Universidade de Brasília

Departamento de Ciência da Computação

Lista de Exercícios 5
Organização de Arquivos

Autores:

Giovanni M Guidini 16/0122660

Vitor F Dullens 16/0148260

Brasília

21 de Maio de 2018

Lista de Exercicios 4 **Organização de Arquivos**

Alunos: Giovanni Guidini 16/0122660; Vitor Dullens 16/0148260. CIC 116327
Prof. Oscar Gaidos Data: 21 de Maio de 2018

Exercicio 1

Implemente o método de Huffman para compressão de dados. Pesquise o algoritmo FGK e o implemente.

Código

O código para esta questão está no apêndice A. O algoritmo comprime um arquivo base transformando cada byte em um código de tamanho variável calculado com base na frequência de cada byte no arquivo original. Bytes com maior frequência recebem códigos menores, o que permite uma compressão do arquivo. A maior dificuldade na implementação deste código está no fato de precisarmos trabalhar no nível dos bits, mas as linguagens de programação de alto nível oferecem poucas opções de bibliotecas neste sentido. Para contornar o problema utilizamos uma biblioteca compactstorage, encontrada no GitHub, e adaptamos levemente seu uso.

No final da compressão do arquivo é mostrada uma percentagem de compressão, isto é, quão menor é o arquivo comprimido em comparação com o original.

