Universidad Politécnica
de Madrid

Escuela Técnica Superior de
Ingenieros Informáticos

Grado en Ingeniería Informática

Trabajo Fin de Grado

Compresión de Archivos CSS con Diccionario Compartido

Autor: Javier González González

Tutor: Jesús Martínez Mateo

Madrid, enero 2026

Este Trabajo Fin de Grado se ha depositado en la ETSI Informáticos de la Universidad Politécnica de Madrid para su defensa.

Trabajo Fin de Grado
Grado en Ingeniería Informática
Título: Compresión de Archivos CSS con Diccionario Compartido
Enero 2026

Autor: Javier González González

Tutor: Jesús Martínez Mateo
Matemática Aplicada a las Tecnologías de la Información y las Comunicaciones.
ETSI Informáticos
Universidad Politécnica de Madrid

Resumen

El presente Trabajo de Fin de Grado describe el diseño, la implementación y la evaluación de un sistema de compresión y descompresión de datos sin pérdida, enfocado a archivos de hojas de estilo en cascada (CSS) Mediante el uso de un diccionario compartido.

La propuesta nace de lograr una mejora en la experiencia del usuario optimizando los recursos web. Aprovechando la naturaleza repetitiva del lenguaje CSS consiguiendo asi una reducción de los tiempos de carga de las páginas.

El algoritmo principal del sistema es la Codificación Huffman. A diferencia de los métodos tradicionales a nivel de carácter, este trabajo propone una tokenización semántica que identifica palabras, propiedades, valores y signos de puntuación como unidades básicas de información.

El sistema se ha desarrollado bajo una arquitectura híbrida combinando la versatilidad de Python para el análisis estadístico y la robustez de Java para el procesamiento de archivos.

Dividiendo en fases el proceso, se comenzó con la creación de un corpus representativo de archivos CSS de diversas fuentes. Mediante los scripts de Python se realizó un análisis de frecuencias para identificar los tokens más comunes, generando un árbol binario de Huffman que sirve de base para el diccionario compartido.

La implementación del software de compresión y descompresión se realizó en Java, gestionando el flujo de datos a nivel de bit. Y logrando una descompresión de archivo sin pérdida de datos, igual al original. Logrando hasta un 80% de compresión en ciertos escenarios.

El proyecto busca contribuir al ámbito de la tecnología proporcionando una base para crear un método de paso de información sin pérdida. Su diseño está orientado a optimizar el paso de información a través de la web y a facilitar la transferencia eficiente de datos estructurados, reduciendo costes de ancho de banda y latencia de red.

Abstract

This Final Degree Project describes the design, implementation, and evaluation of a lossless data compression and decompression system, specifically focused on Cascading Style Sheets (CSS) files through the use of a shared dictionary.

The proposal aims to improve user experience by optimizing web resources, leveraging the repetitive nature of the CSS language to achieve a significant reduction in page loading times.

The system's core algorithm is Huffman Coding. Unlike traditional character-level methods, this work proposes a semantic tokenization approach that identifies words, properties, values, and punctuation marks as basic units of information.

The system was developed under a hybrid architecture, combining the versatility of Python for statistical analysis with the sturdiness of Java for file processing.

The process was divided into several phases, beginning with the creation of a representative corpus of CSS files from various sources. Using Python scripts, a frequency analysis was conducted to identify the most common tokens, generating a Huffman tree that serves as the foundation for the shared dictionary.

The implementation of the compression and decompression software was carried out in Java, managing the data flow at the bit level and achieving lossless decompression that restores the file to its exact original state. Experimental results show compression rates of up to 80% in specific scenarios.

This project seeks to contribute to the field of technology by providing a foundation for creating lossless information transfer methods. Its design is oriented toward optimizing data flow across the web and facilitating the efficient transfer of structured data, thereby reducing bandwidth costs and network latency.

Índice General

Tabla de Ilustraciones

1 Introducción

1.1 Contexto y Motivación

El continuo volumen de datos, transferidos a través de la red, ha convertido a los recursos web en un área critica para mejorar la experiencia de los usuarios al reducir el tiempo de carga de las páginas. [1]

Dentro de la tríada de los componentes web HTML, JavaScript y CSS, los CSS representan una porción significativa y altamente repetitiva del tráfico de datos. Gracias a esta estructura que tienen, este tipo de archivos puede llegar a ser comprimidos mediante algoritmos especializados.

Normalmente la compresión de este tipo de recursos se aborda mediante algoritmos genéricos de compresión sin pérdida como Gzip. Sin quitar merito a su gran efectividad, no aprovechan del todo la repetitividad del lenguaje CSS. Operan a nivel de flujo de caracteres o bloques de datos sin una comprensión profunda de la estructura semántica del lenguaje que están comprimiendo.

En este trabajo de fin de grado intentaremos lograr una solución que supere el rendimiento de los compresores genéricos.

1.2 Objetivo General

El objetivo principal de este proyecto es el diseño, implementación y evaluación de un sistema de compresión y descompresión de datos sin pérdida, aplicado específicamente para archivos hojas de estilo en cascada (CSS).

La solución se basa en la creación de un diccionario compartido de tokens especializado y el uso del algoritmo de Huffman para generar códigos de prefijo óptimos. Buscando demostrar que es posible alcanzar una eficiencia de transmisión superior a los métodos de compresión genéricos en escenarios de alta redundancia, proporcionando una herramienta que reduzca la latencia de red y el consumo de ancho de banda en entornos web y corporativos.

1.3 Objetivos Específicos

Para alcanzar el objetivo propuesto, se han definido los siguientes hitos y metas para el desarrollo del proyecto:

- **Fase de investigación y análisis**
    - Realizar un estudio sobre las técnicas de compresión actuales, haciendo énfasis en la codificación de Huffman.
    - Recopilar y analizar corpus representativos de archivos CSS de diversas fuentes, identificando patrones en ellas.

- Desarrollar herramientas en lenguaje Python que nos permitan analizar de forma automática los tokens más frecuentes y generen el árbol de Huffman.

- **Fase de Diseño del Sistema**

  - Definir el formato binario del archivo comprimido para asegurar una estructura que permita la reconstrucción exacta del código original.

  - Diseño de la estructura de datos para el diccionario compartido

  - Establecer una estrategia para el código de escape (CE) para la gestión de los tokens que no se encuentren en el diccionario.

- **Fase de Implementación Técnica**

  - Evaluar el rendimiento del sistema utilizando diferentes tipos de corpus, para medir el factor de compresión promedio usado.

  - Identificar el tamaño ideal del árbol binario de Huffman para maximizar el tamaño árbol/ancho de banda necesario para enviarlo.

  - Contrastar la eficacia de la solución frente a estándares de compresión.

1.4 Justificación de la elección de CSS

La elección de las hojas de estilo en cascada (CSS) como objeto de estudio para este sistema de compresión no es casual, sino que responde a tres factores extraídos del análisis previo de la tecnología web actual:[10]

- **Naturaleza Altamente Repetitiva:** CSS es un lenguaje declarativo donde un conjunto de propiedades y valores se reutilizan de forma masiva a lo largo de cualquier hoja de estilos. Esto lo convierte en el candidato ideal para algoritmos basados en frecuencia como Huffman.

- **Impacto en el Rendimiento:** Los Navegadores Web no pueden mostrar el contenido hasta que se ha descargado y procesado los estilos. Por tanto, reducir su tamaño se traduce en una mejora en la fluidez y en los tiempos de carga.

- **Crecimiento del Ecosistema Web:** Con el auge de los frameworks y el código generado automáticamente los archivos CSS han crecido de tamaño y redundancia,[10]el desarrollo de un método especializado ofrece una solución para manejar grandes cantidades de datos estructurados

2 Fundamentos teóricos

En este capítulo se exponen los pilares teóricos sobre los que se sustentan el desarrollo del sistema de compresión de archivos CSS.

2.1 Problema de la Compresión en la Web

La fluidez que genera un corto tiempo de carga en las páginas web es un pilar para mejorar la experiencia de los usuarios. Una latencia elevada no solo genera frustración, sino que incrementa la tasa de abandono, provocando que el usuario pierda el interés [2].

Para mitigar este problema, tradicionalmente se han empleado técnicas como la minificación, sin embargo, este enfoque resulta insuficiente, ya que se limita a una limpieza superficial del archivo y no trata la redundancia del lenguaje.[3]

Es precisamente en esta redundancia donde la codificación estadística y el uso de diccionarios especializados ofrecen un potencial de optimización significativamente mayor al de las técnicas genéricas.

2.2 Fundamentos de la Compresión Sin Pérdida

La necesidad de reducir el volumen de los datos conservando la integridad total de la información es un requisito esencial en la computación moderna. A diferencia de los algoritmos de compresión con pérdida, típicamente usados en formatos de audio o imagen, los algoritmos de compresión sin pérdida garantizan que el archivo descompresor sea una réplica exacta del original.[4]

Esta característica es imprescindible para el envío de código, como pueden ser los archivos CSS, donde la alteración de un solo carácter corrompe la sintaxis y el funcionamiento de la hoja de estilos.

Estos algoritmos operan principalmente mediante la identificación y eliminación de la redundancia de los datos. Para lograrlo, se dividen fundamentalmente en dos grandes estrategias:

- **Codificación estadística:** Estos métodos se basan en la probabilidad de aparición de los símbolos. El más relevante es el Algoritmo de Huffman donde se asignan códigos binarios de longitud variable. Siendo los códigos más cortos para los caracteres (en este proyecto, tokens) con mayor frecuencia de aparición, mientras que los menos frecuentes reciben códigos de mayor longitud.

- **Compresión basada en diccionarios:** Esta categoría, donde destacan los algoritmos de la familia LZ (Lempel-Ziv), no se basa tanto en la frecuencia individual, sino en la repetición de secuencias o patrones de texto. Estos patrones se sustituyen por referencias o entradas a una tabla/diccionario.

2.3 Algoritmo de Codificación Huffman

Desarrollado por David A. Huffman en 1952, este algoritmo es un método de codificación estadística que produce lo que se conoce como una codificación de prefijo óptima. Esto significa que cada código puede identificarse sin tener que mirar los siguientes bits, lo que permite que el flujo de datos sea decodificado bit a bit sin ninguna ambigüedad y sin necesidad de marcadores de separación entre símbolos.

El proceso se basa en la construcción de un árbol binario, donde la frecuencia de aparición de cada símbolo determina su posición jerárquica en el árbol y, por consiguiente, la longitud de su palabra de código.

A símbolos con mayor probabilidad de aparición se le asignan ramas más cercanas a la raíz obteniendo estos códigos binarios muy cortos.

A símbolos menos frecuentes se les coloca en las hojas más profundas del árbol donde reciben códigos más extensos.[5]

Símbolo	Frecuencia
A	0,15
B	0,30
C	0,20
D	0,05
E	0,15
F	0,05
G	0,10

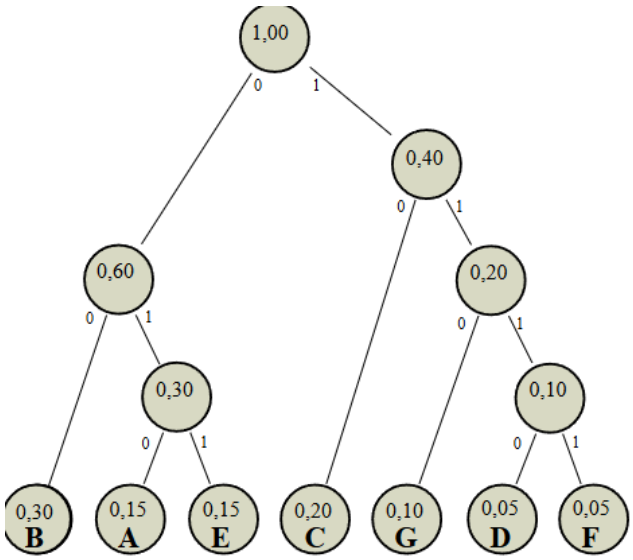

Ilustración 1 - ejemplo de Árbol de Huffman

En el contexto de este Trabajo de Fin de Grado introducimos una innovación clave, los “símbolos” no son caracteres individuales (bytes), sino Tokens CSS completos, como valores, propiedades, o palabras.

Al operar a ese nivel semántico, el algoritmo puede asignar, por ejemplo, un código de apenas 8 bits a una cadena larga como background-color, logrando una ganancia de compresión significativamente superior a la que obtendría un compresor Huffman tradicional que procesara cada letra por separado.

2.4 Compresión Basada en Diccionarios

Los algoritmos de la familia LZ funcionan identificando patrones repetitivos y sustituyéndolos por referencias de posición y longitud a cadenas previas.

Aunque este proyecto se centra en la codificación Huffman, el proceso de tokenización usado para CSS guarda una estrecha relación con el concepto de diccionario. Las palabras o tokens más frecuentes se extraen y codifican mediante Huffman para asignarles un código binario optimo según su probabilidad de aparición, y posteriormente se almacenan en un diccionario el cual ambas partes, tanto emisor como receptor deben poseer.

2.5 Herramientas de compresión de CSS Existentes

En la actualidad, la optimización de archivos CSS se maneja principalmente mediante dos metodologías complementarias: la minificación y la compresión genérica. Como se ha mencionado, la minificación se limita a una limpieza de caracteres irrelevantes para el funcionamiento. Por otro lado, la compresión genérica, se implementa directamente en los servidores web para reducir el volumen de datos enviados.[6][7]

A pesar de su alta eficacia, estos algoritmos genéricos procesan la información como un simple flujo de bytes, sin poseer un conocimiento sintáctico del lenguaje CSS. La propuesta de este Trabajo Fin de Grado se posiciona en un punto intermedio, ofrece una compresión semánticamente de un corpus de archivos, y al emplear un diccionario específico y optimizado para los tokens más comunes de los CSS, el sistema busca superar el rendimiento de las herramientas genéricas.

2.6 Minificación

La minificación es una técnica de optimización de recursos web que consiste en la eliminación de todos los caracteres que son necesarios para la legibilidad humana, pero totalmente prescindibles para la ejecución por parte del navegador.[8] Por ejemplo los comentarios, saltos de línea, espacios en blanco redundantes. Llegando a ahorrar un 60% de tamaño [9]

A lo largo de la fase de pruebas se ha observado que el rendimiento del compresor desarrollado en este trabajo mejora drásticamente cuando se procesa código minificado, alcanzando tasas de reducción de tamaño de hasta 81%. Ya que se elimina la varianza del formato visual, permitiendo que la codificación se centre únicamente en tokens lógicos totalmente.

2.7 Análisis de la Naturaleza Repetitiva del Lenguaje CSS

El lenguaje CSS es un lenguaje declarativo con un grado de redundancia muy elevado. Esta repetitividad se manifiesta en patrones predecibles. Tras el análisis del corpus inicial se ha confirmado que estas repeticiones son fijas y es posible aprovecharlas.

Esta predictibilidad constituye el pilar de la ganancia de compresión del sistema, el algoritmo de Huffman permite que tokens extensos y frecuentes sean sustituidos por códigos binarios mínimos. Un ejemplo representativo es la propiedad background-color. Mientras que en texto plano este token ocupa 16 bytes (128 bits), mediante la codificación de Huffman puede reducirse a

una cadena de tan solo 8 bits, logrando una reducción de peso superior al 90% para ese token específico.

Esta capacidad de compactar tokens completos es la que permite que el sistema alcance un gran rendimiento, especialmente en entornos donde la estructura del código se repite de forma masiva.

