UFES / CEUNES

DEPARTAMENTO COMPUTAÇÃO E ELETRÔNICA

CURSO DE GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

CHRISTIAN JONAS OLIVEIRA

JOÃO VICTOR DO ROZÁRIO RECLA

IMPLEMENTAÇÃO DO ALGORITMO DE HUFFMAN

SÃO MATEUS – ES

2021

Sumário

Neste trabalho, o objetivo era implementar o algoritmo de Huffman afim de realizar a compactação e descompactação de um arquivo de texto. Listaremos a seguir as funções utilizadas e suas devidas explicações.

1. Definição das estruturas

```
typedef struct Tree{

        struct Tree *left;
        struct Tree *right;
        int freq;
        char elemento;

}Tree;
```

```
typedef struct Head{

        Tree **vetor;
        int tam;
        int qnt_elem;
        int qnt_nos;
        int qnt_Bits;

}Head;
```

```
typedef struct Codigo{

        int tamanho;
        char elemento;
        char *cod;

}Codigo;
```

2. Compactação

2.1 ReadFile_zip

```
char *readFile_Zip();
```

Função responsável por ler um arquivo a ser compactado, nela o usuário entra com o nome do arquivo e a função retornará um vetor contendo todos os caracteres do arquivo.

/*

Leitura do nome do arquivo

*/

```
while(fscanf(fp, "%c", &c)!=EOF){
        qnt++;

}

rewind(fp);

char *vet = (char *)calloc(qnt, sizeof(char));

// leitura dos caracteres do arquivo de entrada
if(vet){

        while(fscanf(fp, "%c", &c)!=EOF){
                vet[i] = c;
                i++;
        }

        vet[strlen(vet)] = 0;
}

fclose(fp);


return vet;
```

O arquivo é percorrido no primeiro *while* afim de contar a quantidade de caracteres nele presente. Após a contagem, será alocado um espaço de memória referente o tamanho do arquivo. Por fim, realiza-se a leitura e o posicionamento dos caracteres no vetor.

2.2 Count_Freq

```
Head *count_Freq(char *vet);
```

Neste algoritmo, é passado o vetor ordenado (através do algoritmo QuickSort) para realizar a contagem da frequência de cada caractere, e por fim é gerado um cabeçalho que guardará os Nós criados com o elemento e sua frequência.

/*

Alocação de memória

*/

```
// percorre o vetor
while(i < strlen(vet)){

        qnt = 0;
        c = vet[i];
        j = i;

// conta a frequência em um vetor ordenado
        while(j < strlen(vet) && vet[j] == c){

                qnt++;
                j++;
                i++;
        }

        No = create_Tree(c, qnt);

// posicionamento e inicialização do Nó
        if(No){

                aux[k] = No;
                k++;
        }

        else{
                printf("\n\t Erro de alocacao");
                exit(1);
        }

}

cab->qnt_nos = k;
cab->vetor = aux;
cab->tam = k;
cab->qnt_elem = k;

return cab;
```

2.3 Huffman

```
void Huffman(Head *cab);
```

Aqui, será aplicado o algoritmo de Huffman propriamente, onde o vetor presente no cabeçalho sofrerá reapontamentos, e a cada reapontamento, uma atualização na sua ordem será aplicada, que ao final formarão uma árvore com raiz na posição 0 do vetor.

```
for(i = 0; i<n-1; i++){

        z = create_Tree('#', 0);

        if(z){
                x = cab->vetor[0];
                y = cab->vetor[1];
                z->left = x;
                z->right = y;
                z->freq = x->freq + y->freq;
                cab->qnt_nos++;
                update_Priority(cab, z);

        }

        else{
                printf("\n\t Erro de Alocacao");
                exit(1);
        }
}
```

2.4 Create_Code

```
void create_Code(Tree *ptr, char *codigo, Codigo **vet,
             char *vet_aux, int *tamCODE, int *iCODE)
```

