# Práctica CDI - BPR5: Compresión sin pérdidas para datos de alturas sobre el nivel del mar

Compresión de Datos e Imágenes (FIB-UPC)
Cuatrimestre de primavera, 2024-2025

Sol Torralba
sol.torralba@estudiantat.upc.edu

Fernando Guirao
fernando.guirao@estudiantat.upc.edu

Nicolás Llorens
nicolas.llorens@estudiantat.upc.edu

30 de mayo de 2025

# Índice

# 1. Introducción

En esta práctica implementamos un pipeline de compresión sin pérdidas específicamente creado para ficheros que contienen secuencias de alturas sobre el nivel del mar. El objetivo es conseguir el **mayor ratio de compresión**:

$$R = \frac{\sum \text{Tamaño original}}{\sum \text{Tamaño comprimido}}$$

Se prioriza el ratio de compresión, por lo que el tiempo de compresión/descompresión no es tan relevante mientras sea manejable. El único requisito es que la velocidad de compresión/descompresión debe ser mayor que 250 kB/s.

La idea central de nuestro pipeline es aprovechar la redundancia inherente a los datos: las secuencias de alturas suelen variar poco entre elementos consecutivos, lo que se traduce en valores grandes y muy similares. Por ello, la estrategia consiste en aplicar un preprocesado que retiene únicamente la información esencial de cada altura, permitiendo así una representación más compacta y eficiente.

Una vez transformados, los datos son comprimidos mediante el algoritmo `lzma`, disponible en la librería estándar de Python. En este documento se detallan las técnicas utilizadas en el preprocesado, así como el diseño completo de nuestra solución, a la que hemos denominado **BPR5** (*Blockwise Predictive Reduction*).

# 2. Técnicas utilizadas

## 2.1. Segmentación por bloques y predicción adaptativa por fila

Para procesar los datos de manera eficiente, dividimos la matriz de alturas en **bloques de 8 filas**. Escogemos un tamaño de 8 como equilibrio práctico: bloques más pequeños pierden contexto para la predicción, y bloques más grandes incrementan el uso de memoria y la latencia.

Dentro de cada bloque, se aplica una **predicción adaptativa fila a fila**. Para cada fila, probamos cuatro predictores clásicos que estiman el valor de cada elemento a partir de su contexto inmediato:

- **LEFT**: el valor anterior en la misma fila.
- **UP**: el valor en la misma posición de la fila anterior.
- **PAETH**: una combinación no lineal entre LEFT, UP y la esquina superior izquierda (estándar en formatos como PNG).

- **MED**: la mediana entre LEFT y UP respecto a la esquina superior izquierda.

Todos estos predictores tienen coste constante $\mathcal{O}(1)$ (y lineal respecto al tamaño de la fila). Escogemos el mejor para cada fila comparando el **error absoluto total** que genera cada predictor (es decir, qué tan lejos se queda de los valores reales).

Probamos también a aplicar la predicción por columnas en lugar de por filas, pero el error total resultante fue considerablemente mayor. En un fichero de prueba, el error absoluto al predecir por columnas fue de más del doble que por filas:

$$\text{Error por filas: } 492\,175\,362 \quad \text{vs.} \quad \text{Error por columnas: } 1\,110\,570\,178$$

Por tanto, descartamos la predicción por columnas y mantenemos la predicción por filas como estrategia principal.

### Ejemplo

Por ejemplo, supongamos que tenemos:

$$\begin{aligned} \text{Fila anterior:} &\quad [99,\ 100,\ 100,\ 101] \\ \text{Fila actual:} &\quad [100,\ 101,\ 102,\ 104] \end{aligned}$$

Aplicando el predictor MED:

- En la primera posición no hay valor a la izquierda, así que usamos el de arriba: predicción $= 99 \rightarrow$ delta $= 100 - 99 = 1$
- En la segunda posición: LEFT $= 100$, UP $= 100$, DIAG $= 99 \rightarrow$ MED $= \text{med}(100, 100, 99) = 100 \rightarrow$ delta $= 101 - 100 = 1$
- En la tercera: LEFT $= 101$, UP $= 100$, DIAG $= 100 \rightarrow$ MED $= 101 \rightarrow$ delta $= 102 - 101 = 1$
- En la cuarta: LEFT $= 102$, UP $= 101$, DIAG $= 100 \rightarrow$ MED $= 101 \rightarrow$ delta $= 104 - 101 = 3$

Resultado: deltas = [1, 1, 1, 3] con errores muy bajos, lo que facilita una buena compresión posterior.

