UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO - UFES
CENTRO UNIVERSITÁRIO NORTE DO ESPÍRITO SANTO - CEUNES
CIÊNCIA DA COMPUTAÇÃO

JOÃO PAULO SOUZA FERRETE
RAMON PEZZIN TON

RELATÓRIO

IMPLEMENTAÇÃO DO ALGORITMO DE HUFFMAN

SÃO MATEUS
11 DE MAIO DE 2021

INTRODUÇÃO

Neste relatório temos como objetivo a implementação do algoritmo de compressão de Huffman, utilizado para o tratamento de strings. Desta forma, neste relatório, serão apresentadas as funções feitas e as lógicas utilizadas para a implementação e funcionamento do programa, bem como os conceitos aplicados e como funcionam cada função. A codificação foi feita na linguagem C, com a compilação feita a partir de um arquivo Makefile.

IMPLEMENTAÇÕES

Arquivo '*huffman.h*':

Este arquivo é o arquivo que contém a inclusão das bibliotecas necessárias para a execução do programa, bem como a definição das estruturas que serão utilizadas.

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

typedef struct tree{
   struct tree *esq;
   struct tree *dir;
   long long freq;
   char caractere;
} Arvore;

typedef struct node{
   long long freq;
   char caractere;
   Arvore * tree;
   int * codigo;
   int tamCod;
} No;
```

Em primeiro lugar temos a estrutura de árvore, que será utilizada no algoritmo de Huffman para montar os códigos binários. Essa consiste em um ponteiro para a subárvore direita e um ponteiro para a subárvore esquerda, um elemento para guardar a quantidade de vezes que o caractere aparece no texto, e o caractere que aquele nó representa.

Na estrutura de *No*, que é utilizada para a criação do array, temos uma variável para guardar a frequência do caractere, o caractere, um ponteiro para a árvore, que será utilizado na hora de montar a árvore, um ponteiro para *int*, que armazenará o código binário para cada caractere, e o tamanho do código binário.

Neste arquivo, também, temos o cabeçalho de todas as funções do arquivo *huffman.c*, que serão descritas a seguir.

Arquivo "*huffman.c*"

Este arquivo contém a implementação, propriamente dita, das funções utilizadas para o desenvolvimento do programa.

```
void * mallocSafe(size_t tam){

   void * a = malloc(tam);
   if(!a){
       printf("ERRO: SEM MEMORIA!\n");
       exit(1);
   }
   return a;
}
```

A função *mallocSafe* é responsável por alocar um espaço de memória. Inicialmente ela aloca um espaço de memória e verifica se o espaço foi alocado com sucesso. Em caso positivo, a função retorna um ponteiro para o espaço alocado. Mas, caso contrário, é impresso uma mensagem de erro e o programa é encerrado.

```
void imprimeVetor(No * vet, long long tam){

   for(long long i=0; i<tam; i++){
       printf(" [%c : %ld] ", vet[i].caractere, vet[i].freq);
   }
   printf("\n");
}
```

Esta função é responsável por imprimir o caractere e a quantidade de incidências desse caractere no texto. Ela recebe por parâmetro um vetor e seu tamanho, e consiste, basicamente, em um laço que percorre todo o vetor imprimindo as informações.

```
void imprimeVetorCod(No * vet, long long tam){

   for(long long i=0; i<tam; i++){
       printf(" [%c : %ld : ", vet[i].caractere, vet[i].freq);
       for(int j=0; j<vet[i].tamCod; j++){
           printf("%d", vet[i].codigo[j]);
       }
       printf("] ");
   }
   printf("\n");
}
```

De forma similar a função anterior, essa função também imprime todos os elementos do vetor, mas também imprime o código binário correspondente àquele caractere. Ela possui mais um laço, que percorre o vetor do código, imprimindo-o.

```
long long pai(long long i){
   return (long long)i/2;
}
long long esquerdo(long long i){
   return 2*i+1;
}
long long direito(long long i){
   return 2*i+2;
}
```

Essas funções, que serão utilizadas na função de criar um heap mínimo, consistem, basicamente, em calcular, e retornar, o índice correspondente ao pai, filho à esquerda e filho à direita, respectivamente, de um índice *i* passado por parâmetro.

