Proyecto Final Redes De Comunicaciones II

Lancheros Jhon Jairo - Código: 20162020077
Rojas Juan Camilo - Código:20162020427
Rojas Jorge Andrés - Código:20161020079
†Universidad Distrital Francisco José de Caldas

***Abstract*—El presente documento tiene el propósito de sustentar el trabajo final para la clase de redes de comunicación II, en el cual por medio de un software propuesto, diseñado e implementado busca sustentar varios de los conceptos claves que hemos estudiado durante el semestre. Hemos hecho énfasis en los algoritmos de codificación y de compresión de archivos.**

The present document has the purpose of supporting the final work for the class of communication networks II, in which by means of a software proposed, designed and implemented it seeks to support several of the key concepts that we have studied during the semester. We have emphasized the file encoding and compression algorithms.

I. Introducción

La comunicación es un proceso que consiste de emisión y recepción de mensajes. Los elementos principales de la comunicación son: el emisor, el mensaje, el receptor y el canal; la dinámica de estos actores es básicamente donde el emisor envía un mensaje al receptor por medio de un canal. Un sistema de comunicaciones proporciona toda la infraestructura necesaria para que este proceso se realice de la mejor manera posible.

Los sistemas de comunicaciones se desarrollo a un ritmo bastante acelerado lo cual ha incrementado el numero de canales disponibles como, por ejemplo, lıneas telefónicas, enlaces de radio, dispositivos de almacenamiento magnético u óptico, entre otros. Existen diversos factores (distorsión, interferencia, radiación, magnetización) que introducen ruido en la transmisión de datos sobre los canales de comunicación. Cuando existe ruido al transmitir datos sobre un canal, es probable que el mensaje recibido por el receptor no sea idéntico al mensaje enviado por el emisor. Cuando esto ocurre, se dice que se produjeron errores en la transmisión de datos sobre el canal.

Estos motivos han impulsado la creación de la teoría de la codificación que estudia la transmisión de datos sobre canales de comunicación con ruido y realiza la búsqueda de códigos para la detección y la corrección de errores introducidos en el canal. Sin embargo, es inútil tener algoritmos para codificar, decodificar, detectar y corregir errores si no son eficientes o no permiten elevadas velocidades de transmisión de datos.

Existen diversos métodos para abordar este problema, uno de los mas conocidos es el código Hamming, publicado por Richard Hamming en 1950[5]. Su propuesta, aun vigente, consiste en agregar redundancia a los datos, a través de bits de paridad (o bits de control) colocados en posiciones especıficas, de manera tal que permitan detectar la presencia de errores dentro del mensaje, lo que proporciona al receptor la posibilidad de corregirlos. Los códigos Hamming pueden detectar errores en uno o en dos bits, y también corregir errores en un solo bit. Siguen siendo los códigos correctores de errores mas importantes desde diversos puntos de vista, tanto teóricos como prácticos, en sistemas modernos de comunicaciones o almacenamiento digital. Los códigos Hamming pueden implementarse tanto en hardware como en software. Los métodos por hardware, en general, tienden a ser mas eficientes. Sin embargo, requieren un gran numero de componentes lo que, además de los costos asociados, los hace inaplicables en algunos contextos como, por ejemplo, comunicaciones micro-satelitales[13]. Pese a que los metodos por software se consideran menos eficientes, esto es cierto para códigos sofisticados de detección de errores como, por ejemplo, Cyclic Redundancy Check (CRC). Existen códigos alternativos a CRC (entre ellos los de Hamming) que, además de proporcionar la posibilidad de realizar implementaciones eficientes por software, ofrecen varias ventajas: independencia de la plataforma, simplicidad de implementación y manipulación mas conveniente de la información (Bytes o palabras en lugar de bits). Estas ventajas y la poca disponibilidad de trabajos que no utilicen enfoques basados en álgebra lineal[7], motivaron la búsqueda de algoritmos eficientes no tradicionales para la codificación y decodificación utilizando códigos Hamming.

Esto también creo la necesidad de que la información no solo viaje protegida sino que también ahorre la mayor cantidad de recursos para su envío mediante procesos de compresión, es el proceso de reducción del volumen de datos para representar una determinada cantidad de información.[9] Es decir, un conjunto de datos puede contener datos redundantes que son de poca relevancia o son datos que se repiten en el conjunto, los cuales si se identifican pueden ser eliminados.

La manera óptima de comprimir datos es utilizar un Código de Huffman. Pero hay otros códigos compresores que, sin ser óptimos, son también muy eficaces, y presentan la ventaja añadida de ser mas sencillos de poner en practica. Un ejemplo son los códigos ZIP que se emplean para comprimir documentos electrónicos.[4]

II. Marco teorico

A. *Código Hamming*

El código de Hamming sirve es un código de distancia 3. Dicho código, consiste en agregar una cierta cantidad de bits adicionales llamados “Bits de Paridad”; la cantidad (K) de bits

de paridad a agregar junto con la cantidad (M) de bits(datos) a enviar debe cumplir con la siguiente ecuación[12]:

$$2^K \geq M + K + 1$$

Fig. 1. Ecuación bits de pariedad

1) Detección de errores: Para detectar un error, en el caso de un código binario, se debe agregar un símbolo binario (0 o 1) a cada cadena de datos de K símbolos de información, de forma que la cantidad total de unos en la cadena codificada sea par, es decir, que la cadena tenga paridad par. La distorsión de algún símbolo traslada la palabra codificada permisible al conjunto de las palabras prohibidas, lo que se detecta en el extremo receptor, debido a la cantidad impar de unos.

