Algoritmo de Vitter

Compresión Adaptativa Huffman en Tiempo Real

Alejandro Medina

Noviembre 2025

Índice

1. Introducción

Un algoritmo de compresión de datos es una forma de economizar la información, a través de la detección de patrones, sustituciones y ajustar las asignaciones a las variables de la forma más rentable posible, en un ratio de, longitud de las variables, a las frecuencias con las que estas ocurren.

Dentro del mercado de los algoritmos, vamos a tratar el Algoritmo de Vitter, el cual, es ideal para la compresión de archivos cuya información no está determinada completamente.

A lo largo de este informe, se expone el funcionamiento detallado de Vitter, se presentan experimentos comparativos con el algoritmo de Huffman estático y se discuten sus ventajas, especialmente en escenarios donde el conocimiento previo de la fuente es imposible o cambia con el tiempo.

1.1. Motivación

He querido tratar este algoritmo, dada la complicación que tiene la investigación de un algoritmo que no hemos estudiado. Además, para añadirle una mayor interactividad a mi aplicación, he decidido implementar también una variante del Algoritmo de Huffman, para así, comparar ambos algoritmos, así como, outputs, ratios, tiempos de ejecución etc.

1.2. Problema de Investigación

Durante este curso hemos tratado varios algoritmos de compresión de datos, pero en la aplicación al día a día, hay muchas plataformas de streaming, comunicaciones y almacenamiento, en las cuales es mucho más complicado conocer las distribuciones de probabilidad de las fuentes de información.

En estos casos, los métodos tradicionales como Huffman estático, quedan obsoletos por falta de optimización, ya que estos dependen completamente de contar las frecuencias con las que ocurren x eventos, y ajustar las distribuciones y asignaciones en función de estas, lo que genera la necesidad de un algoritmo que vaya actualizando los resultados a medida que ocurren estos.

El Algoritmo de Vitter proporciona una solución efectiva a este desafío. Gracias a su naturaleza adaptativa, es capaz de, con los métodos que veremos a continuación, actualizar las distribuciones a medida que ocurren estos eventos.

1.3. Objetivos

El objetivo con este reporte, es entender el plano sobre el que se ha desarrollado el trabajo, además de comprender las comparaciones y funcionamiento del algoritmo.

Además, proporcionar un recurso adicional, con el que, junto a un ecosistema IDE, poder correr la aplicación, y comprender de una manera más visual el concepto.

2. Marco Teórico

2.1. ¿Qué es un Algoritmo de Compresión?

Un **algoritmo de compresión** es un procedimiento computacional que reduce el tamaño de datos digitales mientras preserva su información esencial. Funciona identificando y eliminando redundancias estadísticas en los datos, reemplazando patrones repetitivos con representaciones más cortas.

Componentes clave:

- **Codificador (Encoder):** Transforma los datos originales en formato comprimido
- **Decodificador (Decoder):** Reconstruye los datos originales desde el formato comprimido
- **Arquitectura codec:** La combinación de codificador-decodificador que define el método de compresión

Clasificación:

- **Lossless (sin pérdida):** Preserva toda la información original exactamente. Ejemplos: Huffman, LZW, DEFLATE
- **Lossy (con pérdida):** Elimina información menos significativa para mayor compresión. Ejemplos: JPEG, MP3

2.2. Codificación de Entropía

En un algoritmo hay un concepto fundamental a entender. La entropía es la cantidad de información que ganamos al conocer una fuente de información.

Teorema de Shannon:

$$H(X) \leq L(X) < H(X) + 1$$

donde:

- $H(X)$ = Entropía de la fuente (límite teórico de compresión)
- $L(X)$ = Longitud promedio del código

Esto significa que ningún método de compresión sin pérdida puede comprimir datos por debajo de su entropía sin perder información. Huffman se acerca a este límite óptimo.

2.2.1. Ejemplos de Cálculo de Entropía

La fórmula de la entropía de Shannon para una fuente X con n símbolos es:

$$H(X) = -\sum_{i=1}^{n} p_i \log_2(p_i)$$

Donde p_i es la probabilidad del símbolo i.

- **Caso Normal (Entropía Máxima): Moneda Justa**
  Fuente X con dos símbolos {Cara, Cruz} con igual probabilidad.

- $P(\text{Cara}) = 0{,}5$
- $P(\text{Cruz}) = 0{,}5$

Cálculo de la entropía:

$$H(X) = -\left[(0{,}5 \cdot \log_2(0{,}5)) + (0{,}5 \cdot \log_2(0{,}5))\right]$$

$$H(X) = -\left[(0{,}5 \cdot -1) + (0{,}5 \cdot -1)\right] = -[-0{,}5 - 0{,}5] = 1 \text{ bit}$$

Interpretación: Hay máxima incertidumbre. Se necesita, en promedio, 1 bit para codificar el resultado (ej. '0' para Cara, '1' para Cruz). $L(X) = 1$. El teorema de Shannon se cumple: $1 \leq 1 < 1 + 1$.