```
void minHeapify (No* vet, long long i, long long tam){

   long long l, r, menor;
   No aux;
   menor=i;
   l=esquerdo(i);
   r=direito(i);
   if((l<tam) && (vet[l].freq<vet[menor].freq))menor=l;
   if((r<tam) && (vet[r].freq<vet[menor].freq)) menor=r;
   if(menor!=i){
       aux=vet[i];
       vet[i]=vet[menor];
       vet[menor]=aux;
       minHeapify(vet, menor, tam);
   }
}
```

Esta função é responsável por criar um heap mínimo a partir do elemento *i* passado por parâmetro. Ela recebe por parâmetro um ponteiro para o array, o tamanho do array e a posição inicial do array que deve ser considerado. Ela consiste em encontrar o menor elemento entre os filhos direito e esquerdo do elemento *i*, e permutá-los no array, fazendo com que ele contenha as características de um heap e, principalmente, que o menor elemento esteja no início do array.

```
void construirheapmin(No* vet, long long tam){

   long long i;
   for (i= (long long)(tam/2)-1; i>=0;i--){
       minHeapify(vet, i, tam);
   }
}
```

A função *construirheapmin* é responsável por chamar a função anterior para a criação do heap. Ela recebe por parâmetro um array e seu tamanho, e consiste em um laço para chamar a função *minHeapify* para a criação do heap.

```
No * insereFrequencia(char * nomeArquivo, long long *tam){

    FILE * arquivo = fopen(nomeArquivo, "r");
    char carac;
    long long i=0, j=0, n=4, verif;

    if(!arquivo){
        printf("Não foi possível abrir o arquivo!\n");
        exit(1);
    }
    No * vet = (No *) mallocSafe(sizeof(No)*n);
```

A função *insereFrequencia* é responsável pela criação do array que, inicialmente, irá conter os caracteres presentes no texto, e a quantidade de vezes em que eles aparecem. Ela recebe por parâmetro o nome do arquivo que contém os arquivos e um ponteiro para uma variável que irá conter o tamanho do vetor. Inicialmente, abrimos o arquivo para, apenas, leitura e verificamos se o arquivo pode ser lido. Caso negativo, o programa é encerrado, mas, caso positivo, o programa aloca um array, inicialmente com 4 elementos.

```
while(1){
        carac = (char)fgetc(arquivo);
        if(carac == EOF) break;

        if(i==0){
            vet[0].freq=1;
            vet[0].caractere = carac;
            vet[0].tree = NULL;
            i++;
        }
```

Então, é feito um laço, e pegamos um caractere do arquivo. Caso esse caractere seja o indicador de fim do arquivo, o laço é interrompido. Caso seja o primeiro caractere a ser pego, ele é inserido diretamente no array, com sua frequência inicial 1.

```
    else{
          for(j=0; j<i; j++){
              if(vet[j].caractere == carac) {
                  vet[j].freq++;
                  verif=1;
              }
          }
```

Caso não seja o primeiro elemento, então fazemos, em um for, uma busca para ver se

o caractere atual já está presente no array. Em caso positivo, aumentamos a frequência em uma unidade, e atribuímos o valor 1 à variável verif, indicando que aquela variável já está presente no array.

```
        if(!verif){
                  if(n==i){
                      vet = (No *) realloc(vet, sizeof(No)*(n+1));
                      n++;
                  }
                  vet[i].caractere = carac;
                  vet[i].freq=1;
                  vet[i].tree = NULL;
                  i++;
              }
```

Caso o elemento ainda não tenha sido inserido no array, primeiramente verificamos se ainda há espaço disponível no array. Caso não haja, então fazemos uma realocação de mais um campo no array, e atualizamos a variável de tamanho. Então, é feita a atribuição dos campos do array.

```
    *tam = n;
     fclose(arquivo);
     construirheapmin(vet, n);
     return vet;
}
```

Ao fim do laço, então, atribuímos à variável de tamanho, cujo ponteiro foi passado por parâmetro, o tamanho da lista, fechamos o arquivo, e construímos o heap a partir do vetor. E, então, o retornamos.

