Implementation notes on A simple online adaptation of Lempel Ziv compression with Random access Support

Akashnil Dutta, Reut Levi, Dana Ron, Ronitt Rubinfeld

July 6, 2012

We present a Java implementation for the randomized algorithm formulated in [1]. To keep the code in context of the theoretical description, we hereby elaborate on some of the practical challenges we needed to solve in order to translate our work into the implementation. In the implementation we included minor changes from our theoretical description in order to deal with the case in which the total size of the input stream is unknown during the encoding, and also included some optimization for encoding information to improve the compression ratio. First we will describe the implementation choices made for the general LZ78 algorithm without random access and then those specific to our modification.

Choice of alphabet

For practical reasons of memory addressing and file formatting, we set our alphabet to consist of 256 characters corresponding to the byte encoding in files. Accordingly, each appended character b in the LZ78 phrases are 8-bits long. One issue with a large alphabet $\sum$ is that the memory allocation for each node in the trie may take up $\Theta(|\sum|)$ space to keep the children in an array. We solved the problem by using a hashmap instead, taking up space proportional to the number of children only.

Variable address size and phrase encoding

The output of the encoder is a sequence of phrases, each with pointer(s) to preceding phrases. We used the sequence id i of the i-th phrase to refer

to the i-th phrase and 0 to refer to the null phrase at the root. Hence the back pointer(s) at phrase i can be one of i possibilities. These numbers were encoded in binary with $\lceil \lg i \rceil$ bits and appended to the binary output stream. Finally the output stream is broken down to 8-bit words and written in the file as a sequence of bytes. To read the i-th bit in the stream we read the $\lfloor i/8 \rfloor$-th byte and find the corresponding bit from there.

Random access to read the i-th phrase

One notable property of our encoding is that there is no delimiter to indicate the start of the phrases or the number of bits used to encode addresses. However, our (Random access) decoding algorithm requires us to read the contents of the i-th phrase given the index i in constant time.

The decoder knows the address size to encode the i-th phrase. Hence it is sufficient to calculate the starting position $p(i)$ of phrase i in the bit stream, which is the sum of the lengths of all preceding phrases. To find that quickly, we first pre-compute $p[2^t]$ for all required values of t. Since all phrase lengths are constant from phrase 2^t to 2^{t+1}, we can obtain

$$p[2^t] = p[2^{t-1}] + (t+8) \cdot 2^{t-1}$$

This only needs to be computed once. To find $p(i)$ generally,

$$p(i) = p[2^{\lfloor \lg i \rfloor}] + (\lceil \lg i \rceil + 8) \cdot (i - 2^{\lfloor \lg i \rfloor})^1$$

This formula applies to the LZ78 algorithm without random access in which case every i-th phrase is $\lceil \lg i \rceil + 8$ bits long. We will subsequently describe how this extends to our random access algorithm with minor modification.

Selecting and encoding special phrases

In order to save space and improve efficiency, we also did not use any delimiters in the compressed stream to indicate which phrases are special. Instead we used a deterministic pseudo-random function to decide if phrase i is special. The seed for the generator is known to both the encoder and decoder in advance. Hence the encoder and decoder can agree on this information without any communication via the transmitted stream. If strong randomization

[1] In modern processors, there are usually ways to find 2^t in one CPU instruction and $\lfloor \lg i \rfloor$ in $\lg \lg i$ instructions, sufficient for all practical purposes.[2]

is required, the seed for the generator may be transmitted in the beginning of the compressed stream.

Additionally, for the addressing to work out, we need the cumulative number of special phrases up to some phrase i. To facilitate this calculation, we selected the special phrases ensuring that there is exactly one special phrase among the phrases $[ki+1, ki+2, \cdots, k(i+1)]$ where k is a constant, such that there are approximately one special phrase every k phrases. We used a pseudo-random function R which uses seed S to evaluate

$$R(i, S) \in \{ki+1, ki+2, \cdots, k(i+1)\}$$

This makes it easy to find the number of special phrases up to phrase i, which is either $\lfloor i/k \rfloor$ or $\lceil i/k \rceil$ depending on $R(S, \lceil i/k \rceil)$. We also know the number of bits used for encoding special phrase i, given i. Hence we can determine the cumulative number of bits used for encoding special phrases up to phrase i in an way analogous to the previous case. This lets us determine the starting position of i-th phrase in the output stream.

Bit encoding in special phrases

We also used variable number of bits to encode the special phrases. Since the maximum size of the position or the depth field for i-th phrase is not well known to the decoder, we had to use a large number of bits, $2\lceil \lg i \rceil$ and $\lceil \lg i \rceil$ respectively. Although these numbers may be much smaller, it is the theoretical maximum corresponding to the case when the input stream consists of continuous repeat of one character.

However, for addressing special-parent and special-ancestor we used less number of bits by addressing it as the i-th special phrase instead of as i-th phrase. This requires only $\lceil \lg(i/k) \rceil$ bits instead of $\lceil \lg i \rceil$ and is sufficient to locate the special phrase.

Implementation details of the TC-spanner

Apart from the addressing scheme, we needed a construction of the Transitive Closure spanner of the special phrase tree which does not require knowledge of the maximum depth to find special-ancestors at any stage.

