Sliding Window Dictionary Encoding

Sigtryggur Kjartansson

April 28, 2015

Contents

1 Algorithm Overview **1**
1.1 When to Encode 1
1.2 Encoding Steps 2
1.3 Decoding Steps 2

2 Design and Implementation **2**
2.1 Which Programming Language 2
2.2 Dictionary Representation 2
2.3 Find Matches in Dictionary 3
2.4 How to Denote Offset 3
2.5 Bit-Packing vs. Wasting Bits 4
2.6 Length & Offset Optimizations 4

3 Further Discussion **4**
3.1 Improvements 4
3.2 Extensions 4

1 Algorithm Overview

This compression scheme compresses/decompresses an input file according to the following sliding window dictionary encoding of the last `WINDOW_SIZE` symbols encoded/decoded:[1].

unencoded "byte"	1-bit flag = 0 8-bit chat
encoded "copy byte"	1-bit flag = 1 `OFFSET_LEN`-bit offset (here 16) `NBYTES_LEN`-bit length (here 6)

Table 1: Encoding Scheme

The algorithm tries to match as many unencoded symbols to symbols in the dictionary. The number of bits we choose for each affects the runtime of

[1]My implementation is generic, so you can set `OFFSET_LEN` and `NBYTES_LEN` to any "reasonable" values (i.e. `len(copy byte) > len(byte)` and no guarantees on when `NBYTES_LEN> OFFSET_LEN`, although it has worked for couple of test values)

compression as well as the compression ratio we achieve. With `OFFSET_LEN` bits for offsets we can encode $2^{\texttt{OFFSET\_LEN}}$ different offsets and likewise with `NBYTES_LEN` for lengths we can encode $2^{\texttt{NBYTES\_LEN}}$. This means that the size of dictionary is a function of $2^{\texttt{OFFSET\_LEN}}$ and the lengths of its values (length of matches) are a function of $2^{\texttt{NBYTES\_LEN}}$ (see Length & Offset Optimizations subsection for details).

1.1 When to Encode

It's not worth it to encode if the encoding takes more bits than writing out individual bytes. Here this minimum matching length is 3, generally this is given by

$$\texttt{MIN\_MATCH\_LEN} = \frac{\texttt{COPY\_BYTE\_LEN}}{\texttt{BYTE\_LEN}} + 1$$

1.2 Encoding Steps

1. Initialize dictionary
2. Initialize an array of uncoded chars of `MAX_MATCH_LEN`.
3. Search for longest match in dictionary.
4. (a) If match *length* > `MIN_MATCH_LEN`: encode string as copy byte
   (b) Else: stream 1 byte
5. Add streamed bytes to dictionary and read in more uncoded chars
6. Repeat from 3 until input has been encoded.

1.3 Decoding Steps

1. Initialize the dictionary
2. Read flag bit
3. (a) If 0: Read 1 char and write to output
   (b) Else: Read offset and length, and retrieve from dictionary and write to output
4. Add streamed bytes to dictionary
5. Repeat from 2 until input has been decoded

2 Design and Implementation

Here I highlight some the challenges and decisions I faced in design and implementation.

2.1 Which Programming Language

Because we're gonna be dealing with reading/writing at the bit level, for this reason I initially picked C, because it has pointers and efficiently handles bitwise arithmetic. I quickly realized that I would want to create some Objects for simplification and thus I picked C++. The language choice posed a few challenges in it's own, since I haven't programmed anything this extensive in C++.

2.2 Dictionary Representation

The dictionary is more like a sliding window of fixed length `WINDOW_SIZE`, so the most natural and straight-forward way to represent it is with a circular char array buffer. It allows $O(1)$ puts and let's us access k chars in $O(k)$ time.