Após a criação da árvore será gerado o código de Huffman para cada elemento com base no caminho recursivo pela descida na árvore.

```
if(!ptr){
        *tamCODE-=1;
}

else{

        // chamada da sub-arvore esquerda
        if(ptr->left){
                codigo[*tamCODE] = '0';
                *tamCODE+=1;
                create_Code(ptr->left, codigo, vet, vet_aux, tamCODE,
iCODE);
        }

        // chamada da sub-arvore direita
        if(ptr->right){
                codigo[*tamCODE] = '1';
                *tamCODE+=1;
                create_Code(ptr->right, codigo, vet, vet_aux, tamCODE,
iCODE);
        }

// caso encontre o elemento, aloca novo no e o guarda no vetor de
codigos
        else{

                strncpy(vet_aux, codigo, *tamCODE);
                vet_aux[*tamCODE] = '\0';
                vet[*iCODE] = create_CodNode(ptr->elemento, vet_aux);
                *iCODE+=1;
        }
}
*tamCODE-=1;
```

2.5 Create_vetSaida

```
char *create_vetSaida(Codigo **ptr, char *org, Head *cab);
```

Com a aplicação do algoritmo de Huffman e a criação da tabela de códigos, é realizada a chamada desta função afim de gerar o código de Huffman referente ao texto por completo.

```
// cria o vetor com o codigo de Huffman referente ao texto inteiro
lido

for(i = 0; i < cab->qnt_elem; i++){
        soma += percorre_arvore(cab->vetor[0], ptr[i]->cod) *
                                strlen(ptr[i]->cod);
}

char *saida = (char *)calloc(soma, sizeof(char));

// percorre o vetor contendo o texto
for(i = 0; i < strlen(org); i++){

        // percorre a tabela de codigos
        for(j = 0; j < cab->qnt_elem; j++) {

                if(ptr[j]->elemento == org[i]){
                        strcat(saida, ptr[j]->cod);
                        break;
                }
        }
}

return saida;
```

2.6 Zip_File

```
void zip_File(FILE *fp, char *vet, Head *cab, Codigo **ptr);
```

Por fim, é realizada a chamada do algoritmo que compactará e escreverá no arquivo de saída todos os dados da tabela, o cabeçalho e o texto compactado.

```
for(k = 0; k<strlen(vet); k++){

        if(vet[k]=='0') cv[k] = 0;
        else cv[k] = 1;
}

// deslocamento
for(i = 0; i < strlen(vet); i++){
        posbyte = pos/8;
        posbit = pos%8;
        aux = cv[i];
        aux = aux << (8 - posbit - 1);
        saida[posbyte] = saida[posbyte] | aux;
        pos++;
}

// gravacao no arquivo
cab->qnt_Bits = strlen(vet);
arq_Head(fp, cab);
arq_Codigo(fp, ptr, cab->qnt_elem);
fwrite(saida, sizeof(saida), 1, fp);
```

A compactação será feita usando a técnica *bitwise* que reduzirá o código de Huffman deslocando-o para o inserir no arquivo binário.

3. Descompactação

3.1 Unzip_File

```
void unzip_File(FILE *fp, FILE *fp2);
```

Função recebe dois ponteiros de arquivo, um para o arquivo a ser descompactado e outra para o arquivo de saída. Nela será controlado todo o processo de descompactação.

/*

Leitura do arquivo e alocações necessárias

*/

```
// calculo da quantidade de bits necessarios para alocar o codigo de
Huffman apos descompactação

if((cab->qnt_Bits/8) != 0){

        t = cab->qnt_Bits/8;
        s = t * 8 + 8;
}

unsigned char vet[s];
char *result = (char *)calloc(s, sizeof(char));
fread(vet, s, 1, fp);
pos = cab->qnt_Bits;

// deslocamento e tradução (para Huffman) do codigo lido
for (i=0; i<pos; i++) {
      posbyte = i/8;
      posbit = i%8;
      aux = 1;
      aux = aux << (8 - posbit - 1);
      aux = vet[posbyte] & aux;
      aux = aux >> (8 - posbit - 1);
      if(aux == 0) result[i] = '0';
      else result[i] = '1';
}

// chamada da função que decoficara o codigo de Huffman descompactado
decode_File(c, result, fp2, cab->qnt_elem);
```