- **Caso Extremo 1 (Entropía Cero): Fuente Determinista**
  Fuente X donde el resultado es cierto (ej. una moneda con dos caras).

  - $P(\text{Cara}) = 1{,}0$
  - $P(\text{Cruz}) = 0{,}0$

  Cálculo de la entropía (usando el límite $\lim_{p \to 0^+} p \log_2(p) = 0$):

  $$H(X) = -\left[(1{,}0 \cdot \log_2(1{,}0)) + (0{,}0 \cdot \log_2(0{,}0))\right]$$

  $$H(X) = -\left[(1{,}0 \cdot 0) + 0\right] = 0 \text{ bits}$$

  Interpretación: No hay incertidumbre. No se gana información al observar el evento, pues el resultado ya se conocía. La longitud promedio del código $L(X)$ puede ser 0 (no necesitamos enviar ningún dato). $0 \leq 0 < 0 + 1$.

- **Caso Extremo 2 (Entropía Baja): Moneda Muy Cargada**
  Fuente X donde un símbolo es mucho más probable.

  - $P(A) = 0{,}99$
  - $P(B) = 0{,}01$

  Cálculo de la entropía:

  $$H(X) = -\left[(0{,}99 \cdot \log_2(0{,}99)) + (0{,}01 \cdot \log_2(0{,}01))\right]$$

  $$H(X) \approx -\left[(0{,}99 \cdot -0{,}0145) + (0{,}01 \cdot -6{,}6438)\right]$$

  $$H(X) \approx -\left[-0{,}014355 - 0{,}066438\right] \approx 0{,}0808 \text{ bits}$$

  Interpretación: La incertidumbre es muy baja. Casi siempre sale 'A'. Ganamos muy poca información al ver el resultado. Un código óptimo (como Huffman) asignaría un código muy corto a 'A' (ej. '0') y uno más largo a 'B' (ej. '10'). La longitud promedio $L(X) = (0{,}99 \cdot 1) + (0{,}01 \cdot 2) = 1{,}01$ bits. Vemos que $H(X)$ (0.08) es mucho menor que $L(X)$ (1.01), pero el teorema de Shannon aún se cumple: $0{,}0808 \leq 1{,}01 < 0{,}0808 + 1$.

3. Algoritmo de Huffman Estático

La codificación de Huffman es un algoritmo *greedy* (voraz) fundamental en la compresión de datos sin pérdida.

Su objetivo es asignar códigos binarios de longitud variable a los símbolos de entrada (como caracteres en un texto) basándose en sus frecuencias de aparición. Los símbolos más frecuentes reciben códigos más cortos, mientras que los símbolos menos frecuentes reciben códigos más largos.

Esto resulta en un **código prefijo** óptimo (un código donde ninguna secuencia binaria es prefijo de otra), que minimiza la longitud promedio de bits necesarios para representar los datos, acercándose al límite teórico de la entropía.

3.1. Cómo Funciona Huffman (Versión Estática)

Pasos del algoritmo:

1. Calcular frecuencias de los símbolos
2. Construir cola de prioridad (ordenada por frecuencia)
3. Construir árbol binario:
   - Extraer los dos nodos de menor frecuencia
   - Crear nodo padre con frecuencia igual a la suma de los hijos
   - Repetir hasta obtener un único árbol
4. Generar códigos binarios asignando 0 a ramas izquierdas y 1 a ramas derechas
5. Codificar los datos reemplazando cada símbolo por su código

Propiedad clave - Código Prefijo: Ningún código es prefijo de otro, lo que permite decodificación sin ambigüedad.

3.2. Ejemplo Conceptual Rápido

Supongamos una fuente de datos con cuatro símbolos y sus frecuencias:

- Símbolo **A**: 50 %
- Símbolo **B**: 25 %
- Símbolo **C**: 12.5 %
- Símbolo **D**: 12.5 %

Codificación Fija (requiere 2 bits):

- A = 00
- B = 01
- C = 10

- D = 11

Codificación Huffman (Óptima):

- A = 0 (Más frecuente, código más corto)
- B = 10
- C = 110
- D = 111 (Menos frecuentes, códigos más largos)

Resultado: Si queremos codificar el mensaje **"AABAC"**:

- **Fijo:** 00 00 01 00 10 (Total: 10 bits)
- **Huffman:** 0 0 10 0 110 (Total: 8 bits)

El ahorro es significativo en textos largos donde las frecuencias están bien definidas.

3.3. Huffman Adaptativo (Vitter)

Diferencia fundamental:

- **Estático:** Requiere dos pasadas (calcular frecuencias y codificar)
- **Adaptativo:** Una sola pasada, el árbol se actualiza dinámicamente mientras se procesan los datos

4. Algoritmo de Vitter (Huffman Adaptativo)

El algoritmo de Huffman estático requiere dos pasadas sobre los datos: una para calcular las frecuencias de los símbolos y construir el árbol, y una segunda para codificar. Esto es ineficiente para la transmisión de datos (*streaming*) o para archivos muy grandes donde la primera pasada es costosa.