Las propiedades de detección y corrección de errores de un código dependen de su distancia de Hamming. Esta representa la cantidad de símbolos en los que una cadena de datos se diferencia de otra cadena y se simboliza con la letra "d". Para determinar la cantidad de bits diferentes, en el caso de una cadena binaria, basta aplicar una operación OR exclusivo a las dos cadenas y contar la cantidad de bits 1 en el resultado; por ejemplo:

10001001
10110001 d=3, N=8
00111000

Fig. 2. Detección del error

En el caso anterior se muestra que la distancia de Hamming es d=3 y la longitud de la cadena de datos es N=8.

La manera de saber si una cadena recibida tiene errores es identificando si dicha cadena es un código permisible. Entonces, si se tiene un código con longitud N=3, indica que se tiene el siguiente código binario:

[000, 001, 010, 011, 100, 101, 110, 111]

Fig. 3. Comprobación del error

Del código anterior se puede aplicar una distancia d=2; esto indica que el código anterior se divide en un código permisible y un código prohibido.

[000, 011, 101, 110] Código permisible
[001, 010, 100, 111] Código prohibido

Fig. 4. Validación del error

El decodificador después de la recepción puede realizarse de forma que la palabra codificada recibida se identifique con la palabra permisible que se encuentre con ella. Si el receptor encuentra que el código que llegó es una palabra prohibida, detecta que hubo un error y pide retransmisión[3].

B. Código Manchester

La idea fundamental detrás de la codificación Manchester es la siguiente: podemos usar transiciones de voltaje , en lugar de niveles de voltaje , para representar unos y ceros. Considere el siguiente diagrama:

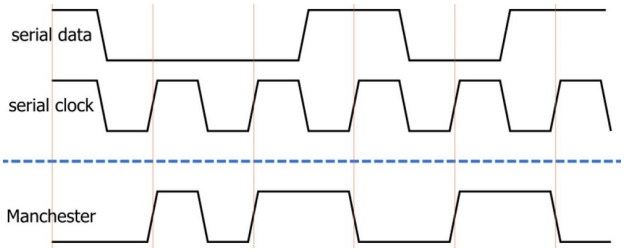


Fig. 5. Transiciones de voltaje

En la parte superior del diagrama tenemos una interfaz digital estándar que consta de una señal de datos y una señal de reloj.

En la parte inferior del diagrama hay una señal codificada por Manchester para los mismos datos. Observe cómo ocurren las transiciones en el medio de los estados lógicos de la señal de datos estándar (en otras palabras, la transición Manchester está alineada con el borde del reloj que se usaría para muestrear los datos). Observe también que un bit lógico alto siempre corresponde a una transición de alto a bajo, y un bit lógico bajo siempre corresponde a una transición de bajo a alto. (También puede usar una transición de bajo a alto para lógica alta y una transición de alto a bajo para lógica baja; lo importante es que el circuito del receptor sepa qué formato esperar).

Está claro de inmediato que se elimina el problema del acoplamiento de CA: cada bit requiere una transición y, por lo tanto, la señal de datos nunca permanecerá en nivel lógico bajo o alto lógico durante un período prolongado de tiempo. Esto es evidente en el siguiente diagrama, que muestra una señal digital estándar para el binario 111111 y una señal codificada en Manchester para la misma secuencia binaria.

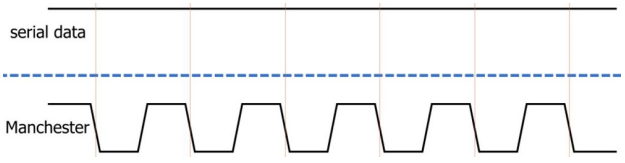


Fig. 6. Secuencia binaria

El problema de sincronización es un poco menos directo porque todavía necesitamos extraer de alguna manera el reloj de la señal; sin embargo, podemos ver intuitivamente que la regularidad de las transiciones proporciona información sobre cuándo se debe muestrear la señal de datos.

El diagrama anterior también muestra una desventaja no trivial de la codificación Manchester: la velocidad de datos se reduce a la mitad en relación con el ancho de banda de la señal de datos. Una señal codificada en Manchester necesita una transición para cada bit, lo que significa que se utilizan dos estados lógicos de Manchester para transmitir un estado lógico estándar. Por lo tanto, se necesita el doble de ancho de banda para transferir datos a la misma velocidad.

Esto puede no parecer un problema, ¿por qué no usar una señal de frecuencia más alta? Bueno, si el ancho de banda de la señal es el factor limitante en la rapidez con que los datos se pueden mover del transmisor al receptor, y si ya está a la velocidad máxima de datos, no puede aumentar la frecuencia de la señal en un factor de dos; en su lugar, debe reducir la velocidad de datos en un factor de dos[4].

