Project 2:Huffman Encoding
Submission Due:Monday, 17, October 10AM

COMP20003 Semester 2 2011

September 16, 2011

1 Aim

The aim of this project is for you to get experience with writing ANSI C code to implement non-trivial greedy algorithms using advanced data structures like tree

2 Project Overview

Data compression algorithms are an important tool used to store data more efficiently, by encoding the data such that it takes up a fewer number of bits. Many modern compression standards such as MP3 Audio use Huffman encoding procedure. In this project you will write a program that can encode files using Huffman compression, as well as decode those files. You will also work on different implementations of priority queues. You will also provide a report that shows an understanding of the algorithms involved.

In normal text files, each character is represented by 1 byte (ASCII format). Huffman compression is a particular encoding scheme which realises that some characters are used more than others. It encodes the data using less bits for the more common characters and more bits for the less common ones. This works well for English text where characters like 'e' are very common and characters like 'q' arise much less. Huffman compression looks at the frequency of each character used, and generates codes of appropriate length for each character. Consult Section 5.2 of DPV for details on how the algorithm works.

Any encoding scheme only works if both the encoding side and the decoding side both know the compression scheme used. For example, files will often have an appropriate file extension to let the user know what program to read it with. Furthermore, the files will have a header which tells the decoding program the details of how the file is encoded. In this project the header will specify the character frequencies, from which the code for each character can be determined.

3 Your work

You will be implementing Huffman coding and decoding algorithms. Some starting files are given to you. Your program should work with options for performing various functions of Huffman encoding and decoding. These options will be explained below:

3.1 Task 1: Encoder

This part of the program takes an input file and compresses it using Huffman encoding. The program should be called `huffman` and the `-e` option used to specify that the input file is being encoded. The other options will be explained below.

```
huffman -e [-f] [-c] [-o outputfile] inputfile
```

First read the input file and determine the frequency of each character in the input file (this includes the frequency of all non-printing chracters). If the `-f` option is specified print out the frequency of each character to `stdout`. Give the characters by their ASCII values, in ascending order. Example:

```
? echo Hello World > hw.txt
? huffman -e -f hw.txt
Frequencies:
 10: 1
 32: 1
 72: 1
 87: 1
100: 1
101: 1
108: 3
111: 2
114: 1
```

The above function is already implanted for you. You could modify if you like to suit your style of programming.

Note: Your output should be formatted exactly as it is shown above, with the ASCII values right justified to the third column. This can be done with the `%3d` option in `printf`, see the `man` pages for more details.

Create a single node Huffman tree for each character that is used in the input file (ie, has a non-zero frequency). Insert these trees into a priority queue.

In this project you will experiment with different implementations of priority queue. As a default, your implementation of priority queue should be based on sorted linked list. You will work on other implementations for bonus marks (see section on bonus marks).

To implement the Huffman encoding follow the following steps:

- Step1: Initialize n one-node trees and label them with the characters of the alphabet. The node contains the frequency of the character.
- Step 2: Repeat until a single tree obtained:
  Find two trees with smallest weight;
  Make them left and right tree of a new tree;
  Sum the weights of the left and right tree and record it in the new root‘s node.
- Read out the codewords for all alphabets by traversing the tree from root to every leaf in the tree (left edges : label 0 and right edges have label 1)

Traverse the Huffman tree and determine the Huffman code for each character. If the `-c` option is specified, print out the codeword of each character to `stdout`. Give the characters by their ASCII values, in ascending order. Example:

```
? echo Hello World > hw.txt
? huffman -e -f hw.txt
Codewords:
 10: 010
 32: 0111
 72: 0110
 87: 1101
100: 1100
101: 1111
108: 10
111: 00
114: 1110
```

Note: The formatting requirements detailed for the frequency output apply here also.

