Universidade do Minho
Escola de Engenharia

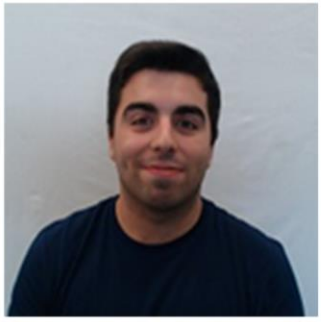
SHAFA
Trabalho Prático
Comunicação de dados

Gonçalo Braz
a93178

Duarte Moreira
a93321

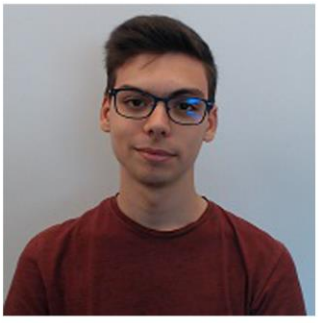
João Barbosa
a93270

Gonçalo Pereira
a93168



Simão Cunha
a93262

Lucas Carvalho
a93176



Tiago Silva
a93277

José Gomes
a82418

Departamento de Informática MIEI 2020/21
3/1/2021

ÍNDICE

1. AUTORIA E DIVISÃO DOS MÓDULOS

O nosso grupo optou por seguir a sugestão do professor e então dividir-se de forma a ficar 2 elementos por cada módulo. Depois de realizada a divisão do grupo foi feita de forma aleatória a atribuição dos módulos.
Segue-se em seguida os autores de cada módulo.

O Módulo F ficou encarregue do Gonçalo Braz e do Simão Cunha.
O Módulo T ficou encarregue do Duarte Moreira e do Lucas Carvalho.
O Módulo C ficou encarregue do João Barbosa e do Tiago Silva.
O Módulo D ficou encarregue do Gonçalo Pereira e do João Gomes.

2. MÓDULO F

2.1. ESTRATÉGIAS E PRINCIPAIS FUNÇÕES DO MÓDULO

Numa primeira fase começamos por descobrir o tamanho de um ficheiro que seria dado ao ***modulof*** para realizar o módulo e, ao mesmo tempo, calcular o número e tamanho dos blocos analisados do ficheiro-fonte. Começamos por implementar uma função para calcular o tamanho do ficheiro, usando as funções ***fseek*** e ***ftell***. Aqui existiram problemas para ficheiros com tamanho superior a 2GB que irão ser relatados noutra secção.

De seguida, criamos uma das funções principais do programa, juntamente com as respetivas auxiliares para realizar a compressão RLE (***compressaoRLE***). Nesta função, lemos um bloco de cada vez, verificando a repetição de caracteres e, caso essa repetição seja maior do que 4 vezes, escrever-se-á no novo ficheiro do tipo *rle* a compressão com a sintaxe de acordo com o enunciado. Caso contrário, escrever-se-á apenas o caractere em questão. Além disso, a função ainda guarda numa lista de inteiros o tamanho de cada bloco do ficheiro do tipo *rle*.

Numa segunda fase, criámos o ficheiro para o cálculo das frequências (*.freq*). Para tal, procedemos do seguinte modo: começamos por criar e iniciar uma matriz de inteiros com o número de linhas igual ao número de blocos do ficheiro original e o número de colunas igual a 256 (número de caracteres disponíveis em código ASCII), e nesse local colocaremos o número de repetições do caractere (coluna) no respetivo bloco (linha).

Estes números são calculados na função ***freq_txt*** ou ***freq_rle*** de acordo com o tipo de ficheiro criado na fase anterior e, lendo bloco a bloco, vamos incrementando o número de repetições do caractere lido na matriz.

Por fim, na função ***freq_file***, utilizando a matriz das frequências e a lista do tamanho dos blocos provenientes do ficheiro do tipo *rle* (caso tenha sido criado), escrevemos no ficheiro do tipo *freq* as frequências e o tamanho dos blocos de acordo com a sintaxe pedida no enunciado do trabalho.

Numa última fase, criamos a função principal ***modulof*** onde apresentámos as várias condicionantes para a execução do módulo f (se existe compressão RLE do primeiro bloco maior do que 5%, se o utilizador decide forçar a compressão, etc) e, em caso afirmativo, executámos o programa.

