FIT3042 System Tools and Programming Languages

Semester 1, 2015

Assignment 1

Preprocessing for Rice Coding in C
- Data Compression as Machine Learning

Worth: 20% of final mark

Must be completed individually

Due: Tuesday, 28th April, 2015, 11.55pm

When students learn to program computers, the emphasis is very much on being able to write programs. However, in the commercial world a lot of programmer effort is spent in maintaining and extending legacy software. This assignment will involve the use of some legacy software, namely a program which carries out data compression and decompression using Rice codes. In this assignment you will write software that can pre-process data so it can be more effectively compressed by that software. Naturally, you will need to write software to undo the effect of the preprocessing so that when the compressed data is decompressed to reproduce the preprocessed data, it is possible to restore the original data exactly.

As part of this assignment you will write shell scripts that verify that the encoded output can be correctly decoded. Your shell scripts must be written in a way which make them easy to apply to any choice of compression/decompression program. These scripts should also allow pre-processing and post-processing of data to be included in the compression process.

Rice coding as a means of compressing data

Suppose we have symbols $0, 1, 2, \cdots, A-1$ where A is the alphabet size, i.e. that there are A symbols. The fundamental Rice encoding scheme when coding symbol i will output i one bits, followed by a 0 bit. This scheme will be at its most effective if symbol i is more likely to occur than symbol j when $i > j$. This scheme can handle an unbounded number of symbols. Decoding is very simple. The number of one bits read before a zero bit is read is the value of the decoded symbol. The simplicity of encoding and decoding has been exploited in space missions where transmissions from deep space probes have been compressed using Rice coding.

Shannon's information theory says that outputting a single bit to encode symbol 0 is optimal if symbol 0 occurs with a probability of 0.5. Similarly, outputting $i+1$ bits when symbol i occurs is optimal if symbol i occurs with a probability of $2^{-(i+1)}$. For this distribution, the expected number of bits that are output is 2. Many information sources cannot be encoded using so few bits on average so the fundamental Rice coding scheme has been generalised by including a coding parameter, k, where k is a non-negative integer. Suppose all symbols can be represented by a b-bit integer. For example, any one of 256 distinct symbols can be represented by an 8-bit integer. An extended Rice coding scheme will encode symbol i by encoding $i \gg k$ using the Fundamental coding scheme and the rightmost k bits in the representation of i will be encoded as they are. If the decoder knows what k is, then decoding is very straightforward.

Suppose a program is given a stream of symbols and tries to encode these using Rice coding. The program `rice_coder` takes a block of symbols and tries encoding the symbols using all possible values of k. It then encodes the optimal value of k, that is, the value that leads to the smallest number of output bits, and then uses that value of k to encode the symbols in the block. The decoder needs to know what the block size is. One blocksize might be used for all blocks. It reads bits from

the compressed data to recover the value of k and then uses that value of k to decode the values in the block.

Preprocessing symbols to increase compressibility via Rice Rice coding

The number of compressed bits would be minimised if we knew how to order the symbols from most frequent to least frequent. We could then map the most frequent symbol to 0, the second most frequent symbol to 1 and so on. As long as the decoder knows how this mapping was carried out, it could decompress the encoded symbols and then undo the mapping.

Suppose that the encoder has a table that tells it how to carry out this mapping. It is possible for the encoder to use the table to encode the current symbol, and then use the knowledge of the identity of the symbol that has just been encoded to 'improve' the mapping table. Consider the following encoder algorithm.

```
Start with mapping table in some standard state
While we have data to encode
  read a symbol
  use the mapping table to map the symbol
  encode the mapped symbol using Rice coding
  change the mapping table using the symbol.
```

The decoder can decode this using the following algorithm:

```
Start with mapping table in some standard state
While we have data to decode
  read a mapped symbol
  decode the mapped symbol using Rice coding
  use the mapping table to recover the symbol
  change the mapping table using the symbol.
```

This will work as long as the encoder and decoder change the mapping table in exactly the same way! They may as well use exactly the same source statements ...

