CSC 310, Fall 2011 — Practical Assignment #1

Due in class on November 21. Worth 10% of the course grade.

Note that this assignment is to be done by each student individually. You may discuss it in general terms with other students, but the work you hand in should be your own.

In this assignment, you will first implement the LZ77 compression method in something close to its original form. You will then see whether it can be improved by using arithmetic coding — first with equal probabilities, then with a simple adaptive model, and finally with additional improvements that you work out.

All programs should be for compressing a file in which the symbols are standard 8-bit bytes (so the alphabet is of size 256), with the compressed file also being a sequence of 8-bit bytes. The input should be read from "standard input", and the output written to "standard output" (though you may change this if you need to for some reason). Routines in C for arithmetic coding and for simple adaptive model updates are provided, along with simple compression/decompression programs that use these routines with an order-0 model, which you can use as a point of comparison. It is probably most convenient for you to write your programs in C too, but you can use another language if you can figure out how to interface it to these routines.

Four files (three of English/Latex text, one an object file stripped of symbols) are provided for testing compression performance.

You should hand in listings of your programs, along with the sizes (in bytes) that they compress the test files to, and discussion of the results as requested below. This is not primarily a programming or data structures assignment, so marking will not be focused on these aspects of your work. However, marks may be deducted for exceptionally disorganized or otherwise incomprehensible programs, or for programs that are extraordinarily inefficient. Marks will not be deducted just for using simple but non-ridiculous methods like linear search. Put another way, you won't get more marks for using more efficient methods like hash tables. Part D, which is somewhat open-ended, is an exception — there, using a clever, fast algorithm might contribute to getting full marks, though there are other ways too.

Part A (40 marks):

As described in the lecture slides, in the LZ77 algorithm, the encoded data consists of a sequence of triples, each containing the length of a string in previous text that matches some following input text, the position of this matched string (irrelevent if the matched length is zero), and the input byte that follows the matching input. In this part of the assignment, you should implement *exactly* the variation of this algorithm described below.

For this implementation, the length of a matched string will be from 0 to 4. The position of the matched string will be identified by the position of its *last* byte, counting backwards from the next input byte, to a maximum of 63 bytes back. The program therefore needs to save the previous 66 bytes, since a match of length 4 ending 63 bytes back will start 66 bytes back.

The length of the match and its position are encoded in one 8-bit byte. Six bits are used to encode the position of the match, in ordinary binary notation. A position of zero indicates that the length of the match is zero. If the position is not zero, the length of the match is encoded in two bits, with 00, 01, 10, and 11 representing lengths of 1, 2, 3, and 4. Each 8-bit byte encoding the length and position of a match is followed by an 8-bit byte containing the symbol following the match, except if there is no symbol following the match, because the input file ends at that point.

The restriction to matches ending only up to 63 bytes allows the length and position of a match to be send as one byte, and also allows simple search methods to be used without the program becoming too slow. Allowing matches further back might well improve compression performance, but you should not try that for this assignment.

Report compression performance on the four test files with this program, and compare to the performance of the order-0 model. Briefly discuss the results.

Part B (20 marks):

In this part, you should modify your program from Part A to use encode the match length, position of match, following character, and end-of-file indicators using arithmetic coding. The arithmetic encoding/decoding routines take as an argument a table of cumulative frequencies for symbols, which determine their probabilities. The arithmetic coding routines assume that symbols are numbered starting at 1. The cumulative frequency table starts at index 0, with its value at index i begin the sum of frequencies for symbols with numbers above i, always ending with a 0 at the index of the last symbol.

In Part A, the decoder could determine the end of the the input file based on the end of the compressed file. Arithmetic coding doesn't encode each symbol in a whole number of bytes, however, so an explicit indication of the end of the input file needs to be encoded. You should do this by setting up a cumulative frequency table for a two-symbol alphabet — EOF and NOT_EOF — with the frequency of EOF being 1 and the frequency of NOT_EOF being 9999 (ie, the probability of EOF is taken to be 1/10000). You can encode EOF or NOT_EOF using this table whenever a flag of this sort needs to be encoded.

You should encode the match length using a table of cumulative frequencies in which the frequencies of 0, 1, 2, 3, and 4 are all one — ie, they are equally probable. If the length of the match is 0, you should not encode the position of the match at all (the decoder will also know that no position is encoded). If the length is non-zero, you should encode the position of the last byte of the match (1 to 63) using a table of cumulative frequencies in which all posiitions have frequency one. You should encode the character following the match using a cumulative frequency table in which all 256 possible 8-bit bytes have frequency 1.

Report compression performance on the four test files with this program, and compare to the performance of the LZ77 program from Part A. Discuss the results, explaining the source any difference you see.

Part C (20 marks):

In this part, you should modify your program for Part B to update the cumulative frequency tables for the length of a match, position of a match, and following byte, adding one to the frequency of a symbol each time it is seen (scaling them down as necessary to keep them in range). This can be done by calling the update routine provided. You should keep the table for encoding EOF or NOT_EOF fixed as in Part B.

Report compression performance on the four test files with this program, and compare to the performance of the programs from Parts A and B, and the order-0 model. Discuss the results, including discussion of which frequency table most benefits from being adaptive (which will require some additional work to separate these effects).

Part D (20 marks):

This part is somewhat open-ended. You should consider ways of improving performance, but stick with the limits that the position matched must end no more than 63 characters back, and that the longest match is of length 4.

One possible direction for improvement is to consider whether some symbols for which code space is allowed are actually never transmitted. If so, the code space allocated to these symbols could be omitted, allowing the possible symbols to be transmitted in fewer bits.

Another possible direction for improvement is to consider whether or not a following byte should always be transmitted after a match, or whether this should happen only for some matches.

You can also consider improving time efficiency. Relatively little credit will be given for improving efficiency of the programs for Parts A, B, or C, but the time efficiency of any improvement you make for this part will be a consideration, when an efficient implementation is not obvious.