3 Herramientas y Entorno de Desarrollo

Para este proyecto se ha elegido una arquitectura híbrida, que aprovecha la fortaleza de diferentes lenguajes y herramientas de automatización.

3.1 Lenguajes de Programación

3.1.1 Python

Se ha utilizado Python para la fase de análisis estadístico y generación del modelo de Huffman. Su flexibilidad para el procesamiento de texto junto con su gran abanico de bibliotecas hace que resulte ideal para manejar el corpus y calcular frecuencias

En el desarrollo de los scripts analizadorCSS.py y arbolHuffman.py, se han empleado las siguientes librerías estándar:

- **re (Regular Expressions):** Utilizada para tokenizar la semántica del corpus, ya que permite definir patrones complejos para extraer las propiedades, valores, símbolos y palabras del CSS[11]

- **collections.Counter:** utilizada para contabilizar de forma eficiente la frecuencia de aparición de cada token del corpus.[12]

- **heapq:** Biblioteca que implementa el algoritmo de cola de prioridad. Es la pieza clave para construir el árbol binario de Huffman. Permite extraer los nodos con menor frecuencia.[13]

- **os:** Utilizada para la gestión del sistema de archivos, permitiendo recorrer directorios y cargar los archivos .css del corpus.[14]

3.1.2 Java

Hemos elegido el lenguaje Java debido a la robustez en el manejo de flujos de datos y porque es el lenguaje que más dominamos. Mientras que Python gestiona el apartado de análisis del modelo, Java se encarga del procesamiento de archivos.

Se han empleado las siguientes librerías y componentes de la API estándar de Java para la manipulación de datos:

- **Java.io:** Se han utilizado sus clases permitirnos leer el archivo .css original y escribir el archivo comprimido .cssc .[15]

- **java.util.regex:** Similar a la biblioteca re de Python la utilizamos para identificar expresiones regulares, palabras, valores, símbolos dentro del código css.[16]

- **java.util.HashMap:** Utilizada para la estructura de datos especifica, donde se almacenan los tokens y códigos binarios. Permite la búsqueda

de códigos con una complejidad mínima, maximizando la velocidad del proceso. [16]

- **java.nio.file:** Empleada para la lectura de bytes del archivo de entrada, garantizando compatibilidad entre diferentes sistemas de archivos.[17]

Además de las librerías destacadas, mencionar la implementación de clases propias para la gestión de bajo nivel:

- **bitWriter y bitReader:** Estas clases permiten la escritura y lectura de bits individuales, algo que Java no permite de forma nativa.
- **NodoHuffman:** Clase diseñada para reconstruir el árbol binario en memoria durante la descompresión, permitiendo recorrer las ramas.

3.2 Inteligencia Artificial

En el desarrollo de este proyecto se han utilizado herramientas de Inteligencia Artificial Generativa, en concreto el modelo de Google, Géminis, como apoyo técnico y metodológico.

Su uso se ha reducido a fases específicas, por ejemplo, en la agilización de las pruebas del sistema, minificación de archivos o creación de código para el corpus de archivos CSS.

Remarcar que en ningún caso se ha usado tal herramienta para el proceso de diseño, análisis, toma de decisiones técnicas o conclusiones.

3.3 Herramientas Automatización

Para mejorar la experiencia del usuario y coordinar de manera eficiente el uso del sistema, se ha utilizado GNU Make para crear un archivo Makefile.

Esta infraestructura no solo reduce la posibilidad de errores en la secuencia de ejecución, sino que también proporciona una interfaz sencilla y estandarizada para la utilización del sistema.

3.4 Entorno de Ejecución

El desarrollo, las pruebas de rendimiento y la validación final del sistema se han llevado a cabo en un entorno basado en Linux.

Los aspectos clave de este entorno son los siguientes:

- **Sistema Operativo:** Al utilizarse una distribución Linux, facilita el manejo de rutas de archivos y la ejecución de scripts mediante la terminal

- **Gestión de Archivos:** La terminal de Linux permite una interacción fluida con el Makefile, coordinando los módulos de Python y Java.

- **Consistencia del Sistema:** El entorno proporciona una base estable para realizar las mediciones de tiempo y tasa de compresión, pudiendo ser precisos en las pruebas de rendimiento.

- **Conocimiento Previo:** La experiencia como estudiante de ingeniería informática, del manejo de dicho sistema operativo, hace más sencillo la creación del proyecto en un entorno Linux.

3.5 Gzip

Gzip es una herramienta de compresión de datos de código abierto ampliamente utilizada en la transferencia de recursos a través de la web.[21][22] su funcionamiento se basa en el algoritmo Deflate, el cual combina dos técnicas de compresión sin pérdida. El algoritmo LZ77, encargado de sustituir secuencias repetidas de datos con punteros a apariciones previas y la codificación Huffman dinámica, que asigna códigos de longitud variable a los símbolos resultantes.

En el contexto de este trabajo, Gzip se utiliza como el principal punto de referencia para evaluar el rendimiento de la solución propuesta. Destacar que, aunque Gzip es altamente eficiente en archivos de gran volumen, requiere la inclusión de metadatos, cabeceras y tablas de frecuencias, en cada archivo individual. Esta característica es la que permite que nuestro sistema basado en un diccionario compartido presente una ventaja en archivos CSS de pequeño tamaño.

4 Diseño y Desarrollo del Sistema

Una vez establecidos los fundamentos teóricos y las herramientas usadas, en este capítulo detallaremos la arquitectura y el proceso de construcción del sistema de compresión y descompresión. El desarrollo se ha estructurado siguiendo un camino lógico, comenzando con análisis estadísticos y culminando en la ejecución del sistema.

El sistema se ha diseñado para que cada módulo cumpla una función específica, garantizando la eficiencia y la simplicidad. Este proceso de desarrollo se divide en cuatro fases fundamentales:

- **Fase de Análisis y Tokenización:** Donde se procesa el corpus de archivos CSS para extraer los tokens y generar un diccionario de frecuencias.
- **Fase de Codificación**: Donde se transforma dicho análisis en una estructura de árbol binario, asignando los códigos de prefijo óptimos.
- **Fase de Ejecución**: Donde se implementa la lógica de manipulación de bits en Java para transformar los archivos originales en archivos comprimidos y viceversa.

4.1 Análisis del Corpus y Estrategia de Tokenización

En esta primera fase del desarrollo, el objetivo es transformar un conjunto de datos no estructurados (archivos CSS de texto plano) en un modelo matemático de frecuencias.

Para lograrlo, el sistema no trata el código como una simple cadena de caracteres, sino que, mediante el uso de herramientas de análisis desarrolladas en Python, se descompone el lenguaje CSS en sus unidades lógicas básicas.

Este proceso de preparación usa la clase analizadorCSS.py y es que permite que el árbol binario de Huffman posterior sea optimo, priorizando aquellos elementos que más peso tienen en los archivos CSS a comprimir.

4.1.1 Selección del Corpus

Para garantizar que el compresor sea versátil y eficiente en escenarios del mundo real, se ha recopilado un corpus diverso compuesto por más de 60 archivos CSS. La selección se puede diferenciar en tres categorías principales:

- **Frameworks y Esqueletos Base:** Archivos provenientes de bibliotecas como Bootstrap o archivos de uso libre publicados en GitHub, que contienen las definiciones estándar y más universales de la web.

- **Sitios Web Reales:** Código extraído de plataformas en producción y personas cercanas, para capturar la variabilidad de estilos, selectores complejos y malas prácticas comunes que el compresor debe ser capaz de gestionar.

- **Código Generado por IA y Herramientas Automáticas:** Con el auge de estas herramientas, se incluyó código generado automáticamente para observar patrones de redundancia distintos a los de la escritura manual.

Al momento de la realización de pruebas se especificará el contenido del corpus para cada resultado.

4.1.2 Proceso de Tokenización

La tokenización es el proceso de descomponer una cadena de texto, en unidades mínimas con sentido [18]. En este proyecto, en vez de los compresores genéricos que trabajan a nivel de carácter o byte, se ha implementado una tokenizacion que divide las estructuras del lenguaje CSS.

Gracias al uso de la librería re de Python podemos identificar y seleccionar los diferentes componentes del código, llegando a identificar cientos de miles de tokens en los corpus seleccionados. El proceso se divide en diferentes etapas:

- **Identificación de estructuras fijas:** se definen patrones para reconocer símbolos de control como llaves ({, }), puntos y comas(;), saltos de línea, tratándolos como tokens de alta frecuencia.

- **Extracción de Propiedades y Valores:** mediante expresiones regulares se capturan nombres de propiedades (display, margin-top) y sus valores asociados.

- **Gestión de Unidades y Medidas:** El analizador identifica unidades comunes como px, o %, permitiendo que el algoritmo de Huffman asigne códigos cortos a estas terminaciones si se repiten frecuentemente.

- **Tratamiento de Espacios y Formato:** En el caso de código no minificado, se identifican los saltos de línea y espacios en blanco como tokens. Sin embargo, como se detalla en las pruebas, la eliminación de estos elementos mediante una minificación aumenta la densidad de los símbolos con importancia lógica, mejorando la tasa de compresión final.

- **Filtrado de Comentarios:** El script identifica y descarta los comentarios, ya que contienen palabras específicas que no forman parte del diccionario afectando la eficiencia del árbol binario.

```
# patron para capturar tokens
 regex_pattern = r"#[0-9a-fA-F]{3,8}|-?(?:\d*\.)?\d+(?:[a-zA-Z%]+)?|[a-zA-Z0-9_\-@]+|\"[^\"]*\"|'[^']*'|[^\s\w]|\s+"
```

Ilustración 2 - patrón utilizado

Al final del proceso, el corpus de archivos CSS se convierte en una lista de tokens lista para ser procesada por el algoritmo de generación del árbol.

4.1.3 Identificación de Frecuencias

Una vez que el proceso de tokenización ha partido el corpus en unidades lógicas, se inicia el sistema de análisis estadístico. El objetivo de esta parte es determinar la probabilidad de aparición de cada token, ya que según dicha probabilidad Huffman asignara un código binario de longitud especifica.

Se sigue utilizando la clase analizadorCSS.py pero en este caso se hace uso de la librería collections.Counter de Python, para recorrer la lista de tokens extraídos y generar un diccionario de frecuencias.

La cantidad de tokens guardados en el diccionario puede variar según la necesidad del usuario. Tras realizar un estudio del número optimo de tokens, concluimos que 1500 consigue una gran compresión y no sobredimensiona el árbol.

Este mapa de frecuencias es el producto final de la fase de análisis generando un archivo .txt. Este archivo sirve como entrada para la construcción del árbol binario de Huffman, garantizando asi que el modelo este adaptado a la realidad del corpus seleccionado.

4.1.4 Implementación del Módulo de Análisis

El código de analizadorCSS.py es el primer escalón de la cadena de compresión. Su función principal es actuar como analizador léxico de las hojas de estilo. En

este punto explicaremos las responsabilidades y funcionamiento de la clase en los siguientes puntos:

- **Procesamiento del Corpus:** La clase utiliza la librería os para realizar un barrido completo de los directorios donde se guardan los archivos CSS.

- **Limpieza**: Antes de la tokenización, el código se encarga de eliminar elementos que introducen ruido, principalmente los comentarios. Como se explicó anteriormente eliminar esta variable hace que no se creen tokens innecesarios, centrándose el modelo únicamente en patrones del lenguaje.

- **Ejecución de la Tokenización Semántica:** Es aquí donde se aplica el patrón para descomponer el archivo en una lista de tokens, identificando propiedades, valores y símbolos estructurales.

- **Generación de la Tabla de Frecuencias:** Una vez obtenida la lista de tokens de todo el corpus, la clase utiliza collections.Counter para generar un diccionario de, en este caso, 1500 frecuencias. Asi se mapea cada token con su número total de apariciones.

- **Salida de Datos:** Por último, la clase guarda estas estadísticas en un archivo de texto con un formato especifico. Estos datos son los que permitirán al siguiente módulo determinar el código binario asignado a cada uno.

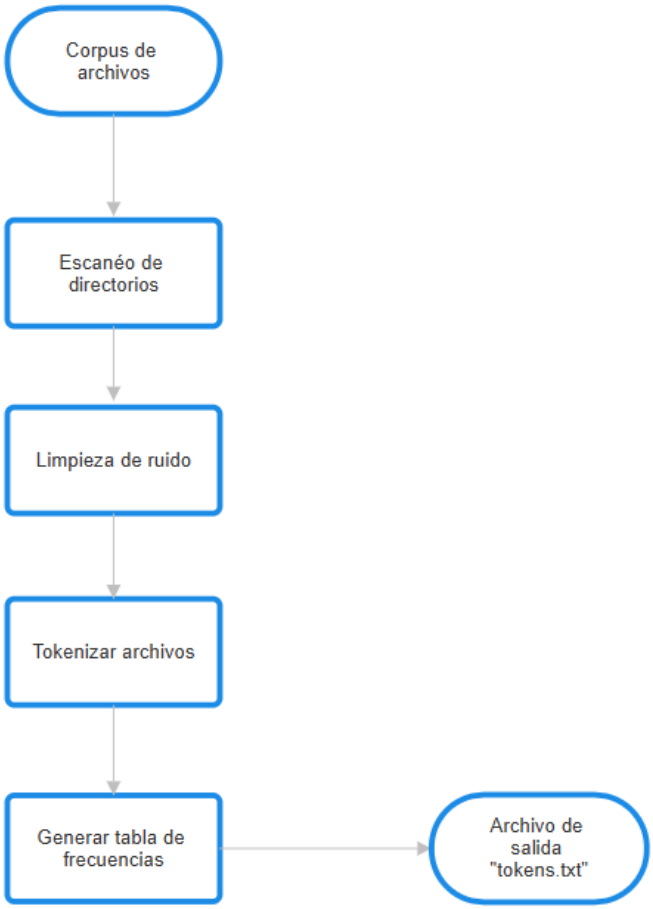

Ilustración 3 - diagrama de flujo módulo de análisis

4.2 Diseño del Diccionario Compartido

El núcleo de este proyecto se basa en el uso de un diccionario compartido entre el compresor del emisor y el descompresor del receptor. En vez de enviar un árbol binario de Huffman único para cada archivo, este sistema envía un diccionario específico para ese primer paso de información. Esta decisión de diseño busca optimización del sistema, priorizando la velocidad y la tasa de compresión.

La clase dedicada al procesamiento de dicho archivo se llama arbolHufmman.py

4.2.1 El Diccionario Estático

La elección de un diccionario estático frente a uno modelo dinámico (que se genera en cada compresión) pretende responder a un problema clásico, el dilema entre "overhead" (coste en enviar el árbol) o sobrecoste de metadatos en los archivos.

Las razones técnicas que justifican dicha elección son las siguientes:

- **Eliminar el sobre coste de la cabecera:** Actualmente es necesario añadir el árbol binario de Huffman o la tabla de frecuencias al archivo comprimido para que el descompresor pueda entenderlo. En archivos pequeños este diccionario adjunto puede ocupar más espacio que la mejora de compresión conseguida. Al usar un diccionario estático este coste desaparece, solo debes enviar el árbol binario una única vez.

- **Optimización para la Red:** Siguiendo la lógica de la justificación anterior**,** en entornos web, donde se envían múltiples archivos, tener un diccionario estático en el cliente y el servidor permite una descompresión sin transferir metadatos repetitivos.

- **Velocidad de Ejecución:** El compresor no necesita realizar un cómputo previo por cada archivo para contar frecuencias ni construir el árbol desde cero. Simplemente mapea los tokens con el diccionario preexistente, reduciendo asi el tiempo de procesamiento.