Learning from the data to change the mapping table

Suppose the encoder has just encoded symbol i. Because it has just seen symbol i, the encoder might decide to map symbol i to a lower integer value. This means that some other symbol must have its mapping changed as well. A really simple way of doing this is **`swap_next`**. The algorithm is as follows:

```
Suppose symbol is mapped to the value rank.
If rank is not already 0
   find the symbol that is mapped to the value, rank-1
   make the value of this symbol be rank
   make the new value of symbol equal to rank-1
```

This can be carried out using two arrays, one which is indexed by symbol to give the value of the mapped symbol and a second array which is indexed by the rank and can be used to find the symbol that is mapped to rank.

Another re-ranking approach, `swap2front`

`swap_next` will change the rank of a symbol by at most 1. If we already have a good translation table, it will change it by a minimal amount. However, if we do not have a good

translation table, perhaps because we have only processed a small number of symbols, we might want a way of letting a symbols value change a lot more quickly. This motivates the use of the following algorithm, **swap2front**.

```
start with number_of_symbols equal to 0.
start with each symbol being mapped to its own symbol value
```

Then, **swap2front** updates the table as follows:

```
If the symbol is mapped to a value which is < number_of_symbols
  use swap_next to change the table
else
  find the symbol whose value is number_of_symbols
  make that symbol have the value of symbol
  make symbol have the value of number_of_symbols
  add one to number_of_symbols.
```

Notice how this works. It depends on the following invariant:

A symbol which has occurred at least once will have a value which is strictly less than `number_of_symbols` while symbols which have never been seen will have a value which is greater than or equal to `number_of_symbols`.

swap_next will be applied to symbols which have already been seen at least once. When a symbol is seen for the first time, it joins the the symbols which have already been seen at the end.

Task 1 - Implementing a preprocessor for Rice coding - 5 marks

Write a program, `adaptive_re-rank`, which reads a stream of bytes, treats those bytes as symbols with values in the range 0 to 255 inclusive, and outputs a stream of bytes which have been preprocessed by applying either **swap_next** or **swap2front**. The appropriate option should be specified as a command line argument. The output stream of bytes should have exactly as many bytes as the input stream.

Write a program, `adaptive_de-rank` which reads a stream of bytes which it assumes have been processed by `adaptive_re-rank`, and undoes that pre-processing to recover the original input to `adaptive_re-rank`. The output stream of bytes should have exactly as many bytes as the input stream. Again, the appropriate option, `swap_next` or **swap2front**, should be specified as a command line argument. If the same option is specified for both `adaptive_re-rank` and `adaptive_de-rank`, then applying `adaptive_re-rank` followed by `adaptive_de-rank` should be to leave the input unchanged.

Verify that your two programs are functioning correctly.

Use the program `rice-coder.c` to compress a number of test files. Note how much compression was achieved. Then preprocess the test files using both of the options and see what effect preprocessing has on the amount of compression that is achieved.

There will be test files provided on the Moodle site but you should also perform testing using files of your own. In particular your testing should include the `.c`, `.o` and final executable image of the C programs you write for this task.

Program usage information:

```
adaptive_re-rank [-swap_next | -swap2front]
adaptive_de-rank [-swap_next | -swap2front]
```

Task 2 - Context-dependent Preprocessing - 5 marks

Amend the two programs you wrote in the previous task and change them so they are now perform context-dependent re-ranking. In task 1 there was a single table that was applied a symbol.

A simple extension of this approach is to have more than one mapping table and to choose that table in a way which depends on the context of the current symbol.

For example, suppose we are processing english language text. If we knew that the symbol which had just been processed as a `q`, we would know expect that the next symbol would be `u`. If we are dealing with 256 possible symbols, then we have 256 tables and we could select the mapping table based on the previous symbol which has just been processed. The very first symbol has no preceding symbol so we can only use the initial mapping table.

