UE Software Engineering: Project Report

Data compressing for speed up transmission

Arno LESAGE (n°2202985)

University: Université Côte d'Azur – EUR DS4H
Curriculum: Master 1 – Informatique, parcours Intelligence Artificielle
Lecture Instructor: Pr. Jean-Charles Régin
Hand-out date: 2nd November 2025

DIGITAL SYSTEMS FOR HUMANS
GRADUATE SCHOOL AND RESEARCH

UNIVERSITÉ CÔTE D'AZUR

Abstract – In statistical physics, the Gas-Lattice Hardcore Model (GLHM) provides a simplified yet quite accurate representation of a gas behaviour by examining the placement of gas particles on a graph and the resulting independent sets. Efficient sampling from this model may offer deeper insights into the dynamic properties of gases, enhance the understanding of critical points and improve the accuracy of gases simulations. Previous studies have established that sampling from the GLHM is NP-Hard for certain parameter values above a specific threshold, thereby complicating sampling from larger graphs. In this bachelor's thesis, we review various approaches, including the use of Markov chains for independent sets and perfect sampling frameworks such as Coupling from the Past and Bounding Chains. By comparing these methodologies, we highlight their advantages and limitations.

Table of Contents

Introduction

In the field of Network protocols and Telecommunication, where delay may affect some applications, transmission speed is considered a key metric. Optimising the former quickly became a topic in itself, one of the global challenges of the internet and remains to this day an actively researched field.

One of the unravelled problems was the optimisation of integer arrays transmission, which might become costly. Indeed, even today, most “normal” integers are stored in a 32-bit format. Unfortunately, this way of proceeding also implies the usage of 32 bits to store numbers that could have been stored in a smaller format, such as 16-bit, 8-bit, or even less.

This observation led to the creation of the “Bit Packing” concept, in other words, a new concept to speed up integer transmission. The principle remains simple though: instead of transmitting raw integer arrays as is, we transmit a compressed version, reducing the number of bits taken by each integer while still being able to access them in the right order.

In this report, we explore three different implementations of Bit packing and investigate their performance in reducing the transmission delay. In the first part (1), we formalise our problem and expose the requested Bit Packing versions to implement. In the second part (2), we cover each separate implementation. Finally, part three (3) compares them before concluding.

1 – Formalisation and definition of Bit packing

1.1 – Problem formalisation

Definition 1: Bit Packing – Let A_n^b be a b-bit signed integer array of containing n integers and $A[i]$ be a function $[i]: A_n^b \to \mathbb{Z}$ which return the integer stored at position i in the array A_n. We define Bit Packing as a function:

$$bitPacking: A_n^b \to A_n^{b'}$$

such that the appearance order of the integers within the array remains the same after compression $\left(\forall i \in [\![0, n-1]\!] : A_n^b[i] = A_n^{b'}\right)$, the number of bits to encode the integers is less or equal to the original array $(b' \leq b)$, and that the compression possess a lossless decompression $\left(bitPacking^{-1}\left(bitPacking(A_n^b)\right) = A_n^b\right)$.

Note that Bit Packing can be used outside the scope of signed integer to other data types or even other data structures, but we will emit the hypothesis along the report that we only use signed integer and array-like structure to store them.

Definition 2: Transmission Delay – Let $X \to Y$ be the representation of the physical linking from the router X to the router Y, let L be the bit-length of a packet and R be the throughput of the linking in $\text{bit}.s^{-1}$, we define the transmission delay d_{trans} as $d_{\text{trans}} = \frac{L}{R}$. By assuming no propagation delay through the link (instant sending-transmission), d_{trans} corresponds to the necessary time to send the whole packet from router X to router Y.

With regards to our definition of Bit Packing and transmission delay, our goal is to implements three different version of Bit Packing to minimize the transmission delay.

1.2 – Requested versions of Bit Packing

In this subpart, we review the requested versions of Bit Packing we will implement in the next part of the report.

To begin with, let's consider a 32-bit signed integer $x \in \mathbb{Z}$, its natural representation lied in base 10, but it is stored in base 2.

