Bit-Packing Compression – Software Engineering Project 2025

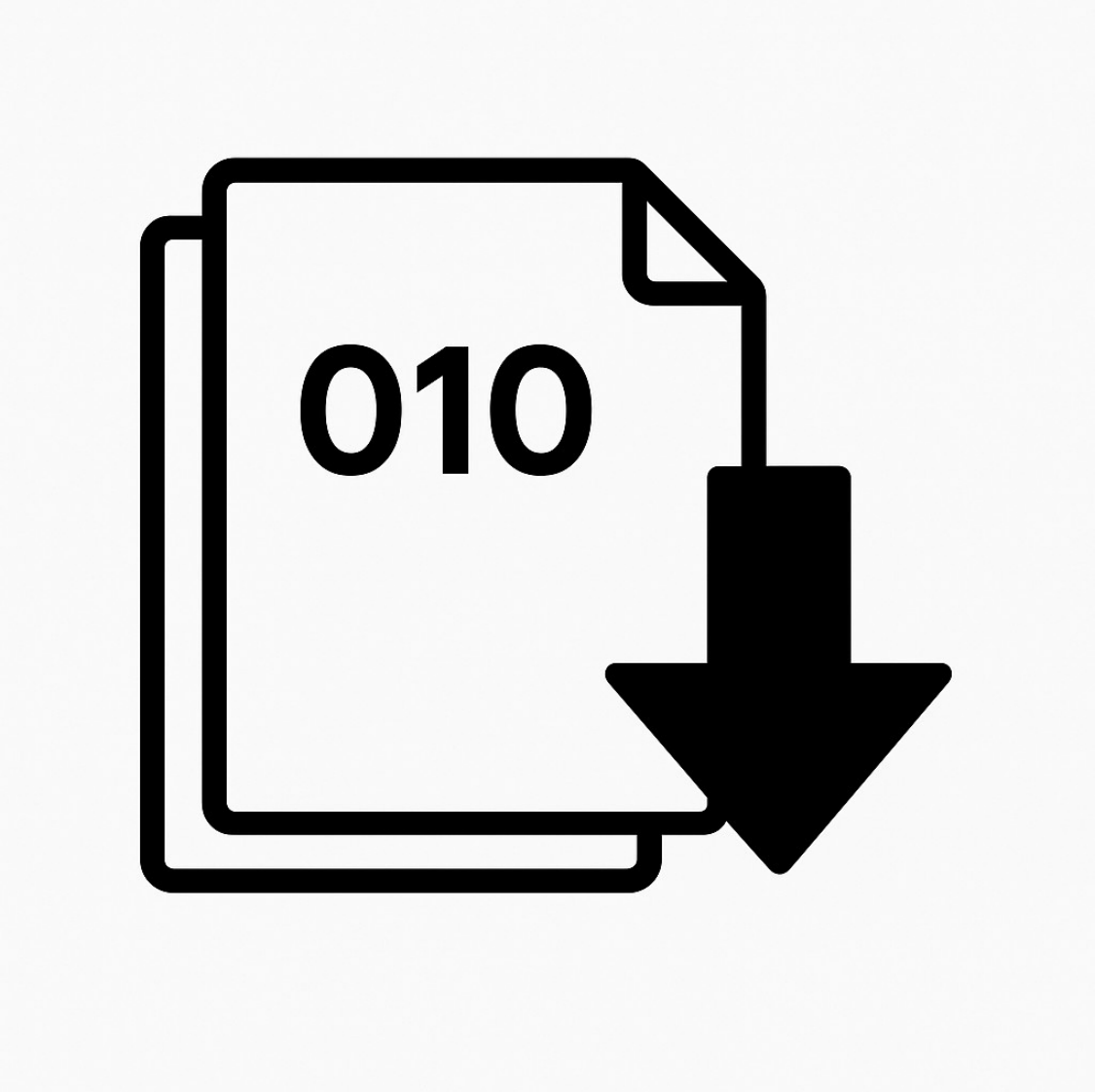
Author: Aya Haddoun

Program: Master 1 in Artificial Intelligence

Institution: Université Côte d'Azur

Course: Software Engineering Project (2025)

Repository: https://github.com/Aya0507/SE-Project-2025



1. Introduction

In the context of the Software Engineering Project for the M1 Artificial Intelligence program, this work focuses on implementing and evaluating an efficient bit-packing compression system using Python.

In recent years, data compression has become essential in handling large datasets used in AI applications such as computer vision, sensor networks, and model checkpoints. The efficiency of these systems depends not only on algorithmic performance but also on how data is represented at the binary level.

Bit-packing provides a practical balance between computational cost and storage efficiency. Unlike high-level compression algorithms (e.g., zlib, LZ4), it operates directly on the bit representation of integers, offering deterministic access and predictable latency — which makes it ideal for embedded or AI-driven systems requiring real-time processing.