Criámos também um *prompt* com as várias informações pedidas no enunciado (nome dos autores, tamanhos dos blocos, etc).

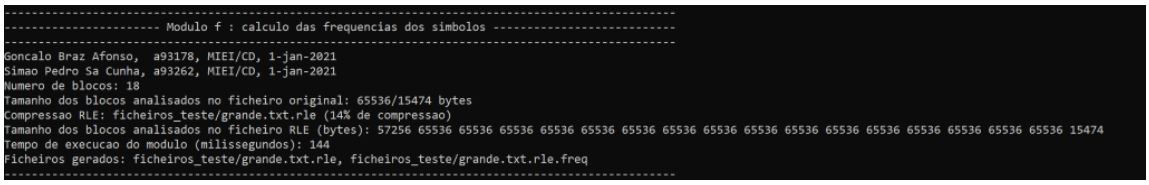
Por último, corrigimos a função que calcula o tamanho dos ficheiros para escolher o tamanho dos blocos a ler de acordo com o tamanho do ficheiro-fonte.

2.2. TESTES

Ficheiro 1: aaa.txt (ficheiro exemplo fornecido pelo professor): 2.92KB

aaa.txt.rle	1,38 Kb
aaa.txt.rle.freq	281 b

Ficheiro 2: grande.txt (teste com tamanho mais considerável): 1104KB



grande.txt.rle	1096 Kb
grande.txt.rle.freq	5,49 Kb

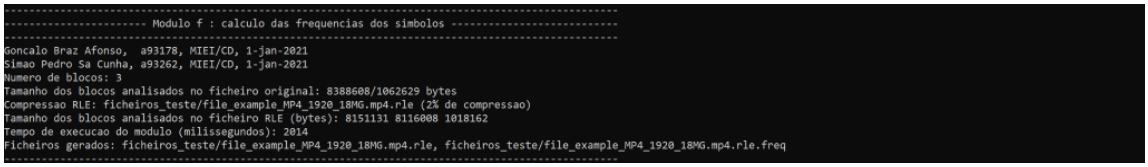
Ficheiro 3: espacos.txt (ficheiro usado para verificar a compressão de espaços): 68.8KB

espacos.txt.rle	5,32 Kb
espacos.txt.rle.freq	558 b

Ficheiro 4: exemplo.txt (ficheiro parecido ao dos professores, com tags no início e no fim para facilitar nos testes): 2.92kb

exemplo.txt.rle	1,38 Kb
exemplo.txt.rle.freq	281 b

Ficheiro 5: file_example_MP4_1920_18MG.mp4 (ficheiro de vídeo - teste para verificar o funcionamento com outros tipos de ficheiros, retirado de um site de ficheiros exemplos): 17,4MB

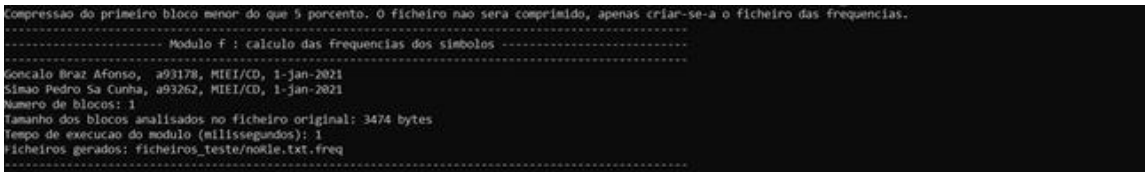


file_example_MP4_1920_18MG.mp4.rle	17 Mb
file_example_MP4_1920_18MG.mp4.rle.freq	5 Kb

