Elec/Comp 326
Fall 2013
C Lab Assignment 2
LZW Compression

Overview

This assignment deals with implementing a version of the Lempel-Ziv-Welch (LZW) compression algorithm, similar to the Unix compress utility. As a part of the implementation you will also learn about I/O in C and implementing a simple hash table based dictionary (or Key-Value (KV) store).

The **first task** in the assignment involves **writing a C implementation** of the compression algorithm. You must test the implementation on several input files and record the compression ratio and the execution time required for the compression.

A stub for the function **compressFile** is provided in the file **compress.c**. You must code up the rest of the function by editing the file. No other files should be changed for this part.

The implementation will use a number of helper functions to do bit-level I/O and to maintain a dictionary (key-value store). Read and understand how these different routines work. You do not need to change any of them for the basic implementation.

In the **second part** of the assignment you will change the **hash()** function code in the file **kvstore.c** to help speed up the compression. The time for compressing files must be compared with the times for the basic implementation of the first part.

Description of LZW Algorithm

A high-level view of the LZW algorithm to be implemented is as follows. The input is treated as a sequence of bytes (characters in the ASCII code). The idea is to replace a string (sequence of consecutive characters) in the input file by a *code* word having a (generally) smaller number of bits. The assignment of strings to code words is done adaptively based on the input that the encoder has been seen so far. The mapping between strings and codes are stored in a dynamic *dictionary* (or KV store). When a new string is encountered, it is assigned the next available code value and inserted into the dictionary; strings that are in the dictionary are replaced by their code value in the compressed output.

Assume that we choose code words to be n bits long: we reserve the codes 0 through 255 for single characters, and reserve code 256 for the end of the input stream. We will assign code values

between 257 and $2^n - 1$ to strings that we encounter in the input. In our implementation, every code word (even the code for single characters) is exactly n bits long, so the encoding of single characters actually causes an expansion rather than a compression. However, for longer strings, using the code word rather than the individual characters in the string results in a savings. There are more complex implementations that dynamically change the value of n as the coding proceeds. These will not be considered in this assignment.

The algorithm scans the input file one character at a time. At each iteration, the algorithm maintains two pieces of information: a prefix **p** consisting of a substring for which a code word exists, and the immediately following character **c**. At the start, **p** is the first character of the file and **c** is the second. The phrase **f** = [**p**][**c**], the concatenation of **p** and **c**, is examined. If **f** is found in the dictionary, then **f** is made the new prefix **p**, and the next character of the input is assigned to **c**. If **f** is not in the dictionary, the code for the current prefix is output, the next unused code is assigned to the phrase **f** and added to the dictionary. For the next iteration, the new prefix **p** is set to **c** and the next character of the input is assigned to c. This continues until an EOF is encountered when reading the input. At this point the code for the current prefix is output, followed by the code for the end-of-stream.

We illustrate the algorithm using an example. Consider the input string: **ABABABAB**. To distnguish different instances of **A** and **B** in the string we index them and represent it as the string: $A_1B_1A_2B_2A_3B_3A_4B_4$. However, keep in mind that the A_i are all the same character **A** and the B_i are all the single character **B**. Initially **p** = A_1 and $c = B_1$; **f** = A_1B_1. Since **f** (the string **AB**) is not in the dictionary, we assign **f = AB** the first unused code 257, add it to the dictionary, and output the code for **A** (code value 66). Update **p** = B_1, $c = A_2$, and **f** = B_1A_2 (**BA**). Since **BA** is not in the dictionary, add **BA** with code value 258 to the dictionary, output the code for **B**, and update **p** = A_2, $c = B_2$, **f** = A_2B_2 (**AB**).

At this stage we find **AB** is in the dictionary so we continue: **p** = A_2B_2, $c = A_3$, **f** = $A_2B_2A_3$ (**ABA**). Since **ABA** is not in the dictionary, we add it (code value 259), output the code for **AB** (257), and update **p** = A_3, $c = B_3$ and **f** = A_3B_3 (**AB**). Once again, **AB** is found in the dictionary, so we continue: **p** = A_3B_3, $c = A_4$ and **f** = $A_3B_3A_4$ (**ABA**). We find that **ABA** is in the dictionary, so we continue updating: **p** = $A_3B_3A_4$, $c = B_4$ and **f** = $A_3B_3A_4B_4$ (**ABAB**). We add **ABAB** to the dictionary (code value 260), output the code for **ABA**, and update **p** = B_4, c = **EOF**. Output the code for **B** followed by the code for the end of stream 256. Hence the output created is: [**65**] [**66**] [**257**] [**259**][**66**][**256**]. Assuming each code is represented in $n = 13$ bits, the

output including the end-of-stream marker is 78 bits. In a larger example one will typically obtain actual compression in the size! However, the example illustrates why the compression algorithm may sometimes result in an expansion of the input file.

