INSTITUTO POLITÉCNICO NACIONAL
ESCUELA SUPERIOR DE FÍSICA Y MATEMÁTICAS
ALUMNO: LEAL CISNEROS VICTOR HUGO **BOLETA:** 2020330097
PROFESOR: CORONADO GARCIA LUIS CARLOS

Compresión de datos Huffmann y RLE

Índice

1. La compresión de datos: conceptos básicos.

Desde principios del siglo se ha dado un aumento tanto de la capacidad de almacenamiento de datos como en la velocidad de procesamiento de las computadoras, aunado a la tendencia en la disminución de costos en memoria principal y secundaria así como también un aumento de velocidad de estos dispositivos de almacenamiento.
Todo este auge tecnológico que han tenido las redes de computadoras, demanda más prestaciones, que están por encima de las posibilidades reales. El principal problema al que se enfrentan las redes de comunicación es la velocidad de transferencia de datos, y es que, el cambio a mayores velocidades no es tarea fácil, básicamente por razones como la no factibilidad de realizar cambios de infraestructura en las grandes compañías de redes (como el cableado, tecnologías, etc.) así como la falta de tecnología que acepte unas velocidades muy elevadas de transmisión.
En este entorno, para conseguir mayores prestaciones de velocidad, se debe recurrir a técnicas que permitan superar de alguna manera las deficiencias físicas de la red. La técnica más importante en este sentido es la ***compresión de datos***.
Esta es beneficiosa en el sentido de que el proceso de **compresión-transmisión-descompresión** es más rápido que el proceso de **transmisión sin compresión**, además, esta no sólo es útil para la transmisión de datos, si no también para el *almacenamiento masivo*.
La necesidad de almacenamiento también crece por encima de las posibilidades del crecimiento de los discos duros o memoria. Por ejemplo, aplicaciones actuales como el proyecto del Genoma Humano o los servidores de vídeo en demanda, requieren de varios Gigabytes de almacenamiento.
En general, podemos resumir en que, la motivación para aplicar compresión a los datos es la reducción de costo, tanto en almacenamiento (con el fin de ahorrar mayor espacio de memoria) como en la transmisión de los datos (pues se transmiten de forma mas rápida empleando el mismo ancho de banda).
Sin embargo, nada otorga solo beneficios, pues el precio que debe pagarse es cierto *tiempo de cómputo para comprimir y descomprimir los datos* y es que siempre ha existido un compromiso entre los beneficios de compresión y el costo comunicacional requerido.

1.1. Importancia de la compresión de datos

Comencemos dando una definición formal:

Def. 1. ***Compresión de datos:*** *es el proceso de convertir una cadena de datos de entrada (la cadena fuente o los datos originales a tratar), en otra cadena de datos (la salida, la cadena de bits, o la cadena comprimida), que tenga un tamaño más pequeño(Una cadena es, o un archivo o un buffer en la memoria.).*

El campo de la compresión de datos se llama, a menudo, **source coding** (codificación de fuente en español). Suponemos que la entrada de símbolos (como bits, códigos ASCII, bytes, ejemplos de audio, o valores pixel), es emitida por una cierta fuente de información y tiene que codificarse antes de ser enviada a su destino. La fuente puede tener o no memoria; en el caso de que la tenga, cada símbolo es independiente de sus predecesores. Caso contrario, cada símbolo depende de algunos de sus predecesores, y tal vez, también de sus sucesores, por lo que están correlacionados. Una fuente sin memoria se denomina también "*independiente e idénticamente distribuida*" o en sus siglas **IID**.
No existe de manera general un método de compresión universal, por lo que hay diversos métodos, todos basados en ideas diferentes, apropiadas para distintos tipos de datos y que producen diferentes resultados, pero todos se basan en el mismo principio: comprimir eliminando la **redundancia** de los datos originales del archivo de fuente.

Cuando hablamos de dicha *redundancia* en los datos, hablamos de que poseen alguna *estructura*, la cual puede ser aprovechada para lograr una representación más pequeña de los datos; una representación, donde *no hay ninguna estructura aparente.*
La idea de comprimir reduciendo la redundancia nos sugiere la:

Def. 2. ***Ley de la compresión de datos:*** *Se debe asignar códigos cortos para los eventos comunes (símbolos o frases) y códigos largos para los eventos raros*

Hay muchas maneras de aplicar esta ley y al final, un análisis de cualquier método de compresión muestra que, en el fondo, funciona obedeciendo a la ley general.
Entonces podemos decir que la compresión de datos *se realiza mediante el cambio de su representación, de ineficiente (i.e., larga) a eficiente (corta)*, por lo tanto:
La compresión es posible, sólo porque normalmente los datos se representan en la computadora en un formato más largo, que el absolutamente necesario.
La razón de que esas ineficientes (en longitud) representaciones de datos se utilicen continuamente, es que hacen más fácil procesar los datos, y el procesamiento de los datos es más común e importante que la compresión de los mismos.
El **código ASCII** para caracteres, es un buen ejemplo de representación de datos más larga de lo necesario. Utiliza códigos de 7 bits, porque es fácil trabajar con códigos de tamaño fijo.
Una vez que ponemos sobre la mesa el principio de la compresión por eliminación de redundancia, es natural hacerse la pregunta: *"¿Por qué un archivo ya comprimido no puede comprimirse más?"*
La respuesta, por supuesto, es que dicho archivo tiene poca ó ninguna redundancia, por lo que no hay nada que eliminar.
Un ejemplo es un fichero con texto aleatorio, ya que en este, cada letra aparece con igual probabilidad, por lo que asignándoles códigos de tamaño fijo no se añade ninguna redundancia, entonces no hay redundancia que eliminar.
Otro aspecto importante, es que si fuera posible comprimir un archivo ya comprimido, tras sucesivas compresiones se podría reducir el tamaño del mismo hasta llegar a un solo byte, o incluso un solo bit, lo cual es ridículo ya que un solo bit no puede contener la información presente en cualquier fichero grande.
Algo importante a resaltar es que:
ningún método de compresión puede aspirar a comprimir todos los archivos, o incluso un porcentaje significativo de ellos.
Para comprimir un archivo de datos, el *algoritmo de compresión* tiene que examinar los datos, encontrar las redundancias existentes y tratar de eliminarlas y dichas redundancias dependen del tipo de datos, ya sea texto, imágenes, sonido, etc., que es por lo que se tiene que desarrollar un nuevo método de compresión, adaptado para que funcione lo mejor posible, para cada tipo específico de datos.

1.2. Teoría SP y compresión con y sin perdida

Como podemos ver la compresión de datos se ha vuelto tan importante, que algunos investigadores, han propuesto **la teoría SP** (*por "sencillez" "poder"*) la cual sugiere que todo cálculo es compresión, en especifico nos dice:

Def. 3. ***Teoría SP:*** *La compresión de datos, puede ser interpretada como un proceso para eliminar la complejidad innecesaria (redundancia) de la información, y de ese modo, maximizar la sencillez, mientras se conserva lo más posible su poder descriptivo, no redundante.*

Esta teoría está basada en una serie de conjeturas o conceptos importantes a la hora de hablar de compresión de datos, dichas conjeturas son las siguientes:

- El proceso para encontrar y eliminar la redundancia, siempre se puede entender a un nivel fundamental, como un proceso de búsqueda de patrones que coincidan entre sí, y la fusión o unificación de los casos repetidos de algún patrón, para construir otro. La mayoría de los métodos, se clasifican en cuatro categorías: codificación *run length* (**RLE** o **Run Length Encoding**), métodos estadísticos, métodos basados en diccionarios y transformadas. Dentro de este trabajo nos enfocaremos únicamente en el método **RLE** y el **método estadístico de codificación Huffmann**.