Ficheiro 6: file-example_PDF_1MB.pdf (teste semelhante ao ficheiro 5): 0.99MB

file-example_PDF_1MB.pdf.rle	1 Mb
file-example_PDF_1MB.pdf.rle.freq	15,7 Kb

Ficheiro 7: noRle.txt (ficheiro que, segundo o enunciado, não deverá sofrer compressão): 3.39KB



noRle.txt.freq	279 b

Forçamos a compressão no mesmo ficheiro:

noRle.txt.rle	3,39 Kb
noRle.txt.rle.freq	279 b

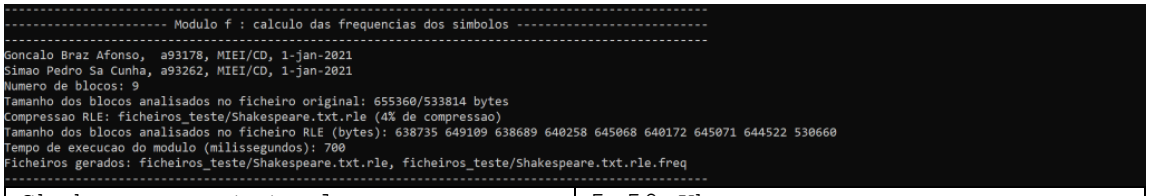
Ficheiro 8: pequeno.txt (teste para ficheiros menores do que 1KB): 25 B

No terminal aparece a seguinte mensagem: *"Ficheiro demasiado pequeno. Programa não pode ser executado."*

O utilizador poderá tentar forçar compressão, mas, dado que o ficheiro fonte tem menos do que 1KB, nada será escrito nos novos ficheiros.

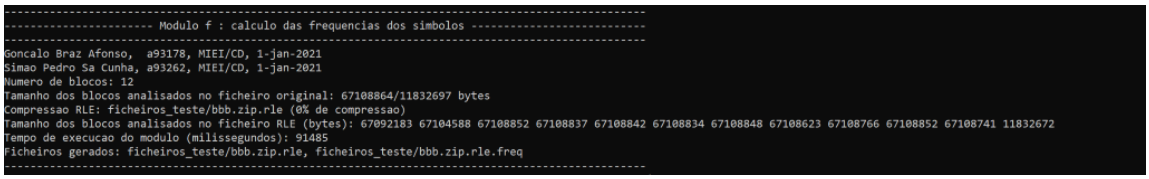
pequeno.txt.rle	0 b
pequeno.txt.rle.freq	6 b

Ficheiro 9: Shakespeare.txt (ficheiro fornecido pelo professor): 5.6 MB



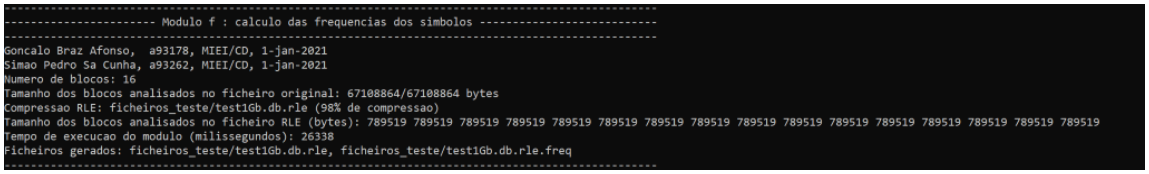
Shakespeare.txt.rle	5,50 Kb
Shakespeare.txt.rle.freq	4,99 Kb

Ficheiro 10: bbb.zip (ficheiro fornecido pelo professor): 715 MB



bbb.zip.rle	720 Mb
bbb.zip.rle.freq	21 Kb

Ficheiro 11: test1Gb.db (ficheiro com 1 gb):1 GB



test1Gb.db.rle	12 Mb
test1Gb.db.rle.freq	5 Kb

2.3. CONCLUSÃO

Durante a implementação do nosso módulo, tivemos alguns percalços.

Ficheiros criados por Gonçalo Braz (que trabalha em Windows) resultavam em *segmentation fault* na máquina de Simão Cunha (em Ubuntu). Numa primeira instância, pensámos que seria uma questão de *encoding*, mas os professores alertaram que isto acontecia devido ao facto de não estarmos a ler o ficheiro-fonte em binário mas sim como um ficheiro de texto. (Superuser, 2011)

Segmentation fault originados por errada alocação de espaço na matriz de frequências.

