NOME: GABRIEL FERREIRA DE SOUZA ARAUJO **NUSP: 12718100**

RELATÓRIO EPSUB - ALGORÍTMO E ESTRUTURA DE DADOS II

1. FUNÇÕES

unordered_map<char, int> getCharacterFrequencies(const string& text): Esta função recebe uma string de texto como entrada e retorna um unordered_map que mapeia cada caractere presente no texto para a sua frequência de ocorrência.

Node* buildHuffmanTree(const unordered_map<char, int>& frequencies): Esta função recebe o mapa de frequências gerado pela função anterior e constrói a árvore de Huffman correspondente. A árvore de Huffman é construída usando uma fila de prioridade (priority_queue) para selecionar os nós com menor frequência e combiná-los até que reste apenas um nó na fila, que será a raiz da árvore.

void traverseHuffmanTree(Node* root, string code, unordered_map<char, string>& huffmanCodes): Esta função percorre a árvore de Huffman e atribui um código binário a cada caractere presente na árvore. O código binário é calculado concatenando '0' para movimento à esquerda na árvore e '1' para movimento à direita. Os códigos são armazenados no mapa huffmanCodes.

unordered_map<char, string> getHuffmanCodes(Node* root): Esta função envolve a função traverseHuffmanTree para obter o mapa completo de códigos Huffman para cada caractere presente na árvore.

string encodeText(const string& text, const unordered_map<char, string>& huffmanCodes): Esta função recebe um texto e o mapa de códigos Huffman e retorna o texto codificado usando os códigos Huffman correspondentes a cada caractere.

void decodeText(const string& encodedText, Node* root): Esta função recebe um texto codificado e a raiz da árvore de Huffman e realiza a descompactação do texto. Ela segue os códigos Huffman para percorrer a árvore e decodificar cada caractere. O texto descompactado é salvo em um arquivo chamado "descompactado.txt".

void textToBinaryFile(const string& binaryText, const string& fileName): Esta função recebe um texto binário e um nome de arquivo e grava o texto binário no arquivo especificado. Os caracteres do texto binário são agrupados em bytes e gravados no arquivo como bytes individuais.

string lerArquivoBinario(const string& nomeArquivoBinario): Esta função lê um arquivo binário e retorna o conteúdo do arquivo como uma string binária. Ela lê os bytes do arquivo e converte cada byte em uma sequência de 8 bits (0s e 1s) representando o conteúdo binário do arquivo.

int main(): A função principal do programa realiza as seguintes etapas:

1. Abre o arquivo de entrada "input.txt" e lê o conteúdo do arquivo para a variável text.
2. Calcula as frequências dos caracteres no texto usando a função getCharacterFrequencies.
3. Constrói a árvore de Huffman usando a função buildHuffmanTree.
4. Obtém os códigos Huffman para cada caractere usando a função getHuffmanCodes.
5. Codifica o texto usando os códigos Huffman usando a função encodeText e salva o texto codificado em um arquivo chamado "compactado.bin".
6. Lê o arquivo binário "compactado.bin" usando a função lerArquivoBinario.
7. Realiza a descompactação do texto usando a função decodeText e salva o texto descompactado no arquivo "descompactado.txt".

2. TESTES

Anexado ao Relatório, temos 6 testes: texto_Codigo (possui um texto em formato de codigo do app logsim), texto_CodigoEPSUB (o próprio código do EPSUB), texto_Arte (Texto que contém uma arte de caractéres ascii), texto_BancoGoverno (contém um banco de dados grande do governo, texto_LorenIpsum (Texto grande com palavras aleatórias em latim), texto_Biblia (A biblia King James em txt). Cada um deles têm diferentes formas de arranjos de caracteres permitindo, portanto, validar a funcionalidade do código, passando em todos os testes com erro apenas em alguns casos que sobram dois caracteres que não estavam no texto, eles foram lidos do arquivo binário (esse erro foi esperado pois não foi corrigido o caso em que sobram binários no arquivo, mas isso pode ser irrelevante para os resultados).

3. CONCLUSÕES

Ao executar o programa com cada teste explicado no item acima e verificar o tamanho do arquivo original, compactado e descompactado, foi possível notar que o arquivo descompactado tem sempre o mesmo tamanho do original, enquanto o compactado tem em média 48% do tamanho do arquivo original, isso prova que o objetivo do algoritmo está sendo cumprido. Vale acrescentar, também, que a velocidade de execução é rápida até para os casos maiores, onde o pior caso dos testes foi em “texto_BancoGoverno” com um tempo de 4 segundos, e o restante dos testes foram em prática “instantâneos”. Esse último resultado comprova a eficiência do algoritmo de Hulffman. Abaixo está a lista do tamanho do arquivo antes e depois da descompressão respectivamente:

• texto_Codigo: 41.980 bytes - 23.023 bytes

• texto_CodigoEPSUB: 5.206 bytes - 2.970 bytes

• texto_Arte: 2.469 bytes - 735 bytes

• texto_BancoGoverno: 56.314.547 bytes - 32.743.996 bytes

• texto_LorenIpsum: 186.622 bytes - 99.443 bytes

• texto_Biblia: 4.437.474 bytes - 2.487.670 bytes

Após a realização do trabalho é possível notar a relevancia desse algorítmo para alguns setores de aplicação como compressão de arquivos, transmissão de dados em redes, compactação de imagens e vídeos, compressão de dados em bancos de dados, entre outros. Portanto, o algoritmo de Huffman desempenha um papel crucial na compressão de dados, permitindo a redução do tamanho de arquivos e otimização da transmissão de dados.

4.OBSERVAÇÕES PARTICULARES DO EP

O enunciado pede para fazer dois arquivos: “compacta” e “descompacta”. Neste ep, isso foi feito de modo diferente. O objetivo principal de realização da compressão de descompressão foi compactado com sucesso, entretanto, não foi possível criar os dois arquivos separadamente por conta da dificuldade de passar o “node” da arvore como parâmetro na função descompacta por meio de uma convenção embutida no arquivo.bin. Isso claramente influenciou num maior potencial de descompressão para

textos pequenos, pois não foram necessários bits para convenção dos dados da arvore no “arquivo.bin”. prosseguindo, neste EP as duas funções estão presentes em um único programa chamado EPSUB.cpp. Apesar disso, a descompressão se torna consitente se analisar as funções usadas na “main” para a execução dessa função. Essas funções estão explicadas no tópico 1.

5. COMO COMPILAR / RODAR

Para compilar digite na linha de comando: “cpp EPSUB.c -o EPSUB”.

Para executar digite na linha de comando: “./EPSUB”.

Para interagir com o programa, digite o nome do arquivo que deseja comprimir ou descomprimir, por exemplo: “texto_Arte.txt” (Os arquivos precisam estar no mesmo diretório que o executável)

Após essas etapas, ao término do programa, serão gerados dois arquivos: “compactado.bin” e “descompactado.txt”. Ao abri-los é possível conferir os dados exibidos no tópico 3. OBS: (o arquivo descompactado.txt usa o compatado.bin para sua construção).