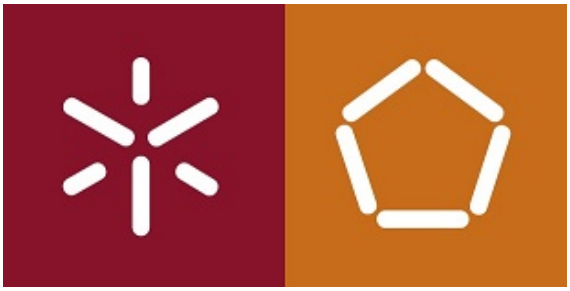

Universidade do Minho
Departamento de Informática

Codificação Shannon-Fano em C

MIEI - Comunicação de Dados

Grupo 21

3 de Janeiro, 2021


Alexandre Flores
(a93220)

Guilherme Fernandes
(a93216)


Rita Lino
(a93196)


Mariana Rodrigues
(a93229)


Matilde Bravo
(a93246)


Miguel Gomes
(a93294)

Contents

1 Introdução **3**

2 Parte A - Módulo F **4**
2.1 Implementação 4
2.1.1 Estruturas de dados do nosso ficheiro por blocos 4
2.1.2 Função Principal 6
2.1.3 Guardar um ficheiro na memória por blocos 6
2.1.4 Cálculo das Frequências 7
2.1.5 Compressão RLE 13
2.1.6 Finalização 15

3 Parte B - Módulo T **16**
3.1 Implementação 16
3.1.1 Estruturas Usadas 16
3.1.2 Algoritmo Principal 16
3.2 Limitações 19

4 Parte C - Módulo C **20**
4.1 Introdução 20
4.2 Implementação 20
4.2.1 Leitura do Ficheiro cod 20
4.2.2 Otimizão com Matriz de Bytes 21
4.2.3 Escrita no ficheiro Shaf 22

5 Resultados **24**
5.1 Ficheiros Usados 24
5.2 Estatísticas de Execução 25
5.2.1 Módulo F 25
5.2.2 Módulo T 26
5.2.3 Módulo C 26

6 Considerações Finais **27**

Capítulo 1

Introdução

Neste projeto o nosso grupo implementou codifição e descodificação *Shannon-Fano*. Dividimo-nos em 4 grupos de forma a cada um concretizar uma parte do trabalho, com a exceção do relatório para o qual todos os grupos contribuíram. Desta forma, o projeto foi organizado do seguinte modo:

Parte A foi realizada por **Mariana Rodrigues** e **Miguel Gomes**

Parte B foi realizada por **Alexandre Flores** e **Rita Lino**

Parte C foi realizada por **Matilde Bravo**

Parte D foi realizada por **Guilherme Fernandes** e será entregue mais tarde

Apesar desta divisão, mantivemos contacto entre todos de forma a evitar redundâncias e dar sugestões e críticas ao código.

Em relação aos resultados documentados nas várias secções deste relatório, todos os ficheiros *.freq, .cod* e.*shaf* foram gerados pelos nossos módulos. Isto implica que quaisquer limitações de um módulo inicial, como por exemplo do módulo *f* para o módulo t, possam vir a ser relevantes num módulo que seja utilizado depois desse.

Capítulo 2

Parte A - Módulo F

2.1 Implementação

O Módulo F encontra-se responsável por conceber uma pré-compressão simples, caso se verifique que compense fazê-lo, utilizando o mecanismo ***Run-Lenght Encoding*** (RLE).

Este módulo receberá um dado ficheiro de entrada e irá guarda-lo por blocos na memória. Um bloco poderá ter um tamanho máximo de 64Kbytes(por defeito), 640Kbytes, 8Mbytes ou 64Mbytes, o que é definido por uma 'flag' na execução do programa.

Para tal, decidimos implementar diversas structs que nos irão ajudar a guardar a informação necessária para a implementação do nosso módulo.

2.1.1 Estruturas de dados do nosso ficheiro por blocos

```
/* Simbolos relativos ao bloco sem compressao. */
typedef struct ByteVec {
        uint8_t *vec; /* Array dos simbolos. */
        size_t used;  /* Numero de elementos usados no array. */
        size_t size;  /* Tamanho total do array. */
} ByteVec;

/* Compressao de um simbolo. */
typedef struct ByteTupple {
        uint8_t byte;  /* Simbolo. */
        uint8_t count; /* Numero de ocorrencias. */
} ByteTupple;

/* Simbolos relativos ao bloco com compressao. */
typedef struct TuppleVec {
        ByteTupple *vec; /* Array dos simbolos. */
        size_t used;     /* Numero de elementos usados no array. */
        size_t n_used;   /* Tamanho do array sem compressao. */
        size_t size;     /* Tamanho total do array. */
} TuppleVec;
```

```c
/* Estrutura para os blocos sem compressao. */
typedef struct block {
  /* Array dinamico */
  ByteVec *blocklist;
  /* Tamanho do bloco */
  unsigned int block_size;
  /* Apontador para o proximo bloco */
  struct block *prox;
} Blocks;

/* Estrutura para os blocos com compressao.*/
typedef struct block_c {
  /* Array dinamico */
  TuppleVec *tBList;
  /* Tamanho do bloco */
  unsigned int block_size;
  /* Apontador para o proximo bloco */
  struct block_c *prox;
} Blocks_C;
```

```c
/* Estrutura para os blocos sem compressao. */
typedef struct block {
  /* Array dinamico */
  ByteVec *blocklist;
  /* Tamanho do bloco */
  unsigned int block_size;
  /* Apontador para o proximo bloco */
  struct block *prox;
} Blocks;

/* Estrutura para blocos comprimidos */
typedef struct block_c {
  /* Array dinamico */
  TuppleVec *tBList;
  /* Tamanho do bloco */
  unsigned int block_size;
  /* se compression_type == N -> block_c = NULL */
  struct block_c *prox;
} Blocks_C;

/* Estrutura para guardar um ficheiro em blocos na memoria */
typedef struct block_file {
  /* Compression_type */
  enum compression compression_type;
  /* Numero de blocos. */
  unsigned int num_blocks;
  Blocks *blocks;
  /* se compression_type == N -> block_c = NULL */
  Blocks_C *blocks_c;
} BlockFiles;
```

