UNIVERSIDAD NACIONAL DE INGENIERÍA
FACULTAD DE INGENIERÍA INDUSTRIAL Y DE SISTEMAS

MATEMÁTICA DISCRETA
FB301 – U

"CODIFICACIÓN DE IMÁGENES CON EL ALGORITMO DE HUFFMAN"

DOCENTE: **Benites Yarleque José Valerio**

INTEGRANTES:

- **Cáceres San Miguel Giancarlo Pedro**
- **Hernández Hernández Jahir Alejandro**
- **Rojas Torres Brayan Jesús Alexander**
- **Rosales Ávila Edison Edwin**
- **Saavedra Balarezo Gabriel Alessandro**

Fecha de entrega: 03 de Julio de 2023

2023-I

ÍNDICE

INTRODUCCIÓN

En la era digital actual, el procesamiento y la transmisión de imágenes se han convertido en elementos esenciales de la comunicación visual. La necesidad de comprimir y codificar imágenes de manera eficiente ha llevado al desarrollo de diversas técnicas y algoritmos. Uno de los métodos más ampliamente utilizados es la codificación de imágenes con Huffman.

La codificación de Huffman es un algoritmo de compresión sin pérdida que permite reducir el tamaño de los archivos de imagen sin comprometer significativamente su calidad visual. Este enfoque se basa en la teoría de la información y se ha utilizado ampliamente en una variedad de aplicaciones, desde la transmisión de imágenes en redes de comunicación hasta el almacenamiento de archivos en dispositivos de almacenamiento limitados.

El objetivo principal de este trabajo es explorar y analizar en detalle el proceso de codificación de imágenes con Huffman. Se abordarán los principios fundamentales de este algoritmo, así como su implementación práctica y sus aplicaciones en el campo de la compresión de imágenes.

En primer lugar, se proporcionará una base teórica sólida sobre los conceptos clave de la codificación de Huffman y la teoría de la información. Se explicarán los fundamentos de la codificación de fuentes y la construcción de árboles de Huffman, destacando su utilidad para representar símbolos de manera eficiente.

A continuación, se presentarán los pasos prácticos para aplicar el algoritmo de Huffman a imágenes digitales. Se describirá el proceso de codificación, que implica la creación de una tabla de códigos de Huffman basada en las frecuencias de aparición de los símbolos en la imagen. Además, se explorarán las estrategias para la decodificación eficiente de los datos codificados.

Posteriormente, se analizarán los resultados obtenidos mediante la codificación de imágenes con Huffman, evaluando la tasa de compresión lograda y el impacto en la calidad visual de las imágenes comprimidas. Se compararán los resultados con otros métodos de compresión existentes, destacando las ventajas y desventajas de la codificación de Huffman en términos de eficiencia y calidad.

Finalmente, se discutirán las aplicaciones actuales de la codificación de imágenes con Huffman y se mencionarán posibles mejoras y desarrollos futuros en esta área. Se analizarán las tendencias emergentes en la compresión de imágenes y cómo la codificación de Huffman se integra en sistemas más complejos de compresión y transmisión de datos.

CAPÍTULO II: FUNDAMENTOS TEORICOS

1.1 COMPRESION DE DATOS

La compresión de datos es el proceso de reducir el tamaño de los archivos sin perder información esencial, además del manejo y transmisión eficiente de información digital. Consiste en reducir el tamaño de los archivos sin perder información esencial. En el caso específico de las imágenes, la compresión de datos desempeña un papel crucial debido a la gran cantidad de información visual que contienen.

La compresión de imágenes busca reducir la redundancia y aprovechar las características intrínsecas de las imágenes para lograr una representación más compacta sin sacrificar la calidad visual. Al comprimir imágenes, se busca ahorrar espacio de almacenamiento y mejorar la eficiencia en la transmisión a través de redes de comunicación, sin comprometer de manera significativa la fidelidad de la imagen original.

Existen dos tipos principales de compresión de datos: la compresión con pérdida y la compresión sin pérdida. En la compresión con pérdida, se elimina información redundante y no perceptible para el ojo humano, lo que conlleva una reducción significativa del tamaño del archivo, pero también una pérdida irreversible de detalles y calidad visual. Por otro lado, la compresión sin pérdida permite recuperar la información original sin ninguna pérdida de calidad, aunque la tasa de compresión suele ser menor.

En el contexto de la codificación de imágenes con Huffman, se emplea la compresión sin pérdida. El algoritmo de Huffman logra la compresión sin pérdida al asignar códigos de longitud variable a los símbolos de una imagen de manera que los símbolos más frecuentes se representen con códigos más cortos y los menos frecuentes con códigos más largos. Esto permite reducir el tamaño del archivo sin comprometer la integridad de la imagen.

La compresión de imágenes con Huffman se basa en la idea de que en una imagen, ciertos valores de píxeles o combinaciones de ellos son más frecuentes que otros. Por lo tanto, al asignar códigos más cortos a estos valores de píxeles comunes, se logra una representación más eficiente y compacta de la imagen.

1.2 INTRODUCCION AL ALGORITMO DE HUFFMAN

El algoritmo de Huffman es un método popular y eficiente para la compresión sin pérdida de datos, incluidas las imágenes. Fue propuesto por David A. Huffman en 1952 y se basa en la teoría de la información.

La compresión de datos es un aspecto esencial en el procesamiento de imágenes y en la transmisión eficiente de información visual. En este contexto, el algoritmo de Huffman se ha

convertido en una técnica ampliamente utilizada para la compresión sin pérdida de datos, incluidas las imágenes. Este algoritmo, propuesto por David A. Huffman en 1952, se basa en la teoría de la información y ofrece una manera eficiente de representar datos de manera más compacta, ahorrando espacio de almacenamiento y facilitando su transmisión.

El algoritmo de Huffman se basa en el principio de asignar códigos de longitud variable a diferentes símbolos, de manera que los símbolos más frecuentes sean representados por códigos más cortos y los menos frecuentes por códigos más largos. Esto permite una codificación más eficiente y compacta, ya que se aprovecha la distribución de probabilidades de los símbolos en los datos.