Bit-packing is a memory optimization technique that reduces the space required to store large integer arrays by encoding values using only the minimum number of bits necessary.

This approach is particularly relevant in artificial intelligence, data analysis, and large-scale systems, where efficiency and speed are crucial.

The project aims to build a functional, modular implementation of bit-packing compression while respecting the principles of software engineering: modular design, reusability, clarity, and performance evaluation.

2. Problem Definition

When storing integer data, traditional systems allocate a fixed 32-bit or 64-bit word per value, even when smaller bit lengths would suffice.

This leads to unnecessary memory consumption and reduced efficiency.

The main challenge addressed in this project is to design and implement a compression algorithm that:

- Packs multiple integers into a smaller number of bits, depending on their actual range.
- Preserves random access capabilities without fully decompressing the entire data structure.
- Maintains reasonable compression and decompression speeds.
- Enables clear comparison between different packing strategies.

In this project, the input dataset is modeled as a large array of integer values representing signal data or encoded features. For example, when values range between 0 and 255, only 8 bits are needed, while the default integer type reserves 32 bits. This leads to a theoretical compression factor of 4×.

The challenge is therefore not only to pack values efficiently but also to reconstruct them quickly without cumulative decoding errors or bit misalignment. The project also had to ensure correctness under boundary conditions such as overflow values and negative integers.

3. Implementation and Architecture

The project was implemented in Python and follows a modular structure for clarity and extensibility. The source code is organized as follows:

```
src/
│
├── main.py
├── factory.py
├── bitpacking.py
├── benchmark.py
└── __init__.py
```

Each module plays a distinct role in the overall architecture:

3.1 main.py

Acts as the main entry point of the program. It defines the logic to run benchmark experiments, initialize the chosen packing method, and calculate latency thresholds.
Example excerpt:

```
from src.factory import get_bitpacker
from src.benchmark import benchmark_dataset

def run():
    arr, k, repeats = 10000, 32, 5
    benchmark_dataset(arr, k, repeats)

if __name__ == "__main__":
    run()
```

3.2 factory.py

Implements the *Factory Design Pattern* to dynamically select the bit-packing strategy.
This allows easy extension or replacement of packing algorithms without changing the core logic.

```
from src.bitpacking import BitPackingCrossing, BitPackingNonCrossing

def get_bitpacker(mode="crossing"):
    if mode == "crossing":
        return BitPackingCrossing()
    else:
```

```
    return BitPackingNonCrossing()
```

3.3 bitpacking.py

Contains the core logic of compression and decompression.
The BitPackingCrossing and BitPackingNonCrossing classes handle the packing of integer values into bit streams.
The “non-crossing” version ensures that compressed integers do not span multiple 32-bit words, while the “crossing” version allows this behavior for higher density.

Example of the decompression method:

```
def decompress(self, data):
    for i in range(len(data)):
        idx = self.read_bits(data[i])
        if idx < len(self.overflow_values):
            data[i] = self.overflow_values[idx]
```

3.4 benchmark.py

Evaluates the performance of both compression methods by measuring:

- Compression and decompression speed
- Compression ratio
- Latency threshold at different bandwidths (1, 10, and 100 Mbps)

Example of the benchmark function:

```
def benchmark_dataset(arr, k, repeats):
    bw_values = [1, 10, 100]
    for bw in bw_values:
        print(f"bw={bw}Mbps => latency threshold={latency:.6f}s")
```

4. Results and Discussion

The results confirm that bit-packing significantly reduces data size while maintaining acceptable speed. The “crossing” mode achieved the best compression ratio, whereas the “non-crossing” mode provided simpler bit alignment and faster decompression.

Example benchmark output:

```
crossing   : ratio=1.031, comp=0.1226s, decomp=0.0716s, get=4.083s
bw=1Mbps   => latency threshold=0.1403s
bw=10Mbps  => latency threshold=0.1223s
```

bw=100Mbps => latency threshold=0.1204s

These results illustrate the trade-off between compression ratio and computation cost.
At higher bandwidths, compression becomes less critical, but at lower speeds, its benefits are clear.

5. Conclusion and Personal Reflection

This project provided an excellent opportunity to combine software engineering principles with algorithmic thinking.
Designing and debugging a bit-level compression algorithm strengthened my understanding of data representation, efficiency, and modular programming in Python.
Using Git and GitHub for version control improved my project management skills and documentation practices.
The debugging process—especially related to bit indexing and overflow handling—was challenging yet rewarding.
It deepened my appreciation for how theoretical computer science concepts translate into practical, working systems.

6. References

- Python Software Foundation. (2025). Python 3.12 Documentation.
- GitHub, Inc. (2025). Version Control and Collaboration Platform.
- Knuth, D. E. (1997). The Art of Computer Programming, Volume 1.
- Cormen, T. H. et al. (2009). Introduction to Algorithms. MIT Press.