190004947
Computer Science
University of St. Andrews

CS3302 Practical 2 Report

Data Compression with Arithmetic Coding

I used Java to implement the arithmetic coding algorithm in two files aenc.java and adec.java which code and decode respectively. They share functions but instead of keeping the shared functions in a helper class I kept a copy of the functions in each program to help me follow in debugging. Both programs are called from the command line with two text files as parameters for the input and output (e.g. java aenc input.txt output.txt). I did not create executables for aenc and adec because my implementation failed tests with messages greater than size 2.

The main function takes these parameters in and parses the text file to an array of elements separated by whitespace. No legal source alphabet can have less than 3 elements in the array so 'Bad Source Input' is written to the output file if the array is too small. This will also be output if there is no 'end' symbol indicating the end of the source alphabet. After the input array has been filled properly I call a function 'encode'/'decode'. This function is preparation for the arithmetic coding algorithms. It takes in the string array of elements from the input file and through various means it will instantiate three useful variables: dict, tokens, and chars. The first of which is a dictionary or hash table containing the characters as keys and their relative probabilities as values. This lookup table will be used in the arithmetic coding and decoding to get the probabilities of different characters when calculating the scale. The next useful variable is tokens which stores all the characters after the 'end' symbol which are the characters that need to be encoded/decoded. Once these characters have been added to the tokens array, an extra character is added at the end; §. This is a random character which I decided to use to represent the end of message symbol. This symbol is also added on the top of the dictionary as a source alphabet character with a relative probability of one. The last of the three useful variables is the chars array. It is (misleadingly) a string array which will contain each of the characters in the source array but ordered in terms of their ASCII value as required by the project specification. To order the characters I wrote a function called orderSpec to take in an array of characters and order them. The order spec function uses bubble sort to arrange the characters which during development would sort an array of probabilities too making all the same swaps as the chars array so that the indexes would remain aligned. After the ordered spec is returned the cumulative probability is checked to match the expected cumulative probability; if they are different then 'Bad Source Input' will be outputted. The arithmetic function is then called with the three useful variables just described.

This function contains the logic behind the arithmetic coding and decoding algorithms and is where the aenc and adec programs differ. In the encoding program, the function starts off by initialising some variables for use in the integer calculations and for storing different parts of the tag that will be built. It then loops through the ordered characters calculating the cumulative probabilities and creating 'Interval' objects for each one. Interval is a an object I built in a file called Interval.java which has three parameters: character, upper bound, and lower bound. The scale of characters is then represented as an array of these Interval objects such that the upper bound of each character is equal to the lower bound of the character 'above' it. Once each source character and the cumulative probabilities have been added to the scale, the function will begin looping through each character in the message to be encoded. For each message character, first the index of that character in the ordered character array is found and stored to be used as an index in the scale to indicate which interval needs to be expanded. After this, the upper and lower bounds are updated to represent the upper and lower cumulative probability bounds of the character's position in the scale; essentially 'zooming in' on it.

Once the bounds have been updated, they must be rescaled to avoid underflow and overflow of the 32 bits. Rescaling will continue until the rescaling conditions are no longer met. There are two rescaling conditions that lead to two different rescaling operations. If the lower bound is greater than half the 32 bit range then you're expanding a character in the top half of the scale so a 1 and s * 0 are added to the tag and the appropriate rescaling operations will be applied. Similarly, if the upper bound is less than half the 32 bit range then you're expanding a character in the lower half of the scale so a 0 and s * 1 is added to the tag and, again, the appropriate rescaling operations will be applied. The last of the rescaling conditions is when the lower bound is greater than one quarter of the range and the upper bound is less than three quarters of the range. In this case no bits are sent but the upper and lower bounds are expanded and the rescaling counter, s, is incremented by one.

Once neither rescale condition is met, the loop breaks and the next message character is passed in. Once all message characters have been expanded, we send the most significant bit of the lower bound and then send s * the complement of the MSB. After these bits have been sent, we append the last 31 bits of the lower bound to the tag and convert the tag to hexadecimal from least significant bit to most significant bit. Finally, the message has been encoded and the hex tag is output to the encoding function which outputs to the main function which writes the tag to the output text file.

The arithmetic function in the decoder is a little different. It begins by storing the hexadecimal tag in binary representation using a converter function I wrote called HexToBinaryTag. Another variable is initialised, v, which stores the first 32 bits of the binary tag in integer form (in a long object). It then fills up the scale with Interval objects like in the encoding algorithm and then starts a loop for each character in the source alphabet. Next, the upper and lower bounds are updated and stored in separate variables denoted by *l'* and *u'* in the lecture notes. The equations for these new bound are identical to those in the encoding algorithm. After the range is updated, if v lies between the expanded bounds, that means during encoding that was the character 'zoomed into' during that round. It it critical that the bounds are updated the same way during encoding and decoding. If the character found within the range is not the end of message symbol then the character is added to the output string. After this check, the lower and upper bound variables are set as the updated bounds from expanding the discovered character. Lastly before the rescaling operations, there is a check if the character found is the end of message symbol, in which case the current output tag is returned to be written to the output file.

The rescaling operations in decoding are the same as the ones in encoding with the addition of rescaling the v variable. Each time the upper and lower bounds are updated, the buffer store of the tag is updated the same way. After these values have been updated, the next bit of the tag is added to v. This process is repeated until no rescaling conditions are met. The decoder will find each interval to expand at each stage until it expands the end of message symbol at which point it will return the decoded message.