SYSTÈMES NUMÉRIQUES POUR L'HUMAIN
ÉCOLE UNIVERSITAIRE DE RECHERCHE

EUR DS4H - Master 1 Informatique
spécialité Intelligence Artificielle

Software Engineering project report

Data compressing to speed up transmission

Author :
Hugo Viana

Academic advisor :
Jean-Charles Régin

2025

Abstract

This project was realized as an assignment for the Software Engineering class of the Master degree of Computer Science. The aim was to implement different algorithms of bitpacking to speed up the time of transmission of arrays of 32-bit integers. The algorithms implemented are a consecutive compression where compressed integers can overlap, non-consecutive compression where compressed integers cannot overlap and overflow compression where an overflow area at the end of the compressed array is allocated for bigger integers that would increase the bit size. This project shows that the non-consecutive compression was not efficient if the array contained integers that were too large. The overflow compression is efficient but the compression time is much longer than for the other algorithms but can be very efficient with unevenly distributed arrays. The most adequate compression method is the consecutive compression as it able to compressed all types of arrays while having a fast compression time.

1 Method

The project was developed in Java using Maven, providing a structured environment and facilitating both compilation and dependency management. The overall architecture is based on the Factory design pattern, enabling flexible creation of different compression strategies.

At the core of the implementation is an abstract class BitPacking, which encapsulates the fundamental properties of a compressed array. This class maintains:

- an array of integers representing the compressed data,
- an integer specifying the bit-width used for compression, and
- an integer storing the size of the original, uncompressed array.

BitPacking defines three abstract methods:

- `compress(int[] array)`: takes an array of integers as input and compresses it into the internal representation.
- `decompress(int[] array)`: takes an empty array as input and reconstructs the original integers from the compressed data.
- `get(int i)`: returns the integer at a specified position from the compressed array, providing direct random access.

Three concrete classes inherit from BitPacking, each implementing a specific compression strategy:

- BPConsecutive: implements bit-packing across consecutive words.
- BPNonConsecutive: implements bit-packing without allowing integers to span multiple words.
- BPOverflow: implements the overflow-based compression to efficiently handle outlier values.

Finally, a factory class BPFactory is provided, which instantiates the appropriate compression class based on the selected compression method, thereby encapsulating the creation logic and promoting modularity.

In addition to the core implementation, the project includes unit tests for each method, developed using JUnit. These tests ensure the correctness and reliability of all functionalities, including compress, decompress, and get. By systematically verifying the behavior of each compression strategy (BPConsecutive, BPNonConsecutive, and BPOverflow), the tests provide confidence that the algorithms produce accurate results across a variety of input arrays and edge cases. This testing framework also facilitates maintenance and future extensions of the system.

1.1 BPConsecutive

`Compression (compress(int[] array)):`

The method first determines the maximum value in the input array to compute bitSize, the minimum number of bits needed to represent any integer. It then calculates the length of the compressedArray to store all bits. If the total number of bits does not divide evenly into 32-bit words, an extra integer is allocated to accommodate the remainder. Each integer is processed bit by bit and packed sequentially into the compressedArray. When the end of a 32-bit integer is reached, the method continues writing into the next integer.

Key detail: The algorithm uses bitwise operations (», «, &, |) to extract and insert individual bits, ensuring exact placement without data loss.

`Decompression (decompress(int[] array)):`

The method reconstructs the original integers from the compressedArray by reading bitSize bits for each integer. If the bits for a single integer span two integers, the algorithm correctly continues reading from the next word. his ensures lossless recovery of the original array, verifying that compression does not alter any data.

`Random Access (get(int i)):`

This method allows retrieval of the i-th integer directly from the compressed array without decompressing the entire array. It calculates the bit position corresponding to the requested integer, identifies the relevant integer(s) in compressedArray, and reconstructs the integer bit by bit. This operation runs in constant time, preserving efficiency for selective access in large datasets.

1.2 BPNonConsecutive

`Compression (compress(int[] array)):`

The method determines the maximum value in the input array to calculate bitSize, the minimum number of bits required for each integer. It calculates bitsPerInteger, representing how many integers can fit in a single compressed integer based on the available 32 bits. The length of the compressedArray is computed to accommodate all integers, ensuring that each compressed integer is fully contained within one integer. Each input integer is packed sequentially but restricted to its allocated 32-bit integer, moving to the next compressed integer once the capacity is reached.

`Decompression (decompress(int[] array)):`
Reconstructs the original integers from the compressedArray by reading bitSize bits per integer, staying within the same compressed integer. The method advances to the next compressed integer only when the current one has been fully read, ensuring correct recovery. This guarantees lossless decompression while maintaining predictable boundaries for each compressed integer.