```
Arvore * criaArvore(char carac, long long freq){

   Arvore * t = (Arvore *) mallocSafe(sizeof(Arvore));
   if(!t) return NULL;
   t->caractere = carac;
   t->freq = freq;
   t->dir = t->esq = NULL;
   return t;
}
```

Esta função é responsável por retornar um nó para a árvore, com os campos de caractere e de frequência passados por parâmetro. Inicialmente, alocamos o espaço para o nó e verificamos se o nó foi alocado. Em caso positivo, é feito o campo de caractere da árvore receber o caractere passado por parâmetro, e a mesma coisa com o campo de frequência. Inicializamos os ponteiros com *NULL*, e, então, retornamos o ponteiro para o elemento.

```
No * copiaVet(No * vet, long long tam){

   No * vetor = (No*)mallocSafe(tam*sizeof(No));
   for(int i=0; i<tam; i++){
       vetor[i].caractere = vet[i].caractere;
       vetor[i].freq = vet[i].freq;
       vetor[i].tree = vet[i].tree;
   }
   return vetor;
}
```

A função *copiaVet* é responsável por copiar um vetor passado por parâmetro. Inicialmente ela aloca o espaço necessário para o vetor, e então é feito um laço onde é copiado cada elemento do vetor de origem para o novo vetor. Então retornamos o ponteiro para esse elemento.

```
Arvore * algoritmoHuffman(No *vetor, long long tam){

   long long i, n=tam;
   Arvore *x, *y, *z;

   No* vet = copiaVet(vetor, tam);

       for(i=0; i<n-1; i++){
           z = criaArvore(' ', 0);
```

Esta função é responsável por executar o algoritmo de Huffman, propriamente dito, retornando um ponteiro para a raiz da árvore a partir do vetor com caracteres e suas frequências, passado por parâmetro. Inicialmente, é feita uma cópia do vetor de elementos, já que durante a execução ele será destruído. Então, temos o início do laço que irá ocorrer até um elemento antes do tamanho do vetor. No laço, primeiramente, criamos um nó para a árvore. Como apenas os nós folha possuem caracteres, esse nó recebe o caractere ‘ ‘, e, inicialmente, a frequência 0, e a variável *z* recebe esse elemento.

```
if(!vet[0].tree) x = criaArvore(vet[0].caractere, vet[0].freq);
else x = vet[0].tree;
z->esq = x;

vet[0] = vet[tam-1];
tam--;
vet = (No *) realloc(vet, tam*sizeof(No));
construirheapmin(vet, tam);
```

Então, já que o vetor já é um heap mínimo, temos que o primeiro elemento é o de menor frequência, então verificamos se esse elemento é um nó folha ou se ele é um nó

interno da árvore. Caso ele seja um nó folha, então alocamos um novo nó de árvore para armazenar os elementos dele, passando por parâmetro o caractere e a frequência, e fazemos *x* receber esse ponteiro. Já, se o elemento for um nó interno, fazemos *x* apontar para esse nó, então x passa a ser filho à esquerda de *z*. Então, como um elemento foi retirado do array, diminuímos o tamanho dele em 1 unidade, e refazemos o heap.

```
if(!vet[0].tree) y = criaArvore(vet[0].caractere, vet[0].freq);
else y = vet[0].tree;
z->dir = y;
z->freq = x->freq+y->freq;

vet[0].caractere = z->caractere;
vet[0].freq = z->freq;
vet[0].tree = z;
construirheapmin(vet, tam);
```

De forma similar, verificamos se o novo menor elemento é um nó interno ou um nó folha, e o inserimos na árvore. E fazemos o campo frequência de *z* receber a soma da frequência de seus filhos. E então re-inserimos *z* no array, na primeira posição, já que ela foi removida do array. E, então, o heap é refeito. Ao fim do laço, liberamos o espaço utilizado pelo vetor e retornamos , que aponta para a raiz da árvore.

```
int altura (Arvore * ptr){

   long long r, l;
   if(!ptr ) return 0;
   r=1+altura(ptr->dir);
   l=1+altura(ptr->esq);
   if(r>l) return r;
   return l;
}
```

A função *altura* é responsável por retornar a altura da árvore gerada pelo algoritmo de Huffman. Ela recebe um ponteiro para a raiz da árvore, e, de maneira recursiva, calcula e retorna o valor da altura da árvore.