Example 3: The following 32-bit integer array $A_4^{32} = \{-128, 0, 65982, 2478\}$ can be represented in base 2 as:

$$A_4^{32} = \left\{ \begin{array}{l} -0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1000\ 0000 \\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000 \\ 0000\ 0000\ 0000\ 0001\ 0000\ 0001\ 0111\ 1110 \\ 0000\ 0000\ 0000\ 0000\ 0000\ 1001\ 1010\ 1110 \end{array} \right\}$$

By observing this structure, we discover that it is not necessary to store up to 32 bits of information per integer. Indeed, we could have stored all the information on 17 bits per integer effectively reducing A_4^{32} to A_4^{17}, which corresponds, after calculation, to a total gain of 60 bits over the whole array.

Virtually, compressing this way means that we could save a whole 32-bit integer in storage and store the compressed array as:

$$A_4^{17} = \begin{Bmatrix} -0000\ 0000\ 0100\ 0000\ 0|000\ 0000\ 0000\ 0000 \\ 00|10\ 0000\ 0011\ 0111\ 110|0\ 0000\ 1001\ 1010 \\ 1110|0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000 \end{Bmatrix} = \begin{Bmatrix} 4194304 \\ 540524698 \\ 3758096384 \end{Bmatrix}$$

Where the grey bits are buffers, bits corresponding to the remaining number of bits necessary to form a 32-bit integer.

Remark 4: In this example, we used the notation A_4^{17} with $n = 4$. In the following parts of the report, n will always reflect the number of encoded integer and not the number of stored 32-bit integer.

Pseudo-algorithm 5: "Split" Bit Packing

The first version **"split"** corresponds essentially to the approach described above:

Step 1: Change base 10 representation to base 2,
Step 2: Find the number of bits required to store the biggest absolute integer,
Step 3: Concatenate all the bits of each number truncated to the maximal number of bits required within a unique array,
Step 4: Pad the array with 0 to the next multiplier of 32.

The created array contains all initial integers encoded with b' bits $(b' \leq 32)$ and is represented using three 32-bit integer.

Remark 6: It is important to remember that when storing a signed number, a bit must be used to store the sign of the integer. This bit will be included in the computation of the maximal number of bits required to encode a number.

The second version **"nosplit"** is also the same as above with the exception that if an b'-bit integer cannot be fully stored on the same 32-bit integer after concatenation then it is stored in the next integer.

In *example 3*, the "compressed" array would have been represented as:

$$A_4^{17} = \begin{Bmatrix} -0000\ 0000\ 0100\ 0000\ 0|000\ 0000\ 0000\ 0000 \\ 0000\ 0000\ 0000\ 0000\ 0|000\ 0000\ 0000\ 0000 \\ 1000\ 0000\ 1101\ 1111\ 0|000\ 0000\ 0000\ 0000 \\ 0000\ 0100\ 1101\ 0111\ 0|000\ 0000\ 0000\ 0000 \end{Bmatrix}$$

In this case, no compression would have been achieved. However, if the encoding size would have been smaller, it could have been possible to stack multiple number per 32-bit integer.

Pseudo-algorithm 7: "Overflow" Bit Packing

The last version to be implemented is **"overflow"**. The principle remains fairly similar as "split"; however, this version implements a maximum bit threshold, for which, if exceeded

store the exceeding value in an overflow area, otherwise the value is compressed using "split".

> **Step 1:** Change base 10 representation to base 2,
> **Step 2:** Search for the biggest value $v_{\max}$ strictly under the threshold (in term of bits),
> **Step 3:** For each value in the array use "split" from step 3 with necessary number of bits $v_{\max}$ if the value is under the threshold, otherwise store it in the overflow area and add a reference in the compressed array at this position to remember the order,
> **Step 4:** Pad the compressed array with 0 to the next multiplier of 32.