`Random Access (get(int i)):`
Directly retrieves the i-th integer from the compressed array without decompressing the full array. The method calculates the specific integer index and bit offset where the target integer resides. Using bitwise extraction, it reconstructs the integer in constant time, preserving efficient random access.

1.3 BPOverflow

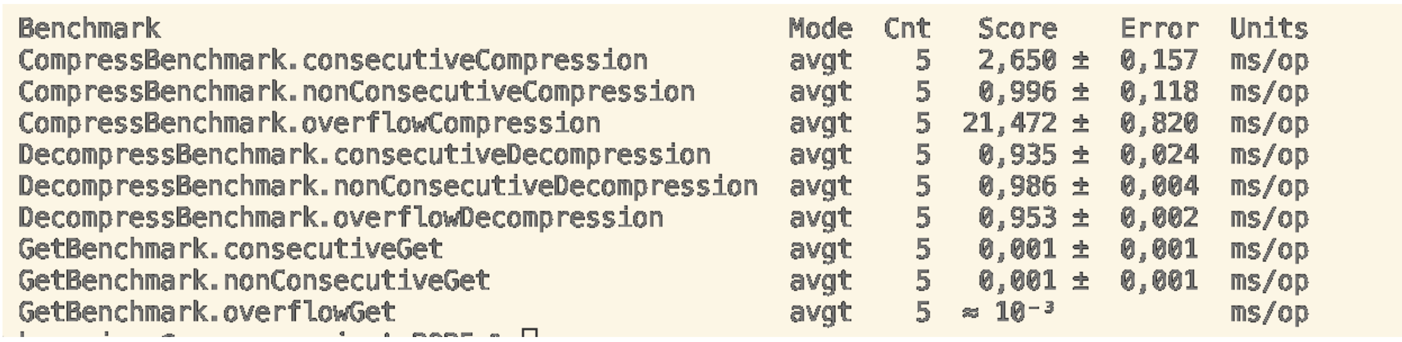
`Compression (compress(int[] array)):`
The method first analyzes the input array to determine the distribution of bit-lengths across all integers. This allows the algorithm to select an optimal bitSize for the main compressed area while identifying how many integers will need the overflow area. Each integer is then processed: If its bit-length fits within the main bitSize, it is compressed normally. If it exceeds bitSize - 1, it is flagged and its value is placed in a separate overflow section. A special high bit in the main compressed integer indicates the presence of an overflow value. The method carefully tracks positions and uses bitwise operations to pack both the main and overflow integers into the compressedArray, ensuring that all data is accessible in a contiguous structure.

`Decompression (decompress(int[] array)):`
The main array is first reconstructed from the primary compressed section. For integers flagged as overflow, the algorithm locates the corresponding value in the overflow section and replaces the placeholder with the actual value. This two-step process guarantees exact recovery of all original integers while maintaining a compact representation for the majority of values.

`Random Access (get(int i)):`
Direct access requires checking whether the i-th integer is in the overflow area. If it is not, the integer is extracted directly from the main compressed section using its bit offset. If it is flagged as overflow, the method calculates the appropriate position in the overflow section and reconstructs the integer from its overflowBitSize. This allows constant-time access to any element, even if it resides in the overflow area.

2 Benchmarks

To compare the performance of the implemented compression strategies, benchmarks were conducted using JMH (Java Microbenchmark Harness). These benchmarks measure



```
Benchmark                                       Mode  Cnt    Score    Error  Units
CompressBenchmark.consecutiveCompression        avgt    5    2,650 ±  0,157  ms/op
CompressBenchmark.nonConsecutiveCompression     avgt    5    0,996 ±  0,118  ms/op
CompressBenchmark.overflowCompression           avgt    5   21,472 ±  0,820  ms/op
DecompressBenchmark.consecutiveDecompression    avgt    5    0,935 ±  0,024  ms/op
DecompressBenchmark.nonConsecutiveDecompression avgt    5    0,986 ±  0,004  ms/op
DecompressBenchmark.overflowDecompression       avgt    5    0,953 ±  0,002  ms/op
GetBenchmark.consecutiveGet                     avgt    5    0,001 ±  0,001  ms/op
GetBenchmark.nonConsecutiveGet                  avgt    5    0,001 ±  0,001  ms/op
GetBenchmark.overflowGet                        avgt    5   ≈ 10⁻³           ms/op
```

Figure 1: Table of the benchmarks

the execution time of the core methods—compress, decompress, and get—for each class (BPConsecutive, BPNonConsecutive, and BPOverflow). By providing high-resolution and statistically rigorous timing, JMH ensures that the results accurately reflect the relative efficiency of each approach.