```
void invertelista(int* vet, long long tam){
   int aux;
   int n=tam-1;
   for(int i=0; i<tam/2; i++){
       aux = vet[i];
       vet[i]= vet[n-i];
       vet[n-i] = aux;
   }
}
```

Esta função é responsável por inverter os elementos de um array. Ela consiste em um laço que vai até o meio da lista, trocando as posições entre os elementos, de forma que, no final do processo, ela esteja invertida.

```
void imprimeArvore(Arvore * ptr){

   if(!ptr) return;
   int i, nivel = altura(ptr);

   if(ptr){
       imprimeArvore(ptr->esq);

       for(i=0; i<nivel; i++) printf("\t");
       printf("[%lld : %c]\n", ptr->freq, ptr->caractere);

       imprimeArvore(ptr->dir);
   }
}
```

A função de impressão de árvore é responsável por imprimir toda a estrutura da árvore e recebe por parâmetro um ponteiro para a raiz da mesma. A impressão é feita a partir do número de tabulações correspondentes ao nível. Primeiramente, a variável '*nivel*' recebe a altura da árvore. Então, chamamos a função para que, primeiramente, seja impresso a subárvore esquerda, para que a impressão seja simétrica, e então imprimimos, de acordo com o nível da árvore, a tabulação, para facilitar a visualização, então imprimimos a frequência do caractere, seguido do próprio caractere, e chamamos a função de impressão para a subárvore direita.

```
int codigo(char carac, int *tamCod, int * cod, Arvore * pt, int
tamMax){

   if(!pt) return 0;
   else if (pt->caractere == carac && !pt->dir && !pt->esq) return
1;
   int esq=0, dir=0;

   esq = codigo(carac, tamCod, cod, pt->esq, tamMax);
   dir = codigo(carac, tamCod, cod, pt->dir, tamMax);
```

A função código é a função responsável por inserir o código binário correspondente de um caractere em um array. Ela funciona de maneira recursiva, e recebe por parâmetro o caractere correspondente, um ponteiro para a variável que irá guardar o tamanho do array, um ponteiro para o array já alocado, um ponteiro para a raiz da árvore e um inteiro que representa o tamanho já alocado do vetor de código.

Inicialmente temos os casos base, que se o ponteiro para a árvore for nulo, então o elemento não foi encontrado, retornando 0. Caso seja um nó folha e o caractere seja igual ao buscado, então retorna 1, indicando que foi encontrado. Assim, caso nenhum dos dois casos sejam satisfeitos, chamamos a função recursivamente para a subárvore esquerda e direita.

```
if(esq){
    if(*tamCod>tamMax) {
        tamMax+=20;
        cod = (int *) realloc((cod), sizeof(int)*(tamMax));
    }
    cod[*tamCod] = 0;
    *tamCod+=1;
    return 1;
}
```

Se *esq* for 1, então o elemento foi encontrado pela esquerda. Então, inicialmente, verificamos se ainda existe espaço no vetor com os códigos. Caso não haja, então realocamos o vetor com mais 20 espaços. Então, como o elemento foi encontrado pela esquerda, adicionamos 0 no vetor, adicionamos 1 ao seu tamanho e retornamos 1, informando que o caractere foi encontrado.

```
 else if (dir){
    if(*tamCod>tamMax) {
        tamMax+=20;
        cod = (int *) realloc((cod), sizeof(int)*(tamMax));
    }
    cod[*tamCod] = 1;
    *tamCod+=1;
    return 1;
}
else return 0;
}
```

Se o elemento for encontrado pela direita, o procedimento é exatamente o mesmo, exceto pelo fato de que o 1 é inserido no vetor. Caso o elemento não seja encontrado, retornamos 0.

```
No * pegaCod(No* vetor, long long n, Arvore* raiz){

    long long i;
    char carac;

    No * vet = copiaVet(vetor, n);
```

A função *pegaCod* é responsável por criar um vetor que contenha o campo de códigos binários para cada caractere. Ela recebe por parâmetro o vetor com a quantidade de

ocorrência dos caracteres, o tamanho dele, e um ponteiro para a raiz da árvore. Inicialmente é feita uma cópia do vetor de elementos, para que se possa guardar os códigos, e não aumentar a quantidade de elementos a serem gravadas no arquivo.

