EE4015 - IDP report
Compression and Decompression with random access in a streaming setting

Supervisor - Dr. Shashank Vatedka

Shaik Mastan Vali (EE18BTECH11039),
Kamparaju Srikanth (EE18BTECH11023)

Introduction

There has been a huge growth in the amount of data that is generated in the past decade. We are seeing an unprecedented surge in data from areas such as genomics, video data, virtual reality, etc. This data deluge has resulted in a need for more efficient compression algorithms for the storage and handling of the data. However, data storage is no longer for archival purposes. In modern applications, most of the stored data needs to be continually accessed and processed to make statistical decisions. This has motivated a new class of data compressors: efficient compressors that provide facilities to perform computation on the compressed data itself while being rate-efficient. This ability to work with the compressed data itself, without first decompressing the dataset, is particularly useful when the compression sizes are very high (say 100x). We already see examples of their utility in various problems. For example in the field of genomics, there has been considerable work on designing compressors that efficiently compress the 2D-tabular variant-call dataset while providing features to randomly decompress any row or column.

We are interested in storing a source sequence using a minimum number of bits while allowing a query of any position of the sequence by accessing only a few bits of the compressed sequence. Using traditional compressors such as Lempel-Ziv based algorithms, one needs to access the entire compressed sequence to recover a single symbol. This is undesirable because the space efficiency of these algorithms shows up when source sequences are very long. In other words, to query a single source symbol, traditional algorithms require accessing a number of stored bits that scale with the length of the sequence. So asymptotically, as the sequence length n tends to infinity, one needs to access a large number of stored bits (scaling with n) to query a single bit. We are interested in constructions that allow close-to-optimal compression with constant random access.

Problem Setup

A sequence X^n is generated from a stationary and ergodic source with alphabet $\mathcal{X}$. We would like to losslessly store X^n using the minimum expected number of bits such that every symbol X_i, i = 1,...,n, can be reliably reconstructed by accessing no more than to of the stored bits in the memory. A compression scheme with blocklength n, random access $t^{(n)}$, and the rate $R^{(n)}$ is defined by an encoder:

$$f^{(n)} : \mathcal{X}^{(n)} \to \{0, 1\}^*$$

And decoder pairs ($S_i^{(n)}$, $g_i^{(n)}$):

$$S_i \subseteq N \ , \ |S_i^{(n)}| \ = \ t^{(n)},$$

$$g_i^{(n)} \ : \{0, 1\}^{t^{(n)}} \to \mathcal{X} \quad , \forall \ i \ = \ 1, ... , n$$

such that the following are true:

$$R^{(n)} \ = \ \frac{E\left[f^{(n)}(\boldsymbol{X}^n)\right]}{n},$$

$$g^{(n)}\left(\left\{f^{(n)}\left(X^{(n)}\right)\right\}_{S_i^{(n)}}\right) \ = \ X_i$$

Here, $\left\{f^{(n)}\left(X^n\right)\right\}_{S_i^{(n)}}$ denotes the subsequence of $f^{(n)}\left(X^n\right)$ that belongs to the positions in $S_i^{(n)}$. The pair $(R, \ t)$ is said to be achievable if there exists a sequence of compression schemes (indexed by n) such that

$$\lim_{n \to \infty} \ R^{(n)} \ \to \ R$$

$$\lim_{n \to \infty} \ t^{(n)} \ \to \ t$$

In this work, we propose compressors with finite random access for i.i.d. sources and $k-$order Markov sources with unknown underlying distributions. We also show the existence of universal compression schemes with finite random-access

---

Using Lempel-Ziv compression

We now discuss a popular class of techniques for source coding that are universally optimal (their asymptotic compression rate approaches the entropy rate of the source for any stationary ergodic source) and simple to implement. This class of algorithms is termed Lempel–Ziv, named after the authors of two seminal papers that describe the two basic algorithms that underlie this class. The algorithms could also be described as adaptive dictionary compression algorithms.

The idea of adaptive dictionary-based schemes was not explored until Ziv and Lempel wrote their papers in 1977 and 1978. The two papers describe two distinct versions of the algorithm. We refer to these versions as LZ77 or sliding window Lempel–Ziv and LZ78 or tree-structured Lempel–Ziv. (They are sometimes called LZ1 and LZ2, respectively.)