2.1.2 Função Principal

Este módulo gira em torno desta nossa função principal:

```
int module_f(char const *filename, size_t const the_block_size,
             int FORCE_FLAG);
```

- module_f → Funçao que executa todo o MODULE_F.
- *filename → Ficheiro a ser analisado.
- the_block_size → Tamanho do bloco a ser analisado.
- FORCE_FLAG → Flag que determina se a compressão RLE é forçada.

Primeiramente, começámos por inicializar a contagem do tempo de execução deste módulo e a obtenção de alguns dados importantes para a construção da nossa estrutura de dados. Tais como o número total de blocos, o tamanho dos blocos e o tamanho do último bloco. Para isso, usamos uma função fornecida pelo professor Bruno Dias, na qual fizemos pequenas alteracões.

```
/* Function fsize() to get the size of files
and the number of blocks contained
in the file (for a given block size).
 */
size_t fsize(FILE *fp_in, char const *filename,
        size_t *the_block_size,size_t *size_of_last_block);
```

(c) 2020, Bruno A.F. Dias - University of Minho, Informatics Department

2.1.3 Guardar um ficheiro na memória por blocos

Após termos recolhido estas informações, encontramo-nos aptos para conseguirmos passar o ficheiro recebido para a memória. Para isso, usamos a seguinte função:

```
size_t building_blocks(FILE *file, BlockFiles *self,
                       size_t n_blocks, size_t size_last_block,
                       size_t block_size);
```

- *file → Ficheiro a ser analisado.
- *self → Iremos guardar a nossa estrutura de dados aqui.
- n_blocks → Número total de blocos.
- size_last_block → Tamanho do último bloco.
- the_block_size → Tamanho do bloco a ser analisado.

Dado que o tamanho do último bloco a ser analisado poderá ser diferente dos restantes blocos, tivemos que ter isso em consideração.

A maneira como armazenamos a informação do bloco na nossa estrutura de dados é efetuada a partir da **ByteVec *loadArray(FILE *file, size_t block_size)**.

```
ByteVec *loadArray(FILE *file, size_t block_size) {
  size_t i = 0;
  ByteVec *self = byte_vec_new();
  uint8_t x = 0;
  while (i < block_size && (fread(&x, sizeof(x), 1, file) == 1)) {
    byte_vec_push(self, x);
    i++;
  }
  return self;
}
```

Esta função irá percorrer o ficheiro dado, byte a byte , lendo-o e colocando-o num array dinâmico, tendo sempre em atenção o tamanho do bloco dado.

2.1.4 Cálculo das Frequências

Tendo já sido convertido o ficheiro recebido em blocos e guardado na nossa estrutura de dados, iremos calcular as frequências de cada um dos símbolos de cada bloco. Para conseguirmos guardar esta informação, decidimos que seria melhor criar uma nova estrutura de dados que nos auxiliasse nesse processo.

```
typedef struct freq_block {
  int freq[uint_range];
  struct freq_block
      *prox;
} FreqBlock;
```

- FreqBlock → Frequências para os blocos.
- freq[uint_range] → Freqência relativa ao bloco.
- *prox → Apontador para as frequências do bloco. seguinte.

Podemos observar que a cada bloco irá ser atribuído um array de ***256*** valores. Para ajudar na compreensão desta estrutura, iremos dar alguns exemplos:

* freq[2]=10 -> A frequência do símbolo 2 é 10.

* freq[20]=0 -> A frequência do símbolo 20 é 0.

Sabendo como as estruturas de dados se encontram definidas, é necessário perceber como se calculam as frequências. Visto que podemos ou não efetuar a compressão ***rle*** e que as estruturas dos blocos com compressão e sem compressão diferem, decidimos elaborar duas funções diferentes.