In *example 3*, the results with a threshold of 12 bits would have looked like this:

$$A_4^{10} = \{-0010\ 0000\ 00|00\ 0000\ 0000|1|1|00\ 0000\ 0000\}$$
$$A_{\text{overflow}}{}_2^{32} = \begin{Bmatrix}0000\ 0000\ 0000\ 0001\ 0000\ 0001\ 1011\ 1110\\ 0000\ 0000\ 0000\ 0000\ 0000\ 1001\ 1010\ 1110\end{Bmatrix}$$

In this case, we have an array where integers are encoded on 9 bits plus 1 bit for the overflow reference, which translates into 32 bits for the compressed array and 64 bits for the overflow area, in other words 96 bits or three 32-bit integer in total to represent this structure.

Note that nothing is stopping us from doing a bit packing like "split" in the overflow area too, this is what is going to happen in the implementation.

Remark 8: For all the above representations, we did not consider the addition of sign bit, in consequence, the representations are simplification of the implementation. Technical details will be discussed in next part.

2 – Implementation details

In this part, we will discuss implementation details of each version of Bit packing but first let us have a look at the project structure and organisation.

2.1 – Code structure and organisation

To begin with, all the code was written in Python 1.12.0. This choice was made for several reasons including personal affinity with this particular programming language, the presence of useful libraries (NumPy, SciPy, bitarray and PyTest) to implement our Bit packing versions and tests them as well as the presence of extensive documentation if needed through the project.

Considering all that, here is the code structure of this project:

```
C:.
│   arrayGenerator.py
│   benchmark.py
│   main.py
│   README.md
│   tests.py
├───Compressor
│   │   AbstractCompressor.py
│   │   NoSplitCompressor.py
│   │   OverflowCompressor.py
│   └───SplitCompressor.py
├───img
├───IN
├───OTHERS
│       requirements.txt
└───OUT
```

In this tree, `arrayGenerator.py` is responsible for the creation of the input files, `benchmark.py` for the benchmark, `main.py` for the `BitPacking` class and `tests.py` for the tests. In `Compressor/`, you will find the implementation for all compressor method (split, nosplit, overflow). Finally, you will find the input files within `IN/` the output files (benchmark) in `OUT/` and the required libraries in `OTHERS`.

2.1.1 – Abstract Compressor class

As you may have noticed while reading the tree, all the compressors are based on an abstract class `Compressor` that can be found in `Compressor/AbstractCompressor.py`. This class declare three abstract methods that has to be implemented by its subclasses:

- {abstract} `compress(np.ndarray[int]) → tuple[Any]` which takes an integer NumPy array in argument and returns a tuple containing the compressed version of the array along with possible other information,
- {abstract} `decompress(tuple[Any]) → np.ndarray[int]` which takes the tuple returned by `compress` as input and return an uncompressed integer NumPy array,
- {abstract} `get(int, tuple[Any]) → int` which takes both a position within the uncompressed array and the tuple returned by `compress` as input. This method returns the integer at the requested position from the compressed array.

A last concrete protected method is implemented within this class:

- {concrete, protected} `_getMaxBitLength(np.ndarray[int]) → int` which takes an integer array as input and returns as output the required bit length to encode the biggest absolute value in the array.

The other compressor classes then inherit this abstract compressor and must implements all the abstract methods to be valid. Additionally, by the signature of each function, we ensure the possibility for different subclasses to return different output (with different additional information on top of the compressed array).

Moreover, the initialisation of each class is not specified by this abstract class and can be free for all subclasses. This possibility also means that we will have to take this information into account to build our factory and our `BitPacking` class.

Remark 9: In Python, the size of integers is not limited to 32 bits, but is rather unfixed, this is why we will use the `np.int32` type from NumPy instead of the built-in Python type `int` through the project. For simplicity, we will still use the notation **"int"** in this report, but this will correspond to **"np.int32"** in the real implementation.

2.1.2 – Bit Packing class and factory

The `BitPacking` class, contained within `main.py`, is the main class in direct contact with the user. This class aims to provide control over the compressor and a simple interface for the user. This class provide three private attributes:

- {private} `__compressor:Compressor` which takes whatever `Compressor` inheriting from the Abstract Compressor,
- {private} `__arr:np.ndarray[int]` which stores the current uncompressed integer NumPy array (either the one after decompression or the one before compression depending on if decompression happened),
- {private} `__compressed:tuple[Any]` which stores the compressed array and the information relative to it.