Dificuldades na leitura de ficheiros de tamanho muito grande, que foi eventualmente corrigido com o uso de **fgetc** em vez de **fscanf**;

Dificuldades no cálculo do tamanho dos ficheiros, cuja função sofreu várias alterações: primeiro usamos as funções **fseek** e **ftell**, mas estas têm um comportamento anómalo em ficheiros superiores a 2GB, pelo que decidimos usar a função **fsize** criada pela equipa docente. No entanto, o problema persistia – o programa não conseguia calcular o tamanho certo o que eventualmente levava o programa a crashar. Foi então que decidimos usar a nossa primeira implementação, tendo em conta que é mais fácil a sua compreensão. (StackOverflow, 2013)

3. MÓDULO T

3.1. ESTRATÉGIAS ADOTADAS

As estratégias que são usadas ao longo deste trabalho não vão muito além da utilização de arreios e strings, com a exceção de uma estrutura de dados referente a listas ligadas. Esta estrutura permitiu-nos que, após a obtenção dos códigos SF dos símbolos de um bloco, estes sejam guardados no mesmo nodo que o seu símbolo correspondente, assim como, a sua frequência. Desta forma, nada era perdido ao longo da execução do programa.

```
typedef struct nodo {
    int frequencia;
    int simbolo;
    char codigoSF[256];
    struct nodo *proximo;
} *LISTA;
```

3.2. FUNÇÕES PRINCIPAIS

De entre todas as funções criadas para este módulo, as que mais se destacam são:

```
LISTA arrayToListOrd(int arr[])
```

Esta função, tal como o próprio nome indica, transforma um arreio de inteiros, passado como argumento, numa lista ligada ordenada e decrescente, em função da frequência de cada símbolo. Para além disso, esta tem também a particularidade de ignorar os símbolos que ocorrem 0 vezes no bloco, uma vez que não vai ser preciso calcular os seus códigos SF.

```
void insertOrd(LISTA *l, LISTA novo)
```

Apesar de ser uma função auxiliar, não perde a sua importância, uma vez que, é esta que permite o correto funcionamento da função falada anteriormente. Assim, o objetivo desta é inserir um nodo (lista de tamanho 1) numa lista ordenada. Tendo em atenção que esta função requer a análise de dois casos distintos, um, onde o nodo é inserido na cabeça da lista e o segundo, quando é inserido na cauda.

```
void storeSFCodes(LISTA l, char codes[][256])
```

Esta função, mais uma vez de extrema importância, vai permitir que após o cálculo dos códigos SF dos símbolos de um bloco (armazenados em strings no arreio codes), estes sejam guardados na lista l, juntamente com os seus símbolos e frequências correspondentes.

```
int getBestDivision(int freq[], int i, int j)
```

Não sendo da nossa autoria, mas sim dos docentes, esta função mostrou ser a mais complexa entre todas as outras, devido à sua dificuldade de compreensão. O seu objetivo é, dado um arreio de frequências, calcular o índice que melhor divide o mesmo de forma a que a soma das frequências das duas partes seja o mais próxima possível.

3.3. TESTES

Estes dois primeiros testes têm como finalidade estudar os casos particulares em que o número de símbolos com frequências diferentes de 0 é nulo ou um.

Teste 1:
INPUT (a.txt.freq):

```
@N@1@1@1;0;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;@0
```

OUTPUT (a.txt.cod):

```
@N@1@1@0;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;@0
```

Teste 2:
INPUT (b.txt.rle.freq)

```
@R@1@0@0;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;@0
```

OUTPUT (b.txt.rle.cod

```
@R@1@0@;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;@0
```

Para finalizar, dois exemplos criados de forma aleatória com o objetivo de ter a certeza de que o programa está a produzir os códigos SF corretos.

Teste 3:

INPUT (c.txt.freq)

```
@N@1@1126@45;;;;;;;;;;0;;;;;;;;;;;;;;;;;;;1;;;;;;;;;;;;;;;0;;;;;;
;;;;;;;;;;;;;25;;195;0;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;105;;0;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;70;75;
80;0;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;3;2;1@0
```

OUTPUT (c.txt.cod)

```
@N@1@1126@10010;10011;1010;10110;10111;1100;11010;11011;11100;11
101;;;;;;;;;;;;;;;;;;;1111110100;1111110101;1111110110;111111011
10; 11111101111;1111111000;1111111001;1111111010;11111110110;111
11110111;1111111100;11111111010;11111111011;1111111110;111111111
10;;;;;;;;;;;;;;;;;;;;11110;111110;000;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;001;010;;;;;;;;;;;;;;;;;;;;;;;
;; ;;;;;;;;;;;1000;0111;0110;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;111111000;1
11111001;11111111111@0
```