Cálculo das frequências dos blocos sem compressão

Comecemos pelo cálculo das frequências dos blocos sem compressão ***(FreqBlock *calFreq(BlockFiles const *file))***:

```
/*Vamos garantir que o array encontra-se todo a 0.*/
for (; i < uint_range; i++)
  array[i] = 0;
/* 1 bloco */
used = byte_vec_used(vec);
for (i = 0; i < used; i++)
  array[byte_vec_index(vec, i)]++;
blocks = blocks->prox;
num++;
/* Inicializamos a nossa estrutura das
   frequencias atraves do array obtido */
FreqBlock *freq = initializeFreq(array);
/* Restantes Blocos. */
while (num <= num_blocks && blocks) {
  vec = blocks->blocklist;
  used = byte_vec_used(vec);
  /* Vamos garantir que o array encontra-se todo a 0. */
  for (i = 0; i < uint_range; i++)
    array[i] = 0;
  /* Vamos preencher o array com as frequencias de um bloco.*/
  for (i = 0; i < used; i++)
    array[byte_vec_index(vec, i)]++;
  /* Adicionar as frequencias ao nosso BlockFiles. */
  arrayToFreqBlock(array, freq);
  num++;
  blocks = blocks->prox;
}
```

Neste pedaço de código, pode-se perceber que se começa por calcular as frequências dos símbolos do primeiro bloco, a fim de conseguirmos inicializar a nossa estrutura das frequências. Após termos feito isso, falta-nos efetuar o mesmo raciocínio para os restantes blocos.

Para enteder melhor como isso é feito, iremos analisar melhor esta parte do código:

```
/* Vamos preencher o array com as
   frequencias de um bloco. */
    for (i = 0; i < used; i++)
      array[byte_vec_index(vec, i)]++;
/* Adicionar as frequencias ao nosso BlockFiles. */
    arrayToFreqBlock(array, freq);
```

- *used* → Número de elementos que o nosso bloco contém.
- *array[uint_range]* → Array[256] inicializado a 0.
- *byte_vec_index(vec, i)* → Irá returnar o elemento que estiver na posição i do nosso array dinámico.
- *array[byte_vec_index(vec, i)]++* → Iremos incrementar um à frequência do símbolo obtido em `byte_vec_index(vec, i)`.

Cálculo das frequências dos blocos com compressão

Neste ponto, iremos analisar como é efetuado o cálculo das frequências dos blocos com compressão ***(FreqBlock *calFreq_RLE(BlockFiles *file))***. Teremos em mente que este cálculo das frequências dos blocos com compressão será só efetuado após a compreesão rle do ficheiro recebido; caso esta não seja realizada não se efetuará o cálculo das frequências. Posteriormente, iremos esclarecer como foi feita a compreensão **rle** do ficheiro por blocos.

Tendo percebido como foi calculado as frequências de cada bloco sem compressão, basta-nos efetuar o mesmo raciocínio com pequenas modificações.

```
for (i = 0; i < used; i++) {
    /* Variaveis auxiliares para nao estar-mos
       sempre a ir a memoria. */
    aux = tupple_vec_index(vec, i);
    /* Numero de vezes que o simbolo ocorre */
    count = aux.count;
    /* Simbolo */
    byte = aux.byte;

    array[byte]++;
    /* Caso em que o nosso simbolo e o 0. */
    if (byte == 0) {
     array[0]++;
     array[count]++;
     block_size += 2;
    } else {
/* Caso em que o nosso simbolo ocorre mais do que 3 vezes. */
      if (aux.count > 3) {
        array[0]++;
        array[count]++;
       } else if (count == 2 || count == 3) {
        /* Caso em que o nosso simbolo ocorre entre
           2 ou 3 vezes. */
          if (count == 3) {
            array[byte] += 2;
          } else {
            array[byte]++;
          }
      }
  }
}
```

Escrita das Frequências

A este ponto, tendo já sido as frequências dos blocos calculadas, só nos falta imprimí-las num ficheiro de output do tipo ***freq***. Para isso, usamos a função seguinte:

```
int writeFreq(FILE *fp_in, const char *filename,
              BlockFiles *BlockFile, FreqBlock *freq) {
```

- **fp_in* → Ficheiro de output.
- *filename* → Nome do ficheiro de output.
- *BlockFile* → Dados guardados relativos ao ficheiro inicial dado.
- *freq* → Frequências que queremos imprimir.

Esta função começa por determinar se houve ou não a compressão ***rle*** e o número de blocos totais.

```
  /* Verificar se o BlockFile encontra-se vazio */
  if (BlockFile != NULL)
    n_blocks = BlockFile->num_blocks;
  else
    return WriteFreq_ERROR_IN_BLOCKFILES;

  /* Compression_type */
  char compression_type;
  if (BlockFile->compression_type == NOT_COMPRESSED)
    compression_type = 'N';
  else
    compression_type = 'R';
```

Após termos essas informações iremos imprimí-las.

```
/* @<R|N>@NumerodeBlocos */
  fprintf(fp, "%c%c%c%lld", uint_Arroba, compression_type,
      uint_Arroba, n_blocks);
```

