String search and compression

In this assignment you will explore a string search and two compression algorithms.

Resources

The assignment webpage also contains:

- An archive of the template code and data.
- A provisional marksheet.

Please read the marking schedule, as it lists the marks for each part of the assignment.

To Submit

You should submit these things:

- All the source code for your program, including the template code. **Please make sure you do this**, without it we cannot give you any marks.
- A report on your assignment that answers the questions asked in this handout. **A significant portion of the marks are for the report**.
- Any other files your program needs to run that *aren't* the data files provided.

No fancy formatting of the report is necessary, and it should be submitted as a `txt` or `pdf` file. Please **do not** submit the provided data files, as they are quite large.

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

Handout code

The handout code is a small GUI with two panes. The first is a very small text editor with one interesting function: string search. This will call the `KMP.search` method that you need to fill in. The second pane provides a GUI for selecting a file to compress with either Huffman coding or Lempel-Ziv compression. You're free to change or ignore the handout code, but there's not much reason to.

Handout data

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.

Minimum: KMP search

In this section, your task is to implement the KMP substring search algorithm to enable searching in the text editor.

A method stub is provided in the `KMP.search` which you should fill in. This method returns an integer as follows:

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

The `search` method takes two arguments: the text to search through the and substring to search for. KMP has two stages: first, you should compute the match table for input substring. Second, you should use that match table to perform the substring search itself. I would recommend having a separate method for each of these.

Core: Huffman coding

Your task in this section is to implement the Huffman coding and decoding algorithm, as described in lectures, and answer questions 1 and 2.

A full implementation does three things:

- Creates a tree of binary codes for each character in the input text.

- Encodes input text using that tree.
- Decodes 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 1: Report the binary tree of codes your algorithm generates, and the final size of War and Peace after Huffman coding.

Question 2: Consider the Huffman coding of `war_and_peace.txt`, `taisho.txt`, and `pi.txt`. Which of these achieves the best encoding, i.e. the best reduction in size? What makes some of the encodings better than others?

Completion: Lempel-Ziv compression

In this section, your task is to implement the Lempel-Ziv 77 compression and decompression algorithm, as described in lectures, and answer questions 3 and 4.

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 informative strings (all one character, one repitition, etc.) and check you get the expected result.

Question 3: 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 4: 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?

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:

- *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.
- *Arithmetic coding.* This takes a different approach to coding, where strings are encoded into a single floating point number between 0 and 1.

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 starting from the end of a pattern instead of the start.

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

Question 5: 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|)` or (where `|Q|` is the length of string `Q`).

You get the marks for this question if you find an algorithm such that `|Z|` is significantly less than `2*(|X| + |Y|)`.

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.