Details of Part 1

- Create subdirectory **lab2** in your **elec326** subdirectory and connect to it.
- Copy all the files needed for the lab from their home position using the command:

  cp ~pjv/326/lab2/* .

  Do not forget the "dot" at the end of the command.
- Check that all the files that have been copied by executing the command **ls -l**. You should see several files:
  - C source files: **driverc.c**, **compress.c**, **kvstore.c**, **bitio.c**
  - C header files: **cons.h**, **bitio.h**
  - Test data files: **X1, X2, X4, X8, X16, X32**
  - Executable file: **demo**, **expandfile**
- The file **compress.c** contains the stub for the routine **compressFile**( ) that you will write.
- Begin by reading and understanding the various source and header files and how they fit together.
- Your task is to implement the function **compressFile**() to implement the LZW algorithm as described. Comments provided in the function will help you along.
- The file **cons.h** defines the various constants used in the program. In particular **BITS** is the size of each code word and is set at **13** bits. Also the constant **LINEAR** is set to **1**. This will cause the use of the default (slow) version of the dictionary lookup.
- Once completed, you must compile and link the various files using the command:

  gcc driverc.c kvstore.c bitio.c compress.c

  The executable **a.out** will be invoked with the command:

./a.out $\langle infile \rangle$ $\langle outfile \rangle$

In the above command $\langle infile \rangle$ is the name of the file to be compressed and $\langle outfile \rangle$ is the name of the file in which you want the compressed output to be stored.

- Compress each of the input files **X1** to **X32** in turn, using your compress routine, and record the compression ratio. For instance to compress input file **X1** and place the output in file **out1** type the command:

  ./a.out X1 out1

  This should create the compressed output in the file **out1**. To find the sizes of the files execute the command:

  ls -l

  The field before the date is the size of the file in bytes. Compute the ratio of the size of the input file to that of the compressed output file to get the **compression ratio**.

  Also record the execution of the programs using the command

  time $\langle executable \rangle \langle infile \rangle \langle outfile \rangle$.

  Record the execution time required for each of the input test files.

Details of Part 2

The slow execution of the compression arises due to the simple implementation of the dictionary that sequentially searches the dictionary for a key match every time it is invoked. In contrast, a **hash table** implementation will associate a key K with a location $h(K)$ in the dictionary in which the key and value pair will be stored. The function h is a deterministic mashup of the bits of the key to minimize the chances of *conflict*. Two keys K_1 and K_2 are said to *conflict* if $h(K_1) = h(K_2)$. When a conflict occurs the key being inserted tries an alternative location at a fixed *offset* from the initial index. (In the provided implementation the offset is chosen (somewhat arbitrarily) as **index+1**). iIf the second location is also occupied it searches another location an additional offset away and so on, as shown in the code for **findPhrase**.

You should not change **findPhrase** in any way.

Your task is to write an appropriate hash function and include it in function **hash**( ) of **kvstore.c**. Do this by appropriately changing the *else* clause associated with the test of variable **LINEAR**. Also change the value of **LINEAR** in **cons.h** to **0** so that the implementation uses your hash function rather than the slow default implementation.

Finally, test the new implementation for both correctness and performance by timing the compression of the files **X1** to **X32** using **time** as before. Record all compression ratios and timings for both parts in a simple text file called README.

Notes

- Do all work in your **elec326/lab2/** subdirectory on **CLEAR**.
- A demo executable **demo** is available for comparison. Your compressed output must match that produced by the demo program. Invoke the demo as:

  ./demo $\langle inputfile \rangle \langle outputfile \rangle$

- An executable **expandfile** is available for decompressing the compressed file. Invoke it as (for example)

  ./expandfile out1 outX1

  Here **out1** is the file obtained by compressing **X1**. The above execution of **expandfile** will expand **out1** back to the original and place it in the file **outX1**, which should match the original **X1** exactly.

Due Date: 11:59 pm, Thursday, October 17th
GOOD LUCK