CSC2525 Project: Weighing Elias-Fano against Golomb Delta Compression Methods for Sorted Integers

Michael Zhou

2025-04-09

1 Introduction

Compression for integers is a fundamental problem in computer science. Due to the way operating system are set up, integers vary by byte size: usually 1 byte, 2 bytes, 4 bytes, or 8 bytes. However, the integers themselves usually do not need all their allocated bits to represent their value.

One common use case of integers involves storing them in a sorted order. For example, timestamps in log files, and IDs of any list of objects. Storing integers successively tends to result in small differences between them. The compression methods that we will explore exploit this property of sorted integers to efficiency compress them, approaching the lower bound of an exact set. We aim to implement these algorithms in $C++$ from the bottom-up, and evaluate their performance on randomized tests.

1.1 Elias–Fano Encoding

Elias–Fano encoding is a succinct representation for *monotonically increasing* sequences. The method leverages the fact that in a sorted set, the difference between successive integers tends to be small — a property that can be used to split and compress the representation.

Given a sorted sequence $S = \{x_0, x_1, \ldots, x_{n-1}\} \subseteq [0, u)$, each integer is divided into two parts:

- A **lower part** of ℓ bits, where $\ell = \lfloor \log_2(u/n) \rfloor$, stored as a contiguous array of bit-packed integers.
- An **upper part**, which corresponds to the high bits of each number. These are encoded using a unary-coded bitvector of size $n + \frac{u}{2^\ell}$, marking how many times each high value appears.

To decode the original values, one reads the positions of 1s in the upper bitvector (which gives the upper bits of each number) and combines them with the lower bits from the array.

Elias–Fano supports:

- Constant-time `select(i)`: retrieve the i-th element.
- Constant-time `rank(x)`: count how many values are $\leq x$.

Its compression effectiveness is especially good when the input set is moderately dense (i.e., n is not too small compared to u). It achieves space usage close to

$$n \left(\log_2 \frac{u}{n} + 2 \right) \text{ bits,}$$

which is within a small constant factor of the information-theoretic lower bound.

1.2 Golomb Delta Encoding

Golomb Delta encoding compresses a sorted sequence by encoding the *gaps* ($d_i = x_i - x_{i-1}$) between successive elements, rather than the elements themselves. This works especially well when these gaps follow a geometric or exponentially decaying distribution.

Each gap is encoded using:

- The **quotient** $q = \lfloor \frac{d_i}{M} \rfloor$ is encoded in unary.
- The **remainder** $r = d_i \bmod M$ is encoded in truncated binary.

The parameter M is selected based on the average gap size, typically set to $\lceil \log_2 n \rceil$, to minimize expected code length.

In practice, Golomb encoding is implemented using efficient bit-level operations to store unary and binary segments compactly. When decoding, the algorithm reads unary runs to determine quotients and follows with fixed-width reads for remainders.

Golomb Delta achieves high compression ratios when:

- The sequence is very sparse (large gaps),
- The distribution of gaps is skewed toward smaller values (as in real-world document IDs).

It achieves space usage close to

$$n\left(\log_2 \frac{u}{n} + 1.5\right) \text{ bits},$$

However, Golomb encoding is inherently sequential: decoding requires reconstructing all previous gaps to recover a specific index, leading to poor random access performance compared to Elias–Fano.

2 Methods

To evaluate the performance of Elias–Fano and Golomb Delta encoding for sorted integer compression, we implemented both algorithms in C++ with a shared interface to enable uniform testing. Our implementation emphasizes both compression correctness and runtime efficiency, using low-level bitwise operations and custom bitvectors for space-efficient storage.

(Repo link here)

Elias–Fano

We implemented Elias–Fano by splitting each integer in the sorted set into lower and upper bits. The lower bits were stored compactly in a contiguous array, while the upper bits were stored using a unary-encoded bitvector, allowing efficient `select` operations. We optimized storage by implementing our own `BitVector` class, storing true bit-level data and adding auxiliary metadata to support constant-time `rank` and `select` queries.

Golomb Delta

For Golomb Delta encoding, we encoded the differences (gaps) between successive integers using Golomb coding with truncated binary representation. The optimal Golomb parameter M was selected dynamically based on the average gap size. Encoding was done sequentially, and decoding involved reconstructing the original values from the encoded gaps. Due to the inherently sequential nature of Golomb decoding, we did not include rank or select support.

Experimental Setup

We evaluated both algorithms on synthetically generated sorted integer sequences with varying:

- **Universe size** (u): 10^6 and 10^7
- **Density**: 1%, 5%, and 10% (corresponding to 10k, 50k, and 100k elements for $u = 10^6$, etc.)