Saídas

Figura 1: Arquivo .txt a ser comprimido

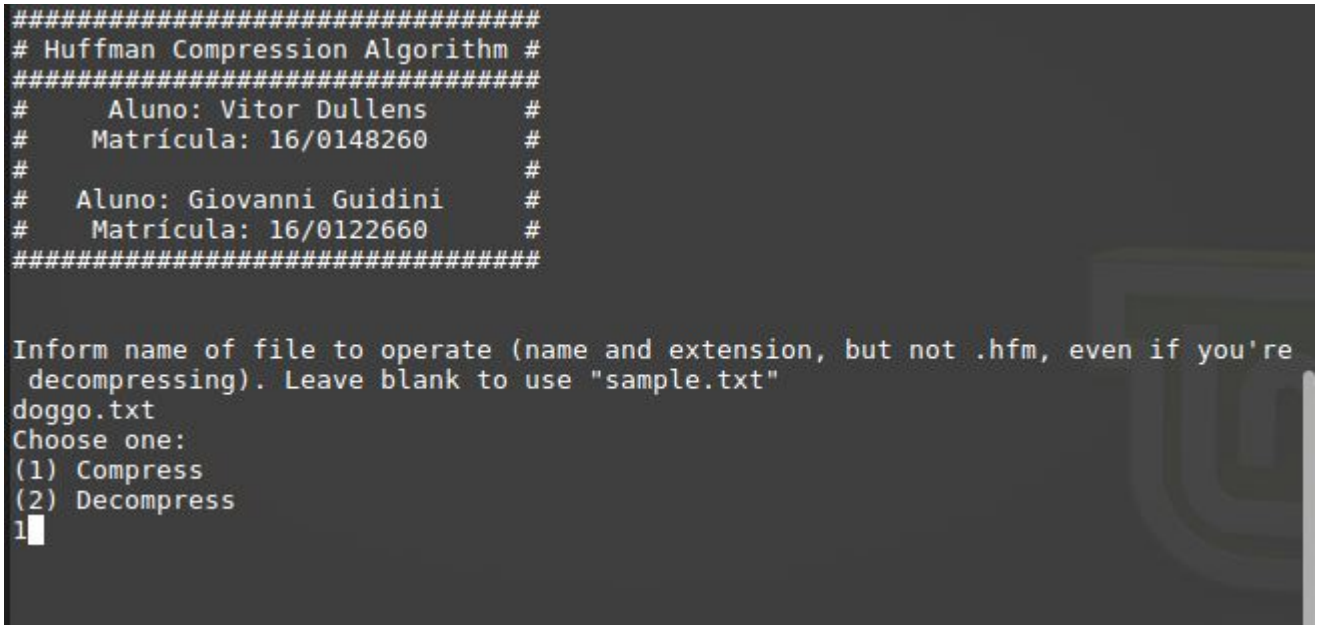

Figura 2: Menu e possíveis opções a serem selecionadas

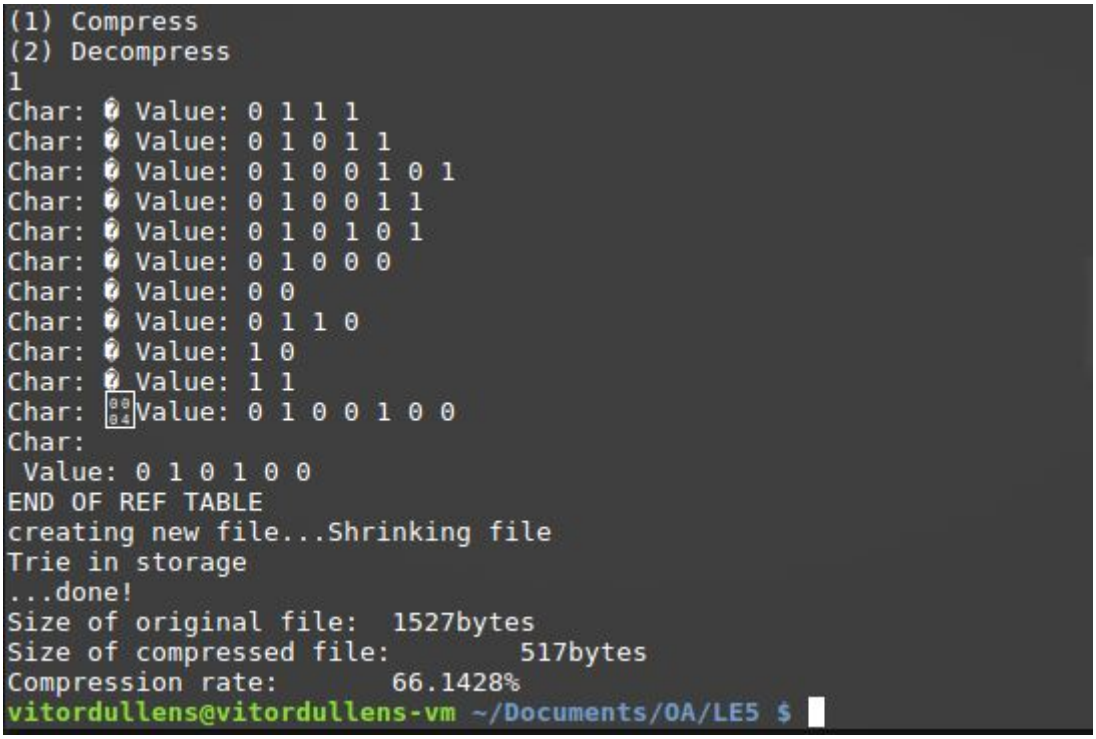

Figura 3: Resultado da compressão do arquivo doggo.txt

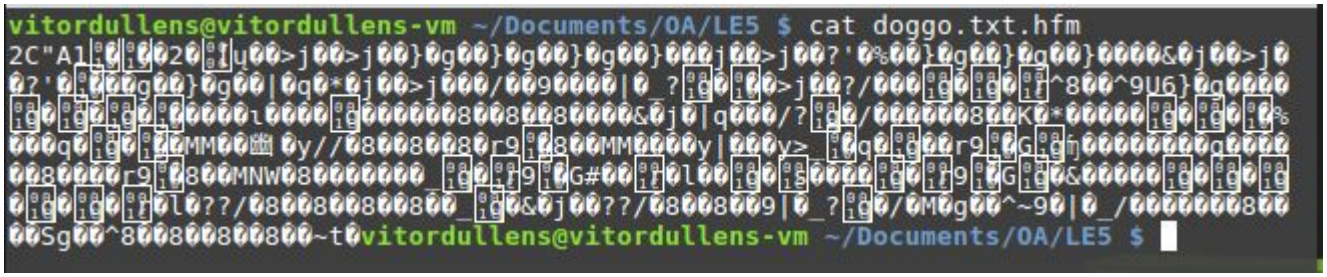

Figura 4: Resultado da visualização do arquivo doggo comprimido

Figura 5: Resultado da descompressão do arquivo

Exercicio 2

Descreva e implemente um método de ordenação dentre os seguintes: quick sort, shell sort ou merge sort.

Código

Resolvemos implementar o metodo de ordenação quicksort, que utiliza um pivô e divide o vetor em duas partes a serem ordenadas com base neste pivô. Sua complexidade na maioria dos casos é O(n*log n) que é o melhor possível para uma ordenação, porém, no pior dos casos o quicksort se comporta de forma quadratica - O(n2). O código se encontra no apêndice B.