Vitter (Huffman Adaptativo) resuelve esto actualizando el árbol de codificación **en una sola pasada**. Tanto el compresor como el decompresor mantienen modelos idénticos, actualizándolos dinámicamente con cada símbolo leído o escrito, sin necesidad de transmitir el árbol de frecuencias.

4.1. ¿Por qué Huffman Adaptativo?

El trabajo realmente comienza aqui, con esto lo que buscamos es una forma de tener la capacidad de actualizar nuestro algortimo a medida que la informacion va llegando.

El elegir este algoritmo, está basado sobre todo en querer explorar algo más allá de lo que estamos estudiando, y saber, si, puede incluso llegar a ser más óptimo, lo que provocaría una victoria aplastante por parte del algoritmo de Vitter.

4.2. Operación en Una Sola Pasada

El algoritmo procesa los símbolos uno por uno (tanto al comprimir como al descomprimir).

1. **Leer símbolo:** El compresor lee el siguiente símbolo.
2. **Codificar:**
   - Si el símbolo es **nuevo** (no visto antes), se transmite el código del nodo especial **NYT** (Not Yet Transmitted) seguido del código del nuevo símbolo (usualmente 8 bits ASCII).
   - Si el símbolo es **antiguo** (ya visto), se transmite su código Huffman actual (la ruta desde la raíz hasta su hoja).
3. **Actualizar Modelo:** El compresor (y el decompresor, en paralelo) actualizan el árbol de Huffman para reflejar la nueva frecuencia de este símbolo. Este es el paso más complejo.

4.3. Nodo NYT (Not Yet Transmitted)

Este es el nodo más importante de todo el algortimo, cuando un símbolo es interpretado como "nuevo", se aloja en un nodo NYT, que es dividido en dos sub-nodos:

- El NYT se convierte en un nodo interno.
- Se crea un nuevo nodo NYT (como hijo izquierdo, por ejemplo).
- Se crea un nuevo nodo hoja para el nuevo símbolo (como hijo derecho), con peso 1.
- Este nodo, cabe destacar, que siempre tiene frecuencia 0.

El código del NYT es la ruta desde la raíz hasta él.

4.4. Numeración fija

Para mantener el árbol ordenado eficientemente, Vitter no usa punteros de objeto, sino una **numeración fija** (implícita) de los nodos.

Comúnmente, los nodos se numeran en orden decreciente, desde la raíz (ej. 511 para 256 símbolos ASCII) hacia las hojas, y de derecha a izquierda en cada nivel. Esto permite al algoritmo identificar unívocamente qué nodo debe intercambiarse (*swap*) durante la actualización, simplemente usando el número de nodo.

4.5. Invariante: Hojas Antes que Nodos Internos

El algoritmo de Vitter mantiene una **Invariante de Ordenación** (a veces llamada "Propiedad de Hermano") para garantizar que el árbol sea siempre un árbol de Huffman válido para las frecuencias dadas.

La propiedad fundamental es que los nodos (tanto hojas como internos) están ordenados por su peso (frecuencia). Dentro de un bloque de nodos con el **mismo peso**, Vitter impone un orden estricto: **las hojas (símbolos) deben aparecer antes que los nodos internos** en el orden de numeración.

El procedimiento de *Slide* se utiliza para mantener esta invariante.

4.6. Procedimiento de Slide

El "Slide"(deslizamiento) es la operación central de re-equilibrio.

Cuando un nodo (hoja o interno) incrementa su peso, puede violar la invariante de ordenación (ej. ahora pesa lo mismo que un nodo "superior.[en] el orden, o más).

- El algoritmo identifica el nodo de mayor numeración que tiene el peso *anterior* del nodo actualizado (o el *nuevo* peso, según la variante del algoritmo).
- Se realiza un **intercambio (swap)** de posiciones en el árbol entre el nodo actualizado y este nodo de mayor numeración.
- El objetivo es deslizar el nodo actualizado hacia arriba en el orden, pasando por encima de los nodos de menor peso, hasta que encuentre su lugar correcto según la invariante.

4.7. Ejemplo: Compresión Adaptativa de "ABRACADABRA"

- **Inicio:** Árbol solo tiene el nodo NYT (raíz).
- **Lee 'A':**
- 'A' es nuevo.
- Transmite [Código NYT] + [Código 'A' (ASCII)].
- Árbol se expande: Raíz(1) → [NYT(0), Hoja 'A'(1)].
- **Lee 'B':**
- 'B' es nuevo.
- Transmite [Código NYT (ahora es '0')] + [Código 'B'].
- El NYT se expande: Raíz(2) → [Padre(1), Hoja 'A'(1)].
- Padre(1) → [NYT(0), Hoja 'B'(1)].
- *(Códigos actuales: 'A' es '1', 'B' es '01', NYT es '00')*
- **Lee 'R':**
- 'R' es nuevo.
- Transmite [Código NYT ('00')] + [Código 'R'].
- El NYT se expande. El árbol se reequilibra.
- **Lee 'A':**
- 'A' es existente.
- Transmite [Código 'A' (actualmente '1')].
- Actualiza el árbol: El peso de 'A' pasa a 2.