La idea fundamental detrás del algoritmo de Huffman es construir un árbol binario en el cual los símbolos se encuentren en las hojas y los códigos se obtengan mediante el recorrido desde la raíz hasta cada hoja. El proceso de construcción del árbol de Huffman implica analizar la frecuencia de aparición de cada símbolo y utilizar esta información para construir un árbol de manera óptima.

En esencia, el algoritmo de Huffman busca encontrar la asignación óptima de códigos que minimice la longitud media de los códigos utilizados para representar los símbolos en los datos. Para lograr esto, se utiliza una estrategia de combinación sucesiva, donde los símbolos menos frecuentes se fusionan en nodos internos y los símbolos más frecuentes se encuentran en las hojas del árbol resultante.

Una vez construido el árbol de Huffman, se asigna un código único a cada símbolo recorriendo el árbol desde la raíz hasta cada hoja. Al asignar los códigos, se utiliza la regla de que el movimiento hacia la izquierda representa el bit "0" y el movimiento hacia la derecha representa el bit "1". Esta asignación de códigos de longitud variable garantiza que los símbolos más comunes se representen con menos bits, lo que resulta en una representación más eficiente y compacta de los datos.

1.3 PRINCIPIOS DE CODIFICACION DE HUFFMAN

El proceso de codificación de Huffman implica varios pasos fundamentales. En primer lugar, se analiza la frecuencia de aparición de cada símbolo en los datos que se desean codificar, en este caso, los valores de los píxeles de una imagen. Luego, se construye un árbol de Huffman utilizando una estrategia de combinación sucesiva, donde los símbolos menos frecuentes se fusionan en nodos internos y los símbolos más frecuentes se encuentran en las hojas del árbol.

Una vez construido el árbol de Huffman, se asigna un código único a cada símbolo recorriendo el árbol desde la raíz hasta cada hoja. Los códigos se asignan de manera que los símbolos más frecuentes tengan códigos más cortos y los menos frecuentes tengan códigos más largos. Este enfoque garantiza una codificación eficiente en términos de compresión, ya que los símbolos más comunes se representan con menos bits. A continuación, exploraremos los principios fundamentales de la codificación de Huffman:

1. Análisis de frecuencias: El primer paso en la codificación de Huffman es realizar un análisis de las frecuencias de aparición de los símbolos en la imagen. Esto implica contar cuántas veces ocurre cada símbolo y determinar su probabilidad de aparición. En el caso de las imágenes, los símbolos pueden ser los niveles de intensidad de los píxeles, por ejemplo.

2. Construcción del árbol de Huffman: Una vez que se tiene el análisis de frecuencias, se procede a construir el árbol de Huffman. Este árbol se construye utilizando una estrategia de combinación sucesiva. Inicialmente, cada símbolo se considera un nodo hoja en el árbol. Luego, los nodos con menor frecuencia se combinan en nodos internos, hasta obtener un único nodo raíz que engloba a todos los símbolos.

3. Asignación de códigos: Una vez construido el árbol de Huffman, se asigna un código único a cada símbolo recorriendo el árbol desde la raíz hasta cada hoja. La asignación de códigos se realiza siguiendo la regla de que el movimiento hacia la izquierda representa el bit "0" y el movimiento hacia la derecha representa el bit "1". Al finalizar este proceso, cada símbolo tiene un código de longitud variable que representa su posición en el árbol de Huffman.

4. Propiedad de prefijo: Uno de los principios fundamentales de la codificación de Huffman es que los códigos asignados a los símbolos no deben ser prefijos de otros códigos. Esto asegura que no haya ambigüedad en la decodificación de los datos comprimidos. Es decir, ningún código puede ser el prefijo de otro código asignado a un símbolo distinto.

5. Decodificación: La decodificación de los datos comprimidos con Huffman implica recorrer el árbol de Huffman utilizando los códigos como guía. Al seguir los bits del código, se llega a una hoja del árbol que corresponde a un símbolo específico. De esta manera, se recupera la información original de la imagen comprimida.

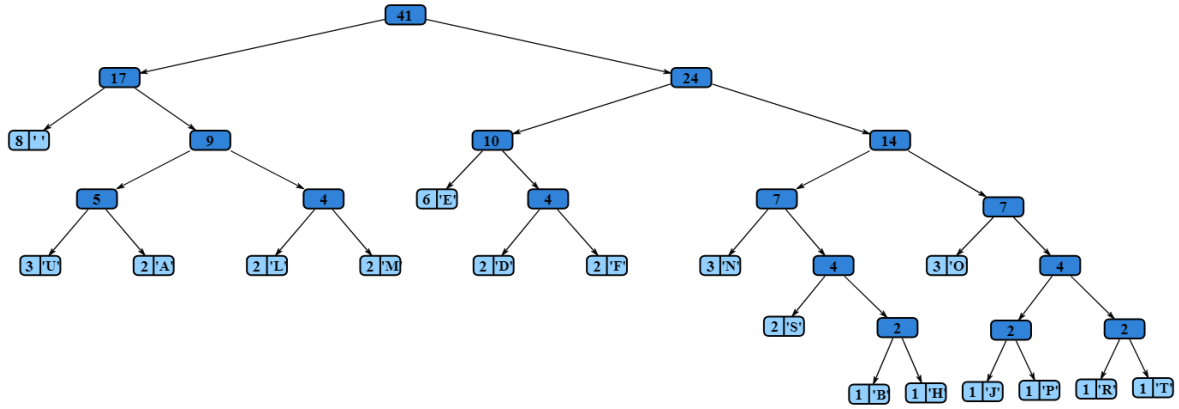
La codificación de Huffman permite una representación más eficiente y compacta de las imágenes al asignar códigos más cortos a los símbolos más frecuentes y códigos más largos a los menos frecuentes. Esto se logra aprovechando la distribución de probabilidades de los símbolos en la imagen. En la siguiente sección, se presentará un ejemplo ilustrativo de codificación de Huffman para una mejor comprensión del proceso.

1.4 EJEMPLO ILUSTRATIVO DE CODIFICACION DE HUFFMAN

Para ilustrar el proceso de codificación de Huffman, consideremos un ejemplo simplificado. Supongamos que tenemos una imagen en blanco y negro de 8 píxeles, donde cada píxel puede tener uno de los siguientes valores: 0, 1, 2 o 3. Calculamos la frecuencia de aparición de cada símbolo y construimos el árbol de Huffman en base a esas frecuencias.

