Scanner(new File("moby10b.txt")).useDelimiter("\\Z").next();

ArrayList<Integer> compressed = compress(lines2);

System.out.println(compressed+"\nLength: "+compressed.size()\*4);

String decompressed = decompress(compressed);

System.out.println(decompressed+"\nLength: "+decompressed.length());

} catch (FileNotFoundException e) {

e.printStackTrace();

}

} // main()

} // class LZW