4.2.2 Determinación del Tamaño Optimo

Una de las decisiones más críticas del diseño del sistema fue la selección del número de entradas o tokens que debía de tener el diccionario estático. Tras realizar varias pruebas de compresión con diferentes tamaños de diccionario, se fijó en 1500 tokens.

Esta cifra es el resultado de buscar el equilibrio entre diversos criterios.

Primero no extender demasiado la longitud del código Huffman. Cuantas más hojas existen en el árbol, más profundo es de media. Esto significa que los códigos son de media más largos, incluso los de los tokens más frecuentes, reduciendo la tasa de compresión.

El consumo de memoria y eficiencia está relacionado con el tamaño del diccionario. Un diccionario de 1500 tokens permite una carga rápida en memoria y reconstruir rápidamente el árbol.

Sin embargo, un numero de tokens reducido (por ejemplo 500) no cubre de manera eficiente todas las propiedades del estándar CSS. Esto provoca que el CE tenga un gran peso en el árbol perjudicando gravemente la compresión del archivo.

Y por último al contrario que el párrafo anterior, ampliar la cobertura más allá del rango de 1500 tokens hace que la ganancia de compresión para cada nuevo token sea marginal ya que se empiezan a incluir términos muy específicos o poco frecuentes.

4.2.3 Estructura del Árbol binario de Huffman

Una vez definido el diccionario de, en mi caso, 1500 tokens, el sistema utiliza el algoritmo de Huffman para construir un árbol binario optimizado. Siendo esta estructura la que orquesta la jerarquía de los códigos que usara el compresor.

Con la propiedad de prefijos garantizamos que mientras se lea el flujo de bits ningún código se confunda con el siguiente, por lo que no necesitamos separadores.

Como mencionamos antes en la parte de introducción, el árbol se genera de forma que los tokens con mayor frecuencia de aparición se sitúan más cerca de la raíz, obteniendo códigos binarios más cortos. Y, el código de escape (CE) se sitúa de manera dinámica.

Normalmente no todos los tokens observados se les asigna un código binario, ya que normalmente hay más de 1500. En estos casos al momento de comprimir el archivo el sistema, al no encontrar el código de dicho token escribe el CE, un código especifico que varía en cada árbol según la cantidad de tokens que se quedan fuera. Este token hace saber al sistema que se viene un token literal sin comprimir. Después se añade la longitud de bits que se deben leer y por último el token literal. De esta manera conseguimos que no haya pérdidas de datos y podamos reconstruir el archivo original.

El código CE se asigna según la cantidad de tokens que se quedan fuera, por ejemplo, si de 100 tokens observados en el corpus, nuestro árbol ha podido guardar 98, ese 2% se asigna como peso al token de CE. De esta manera conseguimos que el código binario asignado no sea ni demasiado corto (y robe posiciones de tokens con mucho peso) ni muy largo, haciendo que empeore la compresión al introducir más bits de los necesarios.

4.2.4 Implementación del Módulo de Codificación

Si el módulo anterior se encargaba de las estadísticas, la clase arbolHuffman.py representa la fase de modelado. Su función es construir el árbol binario que define los códigos binarios basándose en la tabla de frecuencias generada por el analizador.

El funcionamiento se basa en los siguientes pasos:

- **Introducción de Tokens en Cola:** Para construir el árbol de forma eficiente, se utiliza un *min-heap*. Cada token se introduce en la cola como un nodo hoja con su frecuencia asociada. La clase extrae los dos nodos

con menor frecuencia, crea un nodo padre cuya frecuencia es la suma de ambos, y lo reinserta en la cola. Asi aseguramos que los elementos menos frecuentes quedaran más alejados de la raíz donde se quiere encontrar a los más frecuentes.

- **Asignación de Códigos:** Una vez colapsada la cola en un único nodo raíz, la clase recorre el árbol. A cada ramificación a la izquierda se le asigna el bit 0 y a la derecha el bit 1. Los códigos resultantes son almacenados en una estructura de diccionario de Python.

- **Insertar el Nodo de Escape:** En esta clase se fuerza la inserción del token especial CE, con una frecuencia calculada para situarlo en una posición óptima del árbol.

- **Salida de datos:** El resultado final se guarda en un archivo de texto con un formato especifico. Contiene la traducción de cada token a su secuencia de bits. Este es el archivo que se deberá pasar junto con el primer envío de código CSS, el diccionario compartido.

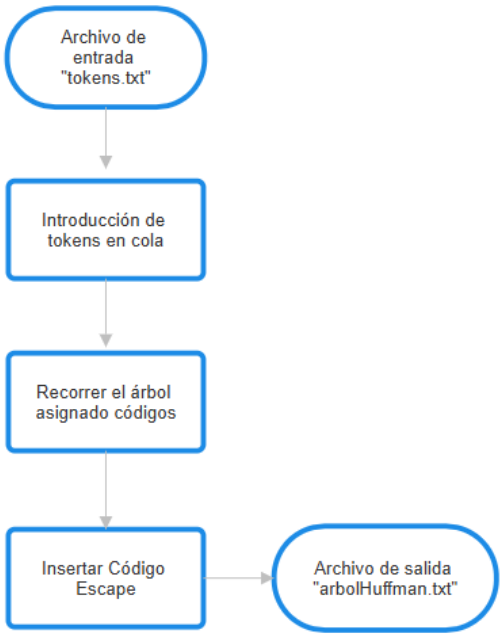

Ilustración 4 - diagrama de flujo módulo de codificación

4.3 Implementación del Compresor

Tras la fase de análisis estadístico y generación del diccionario, se procede con la implementación del motor de compresión. Esta etapa contribuye el núcleo operativo del sistema, donde la teoría de la codificación Huffman se traduce en un software de compresión de archivos reales.

Esta implementación se ha centrado en la precisión en el flujo de bits. Desarrollándose una lógica de escritura bit a bit, ya que Java nativo opera con bloques de 8 bits.

4.3.1 Arquitectura del Compresor y Flujo de Datos

La implementación del sistema de compresión en Java se ha basado en un diseño de gestión de flujos de datos binarios de bajo nivel. Lo que permite una aplicación directa entre la lógica de bits de Huffman y el sistema de archivos que se basa en bytes.

Los componentes centrales de esta parte son:

- **Clase principal, Compresor.java:** Es la clase encargada del control del flujo de la ejecución. Su responsabilidad principal es el ciclo de vida de los streams de E/S, el FileInputStream y FileOutputStream. Actúa como unión entre el analizador sintáctico y el motor de escritura binaria.

- **Clase auxiliar, DiccionarioHuffman.java:** El sistema carga el archivo arbolHuffman.txt en una estructura de tipo HashMap<String, String>. Consiguiendo una complejidad de tiempo constante, ya que cada token del CSS sea traduce a su representación en binario de forma casi inmediata.

- **Clase auxiliar, bitWriter.java:** Esta clase se encarga de escribir el archivo de salida carácter a carácter usando un buffer de tamaño de 1 byte. En dicho buffer se irán introduciendo bit a bit los códigos Huffman y cuando se complete su capacidad se escribe en el archivo de salida.

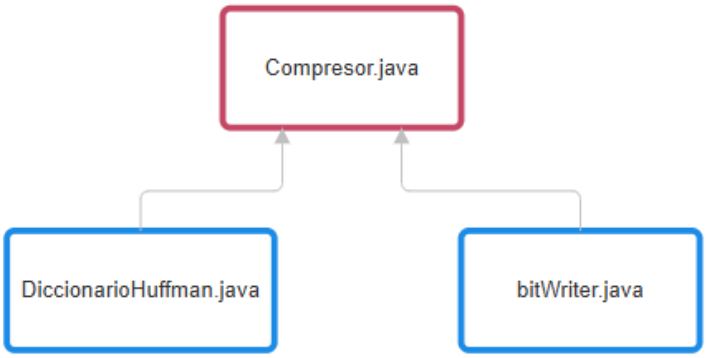

Ilustración 5 - relación clases proceso compresión

4.3.1.1 Flujo de Ejecución

Para garantizar una compresión sin errores el sistema sigue un ciclo de vida secuencial. Asegurando así que cada token sea procesado y escrito en el orden exacto.

El proceso se divide en las siguientes etapas:

- **Inicialización y Carga del Diccionario:** Al arrancar, la clase Compresor invoca un método de carga que lee el archivo arbolHuffman.txt. Cada entrada se almacena en una estructura especial de la clase DiccionarioHuffman.java que se explicara más adelante. Además, se escriben unos metadatos explicados en el punto 4.3.5.

- **Lectura por Búfer:** El archivo .css se abre mediante un BufferedReader. El sistema no carga el archivo completo en memoria, sino que lee carácter a carácter. Va guardando texto para ir formando palabras y en cuanto detecta un delimitador como un espacio cierra el token y lo envía a procesar

- **¿Existe el token?:** Para cada token identificado, la clase realiza una comprobación:
  - Caso A, que sea un token conocido: Si el token está en el HashMap, recupera su código de Huffman.

  - Caso B que sea un token desconocido: Si no está, se activa el protocolo de Código de Escape,

- **Empaquetado de Bits:** Una vez que tenemos el código binario, este se pasa a la clase BitWriter.java. Aquí es donde se escribe el código, pero no como texto si no que se van acumulando como 0 y 1 en un buffer de 1 byte.

- **Finalización:** Al llegar al final del archivo (EOF), se asegura que el último byte se complete (añadiendo ceros si es necesario) y se cierre el canal de salida, generando así el archivo final .cssc .

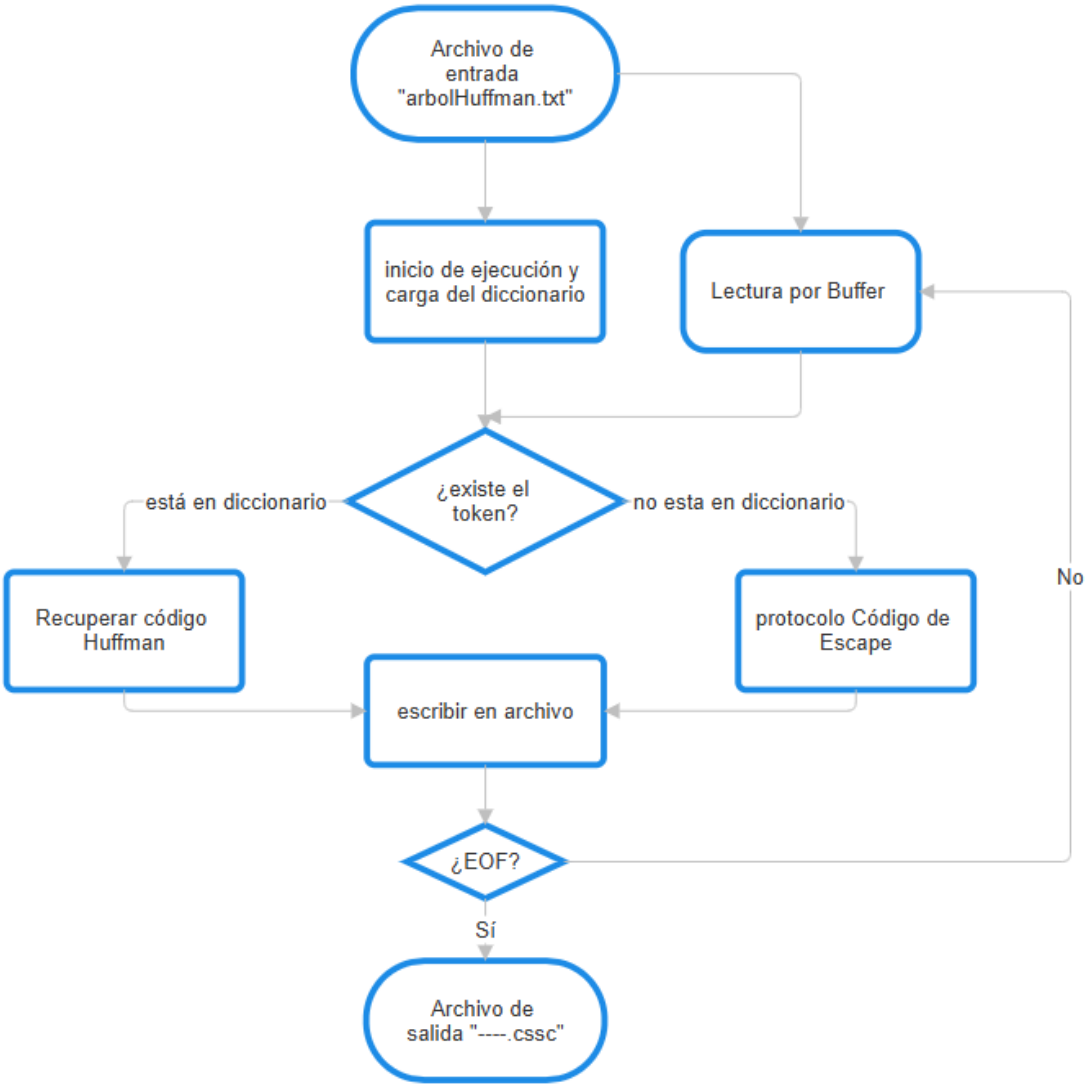

Ilustración 6 - diagrama de flujo de compresión

4.3.2 Gestión de Bits

El reto técnico de plasmar los códigos binarios de Huffman era que la clase usada, java.io.OutputStream, no permite escribir a nivel de bit solo a nivel de byte.[19] Los códigos asignados a los tokens son de longitud variable, por ejemplo el token "¡" puede ocupar solo 3 bits por lo que no podemos escribirlo ocupando un byte entero puesto que no ganaríamos compresión.

La solución fue desarrollar una clase auxiliar, bitWriter, cuya función es actuar como un empaquetador de bits agrupando los códigos binarios de 8 en 8. En lugar de intentar escribir cada código de manera independiente, se utiliza un buffer interno de un solo byte que se va rellenado dinámicamente.

Mediante operaciones binarias de bajo nivel, desplazando hacia la izquierda y usando mascaras lógicas el sistema posiciona cada bit del código Huffman en el lugar exacto donde corresponde.

Este proceso es trasparente para el compresor, el sistema solo envía una secuencia de bits a la clase bitWriter y esta ejecuta un volcado al disco cuando detecta que el buffer se encuentra lleno. Cuando se alcanza el carácter de EOF se completa el ultimo byte con ceros (padding) y se vuelca la información.

Estas operaciones a nivel de bit es lo que permite que el ahorro teórico de espacio se muestre en el tamaño final del archivo.

4.3.3 Interfaz de Traducción:

Si el BitWriter representa la capacidad de ejecución binaria, la clase DiccionarioHuffman constituye el repositorio de información del sistema. Esta clase se encarga de cargar el archivo que guarda los pares token-código binario, se convierte en el puente entre los análisis estadísticos y el uso de estos.

Técnicamente, el diseño de esta clase se fundamenta en la eficiencia de acceso aleatorio. Al iniciarse la compresión, la clase carga el archivo, el diccionario compartido, y vuelca la información en una estructura de datos de tipo HashMap. Esta elección no es casual, al utilizar una tabla de dispersión, el sistema garantiza que la búsqueda de cualquier propiedad CSS tenga una complejidad baja y de rendimiento constante [19]. Este elemento es muy importante ya que evitamos que el proceso de búsqueda escale de manera lineal con el tamaño de diccionario, permitiendo una traducción casi inmediata independientemente de si el token se encuentra al principio o al final de la estructura.

