School of
Engineering and Computer Science
Te Kura Mātai Pūkaha, Pūrorohiko

COMP 261 (2019) - Assignment 5

Goal

In this assignment you will implement two string search and two compression algorithms which have been discussed in lectures, and explore their performance.

The assignment has four parts, worth approx. 30%, 30%, 30% and 10% (see the marksheet linked below for a more detailed breakdown of marks). Each part has some programming, and some questions to be answered in your report.

Resources

- starter code: code.zip
- the data: data.zip
- mark sheet: Marking schedule (from the ECS marking system). *NB!*Please read this, as it lists the marks for each part of the assignment.

To Submit

- All the source code for your programs, including the template code. **Please make sure you do this** - without it your assignment cannot be marked.
- Any other files your program needs to run that *aren't* the data files provided.
- A report which answers the questions asked in this handout. Note that a significant portion of the marks are for this report. Your report should be submitted as a `txt` or `pdf` file. No fancy formatting of the report is required.

Please **don't** submit the data files, as they are quite large.

The submission link can be found on the left.

Note that you will need to sign up for a 15 minute slot with the markers.

Handout code

Several data files have been provided to search and compress.

- **War and Peace.** A novel by Tolstoy that is often used as sample text in machine learning. It is roughly 3 MB.
- **Taisho.** A ninth century dictionary from the Chinese Buddhist Canon, a collection of texts written in Classical Chinese. It uses a very large alphabet and is about 3 MB in size.
- **Pi.** The first million digits of Pi, totalling about 1 MB.
- **Lenna.** A hexdump of the famous image of Lenna, often used as an example in image processing. It is small, only 300 kB. This makes it good for quickly testing your algorithms.
- **Apollo.** A text version of the ``the eagle has landed'' sound recording, from Apollo 11. There are two channels, and hence two numbers at each time step. It is about 6 MB.

Part 1: String Search

In this section, your task is to implement the brute force and KMP string search algorithms to enable searching in the text editor.

A method stub is provided in the `Search` class which you should fill in. The `search` method takes two arguments: the text to search through and the string to search for. The method returns an integer as follows:

- The starting index of the first match in the text if one exists.
- -1 if no match exists.

You should implement two versions of this method, one using brute force search and one using the KMP search algorithm.

Note that KMP has two stages: computing the match table for input string, and performing the string search itself. You should write a separate method for each of these.

Once you have both algorithms implemented, experiment with the provided files to see what differences you can observe in their performance. To do this, you can add code to measure the time taken, or to count the number of steps taken, or both.

By default, the provided code tries to load the War and Peace code initially, assuming that you have the data files in a directory called `data`. You may change this behaviour and/or location if you wish.

Question 1: Write a short summary of the performance you observed using the two search algorithms.

Part 2: Huffman coding

Your task in this part is to implement the Huffman coding and decoding algorithm, as described in lectures, and answer Questions 2 and 3.

A full implementation does three things:

- Create a tree of binary codes for each character in the input text.
- Encode input text using that tree.
- Decode previously encoded text with that tree.

You will need to write methods to do all three of these steps for a particular text. Remember, you need to dynamically generate the tree from a given input text, and *not* use a fixed tree that you supply manually.

Here are some implementation notes:

- The `HuffmanCoding` class has three methods that you should fill in.
- The `encode` method should return a *binary string*, a string containing only `1` 's and `0` 's. Similarly, `decode` takes a binary string as its argument.
- You could store the binary codes for each character in a `Map<Character, String>`.
- To debug your code, you could manually make an encoding tree and then generate a text using its frequencies.

Question 2: Report the binary tree of codes your algorithm generates, and the final size of *War and Peace* after Huffman coding.

Question 3: Consider the Huffman coding of `war\_and\_peace.txt`, `taisho.txt`, and `pi.txt`. Which of these achieves the best compression, i.e. the best reduction in size? What makes some of the encodings better than others?

Part 3: Lempel-Ziv compression

In this part, your task is to implement the Lempel-Ziv 77 compression and decompression algorithms, as described in lecture, and answer Questions 4 and 5.

Implementation notes:

- The `LempelZiv` class has two methods you should fill in.
- None of the provided data files include the characters `[`, `]`, or `|`. This means you can use them to start, end, and delimit your tuples respectively.
- To debug your code, you could make some small files containing carefully constructed strings (all one character, one repetition, etc.) and check you get the expected result.

Question 4: The Lempel-Ziv algorithm has a parameter: the size of the sliding window. On a text of your choice, how does changing the window size affect the quality of the compression?

Question 5: What happens if you Huffman encode War and Peace *before* applying Lempel-Ziv compression to it? Do you get a smaller file size (in characters) overall?

Part 4: Challenge --- Better string search or coding

You have a choice for the challenge, pick **ONE** of the following.

- **Better coding.** Implement a more sophisticated coding scheme. Several of these exist, but two good candidates are:
  - *Arithmetic coding*. As described in lecture, this takes a different approach to coding, where strings are encoded into a single floating point number between 0 and 1.
  - *Adaptive Huffman coding*. This uses a coding tree that changes throughout the text to get better performance in the case where the distribution of characters changes throughout the text.

- **Boyer-Moore.** Implement the Boyer-Moore string search algorithm, which is faster but more complex than KMP. It improves efficiency by doing a more complex search (building two tables, not just one) and starting from the end of a pattern instead of the start.

Lastly, there are a few bonus marks available for an interesting question about coding.

Question 6: Suppose Alice has two binary strings (made only of `1` 's and `0` 's), `X` and `Y` . Make a pair of algorithms so that:

- Alice can encode `X` and `Y` into a single binary string `Z` , which she sends to Bob.
- Bob receives `Z` and can **unambiguously** decode it back into `X` and `Y` .

For example, a simple solution would be to concatenate `X` and `Y` together. This works if `X = 101` and `Y = 011` . Bob then receives `Z = 101011` and splits it in half to retrieve `X` and `Y` . But what if `X = 10` and `Y = 1011` ? The goal is to craft your encoding and decoding algorithms so that `Z` is as short as possible. Start by making algorithms so that `|Z| = 2*(|X| + |Y|)` . You get the marks for this question if you find an algorithm such that `|Z|` is significantly less than that.

Writing it yourself

Make sure that you write the code for the data structures yourself -- you will not learn what you need to learn if you use code from somewhere else. You can build on code examples from somewhere else, but do not simply copy large segments of code, and make sure that you acknowledge the source appropriately. If we identify any plagiarism, we *will* penalise it!