```
    for(i=0; i<n; i++){
        carac = vet[i].caractere;
        int *cod = (int *) mallocSafe(sizeof(int)*20);
        int tamCod=0;

        codigo(carac, &tamCod, cod, raiz, 19);

        invertelista(cod, tamCod);
        vet[i].codigo = cod;
        vet[i].tamCod = tamCod;
    }
    return vet;
}
```

Então, é feito um laço que percorre todo o vetor e, para cada vetor, alocamos um novo array que irá receber o código binário, e então chamamos a função *codigo* para inserir o código no vetor. Ao fim da inclusão, invertemos a lista, já que na função *codigo* a inclusão é feita de baixo para cima. Então, atribuímos aos campos do vetor o ponteiro para o vetor com o código e o tamanho desse vetor. Ao fim do laço retornamos o vetor final.

```
long long calculaTamVet(No *vet, long long n){

    long long soma=0;

    for(long long i=0; i<n; i++){
        soma+=(vet[i].freq * vet[i].tamCod);
    }
    if(soma%8==0)return soma/8;
    return (soma/8)+1;
}
```

Esta função é responsável por calcular o tamanho que deverá ser alocado para o vetor de saída, que terá os bits manipulados. Para saber a quantidade de bits necessária, é feito um somatório do tamanho dos códigos para cada caractere multiplicado pela quantidade de ocorrência dele. Assim, como só é possível alocar uma quantidade de bytes, se o resto da divisão da soma por 8 for zero, então podemos alocar soma/8 bytes. Já, se for diferente, temos que alocar mais 1 byte para os bits restantes.

```
long long calculaBit(No *vet, long long n){

   long long soma=0;

   for(long long i=0; i<n; i++){
       soma+=(vet[i].freq * vet[i].tamCod);
   }
   return soma;
}
```

De forma similar a anterior, essa função calcula a quantidade de bits que serão manipulados.

```
unsigned char * vetorSaida(No* vet, long long n, long long *tamBit,
char * nomeArq){

   *tamBit = calculaBit(vet, n);
   long long tamVet = calculaTamVet(vet, n);
   unsigned char *vetSaida = (unsigned char *) mallocSafe(tamVet);
   memset(vetSaida, 0, tamVet);

   unsigned char aux;
   int pos=0, desl=0, posbyte=0, posbit=0;

   FILE * arq = fopen(nomeArq, "r");
```

Essa função é responsável por alocar e fazer a manipulação bit a bit do vetor *unsigned char* que irá guardar o código de Huffman no arquivo compactado. Ela recebe por parâmetro um ponteiro para o vetor que contém os caracteres, as frequências e os códigos binários, o tamanho dele, um ponteiro para uma variável que irá guardar a quantidade de bits que serão guardados e o nome do arquivo que contém o texto a ser compactado.

Inicialmente é feito o cálculo da quantidade de bits e de bytes que serão necessários para guardar o arquivo, e, com esse valor, o vetor de saída é alocado. Então todos os bits do vetor são inicializados com 0 e abrimos o arquivo para leitura.

```
 while(1){
    char carac = (char)fgetc(arq);
    if(carac == EOF) break;

    int i=0;
    for(i=0; vet[i].caractere!=carac && i<n ; i++);
    for (int j=0; j<vet[i].tamCod && i<n; j++){
        posbyte = pos/8;
        posbit = pos%8;
        desl = 7-posbit;
```

```
            aux = vet[i].codigo[j];
            aux = aux<<desl;
            vetSaida[posbyte] = vetSaida[posbyte] | aux;

            pos++;
        }
    }
    fclose(arq);
    return vetSaida;
}
```

Então, em um laço, pegamos cada caractere do arquivo a ser compactado, para sabermos a ordem em que os elementos deverão ser guardados. Se o caractere lido representar o fim do arquivo, então o laço é interrompido. Caso seja um caractere válido, então é feito um laço para encontrar a posição do vetor em que aquele caractere se encontra. Depois que ele é encontrado, fazemos um outro laço, que corresponde ao tamanho do vetor que contém o código, para manipular cada bit e inseri-lo no vetor de saída. Ao fim da leitura, fechamos o arquivo e retornamos o vetor de saída.