Más allá del simple mapeo, la clase DiccionarioHuffman gestiona la integridad del archivo de lectura. Se encarga de validar que cada código binario recuperado sea coherente con la propiedad de prefijo, garantizando que ningún código sea el inicio de otro, y prepara la lógica de señalización necesaria para el descompresor.

4.3.4 El Código de Escape (CE):

Aunque en el lenguaje CSS las propiedades y símbolos sean finitos, existen una infinidad de elementos impredecibles, por ejemplo, nombre de clases o valores numéricos. Para que el sistema genera un archivo comprimido sin pérdida no se puede simplemente ignorar estos elementos, se debe ser capaz de codificarlos, aunque no figuren en el árbol.

Para resolver esta limitación se ha implementado un mecanismo de señalización denominado código de escape (CE). Este actúa como una bandera al momento de encontrarse en el flujo de datos binarios. Y cuando el compresor identifica un token que no reside en el HashMap del diccionario, el sistema ejecuta los siguientes pasos:

1. **Señalización:** Se escribe en el flujo de salida el código de Huffman correspondiente al nodo de "Escape". Este código se encuentra en el árbol binario de Huffman en una posición que varía según el corpus de archivos y el % que cubre el árbol.

2. **Indicador de Longitud:** Después de la señal de escape, el sistema inserta un byte, un valor numérico que representa la cantidad exacta de bytes que componen el token literal. Este dato indica al descompresor cuantos bytes debe leer en modo texto plano antes de reactivar la lógica Huffman.

3. **Representación Literal:** Finalmente, el sistema escribe el token desconocido en su formato original carácter a carácter.

Aquí se muestra como quedaría un código escape completo:

Código de escape (CE) + longitud de token literal + token literal.

Asi se permite una transición entre lectura modo Huffman y lectura literal. De esta manera se garantiza la compatibilidad total con cualquier archivo CSS.

Uno de los mayores retos de este trabajo ha sido tratar de posicionar el CE dentro del árbol binario de Huffman. Para que este mecanismo sea efectivo, el Código de Escape debe integrarse como un elemento más dentro de la lógica del árbol de Huffman.

La asignación de su secuencia binaria no es aleatoria, sino que se define durante la fase de modelado en Python como un token especial reservado. Al construir el árbol, se le asigna al CE una frecuencia teórica, basada en el % de tokens no cubiertos en el corpus. Se le asigna un código Huffman que varía según la cobertura y se posiciona en el árbol.

Si definiéramos el CE desde un principio sin tener en cuenta lo anteriormente mencionado, podrían ocurrir dos cosas. Si el código binario fuera muy corto, del rango de 3 a 5 bits le quitaría el puesto a token con un gran peso haciendo que la compresión total se redujese. Por el contrario, si se le asigna un código binario extenso también perjudicaría a la compresión, ya que añadiría sobrecarga en el archivo de salida cada vez que se encontrase un token no estipulado en el diccionario. Guardar un token que no posea código binario asociado supone

copiar todo el literal junto con un byte de longitud y además el CE. Por lo cual se optó por una asignación dinámica

4.3.5 Cabecera de Control

Para garantizar que el sistema identifique los archivos compatibles, se ha diseñado una cabecera de control personalizada que se sitúa al inicio de cada archivo .cssc . Esta cabecera actúa como una firma digital (magic number) que el compresor escribe antes de volcar cualquier bit de datos proveniente del algoritmo de Huffman.

Estos metadatos son, un identificador del formato, el magic number, siendo este “CSSC”. Y un long con la cantidad de tokens guardados, para verificar que no ha habido ningún error al momento de descomprimir y que no ha variado el archivo.

4.4 Implementación del Descompresor

El módulo de descompresión es la fase final del ciclo de vida del sistema, cuyo trabajo es revertir el proceso de codificación para recuperar el archivo CSS original. El descompresor opera bajo una lógica de decodificación guiada por el diccionario compartido. Para ello se debe cargar como hemos dicho con anterioridad el mismo diccionario que se usó para comprimir.

El reto se basa en la ambigüedad, el sistema debe ser capaz de distinguir si un bit pertenece a la navegación por el árbol, si es una hoja o es el comienzo de un código escape.

Para lograrlo, el descompresor no usa de tablas de búsqueda (HashMap), opta por una arquitectura basada en nodos enlazados.

4.4.1 Arquitectura del Descompresor y Reconstrucción del Árbol

La arquitectura del descompresor ha sido diseñada como un proceso de decodificación, donde el sistema reconstruye la información original mediante la navegación del árbol binario. En la descompresión el desafío técnico se basa en la interpretación precisa de un flujo de bits sin delimitadores físicos. Extrayendo códigos que posteriormente se traducen a tokens.

Los componentes centrales son:

- **Clase Orquestadora, Descompresor.java:** Es el motor central, encargado de coordinar la lectura del archivo comprimido .cssc y la escritura del archivo final. Su función principal es gestionar y supervisar la lógica entre la navegación del árbol binario y la lectura de texto plano cuando se detectan secuencias de escape.

- **Clase auxiliar, NodoHuffman.java:** Cada objeto de esta clase actúa como un bifurcador, izquierdo para el bit 0 y derecho para el bit 1. Esta estructura sustituye al HashMap del compresor, ya que permite que la recuperación de tokens no requiera búsquedas de claves, sino simples saltos entre punteros**.**

- **Clase auxiliar, bitReader.java:** Escribe en el archivo de salida y proporciona los bits necesarios para la lógica de la Descompresión. Esto garantiza una copia de la original de Python asegurando que cada secuencia de bits lleve inequívocamente al token que fue comprimido.

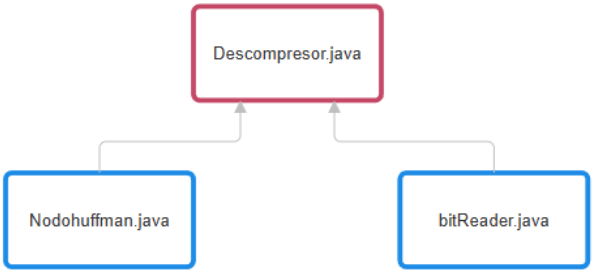

Ilustración 7 - diagrama de relación de clases

4.4.1.1 Flujo de Recuperación de Datos

La recuperación de datos es un proceso de descubrimiento bit a bit, donde el significado de cada unidad de información solo se conoce al alcanzar un estado de hoja en el árbol.

El flujo operativo es el siguiente:

- **Sincronización y Posicionamiento:** El descompresor inicia situando un puntero en el Nodo Raíz del árbol binario reconstruido. Al mismo tiempo se abre el archivo .cssc y se extrae la meta información, que contiene el número de tokens totales a leer y se comprueba que es un archivo comprimido de CSS (CSSC). Por último, se carga el primer bit para preparar el proceso de decodificación.

- **Navegación Binaria:** Por cada bit procesado, el orquestador mueve el puntero hacia el hijo izquierdo o derecho del nodo actual, según si es 0 o 1. Este recorrido se mantiene de forma recursiva mientras el nodo visitado sea una rama del árbol.

- **Identificación de Hoja y Extracción:** En el momento en que el puntero alcanza un Nodo Hoja, el sistema detiene la navegación. Y el contenido de la hoja, el token original, es enviado al buffer de salida para ser escrito en el archivo .css . Tras esto el puntero regresa a la raíz de árbol y repite el proceso.

- **Gestión de Código de Escape:** Si durante la navegación el sistema se posa en la hoja reservada para el Código de Escape, el flujo de navegación se suspende. Se activa el protocolo de lectura literal, consulta la longitud de bytes a leer y se copian literalmente en el buffer de salida. Una vez finalizado se reinicia el ciclo de búsqueda de códigos Huffman.

- **Cierre y Verificación de Integridad:** El ciclo continúa hasta que el lector de bits detecta el final del archivo. Ignora los bits del padding y se asegura que el número de tokens extraído concuerde con el número de los metadatos.

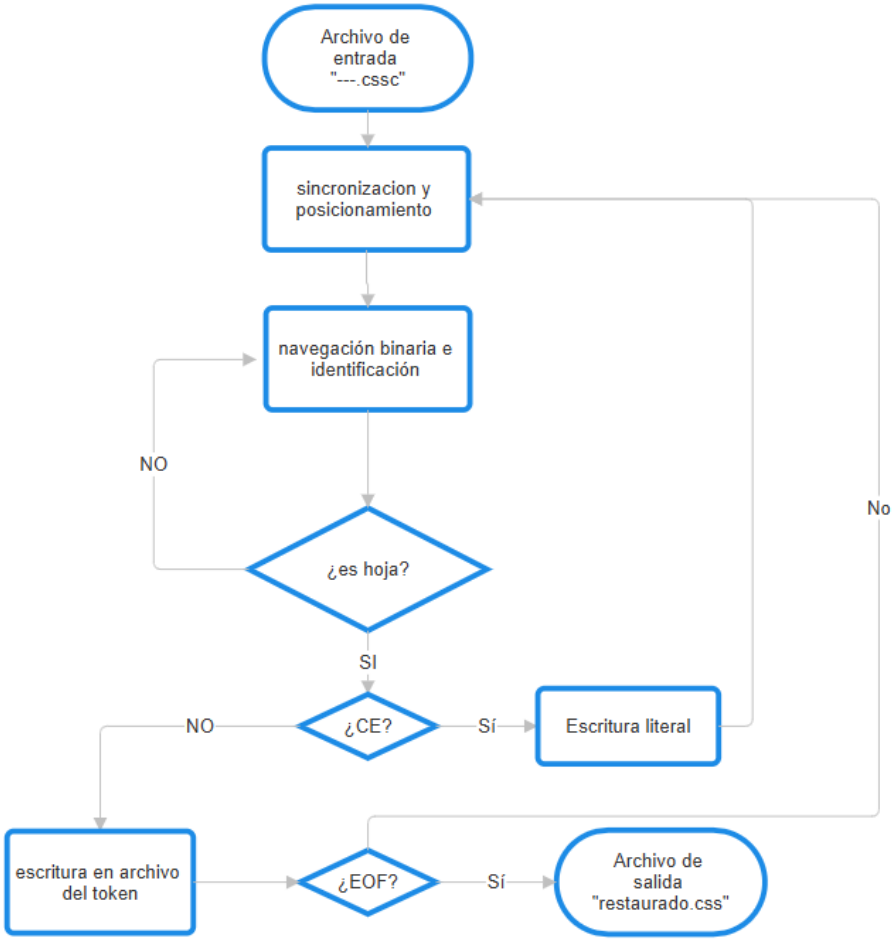

Ilustración 8 - diagrama de flujo de descompresión

4.4.2 Lectura a Nivel de Bit

Si la compresión requería un empaquetador de bits, la descompresión exige un mecanismo de extracción. La clase BitReader se ha implementado para cumplir esta función, permitiendo que el descompresor modifique la estructura de bytes propia de la biblioteca java.io.InputStream.

La clase solicita un byte completo al buffer de entrada (FileInputStream), mediante desplazar hacia la derecha y el uso de máscaras, se entregan bits de uno en uno cada vez que es llamada. No es necesario saber en qué parte del byte esta solo que se proporcionen bits.

4.4.3 Algoritmo de Decodificación y Navegación de Nodos

El algoritmo de navegación del descompresor trasforma secuencias de bits en tokens con significado semántico. Este proceso se basa en un camino de estados, donde cada bit nuevo determina un nuevo sendero hacia un nivel de profundidad mayor en el árbol.

El sendero comienza invariablemente en el nodo raíz de la estructura, desde donde el sistema inicia un descenso condicionado por la recepción de un bit 0 desplaza el puntero de ejecución hacia el hijo izquierdo, o un bit 1, que lo dirige hacia el hijo derecho. Esta exploración recursiva se mantiene de forma constante mientras el sistema visite nodos internos (ramas del árbol).

La navegación solo se detiene al alcanzar un estado de nodo hoja. En ese instante, el sistema extrae la cadena de texto almacenada en dicha hoja y la transfiere al buffer de salida para su escritura en el archivo .css.

Esta metodología elimina la necesidad de realizar comparaciones de cadenas o búsquedas en tablas de datos, permitiendo que la descompresión se ejecute con una carga computacional mínima.

4.4.4 Gestión del Código de Escape

Una parte importante del proceso de descompresión reside en su capacidad para cambiar entre la interpretación de cadenas de código Huffman y la extracción de datos en crudo. Este momento de cambio se produce cuando el algoritmo de navegación alcanza la hoja del árbol dedicada al CE. A partir de ese instante el sistema reconoce que los bits siguientes no representan un camino dentro del árbol de Huffman, sino metadatos que preceden a una cadena de texto que no pudo ser optimizada durante la fase de entrenamiento.

El sistema extrae del flujo binario el valor numérico que representa la cantidad de bytes literales que debe procesar. Esta información es de vital importancia ya que sino no tendríamos manera de saber cuándo acaba la cadena de literales. Una vez establecida la longitud, el descompresor lee los bytes correspondientes, los traduce a su representación de caracteres original y los vuelca directamente en el archivo de salida .css.

Tras acabar de leer los bytes literales el sistema restablece el puntero en el nodo raíz y comienza de nuevo con el algoritmo de navegación. Esto permite

que todo el contenido del archivo original se plasme en el archivo descomprimido.

4.5 Estructura y Organización del Software

Para garantizar la facilidad de uso y la satisfacción del usuario, como se indicó, se desarrolló un documento Makefile al igual que un ReadMe.

Empezando por la organización del proyecto, este se ha estructurado en directorios que separan el código fuente de los corpus de entrenamientos y archivos generados. Asi dividimos las responsabilidades entre scripts de análisis en Python y el motor de procesamiento en Java.

A continuación, se detalla la jerarquía de carpetas y la función de cada componente:

```
/
├── Makefile              # Script de automatización de compilación y ejecución
├── README.md             # Archivo de documentación
├── tokens.txt     #(Generado por el sistema)Frecuencias de tokens del corpus
├── arbolHuffman.txt   #(Generado por el sistema)Diccionario de códigos
                                binarios Huffman
├── corpus/           # Carpeta con archivos .css para el entrenamiento
|── archivo.css          # Archivo para comprimir/descomprimir
├── src/                  # Código Fuente
│   ├── analizadorCSS.py     # Extrae estadísticas del corpus
│   ├── arbolHuffman.py      # Genera el árbol binario y los códigos
│   ├── compresor.java       # Codifica el archivo destino a binario (.cssc)
│   ├── descompresor.java    # Restaura el archivo original
│   ├── bitWriter.java       # Gestión de escritura a nivel de bits
│   ├── bitReader.java       # Gestión de lectura a nivel de bits
│   ├── NodoHuffman.java     # Estructura de datos del árbol
│   └── DiccionarioHuffman.java # Carga del modelo en memoria
└── bin/  (Generado por el sistema) Clases compiladas de Java
```

Esta disposición permite que el sistema sea escalable, facilitando la actualización del corpus, y asegura que todos los archivos generados durante el proceso estén claramente localizados.

5 Pruebas y Resultados

En este capítulo se presenta como hemos evaluado el sistema de compresión desarrollado, analizando su eficacia en diversos escenarios. El objetivo de estas pruebas es validar la hipótesis que se ha planteado al inicio del trabajo de fin de grado, que una tokenización semántica orientada al lenguaje CSS, combinada con un modelo de diccionario compartido, ofrece ventajas frente a los métodos de compresión estadística tradicionales.

Para realizar la evaluación, se han diseñado diferentes baterías de pruebas que cubren tres ejes:

- **La optimización del tamaño del árbol:** determinar el punto óptimo del tamaño del diccionario para equilibrar la tasa de compresión y la sobrecarga de metadatos.