The initialisation of the `BitPacking` class is done by feeding the constructor with a mode which can be either **"split"**, **"nosplit"** or **"overflow"** and represent the compressor to use. If necessary, the constructor also let the user add parameters like the threshold for the **"overflow"** constructor.

The constructor then sends the information to the function `main.createCompressor` which acts as a factory before declaring the other private attributes.

The `BitPacking` class also provides access to the private attributes through two setters and two getters:

- {getter} `getArr() → np.ndarray[int]` which returns the value of the private attribute `__arr`,
- {getter} `getCompressedArr → tuple[Any]` which returns the value of the private attribute `__compressed`,
- {setter} `setArr(np.ndarray[int])` which sets the value of the private attribute `__arr`,
- {setter} `changeMode(str)` which changes the compressor in use and the value of the corresponding private attribute `__compressor`.

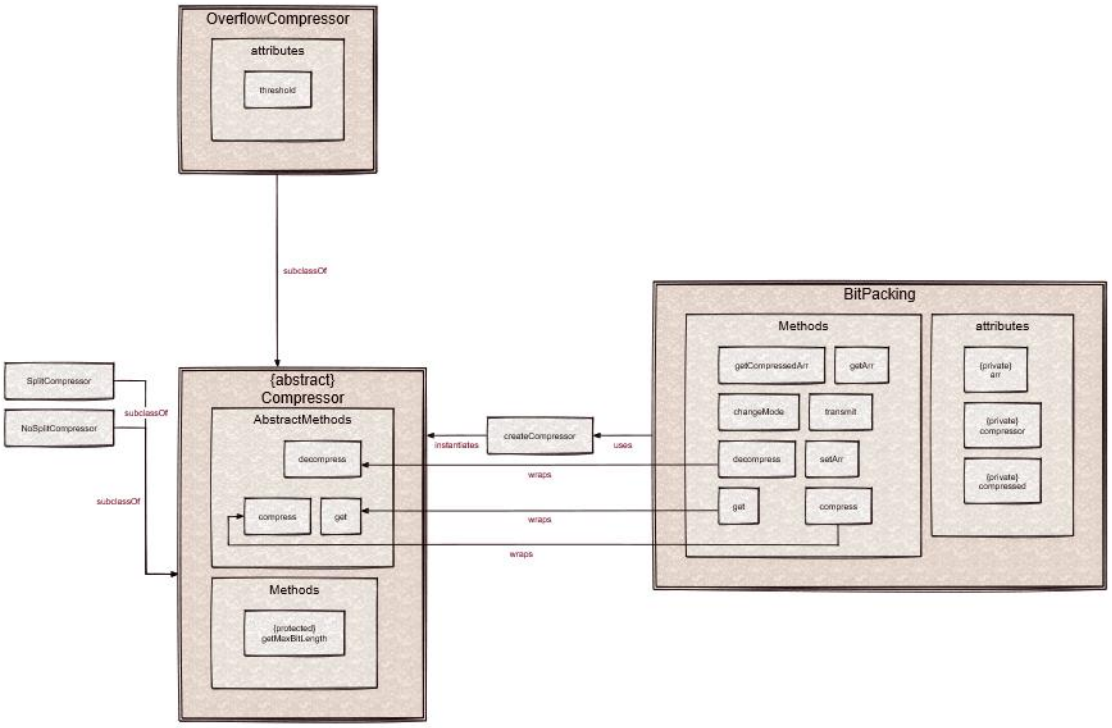
Nevertheless, the important part of the `BitPacking` class remains its wrappers around the methods presents within the `AbstractCompressor.Compressor` class. This way, we can directly execute the following methods from the `BitPacking` class without worrying of which compressor we are using:

- {wrapper} `compress() → None` which compresses the integer array we fed the `BitPacking` class with through the setter `setArr` before storing the compressed array internally,
- {wrapper} `decompress() → None` which decompresses the compressed array stored internally before overwriting the value of the private attribute `__arr`,
- {wrapper} `get(int, bool) → int` which, given a position and a boolean stating if we get the value from the compressed array or the uncompressed array, returns the value at the given position from the either compressed or uncompressed array.