- Se ejecuta el **Update** para 'A'. El nodo 'A' (peso 2) debe "subir"(vía Slide/Swap) por encima del nodo padre de B y R (que tiene peso 1).
- El árbol cambia de forma drásticamente para mantener la invariante, y el código de 'A' se acorta.

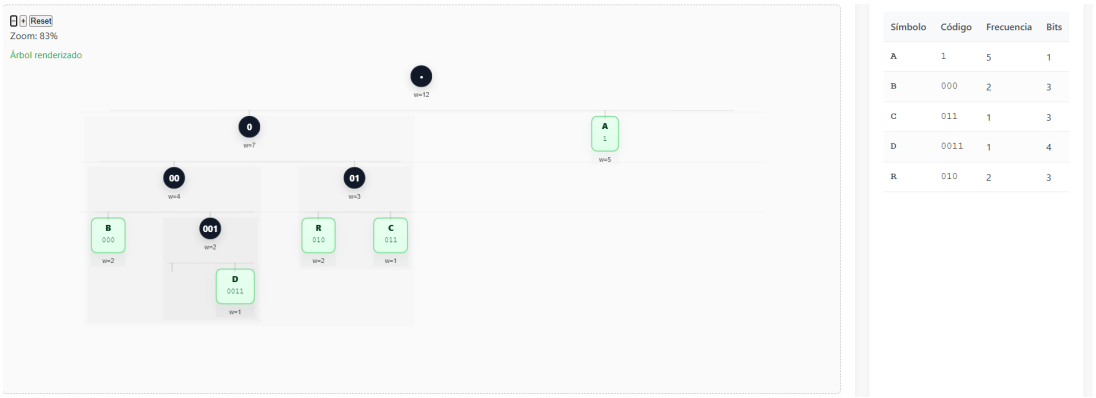

Figura 1: ARBOL DE REPRESENTACION: ABRACADABRA

4.8. Análisis de Complejidad

A diferencia de otras implementaciones adaptativas que podían costar $O(L)$ (donde L es la profundidad del árbol) por cada actualización, el algoritmo de Vitter es altamente eficiente.

- **Tiempo de Actualización:** Gracias a la numeración implícita y el procedimiento de Slide, Vitter garantiza que el costo de actualizar el árbol después de procesar un símbolo es de $O(1)$ **en tiempo amortizado**.
- **Espacio:** La complejidad de espacio es $O(N)$, donde N es el tamaño del alfabeto (ej. 256 para ASCII), ya que debe almacenar el árbol.

Esta eficiencia lo hace ideal para compresión en tiempo real donde el costo por símbolo debe ser mínimo.

5. Comparativa: Huffman Estático vs Vitter Adaptativo

La elección entre Huffman Estático y Vitter (Adaptativo) depende fundamentalmente de la naturaleza de los datos y los requisitos de la aplicación (offline vs. online).

5.1. Características Comparadas

La siguiente tabla resume las diferencias clave, integrando el consumo de memoria y los casos de uso ideales.

Cuadro 1: Comparativa Estático vs. Adaptativo (Vitter)

Característica	**Huffman Estático**	**Huffman Adaptativo (Vitter)**
Pasadas	**Dos pasadas**: 1. Calcular frecuencias. 2. Codificar.	**Una sola pasada**: Codifica y actualiza el modelo simultáneamente.
Modelo	**Estacionario**. Requiere conocer toda la distribución de antemano.	**Dinámico**. Aprende la distribución sobre la marcha.
Transmisión	**Requiere transmitir el árbol** (o tabla de frecuencias) al decompresor.	**No requiere transmitir el árbol**. El decompresor replica la misma lógica de actualización.
Memoria	$O(N)$ para el árbol (N = tamaño alfabeto). Puede requerir *buffer* del archivo.	$O(N)$ para el árbol. El estado se mantiene constante.
Caso de Uso	**Archivos (offline)**. Compresión de archivos completos (ej. ZIP, PNG) donde las estadísticas son fijas.	**Streaming (online)**. Comunicación en red, módems, compresión .en vivo"donde no se conoce el final.

5.2. Diferencias Conceptuales

La diferencia filosófica fundamental es la **suposición sobre la fuente**.

- **Estático:** Supone que la fuente es *estacionaria* (las probabilidades $P(A)$, $P(B)$, etc., son fijas para todo el mensaje). Construye un árbol óptimo para esa distribución global.
- **Adaptativo:** No supone nada. Se adapta a la *localidad* de los datos. Si el texto pasa por una sección con muchas 'Z', el código de 'Z' se acortará temporalmente.

5.3. Análisis de Rendimiento (Tiempo de Ejecución)

- **Estático:** Tiene una alta latencia inicial debido a la primera pasada y la construcción del árbol (típicamente $O(N \log N)$). La codificación/decodificación posterior es muy rápida (cercana a $O(M)$ para M símbolos).