- **La especialización del dominio:** analizar como responde el sistema ante archivos con diferentes procedencias, estilos y corpus analizados.

- **La eficiencia comparativa**: Enfrentar el algoritmo propuesto contra el estándar Huffman a nivel de carácter y contra compresores consolidados como Gzip.

Los resultados no solo permitirán saber el ahorro de espacio en disco y uso de ancho de banda si no revelar nichos de rendimientos específicos.

5.1 Metodología de Evaluación

Para validar el rendimiento del sistema de compresión, se ha establecido una serie de pruebas estructuradas que permite analizar el comportamiento del algoritmo bajo diversas condiciones de carga y estructura de datos.

5.1.1 Selección del Corpus de Prueba

La evaluación se ha fundamentado en el uso de tres conjuntos de datos, lo que llamaremos corpus.

- **Corpus genérico:** Compuesto por una gran mezcla heterogénea de archivos procedentes de diversos marcos de trabajo. Unos 40 archivos CSS con un uso estándar de la web.

- **Corpus específicos:** varios grupos de archivos extraídos de plataformas como navegadores o sitios web comerciales. Estos corpus presentan una alta personalización y una redundancia semántica más específica. Cada corpus se compone de entre 8 a 15 archivos.

- **Corpus Minificado**: Versiones de los archivos anteriores tras un proceso de eliminación de comentarios y espacios en blanco, con el objetivo de

medir la eficiencia del algoritmo sin el ruido del formato visual. El corpus se compone de 20 archivos CSS.

5.1.2 Métricas y Herramientas de Comparación

Para determinar la eficacia del sistema, se han definido una serie de métricas

Métricas de rendimiento:

- **Factor de Compresión Relativo**: Comparación directa del tamaño en bytes del archivo original frente al archivo comprimido .cssc.

- **Porcentaje de Cobertura de Tokens**: Relación entre los tokens almacenados en el diccionario y los tokens totales del corpus. Un bajo porcentaje de cobertura implica un mayor uso del Código de Escape.

La viabilidad del proyecto se basa en la comparación de dos metodologías de compresión existentes:

- **Huffman Tradicional:** Se utiliza para afirmar la superioridad de la tokenización semántica. Mientras que el Huffman estándar analiza la frecuencia de letras individuales, nuestra propuesta analiza la frecuencia de palabras completas.

- **Gzip:** Representa el estándar industrial. Se utiliza para identificar los límites del sistema propuesto. Gzip es increíblemente eficiente en archivos grandes gracias a su ventana deslizante LZ77, además de contar con un nivel más de compresión, pero presenta un *overhead* de metadatos.

5.1.3 Entorno de pruebas

La ejecución de las pruebas se llevo a cabo en un entorno basado en Linux. Para coordinar estos módulos, se implementó un flujo de trabajo automatizado, un MakeFile, para garantizar que cada prueba se realizara sobre la versión adecuada del diccionario y bajo las mismas condiciones de entorno, eliminando variaciones por errores humanos en la cadena de comandos.

El ciclo comenzó con una fase de análisis, donde el script Python procesaba el corpus seleccionado para extraer las frecuencias de los tokens. Una vez generado el árbol binario de Huffman el motor en Java tomaba el control para realizar la codificación del archivo CSS original. Cada prueba incluyó una etapa de validación mediante la descompresión total del archivo .cssc y posterior comparación de archivos para verificar la integridad absoluta.

5.2 Análisis de Optimización del Diccionario

Determinar el tamaño ideal del diccionario compartido forma uno de los puntos importantes del proyecto, ya que define el equilibrio entre la tasa de compresión y la eficiencia operativa del sistema.

Para identificar este punto se realizaron numerosas pruebas variando la cantidad de entradas en el árbol de Huffman, desde los 500 hasta los 5000 tokens, observando el impacto en la cobertura y en el uso del Código de Escape.

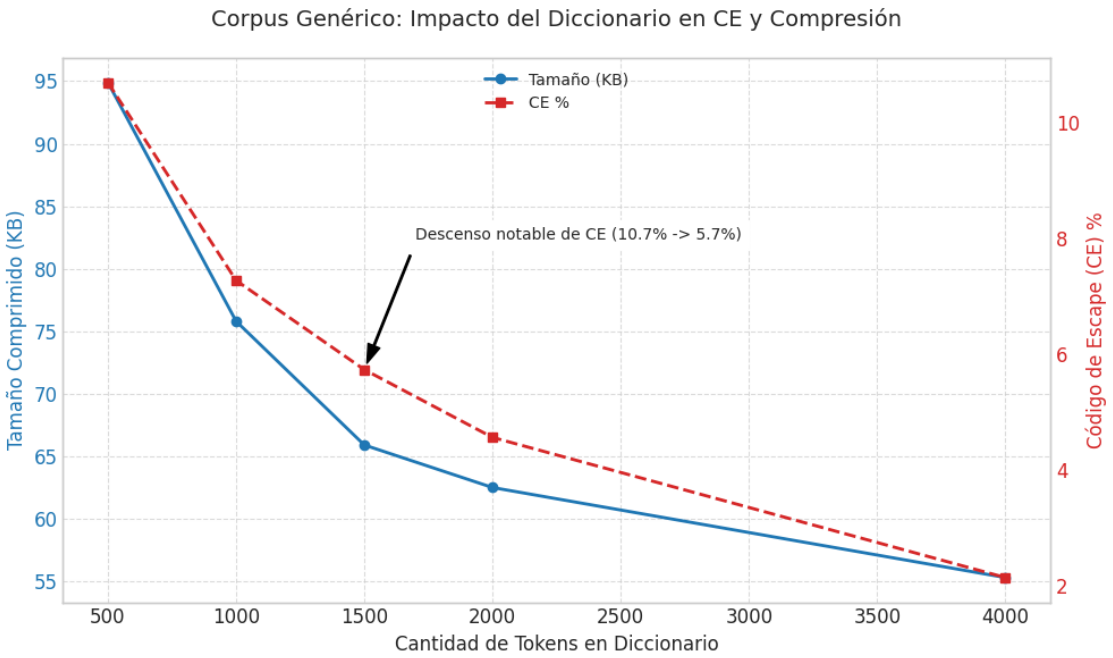

Ilustración 9 -Grafica impacto cobertura CE

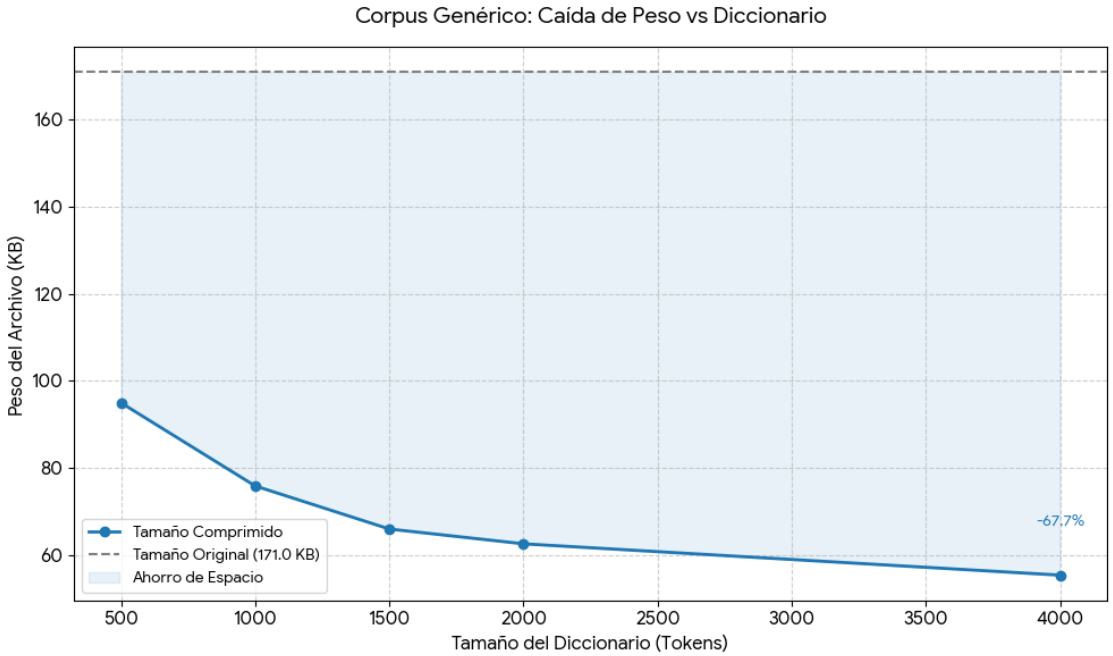

Ilustración 10 - Grafica reducción peso/ aumento tamaño árbol

En el corpus genérico, que presenta la mayor variabilidad de datos, se observó que un diccionario reducido de 500 tokens arrojaba un CE elevado del 10,7%, penalizando la compresión. Sin embargo, al ir escalando hacia los 1500 tokens, el CE descendió al 5,74%, logrando una reducción de tamaño significativa sin que el peso del archivo del árbol fuera excesivo

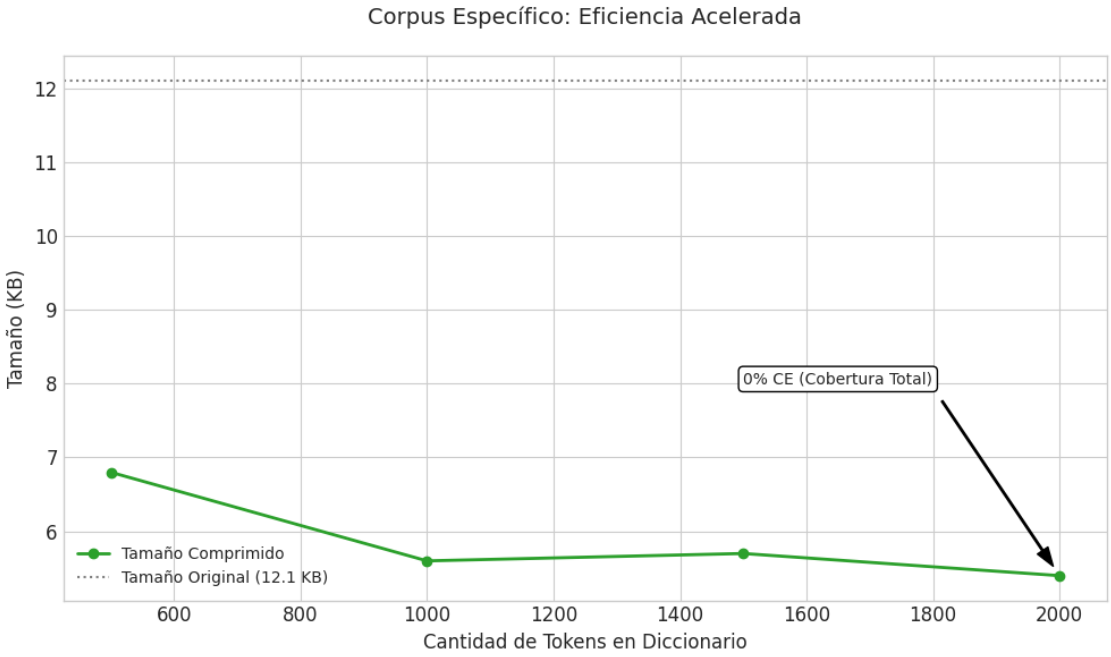

Ilustración 11 - Gráfica muestra CE 0%

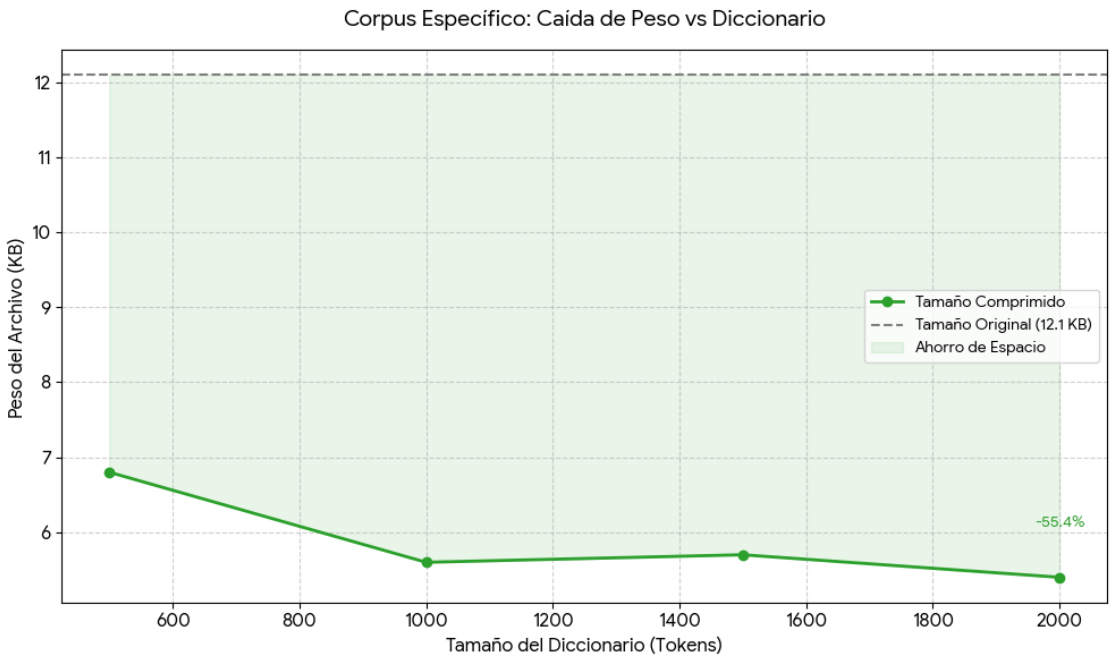

Ilustración 12 - Gráfica corpus especifico 0% CE

Al analizar varios corpus específicos, los resultados muestran una eficiencia mucho más acelerada. El sistema alcanzo el 0% de CE al llegar a los 2000

tokens en algunos corpus. Significando que todos los tokens estarian contemplados en el arbol.