Finally, a last method `transmit(bool) → None` is provided to mimic the transmission of either the compressed or uncompressed array locally. This method serialise the data write them in a buffer and read the buffer to mimic reception. This method is not used normally but can be used to measure performance on a benchmark.

2.1.3 – Organisation summarised

We can now summarise the linking of every class using this schema:



Note that, even though external libraries were used in the project, we did not mention them in the above schema. However, they are still involved, this is why, we will take a bit of this part to explain why we are using these libraries.

In this project we used 10 libraries including four externals (`NumPy`, `SciPy`, `PyTest` and `bitarray`) as well as six built-in modules (`time`, `os`, `pickle`, `typing`, `io` and `abc`). Here are the reasons which pushed us to use them:

- `NumPy`**:** Used for the inclusion of the `np.int32` type to work with 32-bit integers, inclusion of a random number generator to generate and test large array, and the availability of an array type which greatly differ to the built-in list implementation.
- `SciPy`**:** Used to generate random number following very specific distribution (Boltzmann distribution also known as truncated Planck distribution or discrete exponential distribution) to test bit packing with different contexts.
- `PyTest`**:** Widely recognised as the "default" testing solution in Python, it was used to check whether the different bit packing algorithms were well-implemented.
- `bitarray`**:** Used to optimise the memory usage of the compressed integer array (which is essentially an array of bits) compared to the standard solution in Python or `NumPy`.
- time**:** Used to benchmark our process.
- `os`**:** Used to loop through IN/ folder and perform the tests automatically.
- `pickle` and `io`**:** Used to mimic the transmission of data through a network using serialisation and buffering.

- `abc`**:** Used to implement abstract classes in Python.
- `typing`: Used for documentation and function signatures.

2.2 – "Split" implementation

In this subpart, we will present the implementation for the **"split"** version of Bit packing. As explained earlier, this compressor inherits from the abstract compressor and therefore must implements the methods `compress`, `decompress` and `get`. In the following, we will describe the implementation of each method.

The first method we will describe is `compress`. This method takes a NumPy array of integer as input and returns a tuple containing in first position the compressed array, in second position the required bit length to encode the absolute biggest value of the array and in last the length of the uncompressed array.

Pseudo-code 10: Split compression

```
1 def compress(self, arr:np.ndarray[int]) → tuple[bitarray.bitarray, int, int]:
2    maxBitLength ← self._getMaxBitLength(arr)
3    compressedArr ← 0-full bitarray of size 32 ⌈n·(maxBitLength+1)/32⌉
4    Foreach i, element in enumerate(arr) do:
5        Transform element into binary and truncate to maxBitLength
6        If element < 0 then compressedArr[i(maxBitLength + 1)] ← True else False
7        Foreach ibit, bitVal in enumerate(element) do:
8            compressedArr[i(maxBitLength + 1) + ibit + 1] ← bitVal
9    return compressedArr, maxBitLength, length(arr)
```

If we refer to the *pseudo-algorithm 5*, **step 2** (finding the maximum required bit-length) is done at line 2, then we initialise our compressed array with zeros and the particular size:

$$\mathbf{32}\left\lceil\frac{\boldsymbol{n}\cdot(\mathrm{maxBitLength}+\mathbf{1})}{\mathbf{32}}\right\rceil$$

Considering an n-sized array of b-bits encoded integer A_n^b, in this formula, $(\mathrm{maxBitLength}+1)$ corresponds to the number of bits required to represent the biggest number plus a bit to determine the sign of the compressed number. This value times the size of the original array n can be interpreted as the number of bits required to encode the whole compressed array. Divided by the number of bits to encode the uncompressed array $b = 32$ and ceiled, we obtain the required number of 32-bit integers to store the data. Multiply by $b = 32$ again and you will find the number of bits required to represent the compressed array (padding to the next multiplier of b included, **step 4**).

After that, we transform every base ten number into truncated binary number (**step 1, 3**) before inserting them in the compressed array at the right position while ensuring the encoding of the sign before inserting it.

The method ultimately returns the tuple mentioned.