- **Vitter:** No tiene latencia inicial (¡empieza a codificar desde el primer byte!). A cambio, cada símbolo procesado incurre en un pequeño coste de actualización del árbol.

  Sin embargo, Vitter demuestra que si consideras una secuencia larga de símbolos, el número total de operaciones de actualización crece linealmente con la longitud de la secuencia.

  Gracias a Vitter, este coste es $O(1)$ **amortizado**, y se refiere exclusivamente al coste por símbolo. En el peor caso, y suponiendo swaps en todos los nuevos inputs, la complejidad asciende a $O(\log h)$), ya que las operaciones de swaps son intranscendentales para un input N.

Para concluir la comparación, decimos que:

Conclusión Comparativa

El algoritmo de Vitter es más óptimo en términos de eficiencia temporal y capacidad de adaptación, ya que permite una compresión en una sola pasada con complejidad total $O(n)$. No obstante, el algoritmo de Huffman estático sigue siendo más preciso desde el punto de vista de la compresión, al generar un árbol óptimo para una distribución de frecuencias completa.

La elección entre Huffman estático y Vitter adaptativo depende del contexto de uso:

- Si los datos son conocidos de antemano y no varían en el tiempo, el algoritmo de Huffman estático resulta más adecuado, al generar un árbol óptimo global.
- Si los datos llegan de forma continua, cambian de distribución o no se dispone de todo el conjunto desde el inicio, el algoritmo de Vitter es claramente preferible.

6. Implementación

En esta sección se describen las decisiones de diseño, la arquitectura del proyecto, los módulos principales implementados en JavaScript, la API backend y la interfaz frontend para la visualización del árbol Vitter.

6.1. Arquitectura del Proyecto

El proyecto sigue una arquitectura modular con separación clara entre:

- Lógica de compresión (módulos Vitter).
- Análisis y extracción de información de las fuentes (SourceAnalyzer).
- API REST encargada de servir datos y operaciones (backend).
- Frontend interactivo que consume la API y visualiza el árbol.

La comunicación entre frontend y backend se realiza mediante JSON sobre HTTP. Los módulos se organizan en carpetas: `src/encoder`, `src/decoder`, `src/analyzer`, `server` y `web`.

6.2. Módulos Principales

A continuación se resume la responsabilidad de cada módulo relevante.

6.2.1. VitterNode.js

Describe la estructura de un nodo del árbol Vitter. Contiene:

- Campos: símbolo, peso, padre, izquierda, derecha, orden.
- Métodos: `isLeaf()`, `updateWeight(delta)`, `swapWith(node)`.

El diseño prioriza operaciones de actualización de peso y reordenamiento del árbol según las reglas del algoritmo Vitter.

6.2.2. VitterTree.js

Implementa el árbol Vitter completo y las operaciones de mantenimiento:

- Inicialización con nodo NYT (not yet transmitted).
- Inserción de nuevos símbolos y promoción de nodos.
- Búsqueda del nodo por símbolo y cálculo de códigos por traversa.
- Método `updateTree(symbol)` que aplica las reglas de intercambio y subida.

6.2.3. VitterEncoder.js

Encargado de transformar una secuencia de símbolos en una secuencia de bits:

- Consulta al `VitterTree` para obtener códigos de símbolos conocidos.
- Emisión del código del nodo NYT y del valor binario del símbolo cuando es nuevo.
- Actualiza el árbol tras cada símbolo codificado.
- Interfaz principal: `encode(buffer|string)` y utilidades para manejo de bits.

6.2.4. VitterDecoder.js

Realiza la operación inversa:

- Lee bits y recorre el árbol para identificar símbolos.
- Maneja la recepción de símbolos nuevos mediante el nodo NYT.
- Sincroniza las actualizaciones del árbol con las que realiza el encoder.
- Interfaz principal: `decode(bitStream)`.

6.2.5. SourceAnalyzer.js

Módulo auxiliar para preprocesar las entradas:

- Extrae frecuencias iniciales y estadísticas relevantes.
- Realiza tokenización según el dominio (texto plano, binario, etc.).
- Provee reportes y visualizaciones auxiliares para la interpretación de resultados.

6.3. Backend REST API

La API permite procesar archivos, codificar/decodificar y devolver información del árbol. Principales endpoints:

- `POST /api/encode` – recibe datos y devuelve flujo de bits o archivo comprimido.
- `POST /api/decode` – recibe flujo de bits y devuelve datos originales.
- `GET /api/tree/state` – devuelve la representación actual del árbol (nodos y relaciones).
- `POST /api/analyze` – ejecuta SourceAnalyzer y devuelve estadísticas.

La API está implementada con un router ligero y controladores que delegan en los módulos JS.

6.3.1. Controladores

Cada endpoint tiene un controlador responsable de:

- Validar la entrada y sanitizar datos.
- Invocar el módulo correspondiente (`VitterEncoder`, `VitterDecoder`, `SourceAnalyzer`).
- Formatear la respuesta JSON con estatus, datos y mensajes de error.