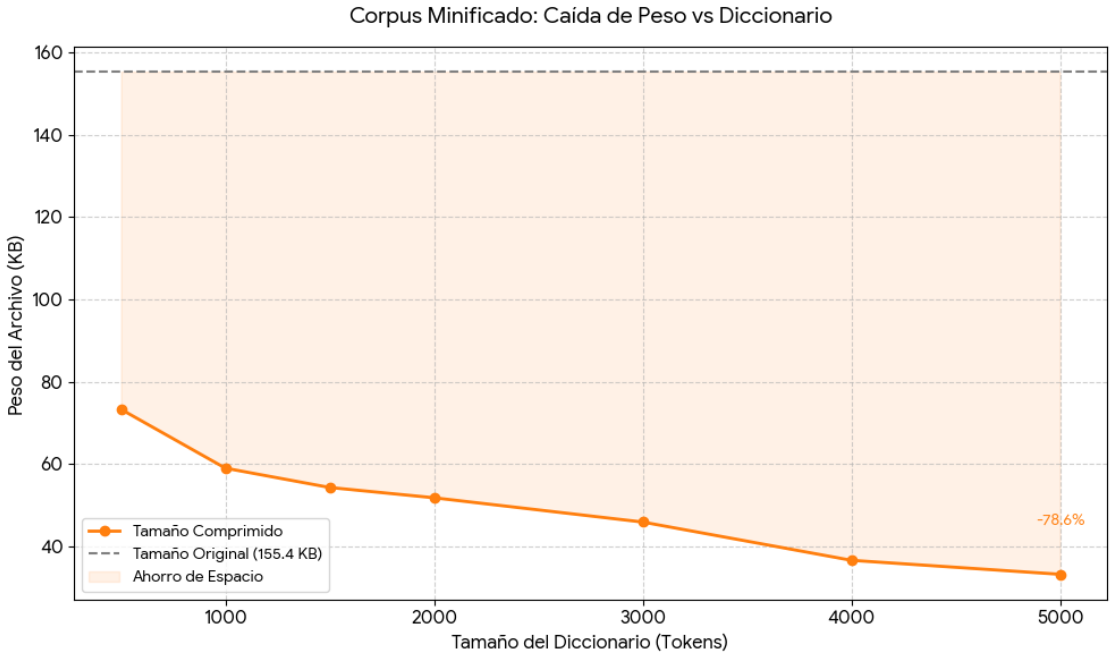

Ilustración 13 - Gráfica corpus minificado aumento compresión/ aumento tamaño árbol

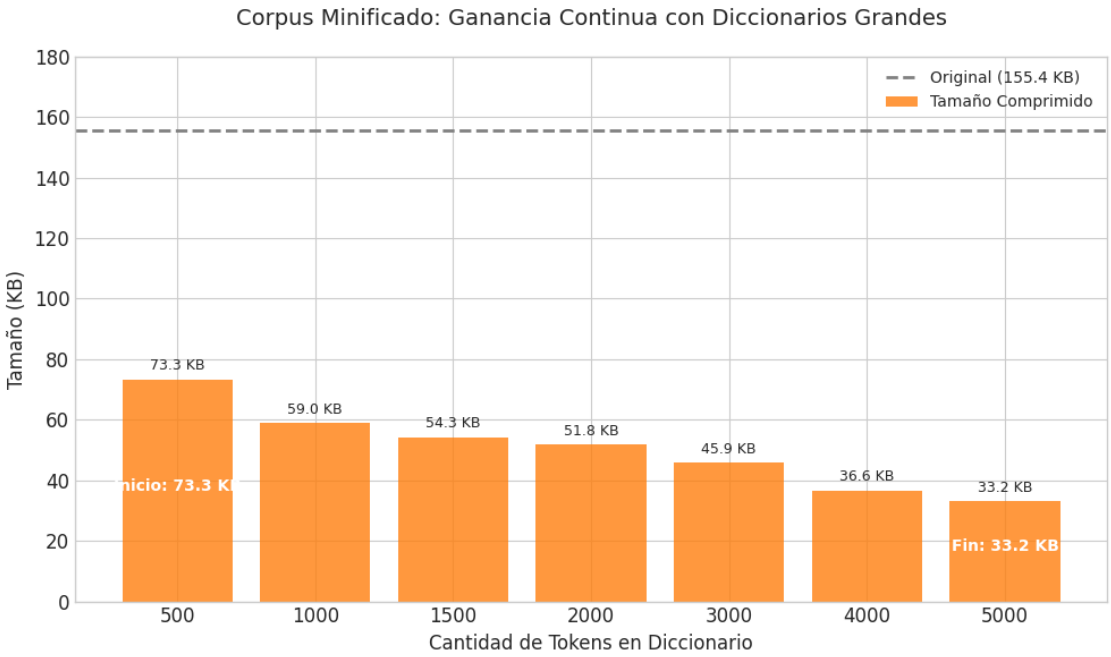

Ilustración 14 - Gráfica corpus minificado progresión compresión

En el corpus minificado, el comportamiento del árbol es distinto, al eliminar los espacios y comentarios, la densidad de tokens lógicos aumenta, lo que permite que incluso con diccionarios grandes, de hasta 5000 tokens, se sigan obteniendo ganancias de compresión notables, bajando de 155,4 KB a 33,2 KB.

No obstante, la continua expansión del diccionario plantea un dilema sobre el coste de metadatos. Cuanto mayor es el diccionario mayor es la compresión, pero mucho mayor es el archivo compartido a enviar.

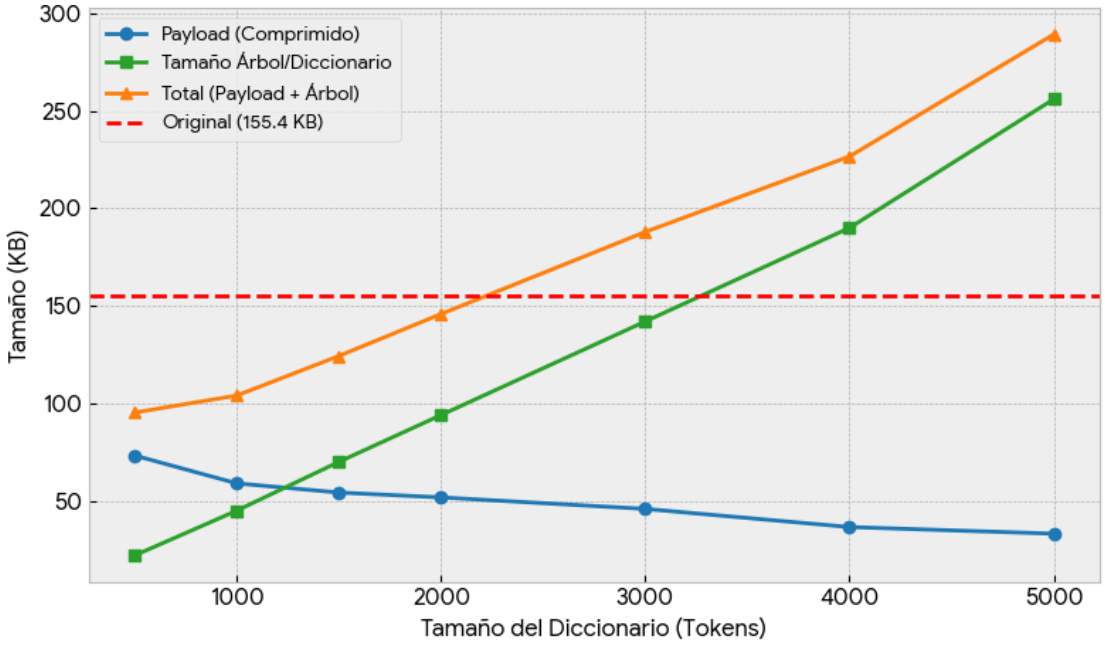

Ilustración 15 - Gráfica comparativa ganancia/pérdida por aumento de tamaño árbol

Los datos recogidos muestran que mientras un diccionario de 1500 tokens genera un archivo de soporte de 70 KB, subir a 5000 tokens dispara este peso hasta los 256 KB

Por ello, tras contrastar los resultados de rendimiento, se determinó que el rango de 1500 a 2000 tokens representa el punto de equilibrio óptimo, garantizando una cobertura robusta de los corpus de archivos CSS con un archivo de diccionario ligero.

5.3 Tokenización Semántica vs. Codificación por Caracteres

En este capítulo se analiza la eficacia de la innovación del proyecto, la sustitución de compresión por carácter por compresión por token. Al comparar el compresor desarrollado frente a una implementación de Huffman estándar, los resultados obtenidos confirman que el conocimiento de la estructura del lenguaje CSS permite una compresión de datos significativamente mayor.

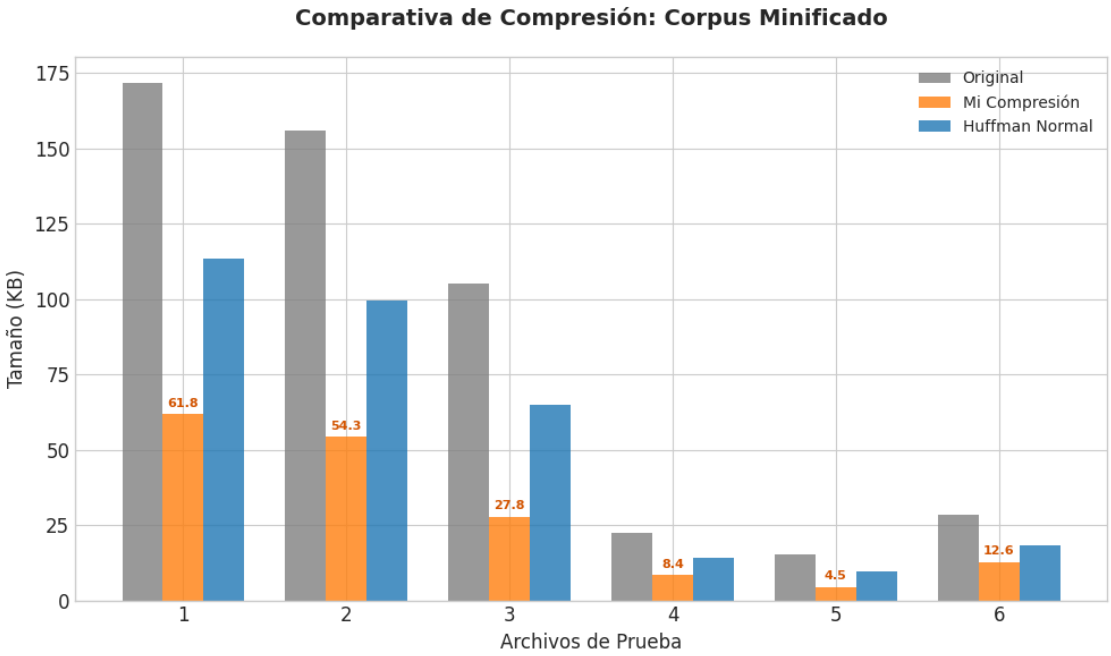

Ilustración 16 - Diagrama de barras corpus minificado contra Huffman genérico

En el corpus minificado, por ejemplo, la diferencia es enorme, para un archivo original de 171,8 KB, el Huffman tradicional solo logra reducirlo a 113,4 KB, mientras que la compresión semántica alcanza los 61,8 KB, demostrando que tratar propiedades como background-color como un único símbolo ahorra hasta un 45% de espacio adicional.

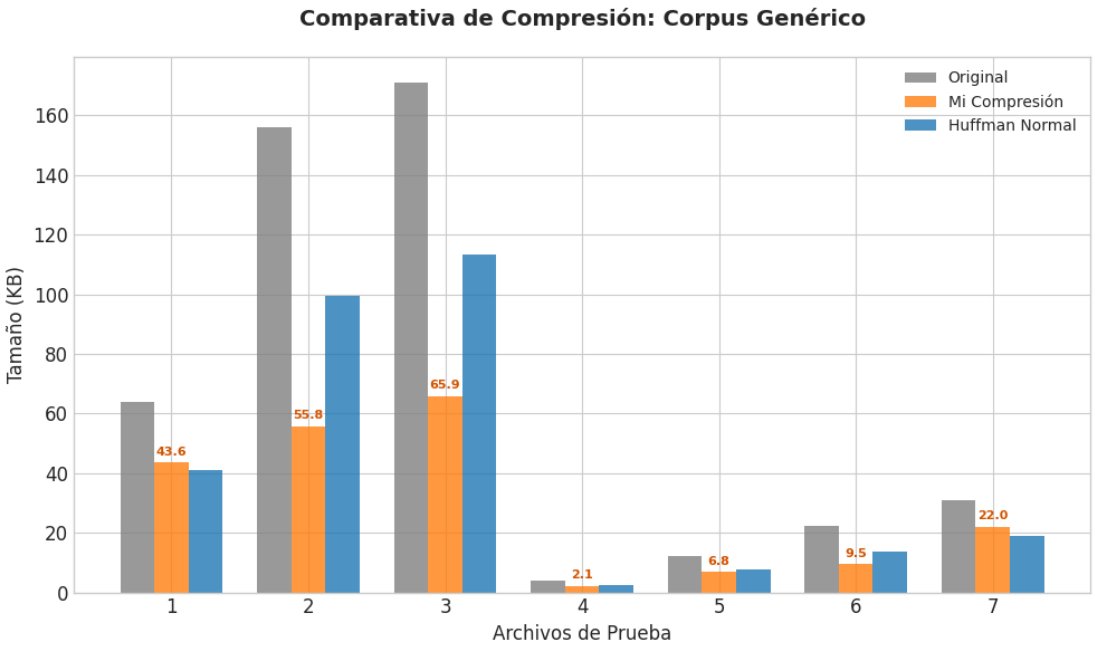

Ilustración 17- Diagrama de barras corpus genérico contra Huffman genérico

El comportamiento en el corpus genérico mantiene esta tendencia de superioridad, logrando reducciones constantes frente al método tradicional en casi todas las pruebas realizadas.

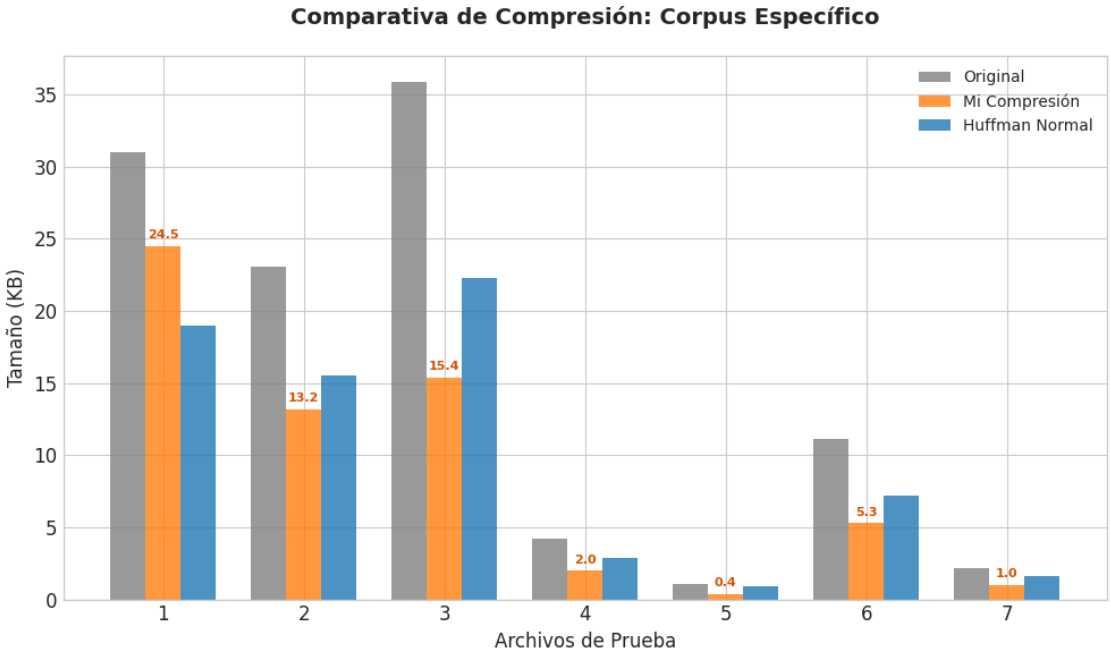

Ilustración 18- Diagrama de barras corpus genérico contra Huffman genérico

De igual manera en el corpus especifico la compresión propia supera a la genérica en casi todos los casos. Sin embargo, se quiso probar los límites del sistema comprimiendo archivos CSS totalmente diferentes a los analizados en el corpus especifico y guardados en el diccionario compartido.

Donde la compresión semántica (24,5 KB) resultó ligeramente superior en tamaño al Huffman tradicional (19 KB) como se puede observar en la primera comparación de las gráficas. Esta serie de pruebas confirman que la dependencia del diccionario compartido es seria.

Al no encontrar los tokens en el diccionario estático, el sistema se vio obligado a recurrir de forma constante al Código de Escape (CE) y a la escritura de literales, lo que introduce una sobreescritura de señalización que el Huffman tradicional no sufre.