Sendo assim, falta-nos imprimir as frequências de cada um dos blocos. Para isso iremos usar um loop.

```
/* write the frequency of a block */
i = 0;
int num_freq;
while ((aux_Blocks || aux_Blocks_C) && aux_Freq && i < n_blocks) {
  if (compression_type == 'N')
    block_size = aux_Blocks->block_size;
  else
    block_size = aux_Blocks_C->block_size;
  /* @[tamanho_do_bloco]@ */
  fprintf(fp, "%c%ld%c", uint_Arroba, block_size,
      uint_Arroba);
  j = 0;
  /* Write the frequencies up to the symbol 255 */
  int count = uint_range - 1;
  /* Simbolo 0 ate ao 254 */
  while (j < (count)) {
    num_freq = aux_Freq->freq[j];
    /* Verificar se o valor da frequencia e
       igual ao do simbolo anterior. */
    if (last != num_freq) {
      fprintf(fp, "%d;", num_freq);
      last = num_freq;
    } else
      fprintf(fp, ";");
    j = j + 1;
  }
  /* Simbolo 255 */
  num_freq = aux_Freq->freq[j];
  /* Verificar se o valor da frequencia e
     igual ao do simbolo anterior. */
  if (last != num_freq)
    fprintf(fp, "%d", num_freq);
  i = i + 1;
  aux_Freq = aux_Freq->prox;
  aux_Blocks = aux_Blocks->prox;
  if (compression_type == 'R')
    aux_Blocks_C = aux_Blocks_C->prox;
  last = -1;
}
```

Podemos observar que começámos por escrever o tamanho do bloco. Em seguida, iremos imprimir as frequências dos símbolos de 0 até ao 254, visto que o símbolo 255 só irá ser escrito caso seja diferente do símbolo anterior.

Feito isto, basta-nos completar o ficheiro de output com *@0* e fechá-lo.

```
/* @0 */
fprintf(fp, "%c0", uint_Arroba);
if (aux_Freq == NULL && i != n_blocks)
  return WriteFreq_ERROR_IN_FREQ;
fclose(fp);
```

2.1.5 Compressão RLE

A compressão ***RLE*** total do ficheiro só será gerada caso se note que a compressão do primeiro bloco é superior a 5%, caso contrário nenhum ficheiro do tipo ***.rle*** será gerado, nem iremos efetuar o cálculo das frequências, passando estes passos à frente.

É de referir que, se o utilizador assim o desejar, pode forçar a utilização de compressão ***RLE*** através dum argumento opcional, mesmo que a compressão do primeiro bloco seja inferior a 5%.

Compressão RLE

Relembrando que os símbolos de cada bloco se encontram guardados na memória da seguinte forma:

```
/* Compressao de um simbolo. */
typedef struct ByteTupple {
        uint8_t byte;  /* Simbolo. */
        uint8_t count; /* Numero de ocorrencias. */
} ByteTupple;

/* Simbolos relativos ao bloco com compressao. */
typedef struct TuppleVec {
        ByteTupple *vec; /* Array dos simbolos. */
        size_t used;     /* Numero de elementos usados no array. */
        size_t n_used;   /* Tamanho do array sem compressao. */
        size_t size;     /* Tamanho total do array. */
} TuppleVec;
```

A estratégia que decidimos implentar foi agrupar os bytes seguidos de cada um dos blocos, independentemente do número de ocorrências. Para esse fim, criou-se a funçao **size_t compress_blocks(BlockFiles *self, size_t FORCE_FLAG)**. Esta função é auxiliada pela **TuppleVec *compress(ByteVec const *self)** que efetua a compressão de apenas um bloco.

```
TuppleVec *compress(ByteVec const *self) {
  TuppleVec *t = tupple_vec_new();
  size_t i = 1, count = 1, used = self->used;
  uint8_t last = byte_vec_index(self, 0);
  while (i < used) {
    if (last == byte_vec_index(self, i) && count < UCHAR_MAX) {
            count++;
    } else {
      tupple_vec_push(t, last, count);
      last = byte_vec_index(self, i);
      count = 1;
    }
    i++;
  }
  tupple_vec_push(t, last, count);
  t->n_used = self->used;
  return t;
}
```

Escrita da compressão RLE

No caso de se ter gerado a compressão ***RLE*** gerar-se-á um ficheiro de output do tipo ***rle.freq*** , usando a seguinte função:

```
int write_compressed(FILE *file, BlockFiles const *self)
```

Neste ponto, iremos escrever em loop cada um dos blocos.

Temos que ter em atenção que todos os símbolos com mais que 3 ocorrências e o símbolo {*0*} têm que ser escritos no seguinte formato:

0{símbolo}{número de ocorrências}

Sendo assim:

```
while (list) {
  yes = list->tBList;
  used = tupple_vec_used(yes);
  for (i = 0; i < used; i++) {
    temp = 0;
    b = tupple_vec_index(yes, i);
    x = b.byte;
    y = b.count;
    if (y > 3 || x == 0) {
      fputc(0, file);
      fputc(x, file);
      fputc(y, file);
    } else
      while (temp != y) {
        fputc(x, file);
        temp++;
      }
  }
  list = list->prox;
}
```

- ****list*** → É um *Blocks_C ** onde estará guardada a informação de todos os blocos.
- ****yes*** → É um *TuppleVec ** que irá conter a informação relativa ao bloco a ser analisado.