Teste 4:

INPUT (d.txt.rle.freq)

```
@R@2@1015@1;2;3;4;5;0;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;1000@5015@1;2;3;4;5;0;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;5000@0
```

OUTPUT (d.txt.rle.cod)

```
@R@2@1015@1111;1110;110;101;100;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;0@5015@1111;1110;110;101;100;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;0@0
```

3.4. CONCLUSÃO

Em jeito de conclusão, estamos conscientes de que o código poderia ser melhorado em termos de qualidade e otimização, no entanto, e de uma forma geral, achamos que está escrito de uma maneira simples e de fácil leitura e acreditamos ter atingido os objetivos propostos para esta fase dois do trabalho prático.

4. MÓDULO C

4.1. ESTRATÉGIAS ABORDADAS

No nosso módulo optamos por realizar, tudo por blocos, desde à leitura do ficheiro original e do tipo cod à escrita do ficheiro tipo shaf. Decidimos utilizar da estratégia sugerida pelo professor de realizar a codificação SF com matrizes de bytes. Criamos para isso uma struct denominada de *code_*. Optamos ainda por fazer a leitura do ficheiro do tipo cod 2 vezes, a primeira leitura para descobrir os símbolos a serem codificados e o maior número de bits encontrado num código, assim sabendo exatamente quanta memória deveríamos dedicar à matriz de bytes, e a segunda para guardar e calcular os códigos dentro da matriz de bytes.

```
typedef struct code_{
    char* value;
    int nextbit;
    int index;
}*code;
```

4.2. FUNÇÕES PRINCIPAIS

```
code **malloc_matrix(code **matrix, int *symbols, int tamBytes)
```

Função responsável por guardar espaço numa matriz de bytes (símbolos por offsets), abrindo apenas espaço para os simbolos que foram utilizados por 8 offsets e dentro da `struct code_` no *value* guarda espaço apenas para o número de bytes necessários (`tamBytes`).

```
void free_matrix(code **matrix, int *symbols)
```

Função responsável por libertar a memória guardada pela função **`malloc_matrix`**.

```
int get_maxBits (int *symbols, FILE *fp_cod)
```

Função responsável por realizar a primeira leitura do ficheiro do tipo cod. Esta função utiliza um algoritmo simples que consiste num ciclo de leitura, até ao bloco acabar, caractere a caractere, com condições dependendo do caractere que ler.

```
void calc_matrix(code **matrix, int *symbols, int maxBits, FILE *fp_cod)
```

Função responsável por realizar a segunda leitura do ficheiro tipo cod. O algoritmo utilizado é extramente parecido ao da

função anterior com a pequena diferença, que quando este acaba de ler um código, coloca-o na matriz para todos os 8 offsets.

```
int write_shaf (code ** matrix, FILE *original, FILE *filename, int size, int sizeBytes)
```

Esta função realiza o algoritmo de codificação e escreve no ficheiro do tipo cod o resultado, retornando ainda o novo tamanho do bloco.

```
void moduloc(char *filename)
```

Por fim a função que trabalha como uma espécie de *main* do nosso módulo, responsável por abrir os ponteiros para todos os ficheiros e chamar as outras funções de maneira sequencial, realizando esse trabalho o número de vezes equivalente ao número de blocos.

4.3. TESTES

Realizamos alguns testes com os ficheiros disponibilizados pelo professor, segue em baixo uma tabela dividida pelos diferentes ficheiros com o tempo em milissegundos de execução do módulo e com o tamanho antes e depois do novo ficheiro gerado.

Nome do Ficheiro	Tempo de execução (ms)	Tamanho antes (Kb)	Tamanho depois (Kb)
aaa.txt	1	3	1
aaa.txt.rle	2	2	1
Shakespeare.txt	276	5642	3420
Shakespeare.txt.rle	252	5540	3397
bbb.zip	29282	732452	732512
bbb.zip.rle	30633	736131	738523

4.4. CONCLUSÃO

Durante a execução do módulo foram surgindo problemas que nos gerou algumas dificuldades, que nos fez optar por novas estratégias e abandonar outras.

Sem dúvida a maior desfeita foi não ter conseguido realizar a otimização por *threads* sugerida no enunciado por falta de conhecimento.