5.4 Comparativa con Gzip

El examen final del sistema se realiza frente a Gzip, el estándar consolidado en Unix. Superar el rendimiento de Gzip representa un desafío técnico muy difícil de superar, pero nuestro objetivo tampoco es ese.

Los resultados obtenidos en las diferentes baterías de pruebas identifican escenarios específicos donde el diccionario compartido ofrece una alternativa.

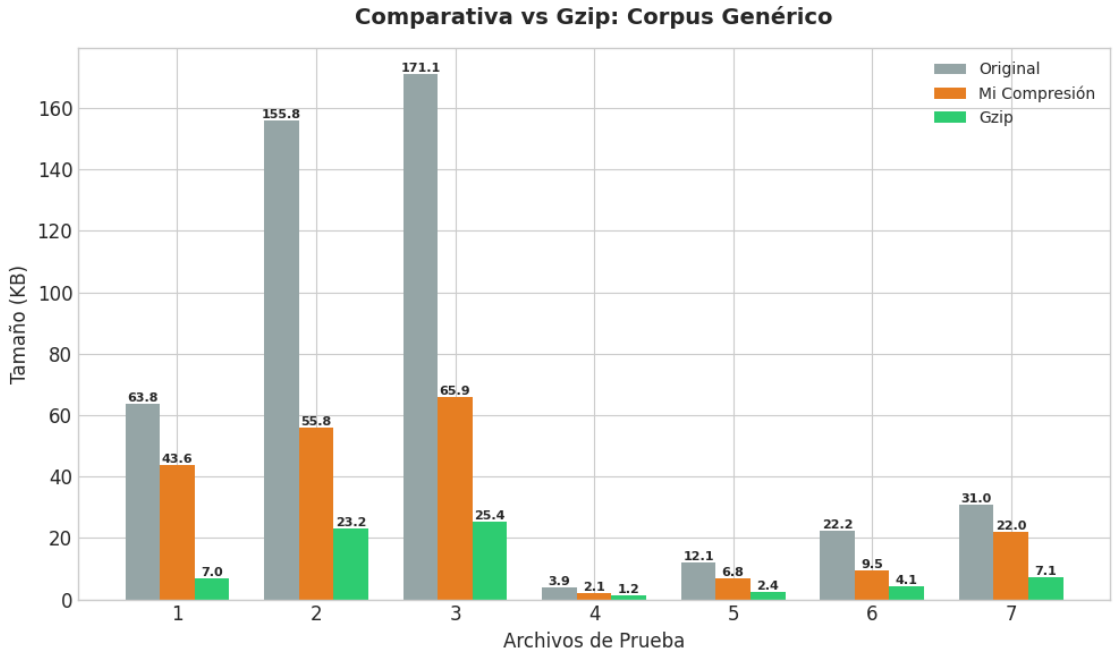

Ilustración 19- Diagrama de barras corpus genérico comparativa con Gzip

En archivos de tamaño medio y grande como podemos observar en las pruebas del corpus genérico, Gzip muestra una clara superioridad, logrando tamaños finales de 7 KB y 25,4 KB respectivamente frente a los 43,6 KB y 65,9 KB de nuestra solución. Esta diferencia se explica por la capacidad de Gzip para referenciar cadenas previas, mientras que nuestro sistema se limita a los tokens predefinidos en el diccionario

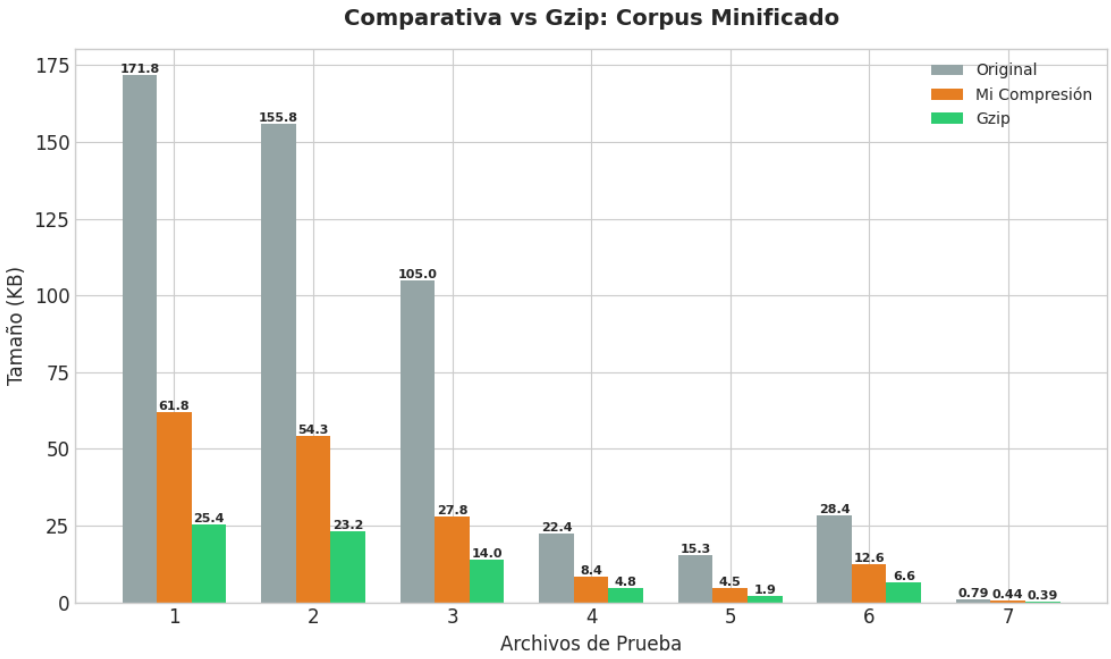

Ilustración 20 - Diagrama de barras corpus minificado comparativa con Gzip

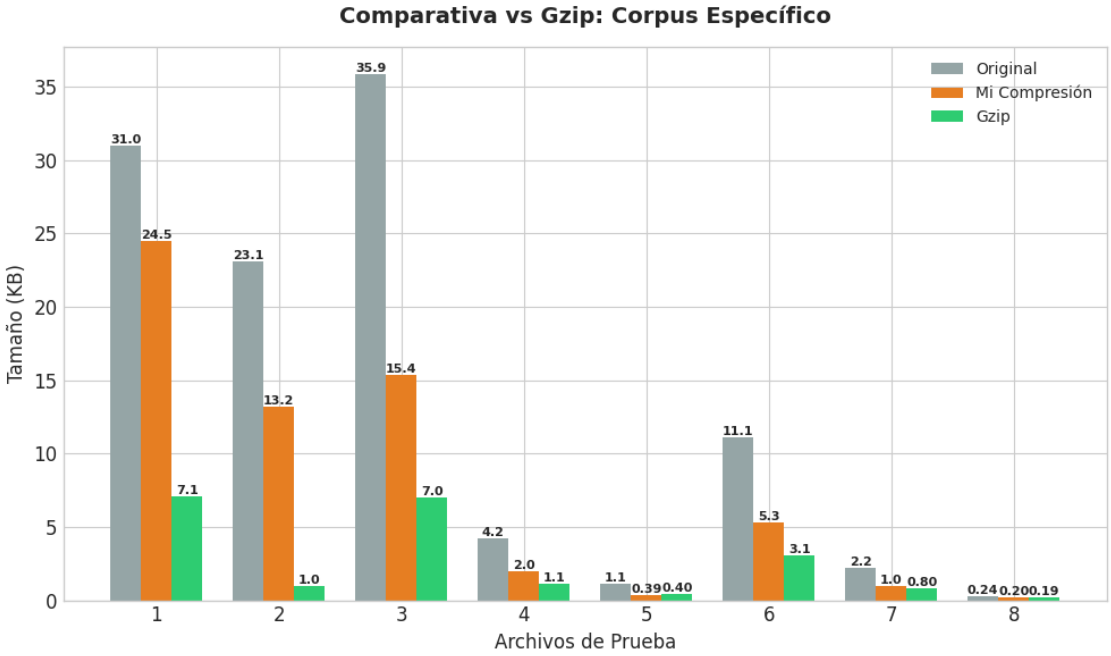

Ilustración 21 - Diagrama de barras corpus específico comparativa con Gzip

Sin embargo, al analizar el corpus específico, y en menos medida en el minificado, se alcanzan pruebas donde la diferencia de rendimiento es mínima. La diferencia es de apenas unas decenas de Bytes, repitiéndose de manera clara. Incluso en un resultado la supera. Una prueba con un archivo original de 1.1 KB, nuestro sistema desarrolla una compresión de 400B mientras que Gzip 409B.

La razón técnica de esta victoria reside en el sobrecoste de información. Gzip debe incluir en cada archivo comprimido las cabeceras, la estructura de bloques

y, en ocasiones, los árboles Huffman. En archivos muy pequeños, este peso de metadatos es muy alto. Por el contrario, nuestro sistema utiliza un diccionario compartido que se asume que lo posé el receptor, lo que permite que el archivo .cssc contenga exclusivamente flujo de datos útil.

5.5 Análisis de casos de éxito

Más allá de las comparativas con otros algoritmos, es fundamental analizar los escenarios donde el sistema propuesto alcanza su máximo. Los resultados obtenidos demuestran que, bajo condiciones óptimas, es posible alcanzar tasas de compresión que superan el 80% de reducción respecto al tamaño original.

Estos casos se observan con especial claridad en el corpus minificado, donde la eliminación visual, como espacios y comentarios, permite que el motor de Huffman trabaje exclusivamente sobre una gran densidad semántica.

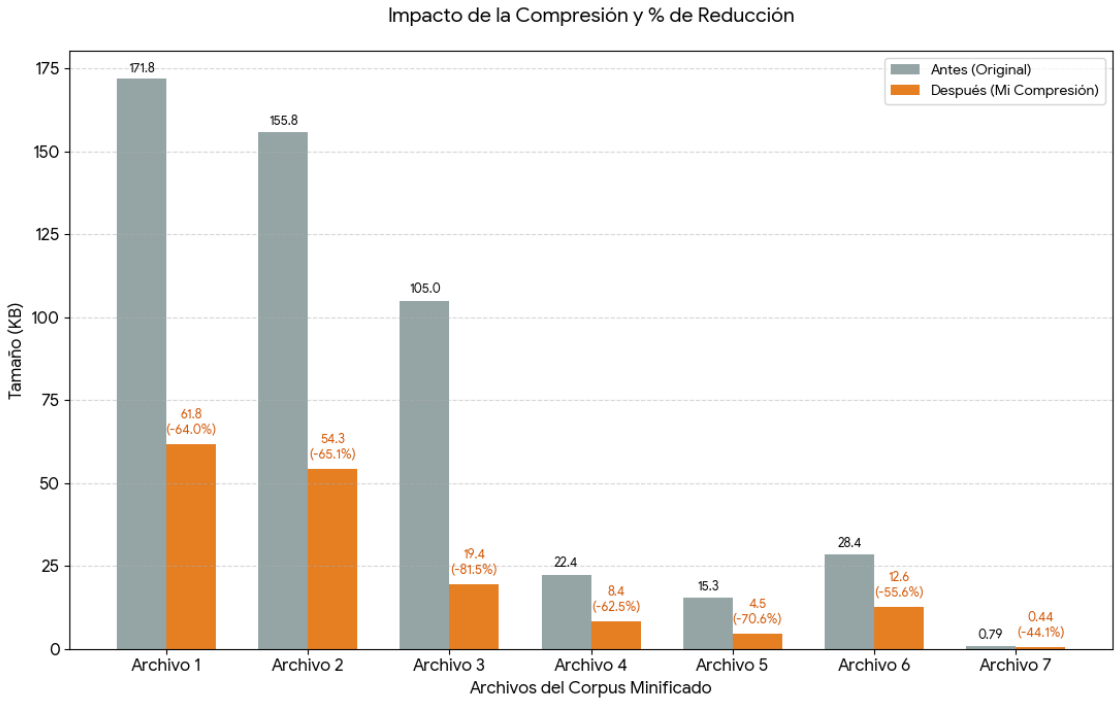

Ilustración 22 - Diagrama de barras nivel de compresión conseguido

En esta grafica podemos observas algunos casos de gran compresión, pero destacar la compresión de un archivo CSS de 104,5 KB, el cual, tras ser procesado por el compresor, se redujo a tan solo 19,4 KB. Este caso supone una tasa de compresión del 81,5%. Una cifra que evidencia la potencia de tratar propiedades y valores como tokens únicos.

Otro ejemplo se puede ver en la misma grafica de pruebas del corpus minificado, otro archivo de 15.3 KB fue comprimido satisfactoriamente hasta los 4,5 KB, manteniendo una reducción superior al 70%.

Estos resultados propios confirman que el sistema no solo es una alternativa teórica, sino una herramienta practica que puede servir como base para compresores más sofisticados.

6 Conclusiones

En este capítulo de reflexión se analizan los resultados derivados del diseño, implementación y resultados del proyecto. A través de la comparación de resultados se extraen lecciones que ahora analizaremos.

6.1 Fortalezas del Sistema

El principal éxito de este proyecto reside en la validación de que la tokenización como una mejora frente a tratar los archivos byte a byte puede lograr, de manera constante y fiable, superar a modelos genéricos como Huffman. Permitiendo reducciones de tamaño de hasta un 45%.

E incluso igualar a compresores estandarizados y probados como Gzip en entornos específicos. Por el sobrecoste de los metadatos de Gzip se logra, que, en archivos pequeños nuestro modelo con solo un nivel de compresión pueda comparar el tamaño conseguido por Gzip.

A diferencia de los métodos de compresión dinámica que deben nuestra arquitectura de diccionario estático compartido permite que el archivo comprimido (.cssc) contenga prácticamente solo carga útil. Siendo este otro punto fuerte a destacar, la optimización del flujo de transmisión.

La facilidad de uso también es otro punto a destacar, el sistema se ha diseñado para que cualquier persona iniciada en el campo de la informática pueda ejecutar fácilmente todo el sistema. Con un documento de apoyo y guía el cual te permite seguir paso a paso el proceso, con comentarios de ayuda si es necesario.

Por último, hablar sobre el elefante en la habitación. Se ha desarrollado un sistema de compresión novedoso, robusto y probado el cual por medios de manipulación de bits a bajo nivel consigue no solo comprimir y descomprimir sin pérdidas archivos de CSS, sino que también es aplicable a más lenguajes que compartan un gran grado de repetición en su código. Consiguiendo compresiones de hasta un 81%.

6.2 Motivo de Victoria General de Gzip

Superar de forma constante a un estándar maduro como Gzip representa un reto tecnológico. Mientras que el sistema desarrollado en este trabajo se basa en la codificación Huffman aplicada a una tokenización semántica, Gzip utiliza el algoritmo Deflate, el cual combina la codificación estadística con la técnica de compresión LZ77.

Esta diferencia de enfoque explica por qué Gzip domina en archivos de gran tamaño, donde la probabilidad de encontrar patrones repetitivos largos es muy alta.

No obstante, esta potencia tiene un coste, la descompresión mediante LZ77 es más exigente, ya que requiere que el sistema mantenga y consulte constantemente un búfer con el historial de datos.

En contraste, la descompresión de nuestro algoritmo es significativamente más ligera y rápida, limitándose a una navegación a través de un árbol binario

Aunque Gzip es imbatible en la detección de redundancia bruta en grandes volúmenes, sufre un "lastre" de metadatos en cada archivo, ya que debe transportar la información necesaria para reconstruir sus propios diccionarios. Es precisamente en este punto donde la arquitectura de este proyecto logra su ventaja.