- Para poder relaizar una tarea de compresión-descompresión es necesario dos herramientas basicas: el *compresor o codificador*, que es el programa que comprime los datos brutos de la secuencia de entrada y crea una secuencia de salida, formada por datos comprimidos (con baja redundancia),y el *descompresor o decodificador*, que realiza la conversión en la dirección opuesta.
  Como parte de la terminología, cuando hablamos del flujo de entrada de origen, se utilizan los términos **sin codificar, en bruto, o datos de origen**; así mismo, al contenido de la cadena final, comprimida, se conoce como **datos codificados o comprimidos**.

- La compresión de datos puede tener 2 tipos principales de variantes:

  Def. 4. ***Compresión sin perdida:*** *En compresión sin pérdida, es posible reconstruir en el proceso de descompresión, exactamente los datos o cadena original dado un archivo comprimido.*

  Este tipo de compresión es común en los archivos de texto, especialmente aquellos que contienen programas informáticos que pueden quedar invalidados, incluso si sólo se modifica un bit, es por ello que tales archivos, suelen comprimirse solamente con métodos de compresión sin pérdidas.

  Def. 5. ***Compresión con perdida:*** *En compresión con pérdida, se logra una "mejorçompresión perdiendo algún tipo de información en el proceso, por lo que, en la descompresión produce solamente una aproximación a los datos originales.*

  En esta variante, cuando se descomprime la cadena comprimida, el resultado no es idéntico a la secuencia de datos original, además es donde se logran mejores *razones de compresión*, pues siempre existe algún tipo de perdida de información. Este método es usada a menudo para aplicaciones de compresión de imágenes, audio y vídeo destinadas principalmente al entretenimiento, perdiendo sobre todo información que a los sentidos humanos es imperceptible. Es importante señalar que también existen método de compresión sin perdida para archivos multimedia, como es el caso en aplicaciones de compresión de imágenes médicas (identificación de tumores o anomalías) o archivos de audio de alta fidelidad.

- ***El modelo de probabilidad***. Este concepto es importante en los métodos de *compresión de datos estadísticos* ya que tiene que construirse un modelo de los datos, antes de poder comenzar la compresión. Un modelo típico, puede construirse leyendo por completo los datos de entrada, contando el número de veces que aparece cada símbolo (es decir su frecuencia de aparición), y calculando la probabilidad de ocurrencia de cada uno de estos. Entonces se vuelven a leer los datos de entrada, símbolo a símbolo, y se comprimen con la información del modelo de probabilidad.
  Como vemos, hacer la lectura completa de los datos de entrada dos veces es lenta, razón por la cual los métodos de compresión más prácticos utilizan estimaciones o se adaptan a los datos a medida que son leídos de la secuencia de entrada y comprimidos.

Es fácil escanear grandes cantidades de, digamos, un texto en inglés y calcular las frecuencias y las probabilidades de cada carácter. Esta información, puede servir después como un modelo aproximado y puede ser utilizado por los métodos de compresión de texto para comprimir cualquier texto en inglés.
También es posible comenzar asignando probabilidades iguales a todos los símbolos de un alfabeto; luego, al mismo tiempo que se leen los símbolos y son comprimidos, se calculan también las frecuencias y se va construyendo el modelo de probabilidad.

1.3. Métricas de evaluación de la compresión

Rendimiento de la compresión: Es posible utilizar ciertas medidas estándar para expresar el rendimiento de un método de compresión, algunas de las más importantes son:

1.

Def. 6. *La **ratio** o **razón de compresión** se define como:*

$$\textit{Razón de compresión} = \frac{\textit{Tamaño de la cadena de salida}}{\textit{Tamaño de la cadena de entrada}}$$

Un valor de 0,6, significa que los datos ocupan, tras la compresión, un 60 % que su tamaño original. Por otro lado, valores mayores que 1, implican un flujo de salida mayor que el de entrada (es decir una *compresión negativa*).
La razón de compresión, puede llamarse también **bpb** (bit por bit), ya que es igual, al número de bits necesarios de la cadena comprimida, en promedio, para comprimir un bit de la cadena de entrada.
En la actualidad, eficientes métodos de compresión de texto, hacen que tenga sentido hablar de *"bits por carácter"* o **bpc**, es decir el *número de bits que se necesitan, en promedio, para comprimir un carácter de la secuencia de entrada.*
Debemos mencionar otros dos términos relacionados con la razón de compresión, el primero es la **tasa de bits**, el cual es un término general para **bpb** y **bpc**, de modo que, podemos decir que el principal objetivo de la compresión de datos, es *representar cualquier dato en tasas de bits bajas.*
El segundo término es el de **bit de coste** y se refiere a la función que desempeñan los bits individuales en la cadena comprimida, es decir, imaginemos una secuencia de datos comprimida, donde el 90 % de los bits son códigos de tamaño variable de algunos símbolos, y el restante 10 % se utiliza para codificar ciertas tablas; el bit de coste para las tablas es del 10 %.

2.

Def. 7. ***Factor de compresión** se define como la inversa de la razón de compresión, es decir:*

$$\textit{Factor de compresión} = \frac{\textit{Tamaño de la cadena de entrada}}{\textit{Tamaño de la cadena de salida}}$$

En este caso, los valores superiores a 1 indican compresión, y los valores menores que 1 implican *expansión.* Esta medida parece natural a mucha gente, ya que *cuanto mayor sea el factor, mayor será la compresión.*

3.

Def. 8. ***Ganancia de compresión*** *se define como:*

$$100 \ln \frac{\textit{Tamaño de referencia}}{\textit{Tamaño comprimido}},$$

donde el *tamaño de referencia* es el tamaño del flujo de datos entrantes, o el tamaño de una cadena comprimida (producida por algún método de compresión de datos estándar y sin pérdidas). Para números pequeños x, es cierto que $\ln(1+x) \approx x$, por lo que un pequeño cambio en la ganancia de compresión es muy similar al mismo cambio en la razón de compresión. Debido a la función logaritmo, dos ganancias de compresión pueden ser comparadas, sencillamente obteniendo su diferencia. La unidad de la ganancia de compresión se llama *tanto por ciento de la razón logarítmica.*

4. La **velocidad de compresión** puede medirse en *ciclos por byte* (**CPB**). Este es el número promedio de ciclos de máquina, que se necesita para comprimir un byte. Esta medida es importante cuando la compresión se hace por un hardware especial.

2. Compresión RLE

Supongamos lo siguiente: Si un dato d aparece n veces consecutivas en el flujo de entrada, entonces podemos cambiar las n ocurrencias con el par único nd. Dichas n apariciones consecutivas de un elemento en nuestros datos se llama **run length** de n, y este enfoque para la compresión de datos se llama codificación ***RLE*** *(Run-Length Encoding).*
Este método de compresión es uno de los mas simples de compresión de datos utilizado para reducir la cantidad de información necesaria para representar una secuencia repetitiva de datos. Como vemos el algoritmo se basa en el concepto de *çorridas"* o *repeticiones"*, donde una secuencia de elementos repetidos se reemplaza por un único elemento seguido del número de veces que se repite.
Podemos describir el proceso de compresión RLE de la siguiente manera:

- Se recorre la secuencia de datos y se identifican las repeticiones. Cuando se encuentra una secuencia repetitiva, se cuenta el número de veces que se repite.
- Se reemplaza la secuencia repetitiva por un solo elemento seguido del número de repeticiones. Por ejemplo, si se tiene una secuencia de "$AAAAA$", se reemplaza por "$A5$".
- Este proceso se repite hasta que se haya recorrido toda la secuencia de datos.

como ejemplo tenemos que, si se tiene una secuencia como "$AAAABBBCCCCDD$", se puede comprimir como "$A4B3C4D2$", lo que reduce significativamente el tamaño de la secuencia.
Sin embargo podemos encontrar ciertos problemas cuando hablamos de este tipo de compresión. Supongamos que tenemos que comprimir la cadena $all \sqcup is \sqcup too \sqcup well$, entonces, realizando el proceso descrito anteriormente tendremos $a2l \sqcup is \sqcup t2o \sqcup we2l$; el primer problema en el proceso es que, a pesar de que para nosotros puede parecer adecuado, para que un descompresor lo lea, resultara ambiguo, puesto que no tiene una forma de identificar en que momento se esta realizando el remplazo de las repeticiones.
Un camino para resolver el problema es preceder cada repetición con un **carácter especial de cambio de código** (o *código de escape*). Si usamos @ como carácter de cambio de código, entonces la cadena $a@2l \sqcup is \sqcup t@2o \sqcup we@2l$, puede ser descomprimida sin ambigüedad.
Sin embargo, ahora tenemos otro problema, y es que esta cadena es más larga que la original, ya que *sustituye dos letras consecutivas, con tres caracteres.* Para resolver este problema podemos adoptar

la convención de que sólo se reemplacen por un factor de repetición, aquellos grupos compuestos por tres o más repeticiones de un mismo carácter.
Una vez resuelto estos problemas podemos generar un diagrama de como funcionaria un compresor de texto *RLE*.

