CS358: Applied Algorithms

Assignment 5: Compression (due Saturday, 11/15 at 10PM)

Instructor: Sam McCauley

Instructions

There are two parts to the assignment: code, and a pdf submission.

Code can be submitted by committing and pushing them to gitlab. I strongly suggest that you access `evolene.cs.williams.edu` through a web browser to make sure everything was uploaded as you expected. You may collaborate on code with your classmates, as well as the instructor and TA, but you may not use any LLM assistance.

The pdf should be uploaded to Gradescope for submission and feedback. The pdf should be entirely your own work; you should only use "hands in pockets" discussions with your classmates and the TA; you should not use any online (including LLM) assistance.

Please contact me at srm2@williams.edu if you have any questions or find any problems with the assignment materials.

Problem Description

In this problem, we will be creating a program that can compress (and decompress) a given data file.

To compress, the program is given the name of a file. The program compresses the file, and outputs a new file that has the same filename with `.code` appended.

To decompress, the program uses the flag `-d`. The program compresses the `.code` file, and outputs a new file with a `.decode` extension.

Please note that there is a constant, `NUM_CHARS`, storing 256, the number of ASCII characters. You can use it throughout your implementation (it should be accessible from any file).

Bear in mind while going through this project: while there is quite a lot of starter code to make all of this work, you only need to implement the two functions in `mtf.c`, and the two functions in `bwt.c`.

Step 0: Suffix Array Implementation. Check to make sure that your suffix array implementation from last week is in good working order. You should remove any debugging output so that it does not get in your way as you implement the BWT.

> **Implementation 1.** Check to make sure that your suffix array implementation in the `suffixarray/` folder is working properly.

Step 1: Check that Huffman Works. Huffman coding is already in your starter code. If you use the `-h` flag, you can test it. Try the following:

```
./test.out -h inputs/test.txt
```

It should work without issue, generating a file `test.txt.code` in the `inputs/` folder. (This file is larger than the original.) Then, we want to decode:

```
./test.out -hd inputs/test.txt.code
```

After this, there should be a file `test.txt.decode` in `inputs/`. This file should be identical to `test.txt`. We can test this as follows:

```
diff inputs/test.txt inputs/test.txt.decode
```

This program should display no output. You can do these tests with a larger file, like `inputs/wikiASCII.txt` as well—for that file, there should be some space savings. (This file is a text dump of a number of concatenated Wikipedia files; check it out! There are a few Williams College references in there.)

Step 2: Move-to-Front. Now, your goal is to implement the move-to-front transform we saw in class. There is a useful helper function `move_up_char(char* str, int index)` provided for you: this takes the array given by `str`, and moves the item at index `index` to the front, pushing the remaining elements down one slot each.

> **Implementation 2.** In `mtf.c`, implement the move-to-front algorithm in `move_to_front`; your function should store the move-to-front transform in the array `output` (this has already been allocated; you just need to fill in the values). Then, implement the decoding move-to-front algorithm in `dec_move_to_front`, storing the result in `decode`.

To test your move-to-front implementation, use the `-m` flag. You probably also want to use the `-v` flag, so that the program outputs the move to front transform it computes.

```
./test.out -mv inputs/test.txt
```

Be sure to also make sure that decoding works. First, do the following to decode:

```
./test.out -mvd inputs/test.txt.code
```

Then, ensure that the decoding gave the same file (this should output nothing if it worked):

```
diff inputs/test.txt inputs/test.txt.decode
```

You can try other test files as well. Once you get this working, try it on a larger file:

```
./test.out -mv inputs/chrom1ln.txt
./test.out -mvd inputs/chrom1ln.txt.code
diff inputs/chrom1ln.txt inputs/chrom1ln.txt.decode
```

And finally, turn debugging information off (no `-d` flag) and check that it works on a large file.

```
./test.out -m inputs/wikiASCII.txt
./test.out -md inputs/wikiASCII.txt.code
diff inputs/wikiASCII.txt inputs/wikiASCII.txt.decode
```

Step 3: Burrows-Wheeler Transform. Now, your goal is to implement the Burrows-Wheeler transform we saw in class.

As we saw, the best way to implement the Burrows-Wheeler transform is with a suffix array. The executable will call *your code* (in the `suffixarray/` folder of your repo, where you worked last week) to do this.

> **Implementation 3.** Implement the Burrows-Wheeler transform `bwt()` in `bwt.c`, and the inverse of the transform in `bwtDec()`. Note that `bwt()` is passed the suffix array from your code as input, and the result array `res` has already been `malloc`'d: you just need to fill `res` with the correct values.

To test your implementation as you do it, you don't need to use any flag to swpecify using the BWT (Burrows-Wheeler transform is called by default). When debugging, use the `-v` flag, so that the program outputs the BWT it computes.

```
./test.out -v inputs/test.txt
```

Be sure to also make sure that decoding works. First, do the following to decode:

```
./test.out -vd inputs/test.txt.code
```

Then, ensure that the decoding gave the same file (this should output nothing):

```
diff inputs/test.txt inputs/test.txt.decode
```

You can try other test files as well. Once you get this working, try it on a larger file:

```
./test.out -v inputs/chrom1ln.txt
./test.out -vd inputs/chrom1ln.txt.code
diff inputs/chrom1ln.txt inputs/chrom1ln.txt.decode
```

And finally, turn debugging information off and check that it works on a large file. Encoding this file likely takes **4–8 minutes** to complete, so be patient! (Decoding is likely quite quick, however.)

```
./test.out  inputs/wikiASCII.txt
./test.out -d inputs/wikiASCII.txt.code
diff inputs/wikiASCII.txt inputs/wikiASCII.txt.decode
```

Questions

Problem 1. A natural question you may ask is why we double prefixes. What happens if we grow the prefixes by a factor of c each time for some constant $c \geq 2$?

In this algorithm, the base case remains the same: we sort prefixes by the first letter. But now, we sort strings by the first c "classes," rather than the first 2.

In terms of c and n, how long does this algorithm take to find the suffix array of a text of length n?

Hint: you want to proceed in two parts. First, how long does radix sort take to sort items with c characters? Then, how many times does the outer loop run?

Solution. □

Problem 2. There are a small number of strings that do not change when applying the Burrows-Wheeler transform. For example, $BWT(\texttt{aaaa\$}) = \texttt{aaaa\$}$), and $BWT(\texttt{bbab\$}) = \texttt{bbab}$. These are called **fixed points** of the Burrows-Wheeler Transform.

1. Prove that for any string s that is a fixed point of the BWT, the first and last characters of s (last meaning the character immediately before the \$) are the same.

   Hint: look at the first "row" of the matrix of sorted suffixes we used to create the BWT.

2. Prove that for any string s that is a fixed point of the BWT, the first character of s must be the last character of s in alphabetical order. (So for example, if s starts with a `c`, every character of s must be equal to or before `c` in alphabetical order.)

   Hint: look at the last "row" of the matrix of sorted suffixes we used to create the BWT.

Solution. □