1) **Sliding Window Lempel–Ziv Algorithm (LZ77)**

We assume that we have a string $x_1, x_2, \ldots$ to be compressed from finite alphabets. A parsing S of a string $x_1, x_2, \ldots x_n$ is a division of the string into phrases, separated by commas. Let W be the length of the window. Then the algorithm can be described as follows: Assume that we have compressed the string until the time $i - 1$. Then to find the next phrase, find the largest k such that for some $j,\ i - 1 - W \le j \le i - 1$ the string of length k starting at x_j is equal to the string (of length k) starting at $x_i \left(i.e.,\ x_{j+1} = x_{i+1}\, for\, all\, 0 \le l < k\right)$ The next phrase is then of length k $\left(i.e.,\ x_i \ldots x_{i+k+1}\right)$and is represented by the pair $(P,\ L)$, where P is the location of the beginning of the match and L is the length of the match. If a match is not found in the window, the next character is sent uncompressed. To distinguish between these two cases, a flag bit is needed, and hence the phrases are of two types: $(F,\ P,\ L)$ or $(F,\ C)$, where C represents an uncompressed character,

For example, if $W = 4$and the string is $ABBABBABBBAABABA$ and the initial window is empty, the string will be parsed as follows: $A, B, B, ABBABB, BA, A, BA, BA$, which is represented by the sequence of "pointers": $(0, A), (0, B), (1, 1, 1,), (1, 3, 6), (1, 4, 6), (1, 1, 1), (1, 3, 2), (1, 2, 2)$, where the flag bit is 0 if there is no match and 1 if there is a match, and the location of the match is measured backward from the end of the window.

2) **Tree-Structured Lempel–Ziv Algorithms(LZ78)**

The source sequence is sequentially parsed into strings that have not appeared so far. For example, if the string is $ABBABBABBBAABABAA$...., we parse it as $A, B, BA, BB, AB, BBA, ABA, BAA$ .... After every comma, we look along the input sequence until we come to the shortest string that has not been marked off before. Since this is the shortest such string, all its prefixes must have occurred earlier. (Thus, we can build up a tree of these phrases.) In particular, the string consisting of all but the last bit of this string must have occurred earlier. We code this phrase by giving the location of the prefix and the value of the last symbol. Thus, the string above would be represented as $(0, A), (0, B), (2, A)(2, B), (1, B), (4, A), (5, A), (3, A)$ …

Data compression using Zstd

Zstd (or Zstandard) is a data compression library developed by Facebook. It is specially optimized for high compression and decompression speeds. Another interesting feature of this library is that it offers "dictionary compression," wherein you can train the algorithm with some files, producing a dictionary that needs to be fed to the compressor and decompressor. This gives improved compression for small files. The main use case for dictionary compression is when you have a lot of small files of the same type (same statistics) that need to be compressed separately.

Bit Vector Compressors:

(Bit Vector Compressors). Any binary sequence of length n with sparsity δn can be compressed losslessly with the following rate and random access:

$$R = O\left(\delta\, log\left(\frac{1}{\delta}\right)\right),\ t = O\left(log\left(\frac{1}{\delta}\right)\right)$$

Random Access Compressor Roadmap:

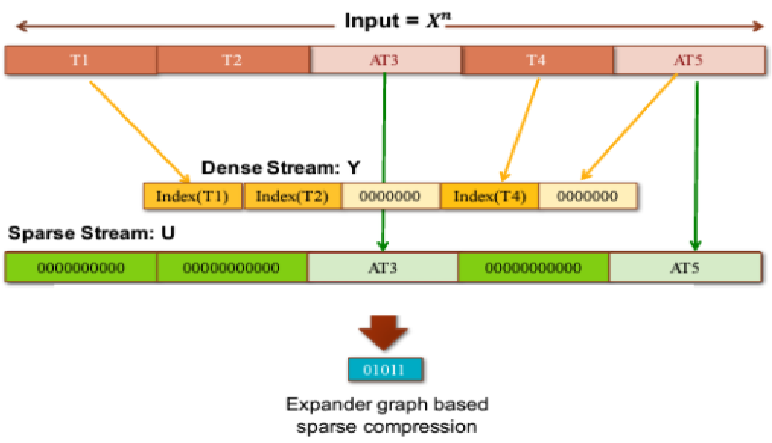
1. **Parameter determination:** The random access decompressor first decodes the compressor parameters. These parameters include information about the type of encoding operation, source parameters, block size, etc.
2. **Block determination:** To decode the symbol x_r, we first determine its corresponding block in the encoding phase. The corresponding block index is $bid = [r/b]$. We next proceed to decompress the dense and the sparse blocks Y_{bid} and U_{bid}.
3. **Dense & Sparse block decompression:** The block Y_{bid} can be directly read after determining its location in-stream. The sparse block U_{bid}

decompression employs the random access properties of the bit vector compressor.