3.2 Decode_File

```
void decode_File(Codigo **c, char *codigo, FILE *decodificacao,
                                        int tam);
```

Essa função será chamada na *unzip_File,* responsável por decodificar o código de Huffman que foi descompactado, gerando e imprimindo o arquivo de saída.

```
printf("\n\t Texto descompactado: ");

for(i = 0; i < strlen(codigo); i++){
        ptr[k] = codigo[i];

        for(j = 0; j < tam; j++){

                if(strncmp(ptr, c[j]->cod, c[j]->tamanho)==0){
                        growup = 1;
                        printf("%c", c[j]->elemento);
                        fwrite(&c[j]->elemento, sizeof(char), 1,
                                decodificacao);
                        memset(ptr, 0, MAX);
                        k = 0;
                        break;
                }
        }

        if(growup==0)
                k++;
        growup = 0;
}
```

4. Funções Auxiliares

4.1 QuickSort

```
void QuickSort(char *vet, int ini, int fim);
void QuickSort_cab(Head *cab, int ini, int fim);
```

Para a implementação do código de Huffman foi-se utilizado o algoritmo de ordenação *QuickSort,* possuindo duas variações, uma que ordena os elementos de acordo com seu valor em sua tabela de codificação (ASC, ASCII..., etc) e outra que ordena os elementos de acordo com sua frequência.

4.2 Imprime_Frequencia

```
void imprime_Frequencia(Head *cab);
```

Essa função é responsável por imprimir os elementos identificados junto de suas frequências.

```
printf("\n\t Elemento\t|\tFrequencia");
printf("\n");
for(i = 0; i<cab->tam; i++){

        if(aux[i]->elemento=='\n')
                printf("\n\t enter \t\t|\t %d", aux[i]->freq);

        else if(aux[i]->elemento==' ')
                printf("\n\t espaco \t|\t %d", aux[i]->freq);

        else printf("\n\t %c \t\t|\t %d", aux[i]->elemento, aux[i]-
>freq);

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

4.3 Update_Priority

```
void update_Priority(Head *cab, Tree *elem);
```

Função responsável por atualizar a fila de prioridades utilizada no algoritmo de Huffman reorganizando o vetor e liberando os espaços sobresalientes.

```
int i;

// auxiliar Huffman que reordena  o vetor de acordo com a retirada do
menor elemento
for(i = 2; i < cab->tam; i++){

        // caso onde a freqüência do novo No é menor que de outro
        if(elem->freq < cab->vetor[i]->freq){
                cab->vetor[i-2] = elem;
                elem = cab->vetor[i];

        }
        // caso onde a freqüência do novo No é maior ou igual que de
outro
        else if(elem->freq >= cab->vetor[i]->freq)
                cab->vetor[i-2] = cab->vetor[i];

        // caso seja o ultimo elemento do vetor
        if(i+1 == cab->tam)
                cab->vetor[i-1] = elem;

}

// caso haja somente dois elementos
if(cab->tam==2){

        cab->vetor[0] = elem;
}

// liberacao do espaco de memoria do no sobresaliente
cab->vetor[i-1] = NULL;
free(cab->vetor[i-1]);
cab->tam--;
```

4.4 Printa_Codigo

```
void printa_Codigo(Codigo **aux, int tam);
```

Função responsável por imprimir cada elemento identificado juntamente com a codificação de Huffman a ele gerado.