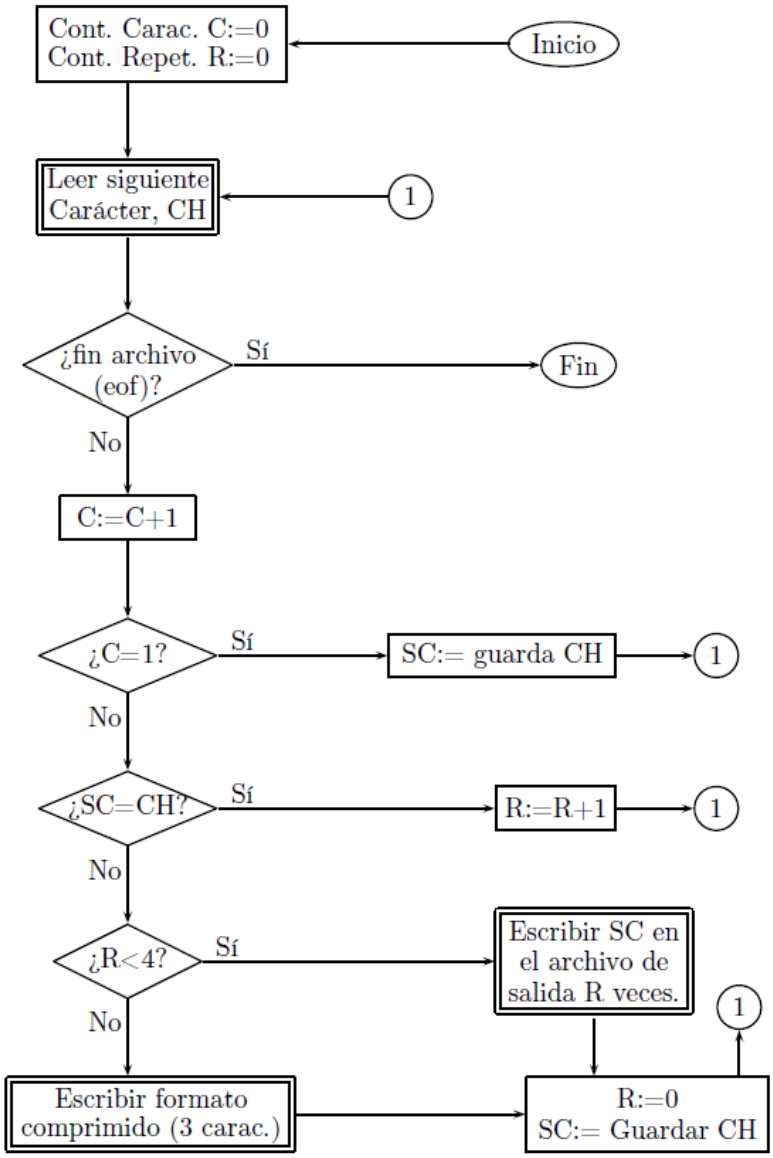

Fig. 1: *RLE: Proceso de compresión*

Después de leer el primer elemento al que llamaremos CH, el contador de caracteres es $C = 1$, por lo que CH se almacena es decir $SC :=$ guarda CH.
Los siguientes elementos se comparan con éste y preguntamos si $SC = CH$; si son idénticos, se incrementa el contador de repeticiones $R := R + 1$.
Cuando se lee un carácter diferente, la operación depende del valor de R.
Si es pequeño (por nuestra consideracion anteriór $R \leq 3$), se escribe el carácter grabado SC en el archivo comprimido (R veces), luego se actualiza SC con el último carácter leído, y se reinicia el

contador ($R := 0$), para seguir con el siguiente carácter de la cadena.
De lo contrario, se escribe el indicador de cambio de código @, seguido por el valor de R y el carácter SC y a continuación se pone a cero R, y se guarda el siguiente elemento del flujo de datos a comprimir en SC. Todo este ciclo se repite hasta procesar todos los datos.
Así mismmo, el proceso de descompresión también es sencillo: Se explora la secuencia de entrada hasta encontrar el carácter de cambio de código @ (todos los caracteres, hasta la aparición de dicho símbolo, se escriben en la salida); inmediatamente después, se lee el contador de repeticiones n y el carácter a repetir, escribiéndolo en la cadena de salida n veces; este proceso se repite hasta que no haya más datos en la entrada. El diagrama de flujo de este proceso es:

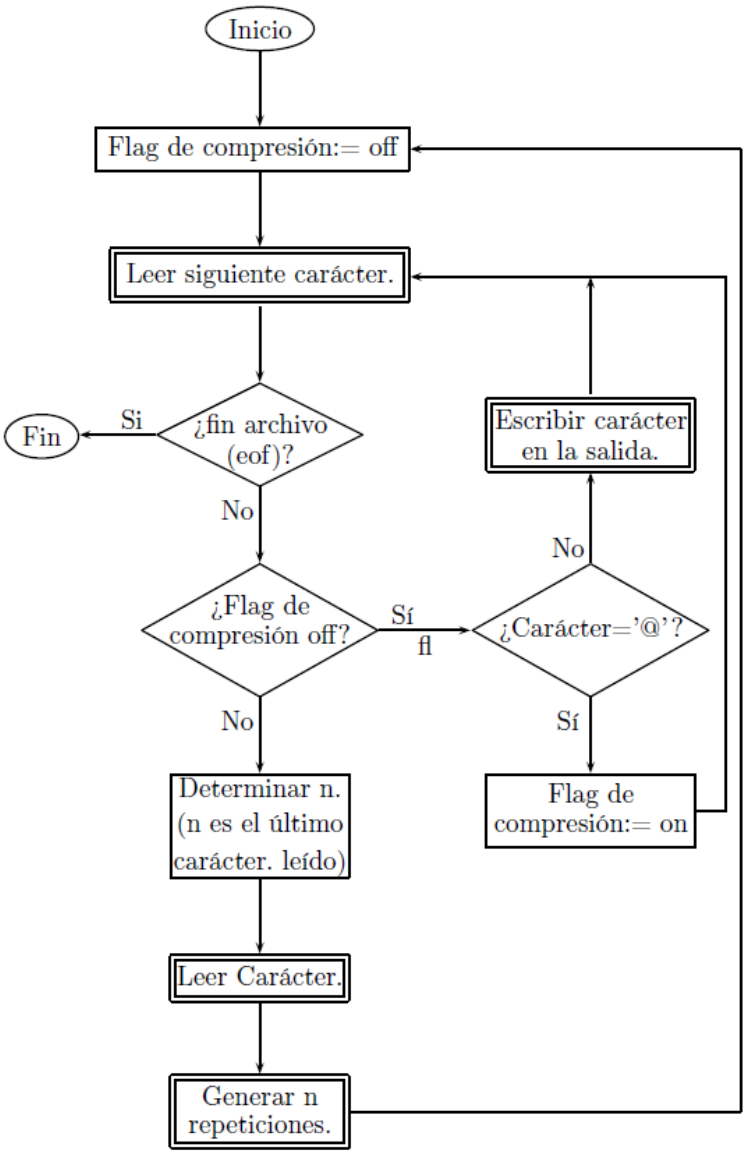

Fig. 2: *RLE: Proceso de descompresión*

Ahora bien para tener una idea de las *razones compresión* producidas por **RLE**, consideramos una cadena de N caracteres, que necesita ser comprimida. Suponemos que la cadena contiene M repeticiones, con una longitud media de L elementos cada una. Cada una de las M repeticiones, se