If an output file is specified by the `-o` option, the program should write the encoded file. First write the header. The header should be formatted as follows:

N	C_1	F_1	C_2	F_2	...	C_N	F_N

Where:

- N is the number of characters used, stored as an `int`.
- C_i is ASCII value of the i^{th} character, stored as a `char`.
- F_i is the frequency of the i^{th} character, stored as an `int`.

Characters should be in increasing order by ASCII value and only characters with non-zero frequency should be included.

Finally read the input file again, encode each character and write it to the output file.
The given file huffman.c implements the option handling routine, you need to fill the details.

3.2 Task 2:Decoder

This part of the program takes an input file that was compressed as in Task 1 and uncompresses it. In this part, use the `-d` option used to specify that the input file is being decoded. The other options will be explained below, but they are similar to what was required in Task 1.

```
huffman -d [-f] [-c] [-o outputfile] inputfile
```

First read the input file and extract the frequencies of each character from the header. The header is formatted as specified in Task 1. If the `-f` option is specified, print the frequencies as in Task 1.
Build the Huffman tree using the frequencies extracted from the header. You may find it useful to reuse the functions you wrote in encoding step. If the `-c` option is specified, extract

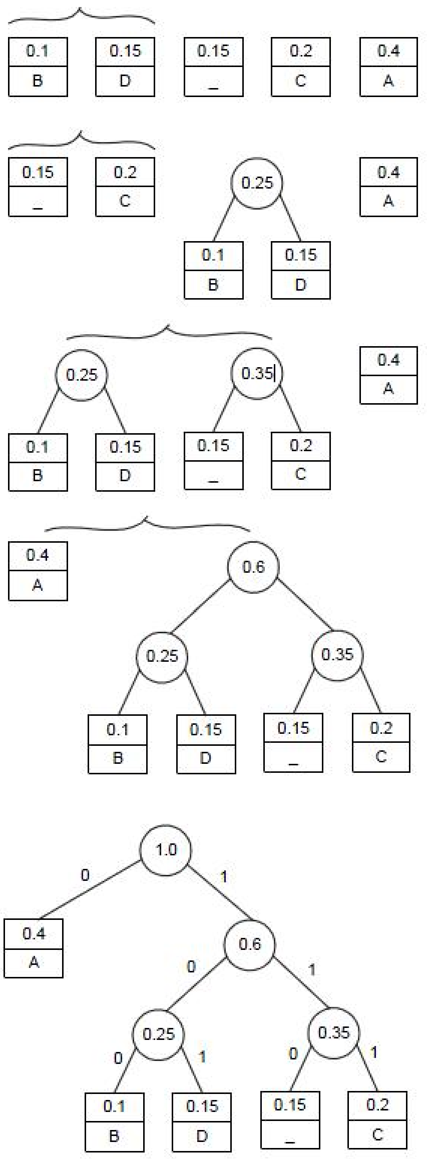
the codewords and print out the codeword for each character as in Task 1.

If an output file is specified using the `-o` option, decode the input file data and write the data to the output file. The sum of all frequencies will give you the length of the data, use this information to determine when to stop decoding.

4 Example

Below is the example for constructing a Huffman code for the following data:

Character	A	B	C	D	_
Probability	0.4	0.1	0.2	0.15	0.15



5 Bonus Marks

In the above implementation, you used priority queue implemented as a linked list. You will implement more efficient versions of priority queue. The tasks in this section are not mandatory. You will implement the following improvements:

- Priority queue using binary search tree (BST). This improvement should work with the option `-B` along with the rest of the options noted in the main project.
- Priority queue using binary heap. This improvement should work with the option `-H` along with the rest of the options noted in the main project.

Your task for bonus mark can receive a maximum of four bonus marks (This is subject to the constraint that you can NOT get more than 30 marks for the project component of the assessment).

If you have implemented `-B` and/or `-H` options and wish to be considered for bonus marks, you should mention it in the beginning of Huffman.c as comments.

6 Report

Your report should provide a formal analysis of the algorithms involved.