Saídas

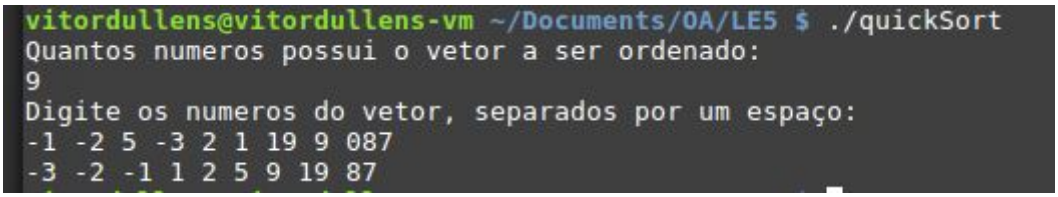

Figura 6: Algoritmo de ordenação Quick Sort

A Exercício 1

```
/*
    [X] Implemente o metodo de Huffman para compressao de dados.
    Implementacao por Vitor F Dullens - 16/0148260 e Giovanni M Guidini - 16/0122660
    Codigo de Huffman de tamanho variavel.

    Fazendo uso da biblioteca compactstorage para manipulacao de bits. Copyright (C) 2012 Franz Liedke
    Fazendo uso da biblioteca hufftrie para criacao da arvore de huffman. Ours.
*/

#include <bits/stdc++.h>
#include "compactstorage-master/compactstorage.h"
#include "hufftrie.hpp"
using namespace std;
namespace hft = hufftrie;

// DEBUG
void showTrie(hft::Huffnode* root){
    cout << *root << " " << root->getLeft() << " " << root->getRight() << endl;
    if(root->getLeft() != NULL)
        showTrie(root->getLeft());
    if(root->getRight() != NULL)
        showTrie(root->getRight());
}
/* ================================== COMPRESSING FILE ==================================*/
// writes trie has a stream of bits
void addTrieToFile(CompactStorage& storage, hft::Huffnode* root){
    // cout << "Node: " << *root << endl; // DEBUG
    if(root->getChar() != INTERNAL_CHAR || root->isLeaf()){
        storage.writeBool(1);
        storage.writeInt(root->getChar(), 8);
        return;
    }
    // write internal node
    storage.writeBool(0);
    // cout << "The trie: " << endl; // DEBUG
    // storage.dump(); // DEBUG
    // explore left
    addTrieToFile(storage, root->getLeft());
    // explore Right
    addTrieToFile(storage, root->getRight());

    // cout << "The trie: " << endl; // DEBUG
    // storage.dump(); // DEBUG
}

string shrinkFile(map<char, hft::Huffnode*>& ref, string fileName){
    int done = 0;
    ifstream in (fileName); // file to read from
    fileName += ".hfm"; // .hfm for huffman
    fstream fd (fileName, ios::out | ios::binary); // file to write into
    CompactStorage storage;
    // first thing in file needs to be the trie
    cout << "Shrinking file\n"; // DEBUG
    addTrieToFile(storage, hft::getRoot());
    cout << "Trie in storage\n"; // DEBUG
```

```
    // storage.dump(); // DEBUG
    char ch;
    while(in.get(ch)){      // hft::Huffnode*    code
        vector<bool> code = ref[ch]->getCode();
        // cout << "char: " << ch << " code: " << code << endl;
        done += code.size();
        for(bool b : code){
            // adds code to storage
            storage.writeBool(b);
            // cout << "written: " << done << endl;
        }
        // TODO: Relief storage for very large files by dumping it partially when number of bits
        // written is a multiple of 8. Otherwise large files will create seg fault
    }
    // last character is EOF, but get() will not get it
    // so we manually add EOF
    for(bool b : ref[EOT]->getCode()){
        storage.writeBool(b);
    }
    // dump remaining contents
    storage.dump(&fd);
    fd.close();
    // returns name of compressed file
    return fileName;
}

// automate file compression process
int compress(string fileName){
    ifstream fd (fileName); // we know it exists because test was in main

    // frequence of characters
    map<char, int>& frequencies = hft::makeFreq(&fd);
    fd.close(); // no longer necessary
    // Trie of codes
    hft::Huffnode* root = hft::makeTrie(frequencies);
    // Important data containers
    vector<bool> codeCreator;
    map<char, hft::Huffnode*> refTable;
    // Extract codes from trie into refTable
    hft::renderCodes(root, codeCreator, refTable);

    // print codes in screen
    for(auto x: refTable){
        cout << "Char: " << x.first << " Value: " << x.second->getCode() << endl;
    }
    cout << "END OF REF TABLE\n";
    // showTrie(root); // DEBUG
    cout << "creating new file...";
    // created the compressed file
    fileName = shrinkFile(refTable, fileName);
    cout << "...done!\n";

    int r; // size of new file
    fd.open(fileName);
    fd.seekg(0, fd.end);
    r = fd.tellg();
    fd.close();