2.3 Find Matches in Dictionary

The bottleneck of compression is finding the largest partial match at each step. I explored several different ways of searching for matches:

1. **Brute Force**
   Find the largest match for every possible starting positions and return the largest.
   Space: $O(\texttt{WINDOW\_SIZE})$, TIME: $O(\texttt{WINDOW\_SIZE} \cdot \texttt{MAX\_MATCH\_LEN})$
2. **Knuth-Morris-Prath**
   Uses a partial match table to skip some starting positions.
   Space: $O(\texttt{WINDOW\_SIZE}+\texttt{MAX\_MATCH\_LEN})$, Worst Case TIME: $O(\texttt{WINDOW\_SIZE}\cdot \texttt{MAX\_MATCH\_LEN})$, but the skipping makes it $O(\texttt{WINDOW\_SIZE}+\texttt{MAX\_MATCH\_LEN})$
3. **Boyer-Moore**
   Regarded as the most efficient practical string matching algorithm. Uses two shift rules + pre-proccessing to cleverly skip checks.
   Space: $O(\texttt{WINDOW\_SIZE}+\texttt{MAX\_MATCH\_LEN})$, Worst Case TIME: $O(\texttt{WINDOW\_SIZE}\cdot \texttt{MAX\_MATCH\_LEN})$, but the skipping makes it $O(\texttt{WINDOW\_SIZE}+\texttt{MAX\_MATCH\_LEN})$
4. **Balanced BST**
   Store Dictionary as a Balanced BST.
   Space: $O(\texttt{WINDOW\_SIZE})$, Time: $O(\log(\texttt{WINDOW\_SIZE}) + \texttt{MAX\_MATCH\_LEN})$

I implemented Brute Force for the MVP and so that I could have a bench mark for future improvements. Ultimately, I wanted to use either Boyer-Moore or BST, since they have the most attractive properties. I choice to do only KMP because:

a) Balanced BST requires implementing (or using existing) something like Splay Trees, AVL trees etc. with a modified search operation and I didn't have time to do so reliably.

b) Boyer-Moore is not easily extensible to handle partial prefix matches, because it matches the pattern in reverse order. It's possible, but too time-consuming!

I believe the the optimal choice is a BBST. Using KMP, I found, brought down the runtime by about a factor of 4.

Additionally, I also considered Layered Hash Tables and Linked Lists but found that the other methods were more viable.

2.4 How to Denote Offset

We're given `OFFSET_LEN` (here 16) bits to encode the offset. That means we can encode up to 2^{16} different offsets. The scheme presented suggests using negative offsets from the current positions. It's wasteful to store negative numbers, so we won't do that and it's more convenient to store the offset as an index into the current window. Hence, an offset of 0 means the copy byte starts at the oldest value in the dictionary.

2.5 Bit-Packing vs. Wasting Bits

The proposed compression scheme uses poor bit lengths for its encoding (see Improvements:Choose Different Encoding Lengths for an alternative approach with "better choices"). I had to choose between packing bits or wasting a couple of bits per encoding. I don't think it's acceptable for a compression scheme to waste 2 bits per encoding (we could have used those bits for longer lengths or larger windows), bit packing was the only option. I implemented efficient read and write classes for files (see BitReader and BitWriter)

2.6 Length & Offset Optimizations

The minimum length a valid match can be is `MIN_MATCH_LEN` (here 3), so a minor optimization is to store the length as *length* − `MIN_MATCH_LEN`. Similarly, the largest offset we can have is `MIN_MATCH_LEN` away from the current, so we can increase our window size by `MIN_MATCH_LEN`.

3 Further Discussion

3.1 Improvements

1. **Choose Different Encoding Lengths**
   If we chose the encoding lengths such that `OFFSET_LEN` + `NBYTES_LEN` was a multiple of 8, then we could pack 8 flags together in one byte and stream the rest as full bytes. This would allow us to avoid packing bytes across byte boundaries (which can be costly). This also suggests an easy improvement to I/0. In fact we can do this in the current implementation:

2. **Bundle Reads**
   Most of the code reads/writes 1 char at a time (with a couple of efficient exceptions). I could refactor the code to read several entities at once, by packing a few flags together (e.g. 8), counting the number of 1's set and reading in the needed bytes.

3. **Bundle Writes**
   Similarly, I could have the expand the write buffer to do larger writes less frequently.

4. **Use Balanced BST** Probably the biggest improvement that could be made is using a BBST for matching. This was discussed in above sections.

5. **Circular Buffer Class**
   A few places in the code use circular buffers. These could (and should) be wrapped up into a class for simplicity.

3.2 Extensions

1. **Compress `UTF-8`**
   The code could be extended to also encode `UTF-8`. We would have to either replace the "byte" encoding unit with a longer one. Or create a new encoding unit and extend the flag to three bits and use a special flag for symbols that require more than just one byte. The `findmatch` logic would also have to be updated to reflect these changes.