I generated a random array of 100,000 integers ranging from 0 to 1,000,000, I then used the JMH library to automate the benchmark process with warmup iterations and then 5 iterations of every benchmark.

On top of the JMH benchmarks, I also generated 1,000 arrays of random size and range to see the difference in results in regard to the size and range of an array, I only worked with uniformly distributed arrays.

We can see in the 1st figure that the non-consecutive compression is the fastest but is comparable with the consecutive while the overflow compression is very slow. For the decompression and the get method, they are all comparable in execution time.

The non-consecutive method is very fast but it doesn't work if an integer in the array takes more than 16 bits because then only one integer can be stored in an integer and the size remain the same as we can see in figure 2. The compression rate is null for a majority of arrays.

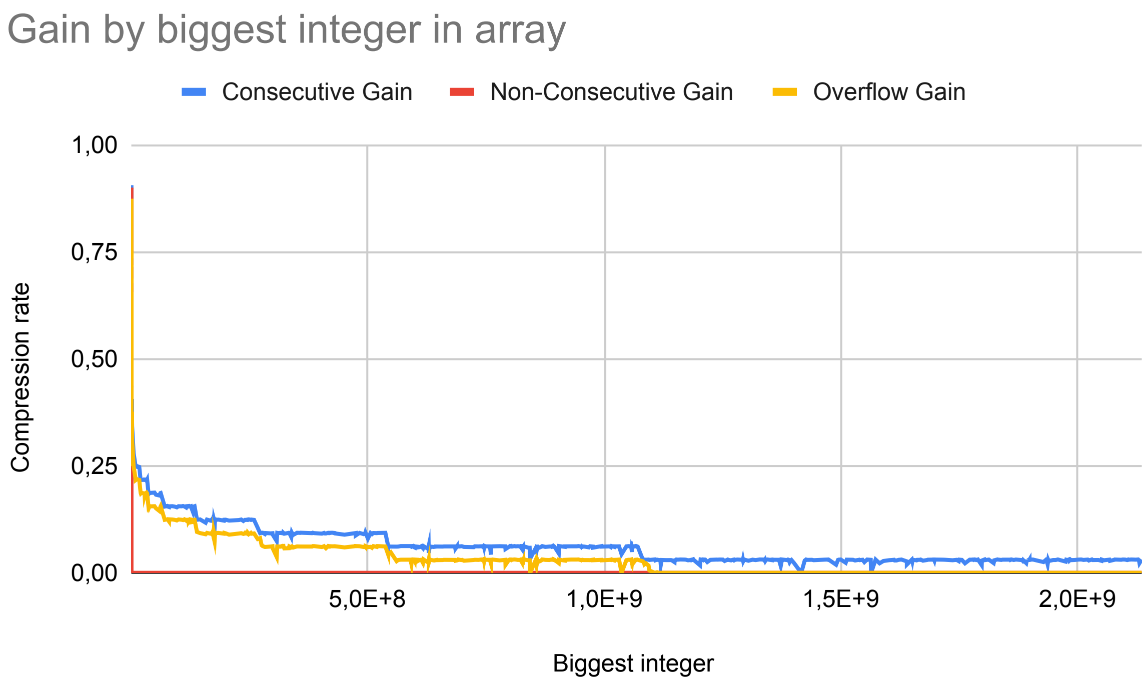


Figure 2: Graph of the compression rate of each compression algorithm by the biggest integer in the array

3 Conclusion

This project demonstrates the implementation and evaluation of three bit-packing compression strategies—consecutive, non-consecutive, and overflow compression—for arrays of 32-bit integers. Through careful design using the Factory pattern and a common abstract class, the system achieves modularity, maintainability, and direct random access to compressed data.

The benchmarks reveal a clear trade-off between compression efficiency and computational overhead. Non-consecutive compression provides fast execution but suffers from poor compression rates when input integers exceed the allocated bit-width. Overflow compression, while capable of handling highly uneven arrays efficiently, incurs a significant cost in compression time. Consecutive compression, on the other hand, strikes the best balance: it compresses a wide variety of arrays effectively while maintaining low execution time, making it the most practical choice for general-purpose applications.

Overall, the study highlights the importance of selecting a compression strategy based on both the characteristics of the input data and the operational requirements. The combination of unit testing and rigorous benchmarking ensures that the implementation is both reliable and performant. This work provides a solid foundation for further exploration of adaptive compression techniques and real-time data transmission optimization.