sustituye por 3 caracteres (el cambio de código, el contador y el dato), por lo que el tamaño de la cadena comprimida es $N - M \cdot L + M \cdot 3 = N - M(L-3)$, y el factor de compresión es:

$$\frac{N}{N - M(L-3)} \tag{1}$$

Ejemplos del factor de compresión son por ejemplo para una cadena de $N = 1000$ caracteres,con $M = 10$ repeticiones, y con una longitud media de $L = 3$, producen un factor de compresión de

$$\frac{1000}{1000 - 10(4-3)} = 1{,}01$$

Un mejor resultado se obtiene en el caso $N = 1000$, $M = 50$, $L = 10$, donde el factor es de:

$$\frac{1000}{1000 - 50(10-3)} = 1,538$$

En conclusión, podemos decir que el método de *compresión RLE (Run-Length Encoding)* es una técnica simple y eficiente para reducir la cantidad de información necesaria para representar secuencias repetitivas de datos. Es especialmente efectivo en datos que contienen patrones repetitivos o secuencias largas de elementos idénticos, sin embargo, tiene sus limitaciones debido a que no es tan eficiente en datos que no presentan repeticiones, e incluso puede aumentar el tamaño de los datos en algunos casos si no se tiene cuidado.
En resumen, la compresión RLE es una herramienta útil en situaciones específicas, pero no es la solución ideal para todos los tipos de datos. Su simplicidad y eficiencia la hacen adecuada en ciertos escenarios, pero cuando se busca una compresión más potente, es necesario recurrir a otros algoritmos y técnicas más sofisticadas.

2.1. Implementación de la compresión RLE para una cadena de texto.

A continuación se coloca una implementación del código en *lenguaje C* para comprimir una cadena de texto por medio del método RLE:

```
\\Compresor de texto por RLE

#include <stdio.h>
#include <string.h>

void comprimirRLE(char* cadena);

int main(int argc, char *argv[]) {
    char cadena[100];

    printf("Ingresa una cadena de caracteres: ");
    fgets(cadena, sizeof(cadena), stdin);

    // Eliminar el salto de linea al final de la cadena
    cadena[strcspn(cadena, "\n")] = '\0';

    printf("Cadena comprimida: ");
    comprimirRLE(cadena);
    printf("\n");

    return 0;
}
```

```c
void comprimirRLE(char* cadena) {
    int longitud = strlen(cadena);

    for (int i = 0; i < longitud; i++) {
        int contador = 1;

        while (i < longitud - 1 && cadena[i] == cadena[i + 1]) {
            contador++;
            i++;
        }

        if (contador >= 3) {
            printf("@%c%d", cadena[i], contador);
        } else {
            for (int j = 0; j < contador; j++) {
                printf("%c", cadena[i]);
            }
        }
    }
}
```

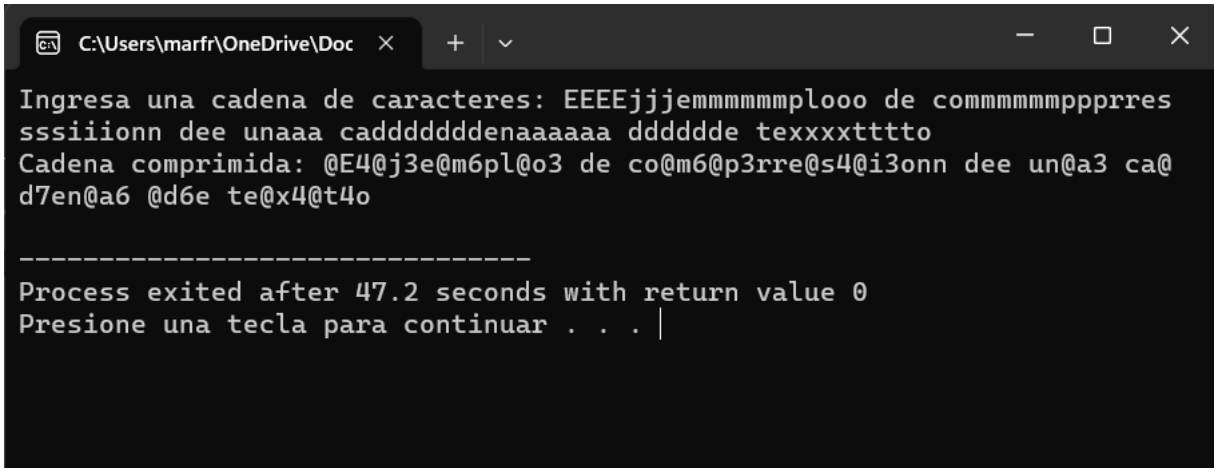

Fig. 3: *Implementación en C del método de compresión de datos RLE*

Así mismo se coloca una implementación del código en *lenguaje C* para descomprimir una cadena de texto comprimida, por medio del método RLE:

```c
\\Descompresor de texto por RLE

#include <stdio.h>
#include <string.h>

void descomprimirRLE(char* cadena);

int main(int argc, char *argv[]) {
    char cadena[100];

    printf("Ingresa la cadena comprimida: ");
    fgets(cadena, sizeof(cadena), stdin);

    // Eliminar el salto de linea al final de la cadena
```

```
    cadena[strcspn(cadena, "\n")] = '\0';

    printf("Cadena descomprimida: ");
    descomprimirRLE(cadena);
    printf("\n");

    return 0;
}

void descomprimirRLE(char* cadena) {
    int longitud = strlen(cadena);

    for (int i = 0; i < longitud; i++) {
        if (cadena[i] == '@') {
            char caracter = cadena[i + 1];
            int repeticiones = cadena[i + 2] - '0';

            for (int j = 0; j < repeticiones; j++) {
                printf("%c", caracter);
            }

            i += 2;
        } else {
            printf("%c", cadena[i]);
        }
    }
}
```

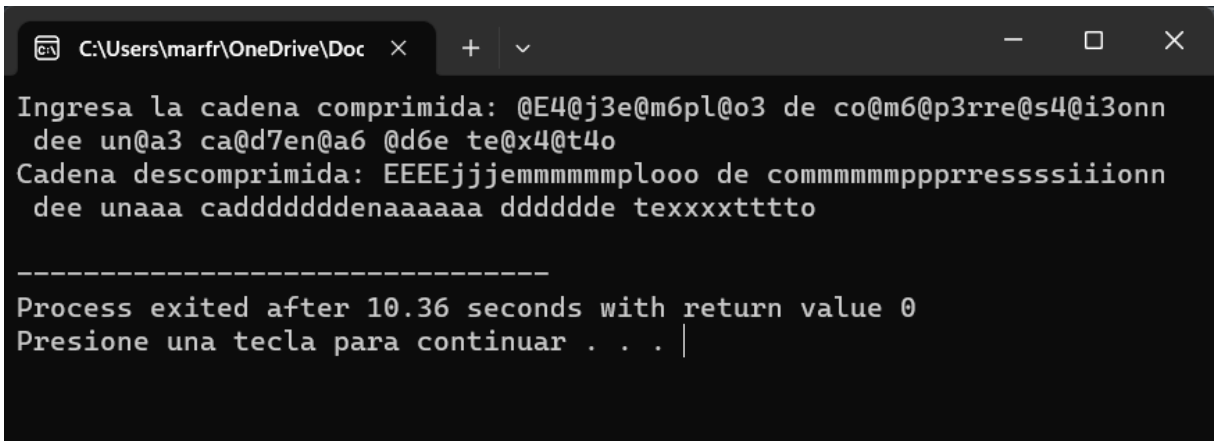

Fig. 4: *Implementación en C del método de descompresión de datos RLE*

A continuación se coloca una implementación del código en *lenguaje Java* para comprimir una cadena de texto por medio del método RLE:

```
\\Compresor de texto por RLE

package rle;

import java.util.Scanner;

public class compresorrle {

    public static void comprimirRLE(String cadena) {
```

```
        int longitud = cadena.length();

        for (int i = 0; i < longitud; i++) {
            int contador = 1;

            while (i < longitud - 1 && cadena.charAt(i) == cadena.charAt(i + 1)) {
                contador++;
                i++;
            }

            if (contador >= 3) {
                System.out.printf("@%c%d", cadena.charAt(i), contador);
            } else {
                for (int j = 0; j < contador; j++) {
                    System.out.print(cadena.charAt(i));
                }
            }
        }
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Ingresa una cadena de caracteres: ");
        String cadena = scanner.nextLine();

        System.out.print("Cadena comprimida: ");
        comprimirRLE(cadena);
        System.out.println();
    }
}
```

```
Output - rle (run)
run:
Ingresa una cadena de caracteres: EEEEjjeeeemmpppplllllo deeee compppppreeeesooorrr deeee texttttttooooo
Cadena comprimida: @E4jj@e4mm@p4@l5o d@e4 com@p4r@e4s@o3@r3 d@e4 tex@t6@o5
BUILD SUCCESSFUL (total time: 25 seconds)
```

Fig. 5: *Implementación en Java del método de compresión de datos RLE*

Así mismo se coloca una implementación del código en *lenguaje Java* para descomprimir una cadena de texto comprimida, por medio del método RLE:

```
\\Descompresor de texto por RLE

package rle;

import java.util.Scanner;

public class descomprle {

    public static void descomprimirRLE(String cadena) {
        int longitud = cadena.length();

        for (int i = 0; i < longitud; i++) {
```

```
            if (cadena.charAt(i) == '@') {
                char caracter = cadena.charAt(i + 1);
                int repeticiones = Character.getNumericValue(cadena.charAt(i + 2));

                for (int j = 0; j < repeticiones; j++) {
                    System.out.print(caracter);
                }

                i += 2;
            } else {
                System.out.print(cadena.charAt(i));
            }
        }
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Ingresa la cadena comprimida: ");
        String cadena = scanner.nextLine();

        System.out.print("Cadena descomprimida: ");
        descomprimirRLE(cadena);
        System.out.println();
    }
}
```

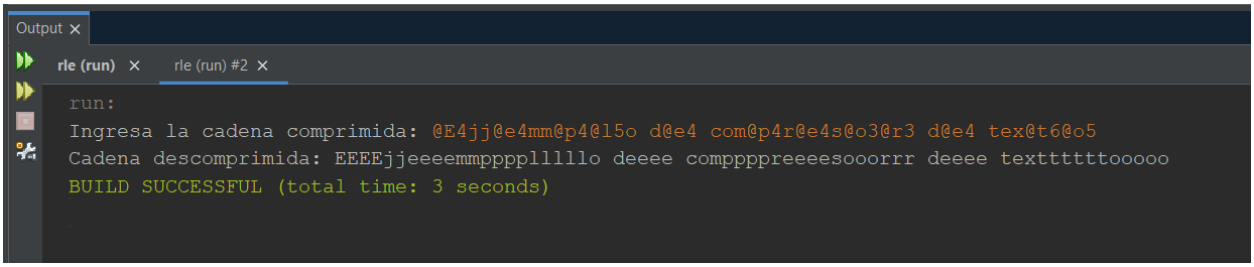

Fig. 6: *Implementación en java del método de descompresión de datos RLE*

3. Compresión Huffmann

3.1. Los métodos estadísticos.

Cuando hablamos de los **métodos estadísticos de compresión** nos referimos a *aquellos que usan códigos de tamaño variable, asignando los más cortos a los símbolos o grupos de símbolos que aparecen con más frecuencia en los datos*, es decir, los que tienen una mayor probabilidad de ocurrencia.
Sin embargo los diseñadores y los implementadores de códigos de tamaño variable tienen que tratar dos problemas:

1. de asignación de códigos que puedan ser decodificados sin ambiguedad
2. de asignación de códigos con el tamaño medio mínimo.

Un ejemplo de uno de estos métodos lo dio *Samuel Morse*, quien utilizó códigos de tamaño variable cuando diseñó su famoso código para el telégrafo (Tabla 2.1). Como dato curioso, es interesante mencionar que la primera versión de su código, desarrollado por Morse durante un viaje transatlántico en 1832, era más compleja que la versión que él estableció en en 1843.

A	.-	N	-.	1	.----	Period [.]	.-.-.-
B	-...	O	---	2	..---	Coma [,]	--..--
C	-.-.	P	.--.	3	...--	Colon [:]	---...
Ch	----	Q	--.-	4	....-	Question mark [?]	..--..
D	-..	R	.-.	5	.....	Apostrophe [']	.----.
E	.	S	...	6	-....	Hyphen [-]	-....-
F	..-.	T	-	7	--...	Dash [–,−,—]	-..-.
G	--.	U	..-	8	---..	Parentheses [( )]	-.--.-
H	....	V	...-	9	----.	Quotation marks ["]	.-..-.
I	..	W	.--	0	-----		
J	.---	X	-..-				
K	-.-	Y	-.--				
L	.-..	Z	--..				
M	--						

Si se toma como unidad la duración de un punto , entonces un guión consta de tres unidades. El espacio entre los puntos y las rayas de un carácter, una unidad; entre caracteres, es de tres unidades; y entre las palabras, seis unidades (cinco para la transmisión automática). Para indicar que se ha producido un error y para que el receptor elimine la última palabra, envíense "....." (ocho puntos).

Fig. 7: *Código morse en su versión inglesa*

En la primera versión se enviaban guiones cortos y largos que eran recibidos y dibujados en una tira de papel, donde las secuencias de los guiones representaban los números. A cada palabra (no a cada letra) se le asignó un código numérico y Morse construyó un libro o diccionario de dichos códigos en 1837.
Esta primera versión fue, por lo tanto, una forma primitiva de compresión. Posteriormente,Morse abandonó esta versión en favor de la más famosa de *puntos y rayas*, desarrollada junto con **Alfred Vail**. En nuestro caso solo indagaremos en uno de los tanto métodos estadísticos,y uno de los mas importantes: el **método Huffmann**.

3.2. Codificación Huffmann

Eñ algoritmo de Huffman, fue desarrollado por *David A. Huffman* y publicado en "*A method for the construction of minimun redundancy codes*", en 1952. Este algoritmo permite comprimir información basándose en una idea simple: utilizar menos de 8 bits para representar a cada uno de los bytes de la fuente de información original.
Por ejemplo, si utilizamos los bits 01 (dos bits) para representar el caracter 'A' (o el byte 01000001), cada vez que aparezca una 'A' estaremos ahorrando 6 bits. Entonces cada 4 caracteres 'A' que aparezcan en el archivo ahorraremos 24 bits, es decir, 3 bytes. Luego, cuanto mayor sea la cantidad de caracteres 'A' que contenga el archivo o la fuente de información original, mayor será el grado de compresión que podremos lograr. Así, a la *secuencia de bits que utilizamos para recodificar cada carácter* la llamamos "**código Huffman**".

El algoritmo consiste en una serie de pasos a través de los cuales podremos construir los *códigos Huffman* para reemplazar los bytes de la fuente de información que queremos comprimir, generando **códigos más cortos para los caracteres más frecuentes y códigos más largos para los bytes que menos veces aparecen**.
El *algoritmo de Huffman* provee un método que permite comprimir información mediante la recodificación de los bytes que la componen. En particular, si los bytes que se van a comprimir están almacenados en un archivo, al recodificarlos con secuencias de bits más cortas diremos que lo comprimimos.
La técnica consiste en asignar a cada byte del archivo que vamos a comprimir un código binario compuesto por una cantidad de bits tan corta como sea posible. Esta cantidad será variable y dependerá de la probabilidad de ocurrencia del byte, es decir: *aquellos bytes que más veces aparecen serán recodificados con combinaciones de bits más cortas, de menos de 8 bits.* En cambio, se utilizarán combinaciones de bits más extensas para recodificar los bytes que menos veces se repiten dentro del archivo. Estas combinaciones podrían, incluso, tener más de 8 bits.
Los códigos binarios que utilizaremos para reemplazar a cada byte del archivo original se llaman **códigos Huffman**.
Veamos un ejemplo. En el siguiente texto:

COMO COME COCORITO COME COMO COSMONAUTA

el carácter 'O' aparece 11 veces y el carácter 'C' aparece 7 veces. Estos son los caracteres que más veces aparecen y por lo tanto tienen la mayor probabilidad de ocurrencia. En cambio, los caracteres 'I', 'N', 'R', 'S' y 'U' aparecen una única vez; esto significa que la probabilidad de hallar en el archivo alguno de estos caracteres es muy baja.
Como ya sabemos, para codificar cualquier carácter se necesitan 8 bits (1 byte). Sin embargo, supongamos que logramos encontrar una combinación única de 2 bits con la cual codificar al carácter 'O', una combinación única de 3 bits con la cual codificar al carácter 'M' y otra combinación única de 3 bits con la cual codificar al carácter 'C':

Carácter	Codificación
O	01
M	001
C	000

Si esto fuera así, entonces para codificar los primeros 3 caracteres del texto anterior solo necesitaríamos 1 byte, lo que nos daría una tasa de compresión del 66,6 %.

Carácter	C	O	M	...
Byte	000	01	001	...

Ahora, el byte 00001001 representa la secuencia de caracteres 'C', 'O', 'M', pero esta información solo podrá ser interpretada si conocemos los códigos binarios que utilizamos para recodifidar los bytes originales. De lo contrario, la información no se podrá recuperar.
Para obtener estas combinaciones de bits únicas, el algoritmo de Huffman propone seguir una serie de pasos a través de los cuales obtendremos un *árbol binario* llamado "**árbol Huffman**".
Luego, las hojas del árbol representarán a los diferentes caracteres que aparecen en el archivo y los caminos que se deben recorrer para llegar a esas hojas representarán la nueva codificación del carácter.

A continuación, analizaremos los pasos necesarios para obtener el árbol y los códigos Huffman que corresponden a cada uno de los caracteres del texto que usamos como ejemplo.

- **Contar la cantidad de ocurrencias de cada carácter.** El primer paso consiste en contar cuántas veces aparece en el archivo cada carácter o byte. Como un byte es un conjunto de 8 bits, resulta que solo existen 28 = 256 bytes diferentes. Entonces utilizaremos una tabla con 256 registros para representar a cada uno de los 256 bytes y sus correspondientes contadores de ocurrencias.

	Carácter	*n*
0		
:		
32	*ESP*	5
:		
65	A	2
:		
67	C	7
:		
69	E	2
:		
73	I	1

	Carácter	*n*
:		
:		
77	M	5
78	N	1
79	O	11
:		
82	R	1
83	S	1
84	T	2
85	U	1
:		
255		

Fig. 8: *Tabla de ocurrencias que indica la cantidad de veces que aparece cada carácter*

- **Crar una lista enlazada.** Conociendo la cantidad de ocurrencias de cada carácter, tenemos que crear una lista enlazada y ordenada de forma ascendente por dicha cantidad. Primero los caracteres menos frecuentes y luego los que tienen mayor probabilidad de aparecer y, si dos caracteres ocurren igual cantidad de veces, entonces colocaremos primero al que tenga menor valor numérico. Por ejemplo: los caracteres 'I', 'N', 'R', 'S' y 'U' aparecen una sola vez y tienen la misma probabilidad de ocurrencia entre sí, por lo tanto, en la lista ordenada que veremos a continuación los colocaremos de forma ascendente según su valor numérico o código **ASCII**.

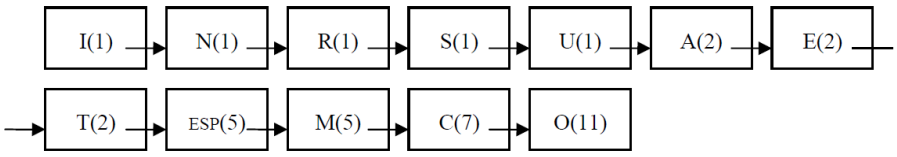

Fig. 9: *Lista de caracteres ordenada por probabilidad de ocurrencia*

Los nodos de la lista tendrán la siguiente estructura:

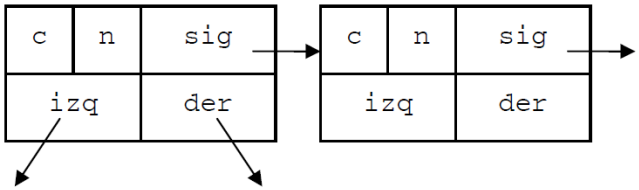

Fig. 10: *Estructura del nodo de la lista enlazada*

El campo c representa el carácter (o byte) y el campo n la cantidad de veces que aparece dentro del archivo. El campo *sig* es la referencia al siguiente elemento de la lista enlazada. Los campos *izq* y *der*, más adelante, nos permitirán implementar el árbol binario. Por el momento no les daremos importancia; simplemente pensemos que son punteros a $NULL$.

- **Convertir la lista enlazada en el árbol de Huffmann.** Vamos a generar el árbol Huffman tomando "de a pares" los nodos de la lista. Esto lo haremos de la siguiente manera: sacamos los dos primeros nodos y los utilizamos para crear un pequeño árbol binario cuya raíz será un nuevo nodo que identificaremos con un carácter ficticio $*1$ (léase "asterisco uno") y una cantidad de ocurrencias igual a la suma de las cantidades de los dos nodos que estamos procesando. En la rama derecha colocamos al nodo menos ocurrente (el primero); el otro nodo lo colocaremos en la rama izquierda.

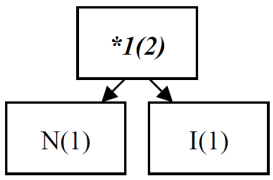

Fig. 11: *Proceso de los dos primeros nodos de la lista*

Luego insertamos en la lista al nuevo nodo (raíz) respetando el criterio de ordenamiento que mencionamos más arriba. Si en la lista existe un nodo con la misma cantidad de ocurrencias (que en este caso es 2), la inserción la haremos a continuación de este.