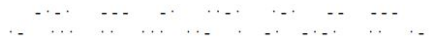
C. Morse

El desarrollo tecnológico es proporcional a la cobertura y velocidad con la cual se transmite la información, es por eso que a lo largo del tiempo se han creado diversos mecanismos para optimizar las comunicaciones. Según Lull, los organismos sobreviven y florecen porque tienen la habilidad de comunicarse efectivamente, ya que la evolución va de la mano con el cambio biológico y cultural. Se han encontrado evidencias arqueológicas de hace 40.000 años que revelan la aparición del lenguaje por parte de los europeos modernos. Desde entonces, han sido diversos los medios de comunicación implementados por parte del ser humano, como las señales de humo, los faroles, los tambores, hasta llegar a técnicas mucho más sofisticadas como la radiofrecuencia o la fibra óptica. Uno de los antecedentes más importantes en el desarrollo de las comunicaciones a distancia se presentó con la invención del semáforo visual en el año 1794 por Claude Chappe, considerado el primer telégrafo, el cual se encontraba limitado por el campo visual del terreno en el que se encontraban los usuarios, ya que se tenía como referencia un mástil que soportaba un elemento llamado regulador, en cuyos extremos contaba con 2 piezas de menor longitud llamados indicadores, este conjunto podía adoptar 196 posiciones y la distancia de transmisión podía aumentar de 12 a 25 Km si se utilizaba un sistema de catalejos (telescopios), la operación del sistema de comunicación era deficiente en la noche. El gran paso en el área de las comunicaciones se dio en el año 1825 con la invención del electroimán por William Sturgeon. Luego, en 1830 Jhosep Henry adaptó el electroimán a un sistema sonoro formado por una campana que se activaba a una distancia de 1.6Kms. No obstante, el desarrollo del telégrafo fue mejorado por Samuel Morse, quien optimizó el diseño de Jhosep por medio de un electroimán que sostenía una pluma que permitía escribir automáticamente puntos o líneas dependiendo de la duración del pulso. Esta técnica sería la base de lo que posteriormente se conocería como Código Morse o Clave Morse[10]. Finalmente, el código Morse fue desarrollado en 1835 por Alfred Vail mientras trabajaba con Samuel Morse. Vail creó un método que aseguraba la transmisión de cada número o letra del alfabeto americano en un código consistente de puntos y rayas (señales telegráficas que se diferencian entre sí por su duración). La idoneidad del sistema fue reconocida por Samuel Morse, quien lo patentó bajo su nombre. Hoy en día, gracias a los avances tecnológicos y a las diferentes necesidades de personas con discapacidad, el código Morse ha llegado a ser implementado como un lenguaje de lectoescritura para compensar algún tipo de dificultad en la comunicación. El código Morse se considera como uno de los Sistemas Alternativos y Aumentativos de Comunicación (SAAC), los cuales son métodos que se han desarrollado para recuperar la capacidad de comunicación en personas que presentan algún tipo de discapacidad. Para esto se han desarrollado diferentes tipos de dispositivos que se adaptan al entorno y a las necesidades de cada persona y, por ejemplo, pueden ser utilizados a partir de la activación de un sensor o un pulsador, por medio de un dedo, la boca, la cabeza, un pie, los párpados etc, dependiendo de la condición motriz en la que se encuentre el usuario. Se han creado diversos métodos para la aplicación del Código Morse como un SAAC. En su versión más básica, no requiere del procesamiento algorítmico de señales, ya que es un intérprete humano el que cumple la función de puente entre el usuario y su entorno. Otros han diseñado diversos tipos de arreglos algorítmicos para lograr hacer los dispositivos más eficientes y efectivos, algunos incluso cuentan con sistemas de predicción de texto. Entre estos se encuentra el algoritmo LeastMean Square (LMS)[2], el cual recalcula la base de tiempo de la señal de menor duración para obtener un resultado acertado, y el arreglo de Huffman, el cual logra disminuir considerablemente el espacio en memoria, aunque requiere de un mayor tiempo en la entrada de datos. Existen interfaces que usan el código Morse para lograr ingresar una cadena de texto, un ejemplo de esto es el aplicativo que por medio de una cámara registra el lado hacia el cual se mueve la lengua y de esta manera lo asocia a los puntos o rayas que maneja el Morse originalmente como lo vemos en el siguiente ejemplo[11]:

Mensaje

"Confirmo asistencia"

Código Morse



Alfabeto (código) Morse

A	·–	H	····	Ñ	––·––	U	··–
B	–···	I	··	O	–––	V	···–
C	–·–·	J	·–––	P	·––·	W	·––
D	–··	K	–·–	Q	––·–	X	–··–
E	·	L	·–··	R	·–·	Y	–·––
F	··–·	M	––	S	···	Z	––··
G	––·	N	–·	T	–		

Fig. 7. Ejemplo de código morse

D. Compresión ZIP

La historia del formato de archivo ZIP se remonta al evento de demanda presentada por System Enhancement Associates

(SEA) contra PKWARE por usar su utilidad ARC sin permisos para su marca registrada y los derechos de autor de la apariencia del producto y la interfaz de usuario. Antes de esto, Phil Katz había reescrito el código fuente de SEA y había lanzado PKXARC, un extractor de ARC, y PKARC, un compresor de archivos, como software gratuito para sistemas basados en MS-DOS. Al perder en la demanda, PKWARE ya no pudo usar nada relacionado con ARC. Aquí es donde surgió la creación de una nueva compresión de archivos, denominada ZIP, que se convirtió en parte de la utilidad PKZIP en PKWARE, Inc.