```
void binCode(unsigned char * vet, long long tam, Arvore *raiz, char
* nomeArq){

    unsigned char aux;
    int pos=0, desl=0, posbyte=0, posbit=0;

    Arvore * auxT = raiz;

    FILE * arq = fopen(nomeArq, "w");
```

A função *binCode* é responsável por fazer a descompressão do arquivo e remontar o arquivo original. Para isso, ela recebe por parâmetro o vetor de saída com os bits alterados, a quantidade de bits inseridos no vetor, a raiz da árvore criada pelo algoritmo de Huffman e o nome do arquivo de saída. Inicialmente, o arquivo é aberto para escrita.

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

        posbyte = pos/8;
        posbit = pos%8;
        desl = 7-posbit;

        aux = 1;
        aux = aux<<desl;
        aux= vet[posbyte]&aux;
        aux = aux>>desl;

        pos++;
```

Então, é feito um laço para cada bit que precisa ser decodificado. Assim, é feita a manipulação de cada bit, de forma que, no fim, *aux* esteja com o bit original gravado.

```
        if(aux==0) auxT = auxT->esq;
          else if (aux==1) auxT = auxT->dir;
          if(!auxT->esq && !auxT->dir){
              fprintf(arq,"%c", auxT->caractere);
              auxT=raiz;
          }
    }
    fclose(arq);
}
```

Se o bit recuperado for *0*, então o elemento buscado está na subárvore esquerda, mas se for *1*, então está na subárvore direita, assim, atualizamos a variável *auxT*, que aponta para o nó atual da árvore. Então é feita a verificação de se o nó atual é um nó folha. Em caso positivo, então o caractere que ele guarda é escrito no arquivo e *auxT* volta a apontar para a raiz da árvore. Ao fim do laço o arquivo é fechado e a função se encerra.

```
void descomprimir(char * nomeArq){

    FILE * arq = fopen(nomeArq, "rb");
    if(!arq){
         printf("Não foi possível abrir o arquivo!\n");
        return;
    }

    long long tamBin, tamVetor;
    long long tamVetSaida;
    No* vetor;
    unsigned char *vetSaida;
```

A função *descomprimir* é responsável por descompactar um arquivo binário de volta para um arquivo de texto idêntico ao original. Ela recebe por parâmetro o nome do arquivo a ser descompactado. No início da função abrimos o arquivo para leitura binária e verificamos se ele foi aberto com sucesso. Em caso negativo é impresso uma mensagem de erro e a função se encerra.

```
    fread(&tamBin, sizeof(long long), 1, arq);
    fread(&tamVetor, sizeof(long long), 1, arq);
    vetor = (No *) mallocSafe(sizeof(No)* tamVetor);
    fread(vetor, sizeof(No), tamVetor, arq);

    fread(&tamVetSaida, sizeof(long long), 1, arq);
    vetSaida = (unsigned char *) mallocSafe(tamVetSaida);
```

```
    memset(vetSaida, 0, tamVetSaida);
    fread(vetSaida, sizeof(unsigned char), tamVetSaida, arq);

    fclose(arq);
```

Se a abertura do arquivo for bem sucedida, é lido a quantidade de bits gravada, e o tamanho do vetor com as frequências. Então alocamos o espaço necessário para esse vetor e lemos seu conteúdo. Após é lido o tamanho do vetor de saída, alocamos o espaço para ele e inicializamos todos os seus campos com *0*. Então o lemos e fechamos o arquivo.

```
    Arvore * raiz = algoritmoHuffman(vetor, tamVetor);
    vetor = pegaCod(vetor, tamVetor, raiz);

    printf("Deseja imprimir o código de Huffman para o arquivo
inserido/ (s/n) ");
    char a;
    getchar();
    scanf("%c", &a);
    if(a=='s' || a=='S'){
        printf("\nImprimindo a ocorrência dos caracteres\n");
        imprimeVetor(vetor, tamVetor);

        printf("\n\nImprimindo a árvore do código de Huffman\n\n");
        imprimeArvore(raiz);

        printf("\n\nImprimindo os códigos de cada caractere\n");
        imprimeVetorCod(vetor, tamVetor);

        printf("\n");
    }
```

