Starleaf LZW Decompressor

Jake Reynolds

April 8, 2016

Zip contents:

- lzw.cpp/h : An api for lzw compression and decompression
- main.cpp : A program that utilises/demonstrates the api
- LzwInputData : Provided sample files
- LzwOutputData : Decompressed sample files, and subsequent recompressions
- runDiff : Small script to test that recompressed files match provided sample files

To build:
The code was developed using CLion, but I have tested building with GCC (4.8.4), which required adding some standard C headers. (C++11 standard is only needed for to string in main()).

```
g++ main.cpp lzw.cpp -o lzw -std=c++0x
```

To run:
Run ./lzw with no arguments to see usage. The line below will decompress provided input file to output.txt.

```
./lzw <input file>
```

The command '-c' or '-d' can be used to set mode to compression or decompression respectively. Default if neither is specified is decompression. -o <outfile name> allows user to change outfile name, default is output.txt/.z

The line below will decompress and recompress all provided sample files, writing output to LzwOutputData. (double dash before test)

```
./lzw –test
```

The line below will build lzw, run it on all sample files, and ensure the output is correct.

```
g++ main.cpp lzw.cpp -o lzw -std=c++0x && ./lzw –test && ./runDiff.sh
```

Development:
Although the decompressor was done in one sitting, I have expanded the task into creating a (more tricky) compressor, mostly because I found the task interesting. I have aimed to make my code functional and correct as opposed to highly optimised. For instance, knowing that the alphabet consisted of only readable text could lead to optimisations, but instead I opted to create an api which should work for any data (treats all bytes as an unsigned char). This has been tested on binary files; lzw can correctly compress and decompress its own binary.

Compressor:
With regard to improving a naive compressor, one approach is to change how the dictionary is ordered, and therefore how it can be searched. Instead of ordering by code, order by the corresponding string. A good

example of this is a map, and I have created an implementation of this (fastDictionary/fastCompress). A quick Valgrind profile (and common sense) shows this is a great improvement.

Further optimisations:
If I were aiming to optimise this tool, there are a few areas I know could be improved:

- Buffer file operations. I am only reading/writing 1/3 byte(s) at a time, which is very inefficient. Blocks (eg 100 bytes) can be read in in one go, and then consumed until data from the next block is needed. Write-behind/read-ahead threads could be used to do this asynchronously.
- A standard map is implemented with a binary tree. This could, memory allowing, be replaced with a custom tree, which is base |alphabet| rather than base 2. For instance, if the alphabet is ABC..Z, each node would have 26 pointers, as well a corresponding code for itself. An example node could be:

  Name : 'HZQ'
  Code : 0x108
  Pointers [26] : {&'HZQA', &'HZQB' .. &'HZQZ'}

  A null pointer would indicate that the item is not in the dictionary. The pointer array could be replaced with a map if memory demands required it; this would be perhaps necessary if the alphabet was 0x00-0xFF, as therefore each tree node would have 256 pointers. Note that due to the more rigid structure, there will be no rearrangement/ordering during updates as can occur when using a std::map.
- Another advantage of a custom map is an ability to store the result of a previous search. This is useful due to reading the input sequentially. A search is done for 'HZQ', and a node is returned. 'A' is read in. The search for 'HZQA' can begin at the previously returned node.
- Attempt to reduce some of the flow control, as there is a fair amount which could slow down the (relatively fast) decompression.
- Increase safety and error reporting, especially with file I/O. The only handled errors are if the files cannot be opened, but there is no catch for any other type of exception or file error.