Given that we are dealing with byte-oriented data, we can actually choose to use 257 mapping tables. We use 256 tables, one for each possible preceding symbol, and we have a default table. The context-dependent tables, i.e. the tables that depend on the preceding symbol, only get updated when they are used. The default table gets updated with every symbol. Apart from the very first symbol, each time we meet a symbol we have a choice of using either a context-dependent table or the default table. The default table gets updated at every symbol so it has more chance to learn about the data than a context-dependent table. For each context-dependent table we can keep a count of how often it has been used. If we are required to choose between a context-dependent table and the default table we might do choose the context-dependent table if it has been used more than a threshold-dependent number of times.

Program usage information:

```
context_re-rank [-swap_next | -swap2front] -tn
context_de-rank [-swap_next | -swap2front] -tn
```

where `-tn` specifies the threshold for switching between using the default table and a context-dependent table.

Task 3 - Lossless Compression of grey-scale images with more than 8 bits per pixel - 5 marks

The Rice coding algorithm can be applied to any kind of file. The basic approach in `rice-coder.c` assumes that the data consists of symbols that fit into one byte. There are many types of data where the basic unit of data takes up two or more bytes. For example, CD sound consists of 16 bit values. Greyscale images often have one bit pixels whose values are in the range 0 to 255 inclusive but there are also greyscale images which can have more than 8 bits per pixel. Often greyscale image files in medicine have 12 bit pixels. An 8 bit technique can be applied to these kinds of images by splitting the each pixel into two parts - the 8 most significant bits and the 4 least significant bits. Thus, an image file containing 12 bit pixels can be split into two files which have the same shape and number of pixels. One of these files has 8 bit pixels i the range 0 to 255 inclusive while the other has 4 bit pixels in the range 0 to 15 inclusive.

Write a C program, `split_pgm`, which can take an image file in pgm format as input. The program reads the header of the image file and if the pixel values can fit into one byte, the program does nothing. If the pixel values need to be stored in 2 bytes, then the program should produce 2 image files in pgm format. The first file should contain the most significant 8 bits of each pixel value while the second should contain the remaining bits in the pixel value.

You should also write a C program, `join_pgm`, which takes the two files produced by running `split_pgm` and reassembles a pgm which contains the same image as the original file.

As well as testing that your programs work correctly, you should see what affect splitting an image file in this way has on the compression when `rice-coder.c` is applied to original file and to the original file split into two files in pgm format. In the first two tasks you came up with 4 variants of preprocessing schemes. For this task, you can carry out testing using that variant which seems to perform best on pgms with 8 bit pixels.

A `pgm` file consists of two components: a header and the image contents. The header consists of lines of text which are terminated by `\n`. The first two characters must be the 'magic number', `P2` for a `pgm` file with the pixel values in ASCII, or `P5` for a `pgm` file in binary (one or two bytes per pixel). The P2 format is rarely encountered these days so we will be referring to the P5 or 'raw'

format whenever we refer to `pgm` in the rest of this document.

After the magic number, the header then contains 3 integer values as ASCII - the number of columns, the number of rows and the maximum pixel value. The header can contain comment lines, line which begin with '#'. After the header come the actual pixel values. Pixel values are stored one row at a time. For a `pgm` in raw form, if we assume that the image data does not contain any comment lines, once we know the number of rows and columns, the image data can be extracted by using a Unix tool like `tail` to extract the last (number of rows) x (number of row bytes). This technique assumes that the file only contains one image.

More information on the `pgm` can be found on the final page of this assignment as well as by looking at Wikipedia for information on `Netpbm`.

Program usage information:

```
split_pgm filename.pgm
join_pgm filename1.pgm filename2.pgm
```

`split_pgm` when given an input file called `image.pgm` should produce two output files called `image.top8.pgm` and `image.residue.pgm`

Task 4 - 5 marks

Write a shell script that is given a directory of files and runs the data compression system on those files, producing a compressed file for each file. A "data compression system" might consist of a single program or 2 or more programs which act in combination. Then "a data decompression system" is run on the encoded files to produce a decoded file. The decoded files are then compared against the original files to verify that decoded file is, in fact, byte-for-byte identical with the original file. For each of the original files, the shell script should report on whether the file could be successfully encoded and decoded.