```

```
    return r;
}

/* ================================== COMPRESSING FILE END
   ==================================*/
/* ================================== DECOMPRESSING FILE
   ==================================*/
void reWrite(CompactStorage& storage){
    int byte = storage.curByte();
    storage.reset();
    vector<char> content;
    while(byte >= 0){
        content.push_back(storage.readInt(8));
        byte--;
    }
    byte = storage.curByte();
    byte--;
    int i = 0;
    storage.resetHard();
    while(i <= byte){
        storage.writeInt(content[i], 8);
        i++;
    }
}

CompactStorage& readTrie(fstream& fd){
    bool done = false;
    static CompactStorage storage(5);
    char ch;
    hft::Huffnode* curr = NULL;
    while(!done){

        // cout << "Sai do for aqui\n"; // DEBUG
        // next byte to process
        if(!done){
            ch = fd.get();
            storage.resetHard();
            storage.writeInt(ch, 8);
        }
        storage.reset();
        for(int i = 0; i < 8 && !done; i++){
            // cout << "Curr Bit " << storage.curBit() << " Curr Byte " <<
   storage.curByte() << endl; // DEBUG
            // storage.dump(); // DEBUG
            bool b = storage.readBool();
            if(b){
                // cout << "Leaf node\n" << endl; // DEBUG
                // rewrite storage
                int byte = storage.curByte(); // current byte
                reWrite(storage);
                // expand with new char
                ch = fd.get();
                storage.writeInt(ch, 8);
                storage.reset(); // back to start
                storage.readInt(byte*8+i+1); // forward to curr position
                char v = storage.readInt(8);
                // cout << "v = " << (int) v << endl; // DEBUG
                hft::Huffnode* n = new hft::Huffnode(v, 0, curr);

                // cout << "Found char: " << v << endl; // DEBUG
```

```
171                 if(curr->getLeft() == NULL)
                        curr->setLeft(n);
173                 else
                        curr->setRight(n);
175             }
                else{
177                 // cout << "found internal node\n"; // DEBUG
                    hft::Huffnode* now = new hft::Huffnode(curr);
179                 if(curr != NULL){
                        if(curr->getLeft() == NULL)
181                         curr->setLeft(now);
                        else
183                         curr->setRight(now);
                    }
185                 curr = now;
                }
187
                while(curr->getLeft() != NULL && curr->getRight() != NULL){
189                 // cout << "Curr " << curr << " Parent "<<  curr->getParent() <<
    endl; // DEBUG
                    if(curr->getParent() != NULL){
191                     curr = curr->getParent();
                        // extra safety
193                     if(curr == NULL){
                            done = true;
195                         break;
                        }
197                 }
                    else{
199                     done = true;
                        break;
201                 }
                }
203         }
        }
205     // get the root of the trie and set it
        while(curr->getParent() != NULL){
207         curr = curr->getParent();
        }
209     hft::setRoot(curr);
        // cout << "returning trie. Storage: " << endl; // DEBUG
211     // storage.dump(); // DEBUG
        return storage;
213 }

215 // write to stdout
    void readFile(fstream& in, hft::Huffnode* root, CompactStorage& storage){
217     int i = storage.curBit();
        hft::Huffnode* it = root;
219     char out = 0;
        char read;
221     // file to be read
        while(out != EOT){
223         // storage.dump(); // DEBUG
            // getchar(); // DEBUG
225         // next byte to read
            while(i < 8){
227             // cout << "Curr Bit " << storage.curBit() << " Curr Byte " <<
    storage.curByte() << endl; // DEBUG
                i++;
```

```
229             bool b = storage.readBool();
                // cout << b << endl; // DEBUG
231             // cout << *it << endl; // DEBUG
                // getchar(); // DEBUG
233             if(b){
                    it = it->getRight();
235             }
                else{
237                 it = it->getLeft();
                }
239             out = it->getChar();
                if(it->isLeaf()){
241                 if (out == EOT)
                        break;
243                 cout << out;
                    it = root;
245             }
            }
247         if(out != EOT){
                storage.resetHard();
249             read = in.get();
                storage.writeInt(read, 8);
251             i = 0;
                storage.reset();
253         }
        }
255 }

257 // write into another file
    void readFile(fstream& in, hft::Huffnode* root, CompactStorage& storage, fstream&
         ou){
259     int i = storage.curBit();
        hft::Huffnode* it = root;
261     char out = 0;
        char read;
263     // file to be read
        while(out != EOT){
265         // storage.dump(); // DEBUG
            // getchar(); // DEBUG
267         // next byte to read
            while(i < 8){
269             // cout << "Curr Bit " << storage.curBit() << " Curr Byte " <<