Durante boa parte do trabalho, o erro mais usual era com a memória e com os *mallocs*, de modo que optamos por fazer 2 leituras do ficheiro do tipo cod para obter a informação de quanta memória necessitávamos de dedicar ao módulo. (RishabhPrabhu, 2020)

No final do trabalho, na escrita do ficheiro do tipo shaf, aparecia-nos uns caracteres invisíveis que tinham o hexadecimal 0d e depois de uma pesquisa, descobrimos que a solução para o problema era abrir o ficheiro do tipo shaf em binário (modo antes "w+", modo depois "wb+"). (Foster, 2011)

Em suma, estamos satisfeitos com o trabalho realizado e acreditamos ter executado o trabalho que nos foi pedido da forma mais otimizada e simples que sabemos.

5. MÓDULO D

5.1. ESTRATÉGIAS E FUNÇÕES PRINCIPAIS DO MÓDULO

Numa primeira fase, começou-se por desenvolver funções de leitura dos ficheiros necessários á descodificação. A leitura de ficheiros ".freq" e ".cod" é feita caractere a caractere, já os ficheiros ".shaf" e ".rle" são lidos em modo binário por blocos de bytes.
Optamos por utilizar *multithreading* de forma a otimizar certas tarefas que são muito repetitivas, sendo estas a leitura de blocos de bytes dum ficheiro ".rle", a descompressão de um bloco de bytes dum ficheiro ".rle", e a descodificação de blocos dum ficheiro ".shaf". Tivemos de recorrer a estruturas de dados para passagem de argumentos às *threads*.

```
typedef struct argDescompBloco {

    char* filename;
    int offset;
    char* bloco;
    int tamanho;

}argDB;

typedef struct argLeBloco {

    char* filename;
    int offset;
    char* bloco;
    int tamanho;
    int *tamDescomp;

}argLB;

typedef struct argDescodSahf{
    char *filename;
    char * bloco;
    int tamanho;
    int tamanhoDescod;
    int offset;
    cArray * codigos;
    char tipo;
}argDS;
```

Numa segunda fase, desenvolvemos as funções de descodificação e descompressão: **descodShaf** e **descompBlocoRle** que descodificam e descomprimem, respetivamente, blocos de bytes previamente guardados em memória. A estratégia usada para descodificar ficheiros tipo ".shaf" consiste em guardar todos os códigos de um ficheiro ".cod" numa lista onde o índice é o decimal do próprio código. As sequências binárias do ".shaf" são previamente em memória e são posteriormente percorridas "bit" a "bit" até ler um código válido e o caractere correspondente é escrito.

Infelizmente, não conseguimos aplicar uma descodificação utilizando o modo sugerido, por matrizes de bytes, mas aplicamos um método mais rápido em termos de procura que arvore binária mas que utiliza mais memória. Por fim, utilizamos ainda uma estrutura *union* para aceder aos bits caracter a caractere.

```
typedef union
{
  struct
  {
    unsigned char bit1 : 1;
    unsigned char bit2 : 1;
    unsigned char bit3 : 1;
    unsigned char bit4 : 1;
    unsigned char bit5 : 1;
    unsigned char bit6 : 1;
    unsigned char bit7 : 1;
    unsigned char bit8 : 1;
  }u;
  unsigned char byte;
}acederB;
```

5.2. TESTES

Execução: Shakespeare.txt.shaf -m d -d s

```
==================SUCESSO==================
Goncalo Pereira & Jose Gomes,a93168,a82418, MIEI/CD
Modulo: D (descodificacao)
Numero de blocos: 9
Tamanho antes/depois do ficheiro gerado (bloco 1): 395147/655360
Tamanho antes/depois do ficheiro gerado (bloco 2): 401893/655360
Tamanho antes/depois do ficheiro gerado (bloco 3): 393101/655360
Tamanho antes/depois do ficheiro gerado (bloco 4): 394297/655360
Tamanho antes/depois do ficheiro gerado (bloco 5): 398597/655360
Tamanho antes/depois do ficheiro gerado (bloco 6): 397092/655360
Tamanho antes/depois do ficheiro gerado (bloco 7): 399884/655360
Tamanho antes/depois do ficheiro gerado (bloco 8): 397911/655360
Tamanho antes/depois do ficheiro gerado (bloco 9): 323897/533814
Tempo de execucao do modulo: 1 segundos e 554 milisegundos
Ficheiro gerado: Shakespeare.txt
```