2.1.6 Finalização

Posteriormente a isto tudo iremos terminar o tempo de execução, imprimir os dados relativos ao nosso módulo e libertar todos os espaços alocados da memória.

```
/* End time */
Ticks[1] = clock();
/* Calcular o tempo de execucao do MODULE F */
double time;
time=(Ticks[1] - Ticks[0])*1000.0 / CLOCKS_PER_SEC;

/* Apresentar menu final relativo ao modulo. */
print_module_f(filename, self, blocks, time, block_size,
               size_last_block, size_block_rle, size_last_rle);
/* Libertar o espaco alocado. */
free_Freq(freq_file);
free_Blocks_file(self);
```

Capítulo 3

Parte B - Módulo T

3.1 Implementação

Este módulo é responsável pela geração de um ficheiro ***.cod*** a partir de um ficheiro de frequências que contém o código *Shannon-Fano* relativo ao ficheiro fornecido.

3.1.1 Estruturas Usadas

Antes da implementação do algoritmo, implementamos uma *struct* que nos vai auxiliar a registar informações sobre os símbolos e, subsequentemente, trabalhar com eles. Esta *struct*, a que chamámos *symbol*, guarda informação sobre o valor inteiro do símbolo, a sua frequência no bloco que estivermos a analisar, e o seu código *Shannon-Fano*.

```
typedef struct symbol {
  int symbolID;
  int freq;
  char *code;
} Symbol;
```

3.1.2 Algoritmo Principal

Em relação ao algoritmo no módulo em si, começamos por inicializar o relógio de tempo de execução e o ficheiro de output. Copiamos os primeiros 2 parâmetros do ficheiro de frequências ***.freq*** ('@N@2@', por exemplo) para o ficheiro de output, o novo ficheiro de código (***.cod***). Registamos também o número total de blocos que vamos analisar.

É importante reconhecer que tanto o ficheiro ***.cod*** como o ficheiro ***.freq*** são mantidos abertos durante a execução do resto do algoritmo. Desta forma, à medida que lemos e codificamos um bloco de símbolos, avançamos os ficheiros de forma a não termos de nos voltar a posicionar nestes mesmos.

Já inicializado o ficheiro ***.cod***, podemos começar efetivamente a criar o código para os blocos de símbolos. Um ciclo *for* que é executado *n_blocks* vezes (onde *n_blocks* é o número total de blocos que obtivemos do ficheiro .freq) vai ler e codificar um bloco de símbolos de cada vez.

```
for (i = 0; i < n_blocks; i++) {
    Symbol *symbol_table =
      (Symbol*) malloc(256*sizeof(Symbol));
    initialize_table (symbol_table, 256);

    int size = read_block (freqs_file, symbol_table);
    qsort(symbol_table, 256, sizeof(Symbol),
      compare_freqs);
    int symbols = get_used_symbols(symbol_table);
    create_shafa_code(symbol_table, 0, symbols);
    qsort(symbol_table, 256, sizeof(Symbol),
      compare_symbolID);

    write_code_block(code_file, size, symbol_table);

    block_sizes[i == n_blocks - 1] = size;

    int j;
    for (j = 0; j < 256; j++) {
        free(symbol_table[j].code);
    }
    free(symbol_table);
}
```

Este ciclo começa por inicializar uma tabela de 256 símbolos, que é representada por um *array* do nosso **struct symbol**. A tabela é incializada com todos os valores dos símbolos (**symbolID**) em ordem crescente. Os códigos *Shannon-Fano* são alocados na memória e definidos como uma *string* vazia, e as frequências não são inicializadas.

Após incializada a tabela, é lido o próximo bloco no ficheiro ***.freq***. São escritas as frequências na tabela de símbolos e é guardado também o tamanho do bloco em *bytes* na variável ***size***. É posteriormente organizada a tabela por ordem decrescente das frequências, para facilitar o algoritmo de criação do código *Shannon-Fano*. Para isto, e também para futuras reorganizações da tabela, utilizamos a função **qsort**, incluída na **stdlib.h**. Também verificamos a quantidade de símbolos que têm frequência não nula, para passarmos apenas os *n* primeiros símbolos da tabela (que correspondem aos símbolos de frequência não nula) à função de codificação dos símbolos.

Em relação ao algoritmo de codificação, este funciona através de uma função recursiva que gere a divisão dos grupos e acrescentação dos *bits* do código às *strings* do código dos símbolos. Esta função também recorre a outra função auxiliar que encontra o pivot do grupo de símbolos, onde ocorre a divisão ótima das frequências de modo a obter 2 sub-grupos de símbolos com frequências totais semelhantes.

```
void create_shafa_code (Symbol *symbol_table,
  int start, int end) {
    /* Check if dimension is only 2 elements */
    if (end - start == 2)
        append_bits(symbol_table, start + 1,
          start, end);
    else {
        /* Find the index of the first element of
           the second subgroup and append
           the '0' and '1' to the subgroups */
        int p = freq_split(symbol_table,
          start, end);
        append_bits(symbol_table, p, start, end);

        /* Recursively code subgroups for the
           subgroups we just calculated */
        if (p - start > 1)
            create_shafa_code(symbol_table,
              start, p);

        if (end - p > 1)
            create_shafa_code(symbol_table,
              p, end);
    }
}
```

O algoritmo começa por verificar se é necessário utilizar a função auxiliar para encontrar o pivot ótimo do grupo. Se o grupo tem apenas 2 símbolos basta imediatamente acrescentar os *bits* '0' e '1' ao primeiro e segundo código dos símbolos, respetivamente. Caso a dimensão do grupo de símbolos indicado for maior que 2, recorremos a função auxiliar **freq_split** que encontra o pivot ideal.