Each algorithm was tested on:

- **Compression Size**: number of bits used after encoding
- **Encoding Time**: measured in microseconds
- **Random Access Time**: time to retrieve the i-th element from the compressed representation
- **Compression Ratio**: computed relative to uncompressed storage assuming 32-bit integers

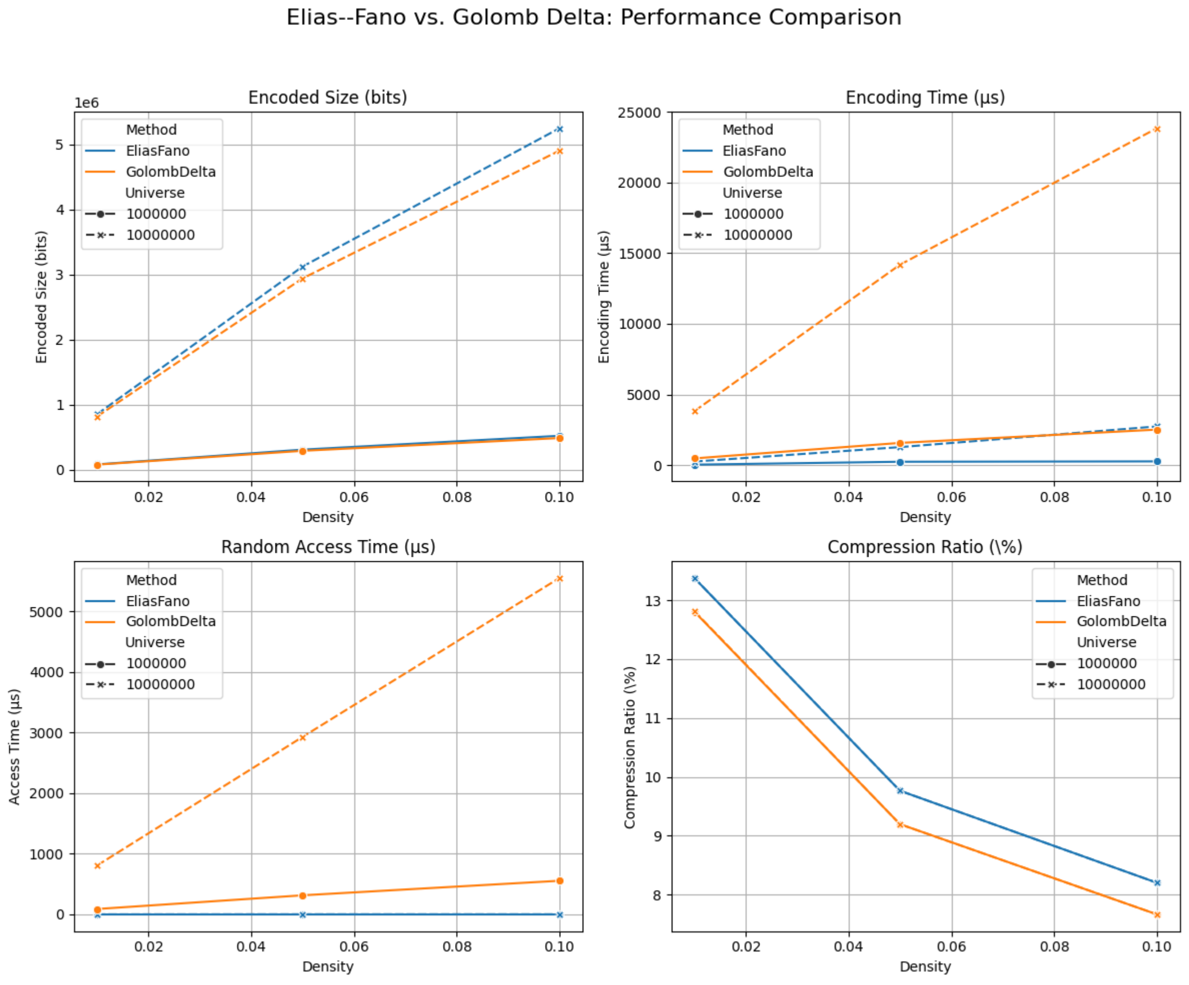
3 Results

Table 1: Elias–Fano vs. Golomb Delta Compression Performance

Universe	Density	#Elems	Size (bits)		Time (µs)		Access (µs)		Ratio (%)	
			EF	GD	EF	GD	EF	GD	EF	GD
10^6	0.01	10,000	85,618	81,961	33	474	0	88	13.38	12.81
10^6	0.05	50,000	312,495	294,336	238	1571	0	314	9.77	9.20
10^6	0.10	100,000	524,997	490,642	264	2512	0	554	8.20	7.67
10^7	0.01	100,000	856,246	820,208	251	3817	0	805	13.38	12.82
10^7	0.05	500,000	3,124,999	2,943,061	1273	14175	1	2927	9.77	9.20
10^7	0.10	1,000,000	5,249,998	4,906,485	2737	23833	1	5550	8.20	7.67

The results in **Table 1.** show that Elias–Fano provides faster encoding and random access times compared to Golomb Delta, which achieves slightly better compression ratios, especially at higher densities. Elias–Fano's two-level bitvector approach offers near-constant-time access, while Golomb Delta's variable-length encoding results in higher access and encoding times. These trade-offs highlight Elias–Fano's advantage in applications requiring fast queries, while Golomb Delta is better for achieving more compact storage at the cost of slower access.

We also present some visualizations with *matplotlib*:


Elias--Fano vs. Golomb Delta: Performance Comparison
Encoded Size (bits)
1e6
Encoded Size (bits)
Encoding Time (µs)
Random Access Time (µs)
Access Time (µs)
Compression Ratio (\%)
Compression Ratio (%)
Density
Method
EliasFano
GolombDelta
Universe