Lab 4

27 Sep 2007

Today we continue our work on compression algorithms. In particular, you will implement a simple version of the sliding-window algorithm, and then you will modify some of its parameters and determine how they improve the system. I suggest you first skim section 1, then read sections 2 and 3 carefully before going back and using section 1 as a reference for the actual lab work.

1 The basic algorithm

This section describes the baseline form of the algorithm you'll be considering. It is a formalisation of what we discussed in class yesterday.

Compression format

Every chunk in the compressed file is either a null (zero) byte followed by a literal character, or an offset byte (with value 1–255) followed by a length byte (with value 0–255; 0 is legal but pointless).

Encoding algorithm

There are two buffers: one, the buffer of "bytes already encoded", should start out with 255 null bytes in it. The other, the buffer of "bytes yet to be encoded", should start out empty.

First, read bytes from the input stream and buffer them until the input stream is empty or the input buffer is full (255 bytes).

Then: find the longest prefix of the input buffer that occurs in the already-encoded buffer, if any. (An easy way to do this is to see if the first character appears, then see if the first two characters appear, and so on. This is $O(n^2)$ in time efficiency, but $n \leq 255$, you can `break` as soon as a search fails, and time efficiency is not what we're measuring anyway. So it's ok for our purposes.)

If there *was* a longest prefix, encode it as an offset-length pair and write it out, and then transfer the entire prefix from the input buffer to the already-encoded buffer.

If there was *no* prefix, meaning the next character in the input buffer doesn't appear anywhere in the sliding window, encode the first byte of the input buffer as a literal, write it out, and then move it to the already-encoded buffer.

Either way, make sure to remove the beginning of the already-encoded buffer, returning it to length 255, after you've added stuff to the end of it.

Repeat until input buffer is empty.

Decoding algorithm

Here you only need one buffer: the buffer of "bytes already decoded". It starts out with 255 null characters.

First, read an encoded chunk from the input. If it represents a literal, print that literal out and add it to the decoded buffer.

If it represents an offset o and a length ℓ, then write a loop that runs ℓ times, each time appending the byte at position $255 - o$ to the end of the buffer and removing the first byte in the buffer. When this is done, print out the last ℓ characters of the decoded buffer.

2 Your program

The actual task you are performing requires you to understand the compression format and the encoding and decoding algorithms, but your implementation won't be exactly like either. Your primary goal here is to evaluate how efficiently (with respect to space) the algorithm works, and whether your ideas to improve it actually improve it. As such, you can skip the decoder entirely.

The *important* output from your program will come at the end of the encoding process: after "encoding" is complete, print the number of bytes in the original, the number of bytes in the compressed version, and the compression rate (i.e. how much smaller is the compressed version, as a percent of the original version). For instance, running on the `castle` input, my implementation prints

```
Original length 2696; Compressed length 986; compressed 63.43%
```

Perhaps more importantly, you don't even have to actually write out the encoded version of the file. For debugging purposes, I recommend implementing an alternate output format, with one line per chunk, "encoded" as either something like

```
(lit 'A')
```

or

```
(Off 5, Len 2)
```

This will be much easier to check visually (not to mention to prepare a "correct answer" to diff against), but it is important to remember that the "number of bytes" that you report at the end of the program must be with respect to the true encoded format—that is, two bytes per chunk, at least in the baseline format.

Combining the intuitiveness of "encoding a picture" with the convenience of operating on characters, I've grabbed a few ASCII art images[1] and placed them in the `lab4` subdirectory. These will be what you use to test the compression algorithm. There is a README there as well; it tells you what each test case is good for.

3 Implementation hints

To actually read in data, use a `Reader`, which reads a single character at a time. In particular, you'll want to make a

```
new InputStreamReader(System.in)
```

and read from that.[2] Every `Reader` has a `read()` method; it returns a single character, though it returns it as an `int` rather than `char` or `Character`—this is partially a legacy of the early days of trying to integrate multi-byte

[1]From `chris.com`, though he assembled them from elsewhere and doesn't hold the copyright. I think this counts as fair use, though.

[2]For future reference, if you want character-by-character input from a file, you can use `new FileReader(filename)`.

characters into Java, and partially so they can just return -1 at the end of the stream. For *this* lab you can do the quick and dirty version, and assume all characters are one byte and can be cast into a `char`. Every `Reader` also has a `ready()` method which will probably prove useful to you.

Because the actual data you're handling here is text, Java's string processing libraries will be of use to you. Although `String`s are immutable, you can modify the contents of `StringBuffer` objects, and the following three methods will be of particular use to you:

```
StringBuffer.setLength(int)
StringBuffer.delete(int, int)
StringBuffer.append(String)
```

Although you are not *required* to use these methods, the most straightforward implementation does, so if you don't see why one of them would be useful, you should ask. Every `StringBuffer` also has `indexOf` and `substring` methods that work exactly like their `String` counterparts.

Some of the interactions with the `Reader`s can throw `IOException`s, which makes the compiler unhappy and yields a message like "unreported exception... must be caught or declared to be thrown". You already know how to handle this by actually catching the exception, but you can instead defer handling the exception until some higher-level method. To *declare* that your method may throw this exception, just say so in the method header:

```
public void meef () throws IOException {
```

That does mean that whoever calls `meef()` now needs to worry about this, but it's possible that that method can handle the exception more cleanly. (For instance, you can put one try/catch in `main`, and then your algorithmic methods can be simple and concise.)

4 Experimental labwork

Once you've debugged your code and are convinced it's giving you good numbers, record (write down, type into a file) its performance on all the provided input sets.

Then, start modifying the algorithm in ways you think will improve the space efficiency. Write down the changed performance numbers and, and this is very important, *exactly what you did to change it.* Whether you're writing on paper or in a file, this is your lab notebook, and for your work to be meaningful, you need to be able to convey what exactly improved (or hurt) the performance.

Some suggested routes of improvement:

- The baseline algorithm will never give you a length greater than your offset, which as we discussed in class, can sometimes help quite a bit.
- Instead of using a whole byte to flag a literal, what would happen if there were a single bit flag ("literal or offset-length pair?") that preceded every chunk?
- Instead of using a whole byte to represent a literal, what would happen if you indexed them in advance and used the index (which might have fewer than eight bits)?
- What happens if you vary the number of bits allocated to the offset and/or the length?

Note that many of these improvements involve handling bits in groupings smaller than a byte. Since you're not writing a decoder, and your encoder doesn't need to actually produce its output (just count the bytes in what *would* be output), all you need to do is track bit counts instead and report those numbers. Handling bit streams is a pain, so fortunately, you don't actually have to worry about them.