Encontrado o pivot, são adicionados os bits relevantes aos dois sub-grupos e a função **create_shafa_code** é chamada novamente, desta vez passando cada um dos sub-grupos calculados (Excetuando o caso em que um sub-grupo tenha apenas 1 elemento, em que damos a codificação como terminada para esse sub-grupo).

De volta ao ciclo principal, uma vez calculado o código de todos os símbolos de frequência não nula, reorganizamos a tabela por ordem crescente do valor dos símbolos, para facilitar a escrita dos códigos. Os códigos são então escritos no ficheiro ***.cod*** e registamos o tamanho do bloco que acabamos de codificar para depois o imprimirmos nas estatísticas de *runtime*. Para finalizar, libertamos a memória alocada para os símbolos, terminando assim uma iteração do ciclo principal do programa.

Após todas as iterações do ciclo principal podemos finalizar o ficheiro ***.cod***. Acrescentamos o bloco nulo no final do ficheiro ('@0'), paramos o relógio de tempo de execução, imprimimos as estatísticas de *runtime* (tendo em conta que previamente guardamos em variáveis locais o tamanho e quantidade dos blocos) e fechamos tanto o ficheiro de frequências como de código.

3.2 Limitações

Devido ao nosso uso da função **qsort** da **stdlib.h**, encontramos diferenças entre o funcionamento do programa no Linux e em outras plataformas. Especificamente, em casos em que dois símbolos têm a mesma frequência, reparamos que estes se encontram em posições trocadas quando organizamos a tabela por ordem decrescente de frequências. Desta forma, símbolos que tenham a mesma frequência estão sujeitos a terem códigos *Shannon-Fano* trocados entre plataformas.

Temos como exemplo os seguintes elementos do segundo bloco de código gerado a partir do ficheiro de teste *bbb.zip.freq*:

Frequência	Símbolo	Plataforma	Código
34956	101	Linux	01010100
		Windows	01010101
	154	Linux	01010101
		Windows	01010100

Capítulo 4

Parte C - Módulo C

4.1 Introdução

Este módulo tem como objetivo gerar o ficheiro final comprimido, **.shaf**, através da leitura do ficheiro **.cod** e do ficheiro original.

4.2 Implementação

4.2.1 Leitura do Ficheiro cod

O algoritmo implementado começa por efetuar a leitura do ficheiro **.cod** correspondente, recorrendo a uma função que lê um bloco de cada a vez. Naturalmente, é necessário armazenar a informação lida em algum tipo de estrutura. Portanto, neste módulo, optou-se por criar uma *struct* principal, designada por **FullSequence** que conterá todos os dados relevantes que resultam da leitura deste ficheiro.

```
typedef struct sequence {
    size_t number_blocks;
    Block * blocks;
} FullSequence;
```

Como se pode constatar, esta estrutura contém, por sua vez, um array de outra estruturas denominadas **Block**. Ora, isto deve-se ao facto de um ficheiro, à partida, ter vários blocos e, portanto, a **FullSequence** terá que ter informação relativa a cada um deles. Segue-se a declaração da *struct* **Block**:

```
typedef struct {
    size_t block_size_before;
    size_t block_size_after;
    uint16_t number_symbols;
    size_t biggest_code_size;
    Piece * matrix;
    SFTuple symbol_dictionary[DICT_SIZE];
} Block;
```

- **block_size_before** → Tamanho do bloco no ficheiro original;
- **block_size_after** → Tamanho do bloco no ficheiro `.shaf`;
- **number_symbols** → Número de símbolos diferentes presentes no bloco;
- **matrix** → Matriz de bytes a ser utilizada na otimização;
- **symbol_dictionary** → Array que contém o código SF e o índice atribuído a cada símbolo presente no ficheiro.

O **symbol_dictionary** é um array de **256** (número de símbolos possíveis) **SFTuple**. Um **SFTuple** é essencialmente um par constituído por uma *string*, a qual contém o código **Shannon-Fano** de um dado símbolo, e um inteiro de **8 bits** que corresponde ao índice desse símbolo. Este índice indica que se trata do n-ésimo símbolo diferente presente no bloco, por ordem crescente de valor, e apenas servirá para facilitar o processo de otimização que será explicado mais à frente.

```
typedef struct {
    char* sf_code;
    uint8_t index;
} SFTuple;
```

À medida que um código é encontrado durante a leitura de um bloco, é criado um `SFTuple` com uma string correspondente ao código SF desse símbolo e com o seu índice, que apenas resulta da incrementação do índice do símbolo anterior encontrado. Após criado esse tuplo, este é adicionado ao array **symbol_dictionary**. O dicionário está indexado segundo o valor do símbolo, de modo que, as posições do array com valor igual ao símbolos que não ocorrem no ficheiro, não conterão valores úteis. Porém, esta implementação permite que seja **sempre** utilizado um array com tamanho fixo de **4096** bytes, uma vez que `DICT_SIZE * sizeof(SFTuple) = 4096`. Além disso, o tempo de acesso a cada símbolo é sempre constante, uma vez que está indexado segundo o valor do símbolo e não é necessário percorrer o array.