Katz lanzó las especificaciones del formato de archivo ZIP al dominio público, mientras conservaba los derechos de propiedad sobre su utilidad de compresión y extracción, es decir, PKZIP. El sistema de compresión ZIP fue (y es) capaz de archivar archivos en una carpeta por medio de un algoritmo de verificación de redundancia cíclica (CRC) de 32 bits para comprimir tamaños de archivos. A diferencia de ARC, las carpetas .ZIP incluían un archivo de directorio que desempeñaba el papel del libro de códigos de un criptógrafo, que contenía la información necesaria para procesar los archivos comprimidos[6].

1) Métodos de compresión admitidos:

- Almacenar: no implica compresión
- Encogimiento
- Reducción (esto implica factores de compresión que van desde el nivel 1 al nivel 4)
- Implorar
- Desinflar
- Deflat64
- BZIP2
- LZMA (EFS)
- WavPack
- PPMd versión I, Rev 1

DEFLATE es el método de compresión comúnmente utilizado, que es un algoritmo de compresión de fecha sin pérdidas que utiliza una combinación de la codificación LZ77 y Huffman y se detalla en RFC 1951[1] .

2) Formato de archivo ZIP general: Cada archivo Zip está estructurado de la siguiente manera:

TABLE I
ESTRUCTURA DE UN ZIP

Formato de archivo ZIP
Encabezado de archivo local 1
Datos de archivo 1
Descriptor de datos 1
Encabezado de archivo local 2
Datos de archivo 2
Descriptor de datos 2
....
....
Encabezado de archivo local N
Datos de archivo N
Encabezado de descifrado de archivos
Archivar registro de datos adicionales
Directorio central

El formato de archivo ZIP utiliza un algoritmo CRC de 32 bits para fines de archivo. Para representar los archivos comprimidos, un archivo ZIP contiene un directorio al final que mantiene la entrada de los archivos contenidos y su ubicación en el archivo. Por lo tanto, desempeña el papel de codificación para encapsular la información necesaria para representar los archivos comprimidos. Los lectores ZIP usan el directorio para cargar la lista de archivos sin leer el archivo ZIP completo. El formato mantiene copias duales de la estructura del directorio para brindar una mayor protección contra la pérdida de datos.

Cada archivo en un archivo ZIP se representa como una entrada individual donde cada entrada consta de un encabezado de archivo local seguido de los datos del archivo comprimido. El directorio al final del archivo contiene las referencias a todas estas entradas de archivo. Los lectores de archivos ZIP deben evitar leer los encabezados de los archivos locales y todo tipo de listas de archivos deben leerse desde el directorio. Este directorio es la única fuente de entradas de archivo válidas en el archivo, ya que los archivos también se pueden agregar hacia el final del archivo. Es por eso que si un lector lee los encabezados locales de un archivo ZIP desde el principio, también puede leer entradas no válidas (eliminadas) y que no son parte del directorio que se está eliminando del archivo.

El orden de las entradas del archivo en el directorio central no tiene por qué coincidir con el orden de las entradas del archivo en el archivo[6].

III. PROCESO DE DESARROLLO

Se creó la clase CodificacionHamming que contendrá todo el algoritmo, primeramente creamos un método llamado “anadir” que cuenta con dos parámetros y que nos permitirá editar y guardar archivos en un archivo de texto plano txt.

```
public static void anadir(String original, String codificado) {
    File archivo = new File("codificados.txt");
    FileWriter writer = null;
    try {
        writer = new FileWriter(archivo, true);
        writer.append("El mensaje original es: " + original + "\n");
        writer.append("El mensaje codificado es: " + codificado + "\n\n\n");
    } catch (IOException ex) {
        System.err.println(ex.getMessage());
    } finally {
        if (writer != null) {
            try {
                writer.close();
            } catch (IOException ex) {
                System.err.println(ex.getMessage());
            }
        }
    }
}
```

Fig. 8. Proceso de desarrollo

Después se creó un método llamado “codificar” en el cual se llenara un arreglo con los bits los cuales serán codificados bajo el método de hamming y posteriormente presentados.

```
public void codificar() {
    int n = Integer.parseInt(JOptionPane.showInputDialog("Ingrese el numero de bits para realizar codificaci
    int a[] = new int[n];
    for (int i = 0; i < n; i++) {
        a[n - i - 1] = Integer.parseInt(JOptionPane.showInputDialog("Ingrese el numero " + (n - i) + ":"));
    }
    JOptionPane.showMessageDialog(null, "Usted ingreso:");
    for (int i = 0; i < n; i++) {
        ingresado = ingresado + a[n - i - 1] + "";
    }
```