Luego, asignamos códigos a cada símbolo en función de su posición en el árbol de Huffman. Por ejemplo, supongamos que el símbolo "0" se encuentra en una hoja en el nivel más bajo del árbol y se le asigna el código "00", el símbolo "1" se encuentra en otra hoja y se le asigna el código "01", y así sucesivamente.

Finalmente, la imagen se codifica reemplazando cada valor de píxel por su correspondiente código Huffman. Por ejemplo, si la imagen original es [1, 3, a actualidad, la tecnología avanza a pasos agigantados, lo que conlleva a que las redes actuales no satisfacen en velocidad a los nuevos servicios multimedia que están apareciendo.“En la Urbanización los Olivos existe una infinidad de postes con una gran cantidad de cables que cuelgan en las instalaciones de conexión de internet y se convierten en un peligro para los ciudadanos, además de atentar contra el paisaje y ornato urbano y el entornovisual de los pobladores. Este hecho nos motiva a analizar esta problemática a fin de tomar medidas, o iniciativas, que nos permita una solución a este tema ; contar con un buena Optimización de en tendido de fibra óptica en la urbanización los Olivos es una tecnología de acceso para servicios de telecomunicaciones mediante la implementación de cables de fibra óptica, con elementos que no requieren de energía para funcionar, es decir elementos pasivos. El beneficio de la implementación de tales componentes es la reducción del coste deequipos que tienen que ser instalados hasta el usuario final de un servicio.



CAPÍTULO III: IMPLEMENTACIÓN DEL ALGORITMO DE HUFFMAN PARA IMÁGENES

3.1 Procesamiento de imágenes en Python

Para la lectura y generación de imágenes se usa la librería OpenCV, la manera en que se interpreta una imagen es por los niveles de color RGB por cada píxel, de manera que se interpreta a la imagen como una matriz de dimensiones iguales a la cantidad de píxeles de la imagen.

```
image = cv2.imread("gato.png")

if image is None:
    print("No se pudo cargar la imagen")
    exit()
for i in range(rows):
    for j in range(cols):
        # Obtener el valor del píxel en la posición (i, j)
        p = image[i, j]
        # Acceder a los canales de color del píxel (B, G, R)
        blue = p[0]
        green = p[1]
        red = p[2]

        px = pixel(1, red, green, blue)
        agregar(lista, px)
```

Como se observa en estas partes del código se lee la imagen con el método imread de la librería cv2 y se accede a cada píxel con `image[i, j]`, luego, se crea un objeto pixel de una clase definida al inicio del código con la frecuencia 1 y sus niveles de color RGB correspondientes.

Clase pixel:

```
class pixel:
    def __init__(self, frec, R, G, B):
        self.R = R
        self.G = G
        self.B = B
        self.frecuencia = frec
        self.izquierda = None
        self.derecha = None

    #private:
    def __inorden_recursivo(self, pixel):
        if pixel is not None:
            self.__inorden_recursivo(pixel.izquierda)
            print(f"[{pixel.R} {pixel.G} {pixel.B}]", end=" ")
            self.__inorden_recursivo(pixel.derecha)

    #public:
    def iguales(self, pixel):
        if self.R == pixel.R and self.G == pixel.G and self.B == pixel.B:
```

```
            return True
        return False

    def inorden(self):
        print("Imprimiendo árbol inorden: ")
        self.__inorden_recursivo(self)
        print("")
```

Instancias de esta clase son usadas después como nodos del árbol binario de Huffman, por eso se tienen los atributos self.derecha y self.izquierda.

3.2 Extracción de frecuencias de píxeles

Para almacenar las frecuencias de los píxeles de la imagen, se crea una lista vacía a la que se le agregará el píxel nuevo de frecuencia 1 si un píxel con sus niveles RGB no existen, en caso contrario, se le aumentará la frecuencia al píxel existente en 1.

```
lista = []
def agregar(A, px):
    for i in range(len(A)):
        if A[i].R==px.R and A[i].G==px.G and A[i].B==px.B:
            A[i].frecuencia += 1
            return
    A.append(px)
rows, cols, channels = image.shape
for i in range(rows):
    for j in range(cols):
        # Obtener el valor del píxel en la posición (i, j)
        p = image[i, j]
        # Acceder a los canales de color del píxel (B, G, R)
        blue = p[0]
        green = p[1]
        red = p[2]

        px = pixel(1, red, green, blue)
        agregar(lista, px)
```

En la función `agregar`, se intentó agregar los elementos a la lista con un algoritmo de inserción de acuerdo a sus niveles RGB para evitar la comparación del elemento a buscar con todos los de la lista, comparando la expresión R*10^6 + G*10^3 + B (RGB tienen a lo mucho 3 cifras), pero se observó que el tiempo de ejecución era mayor para esa parte de la función, es posible que se deba al coste de rendimiento de multiplicar a los niveles por números grandes y compararlos, pero es posible que para una imagen con mucha variedad de píxeles, lo que implica una cantidad enorme de elementos en la lista, el rendimiento sea mejor con el método de inserción.

3.3 Construcción del árbol de Huffman

Para la creación del árbol binario usando el algoritmo de Huffman, se ordenó primeramente la lista de píxeles de acuerdo a su frecuencia en orden creciente, luego para cada iteración se crea una instancia de pixel con niveles RGB -1 (que se usa como nodo), asignándole a sus atributos derecha e izquierda las referencias de los pixeles 1 y 2 de la lista ordenada y a su

atributo frecuencia la suma de las frecuencias de esos píxeles, luego, se elimina el primer elemento de la lista (pixel que ya está referenciado en la hoja izquierda del nodo creado) y se inserta el nodo actual en el lugar que debe de ir en la lista de acuerdo a su frecuencia, de tal manera que el tamaño de la lista se reduce en 1 en cada iteración. Finalmente, la raíz del árbol será el objeto pixel que quede en la lista (entiéndase que los atributos de los nodos tienen valores RGB -1 para evitar confusiones del programa).

```
while(len(lista)>1):
    nodo = pixel(lista[0].frecuencia+lista[1].frecuencia, -1,-1,-1)
    nodo.izquierda = lista[0]
    nodo.derecha = lista[1]
    lista.pop(0)
    #insercion del nuevo nodo
    for i in range(len(lista)):
        if i==len(lista)-1:
            lista[i] = nodo
            break
        if lista[i+1].frecuencia < nodo.frecuencia:
            lista[i] = lista[i+1]
        else:
            lista[i] = nodo
            break

raiz = lista[0]
```