4.2.2 Otimizão com Matriz de Bytes

No fim da leitura de um bloco, é sempre criada a matriz de bytes que será utilizada na codificação. Esta matriz corresponde a um array de **offsets**. Em primeiro lugar, é inicializada a matriz e é construído o primeiro *offset*. Os restantes são criados a partir da manipulação dos **offsets** anteriores. Cada *offset* contém várias estruturas, às quais foi dado o nome de **Piece**. As peças são estruturas que contêm as unidades elementares utilizadas na codificação propriamente dita, **code**, bem como informação relevante que permitirá obter e "fundir" o **code** seguinte.

```
typedef struct piece {
    uint8_t* code;
    size_t next;
    size_t byte_index;
} Piece;
```

Para representar o **code**, decidiu-se utilizar um array de `uint8_t`, números de 8 bits sem sinal, que terá tamanho **CODE_MAX_SIZE**. Este tamanho é calculado do seguinte modo:
`size_t CODE_MAX_SIZE = (((block->biggest_code_size - 1) | 7 ) + 1 + 8) / 8;`

Desta forma, é calculado o número de bits múltiplo de 8 mais próximo do número de bits do tamanho do maior código *Shannon-Fano* atribuído. Se este já for um múltiplo de 8, então obtém-se o próprio valor. A esse valor são somados 8 bits e, de seguida, divide-se por 8 para se obter o tamanho máximo em **bytes**. Na função `start_matrix` é determinado este valor para que se possa criar o offset 0. Ainda nesta função, é alocado espaço para os restantes offsets e os seus campos são inicializados a zero. Na função de otimização propriamente dita, vão ser realizados os shifts necessários para criar os restantes offsets, recorrendo a várias chamadas da função `shift_piece`. Dentro de cada offset, existe uma peça correspondente a cada um dos símbolos. Por exemplo, a peça correspondente ao símbolo 30 no offset 3, consiste na peça da posição 30 do offset 2, após terem sido realizados alguns shifts ao **code** e terem sido feitos ajustes aos restantes campos da estrutura. Como já foi mencionado, a cada símbolo é atribuído um índice a ser utilizado na otimização. A forma como este índice é atribuído está relacionada com o valor do símbolo. Portanto, o menor símbolo presente no ficheiro ficará com índice 0, o segundo menor com índice 1 e assim sucessivamente. Naturalmente, apenas símbolos com frequência superior a 0 terão um índice atribuído. Desta forma, após a leitura do ficheiro cod, a função `start_matrix` recebe um array chamado `symbols` que apenas contém o valor dos símbolos presentes no ficheiro, o qual é indexado por este índice que foi atribuido. Consequentemente, é criado o primeiro offset percorrendo uma única vez um array com tamanho igual ao número de símbolos diferentes presentes no ficheiro de *input*.

A função `shift_piece` recebe a peça do offset anterior ao atual e começa por percorrer o array `code` pelo fim. Para cada `uint8_t` do array, exceto o primeiro, a função realiza um shift para a direita. Além desse shift, é necessário que o último bit do número anterior, o qual vai "desaparecer", passe a ser o primeiro bit do número atual. No caso de o bit final anterior ser igual a 1, terá que ser somado 2^7 ao número atual. Caso contrário, pode-se somar 0 ou não efetuar soma nenhuma. Como tal, optou-se por fazer um *bitwise and* sobre o número anterior, de forma a determinar o seu último bit e, de seguida, multiplicar esse bit por 2^7, isto é, realizar 7 shifts para a esquerda. Esse resultado, é então somado com o número atual. Segue-se o ciclo *for* principal da função `shift_piece`:

```
for (size_t i = previous->byte_index;; i--) {
    current->code[i] = (previous->code[i] >> 1);
    if (i == 0) break;
    current->code[i] += (previous->code[i - 1] & 1) << 7;
}
```

4.2.3 Escrita no ficheiro Shaf

Uma vez criada a matriz com todos os offsets, basta apenas ler o ficheiro original byte a byte, procurar a peça da matriz a ser utilizada para o símbolo lido e "fundir" , se possível, o respetivo `code` com o anterior, recorrendo a uma soma. Caso não seja possível, isto é, o código anterior extendia-se até ao último bit, o novo code é simplesmente acrescentado. Este processo é feito recorrendo à criação de uma array (buffer). Desta forma, o bloco codificado é todo guardado em memória, num array de carateres dinâmico, e é feita apenas uma escrita no ficheiro por bloco, no fim da codificação. Segue-se parte do código da função `write_block`:

```
for (size_t i = 0; i <= piece->byte_index; i++) {
    if (piece->next == 0 && piece->byte_index == i) {
        break;
    }
    else {
        if (end) {
            byte_vec_push(vec, piece->code[i]);
        }
        else {
            vec->vec[byte_vec_used(vec) - 1] += piece->code[i];
            end = i < (piece->byte_index);
        }
    }
}
```

A variável `end` representa precisamente se o código anterior se extendeu até ao último bit ou se ainda é possível acrescentar bits do código atual ao final desse byte. Caso tal não seja possível, o código atual é simplesmente acrscentado à estrutura de dados `CharVec`, que é essencialmente um array de caracteres dinâmico.