Fig. 9. Proceso de desarrollo

Después mostraremos el dato cifrado con la codificación hamming gracias al método "generarHamming" y guardaremos estos datos en un txt con el método "anadir"

```
JOptionPane.showMessageDialog(null, ingresado);
int b[] = generarHamming(a);
JOptionPane.showMessageDialog(null, "El codigo hamming generado es:");
for (int i = 0; i < b.length; i++) {
    generado = generado + b[b.length - i - 1];
}
JOptionPane.showMessageDialog(null, generado);
anadir(ingresado, generado);
```

Fig. 10. Proceso de desarrollo

Ahora un paso opcional es ingresar la posición donde queremos ingresar un error para aplicar un algoritmo que encuentra el error, arregla el dato y lo decodifica.

```
int error = Integer.parseInt(JOptionPane.showInputDialog("Ingrese la posicion de un error para verificar
if (error != 0) {
    b[error - 1] = (b[error - 1] + 1) % 2;
}
JOptionPane.showMessageDialog(null, "Codigo con error es:");
for (int i = 0; i < b.length; i++) {
    generadoError = generadoError + b[b.length - i - 1];
}
JOptionPane.showMessageDialog(null, generadoError);
receive(b, b.length - a.length);
```

Fig. 11. Proceso de desarrollo

En el método "generarHamming" retornaremos un array llamado b cuyo tamaño será el del dato original más los datos de paridad, encontraremos el número de bits de paridad requeridos y su ubicación, llenando así el arreglo b.

```
static int[] generarHamming(int a[]) {
    int b[];
    int i = 0, contadorParidad = 0, j = 0, k = 0;
    while (i < a.length) {
        if (Math.pow(2, contadorParidad) == i + contadorParidad + 1) {
            contadorParidad++;
        } else {
            i++;
        }
    }
    b = new int[a.length + contadorParidad];
    for (i = 1; i <= b.length; i++) {
        if (Math.pow(2, j) == i) {
            b[i - 1] = 2;
            j++;
        } else {
            b[k + j] = a[k++];
        }
    }
    for (i = 0; i < contadorParidad; i++) {
        b[((int) Math.pow(2, i)) - 1] = getParidad(b, i);
    }
    return b;
}
```

Fig. 12. Proceso de desarrollo

En la función "getParidad" hallaremos el valor de la posición de paridad, asignándole un 1 o un 0, si la posición de paridad es potencia de 2 el valor será 1 y sino será 0.

```
static int getParidad(int b[], int bitsCorrectos) {
    int paridad = 0;
    for (int i = 0; i < b.length; i++) {
        if (b[i] != 2) {
            int k = i + 1;
            String s = Integer.toBinaryString(k);
            int x = ((Integer.parseInt(s)) / ((int) Math.pow(10, bitsCorrectos))) % 10;
            if (x == 1) {
                if (b[i] == 1) {
                    paridad = (paridad + 1) % 2;
                }
            }
        }
    }
    return paridad;
}
```

Fig. 13. Proceso de desarrollo

Y por último en el método datoRecibido usaremos los datos y la ubicación de la paridad, donde serán verificados, los bits correctos se almacenaran y nos servirán para comprobar posición por posición donde está el error, una vez encontrado será extraído y corregido y en caso que no exista un error únicamente decodificado.

```
static void datoRecibido(int a[], int contadorParidad) {
    int bitsCorrectos;
    int paridad[] = new int[contadorParidad];
    String ubicacionError = new String();
    for (bitsCorrectos = 0; bitsCorrectos < contadorParidad; bitsCorrectos++) {
        for (int i = 0; i < a.length; i++) {
            int k = i + 1;
            String s = Integer.toBinaryString(k);
            int bit = ((Integer.parseInt(s)) / ((int) Math.pow(10, bitsCorrectos))) % 10;
            if (bit == 1) {
                if (a[i] == 1) {
                    paridad[bitsCorrectos] = (paridad[bitsCorrectos] + 1) % 2;
                }
            }
        }
        ubicacionError = paridad[bitsCorrectos] + ubicacionError;
    }
    int posicionError = Integer.parseInt(ubicacionError, 2);
    if (posicionError != 0) {
        JOptionPane.showMessageDialog(null, "El error esta en la posicion " + posicionError + ".");
        a[posicionError - 1] = (a[posicionError - 1] + 1) % 2;
        JOptionPane.showMessageDialog(null, "El codigo correcto es:");
        for (int i = 0; i < a.length; i++) {
            generadoCorrecto = generadoCorrecto + a[a.length - i - 1];
        }
        JOptionPane.showMessageDialog(null, generadoCorrecto);
    } else {
        JOptionPane.showMessageDialog(null, "No hay error en el dato recibido.");
```

Fig. 14. Proceso de desarrollo

```
JOptionPane.showMessageDialog(null, "El dato decodificado es:");
bitsCorrectos = contadorParidad - 1;
for (int i = a.length; i > 0; i--) {
    if (Math.pow(2, bitsCorrectos) != i) {
        ingresadoDecodificado = ingresadoDecodificado + a[i - 1];
    } else {
        bitsCorrectos--;
    }
}
JOptionPane.showMessageDialog(null, ingresadoDecodificado);
```