It should be possible to use the script on any of the data compression systems developed in the first three tasks of this assignment.

Testing and error handling

Your software should produce correct output for all legitimate input files, and fail gracefully, with useful error messages, when faced with incorrect user inputs (for instance, the wrong number of command-line arguments). It is your responsibility to test this! You will be penalized if your program crashes, hangs, or otherwise misbehaves, even on malformed user inputs.

Target system

Your program will be compiled and run on the same Ubuntu virtual machine configuration as used in labs. It is *your* responsibility to make sure the program compiles and runs on this machine.

C is not as trivially portable as Java. If you do not make sure it works on the target machine, it probably won't. You have been warned!

Makefiles and submission

You must submit a single archive file in gzipped `tar` format through Moodle. See `info tar` for information how to create a gzipped tar archive.

You must include a working `Makefile` with your submission. It should be possible for the marker to build all the executables by changing to the root directory of your submission and running `make` at the command line.

You should include a README file, in text format, that describes (at a minimum):
your name and student ID number, how to compile and run the program, what functionality is and is not supported, and any known bugs or limitations.

You should also complete the electronic plagiarism statement as normal.

Good programming practice

It is expected that you will follow good C programming practice in this assignment. This includes generic good programming practices such as:

- Code that compiles! Code that does not compile will receive a maximum mark of 8/ 20.
- Decomposition of programs into functions (methods), and grouping of methods into source files to allow reuse.
- Appropriate layout. We do not care which brace/indentation style you use, as long as you are consistent.
- Use of sensible variable and function names.
- Appropriate commenting. If you have no idea how to comment your code you could start with the GNU Coding Standards.

Some C-specific issues to remember include:

- Correct memory management.
- C does not support exception handling). Everything that can produce an error should be checked to see if it does!
- Use of function prototypes.
- Correct use of header files.
- Minimizing use of high-risk programming constructs (global variables, `goto`).
- Clean compilation, i.e. code that compiles without warnings with `-Wall` turned on.

Marking criteria

Your assignment will be marked on:

- Functional correctness: does the system do what it is supposed to do?
- Efficiency: are your programs efficient in their use of resources, primarily CPU time and memory?
- Design: are your programs cleanly engineered?
- Coding practices, including readability.
- Compliance with submission instructions.

Due date: Tuesday, April 28th, 2015.

The following information about the `pgm` file format is extracted from
`http://netpbm.sourceforge.net/doc/pgm.html`

THE FORMAT

The format definition is as follows. A PGM file consists of a sequence of one or more PGM images. There are no data, delimiters, or padding before, after, or between images. Each PGM image consists of the following:

(1) A "magic number" for identifying the file type. A pgm image's magic number is the two characters "P5".

(2) Whitespace (blanks, TABs, CRs, LFs).

(3) A width, formatted as ASCII characters in decimal.

(4) Whitespace.

(5) A height, again in ASCII decimal.

(6) Whitespace.

(7) The maximum gray value (Maxval), again in ASCII decimal. Must be less than 65536, and more than zero.

(8) A single whitespace character (usually a newline).

(9) A raster of Height rows, in order from top to bottom. Each row consists of Width gray values, in order from left to right. Each gray value is a number from 0 through Maxval, with 0 being black and Maxval being white. Each gray value is represented in pure binary by either 1 or 2 bytes. If the Maxval is less than 256, it is 1 byte. Otherwise, it is 2 bytes. The most significant byte is first.

A row of an image is horizontal. A column is vertical. The pixels in the image are square and contiguous.

All characters referred to herein are encoded in ASCII. "newline" refers to the character known in ASCII as Line Feed or LF. A "white space" character is space, CR, LF, TAB, VT, or FF (I.e. what the ANSI standard C isspace() function calls white space).