Remark 11: Because of the way we choose to encode the sign, we may have conflict while we are encoding a number in 31 bits and not really 32 bits, but this can be overlooked as we can increase the number of bit to represent a number as stated in the next remark.

Remark 12: For all the compressors, a global constant can be set to move from 32-bit to other formats. By consequence, it is possible in theory to encode on 64-bit integers for example.

Now, we can move on the decompression method `decompress`. This method takes, according to its signature, a variable number of entries (`*args`), but in reality, it is just, in this case, a gimmick notation. Indeed, the real entry consists only of the tuple generated by the `compress` method. In the end, the method returns a `NumPy` array of integer.

Remark 13: Due to the `*args` notation in the signature, we are obliged to unpack the entry when passing the values to the function. For example, if `val` is the tuple generated by compress, then to decompress the array, we use `decompress(*val)` instead of `decompress(val)`, this may be misleading, but this notation avoids us to separate the information relative to the compressed array into different variables, which may become heavy syntactically.

Pseudo-code 14: Split decompression

```
1 def decompress(self, *args) → np.ndarray[int]:
2    Retrieve cArr, maxBitLength and initialLength from args
2    arr ← [] #Python list
3    i ← 0
4    While True do:
5       window ← Slice from i(maxBitLength + 1) to (i + 1)(maxBitLength + 1)
6       If window out of cArr bound or length(arr) ≥ initialLength then break
7       val ← cArr[window] #end excluded
8       elem ← '-' if val[0] else '' #sign bit
9       Concatenate elem with val[1:] #all element of val after index 1
10       arr.append(int(elem, base=2)); i++
11   return np.array(arr)
```

In this pseudo-code, we manage our decompression using a while loop and a window. The principle is quite simple, while the array is not fully decompressed (we do not have the same number of elements as in the original array), we create a window encompassing a number encoded over `maxBitLength+1` (b' plus the sign bit), decode the value in the window, update the uncompressed array, move the window by $b' + 1$ and repeat until we finish decompression.

Remark 15: The decompression, compression and get methods coded for each the compressors are quite similar to each other; however, you will find some difference in the implementation due to the development time which span over multiple days (for example, `overflow` uses a `for` loop for decompression instead of a `while` loop.)

Finally, the last method `get` takes an integer `i` indicating the position we want to look at as well as the tuple we mentioned for the decompression. For the implementation of this method, the main "challenge" was to translate the position of the user to the position in the compressed array. In the case of split, this is quite simple, the position in the compressed corresponds to the value $i(\text{maxBitLength} + 1)$.

When we have this value, we then just need to take a slice of size `maxBitLength` from this position, decode the value obtained (by considering first bit as sign bit), and finally translate it in base 10.

2.3 – "Nosplit" implementation

In this subpart, we will present the implementation for the **"nosplit"** version of Bit packing. As in previous part, we will explain the main function of the compressor.

The first function `compress` takes the same argument as earlier, but results in a different tuple output. In the case of "**nosplit"**, the bit sign is managed outside of the main compressed array in a separated bit-array. The output is therefore the same as earlier plus the sign bitarray.

Remark 16: The usage of an additional array to store the sign bits was motivated by the split-avoiding nature of this structure. Indeed, if the sign was managed within the compressed array, we would have been limited to integer of 31 bits, because 32-bit integer would have been stored (in the compressed array) on 33 bits plus (32 bits + 1 bit for the sign) resulting automatically on a cut.

Here is how the compression function works:

Pseudo-code 17: Nosplit compression

Let the following variable be:

- `capacity` $\leftarrow \left\lfloor \frac{32}{\text{maxBitLength}} \right\rfloor$: the value representing how many b'-bit encoded integer can contain a 32-bit integer.
- `criticalIndex` $\leftarrow \text{maxBitLength} \cdot \text{capacity}$: the value representing the index in a 32-bit integer at which we have to move on another integer.
- `criticalShift` $\leftarrow 32 - \text{capacity} \cdot \text{maxBitLength}$: the value representing the shift to apply in the compressed array to move on the next integer (this shift is constant through the same compression).
- `n` be the length of the original array

```
1 def compress(self, arr:np.ndarray[int]) → tuple[bitarray, bitarray, int, int]:
2    maxBitLength ← max(self._getMaxBitLength(arr), 1)
3    Initialise capacity, criticalIndex, criticalShift and n
4    compressedArr ← 0-full bitarray of size 32 ⌈n / capacity⌉
5    signArr ← 0-full bitarray of size n
6    intNo ← 0
```

```
7    Foreach i, element in enumerate(arr) do:
8        Transform element into binary and truncate to maxBitLength
9        pos ← i · maxBitLength + intNo · criticalShift
10       If pos in critical area then intNo++, pos ← pos + criticalShift
11       If element < 0 then signArr[i] ← True else False
12       Foreach ibit, bitVal in enumerate(element) do:
13           compressedArr[pos+ibit] ← bitVal
14   return compressedArr, signArr, maxBitLength, length(arr)
```

In this compression, the main goal is to adapt our index to the critical area (where no integer should be present to avoid splitting). For this, we compute a critical shift representing a value to add to a global shift every time we found ourselves in the critical area; you can find the implementation on line 9 and 10.

Apart from this, nothing really changes with respect to the split compression.

The next method we will present is the `get` function. It takes the same arguments as the split's get (with the nosplit tuple instead of the split tuple though), but finding the right position is much harder.

In this situation, we have two cases depending on the `capacity`:

- If the capacity is equal to 1, then we know each integer are stored at indices multiple of 32. In consequence, finding the integer we look for is easy and just computed as $32i$.
- If the capacity is greater than 1, then multiple integers can be stored on a single 32-bit integer. As a consequence, we need to take this change into account if we look for an integer in the middle of a 32-bit integer. In this situation we compute the position to look at as:

$$\text{pos} = \text{maxBitLength} \cdot (i \bmod \text{capacity}) + 32 \left\lfloor \frac{i}{\text{capacity}} \right\rfloor$$

  Where the first parts account for the number of bits to move within the 32-bit integer and the second part to the number of integers to move on.

When we have the starting position, we can proceed exactly as in **"split"** `get`, but by considering that the sign is in the `signArr`.

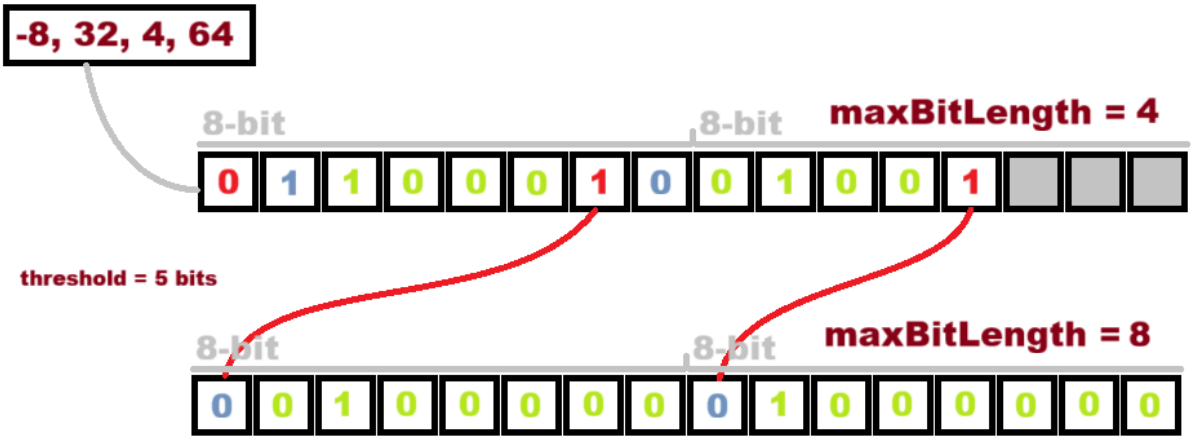
The last method to present is then `decompress`. It also takes the same argument as **"split"** `decompress`, but with the **"nosplit"** tuple. Here the method is by far the simplest: use `get` iteratively with $i \in [0, n[$ and returns the resulting array.