4. **Block merging:** In the final stage, the decoded blocks Y_{bid} and $U_{bid}$are merged by the decompressor to obtain the input block B_{bid}. Finally, we output the source symbol x_r which is a part of the block B_{bid}

Essentially, every random access query requires decoding the parameters and the entire block of which the query symbol is a part. Thus, effectively the choice of block size b is one of the major design decisions for different types of sources and block encoding schemes. For unknown sources, the source parameters, which describe the source (and hence the compressor), need to be encoded in an efficient manner, using a fixed number of bits (independent of the sequence length n).

Random Access For Known I.I.D Sources:



Compression Roadmap: block encoding is done via a typicality encoder. Typical sequences are encoded in-stream Y while a-typical sequences are encoded in-stream U.

Encoding Scheme

1. **Block encoding:** The sequence x_n is divided into blocks of length b and is processed block by block. Let

$$L_b = b\left(H(X) + \epsilon_1\right)$$
$$L_N = (n/b)L$$

We represent each block $B_i = x_{ib}^{(i+1)b-1}$, $i = 1, \ldots\ldots, \frac{n}{b}$ with the help of sequences: $Y_i = y_{iL_b}^{(i+1)L_n-1}$ and $U_I = u_{ib}^{(i+1)b-1}$, the sequence x^n is thus represented by the streams.

$$Y = \left(Y_1, \ldots\ldots Y_{\frac{n}{b}}\right)$$
$$U = \left(U_1, \ldots\ldots U_{\frac{n}{b}}\right)$$

Fix $\epsilon_2 > \epsilon_1 > 0$. To generate Y_i *and* U_i we use a typicality encoder that enumerates all ϵ_1 - typical sequences and encodes B_i as follows:

- If $B_i \in \mathcal{T}_{\epsilon_1}^{b}(X)\ then\ Y_i = I\left(B_i\right)$ where $I(B_i)$ is the index of B_i in the enumerated typical set. In this case, we set $U_i = 0^b$.
- If $B_i \notin \mathcal{T}_{\epsilon_1}^{b}(X)\ ,\ then\ Y_i = 0^{L_b}\ and\ U_i = B_i$

2. **Dense stream encoding:** The dense stream $Y = y^{L_n}$ is encoded as is with an identity map. Note that each block Y_i is already compressed in the block encoding phase.
3. **Sparse stream encoding:** To compress the sparse stream $U \equiv u_n$, we distinguishes two cases:
    - If the sparsity of the u_n is less than ϵ_2, then u_n is compressed using bit vector compressors (expander graph-based sparse compressors).
    - If not, then u_n is encoded as is without any compression.

   To differentiate between these cases, we also store a bit S_b where $S_b = 1$ if the desired sparsity condition is met.
4. **Parameter encoding:** In this scenario, as we have assumed that the source is known to both the encoder and the decoder, we do not need to store any parameters.

Random Decoding Scheme

The random access to the symbol x_r is as follows.

1. **Parameter determination**: Since the encoder/decoder is assumed to know the source, the decoder does not need to decode any parameters in this stage.
2. **Block determination**: We determine the index of the block containing x_r: $bid = [r/b]$.

3. **Dense & sparse block decompression:** We read the block Y_{bid} from the Y stream by accessing the corresponding $L_b\ bits$. If $Y_{bid} \neq 0^{L_b}$, then we decode Y_{bid} to recover B_{bid} and output the r mod b bit of B_i.If $Y_{bid} = 0^{L_b}$, then we have two cases:
    - If $S_b = 1$ then using the expander graph-based retrieval method, we read symbol u_r from the encoded sequence u^n. This takes $O\left(log\left(\frac{1}{\epsilon_2}\right)\right)$ bits access. We then output u_r.
    - If $S_b = 0$ then we directly output u_r

Universal Compression with Random Access

Let X be an unknown stationary and ergodic source with entropy rate $H(X)$. Then there exists a fixed random access compression scheme that achieves close to optimal compression i.e. rate-random access pairs: (R, t) = $(H(X) + \delta$, t(δ)), for some t(δ).