Capítulo 5

Resultados

5.1 Ficheiros Usados

Como explicado na introdução, os ficheiros que utilizamos para testar são, com a exceção dos source files, os ficheiros gerados pelos módulos anteriores. Utilizamos os seguintes ficheiros:

- 1 ficheiro txt (*Shakespeare.txt*)
- 1 ficheiro zip (*bbb.zip*)
- 1 ficheiro de audio (
- 1 ficheiro de imagem (*java_french.jpg*)
- 1 ficheiro pdf (*enunciado-CD-v4-2.pdf*)
- 2 ficheiros de vídeo (*java-sock.mp4* e *shell-basics.mp4*)

5.2 Estatísticas de Execução

5.2.1 Módulo F

Neste módulo optamos por forçar compressão RLE em todos os ficheiros com a exceção do Shakespeare, pois este apenas é utilizado para mostrar compressão visível no módulo C e não apresenta compressão rle significativa.

Observamos que a compressão RLE é praticamente negligenciável em todos os ficheiros escolhidos com a exceção do vídeo *shell-basics.mp4*. Experimentamos para todos os ficheiros tamanhos de blocos diferentes, registando na tabela todos os que resultam numa melhor taxa de compressão RLE.

Ficheiros		Módulo F				
Nome	Tamanho (M)	Taxa de Compressão (%)	N Blocos	Tamanho dos Blocos (N)	Tamanho dos Blocos (RLE)	Texec (ms)
Shakespeare.txt	5.6	-	89	65536/9526	-	76.4
bbb.zip	716	0	90	8388608/3444089	8418559/3461298	16027.4
bbb.zip	716	0.0007	1145	655360/298361	654335/300058	16293.6
bbb.zip	716	0.002	11445	65536/36217	62218/36532	19844.2
Hannah_[...].mp3	1.02	0	2	655360/386749	665542/395054	23.2
Hannah_[...].mp3	1.02	0.2	16	65536/59069	63488/59989	26.2
java_french.jpg	0.944	0	15	65536/45287	66018/45545	34
enunciado[...].pdf	0.868	0.7	14	65536/33939	66087/33320	25.7
java-sock.mp4	34	2.2	544	65536/13841	13993/7508	806.2
shell-basics.mp4	290	40.4	4626	65536/56411	101867/40071	6003.1

5.2.2 Módulo T

No módulo T testamos os ficheiros de frequência gerados anteriormente, escolhendo aqueles que demonstraram a maior taxa de compressão RLE.

Ficheiros	Módulo T		
Nome	N Blocos	Tamanho dos Blocos (N)	Texec (ms)
Shakespeare.txt.freq	89	65536/9526	6.4
bbb.zip.freq	11445	65536/36217	1990.4
Hannah_[...].mp3.freq	16	65536/59069	7.2
java_french.jpg.freq	15	65536/45287	2.9
enunciado[...].pdf.freq	14	65536/33939	6.2
java-sock.mp4.rle.freq	544	13993/7508	861.6
shell-basics.mp4.rle.freq	4626	56952/40071	803

5.2.3 Módulo C

No módulo C utilizamos os ficheiros de *source* que temos usado até agora e também os ficheiros de código registados que geramos anteriormente (os mesmos que estão registados na tabela do módulo T). Nenhum ficheiro escolhido demonstrou taxa de compressão global significante, mesmo variando o número de blocos no módulo F, com a exceção do ficheiro *Shakespeare.txt*.

Ficheiros	Módulo C		
Nome	N Blocos	Taxa de Compressão Global (%)	Texec (ms)
Shakespeare.txt	89	40	72.2
bbb.zip	11445	0	9850.9
Hannah_[...].mp3	16	0	31.1
java_french.jpg	15	0	32.6
enunciado[...].pdf	14	0	28.8
java-sock.mp4.rle	544	0	463.5
shell-basics.mp4.rle	4626	1	3181

Capítulo 6

Considerações Finais

Consideramos que o nosso grupo se conseguiu adaptar bem a este projeto e conseguiu atingir os objetivos estabelecidos para o trabalho. Todos os elementos do grupo procuraram, não só cumprir o que era exigido, como também obter o melhor desempenho e otimização do programa que conseguiram. Além disso, certificou-se que nenhum módulo possuia *memory leaks* ou outras falhas na gestão de memória. Quanto aos resultados obtidos pelo programa, estes pareceram-nos bastante satisfatórios. No entanto, o número de ficheiros que beneficia da compressão RLE ou de codificação Shannon-Fano não é substancial.

NOTA: Dado que este relatório foi entregue de forma faseada sem o módulo D, faltam ainda conclusões tiradas após a descodificação dos ficheiros, bem como as estatísticas do dito módulo . Estas serão adicionadas ao relatório mais tarde.