Assim, com os arquivos lidos, podemos remontar a árvore do algoritmo de Huffman e o vetor com os códigos de cada caractere. Assim, perguntamos ao usuário se ele deseja ver as estruturas reconstruídas. Em caso positivo, é impresso o vetor com as frequências, a árvore, e o vetor com os códigos.

```
    char nomeSaida[50];
    printf("\nInsira o nome do arquivo de saída: ");
    scanf("%s", nomeSaida);

    binCode(vetSaida, tamBin, raiz, nomeSaida);
    free(vetor);
    free(vetSaida);
    printf("Arquivo descompactado com sucesso!\n");

}
```

Então, pedimos ao usuário que insira o nome do arquivo de saída, e chamamos a

função *binCode* para fazer a decodificação e salvar o arquivo. Ao fim da função liberamos os espaços utilizados pelos vetores e é impressa uma mensagem de sucesso.

```
void salvar(long long tamBin, long long tamVetor, No* vetor,
unsigned char *vetSaida, char* nomeSaida, long long tamVetSaida){

   FILE * arq = fopen(nomeSaida, "wb");
   if(!arq){
       printf("Não foi possível criar o arquivo!\n");
       exit(1);
   }
   fwrite(&tamBin, sizeof(long long), 1, arq);
   fwrite(&tamVetor, sizeof(long long), 1, arq);
   fwrite(vetor, sizeof(No), tamVetor, arq);
   fwrite(&tamVetSaida, sizeof(long long), 1, arq);
   fwrite(vetSaida, sizeof(unsigned char), tamVetSaida, arq);

   fclose(arq);
}
```

A função *salvar* é a responsável por criar um arquivo binário e salvar o arquivo compactado nele. Ela recebe por parâmetro todos os elementos que deverão ser salvos e o nome do arquivo de saída. Primeiramente o arquivo é aberto para escrita binária. Caso a abertura não seja bem sucedida, é impresso uma mensagem de erro. Mas, caso contrário, as informações são salvas e o arquivo fechado.

```
void comprimir(){

   char nomeArq[50];
   printf("Digite o nome do arquivo que deseja comprimir: ");
   scanf("%s", nomeArq);

   printf("\nCarregando arquivo...\n");

   Arvore *raiz;
   No* vetor, *vetorfim;

   long long tamVetor, tamCodBin, tamvetSaida;
   unsigned char * vetSaida;
   char nomeSaida[50];

   vetor = insereFrequencia(nomeArq, &tamVetor);
   raiz = algoritmoHuffman(vetor, tamVetor);
   vetorfim = pegaCod(vetor, tamVetor, raiz);
   tamvetSaida=calculaTamVet(vetorfim, tamVetor);
   vetSaida = vetorSaida(vetorfim, tamVetor, &tamCodBin, nomeArq);
```

```
int op, op2, men=1;
while(men){

    printf("\n\n----------MENU----------\n");
    printf("1 - Comprimir um Arquivo\n");
    printf("2 - Imprimir contagem de ocorrência\n");
    printf("3 - Imprimir código de Huffman\n");
    printf("4 - Testar Algoritmo de Decodificação\n");
    printf("5 - Gerar arquivo Comprimido\n");
    printf("6 - Descomprimir Arquivo\n");
    printf("0 - Sair\n");
    scanf("%d", &op);
```

A função *comprimir* é responsável por fazer todas as etapas da compressão, e imprimir o menu personalizado para quando um arquivo foi carregado. No início da função, é solicitado ao usuário o nome do arquivo que deseja compactar, e então é chamada as funções para criar o vetor com o número de ocorrências, a árvore, o vetor com os códigos e o vetor de saída. E então é feito um laço para imprimir ao usuário o menu personalizado, e solicita que ele escolha uma opção.

```
    switch(op){
        case 1:
            free(vetor);
            free(vetSaida);
            free(vetorfim);
            comprimir();
            men=0;
            break;
```

Caso o usuário escolha a opção 1, então um novo arquivo deverá ser aberto. Assim, os vetores são liberados e a função comprimir é chamada novamente.

```
        case 2:

            printf("\nImprimindo as ocorrências dos
caracteres\n");
            imprimeVetor(vetor, tamVetor);
            printf("\n\n");
            break;
```