Lempel Ziv compressors with constant random access

Let $L_{LZ}(x^n)$ be the compression rate of LZ78 for the sequence x^n . Then, for sufficiently large n, there exists a random access compression scheme that achieves a rate $R^{(n)}$ close to that of LZ78, along with constant random access:

$$R^{(n)} < L_{LZ}(x^n) + \delta, \quad t^{(n)} < t(\delta)$$

for some function $t(\delta)$.

Theorem 4 (Lempel-Ziv Redundancy). Let $L_{LZ}(x^n)$ be the compression rate of LZ-78 for the sequence x^n . Then:

$$L_{LZ}(x^n) \leq H_k(x^n) + \frac{Ck}{f(n)}$$

where $f(n) = \frac{log(n)}{log(log(n))}$, $C$is a constant, $H_k(x^n)$denotes the k^{th}order entropy of the sequence x^n.

The theoretical joint-typical distribution is expensive to construct and store because its complexity is exponential. So, we are using the compressed length lempel-ziv scheme to construct the joint typical distribution, by setting a threshold length and using the aforementioned compression technique.

Results

- We used the Zstd API to simulate dictionary-based lempel ziv compression.
- The following are the obtained compression ratios using different schemes of compression.

S. no.	**Type of Compression**	**Compressed size (in Bytes)**	**Compression ratio**
1	Simple Compression	40,667,764	2.459
2	Dictionary Compression	35,631,311	2.806
3	Dictionary Compression using chunks	35,598,780	2.809

- To observe the trade-off between chunk size and compression ratio and runtime performance, we compressed using varying chunk sizes.

S. no.	**Chunk size (in Bytes)**	**Compressed size (in Bytes)**	**Average compression time (in sec)**
1	512000	36361961	0.714
2	985088	35793266	0.783

3	1458176	35739964	0.766
4	1931264	35704968	0.750
5	2404352	35685085	0.732
6	2877440	35664601	0.713
7	3823616	35656041	0.691
8	4296704	35642683	0.683
9	4769792	35648633	0.676
10	5242880	35631259	0.671

We can observe that the compression ratio increases with increase in chunk size and at the same time compression time decreases, which is the same trend obtained theoretically.

- To observe the trade-off between dictionary size and compression ratio and runtime performance, we compressed using varying chunk sizes.

S. no.	**Chunk size (in Bytes)**	**Compressed size (in Bytes)**	**Average compression time (in sec)**
1	10240	35690263	0.777
2	114073	35631514	0.672
3	217906	35611867	0.670
4	321739	35597781	0.665
5	425572	35566262	0.667
6	529405	35552081	0.667
7	633238	35536256	0.670
8	840904	35517253	0.673

9	944737	35564179	0.675
10	1048570	35491525	0.665

We can observe that the compression ratio increases with increase in dictionary size and at the same time compression time decreases, which is the same trend obtained theoretically.

File structure

- **`src/`** **:**
  All the necessary programs are in this directory.
  - ➢ `simple_compression.c` - Compresses without using any dictionary
  - ➢ `dictionary_compression.c` - Compresses using a trained dictionary
  - ➢ `dictionary_compression_chunks.c` - Compresses using dictionary by dividing the file into chunks
  - ➢ `random_access.c` - Compresses using a dictionary by dividing the file into chunks provides random access i.e; we can obtain the data between any two specified bounds.

- **`benchmark/`** **:**
  All the programs for benchmarking are in this directory. The file nomenclature is the same as that of `src/` directory. Some additional files are -
  - ➢ `createDict.sh` - A shell script for training dictionaries of various sizes on the same data file.

Links

We used the Wikipedia corpus for compressing.

- For instructions to run the programs, refer to https://github.com/srikanth2001/IDP_EE4015#readme

- Source code: https://github.com/srikanth2001/IDP_EE4015

TODO: Simulate streaming decompression. Improvise the chunking scheme rather than using plain chunking. Implement the compression technique described in the paper.

References

- K.Tatwawadi, Shirin Saeedi Bidokhti, Tsachy Weissman, “On Universal Compression with Constant Random Access”
- Thomas M Cover, Joy A Thomas, “Elements of Information Theory”
- Kewen Liao, et al, “Effective Construction of Relative Lempel-Ziv Dictionaries”
- Zstandard - Real-time data compression algorithm
- Getting Started - Zstd dev