3.4 Generación de los códigos binarios

Una vez creado el árbol binario que contiene a los píxeles diferentes de las imágenes como hojas, es necesario definir una función que busque un píxel dado en el árbol binario y retorne su codificación binaria en el árbol, de manera que al acceder a una rama izquierda del nodo agregue a la cadena de codificación un 0 y un 1 para la rama derecha. Esto se implementó con la siguiente función recursiva:

```
def buscar_codigo(raiz, el):
    if raiz is None:
        return None

    if raiz.R==el.R and raiz.G==el.G and raiz.B==el.B:
        return ""
    codigo_izquierda = buscar_codigo(raiz.izquierda, el)
    if codigo_izquierda is not None:
        return "0" + codigo_izquierda

    codigo_derecha = buscar_codigo(raiz.derecha, el)
    if codigo_derecha is not None:
        return "1" + codigo_derecha
    return None
```

En la función creada, se realiza una búsqueda en profundidad y retorna un valor nulo cuando el nodo no coincide con el píxel que se quiere buscar, ya que las hojas que contienen los píxeles apuntan a valores nulos en sus hojas, pero en caso de encontrarse, se retorna la cadena

de codificación de bits.

3.5 Guardado y decodificación de la imagen

Para el guardado de la imagen en un archivo de texto codificado, hay que tener en cuenta que el programa decodificador debe tener información sobre el árbol binario creado, si se quiere implementar un programa único para la decodificación de cualquier imagen, se debe de guardar algo para reconstruir el árbol y las dimensiones de la imagen, para esto se crea un archivo "imagencodificada.txt" y se escriben en primer lugar las dimensiones de la imagen separado por comas, luego, separado por un guion, los elementos de la lista de pixeles ordenada de acuerdo a su frecuencia, escribiendo en el orden: "f,R,G,B;". De esta manera, se puede reconstruir el árbol en el programa decodificador usando el mismo algoritmo de construcción del árbol que el programa codificador, y finalmente se recorre la imagen otra vez para buscar por cada píxel su código respectivo en el árbol binario, para escribirlo en el orden de aparición en el archivo.

```
print("Codificando imagen...")
f = open("imagencodificada.txt", "w")
f.write(f"{rows},{cols}-")
for i in range(len(listacop)):
    if i==len(listacop)-1:
        f.write(f"{listacop[i].frecuencia},{listacop[i].R},{listacop[i].G},{li
stacop[i].B}")
        break
    f.write(f"{listacop[i].frecuencia},{listacop[i].R},{listacop[i].G},{listac
op[i].B};")
f.write("-")
for i in range(rows):
    print(f"{int((i/rows)*100)}%")
    for j in range(cols):
        p = image[i, j]
        blue = p[0]
        green = p[1]
        red = p[2]
        px = pixel(1, red, green, blue)
        f.write(buscar_codigo(raiz,px))
f.close()
print("Imagen codificada...")
```

Por ejemplo, al codificar esta imagen (en su tamaño original):

El contenido del archivo de texto generado es:
1080,1920-
3939,251,132,111;8053,252,132,111;84488,16,68,79;247208,252,133,111;1729912,16,69,79
-00111100111100111100111100111100111100111100111100…
En primer lugar, están las dimensiones de la imagen, luego, cada intensidad de RGB para cada píxel diferente con sus respectivas frecuencias, hay 3939 píxeles con R:251, G:132, B:111, 8053 con 252, 132 y 111, y así sucesivamente, y en tercer lugar se encuentra los códigos de cada píxel de manera consecutiva según el orden de aparición en la imagen.

Finalmente, para decodificar la imagen, se usa la misma librería con la cual se puede crear una imagen y colorear cada píxel con un nivel de RGB dado, este nivel se busca en el árbol binario generado en el mismo programa de decodificación con el mismo algoritmo de generación del árbol a partir de la lista ordenada de frecuencias de píxeles y se colorea mientras se lee el contenido de los bits en el archivo, no es necesario separar los bits de acuerdo a cada píxel ya que no existen códigos contenidos en otros y siempre se encontrará el pixel en la hoja árbol binario para regresar al nodo inicial y continuar leyendo los bits.

```
f = open("imagencodificada.txt","r")
L=f.read().split("-")
f.close()
L[0]=L[0].split(",")
rows = int(L[0][0])
cols = int(L[0][1])

lista = []
for k in L[1].split(";"):
    q = k.split(",")
    lista.append(pixel(int(q[0]),int(q[1]),int(q[2]),int(q[3])))

while(len(lista)>1):
    nodo = pixel(lista[0].frecuencia+lista[1].frecuencia, -1,-1,-1)
    nodo.izquierda = lista[0]
    nodo.derecha = lista[1]
    lista.pop(0)
    #insercion del nuevo nodo
    for i in range(len(lista)):
        if i==len(lista)-1:
            lista[i] = nodo
            break
        if lista[i+1].frecuencia < nodo.frecuencia:
            lista[i] = lista[i+1]
        else:
            lista[i] = nodo
            break

raiz = lista[0]

channels = 3  # 3 canales de color para RGB
image = np.zeros((rows, cols, channels), dtype=np.uint8)
```

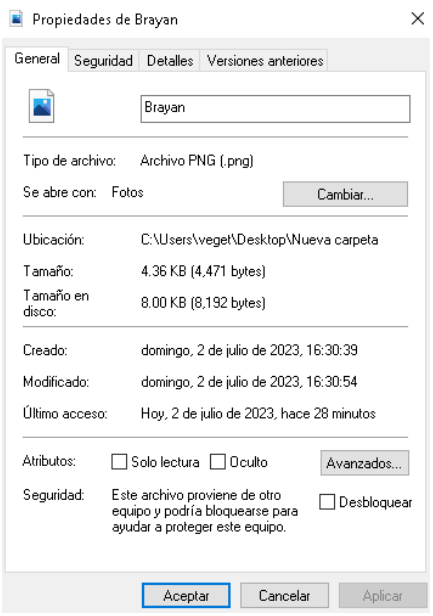
```
n = 0
for i in range(rows):
    for j in range(cols):
        p = raiz
        while p.R==-1:
            if L[2][n] == "0":
                p = p.izquierda
            elif L[2][n] == "1":
                p = p.derecha
            n+=1
        blue = p.B
        green = p.G
        red = p.R
        # Asignar los valores de los canales de color al píxel
        image[i, j] = (blue, green, red)

# Mostrar la imagen
cv2.imwrite("imagen_generada.png", image)
print("Imagen decodificada...")
```

Cabe recalcar que se importa también la librería numpy y se define la clase pixel como en el programa de codificación.