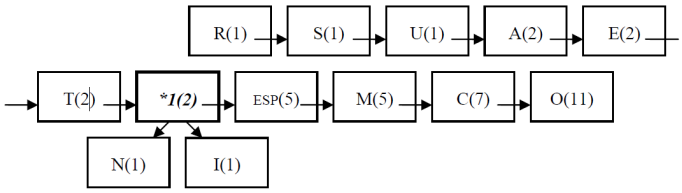

Fig. 12: *Estado de la lista una vez finalizado el proceso de los dos primeros nodos*

Ahora repetimos la operación procesando nuevamente los dos primeros nodos de la lista: $R(1)$ y $S(1)$:

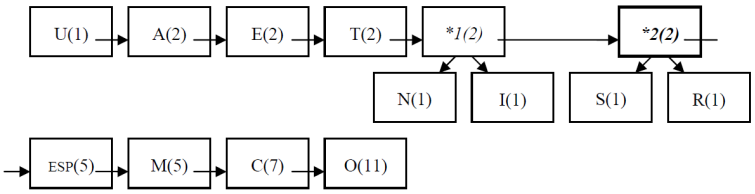

Fig. 13: *Estado de la lista una vez finalizado el proceso de los dos primeros nodos*

Luego continuamos con este proceso hasta que la lista se haya convertido en un árbol binario cuyo nodo raíz tenga una cantidad de ocurrencias igual al tamaño del archivo queremos comprimir.

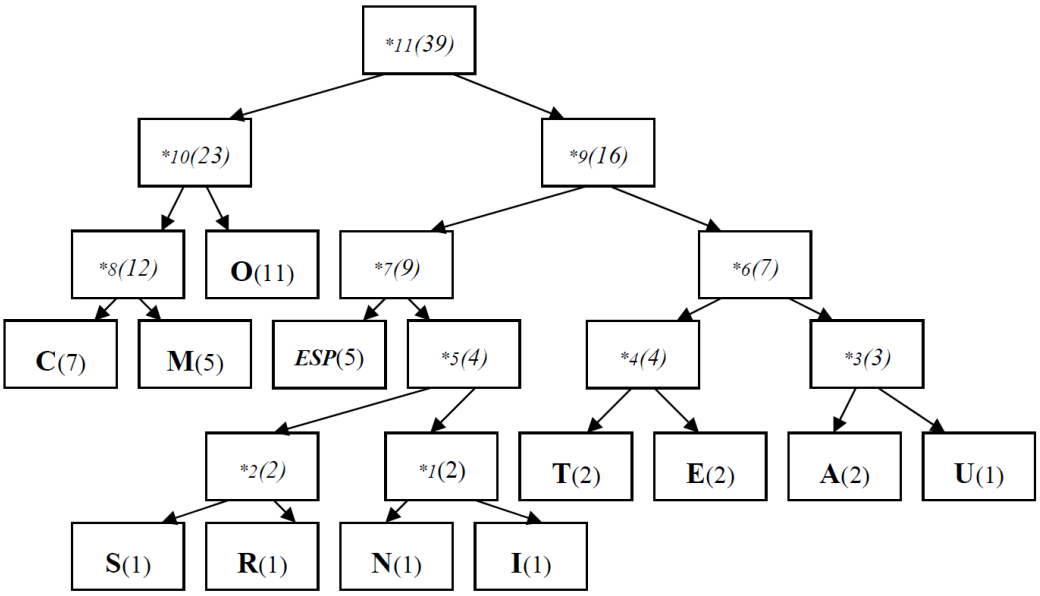

Fig. 14: *Árbol de Huffmann que obtenemos despues de procesar todos los nodos de la lista*

Observemos que el árbol resultante tiene caracteres fi cticios en los vértices y caracteres reales en las hojas

- **Asignación de códigos Huffmann.** El siguiente paso será asignar un *código Huffman* a cada uno de los caracteres reales que se encuentran ubicados en las hojas del árbol. Para esto, consideraremos el camino que se debe recorrer para llegar a cada hoja. El código se forma concatenando un 0 (cero) por cada tramo que avanzamos hacia la izquierda y un 1 (uno) cada vez que avanzamos hacia la derecha, por lo tanto, el código Huffman que le corresponde al carácter 'O' es 01, el código que le corresponde al carácter 'M' es 001 y el código que le corresponde al carácter 'S' es 10100.
  Como podemos ver, la longitud del código que el árbol le asigna a cada carácter es inversamente proporcional a su cantidad de ocurrencias. Los caracteres más frecuentes reciben códigos más cortos mientras que los menos frecuentes reciben códigos más largos. Agreguemos los códigos Huffman (cod) y sus longitudes (nCod) en la tabla de ocurrencias.

	Carácter	n	cod	nCod
0				
:				
32	ESP	5	100	3
:				
65	A	2	1110	4
:				
67	C	7	000	3
:				
69	E	2	1101	4
:				
73	I	1	10111	5

:	Carácter	n	cod	nCod
:				
77	M	5	001	3
78	N	1	10110	5
79	O	11	01	2
:				
82	R	1	10101	5
83	S	1	10100	5
84	T	2	1100	4
85	U	1	11111	5
:				
255				

Fig. 15: *Tabla de ocurrencias completa*

El lector podrá pensar que, en el peor de los casos, el código Huffman más largo que se asignará a un carácter será de 8 bits porque esta es la cantidad original con la que el carácter está codificado, pero esto no es así.
La longitud máxima que puede alcanzar un código Huffman dependerá de la cantidad de símbolos que componen el alfabeto con el que esté codificada la información, de modo que, si n es la cantidad de símbolos que componen un alfabeto, entonces, en el peor de los casos, el código Huffman que se asignará a un carácter podrá tener hasta $n/2$ bits.
En nuestro caso el alfabeto está compuesto por cada una de las 256 combinaciones que admite un byte; por lo tanto, tenemos que estar preparados para trabajar códigos de, a lo sumo, $\frac{256}{2} = 128$ bits.
Obviamente, el hecho de asignar códigos de menos de 8 bits a los caracteres más frecuentes compensa la posibilidad de que, llegado el caso, se utilicen más de 8 bits para codificar los caracteres que menos veces aparecen.

- **Codificación del contenido.** Para finalizar, generamos el archivo comprimido reemplazando cada carácter del archivo original por su correspondiente código Huffman, agrupando los diferentes códigos en paquetes de 8 bits ya que esta es la menor cantidad de información que podemos manipular.

C	O	M	O	*[esp]*	C	...
000	01	001	01	100	000	...
00001001			01100000			...

Fig. 16: *Compresión de los primeros bytes del texto original*

- **Decodificación y descompresión** Para descomprimir un archivo necesitamos disponer del árbol Huffman utilizado para su codificación. Sin el árbol la información no se podrá recuperar. Por este motivo el algoritmo de Huffman también puede utilizarse como algoritmo de encriptación.
Supongamos que, de alguna manera, podemos rearmar el árbol Huffman, entonces el algoritmo para descomprimir y restaurar el archivo original es el siguiente:

  1. Rearmar el árbol Huffman y posicionarnos en la raíz.
  2. Recorrer "bit por bit", el archivo comprimido. Si leemos un 0 descendemos un nivel del árbol colocándonos en su hijo izquierdo. En cambio, si leemos un 1 descendemos un nivel para posicionarnos en su hijo derecho.
  3. Repetimos el paso 2 hasta llegar a una hoja. Esto nos dará la pauta de que hemos decodificado un carácter y lo podremos grabar en el archivo que estamos restaurando.