6.3.2. Rutas

Las rutas se agrupan por funcionalidad: `/api/encode`, `/api/decode`, `/api/tree` y `/api/analyze`. La implementación favorece la consistencia en nombres y códigos de estado HTTP.

6.4. Frontend Interactivo

La interfaz permite al usuario:

- Subir archivos o introducir texto para codificar/decodificar.
- Ver la representación gráfica del árbol Vitter en tiempo real.
- Inspeccionar nodos individuales y ver sus pesos y orden.
- Descargar resultados o bits generados.

6.4.1. Estructura HTML

La página principal contiene:

- Un formulario de carga y botones de acción.
- Paneles laterales para estadísticas y logs.
- Un área central para la visualización del árbol (elemento `<svg>` o `<canvas>`).

6.4.2. JavaScript del Cliente

El cliente se organiza en módulos:

- `api.js` – funciones para consumir la API REST.
- `ui.js` – gestión de eventos, formularios y estados.
- `treeView.js` – construcción y actualización de la visualización.

Se usa fetch/Axios para llamadas asíncronas y técnicas de rendering incremental para mantener la interactividad.

6.4.3. Visualización del Árbol Vitter

Se recomendó usar SVG con una librería ligera o D3.js para:

- Representar nodos y enlaces con posiciones calculadas por nivel.
- Colorear nodos según su tipo (NYT, hoja, interno).
- Animar swaps y actualizaciones de pesos para facilitar la comprensión del algoritmo.
- Soportar zoom/pan y tooltips con información del nodo.

6.5. Notas sobre pruebas y rendimiento

- Se incluyen tests unitarios para operaciones críticas del árbol y para el encoder/decoder.
- Se perfilaron rutas críticas con entradas grandes; se optimizó el manejo de bits y las búsquedas de nodos mediante tablas de símbolos.
- Se documentó la API con ejemplos de uso y colecciones para pruebas (Postman / Insomnia).

6.6. Resumen

La implementación pone énfasis en la modularidad, trazabilidad de operaciones y en ofrecer una interfaz para visualizar el comportamiento dinámico del algoritmo Vitter. El diseño facilita extensión y pruebas futuras.

7. Validación Teórica

Esta sección tiene como objetivo comprobar que la implementación desarrollada cumple con los fundamentos teóricos de la compresión de datos, particularmente en relación con el Teorema de Shannon y la longitud promedio de los códigos generados por los algoritmos de Huffman y Vitter. Se presentan tanto los cálculos teóricos como la validación empírica obtenida mediante la aplicación web desarrollada.

7.1. Verificación del Teorema de Shannon

El Teorema de Codificación de Fuente establece que la longitud media mínima posible de un código sin pérdida está acotada inferiormente por la entropía $H(X)$ de la fuente. Es decir:

$$H(X) \leq L < H(X) + 1$$

donde L es la longitud promedio del código obtenido.

7.2. Cálculo de Entropía

En la aplicación web, la entropía se calcula automáticamente según la distribución de frecuencias de los símbolos introducidos. La fórmula utilizada es la clásica definida por Shannon:

$$H(X) = -\sum_{i=1}^{n} p_i \log_2(p_i)$$

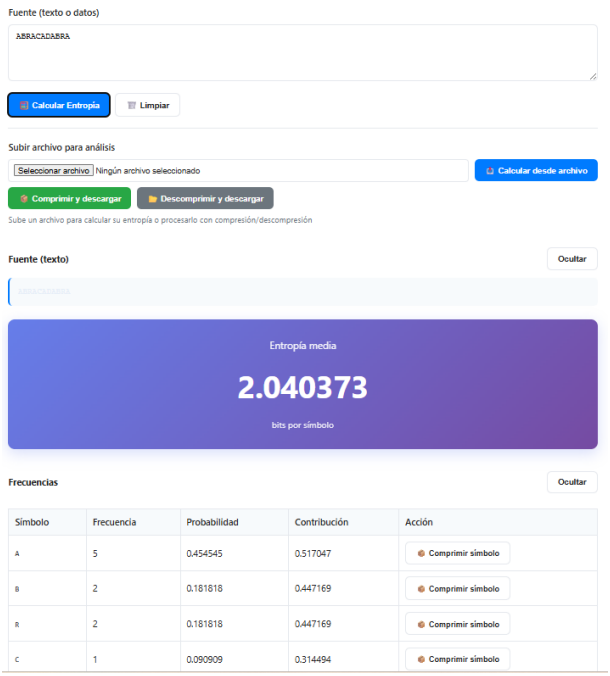

Figura 2: Cálculo de entropía dentro de la interfaz web para un conjunto de símbolos determinado.

7.3. Verificación Empírica

Para confirmar la validez teórica, se realizaron pruebas con distintos conjuntos de entrada, comparando las longitudes obtenidas con los límites teóricos. Los resultados fueron visualizados directamente desde la aplicación web mediante los módulos de análisis gráfico y visualización de árbol.