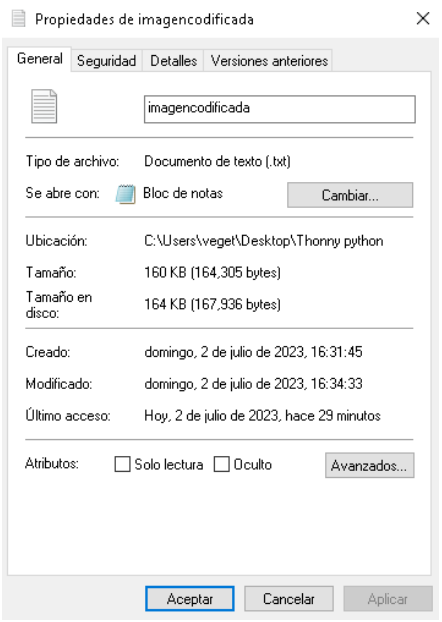
CAPÍTULO IV: ANÁLISIS Y EVALUACIÓN DE RESULTADOS

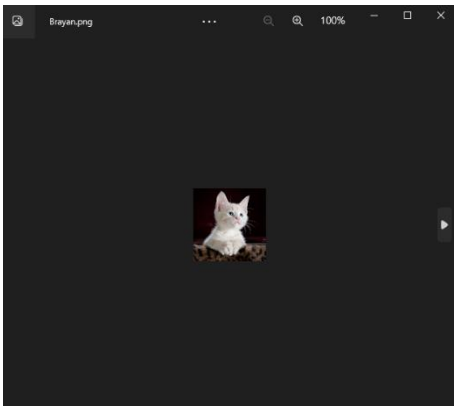
4.1. COMPARACIÓN DE TAMAÑOS DE ARCHIVOS COMPRIMIDOS

Para la comparación en el tamaño de la imagen no codificada con la codificada se usó una imagen de 100x100 pixeles para compararlos.



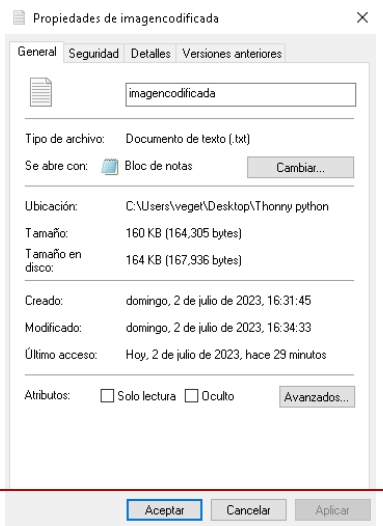
Codificar



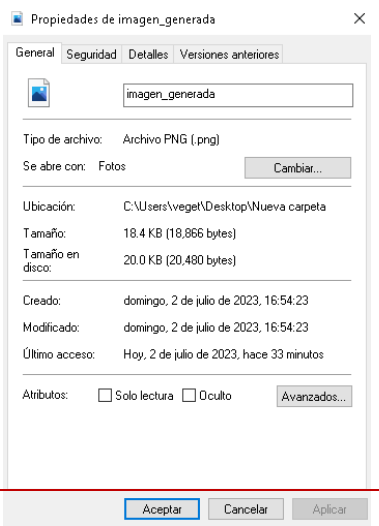


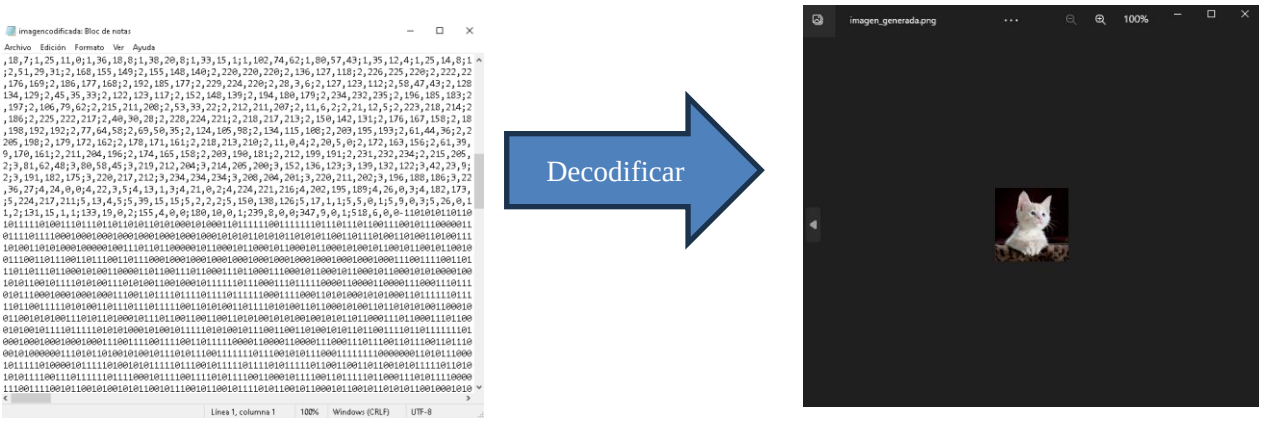
Codificar

Se puede observar que en lo que respecta al tamaño de la imagen, esta se ve incrementada en un 3592.66% con respecto a su tamaño original. Sin embargo, si decodificamos el archivo imagen codificada y lo comparamos con la imagen original se obtiene:

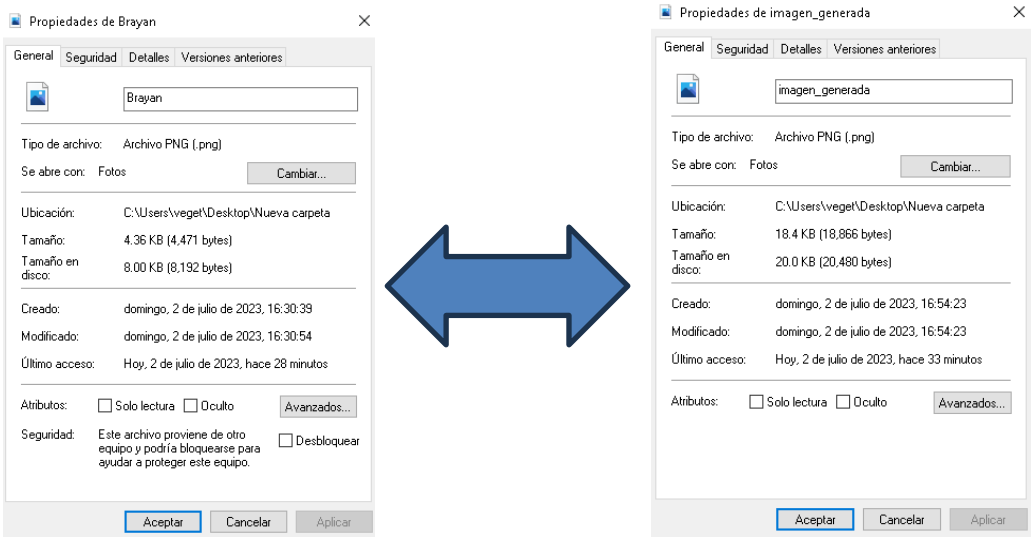


Decodificar





Al momento de decodificar, el tamaño de la imagen se redujo en un 88.5% con respecto al archivo codificado.



Sin embargo, al momento de comparar la imagen decodificada con respecto a la imagen original, esta sigue siendo menor en lo que respecta al tamaño. El tamaño de la decodificada es 280% mayor con respecto a la imagen original.

La conclusión en esta comparación es que nuestro programa no es eficiente en lo que respecta a la compresión, no obstante, se puede aprovechar en la codificación y decodificación de imágenes.

4.2. MEDICIÓN DE LA CALIDAD DE LA IMAGEN COMPRIMIDA

Saber si la imagen ha perdido calidad al momento de comprimirse se verificará la similitud entre la imagen original y la imagen decodificada. Para ello se usará un programa en internet que mide la correlación cruzada normalizada de dos imágenes en línea

Al compararlos, nos sale una similitud del 100%. Esto significa que no se ha perdido información de la imagen al momento de la codificación y la decodificación. Sin embargo, la imagen decodificada es de mayor tamaño lo que quiere decir que contiene ruido o interferencia, pero manteniendo la información de la imagen original al completo.

Imagen original

Imagen decodificada

4.3. EVALUACION DEL TIEMPO DE EJECUCION

En lo que respecta a eficiencia y tiempo de ejecución. Se usará un IDE para observar el tiempo que demora al programa en llevar a cabo la tarea de codificación y decodificación.

Name	Call Count	Time (ms)	Own Time (ms)
iguales	63647888	91390 46.5 %	91390 46.5 %
buscar_codigo	127235412	179192 91.1 %	87802 44.6 %
codigo.py	1	196709 100.0 %	6226 3.2 %
agregar	10000	5719 2.9 %	5709 2.9 %
ordlistap	1	4225 2.1 %	4222 2.1 %
<built-in method builtins.len>	6300004	785 0.4 %	785 0.4 %
<built-in method nt.stat>	707	80 0.0 %	80 0.0 %
<method 'write' of '_io.TextIOWrapper' objec	14781	131 0.1 %	78 0.0 %
<built-in method _imp.create_dynamic>	15	50 0.0 %	45 0.0 %
encode	14781	53 0.0 %	32 0.0 %
__init__	24778	29 0.0 %	29 0.0 %
<built-in method marshal.loads>	111	22 0.0 %	22 0.0 %
<built-in method _codecs.charmap_encode>	14781	20 0.0 %	20 0.0 %
_path_join	1392	26 0.0 %	17 0.0 %
<method 'read' of '_io.BufferedReader' objec	121	15 0.0 %	15 0.0 %
<built-in method io.open>	4	15 0.0 %	15 0.0 %
<built-in method io.open_code>	121	15 0.0 %	15 0.0 %

En lo que respecta al tiempo de carga para la codificación de la imagen es de 196.709 segundos. Esto es si la imagen es de 100x100 pixeles.

Name	Call Count	Time (ms)	Own Time (ms)
decodificar.py	1	6144 100.0 %	5218 84.9 %
<built-in method builtins.len>	6280445	654 10.6 %	654 10.6 %
<built-in method nt.stat>	707	50 0.8 %	50 0.8 %
<built-in method _imp.create_dynamic>	15	37 0.6 %	34 0.6 %
<built-in method marshal.loads>	111	14 0.2 %	14 0.2 %
_path_join	1392	16 0.3 %	11 0.2 %
<method 'read' of '_io.BufferedReader' objec	121	10 0.2 %	10 0.2 %
<built-in method io.open_code>	121	10 0.2 %	10 0.2 %
__init__	9557	4 0.1 %	4 0.1 %
<built-in method builtins.__build_class__>	228	8 0.1 %	4 0.1 %
find_spec	267	63 1.0 %	3 0.0 %
<method 'pop' of 'list' objects>	4955	3 0.0 %	3 0.0 %
<built-in method nt.listdir>	16	2 0.0 %	2 0.0 %
<built-in method builtins.getattr>	4999	2 0.0 %	2 0.0 %
<built-in method builtins.hasattr>	3248	2 0.0 %	2 0.0 %
<built-in method _imp.exec_dynamic>	15	14 0.2 %	2 0.0 %
update_wrapper	324	2 0.0 %	1 0.0 %
decorator	317	10 0.2 %	1 0.0 %
_collect_parameters	159	2 0.0 %	1 0.0 %

Y con respecto al tiempo de carga para la decodificación de la imagen es de 6.144 segundos. Esto también fue realizado para una imagen de 100x100 pixeles.