```
// imprime o codigo referente a cada caractere
int i;
printf("\n");
printf("\n\t Elemento\t|\tCodigo");
printf("\n");
for(i = 0; i < tam; i++){

        if(aux[i]->elemento=='\n')
                printf("\n\t enter \t\t|\t %s", aux[i]->cod);
        else if(aux[i]->elemento==' ')
                printf("\n\t espaco \t|\t %s", aux[i]->cod);
        else printf("\n\t %c \t\t|\t %s", aux[i]->elemento, aux[i]-
>cod);

        printf("\n");

}
```

4.5 Percorre_Arvore

```
int percorre_arvore(Tree *ptr, char *cod);
```

Função auxiliar da *create_vetSaida*, responsável por percorrer a árvore e retornar a frequência, que contribuirá para a alocação de espaço de memória para os vetores.

```
        for(i = 0; i<strlen(cod); i++){

                if(cod[i]=='0')
                        ptr = ptr->left;
                else
                        ptr = ptr->right;
        }
        return ptr->freq;
}
```

4.6 Arq_Head

```
void arq_Head(FILE *fp, Head *c);
```

```
// move o ponteiro para o inicio do arquivo e escreve o cabecalho nele
        rewind(fp);
        fwrite(c, sizeof(Head), 1, fp);
}
```

4.7 Arq_Codigo

```
void arq_Codigo(FILE *fp, Codigo **ptr, int tam);
```

Função responsável por escrever num arquivo compactado a tabela que auxiliará futuramente na tradução do código.

```
// move o ponteiro para o incio do arquivo
int i;
rewind(fp);

// move o ponteiro para a posicao a frente do  cabecalho
fseek(fp, sizeof(Head), SEEK_SET);

// salvamento dos campos da estrutura que guarda os codigos de Huffman
de cada caractere

for(i = 0; i<tam; i++){
        fwrite(ptr[i], sizeof(Codigo), 1, fp);
        fwrite(ptr[i]->cod, ptr[i]->tamanho * sizeof(char), 1, fp);
}
```

4.8 ReadFile_Unzip

```
Codigo **readFile_Unzip(FILE *fp, int tam);
```

Função utilizada na *Unzip_File*, responsável por ler a tabela de códigos do arquivo já compactado.

```
int i;

// move o ponteiro do arquivo para a posicao a frente do cabecalho
fseek(fp, sizeof(Head), SEEK_SET);

// leitura da tabela que guarda o codigo e os elementos contidos na
codificacao
for(i = 0; i < tam; i++){

        c[i] = (Codigo *)calloc(1, sizeof(Codigo));
        fread(c[i], sizeof(Codigo), 1, fp);
        c[i]->cod = (char *)calloc(1, c[i]->tamanho * sizeof(char));
        fread(c[i]->cod, c[i]->tamanho * sizeof(char), 1, fp);
}

return c;
```

4.9 Create_Tree

```
Tree *create_Tree(char c, int qnt);
```

Função que aloca espaço para um Nó de uma arvore.

```
// alocacao de um espaco de memoria para um node da arvore
Tree *Node = NULL;
Node = (Tree *)malloc(sizeof(Tree));

// iniciando seus campos
if(Node){
        Node->elemento = c;
        Node->freq = qnt;
        Node->left = NULL;
        Node->right = NULL;
}

return Node;
```

4.10 Create_CodNode

```
Codigo *create_CodNode(char elem, char *c);
```

Função que aloca espaço para um linha de uma tabela.

```
// alocacao e inicializacao de um novo elemento Codigo, que guarda o
caractere e seu codigo

Codigo *novo = (Codigo *)malloc(sizeof(Codigo));

if(novo){
        novo->tamanho = strlen(c);
        novo->elemento = elem;
        novo->cod = (char *)calloc(strlen(c), sizeof(char));
        strcpy(novo->cod, c);
        }
return novo;
```

4.11 Destroy_Codigo

```
void destroy_Codigo(Codigo **code, int tam);
```

```
// destroi um vetor de Codigos
int i;
for(i = 0; i < tam; i++){

        free(code[i]->cod);
        free(code[i]);
}

free(code);
```