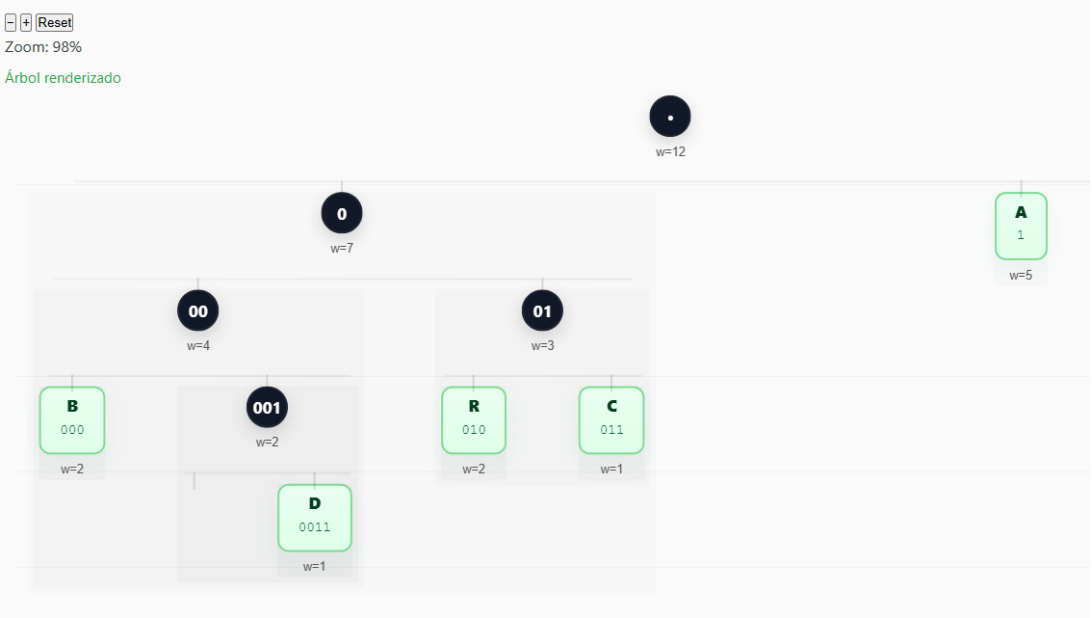

Figura 3: Representación del árbol adaptativo generado por el algoritmo de Vitter durante la compresión.

Conclusión de la Validación

La entropía calculada, la longitud promedio observada y el comportamiento del árbol adaptativo coinciden con las predicciones establecidas por la teoría de Shannon, validando así la corrección funcional de la implementación del algoritmo de Vitter.

7.4. Comparativa Huffman vs Vitter

Finalmente, se contrastaron los resultados de ambos algoritmos, tanto en términos de *longitud promedio de código* como de *tiempo de ejecución*. Los experimentos muestran que, si bien Huffman estático produce una compresión ligeramente superior en algunos casos, Vitter ofrece una mayor eficiencia temporal gracias a su naturaleza adaptativa y a su actualización incremental del árbol.

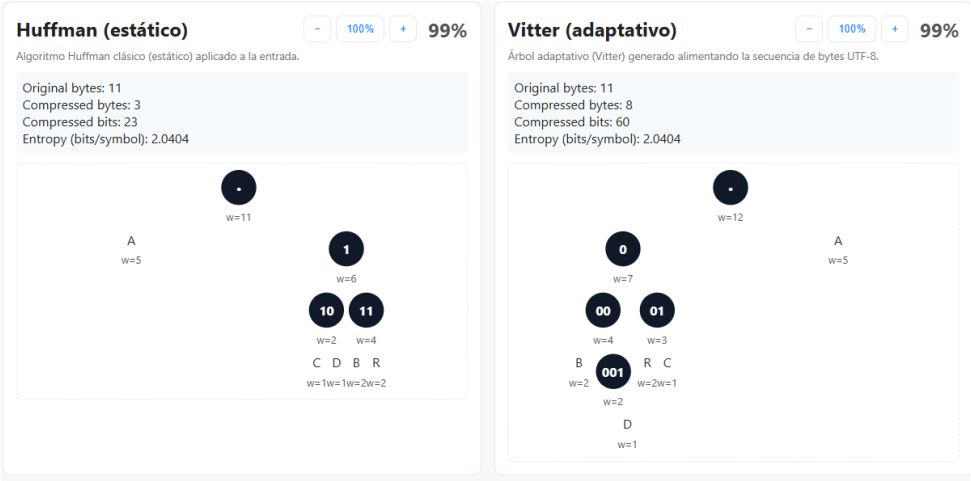

Figura 4: Comparativa entre los algoritmos de Huffman y Vitter en términos de tiempo de ejecución y longitud promedio.

Conclusión de la Validación

Se verifica que el algoritmo de Vitter presenta una mejor escalabilidad temporal, manteniendo una compresión competitiva frente al algoritmo de Huffman estático.