2.4 – "Overflow" implementation

Finally, we can present the implementation of **"overflow"** BitPacking. For this implementation, we will let the user choose, when initialising the compressor, a particular value for which an integer will be passed on the overflow area instead of being compressed in the main array.

Remark 18: Even though it was not really asked for in the specification, we primarily understood that the overflow area must have been compressed as well. As a consequence, and for time reason, the overflow area is still compressed in the final version.

In this BitPacking version, the main challenge is probably the action of considering the reference to the overflow area, for this we choose a format like this:



In this format, we have the first bit indicating whether the value is in the overflow area or not, the second bit, representing the sign, is read either on the overflow area or in the main area depending on the previous bit. Finally, the integer is written in the corresponding area.

In the example above, where integers are encoded on 8 bits, all integer greater or equal to $2^5 = 32$ is thrown in the overflow area and compressed with the maximum number of bits required to represent biggest absolute value in the overflow area and vice versa.

Considering all of this here is how the `compress` method works:

Pseudo-code 19: Overflow compression

```
1 def compress(self, arr:np.ndarray[int]) → tuple[bitarray, int, bitarray, int, int]:
2    maxBitLength ← self._getMaxBitLength([arr[arr < 2^threshold])
3    maxOBL ← self._getMaxBitLength([arr[arr ≥ 2^threshold])
3    compressedArr ← 0-full bitarray of size 32 ⌈n·(maxBitLength+2)/32⌉
4    OArr ← 0-full bitarray of size 32 ⌈n·(maxOBitLength+1)/32⌉
5    OCount ← 0
6    Foreach i, element in enumerate(arr) do:
7        Transform element into binary and truncate to maxBitLeng
8        pos ← i(maxBitLength + 2) − OCount · (maxBitLength + 1)
9        If element ≥ 2^threshold then
10           compressedArr[starPos] ← True
11           Perform splitCompression with pos = OCount · (maxOBL + 1), OArr, maxOBL
12           OCount++; continue;
13       compressedArr[starPos] ← False
```

```
    Perform splitCompression with pos = startPos + 1, compressedArr, maxBitLength
return compressedArr, maxBitLength, OArr, OMBL, length(arr)
```

As we can see, the function takes the same arguments as in other compressor (plus the threshold defined when initialising the object) but return an overflow area and the max bit length of the biggest absolute value in the overflow area.

The main challenge in this implementation was to find the offset generated by the format. This offset was found to be:

$$\text{pos} = i(\text{maxBitLength} + 2) - \text{OCount} \cdot (\text{maxBitLength} + 1)$$

Where the first half controls how far we should have placed the value in the compressed array if no overflow was present and the second half a correction applied depending on the number of overflows that occurred.

Then, the process is essentially the same as in **"split"** compression, but either performed in the compressed area or in the overflow area.

The `decompress` method is essentially the same as the compression method, but in reading instead of writing mode, the main goal is just to parse where the overflows are placed. The reading is then performed by a very similar algorithm that is present in **"split"** compressor.

Finally, the get method follows the same principles as the other get:

> **Step 1.** Find the position to look at, by iterating $\text{arrIndex} \in [0, n[$, computing the current position $\text{p} = \text{arrIndex}(\text{maxBitLength} + 2) - \text{OCount} \cdot (\text{maxBitLength} + 1)$ and increasing the overflow counter if the current position indicates overflowed integer until reaching $\text{arrIndex} = i$. This approach ensure that the integer correctly aligns with the slicer we will use in step 2.
>
> **Step 2.** Apply **"split"** `get` in either the overflow area or the compressed area at the found p.
>
> **Step 3.** Decode the value and return it as in other `get` versions.

Now that we have seen all the implementation part, we can see the tests and benchmarks.

3 – Tests and Benchmarks

In this part, we will test and benchmark our compressor using PyTest and a custom Benchmark environment that will records the time spent for each function used.

3.1 – Tests

3.2 – Benchmarks

Conclusion

...

Acknowledgement: None

Sources

[1] TITLE

Authors: *...*
Date of publication: *dd Month yyyy*
Link: *...*