Fig. 15. Proceso de desarrollo

A. *Código Manchester*

Para la implementación del código Manchester tenemos el siguiente fragmento de código que me da la funcionalidad de un botón que me obtiene los datos en binario para ser graficados.

```
private void manchesterMouseClicked(MouseEvent evt) {

    try {

        this.codigo_captura = this.codigo.getText();
        int[] vector = new int[this.codigo_captura.length()];
        for (int x = 0; x < this.codigo_captura.length(); x++) {
            String aux = this.codigo_captura.substring(x, x + 1);
            vector[x] = Integer.parseInt(aux);
        }
        this.opcion = 4;
        this.referenciad.ponervector(vector, this.codigo_captura.length(), this.opcion);
        this.referenciad.repaint();

    } catch (Exception e) {
        System.out.println("Error Código Manchester.");

    }

}
```

Fig. 16. Capturar Datos - Manchester.

Se obtiene un archivo JPG con el resultado de la codificación en Manchester.

```
private void guardarMouseClicked(MouseEvent evt) {
    BufferedImage imagen = new BufferedImage(this.referenciad.getWidth(), this.referenciad.getHeight(), 1);
    Graphics g = imagen.getGraphics();
    this.referenciad.paint(g);
    File fichero = new File("foto.jpg");
    String formato = "jpg";
    try {
        ImageIO.write(imagen, formato, fichero);
        JOptionPane.showMessageDialog(null, "Se Ha Guardado La Imagen En La Raiz Del Proyecto");
    } catch (IOException e) {
        System.out.println("Error Al Exportar La Imagen");
    }
}
```

Fig. 17. Generar JPG - Manchester.

Para graficar la referencia Unipolar.

```
try{

super.paintComponent(g);
setSize(450, 600);
g.drawString("REFERENCIA - UNIPOLAR", 150, 20);
int y = 58;
while (y <= 400) {
  int x = 40;
  while (x <= 200) {
    g.setColor(Color.BLUE);
    g.drawString("|", y, x);
    x += 20;
  }
  y += 30;
}
g.setColor(Color.black);
g.drawLine(30, 10, 30, 200);
g.drawLine(30, 200, 400, 200);
g.drawLine(25, 100, 35, 100);
g.drawString("1", 18, 105);
int vax = 30;
int vay = 200;
for (int i = 0; i < this.indice; i++) {
  if (this.vectorfinal[i] == 0) {
    g.setColor(Color.RED);
    g.drawLine(vax, vay, vax + 30, vay);
    vax += 30;
    if (i != this.indice - 1 &&
      this.vectorfinal[i + 1] == 1)
      g.drawLine(vax, vay, vax, vay - 100);
  }
  if (this.vectorfinal[i] == 1) {
    g.setColor(Color.RED);
    g.drawLine(vax, vay - 100, vax + 30, vay - 100);
    vax += 30;
    if (i != this.indice - 1 &&
      this.vectorfinal[i + 1] == 0)
      g.drawLine(vax, vay - 100, vax, vay);
  }
}
```

Fig. 18. Referencia Unipolar - Manchester.

Para la lógica del Código Manchester.

```
if (this.opcion == 4) {
  g.setColor(Color.black);
  g.drawString("MANCHESTER", 170, 230);
  int y2 = 58;
  while (y2 <= 400) {
    int x2 = 250;
    while (x2 <= 550) {
      g.setColor(Color.BLUE);
      g.drawString("|", y2, x2);
      x2 += 20;
    }
    y2 += 30;
  }
  g.setColor(Color.black);
  g.drawLine(30, 250, 30, 550);
  g.drawLine(30, 400, 400, 400);
  g.drawLine(25, 300, 35, 300);
  g.drawString("1", 18, 305);
  g.drawLine(25, 500, 35, 500);
  g.drawString("-1", 13, 505);
  vax = 30;
  vay = 400;
  for (int x = 0; x < this.indice; x++) {
    if (x + 1 < this.indice) {
      if (this.vectorfinal[x] == 1 && this.vectorfinal[x + 1] == 1) {
        g.setColor(Color.RED);
        g.drawLine(vax, vay + 100, vax + 15, vay + 100);
        vax += 15;
        g.drawLine(vax, vay + 100, vax, vay - 100);
        vay -= 100;
        g.drawLine(vax, vay, vax + 15, vay);
        vax += 15;
        g.drawLine(vax, vay, vax, vay + 200);
        vay = 400;
      }
      if (this.vectorfinal[x] == 1 && this.vectorfinal[x + 1] == 0) {
        g.setColor(Color.RED);
        g.drawLine(vax, vay + 100, vax + 15, vay + 100);
        vax += 15;
        g.drawLine(vax, vay + 100, vax, vay - 100);
        vay -= 100;
        g.drawLine(vax, vay, vax + 15, vay);
        vax += 15;
        vay = 400;
      }
```