En conclusión, el tiempo de carga para la codificación y decodificación de la imagen utilizando la codificación de Huffman aumenta conforme el tamaño de la imagen aumente.

CAPÍTULO V: APLICACIONES Y USOS DE LA CODIFICACIÓN DE IMÁGENES CON HUFFMAN

5.1. Almacenamiento y transmisión de imágenes:

5.1.1. Importancia de la compresión de imágenes en el almacenamiento y transmisión de datos:

La compresión de imágenes desempeña un papel fundamental en el almacenamiento y transmisión de datos, ya que reduce el tamaño de los archivos de imagen sin perder una cantidad significativa de calidad visual. La compresión de imágenes permite ahorrar espacio de almacenamiento y reducir los tiempos de transmisión, lo que facilita el intercambio rápido y eficiente de imágenes en aplicaciones como el correo electrónico, las redes sociales y las plataformas de comercio electrónico. Además, la compresión de imágenes ayuda a mejorar la experiencia del usuario al reducir los tiempos de carga de las páginas web y optimizar la visualización en dispositivos móviles

5.1.2. Cómo la codificación de imágenes con Huffman permite reducir el tamaño de los archivos y ahorrar espacio de almacenamiento:

El método de compresión de imágenes con Huffman es ampliamente utilizado debido a su simplicidad y eficiencia. La codificación de imágenes con Huffman se basa en la estadística de ocurrencia de los símbolos presentes en la imagen y asigna códigos de longitud variable a cada símbolo. Los símbolos más frecuentes en la imagen reciben códigos más cortos, lo que permite una mayor compresión. Esta técnica de codificación permite reducir el tamaño de los archivos de imagen al eliminar la redundancia y representar los datos de manera más eficiente. Al asignar códigos más cortos a los símbolos más comunes, se logra una compresión mayor en comparación con otros métodos de codificación. La codificación de Huffman se ha demostrado efectiva en la compresión de imágenes y ha sido ampliamente adoptada en aplicaciones prácticas.

5.1.3. Ejemplos de aplicaciones prácticas de la codificación de imágenes con Huffman en el almacenamiento y transmisión de imágenes:

En el campo de la transmisión de imágenes en tiempo real, como la videoconferencia o las transmisiones en vivo, la compresión de imágenes con Huffman se utiliza para reducir el ancho de banda requerido y asegurar una transmisión fluida y de calidad. En plataformas de redes sociales y sitios web, la codificación de imágenes con Huffman se emplea para comprimir las imágenes compartidas por los usuarios, garantizando una carga rápida de las páginas y una experiencia fluida para los usuarios.

5.2. Impacto de la compresión de archivos multimedia en la tecnología:

5.2.1. Análisis del impacto de la compresión de archivos multimedia en el rendimiento y la eficiencia de los dispositivos y sistemas tecnológicos:

La compresión de archivos multimedia tiene un impacto significativo en el rendimiento y la eficiencia de los dispositivos y sistemas tecnológicos. La compresión de archivos multimedia reduce la carga de almacenamiento y transmisión de datos, lo que conduce a una mayor eficiencia y una mejor experiencia de usuario.

5.2.2. Contribución de Huffman en la codificación de imágenes y reducción de carga de recursos de almacenamiento y transmisión:

La codificación de imágenes con Huffman es una técnica eficiente para reducir la carga

en los recursos de almacenamiento y transmisión de datos. Al asignar códigos más cortos a los símbolos más frecuentes en una imagen, la compresión con Huffman logra una reducción significativa en el tamaño de los archivos de imagen. Esto mejora la experiencia del usuario al permitir una visualización más rápida y una carga más ágil de las imágenes. Además, la reducción en el tamaño de los archivos facilita el intercambio de imágenes en aplicaciones y plataformas en línea.

5.2.3. Ejemplos de aplicaciones tecnológicas que se benefician de la compresión de imágenes con Huffman:

Las aplicaciones tecnológicas se benefician ampliamente de la compresión de imágenes con Huffman. Por ejemplo, en la transmisión de video en línea, plataformas como YouTube utilizan la compresión de imágenes con Huffman para comprimir los archivos de video y permitir una transmisión más eficiente y rápida.

Además, en el campo de la realidad virtual y aumentada, la compresión de imágenes con Huffman es esencial para garantizar una experiencia inmersiva y fluida en dispositivos con recursos limitados. Al comprimir las imágenes utilizadas en estas aplicaciones, se reduce la carga de procesamiento y se mejora la capacidad de respuesta en tiempo real.

5.3. Utilización en sistemas de reconocimiento de patrones:

5.3.1. Introducción a los sistemas de reconocimiento de patrones:

Los sistemas de reconocimiento de patrones juegan un papel crucial en diversas áreas, como la medicina, la seguridad y la inteligencia artificial. Estos sistemas utilizan algoritmos y técnicas para identificar patrones y características en conjuntos de datos, incluyendo imágenes. Estos permiten automatizar tareas como el reconocimiento facial, la detección de objetos y la clasificación de imágenes, lo que tiene amplias aplicaciones en el diagnóstico médico, la seguridad y la toma de decisiones inteligentes.

5.3.2. Reducción de la complejidad de los datos y acelerar los procesos de análisis:

La codificación de imágenes con Huffman puede ser utilizada en sistemas de reconocimiento de patrones para reducir la complejidad de los datos y acelerar los procesos de análisis. Al comprimir las imágenes con Huffman, se eliminan redundancias y se representan los datos de manera más eficiente, lo que reduce la cantidad de información que debe ser procesada por los algoritmos de reconocimiento de patrones. Además, en la detección de objetos, la codificación de imágenes con Huffman puede utilizarse para comprimir imágenes de referencia y optimizar la búsqueda y el análisis de características relevantes en imágenes de prueba. Esto facilita la identificación y clasificación precisa de objetos en aplicaciones de seguridad y vigilancia.

CAPÍTULO VI: LIMITACIONES Y DESAFÍOS

6.1. Limitaciones de la codificación de Huffman

La codificación de Huffman, a pesar de ser un algoritmo eficiente y ampliamente utilizado para la compresión de datos, presenta ciertas limitaciones que es importante tener en cuenta.