Aun asi es innegable que Gzip supera constantemente a nuestro modelo en potencia de compresión. Tiene sentido ya que combina dos algoritmos de compresión muy eficientes, pero hemos conseguido resultados que en un principio no pensábamos.

6.3 Limitaciones y Áreas de Mejora

A pesar de los resultados positivos obtenidos, se han podido también identificar ciertas limitaciones que presentan futuras oportunidades de mejora.

La principal debilidad es la dependencia del dominio y del análisis previo. Como se observó en las pruebas del corpus específico, si un archivo CSS utiliza una sintaxis que no fue contemplada durante la fase de generación del árbol, el rendimiento cae significativamente ya que se obligado a utilizar de forma masiva el Código de Escape (CE). Esta rigidez supone que el sistema se beneficia enormemente de funcionar en un corpus especifico, pero pierde versatilidad en corpus genéricos o al comprimir archivos diferentes.

Los archivos generados por los scripts de Python no están optimizados. Esto es un punto que se puede observar al abrir cualquiera de los dos. Como sucede con los CSS minificados, en su momento de creación se hicieron de manera que visualmente se entiendan, con comentarios y una colocación con sentido visual. esto produce que haya que “limpiar” los tokens de los archivos generando una pequeña latencia al momento de procesarlos.

Para mitigar estas limitaciones se proponen las siguientes mejoras:

- **Hibridación con otro método de compresión:** De la misma manera que trabaja Gzip, juntar el algoritmo desarrollado en este proyecto con algún otro conocido, haría que mejorase el % de compresión.

- **Implementación de Diccionarios Dinámicos por Contexto:** Una mejora del proyecto vendría con la creación de librerías donde guardar diccionarios especializados según el framework. Por ejemplo, uno específico para Bootstrap, otro para Tailwind. Podríamos detectar automáticamente el tipo de archivo y seleccionar el árbol binario de Huffman más adecuado.

- **Optimizar los Archivos de Tokens y Árbol:** Obviamente un trabajo a realizar sería que dichos archivos se optimizaran para que solo contuvieran información importante. Así reduciríamos el tiempo de lectura de ellos.

6.4 Conclusión

Como balance final, este Trabajo Fin de Grado ha demostrado que la especialización semántica es una vía eficaz para la optimización de recursos web. Si bien compresores estándar son herramientas increíblemente potentes en la compresión de archivos de tamaño medio y grande, el diccionario compartido desarrollada en este proyecto ha probado tener un posible valor estratégico en segmentos de micro archivos especializados. Se elimina asi el lastre de los metadatos generados en cada envío.

A su vez, hemos observado que aun con un solo método de compresión, en entornos óptimos, se ha conseguido un gran % de compresión. Se ha conseguido un sistema robusto, capaz de alcanzar reducciones superiores al 80%, y todo sin pérdida de datos.

Este trabajo sienta las bases para futuras implementaciones donde la compresión no solo dependa de los caracteres, sino de la comprensión lógica del lenguaje que se pretende transmitir, abriendo la puerta a una web más ligera, rápida y eficiente.

7 Análisis de Impacto

En este capítulo se evalúa el impacto potencial de la solución creada en este proyecto, analizando como la optimización del trafico web afecta a diferentes estratos de la sociedad.

7.1 Impacto Personal y Profesional

Desde el comienzo de esta aventura he estado entusiasmado con el proyecto. El año pasado cuando mi tutor menciono la idea genérica se me quedo grabada y le propuse el tema para la realización de este.

Siempre me ha gustado optimizar problemas comunes, darles una segunda vuelta y sacarles todo el jugo posible, y esta no era la diferencia. Intentar una compresión a nivel de token me parecía algo tan obvio y que solo unos pocos estudios hablaban de ello que quería intentarlo yo y descubrir el porqué. Tras meses de trabajo y con el apoyo en momentos críticos de mi tutor he conseguido un sistema, bajo mi punto de vista, funcional e innovador.

Aparte de la satisfacción personal lograda, me ha servido para afianzar y utilizar gran parte de mi conocimiento como ingeniero en ello. Como dice un gran profesor "Hay dos tipos de ingenieros los que dicen como resolver los problemas y los que de verdad los resuelven" y entiendo la dificultad de los segundos.

Gracias a los conocimientos informáticos de bajo nivel he podido crear el manejo del flujo de bits, el descubrimiento de herramientas como el sistema operativo Linux que te permite una gran cantidad de posibilidades, mi conocimiento de varios lenguajes de programación y sacarle las ventajas a cada uno. Mi manejo en comandos y sistemas operativos, los estudios sobre organización del trabajo y experiencia de usuario, conocimiento sobre estructuras de datos y su funcionamiento que permiten elegir la mejor en cada caso.

En definitiva, la realización de este Trabajo de Fin de Grado ha hecho que vuelque todos mis conocimientos y trabaje con ellos de manera real y para un fin específico.

7.2 Impacto Empresarial y Económico

Centrándonos ahora en el impacto que el proyecto puede causar a empresas interesadas, la principal ventaja sería la reducción de costes operativos:

- **Ahorro en Almacenamiento:** al lograr reducciones, superiores al 80% en algunos casos, las empresas pueden disminuir drásticamente sus facturas en trasferencias de información y almacenamiento de estas. Ya que suelen trabajar con un tipo de archivos específicos podrían generar un corpus para analizar y generar un árbol binario de Huffman especializados para ellos.

- **Mejora en el Servicio y Carga de Archivos:** Un CSS más ligero acelera el renderizado inicial de la web. Está demostrado que una mejora en la carga impacta directamente en la retención de usuarios

7.3 Impacto Social y Cultural

El impacto en este campo se centra en la democratización del acceso a la información. Como cualquier compresor al reducir el tamaño de los datos enviados se logra que redes de baja velocidad o dispositivos antiguos, puedan tener una mejor web usable.

7.4 Impacto Ambiental y Desarrollo Sostenible

Aunque el software es un bien intangible, su transmisión y almacenamiento requieren de infraestructuras físicas que consumen grandes cantidades de energía. Al optimizar el tamaño de los archivos CSS se disminuye el volumen de datos que deben viajar por la red, lo que se traduce en una reducción directa del consumo eléctrico de los servidores de origen y de los nodos de transmisión. [23]

De igual manera la ligereza del descompresor desarrollado basado en la navegación binaria simple por el árbol de Huffman, hace que se requiera menos ciclos de CPU que algoritmos más complejos como el LZ77. Puede ser que desde un inicio no sea posible ver la diferencia de usarlo una vez, pero en servidores que reciben y lanzan miles de peticiones por segundo es una mejora sustancial.

Bajo estas premisas, la solución propuesta se alinea con los siguientes Objetivos de Desarrollo Sostenible (ODS) de la Agenda 2030:

- **ODS 9 (Industria, Innovación e Infraestructura):** El desarrollo de un algoritmo que optimiza el intercambio de datos en la web contribuye a crear infraestructuras digitales más eficientes. E intentar innovar para mejorar mecanismos industriales es necesaria para la evolución de la web.

- **ODS 13 (Acción por el Clima):** La reducción de la transferencia de datos y del esfuerzo de procesamiento computacional tiene un impacto positivo en la reducción de las emisiones de CO2 generadas por los centros de datos.

8 Bibliografía

[1] T. Finn y A. Downie. (s.f.). Experiencia de usuarios [En línea]. Disponible: https://www.ibm.com/es-es/think/topics/user-experience.

[2] J. S. Vitter y P. Krishnan, "Optimal preprocessing for on-line data compression," Elect. Eng. Res. Lab., 1989.

[3] Cloudflare. (s.f.). Cómo minificar CSS [En línea]. Disponible: https://www.cloudflare.com/es-es/learning/performance/how-to-minify-css/.

[4] M. Umar. (2021,sep 03). Algoritmos de compresión con y sin pérdida [En línea]. Disponible: https://blog.fileformat.com/es/compression/lossy-and-lossless-compression-algorithms/.

[5] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proc. IRE*, vol. 40, no. 9, pp. 1098-1101, sep. 1952.

[6] IONOS. (2015, nov 13). Comprimir CSS para optimizar los tiempos de carga [En línea]. Disponible: https://www.ionos.es/digitalguide/paginas-web/diseno-web/comprimir-css-para-optimizar-los-tiempos-de-carga/.

[7] Cadabullos. (2022,may 18). Cómo comprimir y minimizar archivos CSS [En línea]. Disponible: https://www.cadabullos.com/blog/comprimir-minimizar-archivos-css.

[8] I. Novikov. (2025, abr 05). ¿Qué es la minificación y por qué es necesaria? [En línea]. Disponible: https://lab.wallarm.com/what/que-es-la-minificacion-y-por-que-es-necesaria/?lang=es

[9] Surática. (s.f.). Qué es la minificación [En línea]. Disponible: https://www.suratica.es/que-es-la-minificacion/.

[10] C. Buckler. (2023,ago 23). Optimizar CSS y cómo afecta al rendimiento [En línea]. Disponible: https://kinsta.com/es/blog/optimizar-css/.

[11] Python Software Foundation. (s.f.). Librería re — Expresiones regulares [En línea]. Disponible: https://docs.python.org/es/3/library/re.html.

[12] Python Software Foundation. (s.f.). Librería collections — Tipos de datos de contenedores [En línea]. Disponible: https://docs.python.org/es/3/library/collections.html.

[13] Python Software Foundation. (s.f.). Librería heapq — Algoritmo de cola de prioridad [En línea]. Disponible: https://docs.python.org/es/3/library/heapq.html.

[14] Python Software Foundation. (s.f.). Librería os — Interfaces misceláneas del sistema operativo [En línea]. Disponible: https://docs.python.org/es/3/library/os.html.

[15] Oracle. (s.f.). Java Platform Standard Ed. 8 — Class OutputStream [En línea]. Disponible: https://docs.oracle.com/javase/8/docs/api/java/io/OutputStream.html.

[16] Línea de Código. (s.f.). Etiqueta java.util [En línea]. Disponible: https://lineadecodigo.com/tag/java-util/.

[17] Certidevs. (2025, mar 12). API java.nio 2 [En línea]. Disponible: https://certidevs.com/tutorial-java-nio.

[18] A. Awan. (2024,nov 22). What is tokenization [En línea]. Disponible: https://www.datacamp.com/es/blog/what-is-tokenization.

[19] Oracle. (s.f.). Java Platform Standard Ed. 8 — Class HashMap [En línea]. Disponible: https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html.

[20] Gzip. (s.f.). The gzip home page [En línea]. Disponible: https://www.gzip.org/.

[21] IONOS. (2025, jun 05). Qué es Gzip y cómo se utiliza esta herramienta [En línea]. Disponible: https://www.ionos.es/digitalguide/servidores/know-how/que-es-gzip-y-cuales-son-sus-ventajas/.

[22] La Salle URL. (2021,may 04). Consumo de energía de un centro de datos [En línea]. Disponible: https://blogs.salleurl.edu/es/consumo-de-energia-de-un-centro-de-datos.

9 Anexos

Los siguientes anexos proporcionan información técnica complementaria y recursos prácticos para la comprensión y ejecución del sistema de compresión desarrollado

Se detallan los orígenes de los datos utilizados, el procedimiento para la puesta en marcha del software y la procedencia del código de Huffman genérico usado

9.1 Fuentes del Corpus

Como mencionamos en el Trabajo la batería de archivos corpus viene de una mezcla de código abierto, ejemplos web, buscadores genéricos, paginas comunes y archivos prestados por conocidos:

- **Códigos genéricos:**
  - Luis Forgiarini creo una página web para compartir codigo de inicio para la construcción de archivos CSS.
    - https://luisforgiarini.com/codigos-css-para-paginas-web/
  - Daniel fuentes público en su Github unos varios archivos CSS para un proyecto propio
    - https://luisforgiarini.com/codigos-css-para-paginas-web/
  - Framework Bootstrap en Github. Un repositorio de código abierto para archivos
    - https://github.com/twbs
  - Inteligencia artificial generativa de la empresa Google, Géminis. Le pedí que me escribiera varios archivos genéricos CSS únicamente para las pruebas. Esto debido al auge del código auto generativo
    - https://gemini.google.com/

- **Códigos específicos:**
  - YouTube. Para tener un corpus de CSS de páginas de vídeo.
    - https://www.youtube.com/
  - Buscador Brave. Por la inclusión de código especifico de buscadores web
    - https://brave.com/es/
  - Archivos dados por Vinicius Taliari. Un Amigo personal que trabaja en la creación de páginas web. Una gran parte del corpus forma parte del ya que trabajo en diversos proyectos propios.
    - www.linkedin.com/in/vinicius-taliari-a76a5b20

9.2 Código Genérico Huffman

Tras realizar una búsqueda de un código genérico del algoritmo de Huffman, hallé el repositorio del usuario YAIKHOM. Gracias a su publicación puede probar mi corpus con su código y hacer la comparación de compresores.

Aquí muestro el enlace al repositorio: https://github.com/gyaikhom/huffman

9.3 Manual de usuario

El sistema ha sido diseñado para ser ejecutado en entornos Unix mediante el uso de un Makefile que automatiza la cadena de herramientas.

Los pasos son los siguientes:

1. **Ejecución del mandato make**: creara la capeta bin junto con la compilación de las clases Java. Después ver los siguientes mandatos con make help

2. **Entrenamiento y Generación del Árbol**: Ejecutar mandato de make para el script de Python, analizadorCSS.py. este analizará el corpus que selecciones y generará la lista de tokens (para cambiar la longitud de la lista cambiar la constante de dicha clase). Posteriormente ejecutar el siguiente mandato de make.

3. **Compresión:** Ejecutar el compresor con el mandato make pertinente y seleccionar el archivo a comprimir. Se creara un archivo .cssc en la carpeta.

4. **Descompresión:** El último paso sería la restauración del archivo con el mandato make siguiente. Debes seleccionar el archivo .cssc y ponerle nombre al nuevo archivo.

5. **Comprobación:** Este paso es opcional. Puede con el mandato Diff comparar ambos archivos, el original y el restaurado para comprobar que son iguales.

10 Agradecimientos

En primer lugar, de manera muy especial, quiero agradecer a mi familia. Su apoyo infinito junto con la confianza puesta en mí, ha sido el motor que me ha permitido llegar hasta el final de mi experiencia académica. Gracias por todo el amor y comprensión brindados.

Agradecer a mi tutor, Jesús, un profesor de los que pocos quedan. Su entusiasmo y ganas de hacernos aprender demuestran la pasión que siente por la enseñanza. Gracias por sus consejos y el apoyo tanto en este trabajo como en el aula.

A mis compañeros y amigos, por aguantar mis charlas sobre este duro camino y por sacarme una sonrisa en los momentos más difíciles. Un agradecimiento especial a mi amigo Vinicius por su valioso aporte a este proyecto.

Finalmente, agradecer a la universidad por brindarme los recursos y el conocimiento para completar mi formación como ingeniero. Este trabajo es el resultado de todo lo aprendido y, lejos de ser un final, espero que sea el comienzo de nuevas y apasionantes etapas en mi vida.

Este documento esta firmado por

	**Firmante**	CN=tfgm.fi.upm.es, OU=CCFI, O=ETS Ingenieros Informaticos - UPM, C=ES
	Fecha/Hora	Thu Jan 15 13:32:36 CET 2026
	Emisor del Certificado	EMAILADDRESS=camanager@etsiinf.upm.es, CN=CA ETS Ingenieros Informaticos, O=ETS Ingenieros Informaticos - UPM, C=ES
	Numero de Serie	561
	Metodo	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature)