Universidade Federal do Espírito Santo

Uarlley do Nascimento Amorim - Matricula: 2018205346
Uálaci dos Anjos Júnior - Matricula: 2017102015
14 de junho de 2021

Estrutura de Dados II

Professora: Luciana Lee

Ciência/Engenharia da computação

Universidade Federal do Espírito Santo
DCEL

Relatório

Relatório de implementação do Algoritmo de compressão de Huffman

Aluno(a)s: Uarlley Nascimento amorim,
Uálaci dos Anjos Júnior

Professor(a) orientador(a): Luciana Lee

14 de junho de 2021

Sumário

1 Introdução

O algoritmo de Huffman se trata de uma codificação de caracteres que permite compactar arquivos de texto, ou seja, representar um arquivo de texto de entrada para outro de saída menor. O algoritmo de Huffman calcula uma tabela de códigos sob medida para um arquivo de entrada de modo que o arquivo de saída seja o menor possível. Esse algoritmo faz uso da frequência de ocorrência dos caracteres, ou símbolos, no arquivo a ser comprimido a fim de determinar uma representação compactada para cada símbolo. Ele surgiu em 1952 por autoria de David A. Huffman que na época, cursava doutorado no MIT, e foi publicado no artigo "A Method for the Construction of Minimum-Redundancy Codes".

Nesse relatório iremos apresentar a implementação de um programa em c cuja a função é compactar um arquivo de texto para um arquivo binário de tamanho menor, utilizando o conceito do algoritmo de Huffman como base.

2 Estruturas

Nessa seção serão apresentadas as estruturas criadas para a implementação dos TAD's Huffman e BST.

2.1 Estrutura que guarda um caractere e seu caminho na arvore de Huffman

```
typedef struct tabela {
    bool* vet;//Possui um vetor de booleanos para salvar o codigo do caractere
    int size;//O tamanho do vetor de booleanos
    char caractere;//armazena o caractere salvo
}tab;
```

Durante a implementação do algoritmo, iremos utilizar uma lista encadeada para salvar a quantidade de ocorrências de cada caracter no arquivo de entrada. Esse é a estrutura de dados que essa lista irá utilizar, composta por um ponteiro para o próximo elemento da lista, e um ponteiro para um nó da árvore de Huffman.

2.2 Estrutura da Árvore de Huffman

```
typedef struct tnode {
    struct tnode* left;
    struct tnode* right;
    int freq;//possui um campo frequencia para nos nao-folha
    Info* info;//apenas os nos folha possuem um campo info
}BSTNode;
```

Durante a implementação, será necessário uma estrutura de dados para implementar a árvore de Huffman, que será explicada posteriormente. Essa é a estrutura que essa árvore irá utilizar, composta por um ponteiro para o filho a esquerda, ponteiro para um filho a direita, uma frequência para os nós não folha, e um ponteiro para um elemento de informação, utilizado para armazenar as informações dos nós folha.

2.3 Estrutura que armazena um caractere e sua frequência em um texto

```
typedef struct {
    int frequencia;
    char caractere;
}Info;
```

Essa é a estrutura que representa o elemento de informação de cada nó folha da árvore de Huffman. Essa estruturá contém um campo para a frequência de ocorrência de um determinado caractere no arquivo de entrada, e um campo para armazenar esse caractere em si.

2.4 Estrutura de Lista Simplesmente Encadeada

```
typedef struct lnode {
    struct lnode* next;
    BSTNode* bstnode; //Contem um ponteiro para um no de arvore para utilizar no algoritmo de Ruffman
}Node;
```

Esta estrutura possui ponteiro para o próximo nó da lista e um ponteiro para um no de arvore, que será utilizado na construção do algoritmo de Huffman.

3 Funções do TAD BST

Nesta seção são apresentadas as funções implementadas no TAD BST. Não houve necessidade de implementar nenhuma função de manipulação de arvores de busca binária tais como inserção, remoção e busca. Mas, algumas funções de de suporte e funções de manipulação de listas encadeadas foram implementadas.

3.1 Pega Frequencia

```
int getFreq(Node* node) {
    return node->bstnode->freq; //retorna a frequencia contida no no de arvore
}
```

A função acima funciona de forma bem simples. E passado um nó de lista, e, então, a frequência do caractere contida no nó de arvore que está contido nesse nó de lista é retornado.

3.2 Pega Frequencia

```
int getFreq(Node* node) {
    return node->bstnode->freq; //retorna a frequencia contida no no de arvore
}
```

3.3 Pega Caractere

```
char getBstChar(Node* node) {
    return node->bstnode->info->caractere; //retorna o caractere presente no no de arvore
}
```

Analogamente, a função acima recebe um nó de lista e retorna o caractere contido no nó de arvore contido na lista.

3.4 Cria nó de lista

```
Node* createNode(BSTNode* bstnode) {
    Node* newNode = (Node*)malloc(sizeof(Node));//aloca o espaco para o no de lista
    if (newNode) { //se o no for alocado, seu ponteiro para o proximo elemento e definido como nulo
                   //e seu no para o no de arvore aponta para o no passado no parametro
        newNode->next = NULL;
        newNode->bstnode = bstnode;
    }
    return newNode;
}
```

Esta função cria e retorna um nó para uma lista encadeada. Para realiza a criação, é necessário passar como parâmetro um nó de árvore, que será armazenado neste novo nó de lista. Assim, a memória necessária é alocada, e, se a alocação for bem sucedida, seu ponteiro para o próximo elemento da lista é direcionado para $NULL$, e seu ponteiro para nó de arvore é direcionado para o nó fornecido.

3.5 Busca na Lista

```
int searchList(Node* head, char caractere){
    while(head!=NULL){//Segue na lista ate encontrar o ultimo no
        //incrementa a frequencia do caractere e retorna 1 representando sucesso na busca
        if(head->bstnode->info->caractere == caractere) {
            (head->bstnode->info->frequencia)++;
            (head->bstnode->freq)++;
            return 1;
        }
        head = head->next;//segue avancando na lista
    }
    return 0;//retorna 0 pois a lista chegou ao fim e o caractere nao foi encontrado
}
```

Esta função é responsável por realizar a busca de um caractere em uma lista simplesmente encadeada e funciona de forma simples. O caractere passado é comparado com os caracteres da lista d forma sequencial, e, se o mesmo for encontrado, sua posição na lista é retornado. Se a lista chegar ao fim, é retornado -1, sinalizando que o elemento não foi encontrado na lista.

3.6 Insere na lista

```
Node* insertList(Node* head, BSTNode* bstnode) {

    Node* newNode = createNode(bstnode);//cria o novo no a ser inserido na lista
    if (head == NULL) return newNode; //caso a lista esteja vazia o novo no sera a cabe a da lista

    Node* aux = head;//cria um ponteiro auxiliar para percorrer a lista

    if (getFreq(newNode) < getFreq(head)) { // o novo no sera a nova cabeca da lista se sua frequencia
                                            // for menor que a frequencia da cabeca atual
        newNode->next = head;
        head = newNode;
        return head;
    }
    //avanca na lista ate encontrar a posicacao onde o novo no deve ser inserido
    while (aux->next != NULL && getFreq(aux->next) < getFreq(newNode)) aux = aux->next;

    //realiza a insercao do novo no
    newNode->next = aux->next;
    aux->next = newNode;

    return head; //retorna um ponteiro para a cabeca lista
}
```

Esta função realiza a inserção de um nó em uma lista simplesmente encadeada de forma iterativa. Inicialmente o novo nó é criado, e, se a lista estiver vazia ou a frequência do caractere contido neste novo nó for menor do que a frequência do caractere contida na cabeça da lista, tal nó será retornado como o nó cabeça da lista. Caso exista mais de um elemento, é buscada a posição onde este novo elemento deve ser inserido, utilizando a frequência do caractere como critério de ordenação. Após realizar os apontamentos, a cabeça da lista é retornada.

3.7 Cria nó de arvore

```
NBSTNode* createBSTNode(Info* info, int freq) {
    BSTNode* newnode = (BSTNode*)malloc(sizeof(BSTNode)); //Aloca o espaco necessario para o novo no
    if (newnode != NULL) {//se a alocacao for bem sucedida os campos sao preenchidos
        newnode->info = info;
        newnode->left = newnode->right = NULL;
        newnode->freq = freq;
    }
    return newnode; //retorna o no criado
}
```

Esta função é responsável por criar um novo nó para arvore de Huffman. Para isso é necessário um ponteiro para um elemento de informação e a frequência do caractere que será inserido, ou a soma das frequência dos futuros filhos deste novo nó. A partir disto, é realizada a alocação de memória para o novo nó, e, se a alocação for bem sucedida, o ponteiro para o elemento de informação é direcionado para o elemento de informação informado, assim como o campo $freq$. Vale ressaltar que, se este novo nó não for um nó folha, o campo $info$ será $NULL$, pois, tal nó não armazenará nenhum caractere, apenas uma frequência.

3.8 Altura da arvore

```
int heightBST(BSTNode* root) {
    int lH = 0, rH = 0;//define variaveis para representar a altura esquerda e direta da arvore
    if (root == NULL) return 0;//a arvore nao existir sua altura e 0
    lH = 1 + heightBST(root->left);//calcula a altura da subarvore esquerda
    rH = 1 + heightBST(root->right);//calcula a altura da subarvore direita
    //retorna a maior entre as duas alturas
    if (rH > lH) return rH;
    else return lH;
}
```

A função acima calcula e retorna a altura da arvore de Huffman, para isso, são criadas duas variáveis que recebem as alturas das subárvores esquerda e direita, e, então, a partir de chamadas recursivas, a maior das alturas é retornada.

3.9 Cria elemento de informação

```
Info* createInfo(char caractere, int freq) {
    Info* info = (Info*)malloc(sizeof(Info));//aloca o espaco necessario
    if (info) {//Se o ponteiro for alocado, as informacoes sao preenchidas
        info->caractere = caractere;
        info->frequencia = freq;
    }
    return info;//retorna o ponteiro para o novo elemento de informacao
}
```

Essa função recebe como parâmetro um caractere *caractere*, e a frequência $freq$ do mesmo, e então, cria um elemento de informação do tipo Info, e salva o caractere e a frequência em seus respectivos campos no novo elemento de informação criado, e retorna esse elemento de informação.

3.10 Imprime código de Huffman

```
void imprimeCodHuffman(BSTNode* root){
    if(root == NULL) return;//se a arvore nao existir a funcao    finalziada

    imprimeCodHuffman(root->left);//realiza a chamada recursiva para a esquerda
    //se o campo de informacao existir, ou seja, ser um no folha, eh impresso o caracter e sua frequencia
    if(root->info){
        if(root->info->caractere == '\n') printf("Caractere: \\n - Frequencia: %d \n", root->info->frequencia
    );
        else if(root->info->caractere == ' ') printf("Caractere: espaco - Frequencia: %d \n", root->info->
    frequencia);
        else printf("Caractere: %c - %d - Frequencia: %d \n",root->info->caractere, root->info->caractere,
    root->info->frequencia);
    }
    //caso contrario, eh especificado que se trata de um no nao folha
    else printf("No n o folha  - Frequencia: %d \n", root->freq);
    imprimeCodHuffman(root->right);//realiza a chamada recursiva para a direita
}
```

Essa função recebe como parâmetro a raiz da árvore *root*, e imprime toda a árvore, por meio de um percurso de árvore In ordem. Caso o nó seja folha, é impresso o caractere, e a frequência desse nó, do contrário, somente é impresso a frequência e uma notificação de que o nó não é folha.

3.11 Imprime Ocorrências

```
void imprimeOcorrencias(Node* head){
    Node* aux = head;//cria um ponteiro auxiliar para percorrer a lista
    //percorre a lista ate o fim realizando a impressao dos caracteres e suas respectivas frequencias
    while(aux){
        if(getBstChar(aux) == '\n') printf("Caractere: \\n - Ocorrencia(s): %d\n",aux->bstnode->info->
    frequencia);
        else if (getBstChar(aux) == ' ') printf("Caractere: espaco - Ocorrencia(s): %d\n",aux->bstnode->info
    ->frequencia);
        else printf("Caractere: %c - Ocorrencia(s): %d\n", getBstChar(aux),aux->bstnode->info->frequencia);
        aux = aux->next;//segue na lista
    }
}
```

4 Funções Principais

Nessa seção, serão analisados os código das funções principais, que são as funções responsáveis por realizar as tarefas tradicionais da árvore, tais como inserção, remoção e busca.

4.1 Arvore de Huffman

```
BSTNode* huffman(Node* lista, int size) {


    Node* x, * y;
    if(size > 1){//se a lista possuir mais de um elemento
        for (int i = 1; i < size; i++) {
            x = lista;
            y = lista->next;
            lista = y->next;


            BSTNode* conector = createBSTNode(NULL, getFreq(x) + getFreq(y));/*soma a frequencia dos dois
    primeiros nos da lista e utiliza a soma para
                                                                        como frequencia de um no de
    arvore*/

            //torna os ponteiros para no de arvore dos dois primeiros nos da lista filhos deste novo no de
    arvore
            conector->left = x->bstnode;//
            conector->right = y->bstnode;
            lista = insertList(lista, conector);//insere o novo elemento n lista
            //libera apenas a memoria alocada para o no de lista, mantendo o ponteiro para o no de arvore e
    suas informacoes
            free(x);
            free(y);
        }
    }
    BSTNode* retorno = lista->bstnode;//salva o poteiro para a raiz da arvore
    free(lista);//libera a memoria alocada no ultimo no da lista
    return retorno;
}
```

Essa função é repensável por criar a árvore de Huffman. A árvore de Huffman é um algoritmo de árvore no qual cada nó folha representa um caractere, juntamente de sua frequência, e os nós não folha somente existem para fornecer um caminho, e possuem somente a soma da frequência de seus filhos. Esse caminho, do nó raiz até um nó folha, é a representação do código binário que será utilizado na compactação do caractere contido nesse nó folha, no arquivo de saída comprimido. Nesse percurso, um avanço para o filho a esquerda representa o valor 0, e um avanço para o filho a direita representa o valor 1, como pode ser observado na *figura* 1.

A função $huffman$ recebe como parâmetro a lista criada na função $lerArquivo$, e o tamanho da lista, e cria uma árvore de Huffman a partir dessa lista. Para criar essa árvore, a função faz uma iteração que ocorre n

vezes, em que n é o tamanho do da lista. A cada iteração, os dois primeiros elementos da lista são removidos, e então um novo elemento é criado, esse elemento possui um ponteiro para lista e um nó de árvore binária de busca, o filho a esquerda desse nó passa a ser então o primeiro elemento da lista, o segundo, o filho a direita, e a frequência desse nó é a soma das frequências de seus filhos a esquerda e a direita. então esse nó é inserido na lista de forma ordenada, utilizando a frequência como parâmetro de ordenação. Esse processo vai então se repetindo, e ao fim das iterações, uma árvore de Huffman é gerada a partir da lista, e a lista ao fim do processo está vazia. A função então retorna essa árvore.

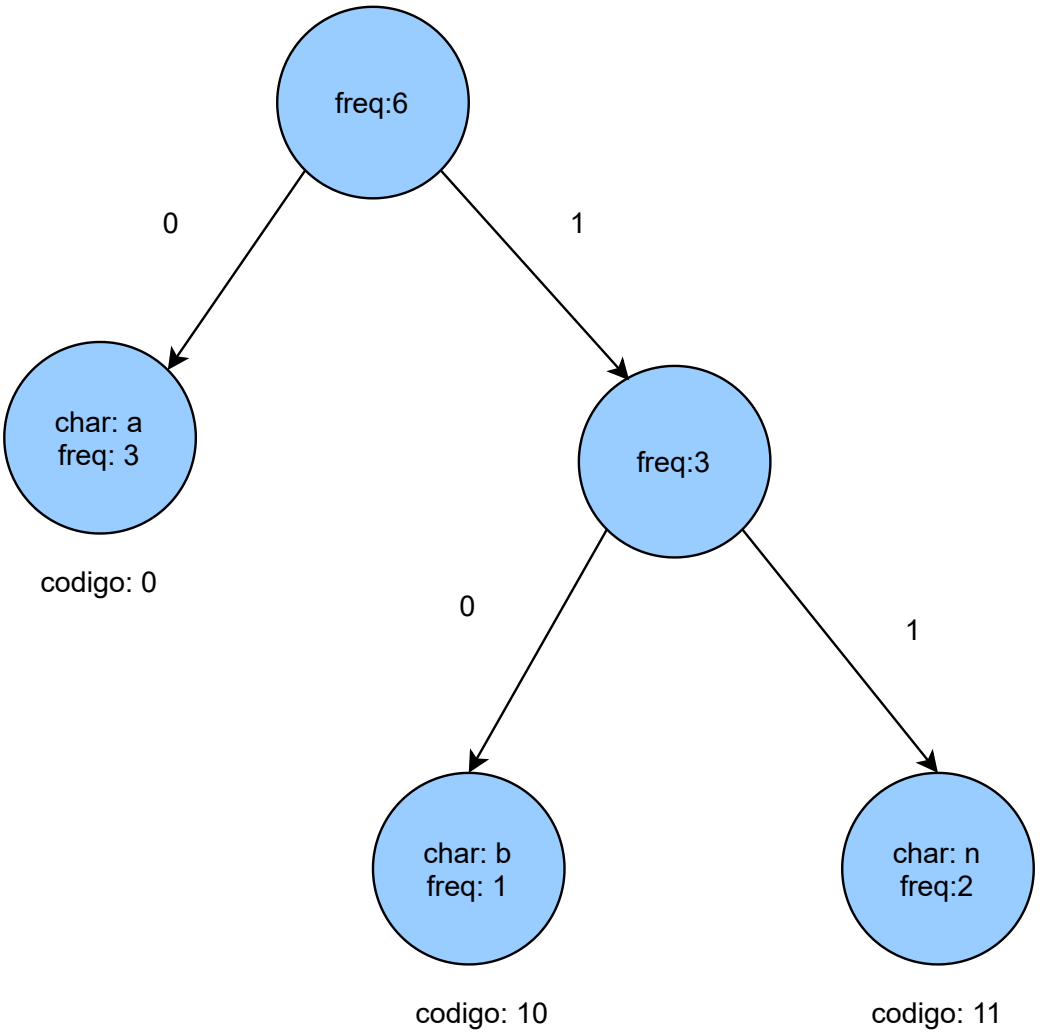

Figura 1: Exemplo da árvore de Huffman para o texto "banana".

4.2 Escreve Tabela

```
void writeTab(BSTNode* root, tab tabela[],bool vet[], int *size, int *posTab) {
    if (!root) {
        (*size)--;//atualiza o tamanho do codigo
        return;
    }

    if (root->left) {
        vet[*size] = 0;//0 pois o caminho vai para esquerda
        (*size)++;//atualiza o tamanho do caminho
        writeTab(root->left,tabela,vet, size, posTab);//realiza a chamada recursiva
```

```
    }

    if (root->right) {
        vet[*size] = 1;//1 pois o caminho vai para direita
        (*size)++;//atualiza o tamanho do caminho
        writeTab(root->right, tabela,vet,size,posTab);//realiza a chamada recursiva
    }

    if (root->info != NULL) {//no folha encontrado sinalizando o fim do caminho
        //salva o caminho na tabela
        for (int i = 0; i < *size; i++) {
            (tabela[*posTab]).vet[i] = vet[i];
        }
        //salva o caractere na tabela
        (tabela[*posTab]).caractere = root->info->caractere;
        (tabela[*posTab]).size = *size;
        (*posTab)++;//incrementa a posicao na tabela
    }
    (*size)--;//Decrementa o tamanho pois o fim da arvore foi alcan ado
}
```

Essa função é responsável por criar uma tabela, que contém em cada posição um caractere, e seu código gerado pelo algoritmo de huffman, que será utilizado durante a compressão do arquivo de entrada.

A função recebe como parâmetro um nó de árvore *root*, que representa a raiz da árvore de huffman criada na função $huffman$, o parâmetro *tabela*, que representa uma tabela com posições vazias, que será a tabela na qual as informações serão armazenadas e o parâmetro *size* que representa o tamanho de um vetor que será utilizado para armazenar o código de cada caractere, e $sizeTab$, que representa o tamanho da tabela.

A função irá percorrer a árvore, e cada vez que for avançando na árvore, o valor 1 ou o valor 0, será escrito no vetor *vet*, e ao chegar num nó folha, o caractere que esse nó folha contém será armazenado na posição atual representada por $sizeTab$, juntamente de seu código. Esse vetor é reaproveitado para salvar o código dos outros elementos, de tal forma que não é necessário percorrer toda a árvore novamente para determinar o código outros nós folhas, otimizando assim o código. Para que isso ocorra, toda vez que um nó nulo é encontrado, ou que a recursão volta um passo, o tamanho *size* é reduzido, para que o vetor *vet* escreva adequadamente o código dos caracteres.

4.3 Lê Arquivo

```
Node* leArquivo(char* nomeArquivo, int* sizeList, long long* charQnt) {
    Node* head = NULL;//Cria a cabeca da lista
    FILE* arquivo = fopen(nomeArquivo, "r");//realiza a abertura do arquivo
    if (!arquivo) printf("Arquivo nao existe");
    //Se o arquivo existir e realiza a leitura caractere por caractere
    else{
        fseek(arquivo, 0, SEEK_END);
        long int size = ftell(arquivo);//Guarda a ultima posicao do arquivo
        fseek(arquivo, 0, SEEK_SET);
        if(size == 0) return NULL;
        while (ftell(arquivo) < size) {//le ate encontrar o final do arquivo
            char t;
            fread(&t, sizeof(char), 1, arquivo);
            /*verifica se o caracter ja esta na lista, se sim sua frequencia e incrementada, caso contrario
            o mesmo e inserido na lista*/
            if(searchList(head,t) == 0){
                head = insertList(head, createBSTNode(createInfo(t, 1), 1));
                (*sizeList) += 1;
            }
            (*charQnt)++;
        }
        fclose(arquivo);//fecha o arquivo
    }
    return head;//retorna o ponteiro para a cabeca da lista
}
```

Essa função é responsável por ler o arquivo de entrada e inserir cada caracter presente nesse arquivo, em uma lista, juntamente da sua frequência de ocorrência. Os parâmetros são o *nomeArquivo*, que representa o nome do arquivo a ser lido, e *sizeList* que representa o tamanho da lista.

A função lê o arquivo, e percorre cada caractere do mesmo, e para caractere, é feita uma busca na lista para verificar se esse caractere não já se encontra nessa lista, caso ele seja encontrado, sua frequência é incrementada em 1, e caso não seja encontrado, esse caractere é inserido na lista com uma frequência inicial de um, e para cada caractere inserido na lista, *sizeList* terá seu valor incrementado em 1, como sizeList é um ponteiro, seu conteúdo será alterado de forma global, e então, ao fim do processo, a lista criada é retornada.

4.4 Escreve Binário

```
void writeBin(tab tabela[], char* nomeEntrada, char* nomeSaida, long long countSizeOut, int sizeTab,int
    sizeTree, long long charQnt) {
    int posicao = 0;
    unsigned char* saida = (unsigned char*)calloc(countSizeOut, sizeof(unsigned char));//aloca o vetor para
    armazenar o texto comprimido

    FILE* arquivo = fopen(nomeEntrada,"r");//leitura do arquivo a ser comprimido
    if (!arquivo) printf("Arquivo nao existe");
    else {//caso o arquivo exista, cada caracter eh lido e comprimido
        fseek(arquivo, 0, SEEK_END);
        long int size = ftell(arquivo);
        fseek(arquivo, 0, SEEK_SET);
        while (ftell(arquivo) < size) {
            char t;
            fread(&t, sizeof(char), 1, arquivo);
            int pos = searchTab(tabela,t,sizeTab);
            bitToBit(tabela[pos].vet, tabela[pos].size, &posicao, saida);
        }
        fclose(arquivo);//fecha o arquivo de entrada
        pack(nomeSaida, saida, countSizeOut, tabela,sizeTab,sizeTree,charQnt);//compacta em um arquivo.cpt
    }
    free(saida);//libera a memoria utilizada
}
```

a função *writeBin* é responsável por criar o texto comprimido e escrevê-lo no arquivo de saída. Essa função recebe como parâmetro *tabela*, que é a tabela com os códigos de cada caractere, *nomeEntrada*, que é o nome do arquivo de entrada, *nomeSadida* que é o nome escolhido para o arquivo de saída, *countSizeOut*, que representa o tamanho em bits necessário para codificar o arquivo de entrada, *sizeTab*, que representa o tamanho da tabela, *sizeTree*, que representa o tamanho da árvore de Huffman utilizada para a conversão, e *charQnt*, que representa a quantidade de caracteres que serão escritos no arquivo de saída.

A função cria um loop, que a cada iteração, lê um caractere do arquivo de entrada, converte o caractere para seu código comprimido, pela função *bitToBit*, e chama a função para ao fim do loop, para realizar a escrita do vetor *saida* e de outras informações adicionais, como a tabela de conversão, no arquivo de saída. Ao fim da escrita, o vetor *saida* é liberado da memória. É válido ressaltar que para um arquivo de entrada, a quantidade de bits alocada para o vetor de saída provavelmente não será exata, isso porque só é possível alocar de 8 em 8 bits na memória, dessa forma, é comum que sobre no vetor *saida*, alguns espaços de bit extras, e para corrigir esse problema, a variável *charQnt* serve, para que durante a descompactação que será descrita posteriormente, esses bits extras não sejam descompactados gerando informações que não estavam no arquivo de entrada, tornando a descompactação perfeita.

4.5 Compacta

```
void pack(char* nomeArquivo, unsigned char saida[], int sizeSaida, tab* tabela, int sizeTab, int sizeTree,
    long long charQnt){
    FILE* write_ptr = fopen(nomeArquivo, "wb");  //abertura do arquivo
    fwrite(&charQnt, sizeof(long long), 1, write_ptr);//escrita da quantidade maxima de caracteres contidos
    no texto
```

```
    fwrite(&sizeTab, sizeof(int), 1, write_ptr);//escrita do tamanho da tabela de codigos
    fwrite(&sizeTree, sizeof(int), 1, write_ptr);//escrita do altura da arvore
    //escrita da tabela e o codigo de caractere contido na mesma
    fwrite(tabela, sizeof(tab), sizeTab, write_ptr);
    for(int i = 0; i < sizeTab; i++){
        int sizeSaida = (tabela[i].size)/8 + 1;
        unsigned char* saidaVet = (unsigned char*)calloc(sizeSaida, sizeof(unsigned char));
        int posicao = 0;
        //compressao e escrita do codigo do caractere da tabela
        bitToBit(tabela[i].vet, tabela[i].size, &posicao,saidaVet);
        fwrite(saidaVet, sizeof(unsigned char*), sizeSaida, write_ptr);
        free(saidaVet);//Liberacao do vetor temporario
    }
    fwrite(&sizeSaida, sizeof(int), 1, write_ptr);//Escrita do tamanho do vetor de codificacao
    fwrite(saida, sizeof(unsigned char), sizeSaida, write_ptr);//escrita do vetor codifica  o
    fclose(write_ptr);//fechamento do arquivo
}
```

A função *pack* é responsável por escrever no arquivo de saída o vetor saída que contém o texto do arquivo de entrada já comprimido e algumas informações adicionais , que serão necessárias para realizar a descompactação do arquivo quando necessário.

Seus parâmetros são *nomeArquivo*, que representa o nome do arquivo de saída, *sizeSaida*, que representa o tamanho do código dos caracteres presentas na tabela criada por *writeTab*,*sizeTab* que representa o tamanho dessa tabela. *saida*, que representa o vetor de saída com os caracteres do arquivo de entrada comprimidos, e *sizeTree* que representa o tamanho da arvore de Huffman criada para realizar a compressão.

A função abre o arquivo de saída e vai escrevendo nele cada posição da tabela. No momento em que o código de conversão presenta na tabela para cada caractere é escrito, a função bitTobit é chamada para realizar a compressão desse código de tal forma que ele fique o mais comprimido possível, pois para textos muito grandes esse código pode conter mais de posições. Após a conversão desse código, ele é escrito no arquivo de saída. Por fim, o vetor *saida* é escrito no arquivo de saída finalizando a compressão.

4.6 Descompacta

```
void unpack(FILE* arquivo, char* nomeSaida){
    int countRead, sizeTab,sizeTree, charQnt;
    fread(&charQnt, sizeof(long long), 1, arquivo);//leitura da quantidade total de caracteres presentes no
    arquivo de texto.
    fread(&sizeTab, sizeof(int), 1, arquivo);//leitura do tramanho da tabela.
    fread(&sizeTree, sizeof(int), 1, arquivo);//leitura do tamanho da arvore.
    //criacao e preenchimento da tabela
    tab* tabela = criaTabela(sizeTree,sizeTab);
    fread(tabela, sizeof(tab), sizeTab, arquivo);
    for(int i = 0; i < sizeTab; i++){
        int sizeSaida = (tabela[i].size)/8 + 1;
        unsigned char* saidaVet = (unsigned char*)calloc(sizeSaida, sizeof(unsigned char));
        fread(saidaVet, sizeof(unsigned char*),sizeSaida, arquivo);
        tabela[i].vet = intToBin(saidaVet,sizeSaida);
        free(saidaVet);
    }
    fread(&countRead, sizeof(int), 1, arquivo);//leitura do tamanho do vetor codificado
    unsigned char* buffer = (unsigned char*)calloc(countRead, sizeof(unsigned char));//alocacao do vetor
    codificado
    fread(buffer, sizeof(unsigned char), countRead, arquivo);//leitura do vetor codificado
    bool* binVet = intToBin(buffer, countRead);//Conversao para binario
    imprimeSaida(nomeSaida, binVet, countRead*8 ,tabela,sizeTab, charQnt);//decodifica o codigo binario
    utilizando a tabela e salva no arquivo de saida

    //liberacao da memoria alocada
    freeTabela(tabela,sizeTab);
    free(binVet);
    free(buffer);
    fclose(arquivo);//fechamento de arquivo
}
```

a função *unpack* é responsável por descompactar um arquivo de texto compactado. Seus parâmetros são *arquivo*, que é um ponteiro para o arquivo compactado, e *nomeSaida*, que é o nome que o arquivo de saída irá possuir.

algumas variáveis são criadas, para pegar certas informações que foram armazenadas no arquivo compactado, como explicado durante a explicação da função *pack*, são essas *countRead*, *sizeTab*, *sizeTree*, *charQnt*. A função *pack* então lê no arquivo compactado todas essas variáveis, e então lê a tabela de conversão e a salva na memória. Esse processo pode ser visto na linha 9, e ocorre por meio de um loop, e a cada posição da tabela, a função *intToBin* é chamada, para converter o valor compactado em um byte, para o código binário correspondente, e escreve então na tabela esse código, no campo de seu caractere correspondente. Ao fim do loop, a tabela foi salva na memória.

Em seguida, cada caractere do arquivo comprimido é lido, a partir da tabela. Isso é feito utilizando a lógica de prefixo. Cada código que representa cada caractere, nunca é prefixo do código de nenhum outro caractere.

os caracteres um a um, são lidos, convertidos pela função *intToBin* para seu código correspondente em binário, e esse código é escrito num vetor chamado *binVet*. Esse vetor então é convertido, pela função *imprimeSaida*, que decodifica os códigos escritos em *binVet*, e escreve no arquivo de saída descompactado.

4.7 Imprime Saída

```
1  void imprimeSaida(char* nomeSaida, bool* binVet, int size, tab* tabela,int sizeTab, long long charQnt){
2      FILE* newArquivo = fopen(nomeSaida, "w");//realiza a criacao do arquivo de saida
3      int i, ini = 0,k = 0;
4      for(i = 0; i < size; i++){//percorre o vetor de codigos de forma sequencial
5          int pos = verificaInLista(binVet, ini, i+1, tabela,sizeTab);//verifica se o subcodigo esta associado
6                                                                      //com algum caractere da tabela
7          if(pos != -1) {//Se for encontrado um caractere, ele eh escrito no arquivo de saida, e a posicao
       inicial
8                         //eh atualizada
9              fwrite(&tabela[pos].caractere,sizeof(char),1,newArquivo);
10             k++;
11             ini = i+1;
12             if (k == charQnt){//finaliza a funcao se a quantidade de caracteres presente no texto
13                                //de entrada for igual a quantidade escrita ate este ponto
14                 break;
15             }
16         }
17     }
18     fclose(newArquivo);//fecha o arquivo de saida
19 }
```

A função *imprimeSaida*, imprime num arquivo de saída o arquivo comprimido, descompactado-o no processo, a partir de um vetor *binVet*, criado na função *unpack*. Seus parâmetros de entrada são todos recebidos ou criados pela função *unpack*, portanto é desnecessário repetir cada um deles.

A função cria um ponteiro de arquivo chamado *newArquivi*, que irá criar um arquivo caso não existente, e irá por meio de um loop, que faz iterações igual ao tamanho do vetor *binVet* passado como parâmetro, decodificar o código, por meio da lógica de prefixo já descrita na função *unpack*.

A cada iteração, é verificado na lista se o código atual a ser verificado não é a representação compactada de nenhum caractere presente na tabela, caso seja verdade, o caractere correspondente será escrito no arquivo de saída, e do contrário, a próxima iteração acontece, até que o processo acabe. Como nenhum código de caractere é prefixo de outro, nunca ocorrerá a situação em que é escrito o pedaço do código de um caractere. Caso a quantidade de caracteres escritos no arquivo de saída chegar a ser a mesma quantidade de *charQnt*, o loop é interrompido, isso para corrigir a quantidade de bits extras que pode ter sido alocada durante a compactação.

4.8 Imprime Tabela

```
1  void printTabela(tab* tabela, int sizeTab) {
```

```
    //Percorre a tabela e imprimindo os caracteres e seus respectivos c digos
    for (int i = 0; i < sizeTab; i++) {
        if (tabela[i].size) {
            printf("Caractere : %c - C digo:  ", tabela[i].caractere);
            for (int j = 0; j < (tabela[i].size); j++) printf("%i ", tabela[i].vet[j]);//Percorre o vetor
                                //booleano e o imprime
            printf("\n");
        }
    }
}
```

A função *printTabela* simplesmente imprime a tabela de conversão dos caracteres. Ela recebe como parâmetro a tabela *tabela* a ser impressa, e o tamanho *sizeTab* dessa tabela. Por meio de um loop, cada campo dessa tabela é impresso.

4.9 Verifica Na lista

```
int verificaInLista(bool* binVet, int ini, int fim, tab* tabela, int sizeTab){
    for(int i = 0; i < sizeTab; i++){//Percorre toda a tabela buscando o codigo do vetor a partir dos indices
                                    //fornecidos
        if(tabela[i].size == fim-ini){//Se o tamanho dos codigos forem iguais seus campos sao comparados
            for(int j = ini; j < fim; j++){
                if(binVet[j] != tabela[i].vet[j-ini]) {
                    break; //Para a comparacao se encontrar um campo diferente
                }
                if(j==fim-1 && binVet[j] == tabela[i].vet[j-ini]) {
                    return i;//retorna a posicao do caractere na tabela se todos os campos forem iguais
                }
            }
        }
    }
    return -1;//retorna -1 se o codigo nao for encontrado
}
```

Esta função busca um código de um caractere na tabela de códigos, retornando a posição do caractere na tabela se o mesmo for encontrado e -1 caso contrário. Como os códigos são salvos de forma sequencial em apenas um vetor, é necessário realizar uma busca na tabela de cada sub código presente no vetor, para isso, são necessários os parâmetros de inicio e fim do vetor, onde todos os campos do sub código entre os índices é comparado com todos campos dos códigos da tabela que possuem o mesmo tamanho. Se todos os campos forem iguais, o caractere foi encontrado, e, se toda a tabela foi percorrida, significa que o sub código não representa um caractere.

5 Funções de suporte

Nesta seção, serão apresentadas cada uma das funções auxiliares utilizadas para a implementação do algoritmo de Huffman.

5.1 Cria Tabela

```
tab* criaTabela(int sizeTree, int sizeList){

    tab* tabela =(tab*) calloc(sizeList,sizeof(tab));//Aloca a memoria necessaria
    //Aloca o vetor de booleanos para armazenar o codigo de cada
    //caractere se a alocacao da tabela for bem sucedida
    if(tabela){
        for(int i=0;i<sizeList;i++){
            tabela[i].vet =(bool*) calloc(sizeTree,sizeof(bool));
        }
    }
    return tabela;
}
```

Essa função tem como objetivo criar uma tabela vazia, com os espaços necessários, para serem preenchidos com os caracteres e seus respectivos códigos de compactação, se tornando assim uma tabela de conversão dos caracteres para sua representação compactada em binário.

Essa função recebe como parâmetro *sizeTree* que representa o tamanho da árvore de Huffman, e *sizeList*, que é o tamanho da lista utilizada para criação da árvore, e ao mesmo tempo será o tamanho de uma tabela. primeiramente, são alocados *sizeList* espaços para essa tabela, em que cada espaço possui o tamanho de um dado do tipo *tab*. Em seguida, por meio de um loop, são alocados os *sizeTree* espaços , em que cada um desses espaços que serão preenchidos com os códigos de conversão dos caracteres respectivos. Isso porque *sizeTree* representa a altura da árvore, portanto a máxima quantidade de bits que um caractere precisará para ser representado corresponde ao valor de *sizeTree*.

Por fim, a tabela alocada com campos vazios é retornada.

5.2 Busca Tabela

```
int searchTab(tab* tabela, char caractere, int tabSize){
    //Realiza a busca em toda tabela
    for(int i = 0; i < tabSize; i++){
        if(tabela[i].caractere == caractere) return i;//retorna a posicao do caractere na tabela se o mesmo
    for encontrado
    }
    return -1;//retorna -1 se o caractere nao for encontrado
}
```

A função *searchTab* recebe como parâmetro a tabela *tabela*, que corresponde a tabela de conversão, um caractere *caractere*, e o tamanho da tabela *tabSize*, e simplesmente busca um determinado caractere na tabela e retorna a posição na qual esse caractere se encontra, e -1 caso esse caractere não for encontrado.

5.3 Limpa Tabela

```
void freeTabela(tab* tabela, int sizeTab){
    //libera a memoria alocada em cada caminho
    for(int i = 0; i < sizeTab; i++) free(tabela[i].vet);
    free(tabela);//libera a memoria alocada na tabela
}
```

Essa função recebe como parâmetros a tabela de conversão *tabela*. e o tamanho *sizeTab* dessa tabela, e somente limpa os espaços de memória alocados para essa tabela, ou seja, realiza a operação de destruição da tabela.

5.4 Frequencia No

```
int frequenciaNo(BSTNode* root, bool* vet){
    int i = 0;
    //realiza a busca pelo caractere utilizando o caminho passado
    while(root->left && root->right){
        if(vet[i] == 0) root = root->left;
        else root = root->right;
        i++;
    }
    return root->info->frequencia;//retorna a frequencia do caractere
}
```

Essa função recebe como parâmetro um vetor *vet*, que representa o código de conversão de um caractere, e a raiz *root*, da árvore de huffman. A função vai então percorrendo a árvore de huffman por meio do código do caractere, e ao fim do vetor que representa esse código, ela encontra o caractere, e retorna sua frequência.

5.5 CountSizeOut

```
long long countSizeOut(tab* tabela, int sizeTab, BSTNode* root) {
    long long countSize = 0;
    for (int i = 0; i < sizeTab; i++) {
        countSize += tabela[i].size * frequenciaNo(root, tabela[i].vet);
    }
    if(countSize%8 > 0){
        countSize = countSize/8;
        countSize++;
    }
    else countSize = countSize/8;
    return countSize;
}
```

Essa função é responsável por contar a quantidade de bytes necessários para escrever todo o arquivo de entrada comprimido.

Os parâmetros de entrada são a tabela de conversão *tabela*, o tamanho *sizeTab*, dessa tabela, e a raiz da árvore *root*.

A função vai acessando na tabela caractere a caractere, e somando o número de bits que o código de cada caractere possui, multiplicado de sua frequência, fornecendo assim no final do processo a quantidade em bits necessários para escrever o arquivo de entrada comprimido, porém esse valor precisa ser dividido por 8, pois esse bits serão alocados em bytes, e caso alguma quantidade fique faltando, devido ao fato de provavelmente esse valor contado não ser divisível por 8, é adicionado 1 a essa contagem para que não fique faltando espaço para esses bits. A função então retorna countSize, que é esse valor.

5.6 Inteiro para Binário

```
bool* intToBin (unsigned char vet[], int size){
    bool* binVet = (bool*) calloc(8*size,sizeof(bool)); //Aloca a memoria necessaria para realizar a
    conversao
    int i=0, j=7;
    for(i=0;i<size;i++){//Percorre o vetor realizando o processo de divisoes sucessivas
                        //adicionando os resultados no vetor de booleanos
        int num = vet[i];
        if(i==0) j=7;
        else j =7*(i+1)+i;
        while(num > 0){
            binVet[j] = num % 2;
            num = num/2;
            j--;
        }
    }
    return binVet;//retorna o vetor de booleandos contendo o valor comprimido convertido em binario
}
```

A função *intToBin* recebe como parâmetro um vetor *vet* que representa o vetor a ser convertido, e o tamanho *size* desse vetor, e então salva num vetor de saída, a conversão correspondente do valor em decimal, em cada posição do vetor *vet*, num vetor *binVet*, de saída, em binário.

5.7 bitToBit

```
void bitToBit(bool vet[], int size, int* posicao, unsigned char saida[]) {
    unsigned char A;
    int posbyte = 0, posbit = 0;
    for (int i = 0; i < size; i++) {
        posbyte = (*posicao) / 8;
        posbit = (*posicao) % 8;

        A = vet[i];
        A = A << 8 - posbit - 1;
        saida[posbyte] = saida[posbyte] | A;
```

```
11          (*posicao)++;
12      }
13  }
```

Essa função tem como objetivo escrever de forma compactada em um vetor de saída, um caractere do arquivo de entrada. Ela recebe como parâmetro um vetor booleano *vet*[], que representa o código do caractere a ser compactado, o tamanho desse vetor *size*, a posição em bits do vetor de saída *posicao*, e o vetor de saída *saida*. que é do tipo *unsigned char*. Em seguida, algumas variáveis são declaradas, A variável *A* é responsável por receber um bit do caractere a ser recebido. As variável posição, representa a primeira posição do vetor *saída*, em bits, que ainda não foi escrita, a *posbyte* representa o byte atual do vetor saída, que está sendo escrito, e a *posbit* representa o último bit ainda não escrito, do byte representado por *posbyte*. Essa função vai na posição primeira posição disponível do vetor *saída*, ou seja, a primeira posição que ainda não foi escrita, e escreve o código do caractere que está sendo inserido, no vetor saída, bit por bit. a variável *A* recebe um bit do código do caractere que está sendo escrito, e então, por meio de uma operação OU, não curto circuitada, escreve o código do caractere na posição bit *posbit* de *posbyte*, no vetor de saída. Para que isso ocorra adequadamente, é necessário que o bit atual sendo escrito seja deslocado até a posição de bit adequada de *A*, que então por meio do ou não curto circuitado, escreve esse bit no vetor saída. Essa operação pode ser vista na linha 9.

6 Resultados e conclusões

Como mencionado anteriormente, o algoritmo de Huffman tem como objetivo reduzir o tamanho de um arquivo de texto, aumentando a eficiência de armazenamento de um dispositivo. E como esperado, os resultados foram de fato, arquivos compactados menores que os arquivos de entrado, foi interessante perceber que os caracteres de maior frequência possuíam sempre um código de compactação mais curto, demonstrando a eficiência do algoritmo.

As maior dificuldade durante a implementação, foi a conversão bit por bit, dos caracteres da entrada, para os caracteres de saída codificados. Um dos pontos que foram considerados importantes na implementação do trabalho por completo, foi avaliar a liberação de memória de toda a estrutura. Para isso, foi utilizado o Valgrind, uma ferramenta que identifica vazamentos de memória durante e após o uso do programa. E, após realizar vários testes, foi comprovado que esta implementação não possui nenhum vazamento de memória. Além disso, o programa foi testado com uma série de arquivos de entrada de variados tamanhos, incluindo arquivos vazios, arquivos com somente um caractere se repetindo várias vezes, arquivos com caracteres de vários idiomas diferentes e arquivos grandes, como um dicionário que foi utilizado para testes.

No geral, muito foi aprendido, principalmente com com relação a algoritmos gulosos, e como os as informações são armazenadas na arquitetura de computadores atuais.

7 Bibliografia

Aulas síncronas; conteúdo da disciplina de Estrutura de Dados II.

8 Apêndices

8.1 Apêndice 1: Makefile

```
1
2 SYSTEM   = x86-64_linux
3 LIBFORMAT = static_pic
```

```
4
5 OPTIONS = -O3
6 COMP = g++ -c
7
8
9 # ------------------------------------------------------------------------
10 # Compiler selection
11 # ------------------------------------------------------------------------
12 CCC = gcc
13 SRCDIR =
14 OBJDIR = obj
15
16 OBJ_PRJ =
17
18 CCFLAGS = $(CCOPT)
19
20 OBJ_LIB =
21
22 Projeto :  $(OBJ_PRJ) $(OBJ_LIB)
23   $(CCC) $(CCFLAGS) -o programa.exe main.c BST.c Huffman.c $(OBJ_PRJ) $(OBJ_LIB)
24
25 clean:
26   rm -f *.o
27   rm -f *~
28   rm Projeto
```

8.2 Apêndice 2: Main

```
1 //Uarlley do Nascimento Amorim - 2018205346
2 //Ualaci dos Anjos Junior - 2017102015
3
4 #include "Huffman.h"
5 #include <string.h>
6
7 int main() {
8     char file[200];
9     char out[200];
10     while(1){
11         puts(" ------------------------------");
12         puts("|  1 - Compactar arquivo     |");
13         puts("|  2 - Descompactar arquivo  |");
14         puts("|  3 - Sair                  |");
15         puts(" ------------------------------");
16         int op;
17         printf("Selecione uma opcao: ");
18         scanf("%d", &op);
19         getchar();
20         if(op == 1){
21             int SIZELIST = 0;
22             long long charQnt = 0;
23             printf("Digite o nome do arquivo a ser compactado (exemplo.txt): ");
24             fgets(file, 200, stdin);
25             file[strlen(file)-1] = 0;
26             system("clear");
27             if(strcmp("sair", file) == 0) exit(1);
28             Node* head = leArquivo(file,&SIZELIST, &charQnt);
29             if(head){
30                 printf("\nOcorrencia dos caracteres no arquivo:");
31                 imprimeOcorrencias(head);
32                 int countSaida = 0;
33                 printf("Pressione enter para continuar...");
34                 getchar();
35                 system("clear");
36                 BSTNode* root = huffman(head, SIZELIST);
37
38                 printf("\nArvore gerada pelo algoritmo de Huffman: ");
39                 puts("");
40                 imprimeCodHuffman(root);
41
```

```
42                 printf("Pressione enter para continuar...");
43                 getchar();
44                 system("clear");
45 
46                 int sizeOut = heightBST(root);
47                 tab *tabela = criaTabela(sizeOut, SIZELIST);
48                 bool *vet =(bool*) calloc(sizeOut,sizeof(bool));
49                 int size = 0, sizeTab = 0;
50 
51                 if(SIZELIST == 1){
52                     tabela[0].caractere = root->info->caractere;
53                     tabela[0].size = 1;
54                     tabela[0].vet[0] = 0;
55                 }
56                 else writeTab(root, tabela, vet, &size, &sizeTab);
57 
58                 printf("\nCodigo gerado pelo algoritmo de Huffman: ");
59                 puts("");
60                 printTabela(tabela,SIZELIST);
61                 puts("");
62                 long long countSizeO = countSizeOut(tabela, SIZELIST,root);
63                 printf("\nPressione enter para continuar...");
64                 getchar();
65                 system("clear");
66 
67                 printf("Digite o nome do arquivo de saida (exemplo.cpt): ");
68                 fgets(out, 200, stdin);
69                 out[strlen(out)-1] = 0;
70                 writeBin(tabela, file, out,countSizeO,SIZELIST,sizeOut, charQnt);
71                 destroyBST(root);
72                 freeTabela(tabela,SIZELIST);
73                 free(vet);
74                 printf("\nArquivo compactado!\nPressione enter para continuar...");
75                 getchar();
76                 system("clear");
77             }
78             else{
79                 printf("ERRO! Arquivo vazio.\n");
80                 printf("\nPressione enter para continuar...");
81                 getchar();
82                 system("clear");
83             }
84         }
85         else if(op == 2){
86             printf("Digite o nome do arquivo a ser compactado (exemplo.cpt): ");
87             fgets(out, 200, stdin);
88             out[strlen(out)-1] = 0;
89             FILE* arquivo = fopen(out,"rb");
90             if(!arquivo) printf("Arquivo inexistente!\n");
91             else {
92                 printf("Digite o nome do arquivo de saida (exemplo.txt): ");
93                 char nomeSaida[200];
94                 fgets(nomeSaida,200, stdin);
95                 nomeSaida[strlen(nomeSaida)-1] = 0;
96                 unpack(arquivo, nomeSaida);
97                 printf("Arquivo descompactado!\nPressione enter para continuar...");
98                 getchar();
99                 system("clear");
100            }
101
102        }
103        else {
104            break;
105        }
106
107    }
108}
```