Fig. 19. Lógica Codificación Manchester PT1.

```
      if (this.vectorfinal[x] == 0 && this.vectorfinal[x + 1] == 1) {
        g.setColor(Color.RED);
        g.drawLine(vax, vay - 100, vax + 15, vay - 100);
        vax += 15;
        g.drawLine(vax, vay - 100, vax, vay + 100);
        vay += 100;
        g.drawLine(vax, vay, vax + 15, vay);
        vax += 15;
        vay = 400;
      }
      if (this.vectorfinal[x] == 0 && this.vectorfinal[x + 1] == 0) {
        g.setColor(Color.RED);
        g.drawLine(vax, vay - 100, vax + 15, vay - 100);
        vax += 15;
        g.drawLine(vax, vay - 100, vax, vay + 100);
        vay += 100;
        g.drawLine(vax, vay, vax + 15, vay);
        vax += 15;
        g.drawLine(vax, vay, vax, vay - 200);
        vay = 400;
      }
    }
  }
  if (this.vectorfinal[this.indice - 1] == 0) {
    g.setColor(Color.RED);
    g.drawLine(vax, vay - 100, vax + 15, vay - 100);
    vax += 15;
    g.drawLine(vax, vay - 100, vax, vay + 100);
    vay += 100;
    g.drawLine(vax, vay, vax + 15, vay);
    vax += 15;
    vay = 400;
  }
  if (this.vectorfinal[this.indice - 1] == 1) {
    g.setColor(Color.RED);
    g.drawLine(vax, vay + 100, vax + 15, vay + 100);
    vax += 15;
    g.drawLine(vax, vay + 100, vax, vay - 100);
    vay -= 100;
    g.drawLine(vax, vay, vax + 15, vay);
    vax += 15;
    vay = 400;
  }
```

Fig. 20. Lógica Codificación Manchester PT2.

B. Compresión zip

Para realizar la compresión **ZIP** de las simulaciones de las codificaciones **Hamming**,**Morse** y **Manchester** utilizamos las siguientes librerias que ayudarán a crear el archi ZIP donde se insertarán el .txt y el .jpg que nos genera el proyecto en su raíz.

- import java.nio.file.Files;
- import java.io.File;
- import java.util.zip.ZipEntry;
- import java.util.zip.ZipOutputStream;

A continuación se muestra el botón encargado de solicitar la tarea de compresión.

```
try {
    final String[] rutaFichero = {
        "codificados.txt",
        "foto.jpg"
    };

    Inicio.ficheros(rutaFichero,"CodificacionCompresaRedesII.zip");
    System.exit(0);
} catch (IOException ex) {
    JOptionPane.showMessageDialog(null, "No Hay Archivos Para Comprimir, Realice Las Simulaciones Antes De Comprimir;");
}
```

Fig. 21. Proceso de desarrollo ZIP

La tarea de compresión se realiza de la siguiente forma.

```
private static ZipOutputStream zos = null;
private static ZipEntry ze = null;

public static void ficheros(final String[] ficheros, final String nombreZip) throws IOException {

    // PREPARACIÓN DEL FICHERO ZIP.

    zos = new ZipOutputStream(new FileOutputStream(new File(nombreZip)));

    // INSERCIÓN DE LOS FICHEROS AL FICHERO ZIP.

    for (int i = 0; i < ficheros.length; i++) {
        // INTRODUCIMOS EL FICHERO VACÍO CON SU NOMBRE Y EXTENSIÓN.
        ze = new ZipEntry(ficheros[i]);
        zos.putNextEntry(ze);
        // INTRODUCIMOS LOS DATOS DEL FICHERO VACIÓ INTRODUCIDO.
        byte[] readAllBytesOfFile = Files.readAllBytes(new File(ficheros[i]).toPath());
        zos.write(readAllBytesOfFile, 0, readAllBytesOfFile.length);
    }

    // CERRAMOS LOS FLUJOS DE DATOS.
    zos.closeEntry();
    zos.close();
}
```

Fig. 22. Proceso de desarrollo ZIP

C. Morse

Creamos una clase llamada Morse en la que importaremos la clase Hashtable para crear datos pares de igual valor, donde definiremos todos los caracteres del código ASCII y su equivalencia en el código morse.

```
public static Hashtable<String, String> obtenerEquivalencias() {
    Hashtable<String, String> equivalencias = new Hashtable<>();
    equivalencias.put("A", ".-");
    equivalencias.put("B", "-...");
    equivalencias.put("C", "-.-.");
    equivalencias.put("CH", "----");
    equivalencias.put("D", "-..");
    equivalencias.put("E", ".");
    equivalencias.put("F", "..-.");
    equivalencias.put("G", "--.");
    equivalencias.put("H", "....");
    equivalencias.put("I", "..");
    equivalencias.put("J", ".---");
    equivalencias.put("K", "-.-");
    equivalencias.put("L", ".-..");
    equivalencias.put("M", "--");
    equivalencias.put("N", "-.");
    equivalencias.put("Ñ", "--.--");
```