Concise and informative reports are preferred of long, verbose ones.

Further Notes

The following are requirements that apply to the program as a whole:

- If neither the `-d` or `-e` options are specified (or if both are), if invalid options are specified or if no input file is specified, print out a usage message.
- If the program is unable to complete, an error message should be written to `stderr`. For example, if an input file cannot be read, and output file cannot be written to, or if a file cannot be decoded due to incorrect formatting.
- Although this program is designed for text compression, your program should be able to encode any file.
- Source code should have lines no longer than 79 characters. This is to ensure that lines do not wrap when printed out.
- Source code should be commented, well formatted and have sensible variable names. It's in your interests to make your code as easy as possible to read for the markers.
- The program should be ANSI C, and should compile on the CSSE student machines, with no errors or warnings.
- Remember to close any files you open, and free any memory allocated.
- Return values for all library functions should be checked, and error conditions handled.
- Please consult the discussion forum and project 2 pages on LMS for any updates to this project specification. Join the discussion list to receive ongoing guidance on this project.

- University Policy on Academic Honesty and Plagiarism: It is University policy that cheating by students in any form is not permitted, and that work submitted for assessment purposes must be the independent work of the student concerned. Please see the policy for more details:
http://www.services.unimelb.edu.au/plagiarism/policy.html.

Hints

- The Huffman tree must be built exactly the same way when encoding and decoding for a file to be output correctly. The easiest way to ensure this is to use the same function in both parts.
- Use the provided `getbit` and `putbit` functions to write to files one bit at a time (rather than one byte at a time). They are provided in the files `bit_io.c` and `bit_io.h`. Descriptions of how they work are in `bit_io.h`.
- The functions `fread` and `fwrite` in the standard library will help you to read and write integers to a file.
- Command line options can be parsed using `getopt`. You have been given code for this.
- The unix programs `xxd` and `hexdump` may be useful when examining the encoded files.
- Tools such as `splint` may help you find memory leaks.
- The implementation for this program is dependant on the byte-order of the machine as well as the exact size of `int`. This is to keep the project as simple and focus more on the algorithms involved, in practice a more robust implementation would be required. Thus please be aware when testing that files encoded on one machine may not decode properly on a different machine.
- In general, if the frequency of each character is not unique then different implementations will produce different encodings.

Provided files

- `Makefile`: A skeleton makefile to build the program with the required options.
- `bit_io.c/.h`: Functions to allow writing to files one bit at a time. Documentation provided in `bit_io.h`.
- `*.txt`: Example text files to test your code with.
- `huffman.c`: The program file with implementation of frequency table and options handling.

Assessment

This project is worth 15% of your final mark. The marks will be broken down as follows:

- **Basics** [4 marks]: Clean compile, proper memory management, good coding style.

- **Correctness** [8 marks]: Program generates correct output, algorithms implemented as specified, handles invalid inputs appropriately.
- **Report** [3 marks]: Accurate responses which are clear and relevant.

Submission

Submit your project by **10AM Monday 17 October**, using `submit comp20003 proj2`. Remember to submit all your source and header files, your Makefile and your report(report.txt). Check that the submission was successful with the `verify` command. The provided `Makefile` contains targets to help you with this.

The lateness policy is on the handout provided at the first lecture and also available on the subject LMS page. If you cannot submit on time you should contact the tutor in charge or lecturer at the soonest possible opportunity (this generally means **before** the deadline). Late submission will be done using the command submit comp20003 proj2.

As suggested before, you should check out your repository from the root: /home/studproj/student repositories/xyz/ where xyz is your login. If you choose to do this from a remote location (via an ssh connection) a command something like: svn+ssh://xyz@cat.cs.mu.oz.au/home/studproj/student repositories/xyz/ should work for you (please spread the load across the CSSE machines). Within that directory is the structure

trunk/comp20003/proj2

You should do your work in the directory proj1 and check your work into the repository regularly.