En la práctica, el predictor que más se selecciona es MED, ya que se adapta bien a variaciones suaves en ambas direcciones (horizontal y vertical). No obstante, en el código mantenemos las cuatro opciones, ya que, si bien usar únicamente MED nos ahorraría una cabecera por fila indicando el predictor empleado (1 byte por fila), este overhead es totalmente negligible respecto al tamño de los ficheros comprimidos.

Durante la descompresión, se recupera la fila original aplicando los mismos predictores a los valores ya reconstruidos, sumando cada delta a su predicción correspondiente para reconstruir los valores originales.

## 2.2. Codificación delta de primer y segundo orden

Una vez elegida la predicción para cada fila, transformamos los valores en **deltas** para reducir su magnitud y facilitar su compresión. Probamos dos tipos de codificación por fila:

- **Delta de primer orden (delta1)**: restamos al valor actual su predicción.
- **Delta de segundo orden (delta2)**: restamos al valor actual una predicción basada en los dos anteriores: $x_i - 2x_{i-1} + x_{i-2}$.

Ambas se calculan por fila, utilizando los valores ya reconstruidos (y, si hace falta, la fila anterior). Para cada fila, se computan ambas codificaciones y se selecciona la que produce el menor **error absoluto total**.

Esto permite capturar tanto variaciones suaves (mejor con delta1) como patrones más regulares o lineales (mejor con delta2). Durante la descompresión, la información sobre qué modo se usó (1 byte por fila) nos permite invertir el proceso de forma exacta.

## 2.3. Transformación ZigZag y codificación variable-length (varint)

Los deltas que obtenemos pueden ser negativos o positivos. Para representarlos de forma eficiente como enteros sin signo (requisito para la codificación variable), aplicamos la **transformación ZigZag**, que convierte enteros con signo en enteros sin signo de forma compacta:

$$0 \to 0,\ -1 \to 1,\ 1 \to 2,\ -2 \to 3,\ \ldots$$

Esto hace que valores pequeños (positivos o negativos) se codifiquen en valores pequeños, lo que favorece la siguiente etapa: la **codificación variable-length** (varint), donde cada entero se representa con un número de bytes proporcional a su magnitud.

Esta técnica permite que la mayoría de deltas, que suelen ser cercanos a cero, ocupen solo 1 byte. Así se consigue una representación mucho más compacta que usar 4 bytes por entero (como haría una codificación fija).

## 2.4. Compresión final de bloques con LZMA

Una vez preprocesados los datos (predicción, delta, ZigZag y varint), agrupamos las filas por bloques y comprimimos cada bloque por separado usando el algoritmo **LZMA** (preset 9 con la opción `EXTREME`), disponible en la librería estándar de Python.

LZMA es un algoritmo de compresión general basado en diccionarios y codificación por rangos. Detecta patrones repetidos en los datos y los reemplaza por referencias más cortas, lo que lo hace muy eficiente cuando los datos han sido previamente transformados para reducir su entropía.

Cada bloque comprimido se almacena junto a su **longitud**, codificada con varint. Esto permite descomprimir cada bloque de forma independiente, sin necesidad de conocer el tamaño del resto.