Fig. 23. Proceso de desarrollo Morse

Creamos un método que retorna la posición de los caracteres del texto que ingresamos

```
public static String morseAAscii(String morseBuscado) {
    Hashtable<String, String> equivalencias = obtenerEquivalencias();
    Set<String> claves = equivalencias.keySet();
    for (String clave : claves) {
        String morse = equivalencias.get(clave);
        if (morse.equals(morseBuscado)) {
            return clave;
        }
    }
    return "";
}
```

Fig. 24. Vocabulario - Proceso de desarrollo Morse

Con el método codificarMorse extraemos el equivalente de cada posición de los caracteres y su equivalente en morse y los separamos con un espacio.

```
public static String codificarMorse(String original) {
    StringBuilder codificado = new StringBuilder();
    for (int i = 0; i < original.length(); i++) {
        String charComoCadenaYEnMayusculas = String.valueOf(original.charAt(i)).toUpperCase();
        String equivalencia = asciiAMorse(charComoCadenaYEnMayusculas);
        codificado
                .append(equivalencia)
                .append(" ");
    }
    return codificado.toString();
}
```

Fig. 25. Proceso de desarrollo Morse

IV. Interfaz Gráfica

En la pantalla de inicio tenemos la opción para elegir cualquiera de los 3 métodos de codificación, además de poder comprimir el proyecto en formato

V. Conclusiones

El estudio experimental realizado mostró que, en todos los casos de prueba, los algoritmos encuentran soluciones óptimas en tiempo con respecto a las funciones teóricas de cota presentadas en el marco teórico del documento. De esta manera, se puede destacar que efectivamente se logra un rendimiento óptimo para el procesamiento de codigos Hamming bajo el esquema propuesto, pasando por todas sus fases internas.

Siguiendo los pasos históricos del esquema Zip hemos mostrado cómo la implementación práctica de una variante del algoritmo Deflate utilizó bit- Análisis sintáctico óptimo

10 años antes de la primera solución teórica eficiente. Recientemente, Google ha anunciado el desarrollo del esquema de compresión Zopfli , con el objetivo de reemplazar la biblioteca zlib basada en Deflate con una nueva implementación optimizada, que preservará el formato de archivo Deflate . En otras palabras, Zopfli utilizará un análisis de bits óptimo[8].

LIST OF FIGURES

1 Ecuación bits de pariedad 2
2 Detección del error 2
3 Comprobación del error 2
4 Validación del error 2
5 Transiciones de voltaje 2
6 Secuencia binaria 2
7 Ejemplo de código morse 3
8 Proceso de desarrollo 4
9 Proceso de desarrollo 5
10 Proceso de desarrollo 5
11 Proceso de desarrollo 5
12 Proceso de desarrollo 5
13 Proceso de desarrollo 5
14 Proceso de desarrollo 5
15 Proceso de desarrollo 5
16 Capturar Datos - Manchester. 6
17 Generar JPG - Manchester. 6
18 Referencia Unipolar - Manchester. 6
19 Lógica Codificación Manchester PT1. 6
20 Lógica Codificación Manchester PT2. 6
21 Proceso de desarrollo ZIP 7
22 Proceso de desarrollo ZIP 7
23 Proceso de desarrollo Morse 7
24 Vocabulario - Proceso de desarrollo Morse . . . 7
25 Proceso de desarrollo Morse 7

LIST OF TABLES

I Estructura de un ZIP 4

REFERENCES

1 Deflate compressed data format specification version 1.3.
2 Shih C.H. and Luo C. H. A morse-code recognition system with lms and matching algorithms for persons with disabilities. *International Journal of Medical Informatics*, 44(3), 1997.
3 Hector Arturo Florez Fernandez and Norberto Novoa Torres. DetecciÓn y correcciÓn de errores mediante el cÓdigo hamming. *Revista vínculos*, 9(2):5–10, dic. 2012.
4 ADOLFO QUIRÓS GRACIÁN. La teoría de códigos: una introducción a las matemáticas de la transmisión de información.
5 R. W Hamming. Error detecting and error correcting codes. *Bell System Technical Journal*, 26:147–160, 1950.
6 Kashif Iqbal, Dec 2019.
7 J. M Jacobsmeyer. Introduction to error-control coding, 2004.
8 Alessio Langiu. On parsing optimality for dictionary-based text compression—the zip case. *Journal of Discrete Algorithms*, 20:65–70, 2013.
9 Javier Lezama. Compresión de imágenes-codificación de huffman. *Revista de educación matemática*, 32(1), 2017.
10 Sergio Leonardo Beltrán Patiño, Sergio Mendoza Puentes, and Alfredo Espitia Beltrán. Teclado basado en código morse para la comunicación de personas con parálisis. *Revista Latinoamericana en Discapacidad, Sociedad y Derechos Humanos*, 3(2), 2019.
11 Alberto Prieto, Antonio Lloris, and Juan Carlos Torres. *Introducción a la Informática*, volume 20. McGraw-Hill, 1989.
12 MUSSO RODRÍGUEZ and GUILLERMO ANDRÉ. CÓdigo hamming, 2004.
13 M. E Saturno. Software error protection technic for high density memory. *4th International Academy of Astronautics Symposium on Small Satellites for Earth Observation*, 1:406–409, 2003.