A continuación se coloca una implementación del código en *lenguaje C* para comprimir y descomprimir una cadena de texto, por medio del método de compresión de HUffmann:

```
\\Programa que implementa la compresion y descompresion de una cadena de texto por
    metodo de Huffman

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_TREE_NODES 256
```

```c
// Estructura para representar un nodo del arbol de Huffman
typedef struct HuffmanNode {
    char character;
    int frequency;
    struct HuffmanNode* left;
    struct HuffmanNode* right;
} HuffmanNode;

// Estructura para representar una tabla de codigos de Huffman
typedef struct HuffmanTable {
    char character;
    char* code;
} HuffmanTable;

HuffmanNode* createNode(char character, int frequency, HuffmanNode* left, HuffmanNode
    * right);
void swapNodes(HuffmanNode** a, HuffmanNode** b);
void sortNodes(HuffmanNode** nodes, int n);
HuffmanNode* buildHuffmanTree(char* text);
void freeHuffmanTree(HuffmanNode* node);
void buildHuffmanTableRecursive(HuffmanNode* node, HuffmanTable* table, char* code,
    int depth);
HuffmanTable* buildHuffmanTable(HuffmanNode* root);
char* compressText(char* text, HuffmanTable* table);
char* decompressText(char* compressedText, HuffmanNode* root, int compressedLength);

int main(int argc, char *argv[]) {
    char text[1000];
    printf("Ingrese el texto a comprimir: ");
    fgets(text, sizeof(text), stdin);

    // Eliminar el caracter de nueva linea al final del texto ingresado por fgets
    text[strcspn(text, "\n")] = '\0';

    HuffmanNode* root = buildHuffmanTree(text);
    HuffmanTable* table = buildHuffmanTable(root);

    printf("Codigos de Huffman:\n");
    int i;
    for (i = 0; i < MAX_TREE_NODES; i++) {
        if (table[i].character != 0) {
            printf("%c: %s\n", table[i].character, table[i].code);
        }
    }

    char* compressedText = compressText(text, table);
    printf("Texto comprimido: %s\n", compressedText);

    char* decompressedText = decompressText(compressedText, root, strlen(
        compressedText));
    printf("Texto descomprimido: %s\n", decompressedText);

    // Liberar la memoria utilizada
    freeHuffmanTree(root);
    free(table);
    free(compressedText);
    free(decompressedText);
```

```
    return 0;
}

// Funcion para crear un nuevo nodo del arbol de Huffman
HuffmanNode* createNode(char character, int frequency, HuffmanNode* left, HuffmanNode
    * right) {
    HuffmanNode* node = (HuffmanNode*)malloc(sizeof(HuffmanNode));
    node->character = character;
    node->frequency = frequency;
    node->left = left;
    node->right = right;
    return node;
}

// Funcion para intercambiar dos nodos del arbol de Huffman
void swapNodes(HuffmanNode** a, HuffmanNode** b) {
    HuffmanNode* temp = *a;
    *a = *b;
    *b = temp;
}

// Funcion para ordenar los nodos del arbol de Huffman en orden ascendente de
    frecuencia
void sortNodes(HuffmanNode** nodes, int n) {
    int i, j;
    for (i = 0; i < n - 1; i++) {
        for (j = 0; j < n - i - 1; j++) {
            if (nodes[j]->frequency > nodes[j + 1]->frequency) {
                swapNodes(&nodes[j], &nodes[j + 1]);
            }
        }
    }
}

// Funcion para construir el arbol de Huffman
HuffmanNode* buildHuffmanTree(char* text) {
    int frequencies[MAX_TREE_NODES] = {0};
    int i, n = strlen(text);

    // Calcular la frecuencia de cada caracter
    for (i = 0; i < n; i++) {
        frequencies[text[i]]++;
    }

    // Crear nodos iniciales para cada caracter no nulo
    HuffmanNode* nodes[MAX_TREE_NODES];
    int nodeCount = 0;
    for (i = 0; i < MAX_TREE_NODES; i++) {
        if (frequencies[i] > 0) {
            nodes[nodeCount] = createNode((char)i, frequencies[i], NULL, NULL);
            nodeCount++;
        }
    }

    // Construir el arbol de Huffman combinando los nodos
    while (nodeCount > 1) {
        sortNodes(nodes, nodeCount);

        HuffmanNode* left = nodes[0];
```

```
        HuffmanNode* right = nodes[1];
        HuffmanNode* parent = createNode('\0', left->frequency + right->frequency,
            left, right);

        nodes[0] = parent;
        nodes[1] = NULL;
        nodeCount--;

        int j;
        for (j = 1; j < nodeCount; j++) {
            nodes[j] = nodes[j + 1];
        }
    }

    return nodes[0];
}

// Funcion auxiliar para liberar la memoria del arbol de Huffman
void freeHuffmanTree(HuffmanNode* node) {
    if (node != NULL) {
        freeHuffmanTree(node->left);
        freeHuffmanTree(node->right);
        free(node);
    }
}

// Funcion para construir la tabla de codigos de Huffman recursivamente
void buildHuffmanTableRecursive(HuffmanNode* node, HuffmanTable* table, char* code,
    int depth) {
    if (node->left == NULL && node->right == NULL) {
        table[node->character].character = node->character;
        table[node->character].code = (char*)malloc((depth + 1) * sizeof(char));
        strcpy(table[node->character].code, code);
        return;
    }

    int codeLength = strlen(code);
    char* leftCode = (char*)malloc((codeLength + 2) * sizeof(char));
    char* rightCode = (char*)malloc((codeLength + 2) * sizeof(char));

    strcpy(leftCode, code);
    leftCode[codeLength] = '0';
    leftCode[codeLength + 1] = '\0';

    strcpy(rightCode, code);
    rightCode[codeLength] = '1';
    rightCode[codeLength + 1] = '\0';

    buildHuffmanTableRecursive(node->left, table, leftCode, depth + 1);
    buildHuffmanTableRecursive(node->right, table, rightCode, depth + 1);

    free(leftCode);
    free(rightCode);
}

// Funcion para construir la tabla de codigos de Huffman
HuffmanTable* buildHuffmanTable(HuffmanNode* root) {
    HuffmanTable* table = (HuffmanTable*)malloc(MAX_TREE_NODES * sizeof(HuffmanTable)
        );
```

```
    memset(table, 0, MAX_TREE_NODES * sizeof(HuffmanTable));
    char code[1] = {'\0'};
    buildHuffmanTableRecursive(root, table, code, 0);
    return table;
}

// Funcion para comprimir el texto utilizando la tabla de codigos de Huffman
char* compressText(char* text, HuffmanTable* table) {
    int textLength = strlen(text);
    int compressedLength = 0;
    int i, j;

    // Calcular la longitud total del texto comprimido
    for (i = 0; i < textLength; i++) {
        compressedLength += strlen(table[text[i]].code);
    }

    // Crear una cadena para almacenar el texto comprimido
    char* compressedText = (char*)malloc((compressedLength + 1) * sizeof(char));
    compressedText[compressedLength] = '\0';

    // Copiar los codigos de Huffman en la cadena del texto comprimido
    int currentIndex = 0;
    for (i = 0; i < textLength; i++) {
        char* code = table[text[i]].code;
        int codeLength = strlen(code);
        for (j = 0; j < codeLength; j++) {
            compressedText[currentIndex] = code[j];
            currentIndex++;
        }
    }

    return compressedText;
}

// Funcion para descomprimir el texto comprimido utilizando la tabla de codigos de
    Huffman
char* decompressText(char* compressedText, HuffmanNode* root, int compressedLength) {
    char* decompressedText = (char*)malloc((compressedLength + 1) * sizeof(char));
    decompressedText[compressedLength] = '\0';

    HuffmanNode* current = root;
    int currentIndex = 0;
    int i;

    for (i = 0; i < compressedLength; i++) {
        if (compressedText[i] == '0') {
            current = current->left;
        } else if (compressedText[i] == '1') {
            current = current->right;
        }

        if (current->left == NULL && current->right == NULL) {
            decompressedText[currentIndex] = current->character;
            currentIndex++;
            current = root;
        }
    }
```

```
    return decompressedText;
}
```

Fig. 17: *Implementación en C del método de Huffmann para la compresión y descompresión de datos*

Referencias

[1] Cairó O, y Buemo, S. G. *Estructura de datos*, 2006, McGraw-HIll Companies.

[2] Noel Kalicharan, *Advanced topics in C*, 2013, Springer.

[3] Salomon D., *Data Compression: The Complete Reference*, 2007, Springer.

[4] Sznajdleder, P. A., *Algoritmos a fondo: con implementaciones en C y Java.*, 2012, Alfaomega.

[5] Miguel Morales S., *Notas sobre Compresión de datos*, 2003, INAOE.