        storage.curByte() << endl; // DEBUG
                i++;
271             bool b = storage.readBool();
                // cout << b << endl; // DEBUG
273             // cout << *it << endl; // DEBUG
                // getchar(); // DEBUG
275             if(b){
                    it = it->getRight();
277             }
                else{
279                 it = it->getLeft();
                }
281             out = it->getChar();
                if(it->isLeaf()){
283                 ou.write(&out, 1);
                }
285         }
            if(out != EOT){
```

```
287             storage.resetHard();
                read = in.get();
289             storage.writeInt(read, 8);
                i = 0;
291             storage.reset();
            }
293     }
}
295
void decompress(string file, string  outFile = ""){
297     CompactStorage storage;
        hft::Huffnode* root;
299     fstream fd (file, ios::in);
        // check for file
301     if(!fd){
            cout << "Compressed file not found. Compress first\n";
303         exit(1);
        }
305     // cout << "reading trie now\n"; // DEBUG
        // getchar(); // DEBUG
307     // created trie. Returns last byte possibly unused
        storage = readTrie(fd);
309     // actual trie root
        root = hft::getRoot();
311     // dumping file
        // cout << "reading trie done\n"; // DEBUG
313     // showTrie(root); // DEBUG
        // getchar(); // DEBUG
315     if(outFile != ""){
            fstream out (outFile, ios::out | ios::trunc);
317         readFile(fd, root, storage, out);
            cout << "Done! Output in file " << outFile << endl;
319     }
        else{
321         cout << "Decrypting file now\n";
            readFile(fd, root, storage);
323         // cout << "Fim :D\n"; // DEBUG
        }
325 }
/* ================================== DECOMPRESSING FILE END
    ==================================*/
327 string intro(){
        printf("#######################################\n");
329     printf("# Huffman Compression Algorithm #\n");
        printf("#######################################\n");
331     printf("#     Aluno: Vitor Dullens      #\n");
        printf("#    Matricula: 16/0148260      #\n");
333     printf("#                               #\n");
        printf("#   Aluno: Giovanni Guidini     #\n");
335     printf("#    Matricula: 16/0122660      #\n");
        printf("#######################################\n");
337     printf("\n\nInform name of file to operate (name and extension, but not .hfm,
        even if you're decompressing). Leave blank to use \"sample.txt\"\n");
        string file;
339     getline(cin,file); // gets entire line to allow for blank characters
        file = file.substr(0, file.find(" ")); // gets only fist word
341     return file;
}
343 int main(){
        system("clear");
```

```
345     string file = intro();
347     if (file == ""){
            file = "sample.txt";
349     }
351     int op = 0;
        cout << "Choose one:\n(1) Compress\n(2) Decompress\n";
353     while (op != 1 && op != 2)
            cin >> op;
355
        if(op == 1){
357         ifstream fd (file);
            if(!fd){
359             cout << "Arquivo " << file << " inexistente\n";
                return 1;
361         }
            // size of original file
363         int fileSize;
            fd.seekg(0, fd.end);
365         fileSize = fd.tellg();
            fd.close();
367         // compress
            int compressedSize = compress(file);
369         // stats
            cout << "Size of original file:\t" << fileSize << "bytes\n";
371         cout << "Size of compressed file:\t" << compressedSize << "bytes\n";
            cout << "Compression rate:\t" << (1 - ((double) compressedSize/(double)
    fileSize))*100 << "%\n";
373     }
        else{
375         file += ".hfm";
            int op = 0;
377         cout << "Decompress into another file?:\n(1) NO - STDOUT\n(2) YES -
    OUT_FILE\n";
            while (op != 1 && op != 2)
379             cin >> op;
            if(op == 1)
381             decompress(file);
            else
383             decompress(file, "OUT_FILE");
            cout << endl; // end of file doesnt have \n
385     }
}
```

Listing 1: "Code for Huffman compressing algorithm"

B Exercício 2

```
#include <bits/stdc++.h>

using namespace std;

void quickSort(int array[], int low, int high) {
      int i = low, j = high;
      int pivot = array[(low + high) / 2];
      // partition
      while (i <= j) {
            while (array[i] < pivot)
                  i++;
            while (array[j] > pivot)
                  j--;
            if (i <= j) {
                swap(array[i], array[j]); // swap the position of two elements,
    to ajust the array
                  i++;
                  j--;
            }
      }
      // recursion
      if (low < j) quickSort(array, low, j);
      if (i < high) quickSort(array, i, high);
}

int main(){
    int N;
    cout << "Quantos numeros possui o vetor a ser ordenado:" << endl;
    cin >> N;
    int array[N];
    cout << "Digite os numeros do vetor, separados por um espaco:" << endl;
    for(int i=0; i < N; i++)
        cin >> array[i];
    quickSort(array, 0, N-1);
    for (int i = 0; i < N-1; i++)
        cout << array[i] << " ";
    cout << array[N-1] << endl;
}
```

Listing 2: "Code for Quick Sort"