Já, se o usuário escolher a opção 2, então será impresso o vetor com todos os caracteres do arquivo, e a quantidade de vezes que ele ocorre.

```
            case 3:

                printf("\n\nImprimindo código de Huffman\n");
                printf("1 - Imprimir árvore\n");
                printf("2 - Imprimir caracteres e seus códigos\n");
                scanf("%d", &op2);
                switch(op2){
                    case 1:
                        printf("Imprimindo Arvore\n");
                        imprimeArvore(raiz);
                        printf("\n\n");
                        break;

                    case 2:
                        printf("Imprimindo os caracteres e seus
codigos:  ");
                        printf("[freq : carac : cod]\n\n");
                        imprimeVetorCod(vetorfim, tamVetor);
                        printf("\n\n");
                        break;
                }
                break;
```

Já, caso o usuário escolha a opção 3, então será impresso um segundo menu onde ele poderá escolher a impressão da árvore gerada pelo algoritmo de Huffman, ou o vetor que contém os caracteres, a contagem de ocorrências e seus códigos binários.

```
            case 4:
                printf("Digite o nome do arquivo de saída: ");
                scanf("%s", nomeSaida);
                binCode(vetSaida, tamCodBin,  raiz, nomeSaida);
                break;

            case 5:
                printf("\nInsira o nome do arquivo de saída: ");
                scanf("%s", nomeSaida);
                salvar(tamCodBin,tamVetor,vetor, vetSaida,
nomeSaida, tamvetSaida);
                printf("Arquivo salvo com sucesso!\n");
                men=0;
                free(vetor);
                free(vetSaida);
                free(vetorfim);
                break;
```

Caso o usuário escolha a opção 4, então será feita a decodificação do vetor de saída gerado no início da função. Então solicita-se ao usuário o nome do arquivo de saída, e então, depois do arquivo criado, volta-se ao menu. Caso o usuário selecione a opção 5, então é

solicitado o nome do arquivo de saída, e chama-se a função *salvar* para criar o arquivo compactado. Ao fim, libera-se os vetores e se encerra o loop, voltando para a função *main*.

```
            case 6:

                printf("\n\nInsira o nome do arquivo que deseja
decomprimir: ");
                scanf("%s", nomeSaida);
                descomprimir(nomeSaida);
                break;

            case 0:

                free(vetor);
                free(vetSaida);
                free(vetorfim);
                exit(1);
                break;
        }
```

Caso o usuário selecione a opção 6, então é solicitado que ele insira o nome do arquivo a ser descompactado, chama-se a função *descomprimir*, e retorna-se ao menu. Já, caso o usuário escolha a opção 0, os vetores são liberados e o programa encerrado.

Arquivo "*main.c*"

O arquivo *main.c* é o arquivo que contém a função principal, com o menu principal e a chamada das funções de compactar e de descompactar.

```
int main(){

   int op, enq = 1;
   char nomearq[50];

   while(enq){
       printf("\n\n--------MENU--------\n");
       printf("1 - Compactar um arquivo\n");
       printf("2 - Descompactar um arquivo\n");
       printf("0 - Sair do programa\n");
       scanf("%d", &op);

       switch(op){
           case 1:
               comprimir();
               break;
           case 2:
```

```
                printf("Insira o nome do arquivo que deseja
descompactar: ");
                scanf("%s", nomearq);
                descomprimir(nomearq);
                break;
            case 0:
                enq=0;
                return 0;
                break;
            default:
                break;
        }
    }
}
```

A função consiste, basicamente, em um laço que imprime o menu e solicita uma opção. Caso o usuário escolha a opção 1, a função *comprimir* é chamada. Caso o usuário insira a opção 2, é solicitado o nome do arquivo a ser descompactado, e então se chama a função *descomprimir*. Já, caso a opção escolhida seja a 0, então o programa é encerrado.

CONCLUSÃO

Neste relatório foi possível descrever todas as funções implementadas, aplicando os conceitos aprendidos em aula e exercitando os conhecimentos adquiridos anteriormente, o que ocasionou uma melhora nas práticas de programação e de entendimento de códigos.

Na questão da compactação, temos que o algoritmo implementado não é eficiente para arquivos de texto pequenos (abaixo de 5Kb), mas apresenta grau de compressão entrem 40% e 60% para arquivos de texto acima de 3MB.