Execução: Shakespeare.txt.rle -m d -d r

```
====================SUCESSO====================
Goncalo Pereira & Jose Gomes,a93168,a82418, MIEI/CD
Modulo: D (descodificacao)
Numero de blocos: 9
Tamanho antes/depois do ficheiro gerado (bloco 1): 638735/655360
Tamanho antes/depois do ficheiro gerado (bloco 2): 649109/651276
Tamanho antes/depois do ficheiro gerado (bloco 3): 638689/651294
Tamanho antes/depois do ficheiro gerado (bloco 4): 640258/655360
Tamanho antes/depois do ficheiro gerado (bloco 5): 645068/651275
Tamanho antes/depois do ficheiro gerado (bloco 6): 640172/655360
Tamanho antes/depois do ficheiro gerado (bloco 7): 645071/655360
Tamanho antes/depois do ficheiro gerado (bloco 8): 644522/655360
Tamanho antes/depois do ficheiro gerado (bloco 9): 530660/533814
Tempo de execucao do modulo: 0 segundos e 954 milisegundos
Ficheiro gerado: Shakespeare.txt
```

Execução: Shakespeare.txt.rle.shaf -m d

```
==================SUCESSO==================
Goncalo Pereira & Jose Gomes,a93168,a82418, MIEI/CD
Modulo: D (descodificacao)
Numero de blocos: 9
Tamanho antes/depois do ficheiro gerado (bloco 1): 395847/655360
Tamanho antes/depois do ficheiro gerado (bloco 2): 405344/655360
Tamanho antes/depois do ficheiro gerado (bloco 3): 396650/655360
Tamanho antes/depois do ficheiro gerado (bloco 4): 398836/655360
Tamanho antes/depois do ficheiro gerado (bloco 5): 402135/655360
Tamanho antes/depois do ficheiro gerado (bloco 6): 400419/655360
Tamanho antes/depois do ficheiro gerado (bloco 7): 403721/655360
Tamanho antes/depois do ficheiro gerado (bloco 8): 401581/655360
Tamanho antes/depois do ficheiro gerado (bloco 9): 324603/533814
Tempo de execucao do modulo: 2 segundos e 519 milisegundos
Ficheiro gerado: Shakespeare.txt
```

5.3. CONCLUSÃO

Durante a implementação do nosso módulo, tivemos alguns percalços:

- Dificuldade a ler ficheiros de tamanho muito grande;
- Aprender a usar *threads*;
- Dificuldade a perceber como aplicar descodificação por matrizes de bytes.

Referências

AsciiCode. (s.d.). Obtido de https://www.ascii-code.com/

Foster, J. (4 de 4 de 2011). *StackOverflow*. Obtido de https://stackoverflow.com/questions/5537066/strange-0x0d-being-added-to-my-binary-file

Geeksforgeeks. (7 de Março de 2019). Obtido de https://www.geeksforgeeks.org/measure-execution-time-with-high-precision-in-c-c/

Guru99. (s.d.). Obtido de https://www.guru99.com/c-file-input-output.html

RishabhPrabhu. (24 de 5 de 2020). *Geeksforgeeks*. Obtido de https://www.geeksforgeeks.org/dynamic-memory-allocation-in-c-using-malloc-calloc-free-and-realloc/

StackOverflow. (2013). Obtido de https://stackoverflow.com/questions/16696297/ftell-at-a-position-past-2gb

StackOverflow. (2017). Obtido de https://pt.stackoverflow.com/questions/215477/como-usar-os-par%C3%A2metros-na-linha-de-comando

Superuser. (2011). Obtido de https://superuser.com/questions/294219/what-are-the-differences-between-linux-and-windows-txt-files-unicode-encoding)

TutorialsPoint. (s.d.). Obtido de https://www.tutorialspoint.com/c_standard_library/c_function_malloc.htm

Wiki Portugal a programar. (14 de 5 de 2018). Obtido de https://wiki.portugal-a-programar.pt/dev_geral:c:ficheiros