Proyecto compresión de archivos por Huffman

Alumna: Brisa Jael Lima Arenas.

Matricula: 200300606.

Carrera: Ingeniería en Datos e Inteligencia Organizacional.

Asignatura: Teoría de la Información.

Profesor: Ismael Domínguez Jiménez.

Fecha de entrega: 29 de Noviembre.

Estructura del Programa

El programa se diseñó para comprimir archivos utilizando los algoritmos trabajados en tareas previas, como ordenamiento, árboles binarios y un vector para la reducción de probabilidades. El objetivo principal es crear una versión funcional que permite comprimir y descomprimir archivos en formatos de texto, audio y video, además de generar métricas clave como el factor de compresión, tamaño original y tamaño comprimido, sin embargo por ciertos bloqueos en el proceso, me tuve que enfocar únicamente en en el archivo de texto.

2. Entrada y Salida:

- **Entrada:** Archivos en diferentes formatos (texto, audio, video).
- **Salida:** Archivo comprimido, archivo descomprimido, reporte de métricas.

Código:

- **Librerias:**
  - #include <iostream>
  - #include <vector>
  - #include <fstream>
  - #include <sstream>
  - #include <unordered_map>
  - #include <queue>
  - #include <bitset>
  - using namespace std;

- **Estructura nodo:**

Esta estructura representa cada nodo del árbol de Huffman.

```
struct nodo {
    float prob;
    char letra;
    nodo *izq;
    nodo *der;

    nodo(char c, float p) : letra(c), prob(p), izq(nullptr), der(nullptr) {}
};
```

- prob: Es la probabilidad o frecuencia con la que aparece un carácter en el archivo.
- letra: El carácter asociado a ese nodo.
- izq y der: Son punteros a los nodos hijos izquierdo y derecho en el árbol de Huffman.

- **Estructura Comparador:**

Es utilizada por la priority_queue para ordenar los nodos en base a su probabilidad.

```
struct Comparador {
    bool operator()(nodo* a, nodo* b) {
        return a->prob > b->prob;
    }
};
```

Esta estructura se usa para asegurar que la priority_queue extraiga primero el nodo con menor probabilidad, que es esencial en la construcción del árbol de Huffman.

- **Funciones Principales del Programa:**

```
int menu();
void mostrarArbolHuffman(nodo* raiz, int espacio = 0);
nodo* construirArbolHuffman(vector<nodo *> &vec);
void insertarNodo(vector<nodo *> &nodos);
void leerArchivoYConstruirHuffman(const string &nombreArchivo, nodo*&
raiz);
void generarTablaCodigos(nodo* raiz, unordered_map<char, string> &tabla,
string codigo = "");
void comprimirArchivo(const string &archivoOriginal, const string
&archivoComprimido, nodo* raiz);
void descomprimirArchivo(const string &archivoComprimido, const string
&archivoRecuperado, nodo* raiz);
void calcularMetricas(const string &archivoOriginal, const string
&archivoComprimido);
```

- int menu();

Muestra un menú interactivo para que el usuario seleccione entre las diferentes opciones.

- void mostrarArbolHuffman(nodo* raiz, int espacio = 0);

Muestra el árbol de Huffman en forma visual (por ejemplo, en la consola). La idea es imprimir la estructura del árbol con sangrías para que sea más fácil de entender.

- nodo* construirArbolHuffman(vector<nodo *> &vec);

Construye el árbol de Huffman a partir de un vector de nodos que contienen los caracteres y sus frecuencias.

- void insertarNodo(vector<nodo *> &nodos);

Permite insertar un nodo (con carácter y frecuencia) en el vector nodos. Esto es útil para construir o modificar el árbol de Huffman.

- void leerArchivoYConstruirHuffman(const string &nombreArchivo, nodo*& raiz);

Lee un archivo de texto, cuenta las frecuencias de los caracteres y construye el árbol de Huffman basándose en esas frecuencias.

- void generarTablaCodigos(nodo* raiz, unordered_map<char, string> &tabla, string codigo = "");

Genera la tabla de códigos binarios para cada carácter a partir del árbol de Huffman. Cada carácter se asocia con una cadena de bits (código de Huffman).

- void comprimirArchivo(const string &archivoOriginal, const string &archivoComprimido, nodo* raiz);

Comprime el archivo original utilizando el árbol de Huffman y genera un archivo comprimido.

- void descomprimirArchivo(const string &archivoComprimido, const string &archivoRecuperado, nodo* raiz);

Descomprime el archivo utilizando el árbol de Huffman, devolviendo el archivo recuperado.

- void calcularMetricas(const string &archivoOriginal, const string &archivoComprimido);

Calcula métricas de la compresión, como la reducción del tamaño del archivo (porcentaje de compresión) entre el archivo original y el comprimido.

```
int main() {
    int op;
    vector<nodo *> nodos;
    nodo* raiz = nullptr;

    do {
        op = menu();
        switch (op) {
            case 1:
                insertarNodo(nodos);
                break;
            case 2:
                raiz = construirArbolHuffman(nodos);
                cout << "\nArbol de Huffman generado con los nodos insertados:"
<< endl;
                mostrarArbolHuffman(raiz);
                break;
            case 3:
```

```
                leerArchivoYConstruirHuffman("texto.txt", raiz);
                break;
            case 4:
                if (raiz) {
                    comprimirArchivo("texto.txt", "texto_comprimido.huff", raiz);
                } else {
                    cout << "Primero genera un Arbol de Huffman." << endl;
                }
                break;
            case 5:
                if (raiz) {
                    descomprimirArchivo("texto_comprimido.huff",
"texto_recuperado.txt", raiz);
                } else {
                    cout << "Primero genera un Arbol de Huffman." << endl;
                }
                break;
            case 6:
                if (raiz) {
                    calcularMetricas("texto.txt", "texto_comprimido.huff");
                } else {
                    cout << "Primero genera un Arbol de Huffman." << endl;
                }
                break;
            case 7:
                cout << "Saliendo..." << endl;
                break;
            default:
                cout << "Opcion no valida." << endl;
                break;
        }
    } while (op != 7);

    return 0;
}
```

- Caso 1 (insertarNodo(nodos)): Inserta un nuevo nodo en el vector nodos.
- Caso 2 (construirArbolHuffman(nodos)): Construye el árbol de Huffman y lo muestra en consola.
- Caso 3 (leerArchivoYConstruirHuffman("texto.txt", raiz)): Lee un archivo de texto, cuenta las frecuencias y construye el árbol de Huffman.

- Caso 4 (comprimirArchivo("texto.txt", "texto_comprimido.huff", raiz)): Comprime un archivo usando el árbol de Huffman.

- Caso 5 (descomprimirArchivo("texto_comprimido.huff", "texto_recuperado.txt", raiz)): Descomprime un archivo comprimido.

- Caso 6 (calcularMetricas("texto.txt", "texto_comprimido.huff")): Calcula las métricas de compresión.

- Caso 7 (Salir): Termina el programa.

- Caso por defecto: Si el usuario elige una opción no válida, se muestra un mensaje de error.

Para las opciones de compresión, descompresión y cálculo de métricas (casos 4, 5 y 6), el programa verifica si el árbol de Huffman ha sido construido. Si no se ha generado, muestra un mensaje indicando que primero debe generarse el árbol antes de proceder.

- **Menú:**

```
int menu() {
    int opt = 0;
    cout << "\n*** Menu ***" << endl;
    cout << "1. Insertar nodo" << endl;
    cout << "2. Ordenar nodos y mostrar Arbol de Huffman" << endl;
    cout << "3. Leer archivo y construir Arbol de Huffman" << endl;
    cout << "4. Comprimir archivo" << endl;
    cout << "5. Descomprimir archivo" << endl;
    cout << "6. Calcular metricas" << endl;
    cout << "7. Salir" << endl;
    cout << "Selecciona tu opcion: ";
    cin >> opt;
    return opt;
}
```

Al ejecutar el programa, el usuario verá el menú interactivo impreso en la pantalla, con las opciones numeradas, luego el programa espera que el usuario ingrese una opción (un número entre 1 y 7), después de ingresar el número, el valor de la opción se devuelve a la función principal (como un valor entero), y el programa puede tomar la acción correspondiente según la opción seleccionada.

```
void insertarNodo(vector<nodo *> &nodos) {
```

```
    char letra;
    float prob;
    cout << "Ingresa la letra: ";
    cin >> letra;
    cout << "Ingresa la probabilidad: ";
    cin >> prob;
    nodo* nuevoNodo = new nodo(letra, prob);
    nodos.push_back(nuevoNodo);
    cout << "Nodo insertado correctamente." << endl;
}

nodo* construirArbolHuffman(vector<nodo *> &vec) {
    priority_queue<nodo*, vector<nodo*>, Comparador> pq;
    for (nodo* n : vec) {
        pq.push(n);
    }
    while (pq.size() > 1) {
        nodo* izquierdo = pq.top(); pq.pop();
        nodo* derecho = pq.top(); pq.pop();
        nodo* nuevoNodo = new nodo(' ', izquierdo->prob + derecho->prob);
        nuevoNodo->izq = izquierdo;
        nuevoNodo->der = derecho;
        pq.push(nuevoNodo);
    }
    return pq.top();
}

void mostrarArbolHuffman(nodo* raiz, int espacio) {
    if (raiz == nullptr) return;
    espacio += 10;
    mostrarArbolHuffman(raiz->der, espacio);
    cout << endl;
    for (int i = 10; i < espacio; i++) {
        cout << " ";
    }
    cout << raiz->prob << " (" << raiz->letra << ")" << endl;
    mostrarArbolHuffman(raiz->izq, espacio);
}

void leerArchivoYConstruirHuffman(const string &nombreArchivo, nodo*&
raiz) {
    ifstream archivo(nombreArchivo);
    if (!archivo.is_open()) {
        cout << "No se pudo abrir el archivo." << endl;
```

```
        return;
    }
    unordered_map<char, int> freqMap;
    char c;
    while (archivo.get(c)) {
        freqMap[c]++;
    }
    archivo.close();

    vector<nodo *> nodos;
    for (const auto &pair : freqMap) {
        nodos.push_back(new nodo(pair.first, pair.second));
    }

    raiz = construirArbolHuffman(nodos);
    cout << "\nArbol de Huffman generado desde el archivo:" << endl;
    mostrarArbolHuffman(raiz);
}

void generarTablaCodigos(nodo* raiz, unordered_map<char, string> &tabla,
string codigo) {
    if (!raiz) return;
    if (raiz->letra != ' ') {
        tabla[raiz->letra] = codigo;
    }
    generarTablaCodigos(raiz->izq, tabla, codigo + "0");
    generarTablaCodigos(raiz->der, tabla, codigo + "1");
}

void comprimirArchivo(const string &archivoOriginal, const string
&archivoComprimido, nodo* raiz) {
    unordered_map<char, string> tablaCodigos;
    generarTablaCodigos(raiz, tablaCodigos);

    ifstream archivoEntrada(archivoOriginal, ios::binary);
    ofstream archivoSalida(archivoComprimido, ios::binary);

    if (!archivoEntrada.is_open() || !archivoSalida.is_open()) {
        cout << "Error al abrir los archivos." << endl;
        return;
    }

    stringstream ss;
    char c;
```

```
    while (archivoEntrada.get(c)) {
        ss << tablaCodigos[c];
    }

    string contenidoCodificado = ss.str();
    archivoSalida << contenidoCodificado;

    archivoEntrada.close();
    archivoSalida.close();

    cout << "Archivo comprimido correctamente." << endl;
}

void descomprimirArchivo(const string &archivoComprimido, const string
&archivoRecuperado, nodo* raiz) {
    ifstream archivoEntrada(archivoComprimido, ios::binary);
    ofstream archivoSalida(archivoRecuperado, ios::binary);

    if (!archivoEntrada.is_open() || !archivoSalida.is_open()) {
        cout << "Error al abrir los archivos." << endl;
        return;
    }

    nodo* actual = raiz;
    stringstream contenidoCodificado;
    char bit;
    while (archivoEntrada.get(bit)) {
        contenidoCodificado << bit;
    }

    string contenido = contenidoCodificado.str();
    for (char bit : contenido) {
        actual = (bit == '0') ? actual->izq : actual->der;
        if (!actual->izq && !actual->der) {
            archivoSalida.put(actual->letra);
            actual = raiz;
        }
    }

    archivoEntrada.close();
    archivoSalida.close();

    cout << "Archivo descomprimido correctamente." << endl;
}
```

```
void calcularMetricas(const string &archivoOriginal, const string
&archivoComprimido) {
    ifstream archivoO(archivoOriginal, ios::binary | ios::ate);
    ifstream archivoC(archivoComprimido, ios::binary | ios::ate);

    if (!archivoO.is_open() || !archivoC.is_open()) {
        cout << "Error al abrir los archivos para metricas." << endl;
        return;
    }

    size_t tamOriginal = archivoO.tellg();
    size_t tamComprimido = archivoC.tellg();
    double factorCompresion = (double)tamOriginal / tamComprimido;

    cout << "Tamano original: " << tamOriginal << " bytes" << endl;
    cout << "Tamano comprimido: " << tamComprimido << " bytes" << endl;
    cout << "Factor de compresion: " << factorCompresion << endl;

    archivoO.close();
    archivoC.close();
}
```

- **Funciones:**
    - insertarNodo: Solicita al usuario ingresar una letra y su probabilidad, luego crea un nodo con esos valores y lo agrega a un vector de nodos.

    - construirArbolHuffman: Toma un vector de nodos, los coloca en una cola de prioridad, y luego construye el árbol de Huffman combinando nodos hasta que quede uno solo, el cual se retorna como la raíz del árbol.

    - mostrarArbolHuffman: Imprime el árbol de Huffman de forma visualmente estructurada, mostrando las probabilidades y letras de cada nodo con una indentación para reflejar la jerarquía del árbol.

    - leerArchivoYConstruirHuffman: Lee un archivo, cuenta la frecuencia de cada carácter, construye los nodos con esas frecuencias, y luego genera el árbol de Huffman y lo muestra.

    - generarTablaCodigos: Recursivamente genera una tabla de códigos Huffman para cada letra en el árbol, donde cada letra tiene un código binario asociado.

- comprimirArchivo: Comprime un archivo utilizando la tabla de códigos generada, reemplazando cada carácter por su código binario correspondiente y escribiendo el resultado en un archivo comprimido.

- descomprimirArchivo: Lee un archivo comprimido, utiliza el árbol de Huffman para decodificar los códigos binarios y reconstruye el archivo original, escribiendo el resultado en un archivo recuperado.

- calcularMetricas: Calcula el tamaño de los archivos original y comprimido y muestra el factor de compresión, que es la relación entre el tamaño original y el comprimido.

Conclusiones:

El proyecto cumple con los requisitos base de la compresión de datos utilizando el algoritmo de Huffman que se habían trabajado ya en tareas pasadas implementando un poco más de funciones y de librerías necesarias para completarlo, aunque el objetivo inicial era incorporar tanto el procesamiento de archivos de texto como de audio y video, me encontré con múltiples dificultades debido a la necesidad de librerías adicionales que al momento de ser instaladas me generaron muchos errores recurrentes teniendo así un bloqueo por ese medio, por lo mismo opté por centrarme en lo esencial del proyecto, lo que fue lograr insertar nodos de manera manual y llevar a cabo todo el proceso de compresión y descompresión usando texto.