# 3. Estructura del código fuente

El fichero `compress_debug.py` implementa todo el pipeline de compresión y descompresión. A continuación, se indica qué funciones intervienen en cada etapa del proceso:

- **Lectura y preparación de los datos:**
  - `compress_file`: carga y organiza las filas del fichero original.
- **Selección del predictor por fila:**
  - `select_predictor_for_row`: evalúa LEFT, UP, PAETH y MED, y selecciona el que minimiza el error absoluto.
- **Cálculo de deltas:**
  - `delta1`, `delta2`: calculan las diferencias de primer y segundo orden respecto al predictor seleccionado.
- **Codificación de los deltas:**
  - `zigzag_encode`, `write_varint`: transforman los deltas a enteros sin signo y los codifican en formato variable.
- **Compresión por bloques:**
  - `lzma.compress`: se utiliza directamente en `compress_file` para comprimir los bloques codificados.
- **Escritura del fichero comprimido:**
  - `compress_file`: construye la cabecera (predictores, modos, tamaños) y escribe los bloques comprimidos.
- **Descompresión y reconstrucción:**
  - `decompress_file`: decodifica los bloques, aplica la predicción y los deltas para reconstruir las filas originales.
  - `zigzag_decode`, `read_varint`: funciones auxiliares para decodificar los enteros.
- **Entrada por línea de comandos:**
  - `main`: gestiona los argumentos, detecta si se va a comprimir o descomprimir, mide el tiempo y calcula el ratio.

# 4. Uso del ejecutable

El ejecutable se puede invocar desde línea de comandos con la siguiente sintaxis:

```
$ python3 compress.py infile outfile [--verify]
```

- Para **comprimir**, basta con proporcionar el fichero de entrada y el de salida:

  ```
  $ python3 compress.py datos.txt comprimido.bin
  ```

- Para **descomprimir**, el programa detecta automáticamente si el fichero ya está comprimido y realiza la descompresión:

  ```
  $ python3 compress.py comprimido.bin recuperado.txt
  ```

- El flag opcional `--verify` permite comprobar que la descompresión es correcta:

  ```
  $ python3 compress.py datos.txt comprimido.bin --verify
  ```

  Esto genera temporalmente un fichero descomprimido y lo compara con el original. Si coinciden, se imprime un mensaje de verificación correcta.

# 5. Formato del fichero comprimido

El fichero comprimido sigue una estructura binaria personalizada, compuesta por:

- **Cabecera:**
  - `MAGIC` (4 bytes): firma del fichero (`BPR5`).
  - Número de filas (varint).
  - Longitud de cada fila (varint por fila).
  - Primer valor original (codificado con ZigZag + varint).
  - Lista de predictores usados (1 byte por fila).
  - Lista de modos delta (1 byte por fila).
- **Cuerpo:**
  - Para cada bloque:
    - Tamaño del bloque comprimido (varint).
    - Datos comprimidos con LZMA (bloque de 8 filas).

# 6. Resultados y conclusiones

Para poner a prueba el compresor, utilizamos cuatro ficheros diferentes. A continuación se muestran los mejores ratios de compresión obtenidos en cada uno de ellos:

| **Fichero** | **Ratio de compresión** |
|---|---|
| `file11111.txt` | 8.44x |
| `file22222.txt` | 10.18x |
| `file21212.txt` | 9.12x |
| `file22121.txt` | 10.25x |

Estamos satisfechos con estos resultados, ya que los ratios son notablemente superiores a los obtenidos con compresores genéricos como `zip`. Uno de los aprendizajes clave de esta práctica ha sido la importancia del **preprocesamiento**, especialmente la etapa de predicción. Sin esta transformación previa, los ratios observados se reducen casi a la mitad y los tiempos de compresión aumentan considerablemente.

También hemos experimentado con predictores más sofisticados, como regresión lineal o pequeños modelos MLP (*tiny neural networks*) entrenados para predecir el siguiente valor de la fila. Sin embargo, incluso en esos casos, el predictor **MED** seguía ofreciendo mejores resultados en la mayoría de situaciones. Es posible que, ajustando mejor estas técnicas más avanzadas, se puedan superar los resultados actuales, pero hay que tener en cuenta que el coste computacional se incrementa rápidamente, lo que puede hacer que la compresión sea inviable en ordenadores domésticos o en entornos con recursos limitados.