8. Funcionalidades de la app web

La aplicación web ofrece un conjunto de funcionalidades orientadas tanto al procesamiento como a la visualización y gestión de resultados:

8.1. Compresión de vídeos

La aplicación soporta la compresión de archivos de vídeo además de datos y texto. Los vídeos se procesan en bloques para permitir compresión incremental y manejo de archivos de gran tamaño. El flujo de trabajo incluye:

- Preprocesado y tokenización específica para flujos multimedia.
- Codificación por bloques mediante el encoder adaptativo (Vitter).
- Exportación de resultados en formatos comprimidos aptos para descarga.

8.2. Sección de codificación y decodificación (encode / decode) y *de co out*

La interfaz dispone de un área dedicada a las operaciones de codificación y decodificación:

- Panel `Encode` para iniciar la compresión de la entrada seleccionada.
- Panel `Decode` para restaurar datos a partir del flujo de bits comprimido.

- La llamada sección de *de-co-out* corresponde al espacio de entrada/salida donde se muestran los resultados (output) de las operaciones: longitud media del código, entropía de la fuente, y el flujo de bits generado. Aquí el usuario puede inspeccionar códigos por símbolo y verificar desigualdades teóricas (por ejemplo, la desigualdad de Shannon).

8.3. Manejo de archivos

La aplicación maneja la carga, almacenamiento temporal y descarga de archivos con las siguientes capacidades:

- Subida de archivos mediante formularios o arrastrar y soltar, con validación de tipos y tamaños.
- Procesamiento en servidor (endpoints `/api/encode`, `/api/decode`) y streaming de resultados.
- Posibilidad de descargar el archivo comprimido o el resultado de la decodificación.
- Gestión de errores y notificaciones para operaciones sobre archivos (pérdida de conexión, límites de tamaño, formatos no soportados).

Esta sección debe insertarse en ‘implementacion.tex‘ donde se describen las características de la interfaz y la interacción con el backend.

9. Ventajas y Limitaciones

9.1. Ventajas del Algoritmo de Vitter

El algoritmo de Vitter presenta diversas ventajas frente al método de Huffman estático, especialmente en entornos donde la información se procesa de manera continua o en tiempo real. Entre sus principales fortalezas se encuentran:

- **Compresión adaptativa:** permite ajustar dinámicamente las probabilidades de los símbolos a medida que se leen, sin necesidad de conocer previamente la distribución de frecuencias.
- **Una sola pasada:** comprime los datos en un único recorrido, lo que lo hace ideal para flujos de datos o transmisión en directo.
- **Eficiencia temporal:** presenta un coste amortizado de $O(1)$ por símbolo, alcanzando una complejidad total $O(n)$.
- **Reducción de latencia:** el proceso de compresión comienza desde el primer símbolo, evitando el retardo inicial de los métodos estáticos.

9.2. Limitaciones del Algoritmo de Vitter

A pesar de su eficiencia, el algoritmo de Vitter no está exento de desventajas, especialmente cuando se busca la máxima compresión posible sobre un conjunto de datos completo:

- **Complejidad de implementación:** el mantenimiento del árbol adaptativo y la gestión del número de bloques requieren estructuras de datos y operaciones más complejas.
- **Ligera pérdida de compresión:** al actualizar el árbol de manera incremental, puede no alcanzar la misma eficiencia en la longitud media del código que el árbol óptimo generado por Huffman estático.
- **Mayor sobrecarga en memoria:** el mantenimiento de información dinámica sobre los pesos y las posiciones de los nodos implica un pequeño coste adicional.

Referencias

1. Vitter, J. S. (1987). “Design and Analysis of Dynamic Huffman Codes”. Journal of the ACM, 34(4), 825-845.

2. Shannon, C. E. (1948). “A Mathematical Theory of Communication”. Bell System Technical Journal, 27(3), 379-423.

3. Huffman, D. A. (1952). “A Method for the Construction of Minimum-Redundancy Codes”. Proceedings of the IRE, 40(9), 1098-1101.

4. Sayood, K. (2017). “Introduction to Data Compression” (5th ed.). Morgan Kaufmann.

5. Cover, T. M., & Thomas, J. A. (2006). “Elements of Information Theory” (2nd ed.). Wiley.

10. Glosario de Términos

Adaptativo Que se modifica dinámicamente según las características de los datos.

Cabecera Información transmitida al inicio (tabla de códigos, diccionario).

Código Prefijo Propiedad donde ningún código es prefijo de otro.

Compresión sin Pérdida Reconstrucción exacta de datos originales.

Entropía Medida de incertidumbre o información promedio (bits/símbolo).

Invariante Propiedad que se mantiene cierta en todo momento.

Nodo NYT “Not Yet Transmitted”, representa símbolos nunca vistos.

Numeración Implícita Sistema de asignación de números a nodos del árbol.

Streaming Procesamiento de datos en línea a medida que llegan.

Swap/Slide Intercambio de posiciones de nodos en el árbol.