A continuación, se describen algunas de estas limitaciones:

I. Pérdida de información: La codificación Huffman es un método sin pérdidas, lo que significa que no se pierde información durante la codificación y decodificación. Sin embargo, la cantidad de compresión que se puede lograr depende en gran medida de la redundancia presente en los datos. Si los datos no son redundantes, la codificación Huffman no podrá comprimir significativamente los datos e incluso puede aumentar el tamaño del archivo en algunos casos.

II. Dependencia del contexto: La codificación de Huffman es sensible al contexto y depende de la distribución de frecuencia de los símbolos en los datos de entrada. Si la distribución de frecuencias cambia, por ejemplo, al comprimir diferentes tipos de imágenes o documentos de texto, se requiere una nueva codificación Huffman para cada conjunto de datos. Como resultado, aumenta la complejidad de los algoritmos y la necesidad de almacenar más tablas de códigos.

III. Ineficiencia en datos pequeños: La codificación Huffman puede no ser efectiva para comprimir pequeños conjuntos de datos. Dado que el algoritmo se basa en la frecuencia de los símbolos, la compresión resultante puede ser mínima o incluso inexistente si hay pocos símbolos.

IV. Dificultad de acceso a datos comprimidos: La codificación Huffman produce códigos de longitud variable que dificultan el acceso accidental a los datos comprimidos. Para acceder a un símbolo específico, se debe recorrer el código desde el origen o desde un punto de referencia.

V. Complejidad con respecto a otros tipos de codificación: La codificación de Huffman es óptima para la codificación de símbolo por símbolo dada la distribución de probabilidad, pero no es óptima para la codificación de bloques o la codificación dependiente del contexto. Existen otros métodos, como la codificación aritmética o la codificación LZW, que pueden proporcionar una mayor capacidad de compresión en tales casos.

6.2. Desafíos en la implementación y optimización

La implementación de la codificación de Huffman implica varios desafíos relacionados con la construcción y el uso del árbol de Huffman, que es la estructura que almacena los códigos de los símbolos. Algunos de estos desafíos son:

I. Complejidad de construir un árbol de codificación: El primer paso del algoritmo de Huffman consiste en construir un árbol de codificación a partir de la distribución de frecuencias de los símbolos. Este proceso puede ser computacionalmente costoso, especialmente cuando se utilizan grandes conjuntos de datos. Encontrar la estructura de árbol óptima puede llevar mucho tiempo, lo que afecta el rendimiento general del algoritmo.

II. Necesidad de almacenar información adicional: Para decodificar correctamente los datos codificados por Huffman, es necesario almacenar información adicional, como una tabla de codificación o un árbol de codificación. Esto puede requerir espacio de almacenamiento adicional y puede ser difícil en entornos con recursos limitados, como dispositivos móviles o sistemas integrados.

III. Optimización de la eficiencia de la compresión: Aunque el algoritmo de Huffman es intrínsecamente eficiente en términos de compresión, la optimización adicional de la eficiencia de la compresión sigue siendo un desafío. Esto incluye explorar técnicas avanzadas, como adaptar la longitud de la codificación de Huffman a los datos o combinar la codificación de Huffman con otros algoritmos de compresión para obtener mejores resultados.

CAPÍTULO VII: CONCLUSIONES

7.1. Recapitulación de los resultados y hallazgos

En esta monografía, se ha analizado y estudiado la codificación de imágenes con Huffman. Se ha demostrado que el algoritmo de Huffman es capaz de comprimir datos de manera eficiente al aprovechar la redundancia presente en la distribución de frecuencia de los símbolos. Se ha analizado también cómo varía la eficiencia del algoritmo en función del número y la distribución de los símbolos. Y se han presentado ejemplos de imágenes comprimidas utilizando este algoritmo, mostrando la reducción en el tamaño del archivo sin pérdida significativa de calidad de la imagen.

7.2. Contribuciones y aplicabilidad del algoritmo de Huffman

El algoritmo de Huffman ha demostrado ser una herramienta valiosa para comprimir datos, incluidas imágenes. Su capacidad para reducir el tamaño del archivo sin pérdida de datos lo hace útil en aplicaciones que requieren un almacenamiento de datos eficiente, como la transferencia de imágenes a través de una red o el almacenamiento de archivos en dispositivos con recursos limitados. Al igual que el algoritmo de Huffman, también inspiró y contribuyó al desarrollo de otros métodos de compresión como la codificación aritmética, la codificación LZW o la codificación JPEG1.

7.3 Posibles mejoras y futuras investigaciones

Aunque la codificación de Huffman ha logrado algunos resultados, todavía hay margen para mejorar aún más su eficiencia y facilidad de uso. Algunas mejoras potenciales incluyen la exploración de técnicas de codificación adaptativa de longitud variable, la exploración de combinaciones de Huffman y otros algoritmos de compresión, y la exploración de formas de mejorar la eficiencia de la codificación para pequeños conjuntos de datos. Además, se pueden explorar extensiones de la codificación Huffman para abordar problemas específicos de compresión de imágenes, como la preservación de los detalles finos y la reducción del ruido en las imágenes comprimidas.

En resumen, la codificación de Huffman es un algoritmo eficiente y ampliamente utilizado en la compresión de datos, incluyendo imágenes. Aunque presenta limitaciones y desafíos en su

implementación y optimización, sus contribuciones y aplicabilidad en diversas aplicaciones son significativas. Con investigaciones futuras y mejoras adicionales, es posible aprovechar aún más el potencial de la codificación de Huffman en la compresión de imágenes y otros tipos de datos.

BIBLOGRAFIA:

Portal - Hermesoft IG - Plataforma - Universidad de Pamplona. (s. f.). https://www.unipamplona.edu.co/unipamplona/portalIG/home_2++3/recursos/general/11072012/

Garcia, R., Araujo, V., Mascarini, S., Santos, E. G. D., & Costa, A. R. (2018). Is cognitive proximity a driver of geographical distance of university–industry collaboration? Area Development and Policy, 3(3), 349–367. https://doi.org/10.1080/23792949.2018.1484669

Algoritmo de compresión de codificación de Huffman. (s. f.).

https://www.techiedelight.com/es/huffman-coding/