Recall that in our theoretical model of the line-graph spanner, we had blocks of size $\lceil \lg n \rceil$ such that the i-th node in the block j contained a jump-pointer

to the i-th node in $(j - 2^i)$-th block. From an implementation perspective, we found one small optimization on this by changing this jump-pointer to the $(i - 1)$-th node in $(j - 2^i)$-th block. This works because in our back-tracking algorithm, every time a jump-pointer was used, the parent pointer was taken at least once immediately following it. (For the first node in a block we point to the last node two blocks before the current block to keep consistency because the decoder should never need to jump one block at a time twice.)

This also simplifies the back-tracking algorithm to the following:

1. If current depth is equal to desired depth, stop.
2. If depth of node in jump-pointer is more than desired depth, move to node at jump-pointer and repeat from step 1.
3. Else move to node at parent-pointer and repeat from step 1.

This algorithm will find the desired depth d' from a node at depth d in $O(\lg n)$ time. To see why that is true, notice that in each step in the process, it is guaranteed that the in-block index in the next node is always one less $(mod \lg n)$ than the in-block index in current node. Within $\lg n$ steps all $\lg n$ block indices will be visited in decreasing order. Within that sequence of steps, the jump pointers will also be matching the binary representation of the difference in block-indices between starting block and desired block. Following this modified algorithm improves the number of steps from $4 \lg n$ to $3 \lg n$ in the worst case since it takes at most $2 \lg n$ steps to make one decreasing pass through the in-block indices during which time the algorithm is guaranteed to navigate across the blocks to reach a node which is at most $\lg n$ depth away from the target node.

Now we will modify this for an online version. Essentially the idea is to have variable block sizes. We construct the spanner from depth 0 onwards one by one. For an index i following block j, if 2^i exceeds j, a new block is made starting at that index.

In this case, because of variable block sizes, identifying the i-th node in $(j - 2^i)$-th block is more difficult since the block sizes are variable. We use recursion to solve this problem. i.e. we reuse the jump-pointers already assigned to nodes at smaller depths.

Specifically, we find the depth of the $(j-2^{i-1})$-th block from the jump-pointer

in the $(i-1)$-th node in block j (i.e. node at depth $d-1$) and subsequently the depth of $((j-2^{i-1})-2^{i-1})$-th block from the jump pointer in the $(i-1)$-th node in block $(j-2^{i-1})$.

This still requires suitable assignment of base cases of induction and minor details. The full description can be found in function anc() in the implementation which computes the depth of the jump-pointer of node at depth d given d. This is stored in an array to efficiently compute new values using dynamic programming without having to recalculate values in the recursion. The following table shows the values of the function, which indicates the depth of the jump pointer from each node with the nodes grouped into blocks B_i of variable sizes.

- $B_1 \{ 1 \to 0$
- $B_2 \{ 2 \to 0$
- $B_3 \{ 3 \to 1$
- $B_4 \{ 4 \to 2$
- $B_5 \begin{cases} 5 \to 3 \\ 6 \to 3 \\ 7 \to 2 \end{cases}$
- $B_6 \begin{cases} 8 \to 4 \\ 9 \to 4 \\ 10 \to 3 \end{cases}$
- $B_7 \begin{cases} 11 \to 7 \\ 12 \to 5 \\ 13 \to 3 \end{cases}$
- $B_8 \begin{cases} 14 \to 10 \\ 15 \to 8 \\ 16 \to 4 \end{cases}$
- $B_9 \begin{cases} 17 \to 13 \\ 18 \to 11 \\ 19 \to 6 \\ 20 \to 2 \end{cases}$
- $B_{10} \begin{cases} 21 \to 16 \\ 22 \to 14 \\ 23 \to 9 \\ 24 \to 3 \end{cases}$
- $B_{11} \begin{cases} 25 \to 20 \\ 26 \to 17 \\ 27 \to 12 \\ 28 \to 3 \end{cases}$
- $\cdots$
- $B_{16} \begin{cases} 45 \to 40 \\ 46 \to 37 \\ 47 \to 30 \\ 48 \to 16 \end{cases}$
- $B_{17} \begin{cases} 45 \to 44 \\ 46 \to 41 \\ 47 \to 34 \\ 48 \to 19 \\ 49 \to 2 \end{cases}$
- $\cdots$
- $B_{88} \begin{cases} 475 \to 467 \\ 476 \to 461 \\ 477 \to 448 \\ 478 \to 421 \\ 479 \to 366 \\ 480 \to 265 \\ 481 \to 83 \end{cases}$
- $B_{89} \begin{cases} 482 \to 474 \\ 483 \to 468 \\ 484 \to 455 \\ 485 \to 428 \\ 486 \to 373 \\ 487 \to 271 \\ 488 \to 88 \end{cases}$

With this spanner structure, the back-tracking from node at depth 477 to that at depth 50 proceeds as follows:

Depth	477	448	433	425	424	217	122	81	60	59	53	52	51	50
Block number	88	84	82	80	80	48	32	24	20	20	18	18	18	18
In-block index	2	1	0	6	5	4	3	2	1	0	4	3	2	1

Table 1: Backtracking example

Notice that from depth 425 to depth 50, 61 block jumps were required. $61 = 111101_b$ in binary. Hence the jumps have been taken at indices $5, 4, 3, 2, 0$ respectively, corresponding to the 1 bit positions in 61.

[1] A. Dutta, R. Levi, D. Ron, R. Rubinfeld. A Simple online adaptation of Lempel Ziv Compression with random access Support. *Unpublished manuscript, 2012*

[2] S. E. Anderson. *Bit Twiddling Hacks* `http://graphics.stanford.edu/~seander/bithacks.html#IntegerLog`